Git Apprentice

FIRST EDITION

Getting Started with Git Commands & Concepts

By the **raywenderlich Tutorial Team**

Chris **Belanger**

Based on material by Sam Davies

Git Apprentice

Chris Belanger

Copyright ©2020 Razeware LLC.

Notice of Rights

All rights reserved. No part of this book or corresponding materials (such as text, images, or source code) may be reproduced or distributed by any means without prior written permission of the copyright owner.

Notice of Liability

This book and all corresponding materials (such as source code) are provided on an "as is" basis, without warranty of any kind, express of implied, including but not limited to the warranties of merchantability, fitness for a particular purpose, and noninfringement. In no event shall the authors or copyright holders be liable for any claim, damages or other liability, whether in action of contract, tort or otherwise, arising from, out of or in connection with the software or the use of other dealing in the software.

Trademarks

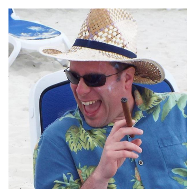
All trademarks and registered trademarks appearing in this book are the property of their own respective owners.

Dedications

"For Russ and Skip."

— Chris Belanger

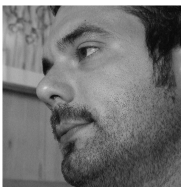
About the Author



Chris Belanger is an author of this book. He is the Editor-in-Chief of raywenderlich.com. If there are words to wrangle or a paragraph to ponder, he's on the case. In the programming world, Chris has over 25 years of experience with multiple database platforms, real-time industrial control systems, and enterprise healthcare information systems. When he kicks back, you can usually find Chris with guitar in hand, looking for the nearest beach, or exploring the lakes and rivers in his part of the world in a canoe.

About the Editors

Bhagat Singh is a tech editor for this book. Bhagat started iOS Development after the release of Swift, and has been fascinated by it ever since. He likes to work on making apps more usable by building great user experiences and interactions in his applications. He also is a contributor in the Raywenderlich tutorial team. When the laptop lid shuts down, you can find him chilling with his friends and finding new places to eat. He dedicates all his success to his mother. You can find Bhagat on Twitter: @soulful_swift



Cesare Rocchi is a tech editor of this book. Cesare runs Studio Magnolia, an interactive studio that creates compelling web and mobile applications. He blogs at upbeat.it, and he's also building Podrover and Affiliator You can find him on Twitter at @_funkyboy.

Manda Frederick is an editor of this book. She has been involved in publishing for over ten years through various creative, educational, medical and technical print and digital publications, and is thrilled to bring her experience to the raywenderlich.com family as Managing Editor. In her free time, you can find her at the climbing gym, backpacking in the backcountry, working on poems, playing guitar and exploring breweries.

Sandra Grauschopf is an editor of this book. Sandra has over 20 years' experience as a writer, editor, copy editor, and content manager and has been editing tutorials at raywenderlich.com since 2018. She loves to travel and explore new places, always with a trusty book close at hand.

Aaron Douglas is the final pass editor for this book. He was that kid taking apart the mechanical and electrical appliances at five years of age to see how they worked. He never grew out of that core interest - to know how things work. He took an early interest in computer programming, figuring out how to get past security to be able to play games on his dad's computer. He's still that feisty nerd, but at least now he gets paid to do it. Aaron works for Automattic (WordPress.com, WooCommerce, Tumblr, SimpleNote) as a Mobile Lead primarily on the WooCommerce mobile apps. Find Aaron on Twitter as @astralbodies or at his blog at https://aaron.blog.

About the Artist

Vicki Wenderlich is the designer and artist of the cover of this book. She is Ray's wife and business partner. She is a digital artist who creates illustrations, game art and a lot of other art or design work for the tutorials and books on raywenderlich.com. When she's not making art, she loves hiking, a good glass of wine and attempting to create the perfect cheese plate.

Table of Contents: Overview

Book License ... 11

Before You Begin ... 12

What You Need ... 13

Book Source Code & Forums ... 14

About the Cover ... 15

Introduction ... 17

Section I: Beginning Git ... 20

Chapter 1: A Crash Course in Git ... 21

Chapter 2: Cloning a Repo ... 34

Chapter 3: Committing Your Changes ... 43

Chapter 4: The Staging Area ... 63

Chapter 5: Ignoring Files in Git ... 76

Chapter 6: Git Log & History ... 84

Chapter 7: Branching ... 99

Chapter 8: Merging ... 110

Chapter 9: Syncing With a Remote ... 124

Chapter 10: Creating a Repository ... 139

Conclusion ... 153

Section II: Appendices ... 154

Appendix A: Installing & Configuring Git ... 155

Table of Contents: Extended

Book License 11

Before You Begin 12

What You Need 13

Book Source Code & Forums 14

About the Cover 15

Introduction 17

Enter the video courses 18

How to read this book 18

Section I: Beginning Git 20

Chapter 1: A Crash Course in Git 21

What are remote repositories? 22

Forking the remote repository 22

Cloning the repository 25

Creating a branch 26

Making and staging changes 28

Committing changes 29

Pushing your changes 30

Creating a pull request 31

Chapter 2: Cloning a Repo 34

What is cloning? 35

Using GitHub 35

Forking 40

Challenge 41

Key points 42

Where to go from here? 42

Chapter 3: Committing Your Changes 43
What is a commit? 44
Working trees and staging areas 47
Committing your changes 53
Adding directories 54
Looking at git log 59
Challenge 61
Key points 62
Where to go from here? 62
Chapter 4: The Staging Area 63
Why staging exists 64
Undoing staged changes 66
Moving files in Git 69
Deleting files in Git 72
Challenge 74
Key points 75
Where to go from here? 75
Chapter 5: Ignoring Files in Git 76
Introducing .gitignore 77
Getting started 77
Nesting .gitignore files 79
Looking at the global .gitignore 81
Finding sample .gitignore files 82
Challenge 82
Key points 83
Where to go from here? 83
Chapter 6: Git Log & History 84
Viewing Git history 85
Vanilla git log 85
Limiting results 86

Graphical views of your repository 88
Viewing non-ancestral history 90
Using Git shortlog.... 90
Searching Git history 92
Challenges 95
Key points.... 97
Where to go from here?.... 98

Chapter 7: Branching 99
What is a commit? 100
What is a branch?.... 101
Creating a branch 102
How Git tracks branches 102
Checking your current branch.... 103
Switching to another branch 103
Viewing local and remote branches 105
Explaining origin.... 105
Viewing branches graphically 106
A shortcut for branch creation 107
Challenge.... 108
Key points 109
Where to go from here? 109

Chapter 8: Merging.... 110
A look at your branches 111
Three-way merges.... 113
Merging a branch.... 115
Fast-forward merge 118
Forcing merge commits 120
Challenge.... 122
Key points 123
Where to go from here? 123

Chapter 9: Syncing With a Remote 124
Pushing your changes 125
Pulling changes 127
Dealing with multiple remotes 133
Key points 138
Where to go from here? 138
Chapter 10: Creating a Repository 139
Getting started 140
Creating a LICENSE file 141
Creating a README file 143
Creating and syncing a remote 146
Key points 151
Where to go from here? 152
Conclusion 153
Section II: Appendices 154
Appendix A: Installing & Configuring Git 155
Installing on Windows 155
Installing on macOS 158
Configuring credentials 160
Setting your username and email 160
Persisting your password 160

Book License

By purchasing *Git Apprentice*, you have the following license:

- You are allowed to use and/or modify the source code in *Git Apprentice* in as many apps as you want, with no attribution required.
- You are allowed to use and/or modify all art, images and designs that are included in *Git Apprentice* in as many apps as you want, but must include this attribution line somewhere inside your app: "Artwork/images/designs: from *Git Apprentice*, available at www.raywenderlich.com".
- The source code included in *Git Apprentice* is for your personal use only. You are NOT allowed to distribute or sell the source code in *Git Apprentice* without prior authorization.
- This book is for your personal use only. You are NOT allowed to sell this book without prior authorization, or distribute it to friends, coworkers or students; they would need to purchase their own copies.

All materials provided with this book are provided on an "as is" basis, without warranty of any kind, express or implied, including but not limited to the warranties of merchantability, fitness for a particular purpose and noninfringement. In no event shall the authors or copyright holders be liable for any claim, damages or other liability, whether in an action of contract, tort or otherwise, arising from, out of or in connection with the software or the use or other dealings in the software.

All trademarks and registered trademarks appearing in this guide are the properties of their respective owners.

Before You Begin

This section tells you a few things you need to know before you get started, such as what you'll need for hardware and software, where to find the project files for this book, and more.

What You Need

To follow along with this book, you'll need the following:

- **Git 2.28 or later.** Git is the software package you'll use for all the work in this book. There are installers for macOS, Windows and Linux available for free from the official Git page here: https://git-scm.com/downloads. We've tested this book on Git 2.28.0, but you can follow along with older versions of Git as well.

Book Source Code & Forums

Book source code

The materials for this book are all available in the GitHub repository here:

- https://github.com/raywenderlich/gita-materials/tree/editions/1.0

You can download the entire set of materials for the book from that page.

Forum

We've also set up an official forum for the book at https://forums.raywenderlich.com/c/books/git-apprentice. This is a great place to ask questions about the book or to submit any errors you may find.

About the Cover

Git Apprentice

While not the most elegant or agile creature, the flightless penguin should not be underestimated. Very few other animals can boast the wide adaptability of these birds. Found in both global hemispheres, penguins are both animals of the land and the sea, spending half of their lives on each.

In water, they are independent, graceful swimmers and formidable hunters, feeding on fish, squid and other sea life as they swim and dive — sometimes up to depths of over 500 meters for up to 22 minutes at a time. On land — well, we know about penguins on land. Their colonies are a comical flurry of waddling, rock-hopping and belly sliding — but they are also social, gentle and maternal.

Like penguins, Git thrives in multiple environments and is incredibly adaptable, and its utility should not be underestimated. Though Git seems unassuming at first glance, not many other tools will allow you to leverage your work in so many environments, both independently and socially. And like these resilient birds who manage to slip and tumble, getting back up each time, Git will allow you to work knowing any mistake can be corrected. The key is just to keep waddling along.

It should also be noted that both penguins and the authors of this book look great dressed in tuxedos.

Introduction

There are usually two reasons a person picks up a book about Git: one, they are unusually curious about how the software works at a deeper level; or two, they're frustrated and need something to solve their problems *now*.

Whatever situation brought you here, welcome! I'm happy to have you onboard. I came to write this book for *both* of the above reasons. I am a tinkerer and hacker by nature, and I love going deep into the internals of software to see what makes them tick. But I, like you, found Git at first to be an inscrutable piece of software. My brain, which had been trained in software development through the late 1990s, found version control packages like SVN soothing, with their familiar client-server architecture, Windows shell integration, and rather straightforward, albeit heavy, processes.

When I came to use Git and GitHub about seven years ago, I found it *inscrutable* at best; it seemed no matter which way I turned, Git was telling me I had a merge conflict, or it was merging changes from the master branch into my current branch, or quite often complaining about unstaged changes. And why was it called a "pull request", when clearly I was trying to *push* my changes into the master branch?

Little by little, I learned more about how Git worked; how to solve some of the common issues I encountered, and I eventually got to a point where I felt comfortable using it on a daily basis.

Enter the video courses

In early 2017, my colleague Sam Davies created a conference talk, titled "Mastering Git", and from that, two video courses at raywenderlich.com: "Beginning Git" and "Mastering Git". Those two courses form the basis of this book, but it always nagged me a little that, while Sam's video version of the material was quite pragmatic and tied nicely into using both the command line and graphical tools to solve common Git workflow problems, I always felt like there was a bit of detail missing; the kind of information that would lead a curious mind to say "I see the *how*, but I really want to know more about the *why*."

This book gives a little more background on the *why*: or, in other words, "*Why* the %^&$ did you do that to my repository, Git?!" Underneath the hood, you'll find that Git has a rather simple and elegant architecture, which is why it scales *so* well to the kinds of globally distributed projects that use Git as their version control software, via GitHub, GitLab, Bitbucket, or other cloud repository management solutions.

And while GUI-based Git frontends like Tower or GitHub Desktop are great at minimizing effort, they abstract you away from the actual guts of Git. That's why this book takes a command-line-first approach, so that you'll gain a better understanding of the various actions that Git takes to manage your repositories — and more importantly, you'll gain a better understanding of how to fix things when Git does things that don't seem to make much sense.

How to read this book

This book covers **Beginning Git**. If you are still struggling to figure out the difference between a push and a commit, or you're coming to Git from a different version control system, start here. This section takes you through concepts such as cloning, staging, committing, syncing, merging, viewing logs, and more. The very first chapter is a crash course on using Git, where you'll go through the basic Git workflow to get a handle on the *how* before you move into the *what* and the *why*.

This book works with a small repository that houses a simple ToDo system based on text files that hold ideas (both good and bad) ideas for content for the website. It's an ideal way to learn about Git without getting bogged down in a particular language or framework.

The next book in our mastering Git series, *Advanced Git*, which we encourage you to explore once you've completed this book, covers:

Mastering Git

If you've been using Git for a while, you may choose to start in this section first. If you know how to do basic staging, committing, merging and .gitignore operations, then you'll likely be able to jump right in here. This section walks you through concepts such as merge conflicts, stashes, rebasing, rewriting history, fixing .gitignore after the fact, and more.

If you've ever come up against a scenario where you feel you just need to delete your local repository and clone things fresh, then this section is just what you need to help you solve those sticky Git situations.

Workflows

This section takes a look at some common Git workflows, such as the feature branch workflow, Gitflow, a basic forking workflow, and even a centralized workflow. Because of the flexibility of Git, lots of teams have devised interesting workflows for their teams that work for them — but this doesn't mean that there's a single *right* way to manage your development.

Learning by doing

Above all, the best advice I can give is to *work* with Git: find ways to use it in your daily workflows, find ways to contribute to open-source projects that use Git to manage their repositories, and don't be afraid to try some of the more esoteric Git commands to accomplish something. There's little chance you'll screw anything up beyond repair, and most developers learn best when they inadvertently back themselves into a technical rabbit-hole — then figure out how to dig themselves out.

A note on master vs. main

At the time that this book went to press, GitHub (and potentially other hosts) were proposing changing the name of the default repository branch to `main`, instead of `master`, in an attempt to use more culturally-aware language. So if you're working through this book and realize that some repos use `main` as the central reference branch, don't worry — simply use `main` in place of `master` where you need to in these commands. If the point comes when there seems to be a consensus on `main` vs `master` in the Git community, we'll modify the book to match.

I wish you all the best in your Git adventures. Time to *Git* going! — Chris Belanger

Section I: Beginning Git

This section is intended to get newcomers familiar with Git. It will introduce the basic concepts that are central to Git, how Git differs from other version control systems, and the basic operations of Git like committing, merging, and pulling.

You may discover things in this section you didn't quite understand about Git, even if you've used Git for a long time.

Chapter 1: A Crash Course in Git

It can be a bit challenging to get started with command-line Git if you haven't done much work on the command line before. Since you'll be interacting with Git through the command line throughout this book, this chapter will take you through a quick crash course on how to do it.

There's a common workflow that serves as the foundation of most interactions you'll have with Git:

- Create a fork of an existing repository.
- Copy a remote repository to your own computer.
- Create a separate working area in the repository where you can make changes without affecting anyone else.
- Flag those changes to be saved to the local copy of the repository.
- Save those changes in your local copy of the repository.
- Synchronize those changes with the remote repository.
- Optionally, notify the repository owner that your changes are ready to be reviewed.

This chapter will take you through all the above actions to help you get familiar with the basics of working with Git through the command line.

Although this chapter won't explain everything in detail, it will give you enough familiarity with a Git repository and the basic Git workflow to better understand the chapters to come.

What are remote repositories?

A remote repository is simply a collection of all the files of a project, hosted somewhere other than your local machine. They could be hosted internally on your network, but more often, you'll work with remote repositories hosted on cloud services like GitHub and GitLab.

Having a centralized remote repository makes sharing and contributing to a project easy. Instead of sending files to interested people, you simply point them to the hosted remote repository to get them up and running as quickly as possible.

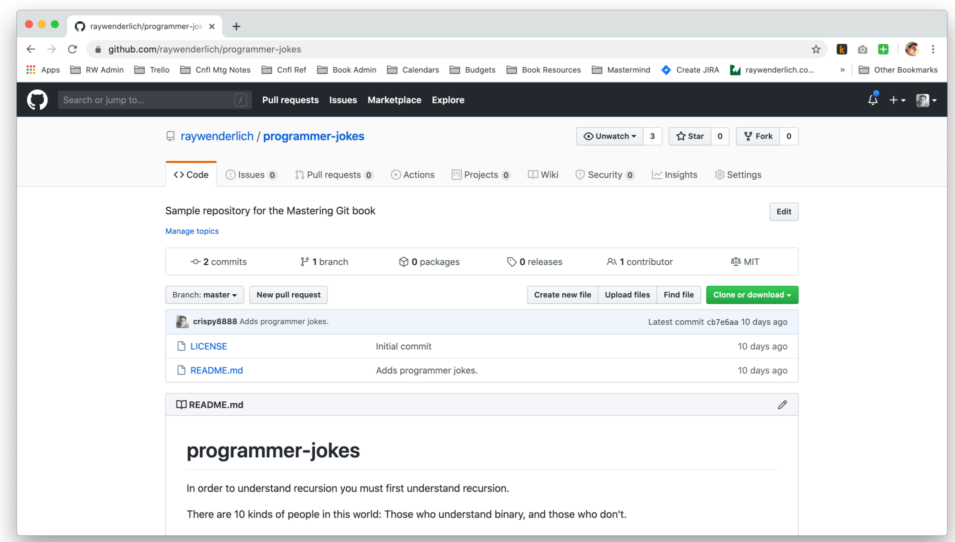
The first step is to create your own personal online copy, or **fork**, of the remote repository. That gives you a place to work online and lets you follow the instructions in this chapter without affecting any of the millions of other people reading this book and following along themselves.

Forking the remote repository

Navigate to the following URL in a browser:

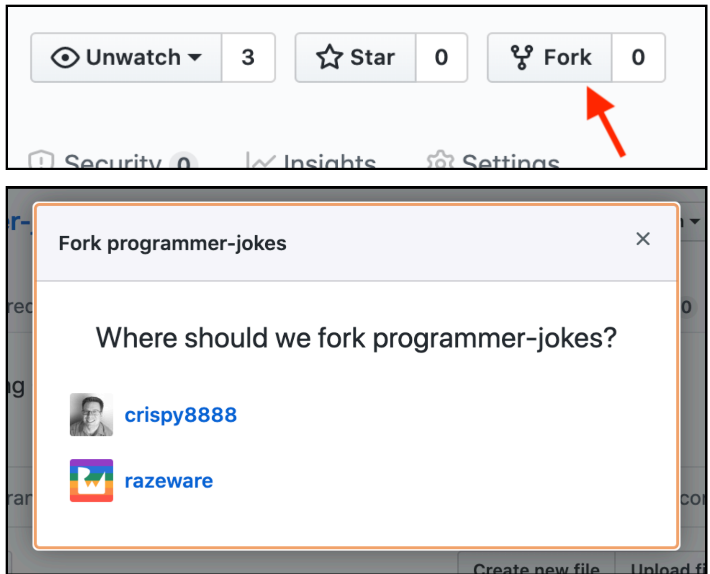
- https://github.com/raywenderlich/programmer-jokes

You'll see a screen like the following:



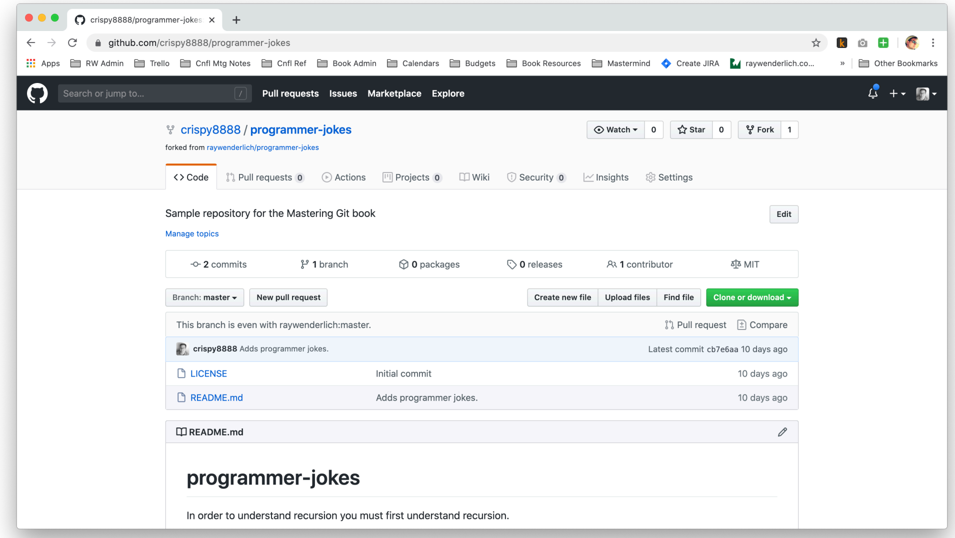
This is the main GitHub page for the project you'll use in this book. You'll cover all the details about GitHub later.

Ensure you're logged in with your own GitHub username, then click the **Fork** button in the top right-hand corner of the page:



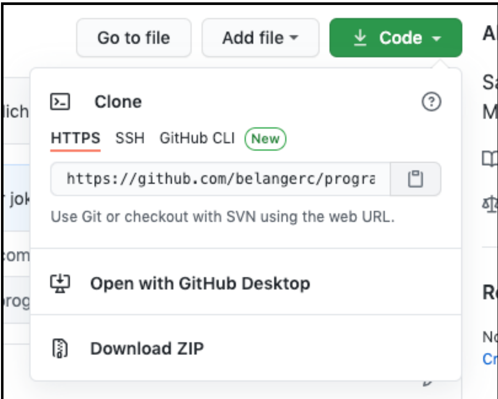
> **Note**: If you belong to more than one organization on GitHub, you'll likely see a dialog similar to the one below, asking you where to fork the **programmer-jokes** repository:
>
> In this case, GitHub isn't really asking you *where* to fork to physically; it's asking under which account you want to create the fork. Choose your own username.

You'll see a progress screen while GitHub creates your repository fork under your account. When GitHub's done creating that fork, you'll see another screen that looks a lot like the original page, except that now you're working from a different location:



And that's just what you want. This is an exact replica of the original programmer-jokes repository, except this copy lives under your own account. That means you can do anything you like to this repository, even delete it, without affecting the original repository that lives under the `raywenderlich` organization.

To get started, you'll need to copy, or **clone**, this remote repository to your local workstation. To do that, you'll need the remote URL of this repository. It's easy to get — simply click the **Code** button on the page, then click the small clipboard icon next to the `https://github.com/username...` URL in the dialog:



You now have the remote URL of this repository in your clipboard.

You're done with this webpage for a bit — you're now ready to start working with Git on the command line.

Open Terminal, PowerShell or the appropriate console prompt on your system and get ready to follow along.

Cloning the repository

At the command prompt, type the following command *without* pressing the **Enter** key:

```
git clone
```

After that, press the spacebar to insert a space character. Then paste what's in your clipboard to the command line using **Command-V** or **Control-V**, depending on what the **Paste** command is on your operating system.

You should have something similar to the following in your command prompt:

```
git clone https://github.com/<your-username>/programmer-
jokes.git
```

Now, to break that down a little:

- `git` is the name of the command-line Git tool. Every interaction you have with Git on the command line will start with `git` and be followed by the Git command you want to execute.
- `clone` is the name of the command you want to execute. `clone` tells Git to copy a specific named repository to your local machine.
- `https://github.com/<your-username>/programmer-jokes` is the full URL to the repository you want to clone. Breaking that down further, `https://github.com` is the cloud service that hosts the repository, `<your-username>` is the owner of the fork of this repository, and `programmer-jokes` is the name of the repository you want to clone.

Press **Enter** or **Return** to execute that command. Git gives you a bit of output on the command line to tell you what it's done:

```
Cloning into 'programmer-jokes'...
remote: Enumerating objects: 7, done.
remote: Total 7 (delta 0), reused 0 (delta 0), pack-reused 7
Unpacking objects: 100% (7/7), 2.13 KiB | 311.00 KiB/s, done.
```

The details of that output aren't important, but do take a look at that first line:

```
Cloning into 'programmer-jokes'...
```

Git's telling you that it's cloning the remote repository into a new directory it's created: **programmer-jokes**.

Navigate into that directory from the command line with the following command:

```
cd programmer-jokes
```

Next, execute the following to get a list of the files in that directory in long format — just because it's easier to read:

```
ls -l
```

You'll see output similar to the following:

```
-rw-r--r--  1 chrisbelanger  staff  1070 29 May 11:25 LICENSE
-rw-r--r--  1 chrisbelanger  staff   370 29 May 11:25 README.md
```

There are two files in this repository: **LICENSE**, which has some boring legal information about the contents of the repository, and **README.MD**, which is a simple text file that contains some groan-worthy programming jokes.

Now that you have the repository cloned to your local machine, the next step is to create a separate working space, or **branch**, where you can change the contents of **README.md** without fear of messing up the original contents of the repository.

Creating a branch

Branches are, conceptually, *copies* of the original contents of the repository. You can work in a branch without affecting the original contents of the repository until you are ready to merge all your work back together again.

If you've ever made a copy of an important document before you started editing it, the concept of branching is exactly the same.

At the command line, execute the following to create a new branch:

```
git branch my-joke
```

Breaking this down:

- **git**, again, is the name of the command-line tool.
- **branch** is the name of the command you want Git to execute.
- **my-joke** is the name of the branch you want to create. The name you give to a branch isn't important, but you generally want to give it a descriptive name, just as you would when creating new folders on your desktop.

You can see that Git's created a new branch by executing the following command:

```
git branch
```

This looks similar to the command above, but in this case, you haven't supplied a branch name. Git understands this to mean, "Oh, you don't want to create a branch, you just want to look at all the branches I know about."

Git responds with the following output:

```
* master
  my-joke
```

`master` is the original copy of the repository, while `my-joke` is the branch you just created. The asterisk `*` indicates which branch you're currently working in. Right now, you're still on `master`, but that's not what you want — you want to change to `my-joke` so you don't affect `master`.

To switch to the `my-joke` branch, execute the following:

```
git checkout my-joke
```

Ah, a new command: `checkout`. You might have expected a command like `switch-branch`, but Git thinks of switching branches in terms of "checking out". It's similar to how you check out a book at a library: That copy of the book is now exclusively yours to work with until you return it to the library.

Git responds to the `checkout` command with the following:

```
Switched to branch 'my-joke'
```

If you're the paranoid type, like me, you can confirm this with the command below:

```
git branch
```

Git puts your fears to rest with the following response:

```
  master
* my-joke
```

The asterisk tells you that you're now working securely inside the `my-joke` branch, and that your changes won't affect the master copy of the repository.

Now, it's time for you to add a stunning joke to **README.md**.

Making and staging changes

README.md is simply a text file. Open it in a text editor of your choice and you'll see it has the following contents:

```
# programmer-jokes

In order to understand recursion you must first understand
recursion.

There are 10 kinds of people in this world: Those who understand
binary, and those who don't.

An SEO expert walked into a bar, pub, liquor store, brewery,
alcohol, beer, whiskey, vodka...

Why did the two functions stop calling each other? Because they
had constant arguments.
```

They're *hilarious*, right!? Well, you can definitely improve upon them by adding your own joke to this list.

Just in case you don't have a handy stash of programmer jokes at your disposal, add the following line to the text file:

```
Why couldn't the confirmed bachelor use Git? Because he was
afraid to commit!
```

Save your changes and exit the text editor.

Git's pretty smart, but it also knows not to assume too much. Just because you've modified a file, Git isn't going to assume that you want this in the repository. Instead, it will wait for you to flag those changes to be made to the repository.

Execute the command below to flag the change you made to the file, remembering that case is important:

```
git add README.md
```

The `add` command tells Git to add, or **stage**, the changes you've made to **README.md** to the list of things to add to the repository. In this case, you only have one change to one file, but in practice, you'll usually have lots of changes to lots of files.

Now that Git is aware that you want to make that change, you'll need to **commit** it to your local repository.

Committing changes

Git's got your back here. It knows that sometimes you might add a change, but then have second thoughts, or you might have other changes that you want to stage at the same time. That's why Git separates the act of staging files from the act of committing those changes.

Committing is the act of saying, "Yes, I have these changes ready, and I want to formally record those changes in my local copy of the repository."

Not only does Git record those changes formally, it also allows you to provide a **commit message** to give some context about the content of those changes to others — or even to your future self.

Commit your changes now with the following command:

```
git commit -m "Adding my new joke"
```

That tells Git to formally record your current set of changes — although in this case, you only have one change, which is the joke you added.

Git responds with output similar to the following:

```
[my-joke f8f8854] Adding my new joke
 1 file changed, 1 insertion(+)
```

That may look confusing, but there's a lot of information in there:

- **my-joke** is the branch you're committing your changes to.
- **f8f8854** is the unique identifier for your commit, also known as the **commit hash** of your changes. You use this identifier to uniquely reference this specific commit in the future. Note, if you're following along and typing commands as you read, your hash will be different.
- **Adding my new joke** is the commit message you added above.
- **1 file changed, 1 insertion(+)** provides some context about what was changed in this commit: one file, with one line added.

You've formally recorded these changes in your local repository, but now you need to get them synchronized, or **pushed**, back to the remote repository.

Pushing your changes

Most of Git's actions are performed from the perspective of your local workstation. So the action you're taking is to **push** your local commit to the remote server.

Execute the following command to send your local changes to your forked remote repository:

```
git push --set-upstream origin my-joke
```

That's a little obtuse, for sure. Don't be too concerned at this point; you'll cover what this means in future chapters. Essentially, here's what the various pieces mean:

- **push** tells Git to put your local changes on the server.
- **–set-upstream** tells Git to form a tracking link for this branch between your local repository and the remote repository.
- **origin** is a convention that references the remote repository.
- **my-joke** is the branch you want to push.

Git responds with a pile of output:

```
Enumerating objects: 5, done.
Counting objects: 100% (5/5), done.
Delta compression using up to 4 threads
Compressing objects: 100% (3/3), done.
```

```
Writing objects: 100% (3/3), 396 bytes | 396.00 KiB/s, done.
Total 3 (delta 1), reused 0 (delta 0)
remote: Resolving deltas: 100% (1/1), completed with 1 local
object.
remote:
remote: Create a pull request for 'my-joke' on GitHub by
visiting:
remote:      https://github.com/<your-username>/programmer-
jokes/pull/new/my-joke
remote:
To https://github.com/<your-username>/programmer-jokes.git
 * [new branch]      my-joke -> my-joke
Branch 'my-joke' set up to track remote branch 'my-joke' from
'origin'.
```

Note: You may be prompted by the command line to enter your GitHub username and password. If so, then provide them and hit Enter or Return after you do.

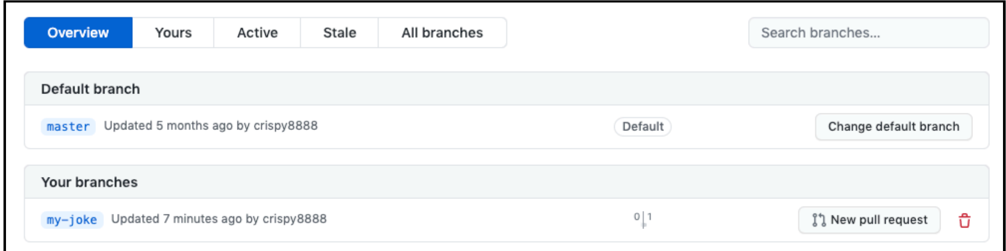
Git's successfully pushed your changes to the remote repository, but there's one thing left to do: Signal to anyone else using this repository that you have something you'd like to integrate, or **pull**, into the remote repository. You do that with a mechanism called a **pull request**.

Creating a pull request

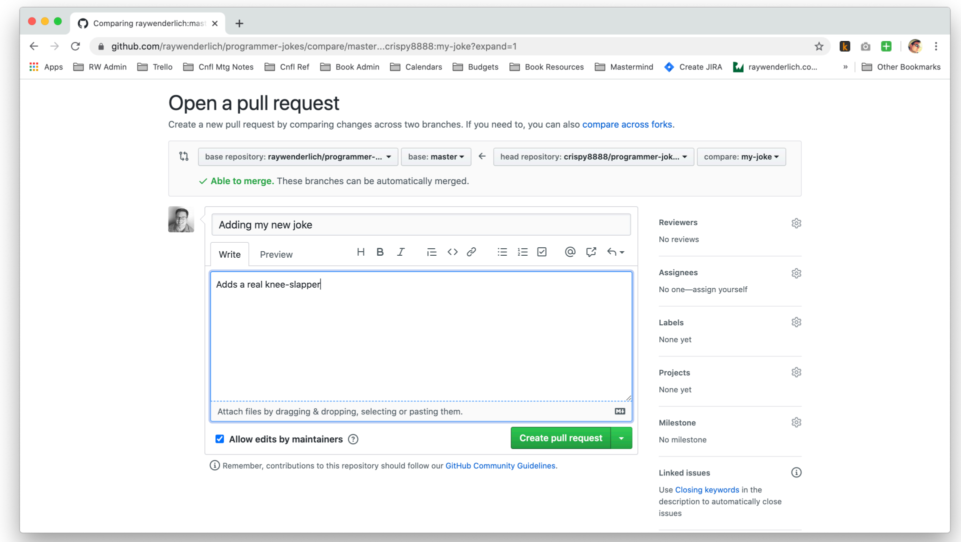
Return to your browser, and if you don't already have it open, open the main GitHub page for your forked repository at `https://github.com/<your-username>/programmer-jokes`.

If you look *really* closely, You'll notice that the page has changed a little, to reflect the fact that your changes made it successfully to the remote repository:

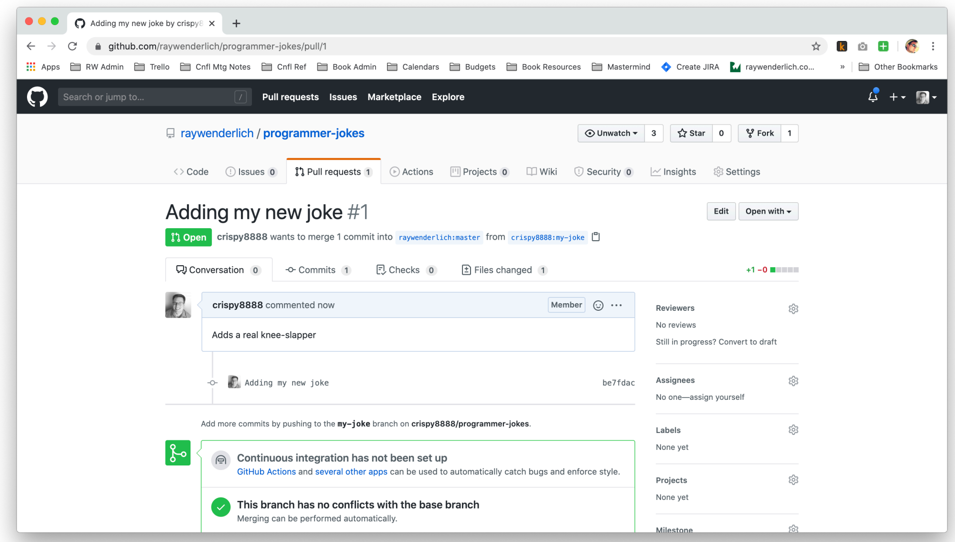
You can see GitHub is telling you that there's now two branches in this repo. Click on that "2 branches" link, and you'll see the following page, showing your new branch, along with a "New pull request" button:



Click that button and you'll be taken to another page, where you can enter some details about your change. Enter **Adds a real knee-slapper** to the large text box to give some extra detail about the changes contained in this pull request, then click the **Create pull request** button:



GitHub takes you to yet another page, where you can see that your pull request is now active, along with various information related to your pull request:



That's as far as you need to go with this short crash course in Git. If you didn't understand quite everything that happened along the way, don't worry! This chapter was just to get you familiar with the basic fork → clone → branch → add → commit → push → pull request mechanism of Git.

The rest of the book will look at each of these steps in detail, along with much, much more information about the more obscure commands in Git. You'll also get a tour of the internals of Git, which may help you understand a little better why Git does what it does.

Head on to the next chapter, where you'll start by looking at the `clone` command in more depth. See you there!

Chapter 2: Cloning a Repo

The preceding chapter took you through a basic crash course in Git, and got you right into using the basic mechanisms of Git: cloning a repo, creating branches, switching to branches, committing your changes, pushing those changes back to the remote and opening a pull request on GitHub for your changes to be reviewed.

That explains the *how* aspect of Git, but, if you've worked with Git for any length of time (or haven't worked with Git for any time at all), you'll know that the *how* is not enough. It's important to also understand the *why* of Git to gain not just a better understanding of what's going on under the hood, but also to understand how to fix things when, not if, your repository gets into a weird state.

So, first, you'll start with the most basic aspect of Git: getting a repository copied to your local system via **cloning**.

What is cloning?

Cloning is exactly what it sounds like: creating a copy, or clone, of a repository. A Git repository is nothing terribly special; it's simply a directory, containing code, text or other assets, that tracks its own history. Then there's a bit of secure file transfer magic in front of that directory that lets you sync up changes. That's it.

A Git repository tracks the history of all changes inside the repository through a hidden **.git** directory that you usually don't ever have to bother with — it's just there to quietly track everything that happens inside the repository. You'll learn more about the structure and function of the hidden .git directory later on in this book.

So since a Git repository is just a special directory, you could, in theory, effect a pretty cheap and dirty clone operation by zipping up all the files in a repository on your friend's or colleague's workstation and then emailing it to yourself. When you extract the contents of that zipped-up file, you'd have an exact copy of the repository on your computer.

However, emailing things around can (and does) get messy. Instead, many organizations make use of online repository hosts, such as GitHub, GitLab, BitBucket or others. Some organizations even choose to self-host repositories, but that's outside the scope of this book. For now, you'll stick to using online hosts — in this example, GitHub.

Using GitHub

GitHub, at its most basic level, is really just a big cloud-based storage solution for repositories, with account and access management mixed in with some collaboration tools. But you don't need to know about all the features of GitHub to start working with repositories hosted on GitHub, as demonstrated in the Git crash course in the previous chapter.

Cloning from an online repository is a rather straightforward operation. To get started, you simply need the following things:

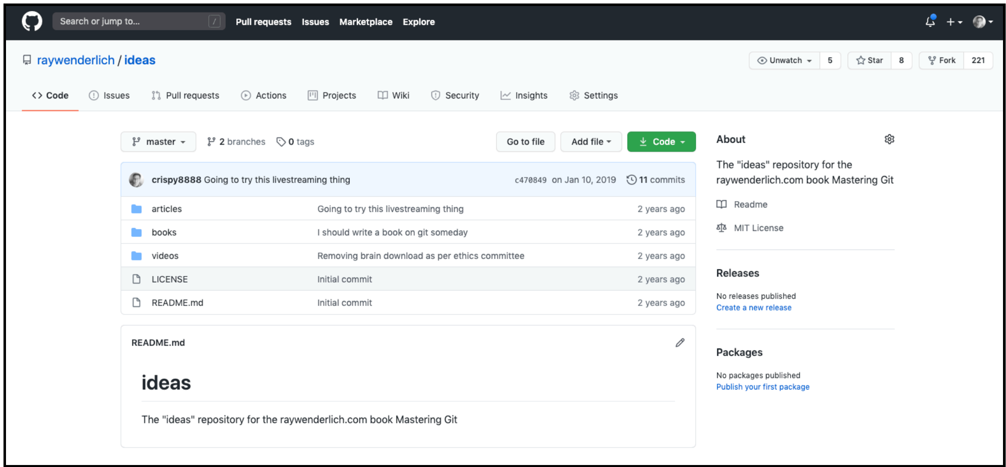
- A working installation of Git on your local system.
- The remote URL of the repository you want to clone.
- Any credentials for the online host.

> **Note**: It *is* generally possible to clone repositories without using credentials, but you won't be able to propagate the changes you make on your local copy back to the online host.

The GitHub repository homepage

There's a repository already set up on GitHub for you to clone, so you first need to get the remote URL of the repository.

To start, navigate to https://github.com/raywenderlich/ideas and log in with your GitHub username and password. If you haven't already set up an account, you can do so now.



The main page for the ideas repository.

Once you're on the homepage for the repository, have a look at the list of files and directories listed on the page. These lists and directories represent the contents of the repository, and they are the files that you'll clone to your local system.

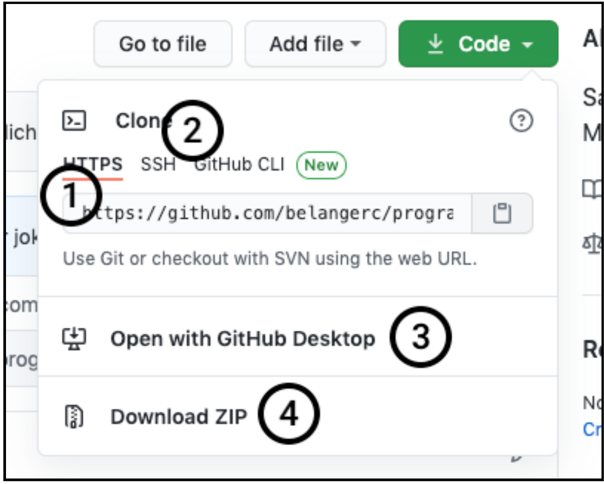
But where do you find the remote URL of the repository to clone it? Like many things in Git (and with computers, in general), there are multiple ways to clone a repository. In this chapter, you'll use the easiest and most common cloning method, which starts on the GitHub repository homepage.

Finding the repository clone URL

Look for and click on the **Code** button on the repository homepage.

The 'Clone or download' button displays the various cloning options for a repository.

The little pop-up dialog gives you a few options to get a repository cloned to your local system:



The cloning options for at GitHub repository.

1. This is the main HTTPS URL for the repository. This is the URL that you'll use in this chapter to clone from the command line `git` client.

2. You can also use SSH to clone a repository. Clicking this link lets you toggle between using SSH and HTTPS to work with the repository.

3. If you have the GitHub Desktop app installed, you can use the **Open in Desktop** button to launch GitHub Desktop and clone this repository all in one step.

4. If you just want a zipped copy of what's in the repository (but not all the repository bits itself), the **Download ZIP** button will let you do this.

For now, copy the HTTPS URL that you see in the dialog via the little clipboard icon button to the right of the URL. This places a copy of the HTTPS URL in your clipboard so that you can paste it into your command line later.

Cloning on the command line

Now, go to your command prompt. Change to a suitable directory where you want your repositories to live. In this case, I'll create a directory in my home directory named **GitApprentice** where I would like to locally store all of the repos for this book.

Execute the following command to create the new directory:

```
mkdir GitApprentice
```

Now, execute the following command to see the listing of files in the directory (yours will be different than shown below):

```
ls
```

I see the following directories on my system, and there's my new **GitApprentice** directory:

```
~ $ ls
Applications          Downloads       Music
Dropbox               Pictures        Library
Public                Desktop         GitApprentice
Documents             Movies
```

Execute the following command to navigate into the new directory:

```
cd GitApprentice
```

You're now ready to use the command line to clone the repository.

Enter the following command, but don't press the **Enter** key or **Return** key just yet:

```
git clone
```

Now, press the **Space bar** to add one space character and paste in the URL you copied earlier, so your command looks as follows:

```
git clone https://github.com/raywenderlich/ideas.git
```

Now, you can press **Enter** to execute the command.

You'll see a brief summary of what Git is doing below:

```
~/MasteringGit $ git clone https://github.com/raywenderlich/
```

```
ideas.git
Cloning into 'ideas'...
remote: Enumerating objects: 49, done.
remote: Total 49 (delta 0), reused 0 (delta 0), pack-reused 49
Unpacking objects: 100% (49/49), done.
```

Execute the `ls` command to see the new contents of your **GitApprentice** directory:

```
~/MasteringGit $ ls
ideas
```

Use the `cd` command, followed by the `ls` command, to navigate into the new **ideas** directory and see what's inside:

```
~/MasteringGit $ cd ideas
~/MasteringGit/ideas $ ls
LICENSE      README.md    articles     books        videos
```

So there's the content from the repository. Well, the *visible* content at least. Run the `ls` command again with the `-a` option to show the hidden .git directory discussed earlier:

```
~/MasteringGit/ideas $ ls -a
.        .git         README.md    books
..       LICENSE      articles     videos
```

Aha — there's that magical **.git** hidden directory. Take a look at what's inside.

Exploring the .git directory

Use the `cd` command to navigate into the .git directory:

```
cd .git
```

Execute the `ls` command again to see what dark magic lives inside this directory. This time, use the `-F` option so that you can tell which entities are files and which are directories:

```
ls -F
```

You'll see the following:

```
~/GitApprentice/ideas/.git $ ls -F
HEAD         config       hooks/       info/        objects/
refs/
branches/    description  index        logs/        packed-refs
```

So it's not quite the dark arts, I'll admit. But what *is* here is a collection of important files and directories that track and control all aspects of your local Git repository. Most of this probably won't make much sense to you at this point, and that's fine. As you progress through this book, you'll learn what most of these bits and pieces do.

For now though, leave everything as-is; there's seldom any reason to work at this level of the repository. Pretty much everything you do should happen up in your working directory, not in the .git subfolder.

So backtrack up one level to the the working directory for your repository with the `cd` command:

```
cd ..
```

You're now back up in the relative safety of the top level of your repository. For now, it's enough to know where that .git directory lives and that you really don't have a reason to deal with anything in there right now.

Forking

You've managed to make a clone of the **ideas** repository, but although **ideas** is a public repository, the **ideas** repository currently belongs to the raywenderlich organization. And since you're not a member of the raywenderlich organization, the access control settings of the **ideas** repository mean that you won't be able to push any local changes you make back to the server. Bummer.

But with most public repositories, like **ideas**, you can create a remote copy of the repository up on the server under your own personal user space. You, or anyone you grant access to, can then clone that copy locally, make changes and push those changes back to the remote copy on the server. Creating a remote clone of a repository is known as **forking**.

First, you'll need to rid your machine of the existing local clone of the **ideas** repository. It's of little use to you in its current state, so it's fine to get rid of it.

First, head up one level, out of your working directory, by executing the following command:

```
cd ..
```

You should be back up at the main **GitApprentice** directory:

```
~/GitApprentice $
```

Now, get rid of the local clone with the `rm` command, and use the `-rf` options to recursively delete all subdirectories and files, and to force all files to be deleted:

```
rm -rf ideas
```

Execute `ls` to be sure the directory is gone:

```
~/GitApprentice $ ls
~/GitApprentice $
```

Looks good. You're ready to create a fork of the raywenderlich **ideas** repository… which leads you to your challenge for this chapter!

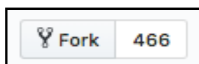
Challenge

Challenge: Fork on GitHub and create a local clone

The goal of this challenge is twofold:

1. Create a fork of the **ideas** repository under your own user account on GitHub.
2. Clone the forked repository to your local system.

Navigate to the homepage for the **ideas** repository at https://github.com/raywenderlich/ideas. In the top right-hand corner of the page, you'll see the **Fork** button. That's your starting point.



The 'Fork' button lets you create a remote copy of a repository.

The steps to this challenge are:

1. Fork the **ideas** repository under your own personal user account.
2. Find the clone URL of your new, forked repository.
3. Clone the forked **ideas** repository to your local system.

4. Verify that the local clone created successfully.

5. Bonus: Prove that you've cloned the fork of your repo and not the original repository.

If you get stuck, you can always find the solution to this challenge under this chapter's **projects/challenge** folder inside the materials you downloaded for this book.

Key points

- **Cloning** creates a local copy of a remote Git repository.
- Use `git clone` along with the clone URL of a remote repository to create a local copy of a repository.
- Cloning a repository automatically creates a hidden **.git** directory, which tracks the activity on your local repository.
- **Forking** creates a remote copy of a repository under your personal user space.

Where to go from here?

Once you've successfully completed the challenge for this chapter, head into the next chapter where you'll learn about the `status`, `diff`, `add` and `commit` commands. You'll also learn just a bit about how Git actually tracks the changes that you make in the local copy of your repository.

Chapter 3: Committing Your Changes

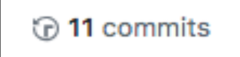
The previous chapter showed you how to clone remote repositories down to your local system. At this point, you're ready to start making changes to your repository. That's great!

But, clearly, just making the changes to your local files isn't all you need to do. You'll need to stage the changes to your files, so that Git knows about the changes. Once you're done making your changes, you'll need to tell Git that you want to commit those changes into the repository.

What is a commit?

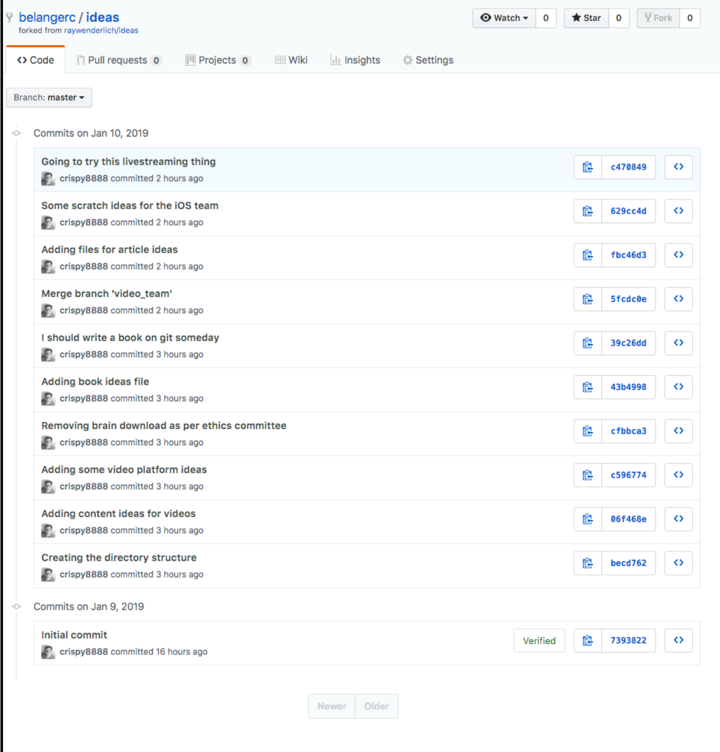
As you've probably guessed by now, a Git repo is more than a collection of files; there's quite a bit going on beneath the surface to track the various states of your changes and, even more importantly, what to do with those changes.

To start, head back to the homepage for your forked repository at https://github.com/[your-username]/ideas, and find the little "11 commits" link at the top of the repository page:



> **Note**: If you didn't complete the challenge for the last chapter, then go create a fork of https://github.com/raywenderlich/ideas and clone it to your local workstation.

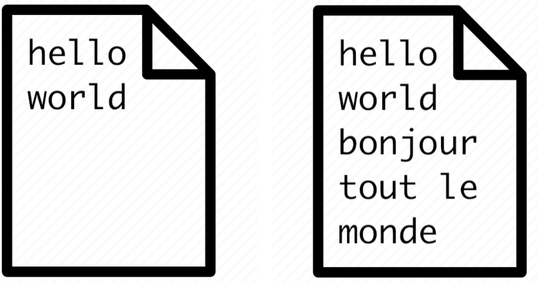
Click that link, and you'll see a bit of the history of this repository:



Each of those entries is a **commit**, which is essentially a snapshot of the particular state of the set of files in the repository at a point in time.

Generally, a commit represents some logical update to your collection of files. Imagine that you're adding new items to your ideas lists, and you've added as many as you can think of. You'd like to capture that bit of work as a commit into your repository.

The state of the repository before you began those updates — your starting point, in effect — is the **parent** commit. After you commit your changes — which is the **diff** — that next commit would be the **child** commit. The diagram below explains this a little more:



Example of two commits, the parent on the left, and the child on the right.

In this example, I've added new text to a file between commits. The parent commit is the left-hand file, and the child commit is the right-hand file. The diff between them are the changes I made to a single file:

The diff is the difference between the above two commits.

And a diff doesn't just have to be additions to files; creating new content, modifying content and deleting content are other common changes that you'll make to the files in your repository.

In Git, there are a few steps between the act of changing a file and creating a commit. This may seem like a bit of a heavy approach, at first, but, as you move through building up your commits, you'll see how each step helps create a workflow that keeps you in tune with the files in your repository and what's happened to them.

The easiest way to understand the process of building up commits is to actually create one. You'll create a change to a file, see how Git acknowledges that change, how to stage that change, and, finally, how to commit that change to the repository.

Starting with a change

Open your terminal program and navigate to the **ideas** repository; in my case, I've put it inside of the **GitApprentice** directory. This should be the clone of the forked repository that you created in the previous chapter.

> **Note**: If you missed completing the challenge at the end of the Chapter 2, go back now and follow the challenge solution so that you have a local clone of the forked **ideas** repository to work with.

Assume that you want to add more ideas to the books file. Open **books/book_ideas.md** in any plaintext editor. I like to use `nano` since it's quick and easy, and I don't need to remember any obscure commands to use it.

Add a line to the end of the file to capture a new book idea: "Care and feeding of developers." Take care to follow the same format as the other entries. Your file should look like this:

```
# Ideas for new book projects

- [ ] Hotubbing by tutorials
- [x] Advanced debugging and reverse engineering
- [ ] Animal husbandry by tutorials
- [ ] Beginning tree surgery
- [ ] CVS by tutorials
- [ ] Fortran for fun and profit
- [x] RxSwift by tutorials
- [ ] Mastering git
- [ ] Care and feeding of developers
```

When you're done, save your work and return to your terminal program.

In the background, Git is watching what you're doing. Don't believe me? Execute the following command to see that Git knows what you've done, here:

```
git status
```

`git status` shows you the current state of your working tree — that is, the collection of files in your directory that you're working on. In your case, the working tree is everything inside your **ideas** directory.

You should see the following output:

```
~/GitApprentice/ideas $ git status
On branch master
Your branch is up to date with 'origin/master'.

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

    modified:   books/book_ideas.md

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

Ah, there's the file you just changed: **books/book_ideas.md**. Git knows that you've modified it… but what does it mean when Git says, `Changes not staged for commit`?

It's time for a short diversion to look at the various states of your files in Git. Building up a mental model of the various states of Git will go a long way to understanding what Git is doing… especially when Git does something that you don't quite understand.

Working trees and staging areas

The **working copy** or **working tree** or **working directory** (language is great, there's always more than one name for something) is the collection of project files on your disk that you work with and modify directly, just as you did in **books/book_ideas.md** above.

Git thinks about the files in your working tree as being in three distinct states:

- Unmodified
- Modified
- Staged

Unmodified simply means that you haven't changed this file since your last commit. **Modified** is simply the opposite of that: Git sees that you've modified this file in some fashion since your last commit. But what's this "staged" state?

If you're coming from the background of other version control systems, such as Subversion, you may think of a "commit" as simply saving the current state of all your modifications to the repository. But Git is different, and a bit more elegant. Instead, Git lets you build your next commit incrementally as you work, by using the concept of a **staging area**.

> **Note**: If you've ever moved houses, you'll understand this paradigm. When you are packing for the move, you don't take all of your belongings and throw them loosely into the back of the moving van. (Well, maybe you do, but you *shouldn't*, really.)
>
> Instead, you take a cardboard box (the staging area), and fill it with similar things, fiddle around to get everything packed properly in the box, take out a few things that don't quite belong, and add a few more things you forgot about.
>
> When you're satisfied that the box is *just* right, you close up the box with packing tape and put the box in the back of the van. You've used the box as your staging area in this case, and taping up the box and placing on the van is like making a commit.

Essentially, as you work on bits and pieces of your project, you can mark a change, or set of changes, as "staged," which is how you tell Git, "Hey, I want these changes to go into my next commit… but I might have some more changes for you, so just hold on to these changes for a bit." You can add and remove changes from this staging area as you go about your work, and only commit that set of carefully curated changes to the repository when you're good and ready.

Notice above that I said, "Add and remove *changes* from the staging area," not "Add and remove *files* from the staging area." There's a distinct difference, here, and you'll see this difference in just a bit as you stage your first few changes.

Staging your changes

Git's pretty useful in that it (usually) tells you what do to in the output to a command. Look back at the output from `git status` above, and the `Changes not staged for commit` section gives you a few suggestions on what to do:

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

So since you want to get this change eventually committed to the repository, you'll try the first suggestion: `git add`.

Execute the following command:

```
git add books/book_ideas.md
```

Then, execute `git status` to see the results of what you've done:

```
~/GitApprentice/ideas $ 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:   books/book_ideas.md
```

Ah, that seems a little better. Git recognizes that you've now placed this change in the staging area.

But you have another modification to make to this file that you forgot about: Since you're reading this book, you should probably check off that entry for "Mastering git" in there to mark it as complete.

Open **books/book_ideas.md** in your text editor and place a lower-case **x** in the box to mark that item as complete:

```
- [x] Mastering git
```

Save your changes and exit out of your editor. Now, execute `git status` again (yes, you'll use that command often to get your bearings), and see what Git tells you:

```
~/GitApprentice/ideas $ 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:   books/book_ideas.md

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:   books/book_ideas.md
```

What gives? Git now tells you that **books/book_ideas.md** is *both* staged and not staged? How can that be?

Remember that you're staging *changes* here, not *files*. Git understands this, and tells you that you have one change already staged for commit (the `Care and feeding of developers` change), and that you have one change that's not yet been staged — marking `Mastering git` as complete.

To see this in detail, you can tell Git to show you what it sees as changed. Remember that **diff** we talked about earlier? Yep, that's your next new command.

Execute the following command:

```
git diff
```

You'll see something similar to the following:

```
diff --git a/books/book_ideas.md b/books/book_ideas.md
index 76dfa82..5086b1f 100644
--- a/books/book_ideas.md
+++ b/books/book_ideas.md
@@ -7,5 +7,5 @@
 - [ ] CVS by tutorials
 - [ ] Fortran for fun and profit
 - [x] RxSwift by tutorials
-- [ ] Mastering git
+- [x] Mastering git
 - [ ] Care and feeding of developers
```

That looks pretty obtuse, but a diff is simply a compact way of showing you what's changed between two files. In this case, Git is telling you that you're comparing two versions of the same file — the version of the file in your working directory, and the version of the file that you told Git to stage earlier with the `git add` command:

```
--- a/books/book_ideas.md
+++ b/books/book_ideas.md
```

And it also shows you what's changed between those two versions:

```
-- [ ] Mastering Git
+- [x] Mastering Git
```

The – prefix means that a line (or a portion of that line) has been deleted, and the + prefix means that a line (or a portion of that line) has been added. In this case, you deleted the space and added an **x** character.

You'll learn more about `git diff` as you go along, but that's enough to get you going for now. Time to stage your latest change.

It gets a bit tedious to always type the full name of the file you want to stage with `git add`. And, let's be honest, most of the time you really just want to stage *all* of the changes you've made. Git's got your back with a great shortcut.

Execute the following:

```
git add .
```

That full stop (or period) character tells Git to add all changes to the staging area, both in this directory and all other subdirectories. It's pretty handy, and you'll use it a lot in your workflow.

Again, execute `git status` to see what's ready in your staging area:

```
~/GitApprentice/ideas $ 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:   books/book_ideas.md
```

That looks good. There's nothing left unstaged, and you'll just see the changes to **books/book_ideas.md** that are ready to commit.

As an interesting point, execute `git diff` again to see what's changed:

```
~/GitApprentice/ideas $ git diff
~/GitApprentice/ideas $
```

Uh, that's interesting. `git diff` reports that nothing has changed. But if you think about it for a moment, that makes sense. `git diff` compares your working tree to the staging area. With `git add .`, you put everything from your working tree into the staging area, so there *should* be no differences between your working tree and staging.

If you want to be *really* thorough (or if you don't trust Git quite yet), you can ask Git to show you the differences that it's staged for commit with an extra option on the end of `git diff`.

Execute the following command, making note that it's two -- characters, not one:

```
git diff --staged
```

You'll see a diff similar to the following:

```
~/GitApprentice/ideas $ git diff --staged
diff --git a/books/book_ideas.md b/books/book_ideas.md
index 1a92ca4..5086b1f 100644
--- a/books/book_ideas.md
+++ b/books/book_ideas.md
@@ -7,4 +7,5 @@
 - [ ] CVS by tutorials
 - [ ] Fortran for fun and profit
 - [x] RxSwift by tutorials
-- [ ] Mastering git
+- [x] Mastering git
+- [ ] Care and feeding of developers
```

Here's the lines that have changed:

```
-- [ ] Mastering git
+- [x] Mastering git
+- [ ] Care and feeding of developers
```

You've removed something from the `Mastering Git` line, added something to the `Mastering Git` line, and added the `Care and feeding of developers` line. That seems to be everything. Looks like it's time to actually commit your changes to the repository.

Committing your changes

You've made all of your changes, and you're ready to commit to the repository. Simply execute the following command to make your first commit:

```
git commit
```

Git will take you into a rather confusing state. Here's what I see in my terminal program:

```
# Please enter the commit message for your changes. Lines
starting
# with '#' will be ignored, and an empty message aborts the
commit.
#
# On branch master
# Your branch is up to date with 'origin/master'.
#
# Changes to be committed:
#       modified:   books/book_ideas.md
#
~
~
~
~
"~/GitApprentice/ideas/.git/COMMIT_EDITMSG" 10L, 272C
```

If you haven't been introduced to `vim` before, welcome! **Vim** is the default text editor used by Git when it requires free text input from you.

If you read the first little bit of instruction that Git provides there, it becomes apparent what Git is asking for:

```
# Please enter the commit message for your changes. Lines
starting
# with '#' will be ignored, and an empty message aborts the
commit.
```

Ah — Git needs a message for your commit. If you think back to the list of commits you saw earlier in the chapter, you'll notice that each entry had a little message with it:

Adding content ideas for videos
crispy8888 committed 3 hours ago

Working in Vim isn't terribly intuitive, but it's not hard once you know the commands.

Press the **I** key on your keyboard to enter **Insert** mode, and you'll see the status line at the bottom of the screen change to `-- INSERT--` to indicate this. You're free to type what you like here, but stay simple and keep your message to just one line to start.

Type the following for your commit message:

```
Added new book entry and marked Git book complete
```

When you're done, you need to tell Vim to save the file and exit. Exit out of Insert mode by pressing the **Escape** key first.

Now, type a colon (**Shift** + **;** on my American keyboard) to enter **Ex** mode, which lets you execute commands.

To save your work and exit in one fell swoop, type `wq` — which means "write" and "quit" in that order, and press **Enter**:

```
:wq
```

You'll be brought back to the command line and shown the result of your commit:

```
~/GitApprentice/ideas $ git commit
[master 57f31b3] Added new book entry and marked Git book
complete
 1 file changed, 2 insertions(+), 1 deletion(-)
```

That's it! There's your first commit. One file changed, with two insertions and one deletion. That matches up with what you saw in `git diff` earlier in the chapter.

Now that you've learned how to commit changes to your files, you'll take a look at adding new files and directories to repositories.

Adding directories

You have directories in your project to hold ideas for books, videos and articles. But it would be good to have a directory to also store ideas for written tutorials. So you'll create a directory and an idea file, and add those to your repository.

Back in your terminal program, execute the following command to create a new directory named **tutorials**:

```
mkdir tutorials
```

Then, confirm that the directory exists, using the `ls` command:

```
~/GitApprentice/ideas $ ls
LICENSE      articles     tutorials
README.md    books        videos
```

So the directory is there; now you can see how Git recognizes the new directory. Execute the following command:

```
git status
```

You'll see the following:

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

nothing to commit, working tree clean
```

Er, that doesn't seem right. Why can't Git see your new directory? That's by design, and it reflects the way that Git thinks about files and directories.

How Git views your working tree

At its core, Git really only knows about *files*, and nothing about *directories*. Git thinks about a file as "a string that points to an entity that Git can track". If you think about this, it makes some sense: If a file can be uniquely referenced as the full path to the file, then tracking directories separately is quite redundant.

For instance, here's a list of all the files (excluding hidden files, hidden directories and empty directories) currently in your project:

```
ideas/LICENSE
ideas/README.md
ideas/articles/clickbait_ideas.md
ideas/articles/live_streaming_ideas.md
ideas/articles/ios_article_ideas.md
ideas/books/book_ideas.md
ideas/videos/content_ideas.md
ideas/videos/platform_ideas.md
```

This is a simplified version of how Git views your project: a list of paths to files that are tracked in the repository. From this, Git can easily and quickly re-create a directory and file structure when it clones a repository to your local system.

You'll learn more about the inner workings of Git in the intermediate section of this book, but, for now, you simply need to figure out how to get Git to pick up a new directory that you want to add to the repository.

.keep files

The solution to making Git recognize a directory is clearly to put a file inside of it. But what if you don't have anything yet to put here, or you want an empty directory to show up in everyone's clone of this project?

The solution is to use a placeholder file. The usual convention is to create a hidden, zero-byte **.keep** file inside the directory you want Git to "see."

To do this, first navigate into the `tutorials` directory that you just created with the following command:

```
cd tutorials
```

Then create an empty file named **.keep**, using the `touch` command for expediency:

```
touch .keep
```

> **Note**: The `touch` command was originally designed to set and modify the "modified" and "accessed" times of existing files. But one of the nice features of `touch` is that, if a specified file doesn't exist, `touch` will automatically create the file for you.
>
> `touch` is a nice alternative to opening a text editor to create and save an empty file. Experienced command line users take advantage of this shortcut much of the time.

Execute the following command to view the contents of this directory, including hidden dotfiles:

```
ls -a
```

You should see the following:

```
~/GitApprentice/ideas/tutorials $ ls -a
.   ..  .keep
```

There's your hidden file. Let's see what Git thinks about this directory now. Execute the following command to move back to the main project directory:

```
cd ..
```

Now, execute `git status` to see Git's understanding of the situation:

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

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

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

Git now understands that there's something in that directory, but that it's **untracked**, which means you haven't yet added whatever's in that directory to the repository. Adding the contents of that directory is easy to do with the `git add` command.

Execute the following command, which is a slightly different form of `git add`:

```
git add tutorials/\*
```

> **Note**: The weird formatting above with the two slashes should work equivalently in a bash shell, or a ksh shell, which is the current default on newer macOS systems.

While you *could* have just used `git add .` as before to add all files, this form of `git add` is a nice way to *only* add the files in a particular directory or subdirectory. In this case, you're telling Git to stage all files underneath the **tutorials** directory.

Git now tells you that it's tracking this file, and that it's in the staging area:

```
~/GitApprentice/ideas $ 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)
```

```
    new file:   tutorials/.keep
```

You can now commit this addition to the repository. But, instead of invoking that whole business with Vim and a text editor, there's a shortcut way to commit a file to the repository and add a message all in one shot.

Execute the following command to commit the staged changes to your repository:

```
git commit -m "Adding empty tutorials directory"
```

You'll see the following, confirming your change committed:

```
~/GitApprentice/ideas $ git commit -m "Adding empty tutorials
directory"
[master ce6971f] Adding empty tutorials directory
 1 file changed, 0 insertions(+), 0 deletions(-)
 create mode 100644 tutorials/.keep
```

> **Note**: Depending on the project or organization you're working with, you'll often find that there are standards around what to put inside Git commit messages.
>
> The early portions of this book kept things simple with a single-line commit message, but as you advance in your Git career you'll see why following some standards like the 50/72 rule proposed by Tim Pope at https://tbaggery.com/2008/04/19/a-note-about-git-commit-messages.html will make your life easier when you get deeper into Git.

Once again, use `git status` to see that there's nothing left to commit:

```
~/GitApprentice/ideas $ git status
On branch master
Your branch is ahead of 'origin/master' by 2 commits.
  (use "git push" to publish your local commits)

nothing to commit, working tree clean
```

You may have realized that all these little commits give you a piecemeal view of what Git is doing with your files. And, as you keep working on your project, you'll probably want to see a historical view of what you've done. Git provides a way to view the history of your files, also known as the **log**.

Looking at git log

You've done a surprising number of things over the last few chapters. To see what you've done, execute the following command:

```
git log
```

You'll get a pile of output; I've shown the first few bits of my log below:

```
commit 761a50d148a9d241712e3be4630db3dad6e010c8 (HEAD -> master)
Author: Chris Belanger <chris@example.com>
Date:   Sun Jun 16 06:53:03 2019 -0300

    Adding empty tutorials directory

commit dbcfe56fa47a1a1547b8268a60e5b67de0489b95
Author: Chris Belanger <chris@example.com>
Date:   Sun Jun 16 06:51:54 2019 -0300

    Added new book entry and marked Git book complete

commit c47084959448d2e0b6877832b6bd3ae70f70b187 (origin/master,
origin/HEAD)
Author: Chris Belanger <chris@razeware.com>
Date:   Thu Jan 10 10:32:55 2019 -0400

    Going to try this livestreaming thing

commit 629cc4d309cdcfe508791b09da447c3633448f07
Author: Chris Belanger <chris@razeware.com>
Date:   Thu Jan 10 10:32:17 2019 -0400

    Some scratch ideas for the iOS team
.
.
.
.
```

You'll see all of your commits, in reverse chronological order.

> **Note**: Depending on the number of lines you can see at once in your terminal program, your output may be paginated, using a reader like `less`. If you see a colon on the last line of your terminal screen, this is likely the case. Simply press the **Space bar** to read subsequent pages of text.
>
> When you get to the end of the file, you'll see (END). At any point, you can press the **Q** key to quit back to your command prompt.

The output above shows you your own commit messages, which are useful... to a point. Since Git knows everything about your files, you can use `git log` to see every detail of your commits, such as the actual changes, or diff, of each commit.

To see this, execute the following command:

```
git log -p
```

This shows you the actual diffs of your commits, to help you see what specifically changed. Here's a sample from my results:

```
commit ce6971fbdb945fc5fb01b739b9dea9c9ae193cae (HEAD -> master)
Author: Chris Belanger <chris@razeware.com>
Date:   Wed Jan 16 08:22:36 2019 -0400

    Adding empty tutorials directory

diff --git a/tutorials/.keep b/tutorials/.keep
new file mode 100644
index 0000000..e69de29

commit 57f31b37ea843d1f0692178c99307d96850eca57
Author: Chris Belanger <chris@razeware.com>
Date:   Fri Jan 11 10:16:13 2019 -0400

    Added new book entry and marked Git book complete

diff --git a/books/book_ideas.md b/books/book_ideas.md
index 1a92ca4..5086b1f 100644
--- a/books/book_ideas.md
+++ b/books/book_ideas.md
@@ -7,4 +7,5 @@
 - [ ] CVS by tutorials
 - [ ] Fortran for fun and profit
 - [x] RxSwift by tutorials
-- [ ] Mastering Git
+- [x] Mastering Git
+- [ ] Care and feeding of developers
.
.
.
```

In reverse chronological order, I've added the **.keep** file to the **tutorials** directory, and made some modifications to the **book_ideas.md** file.

> **Note**: Chapter 6, "Git Log & History," will take an in-depth look at the various facets of `git log`, and it will show you how to use the various options of `git log` to get some really interesting information about the activity on your repository.

Now that you have a pretty good understanding of how to stage changes and commit them to your repository, it's time for the challenge for this chapter!

Challenge

Challenge: Add some tutorial ideas

You have a great directory to store tutorial ideas, so now it's time to add those great ideas. Your tasks in this challenge are:

1. Create a new file named **tutorial_ideas.md** inside the **tutorials** directory.
2. Add a heading to the file: `# Tutorial Ideas`.
3. Populate the file with a few ideas, following the format of the other files, for example, `[ ] Mastering PalmOS`.
4. Save your changes.
5. Add those changes to the staging area.
6. Commit those staged changes with an appropriate message.

If you get stuck, you can always find the solution to this challenge under this chapter's projects/challenge folder inside the materials you downloaded for this book.

As well, if you want to compare the state of your repo or directories with mine at this point, you'll also find my **ideas** directory zipped in that same folder.

Key points

- A **commit** is essentially a snapshot of the particular state of the set of files in the repository at a point in time.
- The **working tree** is the collection of project files that you work with directly.
- `git status` shows you the current state of your working tree.
- Git thinks about the files in your working tree as being in three distinct states: unmodified, modified and staged.
- `git add <filename>` lets you add changes from your working tree to the staging area.
- `git add .` adds all changes in the current directory and its subdirectories.
- `git add <directoryname>/*` lets you add all changes in a specified directory.
- `git diff` shows you the difference between your working tree and the staging area.
- `git diff --staged` shows you the difference between your staging area and the last commit to the repository.
- `git commit` commits all changes in the staging area and opens Vim so you can add a commit message.
- `git commit -m "<your message here>"` commits your staged changes and includes a message without having to go through Vim.
- `git log` shows you the basic commit history of your repository.
- `git log -p` shows the commit history of your repository with the corresponding diffs.

Where to go from here?

Now that you've learned how to build up commits in Git, head on to the next chapter where you'll learn more about the art of staging your changes, including how Git understands the moving and deleting of files, how to undo staged changes that you didn't actually mean to make, and your next new commands: `git reset`, `git mv` and `git rm`.

Chapter 4: The Staging Area

In previous chapters, you've gained some knowledge of the staging area of Git: You've learned how to stage modifications to your files, stage the addition of new files to the repository, view diffs between your working tree and the staging area, and you even got a little taste of how `git log` works.

But there's more to the staging area than just those few operations. At this point, you may be wondering why the staging area is necessary. "Why can't you just push all of your current updates to the repository directly?", you may ask. It's a good question, but there are issues with that linear approach; Git was actually designed to solve some of the common issues with direct-commit history that exist under other version control systems.

In this chapter, you'll learn a bit more about how the staging area of Git works, why it's necessary, how to undo changes you've made to the staging area, how to move and delete files in your repository, and more.

Why staging exists

Development is a messy process. What, in theory, should be a linear, cumulative construction of functionality in code, is more often than not a series of intertwining, non-linear threads of dead-end code, partly finished features, stubbed-out tests, collections of `// TODO:` comments in the code, and other things that are inherent to a human-driven and largely hand-crafted process.

It's noble to think that that you'll work on just one feature or bug at a time; that your working tree will only ever be populated with clean, fully documented code; that you'll never have unnecessary files cluttering up your working tree; that the configuration of your development environment will always be in perfect sync with the rest of your team; and that you won't follow any rabbit trails (or create a few of your own) while you're investigating a bug.

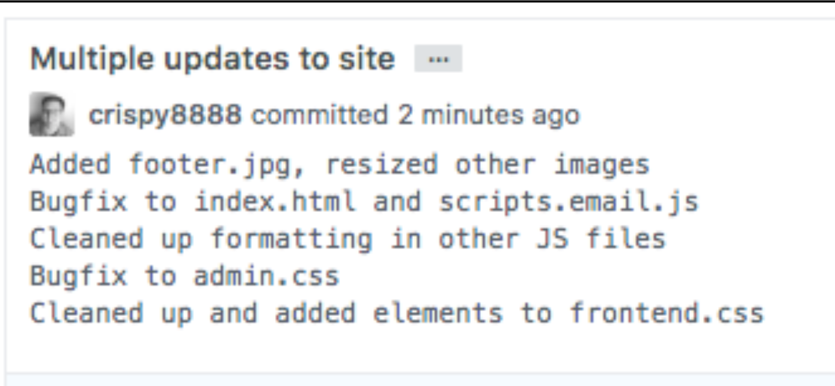
Git was built to compensate for this messy, non-linear approach to development. It's possible to work on *lots* of things at once, and selectively choose what you want to stage and commit to the repository. The general philosophy is that a commit should be a logical collection of changes *that make sense as a unit* — not just "the latest collection of things I updated that may or may not be related."

A simple staging example

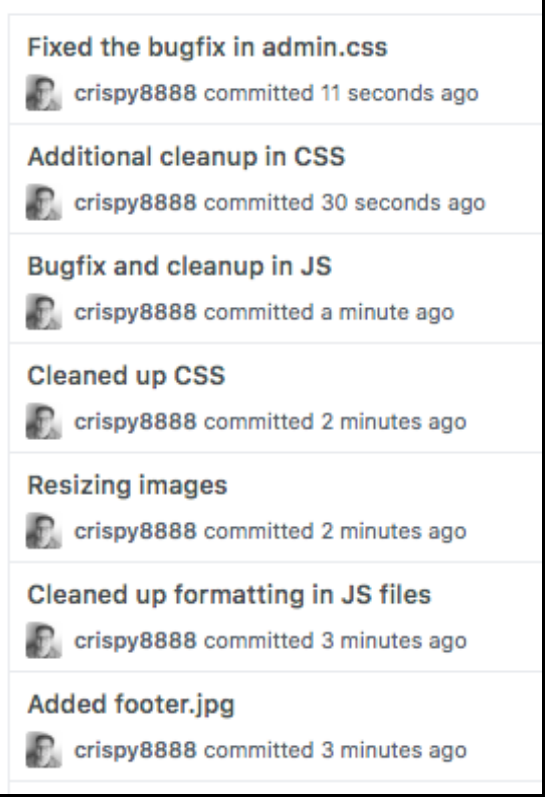
In the example below, I'm working on a website, and I want my design guru to review my CSS changes. I've changed the following files in the course of my work:

```
index.html

images/favicon.ico
images/header.jpg
images/footer.jpg
images/profile.jpg

styles/admin.css
styles/frontend.css

scripts/main.js
scripts/admin.js
scripts/email.js
```

I’ve updated a bunch of files, here, not just the CSS. And if I had to commit *everything* I had changed in my working directory, all at once, I’d have everything jammed into one commit:

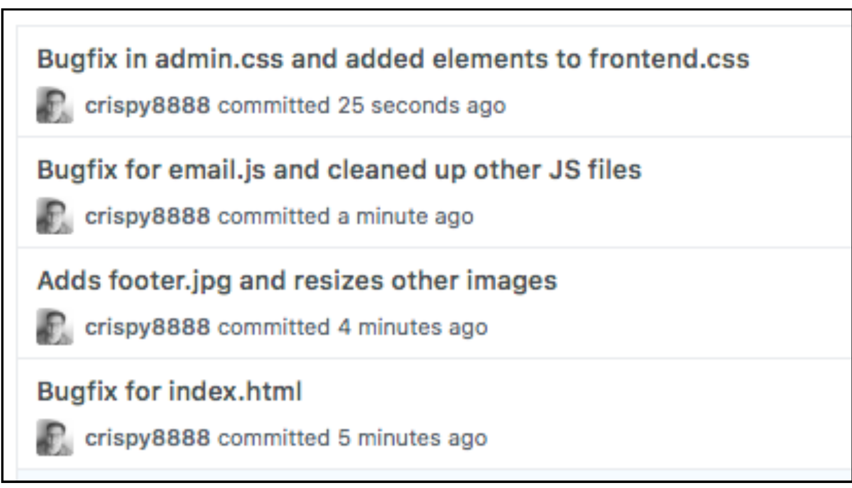


And if I committed each little change as I made it, my commit history might look like the following:



Then, when my design guru wants to take a look at the CSS changes, she'll have to wade through my commit messages and potentially look through my diffs, or even ping me on Slack to figure out what files she's supposed to review.

But, instead, if I were to stage and commit the HTML change first, followed by the image changes, followed by the JavaScript changes, and then the CSS changes after that, the commit history, and even the mental picture of what I did, becomes a *lot* more clear:



In later chapters of the book, you'll come to understand the power of being able to consciously choose various changes to stage for commit, and even choose just a portion of a file to stage for commit. But, for now, you'll explore a few more common scenarios, involving moving files, deleting files, and even undoing your changes that you weren't *quite* ready to commit.

Undoing staged changes

It's quite common that you'll change your mind about a particular set of staged changes, or you might even use something like `git add .` and then realize that there was something in there you didn't quite want to stage.

You've got a file already for book ideas, but you also want to capture some ideas for non-technical management books. Not *everyone* wants to learn how to program, it seems.

Head back to your terminal program, and create a new file in the **books** directory, named **management_book_ideas.md**:

```
touch books/management_book_ideas.md
```

But, wait — the video production team pings you and urgently requests that you update the video content ideas file, since they've just found someone to create the "Getting started with Symbian" course, and, oh, could you also add, "Advanced MOS 6510 Programming" to the list?

OK, not a huge issue. Open up **videos/content_ideas.md**, mark the "Getting started with Symbian" entry as complete by putting an "x" between the brackets, and add a line to the end for the "Advanced MOS 6510 Programming" entry. When you're done, your file should look like this:

```
# Content Ideas

Suggestions for new content to appear as videos:

[x] Beginning Pascal
[ ] Mastering Pascal
[x] Getting started with Symbian
[ ] Coding for the Psion V
[ ] Flash for developers
[ ] Advanced MOS 6510 Programming
```

Now, execute the following command to add those recent changes to your staging area:

```
git add .
```

Execute the following command to see what Git thinks about the current state of things:

```
git status
```

You should see the following:

```
Your branch is ahead of 'origin/master' by 3 commits.
  (use "git push" to publish your local commits)

Changes to be committed:
  (use "git restore --staged <file>..." to unstage)
  new file:   books/management_book_ideas.md
  modified:   videos/content_ideas.md
```

Oh, crud. You accidentally added that empty **books/management_book_ideas.md**. You likely didn't want to commit that file just yet, did you? Well, now you're in a pickle. Now that something is in the staging area, how do you get rid of it?

Fortunately, since Git understands everything that's changed so far, it can easily revert your changes for you. The easiest way to do this is through `git reset`.

git reset

Execute the following command to remove the change to **books/management_book_ideas.md** from the staging area:

```
git reset HEAD books/management_book_ideas.md
```

`git reset` restores your environment to a particular state. But wait — what's this `HEAD` business?

`HEAD` is simply a label that references the most recent commit. You may have already noticed the term `HEAD` in your console output while working through earlier portions of the book.

In case you missed it, execute the following command to look at the log:

```
git log
```

If you look at the top lines of the output in your console, you'll see something similar to the following:

```
commit 6c88142dc775c4289b764cb9cf2e644274072102 (HEAD -> master)
Author: Chris Belanger <chris@razeware.com>
Date:   Sat Jan 19 07:16:11 2019 -0400

    Adding some tutorial ideas
```

That `(HEAD -> master)` note tells you that the latest commit on your local system is as you expect — the commit where you added those tutorial ideas — and that this commit was done on the `master` branch. You'll get into branches a little later in this section, but, for now, simply understand that `HEAD` keeps track of your latest commit.

So, `git reset HEAD books/management_book_ideas.md`, in this context means "use `HEAD` as a reference point, restore the staging area to that point, but only restore any changes related to the **books/management_book_ideas.md** file."

To see that this is actually the case, exit out of `git log` with `Q` if necessary, and execute `git status` once again:

```
~/MasteringGit/ideas $ git status
Your branch is ahead of 'origin/master' by 3 commits.
  (use "git push" to publish your local commits)

Changes to be committed:
  (use "git restore --staged <file>..." to unstage)
  modified:   videos/content_ideas.md

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

That looks better: Git is no longer tracking **books/management_book_ideas.md**, but it's still tracking your changes to **videos/content_ideas.md**. Phew — you're back to where you wanted to be.

Better commit that last change before you get into more trouble. Execute the following command to add another commit:

```
git commit -m "Updates book ideas for Symbian and MOS 6510"
```

Now, you've been thinking a bit, and you don't think you should keep those ideas about the video platform itself in the **videos** folder. They more appropriately belong in a new folder: **website**.

Moving files in Git

Create the folder for the website ideas with the following command:

```
mkdir website
```

Now, you need to move that file from the **videos** directory to the **website** directory. Even with your short experience with Git, you probably suspect that it's not quite as simple as just moving the file from one directory to the other. That's correct, but it's instructive to see *why* this is.

So, you'll move it the brute force way first, and see how Git interprets your actions. Execute the following command to use the standard `mv` command line tool to move the file from one directory to the other:

```
mv videos/platform_ideas.md website
```

Now, execute `git status` to see what Git thinks about what you've done:

```
~/MasteringGit/ideas $ git status
On branch master
Your branch is ahead of 'origin/master' by 4 commits.
  (use "git push" to publish your local commits)

Changes not staged for commit:
  (use "git add/rm <file>..." to update what will be committed)
  (use "git restore <file>..." to discard changes in working
directory)
  deleted:    videos/platform_ideas.md

Untracked files:
  (use "git add <file>..." to include in what will be committed)
  books/management_book_ideas.md
  website/

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

Well, that's a bit of a mess. Git thinks you've deleted a file that is being tracked, and it also thinks that you've added this **website** bit of nonsense. Git doesn't seem so smart after all. Why doesn't it just *see* that you've moved the file?

The answer is in the way that Git thinks about files: as full paths, not individual directories. Take a look at how Git saw this part of the working tree before the move:

```
videos/platform_ideas.md (tracked)
videos/content_ideas.md (tracked)
```

And, after the move, here's what it sees:

```
videos/platform_ideas.md (deleted)
videos/content_ideas.md (tracked)
website/platform_ideas.md (untracked)
```

Remember, Git knows nothing about directories: It only knows about full paths. Comparing the two snippets of your working tree above shows you exactly why `git status` reports what it does.

Seems like the brute force approach of `mv` isn't what you want. Git has a built-in `mv` command to move things "properly" for you.

Move the file back with the following command:

```
mv website/platform_ideas.md videos/
```

Now, execute the following:

```
git mv videos/platform_ideas.md website/
```

And execute `git status` to see what's up:

```
 ~/MasteringGit/ideas $ git status
On branch master
Your branch is ahead of 'origin/master' by 4 commits.
  (use "git push" to publish your local commits)

Changes to be committed:
  (use "git restore --staged <file>..." to unstage)
  renamed:    videos/platform_ideas.md -> website/
platform_ideas.md

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

That looks better. Git sees the file as "renamed," which makes sense, since Git thinks about files in terms of their full path. And Git has also staged that change for you. Nice!

Commit those changes now:

```
git commit -m "Moves platform ideas to website directory"
```

Your ideas project is now looking pretty ship-shape. But, to be honest, those live streaming ideas are pretty bad. Perhaps you should just get rid of them now before too many people see them.

Deleting files in Git

The impulse to just delete/move/rename files as you'd normally do on your filesystem is usually what puts Git into a tizzy, and it causes people to say they don't "get" Git. But if you take the time to instruct Git on what to do, it usually takes care of things quite nicely for you.

So — that live streaming ideas file has to go. The brute-force approach, as you may guess, isn't the best way to solve things, but let's see if it causes Git any grief.

Execute the following command to delete the live streaming ideas file with the `rm` command:

```
rm articles/live_streaming_ideas.md
```

And then execute `git status` to see what Git's reaction is:

```
~/MasteringGit/ideas $ git status
On branch master
Your branch is ahead of 'origin/master' by 5 commits.
  (use "git push" to publish your local commits)

Changes not staged for commit:
  (use "git add/rm <file>..." to update what will be committed)
  (use "git restore <file>..." to discard changes in working
directory)
  deleted:    articles/live_streaming_ideas.md

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

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

Oh, that's not so bad. Git recognizes that you've deleted the file and is prompting you to stage it.

Do that now with the following command:

```
git add articles/live_streaming_ideas.md
```

Then, see what's up with `git status`:

```
~/MasteringGit/ideas $ git status
On branch master
Your branch is ahead of 'origin/master' by 5 commits.
  (use "git push" to publish your local commits)

Changes to be committed:
  (use "git restore --staged <file>..." to unstage)
  deleted:    articles/live_streaming_ideas.md

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

Well, that was a bit of a roundabout way to do things. But just like `git mv`, you can use the `git rm` command to do this in one fell swoop.

Restoring deleted files

First, you need to get back to where you were. Unstage the change to the live streaming ideas file with your best new friend, `git reset`:

```
git reset HEAD articles/live_streaming_ideas.md
```

That removes that change from the staging area — but it doesn't *restore* the file itself in your working tree. To do that, you'll need to tell Git to retrieve the latest committed version of that file from the repository.

Execute the following to restore your file to its original infamy:

```
git checkout HEAD articles/live_streaming_ideas.md
```

You're back to where you started.

Now, get rid of that file with the following command:

```
git rm articles/live_streaming_ideas.md
```

And, finally, commit that change with an appropriate message:

```
git commit -m "Removes terrible live streaming ideas"
```

Looks like you'll have to leave the live streaming to the experts: fourteen-year-olds on YouTube with too much time on their hands and too little common sense.

That empty file for management book ideas is still hanging around. Since you don't have any good ideas for that file yet, you may as well commit it and hope that someone down the road can populate it with good ways to be an effective manager.

Add that empty file with the following command:

```
git add books/management_book_ideas.md
```

And commit it with a nice comment:

```
git commit -m "Adds all the good ideas about management"
```

It's not all bad: Abandoning your attempts to building a career in live streaming *and* management gives you more time to take on this next challenge!

Challenge

Challenge: Move, delete and restore a file

This challenge takes you through the paces of what you just learned. You'll need to do the following:

1. Move the newly added **books/management_book_ideas.md** to the **website** directory with the `git mv` command.
2. You've changed your mind and don't want **management_book_ideas.md** anymore, so remove that file completely with the `git rm` command. Git will give you an error when you do this, but look at the suggested actions in the error closely to see how to solve this problem this with the `-f` option, and try again.
3. But now you're having second thoughts: Maybe you *do* have some good ideas about management. Restore that file to its original location.

Remember to use the `git status` command to get your bearings when you need to. Liberal use of `git status` will definitely help you understand what Git is doing at each stage of this challenge.

If you get stuck, or want to check your solution, you can always find the answer to this challenge under the **challenge** folder for this chapter.

Key points

- The **staging area** lets you construct your next commit in a logical, structure fashion.
- `git reset HEAD <filename>` lets you restore your staging environment to the last commit state.
- Moving files around and deleting them from the filesystem, without notifying Git, will cause you grief.
- `git mv` moves files around and stages the change, all in one action.
- `git rm` removes files from your repository and stages the change, again, in one action.
- Restore deleted and staged files with `git reset HEAD <filename>` followed by `git checkout HEAD <filename>`

Where to go from here?

That was quite a ride! You've gotten deeper into understanding how Git sees the world; building up a parallel mental model will help you out immensely as you use Git more in your daily workflow.

Sometimes, you may have files that you explicitly *don't* want to add to your repository, but that you want to keep around in your working tree. You can tell Git to ignore things in your working tree, and even tell Git to ignore particular files across *all* of your projects through the magic of the simple file known as **.gitignore** — which you'll learn all about in the next chapter!

Chapter 5: Ignoring Files in Git

You've spent a fair bit of time learning how to get Git to track files in your repository, and how to deal with the ins and outs of Git's near-constant surveillance of your activities. So it might come as a wonder that you'd ever want Git to actively *ignore* things in your repository.

Why wouldn't you want Git to track everything in your project? Well, there are quite a few situations in which you might not want Git to track *everything*.

A good example would be any files that contain API keys, tokens, passwords or other secrets that you definitely need for testing, but you don't want them sitting in a repository — especially a public repository — for all to see.

Depending on your development platform, you may have lots of build artifacts or generated content sitting around inside your project directory, such as linker files, metadata, the resulting executable and other similar things. These files are regenerated each time you build your project, so you definitely don't want Git to track these files. And then there are those persnickety things that some OSes add into your directories without asking, such as **.DS_Store** files on macOS.

Introducing .gitignore

Git's answer to this is the **.gitignore** file, which is a set of rules held in a file that tell Git to not track files or sets of files. That seems like a very simple solution, and it is. But the real power of **.gitignore** is in its ability to pattern-match a wide range of files so that you don't have to spell out every single file you want Git to ignore, and you can even instruct Git to ignore the same types of files across multiple projects. Taking that a step further, you can have a global **.gitignore** that applies to all of your repositories, and then put project-specific **.gitignore** files within directories or subdirectories under the projects that need a particularly pedantic level of control.

In this chapter, you'll learn how to configure your own **.gitignore**, how to use some prefabricated **.gitignore** files from places like GitHub, and how to set up a global **.gitignore** to apply to all of your projects.

Getting started

Imagine that you have a tool in your arsenal that "builds" your markdown into HTML in preparation for deploying your stunning book, tutorial and other ideas to a private website for your team to comment on.

In this case, the HTML files would be the generated content that you *don't* want to track in the repository. You'd like to render them locally as part of your build process so you could preview them, but you'd never edit the HTML directly: It's always rendered using the tool.

Create a new directory in the root folder of your project to hold these generated files, using the following command:

```
mkdir sitehtml
```

Now, create an empty HTML file in there (keep that imagination going, friend), with the following command:

```
touch sitehtml/all-todos.html
```

Run `git status` to see that Git recognizes the new content:

```
/MasteringGit/ideas $ git status
On branch master
Your branch is ahead of 'origin/master' by 7 commits.
  (use "git push" to publish your local commits)
```

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

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

So Git, once again, sees what you're doing. But here's how to tell Git to turn a blind eye.

Create a new file named **.gitignore** in the root folder of your project:

```
touch .gitignore
```

And add the following line to your newly created **.gitignore** using a text editor:

```
*.html
```

Save and exit. What you've done is to tell Git, "For this project, ignore all files that match this pattern." In this case, you've asked it to ignore all files that have an **.html** extension.

Now, see what `git status` tells you:

```
~/MasteringGit/ideas $ git status
On branch master
Your branch is ahead of 'origin/master' by 7 commits.
  (use "git push" to publish your local commits)

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

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

Git sees that you've added **.gitignore**, but it no longer views that HTML file as "untracked," even through it's buried down in a subdirectory.

Now, what if you were fine with ignoring HTML files in subdirectories, but you wanted all HTML files in the top-level directory of your project to be tracked? You *could* theoretically re-create the same **.gitignore** files in each of your subdirectories and remove this top-level **.gitignore**, but that would be amazingly tedious and would not scale well.

Instead, you can use some clever pattern-matching in your top-level **.gitignore** to only ignore subdirectories.

Edit the single line in your **.gitignore** as follows:

```
*/*.html
```

Save and exit. This new pattern tells Git, "Ignore all HTML files that *aren't* in the top-level directory."

To see that this is true, create a new HTML file in the top-level directory of your project:

```
touch index.html
```

Run `git status` to see if Git does, in fact, recognize the HTML files in the top-level directory, while still ignoring the ones underneath:

```
/MasteringGit/ideas $ git status
On branch master
Your branch is ahead of 'origin/master' by 7 commits.
  (use "git push" to publish your local commits)

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

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

Git sees the top-level HTML file as untracked, but it's still ignoring the other HTML file down in the **sitehtml** directory, just as you'd planned.

Nesting .gitignore files

You can easily nest **.gitignore** files in your project. Imagine that you have a subdirectory with HTML files that are referenced from your **index.html**. These aren't generated by your imaginary build process but, rather, maintained by hand, and you want to make sure Git is able to track these.

Create a new directory and name it **htmlrefs**:

```
mkdir htmlrefs
```

Now, create an HTML file in that subdirectory:

```
touch htmlrefs/utils.html
```

And create a **.gitignore** file in that directory as well:

```
touch htmlrefs/.gitignore
```

Open `htmlrefs/.gitignore` and add the following line to it:

```
!/*.html
```

Save and exit. The exclamation mark (!) negates the pattern in this case, and the slash (/) means "start this rule from this directory." So this rule says, "Despite any higher-level rules, don't ignore any HTML files, starting in this directory or lower."

Execute `git status` to see if this is true:

```
~/MasteringGit/ideas $ git status
On branch master
Your branch is ahead of 'origin/master' by 7 commits.
  (use "git push" to publish your local commits)

Untracked files:
  (use "git add <file>..." to include in what will be committed)
  .gitignore
  htmlrefs/
  index.html

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

Git now sees the contents of your **htmlrefs** directory as untracked, just as you wanted.

Now that you're happy with the current arrangement of your **.gitignore** files, you can stage and commit those changes.

Stage all changes with the following command:

```
git add .
```

And commit those changes as well:

```
git commit -m "Adding .gitignore files and HTML"
```

Setting up **.gitignore** files on a project-by-project basis will only get you so far, though. There are things — like the aforementioned **.DS_Store** files that macOS *so* helpfully adds to your directories — that you want to ignore all of the time. Git has the concept of a **global .gitignore** that you can use for cases like this.

Looking at the global .gitignore

Execute the following command to find out if you already have a global **.gitignore**:

```
git config --global core.excludesfile
```

If that command returns nothing, then you don't have one set up just yet. No worries; it's easy to create one.

Create a file in a convenient location — in this case, your home directory — and name it something obvious:

```
touch ~/.gitignore_global
```

And now you can use the `git config` command to tell Git that it should look at this file from now on as your global **.gitignore**:

```
git config --global core.excludesfile ~/.gitignore_global
```

So now if I ask Git where my global **.gitignore** lives, it tells me the following:

```
~/MasteringGit/ideas $ git config --global core.excludesfile
/Users/chrisbelanger/.gitignore_global
```

But now that you have a global **.gitignore**… what should you put in it?

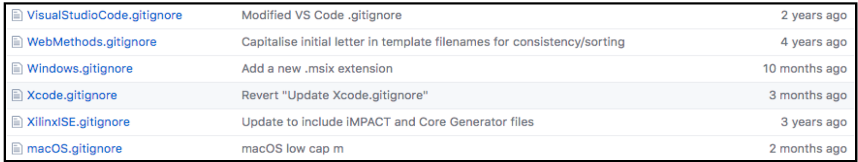
Finding sample .gitignore files

This is one of those situations wherein you don't have to reinvent the wheel. Hundreds of thousands of developers have come before you, and they've already figured out what the best configuration is for your particular situation.

One of the better collections of prefabricated **.gitignore** files is hosted by GitHub — no surprise there, I'm sure. GitHub has files for most OSes, programming languages and code editors.

Head over to https://github.com/github/gitignore and have a look through the packages it offers. Sample files that are appropriate for your OS can be found in the **Global** subfolder of the repository.

Go into the **Global** subfolder (or simply navigate to https://github.com/github/gitignore/tree/master/Global) and find the one for your local system.



There's a **Windows.gitignore**, a **macOS.gitignore**, a **Linux.gitignore** and many more, all waiting for you to add them to your own **.gitignore**. And that brings you to the challenge for this chapter!

Challenge

Challenge: Populate your global .gitignore

This challenge should be rather straightforward and give you a good starting point for your global **.gitignore**. Your goal is to find the correct **.gitignore** for your own OS, get that file from the GitHub repository, and add the contents of that file to your global **.gitignore**.

1. Navigate to https://github.com/github/gitignore/tree/master/Global.

2. Find the correct **.gitignore** for your own OS.

3. Take the contents of that OS-specific **.gitignore**, and add it to your own global **.gitignore**.

If you get stuck, or want to check your solution, you can always find the answer to this challenge under the **challenge** folder for this chapter.

Key points

- **.gitignore** lets you configure Git so that it ignores specific files or files that match a certain pattern.
- `*.html` in your **.gitignore** matches on all files with an **.html** extension, in any directory or subdirectory of your project.
- `*/*.html` matches all files with an **.html** extension, but only in subdirectories of your project.
- `!` negates a matching rule.
- You can have multiple **.gitignore** files inside various directories of your project to override higher-level matches in your project.
- You can find where your global **.gitignore** lives with the command `git config --global core.excludesfile`.
- GitHub hosts some excellent started **.gitignore** files at https://github.com/github/gitignore.

Where to go from here?

As you work on more and more complex projects, especially across multiple code-based and coding languages, you'll find that the power of the global **.gitignore**, coupled with the project-specific (and even folder-specific) **.gitignore** files, will be an indispensable part of your Git workflow.

The next chapter will take you through a short diversion into the various workings of `git log`. Yes, you've already used this command, but this command has some clever options that will help you view the history of your project in an efficient and highly readable manner.

Chapter 6: Git Log & History

You've been quite busy in your repository, adding files, making changes, undoing changes and making intelligent commits with good, clear messages. But as time goes on, it gets harder and harder to remember what you did and when you did it.

When you mess up your project (not if, but *when*), you'll want to be able to go back in history and find a commit that worked, and rewind your project back to that point in time. This chapter shows you how.

Viewing Git history

Git keeps track of pretty much everything you do in your repository. You've already seen this in action in previous chapters, when you used the `git log` command.

However, there are many ways you can view the data provided by `git log` that can tell you some incredibly interesting things about your repository and your history. In fact, you can even use `git log` to create a graphical representation of your repository to get a better mental image of what's going on.

Vanilla git log

Open your terminal app and execute `git log` to see the basic, vanilla-flavor history of your repository that you've become accustomed to:

```
commit 477e542bfa35942ddf069d85fbe3fb0923cfab47 (HEAD -> master)
Author: Chris Belanger <chris@razeware.com>
Date:   Wed Jan 23 16:49:56 2019 -0400

    Adding .gitignore files and HTML

commit ffcedc2397503831938894edffda5c5795c387ff
Author: Chris Belanger <chris@razeware.com>
Date:   Tue Jan 22 20:26:30 2019 -0400

    Adds all the good ideas about management

commit 84094274a447e76eb8f55def2c38b909ef94fa42
Author: Chris Belanger <chris@razeware.com>
Date:   Tue Jan 22 20:17:03 2019 -0400

    Removes terrible live streaming ideas

commit 67fd0aa99b5afc18b7c6cc9b4300a07e9fc88418
Author: Chris Belanger <chris@razeware.com>
Date:   Tue Jan 22 19:47:23 2019 -0400

    Moves platform ideas to website directory
```

This shows you a list of **ancestral commits** — that is, the set of commits that form the history of the current **head**. In this case, that's the most recent commit in the `master` branch of your repository. Press **Q** to exit this view.

The basic `git log` command shows you *all* the ancestral commits for this branch. What if you only wanted to see a few — say, three?

Limiting results

This is straightforward; simply execute the following command to show the number of commits you'd like to see, starting from the most recent:

```
git log -3
```

Git will then show you just the three most recent commits. You can replace the `3` in the above example to show any number of commits you'd prefer.

That's a little more manageable, but there's still a lot of detail in there. Wouldn't it be nice if there was a way to view *just* the commit messages and filter out all the other, extra information?

There is! Execute the following command to see a more compact view of the repository history:

```
git log --oneline
```

You'll see a quick, compact view of the commit history, which is arguably *far* more readable than the original output from `git log`:

```
~/GitApprentice/ideas $ git log --oneline
477e542 (HEAD -> master) Adding .gitignore files and HTML
ffcedc2 Adds all the good ideas about management
8409427 Removes terrible live streaming ideas
67fd0aa Moves platform ideas to website directory
0ddfac2 Updates book ideas for Symbian and MOS 6510
6c88142 Adding some tutorial ideas
.
.
.
```

This also shows you the **short hash** of a commit. Although you haven't looked at hashes in depth yet, there are long and short hashes for each commit that uniquely identify a commit within a repository.

For instance, if I take a look at the first line of the most recent commit on my repo with `git log -1` (that's the number "1", not the letter "l"), I see the following:

```
commit 477e542bfa35942ddf069d85fbe3fb0923cfab47 (HEAD -> master)
```

Now, to compare, I look at that same single commit with `git log -1 --oneline` (yes, you can stack multiple options with `git log`), I get the following:

```
477e542 (HEAD -> master) Adding .gitignore files and HTML
```

The short hash is simply the first seven characters of the long hash; in this case, `477e542`. For the average-sized development project, seven hexadecimal digits provides you with more than a quarter of a *billion* short hashes, so the possibility of hashes colliding between various commits is quite small.

When you ramp up to massively-sized Git repositories that live on for years, or even decades, the chance of two commits having the same hash becomes a reality.

Older versions of Git allowed you to configure the number of hash characters to use for your repository, but more recent versions of Git (from about 2017 onward) dynamically adapt this setting to suit the size of your project, so you don't usually have to worry about it.

> **Note**: Are you wondering why some options to commands are preceded with a single dash, while others are preceded with double dashes?
>
> This has its roots way back in the history of command-line-based operating systems. Generally, commands that have double dashes are the "long form" of a command, and are there for clarity.
>
> For instance, the command `git log -p`, which you've used before, shows the diffs of your commits. But there's another command that only differs by the fact that the option is in uppercase: `git log -P`, which does something *entirely* different.
>
> Since all these commands can get a bit confusing, especially where case matters, many modern command-line utilities provide long form alternatives to commands to be clearer about the the intent of a particular option.
>
> In the above example, you can use `git log --patch` and `git log -p` interchangeably, because they mean *exactly* the same thing. The `--patch` option is more clear, but `-p` is more compact.

Graphical views of your repository

So what else can `git log` do? Well, Git has some simple methods to show you the branching history of your repository. Execute the following command to see a rather verbose view of the "tree" structure of your repository history:

```
git log --graph
```

Page through a few results by pressing the Spacebar (or scroll using the arrow keys), and you'll see where I merged a branch in an early version of the repository:

```
.
.
.
 commit fbc46d3d828fa57ef627742cf23e865689bf01a0
| Author: Chris Belanger <chris@razeware.com>
| Date:   Thu Jan 10 10:18:14 2019 -0400
|
|     Adding files for article ideas
|
*   commit 5fcdc0e77adc11e0b2beca341666e89611a48a4a
|\  Merge: 39c26dd cfbbca3
| | Author: Chris Belanger <chris@razeware.com>
| | Date:   Thu Jan 10 10:14:56 2019 -0400
| |
| |     Merge branch 'video_team'
| |
| * commit cfbbca371f4ecc80796a6c3fc0c084ebe181edf0
| | Author: Chris Belanger <chris@razeware.com>
| | Date:   Thu Jan 10 10:06:25 2019 -0400
| |
| |     Removing brain download as per ethics committee
.
.
.
```

And if you page down a little more, you'll see the point where I created the branch off of `master`:

```
* | commit 39c26dd9749eb627056b938313df250b669c1e4c
| | Author: Chris Belanger <chris@razeware.com>
| | Date:   Thu Jan 10 10:13:32 2019 -0400
| |
| |     I should write a book on git someday
| |
* | commit 43b4998d7bf0a6d7f779dd2c0fa4fe17aa3d2453
|/  Author: Chris Belanger <chris@razeware.com>
|   Date:   Thu Jan 10 10:12:36 2019 -0400
|
```

```
|       Adding book ideas file
|
* commit becd762cea13859ac32841b6024dd4178a706abe
| Author: Chris Belanger <chris@razeware.com>
| Date:   Thu Jan 10 09:49:23 2019 -0400
|
|     Creating the directory structure
|
* commit 73938223caa4ad5c3920a4db72920d5eda6ff6e1
  Author: crispy8888 <chris@razeware.com>
  Date:   Wed Jan 9 20:59:40 2019 -0400

      Initial commit
```

But that's still too much information. How could you collapse this tree-like view to only see the commit messages, but still see the branching history? That's right — by stacking the options to `git log`.

Execute the following to see a more condensed view:

```
git log --oneline --graph
```

You'll see a nice, compact view of the history and branching structure:

```
~/GitApprentice/ideas $ git log --oneline --graph
* 477e542 (HEAD -> master) Adding .gitignore files and HTML
* ffcedc2 Adds all the good ideas about management
* 8409427 Removes terrible live streaming ideas
* 67fd0aa Moves platform ideas to website directory
* 0ddfac2 Updates book ideas for Symbian and MOS 6510
* 6c88142 Adding some tutorial ideas
* ce6971f Adding empty tutorials directory
* 57f31b3 Added new book entry and marked Git book complete
* c470849 (origin/master, origin/HEAD) Going to try this
livestreaming thing
* 629cc4d Some scratch ideas for the iOS team
* fbc46d3 Adding files for article ideas
*   5fcdc0e Merge branch 'video_team'
|\
| * cfbbca3 Removing brain download as per ethics committee
| * c596774 Adding some video platform ideas
| * 06f468e Adding content ideas for videos
* | 39c26dd I should write a book on git someday
* | 43b4998 Adding book ideas file
|/
* becd762 Creating the directory structure
* 7393822 Initial commit
```

Viewing non-ancestral history

Git's not showing you the *complete* history, though. It's only showing you the history of things that have happened on the `master` branch. To tell Git to show you the complete history of everything it knows about, add the `--all` option to the previous command:

```
git log --oneline --graph --all
```

You'll see that there's an `origin/clickbait` branch off of `master` that Git wasn't telling you about earlier:

```
* 477e542 (HEAD -> master) Adding .gitignore files and HTML
* ffcedc2 Adds all the good ideas about management
* 8409427 Removes terrible live streaming ideas
* 67fd0aa Moves platform ideas to website directory
* 0ddfac2 Updates book ideas for Symbian and MOS 6510
* 6c88142 Adding some tutorial ideas
* ce6971f Adding empty tutorials directory
* 57f31b3 Added new book entry and marked Git book complete
* c470849 (origin/master, origin/HEAD) Going to try this
livestreaming thing
* 629cc4d Some scratch ideas for the iOS team
| * e69a76a (origin/clickbait) Adding suggestions from Mic
| * 5096c54 Adding first batch of clickbait ideas
|/
* fbc46d3 Adding files for article ideas
*   5fcdc0e Merge branch 'video_team'
|\
| * cfbbca3 Removing brain download as per ethics committee
| * c596774 Adding some video platform ideas
| * 06f468e Adding content ideas for videos
* | 39c26dd I should write a book on git someday
* | 43b4998 Adding book ideas file
|/
```

Using Git shortlog

Git provides a very handy companion to `git log` in the form of `git shortlog`. This is a nice way to get a summary of the commits, perhaps for including in the release notes of your app. Sometimes "bug fixes and performance improvements" just isn't quite enough detail, you know?

Execute the following command to see who's made commits to this repository:

```
git shortlog
```

I see the following collection of commits for this repository:

```
Chris Belanger (18):
      Creating the directory structure
      Adding content ideas for videos
      Adding some video platform ideas
      Removing brain download as per ethics committee
      Adding book ideas file
      I should write a book on git someday
      Merge branch 'video_team'
      Adding files for article ideas
      Some scratch ideas for the iOS team
      Going to try this livestreaming thing
      Added new book entry and marked Git book complete
      Adding empty tutorials directory
      Adding some tutorial ideas
      Updates book ideas for Symbian and MOS 6510
      Moves platform ideas to website directory
      Removes terrible live streaming ideas
      Adds all the good ideas about management
      Adding .gitignore files and HTML

crispy8888 (1):
      Initial commit
.
.
.
```

I can see that I have 18 commits to this repository — and then there's this `crispy8888` chap that created the initial repository. Well, that was nice of him. There are likely other changes from other users in there, including yourself.

You'll notice that, in contrast to the standard `git log` command, `git shortlog` orders the commits in increasing time order. That makes more sense from a summary standpoint than showing everything in reverse-time order.

So far, you've seen how to use `git log` and `git shortlog` to give you a high-level view of the repository history with as much detail as you like. But sometimes you want to see a particular action in the repository. You know what you want to search for, but do you really have to scroll through all that output to retrieve what you're looking for?

Git provides some excellent search functionality that you can use to find information about one particular file, or even particular changes across many files.

Searching Git history

Imagine that you wanted to see just the commits that this `crispy8888` fellow had made in the repository. Git gives you the ability to filter the output of `git log` to a particular author.

Execute the following command:

```
git log --author=crispy8888 --oneline
```

Git shows you the one change this fellow made:

```
7393822 Initial commit
```

If you want to search on a name that consists of two or more parts, simply enclose the name in quotation marks:

```
git log --author="Chris Belanger" --oneline
```

You can also search the commit messages of the repository, independent of who made the change.

Execute the following to find the commits, which have a commit message that contains the word "ideas":

```
git log --grep=ideas --oneline
```

You should see something similar to the following:

```
ffcedc2 Adds all the good ideas about management
8409427 Removes terrible live streaming ideas
67fd0aa Moves platform ideas to website directory
0ddfac2 Updates book ideas for Symbian and MOS 6510
6c88142 Adding some tutorial ideas
629cc4d Some scratch ideas for the iOS team
fbc46d3 Adding files for article ideas
43b4998 Adding book ideas file
c596774 Adding some video platform ideas
06f468e Adding content ideas for videos
```

> **Note**: Wondering what `grep` means? `grep` is a reference to a command line tool that stands for “global search regular expression and print”. `grep` is a wonderfully useful and powerful command line tool, and “grep” has come to be recognized in general usage as a verb that means “search,” especially in conjunction with regular expressions.

What if you’re interested in just a single file? That’s easy to do in Git.

Execute the following command to see all of the full commit messages for **books/book_ideas.md**:

```
git log --oneline books/book_ideas.md
```

You’ll see all the commits for just that file:

```
57f31b3 Added new book entry and marked Git book complete
39c26dd I should write a book on git someday
43b4998 Adding book ideas file
```

You can also see the commits that happened to the files in a particular directory:

```
git log --oneline books
```

This shows you all the changes that happened in that directory, but it’s not clear *which* files were changed.

To get a clearer picture of which files were changed in that directory, you can throw the `--stat` option on top of that command:

```
git log --oneline --stat books
```

This shows you the following details about the changes in this directory so that you can see what was changed, and even get a glimpse into how much was changed:

```
ffcedc2 Adds all the good ideas about management
 books/management_book_ideas.md | 0
 1 file changed, 0 insertions(+), 0 deletions(-)
57f31b3 Added new book entry and marked Git book complete
 books/book_ideas.md | 3 ++-
 1 file changed, 2 insertions(+), 1 deletion(-)
39c26dd I should write a book on git someday
 books/book_ideas.md | 1 +
 1 file changed, 1 insertion(+)
43b4998 Adding book ideas file
 books/book_ideas.md | 9 +++++++++
 1 file changed, 9 insertions(+)
becd762 Creating the directory structure
 books/.keep | 0
 1 file changed, 0 insertions(+), 0 deletions(-)
```

You can also search the actual contents of the commit itself; that is, the changeset of the commit. This lets you look inside of your commits for particular words of interest or even whole snippets of code.

Find all of the commits in your code that deal with the term “Fortran” with the following command:

```
git log -S"Fortran"
```

You’ll see the following:

```
commit 43b4998d7bf0a6d7f779dd2c0fa4fe17aa3d2453
Author: Chris Belanger <chris@razeware.com>
Date:   Thu Jan 10 10:12:36 2019 -0400

    Adding book ideas file
```

There’s just the one commit, where the book ideas file was initially added. But, again, that’s not quite enough detail. Can you recall which option you can use to show the actual changes in the commit?

That's right: It's the -p option. Execute the command above, but this time, add the -p option to the end:

```
git log -S"Fortran" -p
```

You'll see a bit more detail now:

```
commit 43b4998d7bf0a6d7f779dd2c0fa4fe17aa3d2453
Author: Chris Belanger <chris@razeware.com>
Date:   Thu Jan 10 10:12:36 2019 -0400

    Adding book ideas file

diff --git a/books/book_ideas.md b/books/book_ideas.md
new file mode 100644
index 0000000..f924368
--- /dev/null
+++ b/books/book_ideas.md
@@ -0,0 +1,9 @@
+# Ideas for new book projects
+
+- [ ] Hotubbing by tutorials
+- [x] Advanced debugging and reverse engineering
+- [ ] Animal husbandry by tutorials
+- [ ] Beginning tree surgery
+- [ ] CVS by tutorials
+- [ ] Fortran for fun and profit
+- [x] RxSwift by tutorials
```

That's better! You can now see the contents of that commit, where Git found the term "Fortran".

You've learned quite a lot about `git log` in this chapter, probably more than the average Git user knows. As you use Git more and more in your workflow, and as the history of your project grows from months to years, you'll find that `git log` will eventually be your best friend, and better at recalling things than your brain could ever be.

Challenges

Speaking of brains, why don't you exercise yours and reinforce the skills you learned in this chapter by taking on the four challenges of this chapter?

Challenge 1: Show all the details of commits that mark items as "done"

For this challenge, you need to find all of the commits where items have been ticked off as "done"; that is, ones that have an "x" inside the brackets, like so:

```
[x]
```

You'll need to search for the above string, and you'll need to use an option to not only show the basic commit details, but also show the contents of the changeset of the commit.

Challenge 2: Find all the commits with messages that mention "streaming"

You want to search through the commit **messages** to find where you or someone else has used the term "streaming" in the commit message itself, not necessarily in the content of the commit. Tip: What was that strangely named command you learned about earlier in this chapter?

Challenge 3: Get a detailed history of the videos directory

For this challenge, you need to show everything that's happened inside the **videos** directory, as far as Git's concerned. But, once again, the basic information about the commit is not enough. You also need to show the full details about that diff. So you'll tag a familiar option on to the end of the command… or can you?

Challenge 4: Find detailed information about all commits that contain "iOS 13"

In this final challenge, you need to find the commits whose diffs contain the term "iOS 13." This sounds similar to Challenge 1 above, but if you try to use the same command as you did in that challenge, you won't find any results. But trust me, there is at least one result in there. Tip: Did you remember to search "all" of the repository?

Key points

- `git log` by itself shows a basic, vanilla view of the ancestral commits of the current `HEAD`.
- `git log -p` shows the diff of a commit.
- `git log -`*n* shows the last *n* commits.
- `git log --oneline` shows a concise view of the short hash and the commit message.
- You can stack options on `git log`, as in `git log -8 --oneline` to show the last 8 commits in a condensed form.
- `git log --graph` shows a crude but workable graphical representation of your repository.
- `git log --all` shows commits on other branches in the repository, not just the ancestors of the current `HEAD`.
- `git shortlog` shows a summary of commits, grouped by their author them, in increasing time order.
- `git log --author="<authorname>"` lets you search for commits by a particular author.
- `git log --grep="<term>"` lets you search commit messages for a particular term.
- `git log <path/to/filename>` will show you just the commits associated with that one file.
- `git log <directory>` will show you the commits for files in a particular directory.
- `git log --stat` shows a nice overview of the scope and scale of the change in each commit.
- `git log -S"<term>"` lets you search the contents of a commit's changeset for a particular term.

Where to go from here?

You've learned a significant amount about how Git works under the hood, how commits work, how the staging area works, how to undo things you didn't mean to do, how to ignore files and how to leverage the power of `git log` to unravel the secrets of your repository.

But one thing you haven't yet really touched on is what makes Git so elegant and useful: its powerful branching model.

In fact, Git's branching mechanism is what sets it apart from most other version control systems, since it works extremely well with the way most developers go about their projects.

In the next chapter, you'll learn what `master` really means, how to create branches, how Git "thinks" about branches in your repository, the difference between local and remote repositories, how to switch branches, how to delete branches and more.

Chapter 7: Branching

One of the driving factors behind Git's original design was to support the messy, non-linear approach to development that stems from working on large-scale, fast-moving projects. The need to split off development from the main development line, make changes independently and in isolation of other changes on the main development line, easily merge those changes back in, and do all this in a lightweight manner, was what drove the creators of Git to build a very lightweight, elegant model to support this kind of workflow.

In this chapter, you'll explore the first half of this paradigm: **branching**. You've touched on branching quite briefly in Chapter 1, "A Crash Course in Git," but you probably didn't quite understand what you, or Git, were *doing* in that moment.

Although you can hobble through your development career never really understanding how branching in Git actually works, branching is *incredibly* important to the development workflows of many development teams, both large and small, so knowing what's going on under the hood, and having a solid mental model of your repository's branching structure will help you immensely as your projects grow in size and complexity.

What is a commit?

That question was asked and answered in a shallow manner a few chapters ago, but it's a good time to revisit that question and explore commits in more detail.

Recall that a commit represents the state of your project tree — your directory — at a particular point in time:

```
├── LICENSE
├── README.md
├── articles
│   ├── clickbait_ideas.md
│   ├── ios_article_ideas.md
│   └── live_streaming_ideas.md
├── books
│   └── book_ideas.md
└── videos
    ├── content_ideas.md
    └── platform_ideas.md
```

You probably think about your files primarily in terms of their content, their position inside the directory hierarchy, and their names. So when you think of a commit, you're likely to think about the state of the files, their content and names at a particular point in time. And that's correct, to a point: Git also adds some more information to that "state of your files" concept in the form of metadata.

Git metadata includes such things like "when was this committed?" and "who committed this?", but most importantly, it includes the concept of "where did this commit originate from?" — and that piece of information is known as the commit's **parent**. A commit can have one or two parents, depending on how it was branched and merged back in, but you'll get to that point later.

Git takes all that metadata, including a reference to this commit's parent, and wraps that up with the state of your files as the commit. Git then **hashes** that collection of things using **SHA1** to create an ID, or **key**, that is unique to that commit inside your repository. This makes it extremely easy to refer to a commit by its hash value, or as you saw in the previous chapter, its short hash.

What is a branch?

The concept of a branch is massively simple in Git: It's simply a reference, or a label, to a commit in your repository. That's it. Really. And because you can refer to a commit in Git simply through its hash, you can see how creating branches is a terribly cheap operation. There's no copying, no extra cloning, just Git saying "OK, your new branch is a label to commit `477e542`". Boom, done.

As you make commits on your branch, that label for the branch gets moved forward and updated with the hash of each new commit. Again, all Git does is update that label, which is stored as a simple file in that hidden **.git** repository, as a really cheap operation.

You've been working on a branch all along — did you realize that? Yes, `master`, or `main`, or whatever you've chosen as the originating branch of your repository, is nothing but a regular branch. It's only by convention, and the default name that Git applies to this default branch when it creates a new repository, that we say "Oh, the `master` branch is the *original* branch."

> **Note**: As of version 2.28.0, Git now provides a setting by which you can control the label to be used when you create the first branch in a new repository. This defaults to `master`, but you can choose to set this to `main` or whatever you like.
>
> The setting is `init.defaultBranch`, and you can change it with the following command:`$ git config --global init.defaultBranch main`
>
> This sets the default branch name to `main`.
>
> This only affects new repositories that you create; it doesn't change the default branch name of any existing repositories.

There's nothing special about `master` or `main`; again, Git simply knows that the `master` or `main` branch is a revision in your repository pointed to by a simple label held in a file on disk. Sorry to dash any notion that `master` or `main` was magic or something.

Creating a branch

You created a branch before in the crash-course chapter, but now you're going to create a branch and watch exactly what Git is doing.

The command to create a branch in Git is, unsurprisingly, `git branch`, followed by the name of your branch.

Execute the following command to create a new branch:

```
git branch testBranch
```

Git finishes that action with little fanfare, since a new branch is not a big deal to Git.

How Git tracks branches

To see that Git actually *did* something, execute the following command to see what Git's done in the background:

```
ls .git/refs/heads/
```

This directory contains the files that point to all of your branches. I get the following result of two files in that directory:

```
master      testBranch
```

Oh, that's interesting — a file named **testBranch**, the same as your branch name. Take a look at **testBranch** to see what's inside, using the following command:

```
cat .git/refs/heads/testBranch
```

Wow — Git *is* really bare-bones about branches. All that's in there is a single hash value. To take this to a new level of pedantry, you can prove that the label **testBranch** is pointing to the actual latest commit on your repository.

Execute the following to see the latest commit:

```
git log -1
```

You'll see something like the following (your hash will be different than mine):

```
commit 477e542bfa35942ddf069d85fbe3fb0923cfab47 (HEAD -> master,
testBranch)
```

```
Author: Chris Belanger <chris@razeware.com>
Date:   Wed Jan 23 16:49:56 2019 -0400

    Adding .gitignore files and HTML
```

Let’s pick this apart a little. The commit referenced here is, indeed, the same hash as contained in **testBranch**. The next little bit, `(HEAD -> master, testBranch)`, means that this commit is pointed to by *both* the `master` and the `testBranch` branches. The reason this commit is pointed to by both labels is because you’ve only created a new branch, and not created any more commits on this branch. So the label can’t move forward until you make another commit.

Checking your current branch

Git can easily tell you which branch you’re on, if you ever need to know. Execute the following command to verify you’re working on **testbranch**:

```
git branch
```

Without any arguments or options, `git branch` simply shows you the list of local branches on your repository. You should have the two following branches listed:

```
* master
  testBranch
```

The asterisk indicates that you’re still on the **master** branch, even though you’ve just created a new branch. That’s because Git won’t switch to a newly created branch unless you tell it explicitly.

Switching to another branch

To switch to **testBranch**, execute the `checkout` command like so:

```
git checkout testBranch
```

Git responds with the following:

```
Switched to branch 'testBranch'
```

That’s really all there is to creating and switching between branches.

> **Note**: Admittedly, the term `checkout` is a bit of a misnomer, since if you've ever owned a library card, you know that checking out a book makes that book inaccessible to anyone else until you return it.
>
> That term is a holdover from the way that some older version control systems functioned, as they used a lock-modify-unlock model, which prevented anyone else from modifying the file at the same time. It worked really well for preventing merge conflicts, but pretty much killed any form of distributed, concurrent development.
>
> Speaking of old version control systems, if any of you used PVCS Version Manager back in the day (c. 2000 or so), drop me a line and we can swap horror stories about the amazingly sparse documentation, the endless fighting with semaphores, and all the other fun bits that came along with that piece of software.

That's enough poking around with **testBranch**, so switch back to **master** with the following command:

```
git checkout master
```

You really don't need **testBranch** anymore, since there are other, real branches to be explored. Delete **testBranch** with the following command:

```
git branch -d testBranch
```

Time to take a look at some real branches. You already have one in your repository, just waiting for you to go in and start doing some work… what's that? Oh, you don't remember seeing that branch when you last executed `git branch`? That's because `git branch` by itself only shows the local branches in your repository.

When you first cloned this repository (which was a fork from the original **ideas** repository), Git started tracking both the local repository, as well as the **remote** repository — i.e., the forked repository that you created on GitHub. Git knows about the branches on the remote as well as on your local system.

So because of this synchronization between your local repository and the remote repository, Git knows that any commits you make locally — and will likely push back to the remote — belong on a particular, matching, remote branch. Equally well, Git knows that any changes made on a branch on the remote — perhaps by a fellow developer somewhere in the world — belong in a specific, matching directory on your local system.

Viewing local and remote branches

To see all of the branches that Git knows about on this repository, either local or remote, execute the following command:

```
git branch --all
```

Git will respond with something similar to the following:

```
* master
  remotes/origin/HEAD -> origin/master
  remotes/origin/clickbait
  remotes/origin/master
```

Git shows you all of the branches in your local and remote repositories. In this case, the remote only has one branch: **clickbait**. All of the other branches listed are effectively **master** or pointers to **master**.

You have some work to do on the clickbait branch. If everyone else is doing it, you should, too, right? To get this branch down to your machine, tell Git to start tracking it, and switch to this branch all in one action, execute the following command:

```
git checkout --track origin/clickbait
```

Git responds with the following:

```
Branch 'clickbait' set up to track remote branch 'clickbait'
from 'origin'.
Switched to a new branch 'clickbait'
```

Explaining origin

OK, what is this `origin` thing that you keep seeing?

`origin` is another one of those convenience conventions that Git uses. Just like **master** is the default name for the first branch created in your repository, `origin` is the default alias for the location of the remote repository from where you cloned your local repository.

To see this, execute the following command to see where Git thinks `origin` lives:

```
git remote -v
```

You should see something similar to the following:

```
origin  https://www.github.com/belangerc/ideas (fetch)
origin  https://www.github.com/belangerc/ideas (push)
```

You'll have something different in your URLs, instead of `belangerc`. But you can see here that `origin` is simply an alias for the URL of the remote repository. That's all.

To see Git's view of all local and remote branches now, execute the following command:

```
git branch --all -v
```

Git will respond with its understanding of the current state of the local and remote branches, with a bit of extra information provided by the -v (verbose) option:

```
* clickbait                e69a76a Adding suggestions from Mic
  master                   477e542 [ahead 8] Adding .gitignore
files and HTML
  remotes/origin/HEAD      -> origin/master
  remotes/origin/clickbait e69a76a Adding suggestions from Mic
  remotes/origin/master    c470849 Going to try this
livestreaming thing
```

Git tells you that you are on the **clickbait** branch, and you can also see that the hash for the local **clickbait** branch is the same as the remote one, as you'd expect.

Of interest is the **master** branch, as well. Git is tracking your local **master** branch against the remote one, and it knows that your local **master** branch is eight commits ahead of the remote. Git will also let you know if you're behind the remote branch as well; that is, if there are any commits on the remote branch that you haven't yet pulled down to your local branch.

Viewing branches graphically

To see a visual representation of the current state of your local branches, execute the following command:

```
git log --oneline --graph
```

The tip of the graph, which is the latest commit, tells you where you are:

```
* e69a76a (HEAD -> clickbait, origin/clickbait) Adding
suggestions from Mic
```

Your current `HEAD` points to the clickbait branch, and you're at the same point as your remote repository.

A shortcut for branch creation

I confess, I took you the long way 'round with that command `git checkout --track origin/clickbait`, but seeing the long form of that command hopefully helped you understand what Git actually *does* when it checks out and tracks a branch from the remote.

There's a much shorter way to checkout and switch to an existing branch on the remote: `git checkout clickbait` works equally well, and is a bit easier to type and to remember.

When you specify a branch name to `git checkout`, Git checks to see if there is a local branch that matches that name to switch to. If not, then it looks to the `origin` remote, and if it finds a branch on the remote matching that name, it assumes that is the branch you want, checks it out for you, and switches you to that branch. Rather nice of it to take care of all that for you.

There's also a shortcut command which solves the two-step problem of `git branch <branchname>` and `git checkout <branchname>`: `git checkout -b <branchname>`. This, again, is a faster way to create a local branch.

Now that you have seen how to create, switch to, and delete branches, it's time for the short challenge of this chapter, which will serve to reinforce what you've learned and show you what to do when you want to delete a local branch that already has a commit on it.

Challenge

Challenge: Delete a branch with commits

You don't want to muck up your existing branches for this challenge, so you'll need to create a temporary local branch, switch to it, make a commit, and then delete that branch.

1. Create a temporary branch with the name of **newBranch**.
2. Switch to that branch.
3. Use the `touch` command to create an empty **README.md** file in the root directory of your project.
4. Add that new **README.md** file to the staging area.
5. Commit that change with an appropriate message.
6. Checkout the **master** branch.
7. Delete **newBranch** — but Git won't let you delete this branch in its current state. Why?
8. Follow the suggestion that Git gives you to see if you can delete this branch.

Remember to use `git status`, `git branch` and `git log --oneline --graph --all` to help get your bearings as you work on this challenge.

If you get stuck, or want to check your solution, you can always find the answer to this challenge under the **challenge** folder for this chapter.

Key points

- A commit in Git includes information about the state of the files in your repository, along with metadata such as the commit time, the commit creator, and the commit's parent or parents.
- The hash of your commit becomes the unique ID, or key, to identify that particular commit in your repository.
- A branch in Git is simply a reference to a particular commit by way of its hash.
- `master` is simply a convenience convention, but has come to be accepted as the original branch of a repository. `main` is also another common convenience branch name in lieu of `master`.
- Use `git branch <branchname>` to create a branch.
- Use `git branch` to see all local branches.
- Use `git checkout <branchname>` to switch to a local branch, or to checkout and track a remote branch.
- Use `git branch -d <branchname>` to delete a local branch.
- Use `git branch --all` to see all local and remote branches.
- `origin`, like `master`, is simply a convenience convention that is an alias for the URL of the remote repository.
- Use `git checkout -b <branchname>` to create and switch to a local branch in one fell swoop.

Where to go from here?

Get used to branching in Git, because you'll be doing it often. Lightweight branches are pretty much *the* reason that Git has drawn so many followers, as it matches the workflow of concurrent development teams.

But there's little point in being able to branch and work on a branch, without being able to get your work joined back up to the main development branch. That's **merging**, and that's exactly what you'll do in the next chapter!

Chapter 8: Merging

Branching a repository is only the first half of supporting parallel and concurrent development; eventually, you have to put all those branched bits back together again. And, yes, that operation can be as complex as you think it might be!

Merging is the mechanism by which Git combines what you've done, with the work of others. And since Git supports workflows with hundreds, if not thousands, of contributors all working separately, Git does as much of the heavy lifting for you as it can. Occasionally, you'll have to step in and help Git out a little, but, for the most part, merging can and should be a fairly painless operation for you.

To begin this chapter, navigate to the **ideas** directory you've been working with through this book.

A look at your branches

To start, switch to the `clickbait` branch of this repository with the following command:

```
git checkout clickbait
```

If you were to visualize the branching history of your current **ideas** repository, with you sitting on the `clickbait` branch, it would look something like this :

```
* ac21935 (master) Adding .gitignore files and HTML
* 3113a85 Adds all the good ideas about management
* 600f9d0 Removes terrible live streaming ideas
* 21fcb3e Moves platform ideas to website directory
* 1cf3481 Updates book ideas for Symbian and MOS 6510
* 3ddfe7c Adding some tutorial ideas
* c689d66 Adding empty tutorials directory
* 13d9709 Added new book entry and marked Git book as complete
* c470849 (origin/master, origin/HEAD) Going to try this livestreaming thing
* 629cc4d Some scratch ideas for the iOS team
| * e69a76a (HEAD -> clickbait, origin/clickbait) Adding suggestions from Mic
| * 5096c54 Adding first batch of clickbait ideas
|/
* fbc46d3 Adding files for article ideas
*   5fcdc0e Merge branch 'video_team'
|\
| * cfbbca3 Removing brain download as per ethics committee
| * c596774 Adding some video platform ideas
| * 06f468e Adding content ideas for videos
* | 39c26dd I should write a book on git someday
* | 43b4998 Adding book ideas file
|/
* becd762 Creating the directory structure
* 7393822 Initial commit
```

In the image above, you can see the following:

1. This is your local `master` branch. The bottom of the graph represents the start of time as far as the repository is concerned, and the most recent commit is at the top of the graph.

2. This is the `master` branch on `origin` — that is, the remote repository. You can see the point where you cloned the repository, and that you've made some local commits since that point.

3. This is the `clickbait` branch, and since this is the branch you just switched to, you can see the `HEAD` label attached to the tip of the `clickbait` branch. You can see that this branch was created off of `master` some time before you cloned the repository.

4. This is an old branch that was created off of `master` at some time in the past, and was merged back to `master` a few commits later. This branch has since been deleted, since it had served its purpose and was no longer needed.

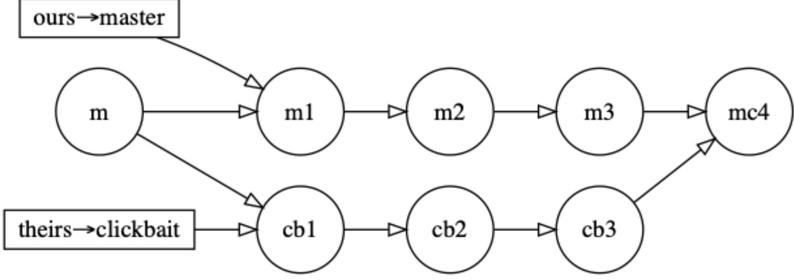
This is a fairly common development workflow; in a small team, `master` can effectively serve as the main development line, and developers make branches off of `master` to work on features or bug fixes, without messing with what's in the main development line. Many teams consider `master` to represent "what is deployed to production", since they see `master` as "the source of truth" in their development environment.

Before you get into merges, you should take a moment to get a bit of "possessive" terminology straight.

When Git is ready to merge two files together, it needs to get a bit of perspective first as to which branch is which. Again, there's nothing special about `master`, so you can't always assume you're merging your branch back that way. In practice, you'll find that you often merge between branches that *aren't* `master`.

So, therefore, Git thinks about branches in terms of **ours** and **theirs**. "Ours" refers to the branch to which you're merging back to, and "theirs" refers to the branch that you want to pull into "ours".

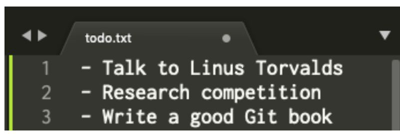
Let's say you want to merge the `clickbait` branch back into `master`. In this case, as shown in the diagram below, `master` is **ours** and the `clickbait` branch would be **theirs**. Keeping this distinction straight will help you immeasurably in your merging career.



Three-way merges

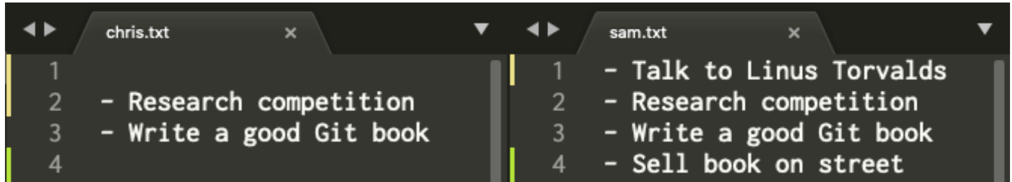
You might think that merging is really just taking two revisions, one on each branch, and mashing them together in a logical manner. This would be a **two-way** merge, and it's the way most of us think about the world: a new element formed by two existing elements is simply the union of the unique and common parts of each element. However, a merge in Git actually uses *three* revisions to perform what is known as a **three-way merge**.

To see why this is, take a look at the two-way merge scenario below. You have one simple text file; you're working on one copy of the file while your friend is working on another, separate copy of that same file.



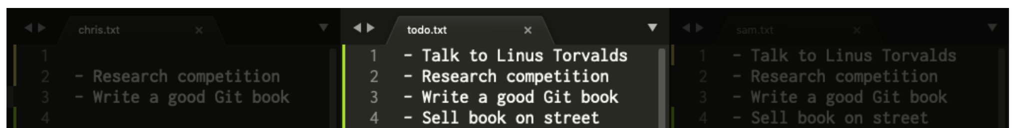
The original file.

You delete a line from the top of the file, and your friend adds a line to the bottom of the file.



Chris' changes on the left; Sam's changes on the right.

Now imagine that you and your friend hand off your work to an impartial third party to merge this text file together. Now, this third party has literally no idea as to what the original state of this file was, so she has to make a guess as to what she should take from each file.

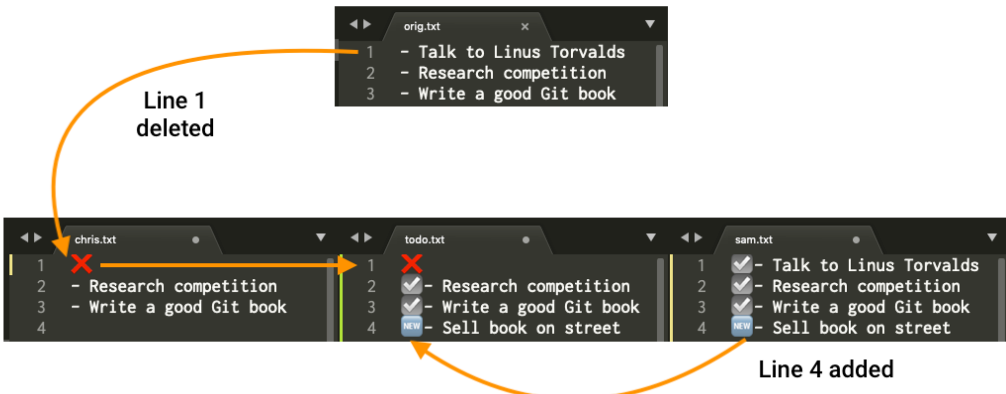


With no background of what the starting point was, the person responsible to merge tries to preserve as many lines as possible in common to both files.

The end result is not quite what you intended, is it? You've ended up with all four lines; the impartial third party reviewer probably assumed Sam added a line to the top as well as a line to the bottom of Chris' work.

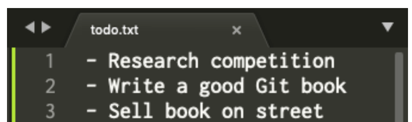
To perform an educated merge of these two files, your impartial third party has to know about the **common ancestor** of both of these files. This common ancestor is the third revision that comes in to play with a three-way merge.

Now, imagine you and your friend *also* provided the original file that you both started with — the common ancestor — to your impartial third party. She could compare each new file's changes to the original file, figure out the diff of your changes, figure out the diff of your friend's changes, and create the correct resulting merged document from the diffs of each.



Knowing the origin of each set of changes lets you detect that Line 1 was deleted by Chris, and Line 4 was added by Sam.

That's better. And this, essentially, is what Git does in an automated fashion. By performing three-way merges on your content, Git gets it right most of the time. Once in a while, Git won't be able to figure things out on its own, and you'll have to go in there and help it out a little bit. But you'll get into these scenarios a little later on in this book when you work on **merge conflicts**, which are a lot less scary than they sound.



The result is what you both intended.

It's time for you to try out some merging yourself. Open up Terminal, navigate to the folder that houses your repository, and get ready to see how merging works in action.

Merging a branch

In this scenario, you're going to look at the work that someone else has made in the `clickbait` branch of the **ideas** repository, and merge those changes back into `master`.

Make sure you're on the `clickbait` branch by executing the following command (if you haven't already done this):

```
git checkout clickbait
```

Execute the following command to see what's been committed on this branch that you'll want to merge back to `master`:

```
git log clickbait --not master
```

This little gem is quite nice to keep on hand, as it tells you "what are the commits that are just in the `clickbait` branch, but not in `master`?" Just executing `git log` shows you *all* history of this branch, right back to the original creation of the `master` branch, which is too much information for your purposes.

You'll see the following output:

```
commit e69a76a6febf996a44a5de4dda6bde8569ef02bc (HEAD ->
clickbait, origin/clickbait)
Author: Chris Belanger <chris@razeware.com>
Date:   Thu Jan 10 10:28:14 2019 -0400

    Adding suggestions from Mic

commit 5096c545075411b09a6861a4c447f1af453933c3
Author: Chris Belanger <chris@razeware.com>
Date:   Thu Jan 10 10:27:10 2019 -0400

    Adding first batch of clickbait ideas
```

Ok, there's two changes to merge back in; guess you'd better get cracking and merge these clickbait ideas before you lose any more traffic to your site.

To see the contents of the new file that's in this branch, execute the following command:

```
cat articles/clickbait_ideas.md
```

Some great ideas in there, for sure.

Recall that merging is the action of **pulling in changes** that have been done on another branch. In this case, you want to pull the changes from `clickbait` into the `master` branch. To do that, you'll have to be on the `master` branch first.

Execute the following to move to the `master` branch:

```
git checkout master
```

Now, what's in that **articles/clickbait_ideas.md** you looked at in the other branch? Execute that same command, again:

```
cat articles/clickbait_ideas.md
```

There's nothing in there. That's OK — you'll soon fill up that file with the ideas you're merging from the `clickbait` branch.

You're now back on the `master` branch, ready to pull in the changes from the `clickbait` branch. Execute the following command to merge the changes from `clickbait` to `master`:

```
git merge clickbait
```

Oh, heck, you're back in Vim. Well, at least Git has created a nice default message for you: `Merge branch 'clickbait' into master`. That's enough detail for this merge, so simply accept this commit message and exit out:

- Press **:** (colon) to enter Command mode.
- Type **wq** and press Enter to write this file and quit the Vim editor.

As soon as you quit Vim, Git starts the merge operation for you and commits that merge, and it's likely done even before you know it.

Now, you can take a look at Git's graphical representation of the repository at this point with `git log --oneline --graph --all`:

```
*   55fb2dc (HEAD -> master) Merge branch 'clickbait' into
master
|\
| * e69a76a (origin/clickbait, clickbait) Adding suggestions
from Mic
| * 5096c54 Adding first batch of clickbait ideas
* | 477e542 Adding .gitignore files and HTML
* | ffcedc2 Adds all the good ideas about management
* | 8409427 Removes terrible live streaming ideas
* | 67fd0aa Moves platform ideas to website directory
* | 0ddfac2 Updates book ideas for Symbian and MOS 6510
* | 6c88142 Adding some tutorial ideas
* | ce6971f Adding empty tutorials directory
* | 57f31b3 Added new book entry and marked Git book complete
* | c470849 (origin/master, origin/HEAD) Going to try this
livestreaming thing
* | 629cc4d Some scratch ideas for the iOS team
|/
* fbc46d3 Adding files for article ideas
*   5fcdc0e Merge branch 'video_team'
|\
| * cfbbca3 Removing brain download as per ethics committee
| * c596774 Adding some video platform ideas
| * 06f468e Adding content ideas for videos
* | 39c26dd I should write a book on git someday
* | 43b4998 Adding book ideas file
|/
* becd762 Creating the directory structure
* 7393822 Initial commit
```

You can see at the top of the graph that Git has merged in your `clickbait` branch to `master` and that `HEAD` has now moved up to the latest revision, i.e., your merge commit.

If you want to prove that the file has now been brought into the `master` branch, execute the following command:

```
cat articles/clickbait_ideas.md
```

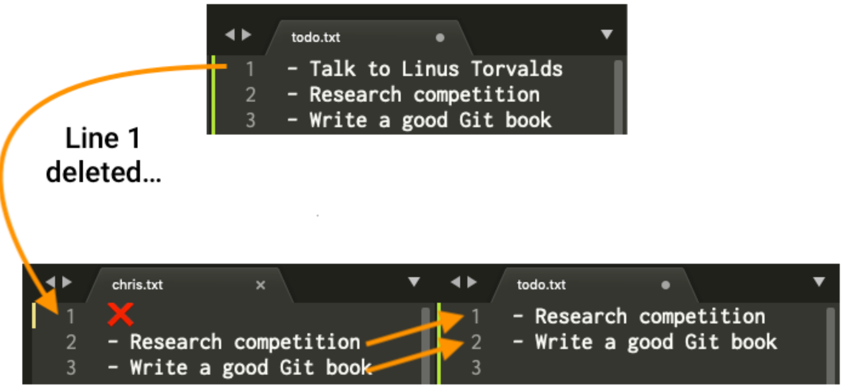
You'll see the contents of the file spat out to the console.

Fast-forward merge

There's another type of merge that happens in Git, known as the **fast-forward** merge. To illustrate this, think back to the example above, where you and your friend were working on a file. Your friend has gone away (probably hired away by Google or Apple, lucky sod), and you're now working on that file by yourself.

Once you've finished your revisions, you take your updated file, along with the original file (the common ancestor, again) to your impartial third party for merging. She's going to look at the common ancestor file, along with your new file, but she isn't going to see a third file to merge.

In this case, she's just going to commit your file on top of of the old file, because *there's nothing to merge*.



If there are no other changes to the file to merge, Git simply commits your file over top of the original.

If no other person had touched the original file since you picked it up and started working on it, there's no real point in doing anything fancy, here. And while Git is far from lazy, it is terribly efficient and only does the work it absolutely needs to do to get the job done. This, in effect, is exactly what a fast-forward merge does.

To see this in action, you'll create a branch off of `master`, make a commit, and then merge the branch back to `master` to see how a fast-forward merge works.

First, execute the following to ensure you're on the `master` branch:

```
git checkout master
```

Now, create a branch named `readme-updates` to hold some changes to the README.md file:

```
git checkout -b readme-updates
```

Git creates that branch and automatically switches you to it. Now, open **README.md** in your favorite text editor, and add the following text to the end of the file:

```
This repository is a collection of ideas for articles, content
and features at raywenderlich.com.

Feel free to add ideas and mark taken ideas as "done".
```

Save your changes, and return to Terminal. Stage your changes with the following command:

```
git add README.md
```

Now, commit that staged change with an appropriate message:

```
git commit -m "Adding more detail to the README file"
```

Now, to merge that change back to `master`. Remember — you need to be on the branch you want to pull the changes *into*, so you'll have to switch back to `master` first:

```
git checkout master
```

Now, before you merge that change in, take a look at Git's graph of the repository, using the `--all` flag to look on all branches, not just `master`:

```
git log --oneline --graph --all
```

Take a look at the top two lines of the result:

```
* 78eefc6 (readme-updates) Adding more detail to the README file
*   55fb2dc (HEAD -> master) Merge branch 'clickbait' into
master
```

Git doesn't represent this as a fork in the branch — because it doesn't need to. Just as you saw in the example above with the single file, there's no need to merge anything, here. And that begs the question: If there's nothing to merge here, what will the resulting commit look like?

Time to find out! Execute the following command to merge `readme-updates` to `master`:

```
git merge readme-updates
```

Git tells you that it's done a fast-forward merge, right in the output:

```
~/GitApprentice/ideas $ git merge readme-updates
Updating 55fb2dc..78eefc6
Fast-forward
 README.md | 4 ++++
 1 file changed, 4 insertions(+)
```

You'll notice that Git didn't bring up the Vim editor, prompting you to add a commit message. You'll see why this is the case in just a moment. First, have a look at the resulting graph of the repository, using the command below:

```
git log --oneline --graph --all
```

Take a close look at the top two lines of the result. It looks like nothing much has changed, but take a look at where `HEAD` points now:

```
* 78eefc6 (HEAD -> master, readme-updates) Adding more detail to
the README file
*   55fb2dc Merge branch 'clickbait' into master
```

Here, all Git has done is move the `HEAD` label to your latest commit. And this makes sense; Git isn't going to create a new commit if it doesn't have to. It's easier to just move the `HEAD` label along, since there's nothing to merge in this case. And *that's* why Git didn't prompt you to enter a commit message in Vim for this fast-forward merge.

Forcing merge commits

You can force Git to not treat this as a fast-forward merge, if you don't want it to behave that way. For instance, you may be following a particular workflow in which you check that certain branches have been merged back to `master` before you build.

But if those branches resulted in a fast-forward merge, for all intents and purposes, it will look like those changes were done directly on `master`, which isn't the case.

To force Git to create a merge commit when it doesn't really need to, all you need to do is add the `--no-ff` option to the end of your `merge` command. The challenge for this chapter will let you create a fast-forward situation, and see the difference between a merge commit and a fast-forward merge.

> **Note**: Why wouldn't you always want a merge commit, especially if branching and merging are such cheap operations in Git? What's the point of moving `HEAD` along? Wouldn't it just be more clear to always have a merge commit?
>
> This is a question that's just about as politically loaded as the age-old PC vs. Mac debate, the Android vs. iOS debate, or the cats vs. dogs debate (in which case, the answer is "dogs," if you were wondering).
>
> This becomes particularly important on larger software projects with multiple contributors, where your commit history can have thousands upon thousands of commits over time. Merge commits can be seen as preserving the historical context of a feature or bugfix branch; it's clear that you branched, fixed, and then merged back in. Conversely, having lots of branches and merge commits — especially implicit merge commits, which you'll encounter later in this book — can make a repository's history harder to read and understand.
>
> There's no real "right" answer, here; but don't believe people on the internet who claim that "merge commits are evil," because they're not. Git's job is to do its best to record what happened in your repository, and your workflow shouldn't necessarily have to change just to make sure that your commit history is linear and clean. However, you'll undoubtedly work with teams on both sides of the issue, so as long as you understand merge commits in Git, you'll do just fine, no matter which workflow your team champions.

Challenge

Challenge: Create a non-fast-forward merge

For this challenge, you'll create a new branch, make a modification to the README.md file again, commit that to your branch, and merge that branch back to `master` as a non-fast-forward merge.

This challenge will require the following steps:

1. Ensure you're on the `master` branch.
2. Create a branch named `contact-details`.
3. Switch to that branch.
4. Edit the README.md file and add the following text to the end of the file: "Contact: support@razeware.com".
5. Save your edits to the file.
6. Stage your changes.
7. Commit your changes with an appropriate commit message, such as "Adding README contact information."
8. Switch back to the `master` branch.
9. Pull up the graph of the repository, and don't forget to use the `--all` option to see history of all branches. Make note of how `master` and `contact-details` look on this graph.
10. Merge in the changes from `contact-details`, using the `--no-ff` option.
11. Enter something appropriate in the merge message in Vim when prompted. Use the cheatsheet above to help you navigate through Vim if necessary.
12. Pull up the graph of the repository again. How can you tell that this is a merge commit, and not a fast-forward commit?

If you get stuck, or want to check your solution, you can always find the answer to this challenge under the **challenge** folder for this chapter.

Key points

- **Merging** combines work done on one branch with work done on another branch.
- Git performs **three-way merges** to combine content.
- **Ours** refers to the branch to which you want to pull changes into; **theirs** refers to the branch that has the changes you want to pull into **ours**.
- `git log <theirs> --not <ours>` shows you what commits are on the branch you want to merge, that aren't in your branch already.
- `git merge <theirs>` merges the commits on the "theirs" branch into "our" branch.
- Git automatically creates a merge commit message for you, and lets you edit it before continuing with the merge.
- A **fast-forward** merge happens when there have been no changes to "ours" since you branched off "theirs", and results in no merge commit being made.
- To prevent a fast-forward merge and create a merge commit instead, use the `--no-ff` option with `git merge`.

Where to go from here?

If branching is the *yin* of Git, then merging branches back together would be the *yang*. Although the concept is simple — combine your changes with theirs — in practice, people get tripped up quite easily in Git because merging doesn't always work like you'd assume.

The next chapter, Syncing with a Remote, takes you beyond your local environment, and shows you how to synchronize your local changes with what's up on the server.

Chapter 9: Syncing With a Remote

Up to this point in the book, you've worked pretty much exclusively on your local system, which isn't to say that's a bad thing — having a Git repository on your local machine can support a healthy development workflow, even when you are working by yourself.

But where Git really shines is in managing distributed, concurrent development, and that's what this chapter is all about. You've done lots of great work on your machine, and now it's time to push it back to your remote repository and synchronize what you've done with what's on the server.

And there's lots of reasons to have a remote repository somewhere, even if you are working on your own. If you ever need to restore your development environment, such as after a hard drive failure, or simply setting up another development machine, then all you have to do is clone your remote repository to your clean machine.

And just because you're working on your own now doesn't mean that you won't always want to maintain this codebase yourself. Down the road, you may want another maintainer for your project, or you may want to fully open-source your code. Having a remote hosted repository makes doing that trivial.

Pushing your changes

So many things in Git, as in life, depends on your perspective. Git has perspective standards when synchronizing local repositories with remote ones: **Pushing** is the act of taking your local changes and putting them up on the server, while **pulling** is the act of pulling any changes on the server into your local cloned repository.

So you're ready to push your changes, and that brings you to your next Git command, handily named `git push`.

Execute the following command to push your changes up to the server:

```
git push origin master
```

This tells Git to take the changes from the `master` branch and synchronize the remote repository (`origin`) with your changes. You'll see output similar to the following:

```
Counting objects: 40, done.
Delta compression using up to 4 threads.
Compressing objects: 100% (36/36), done.
Writing objects: 100% (40/40), 3.96 KiB | 579.00 KiB/s, done.
Total 40 (delta 18), reused 0 (delta 0)
remote: Resolving deltas: 100% (12/12), completed with 3 local
objects.
To https://www.github.com/belangerc/ideas.git
   c470849..f5c54f0  master -> master
```

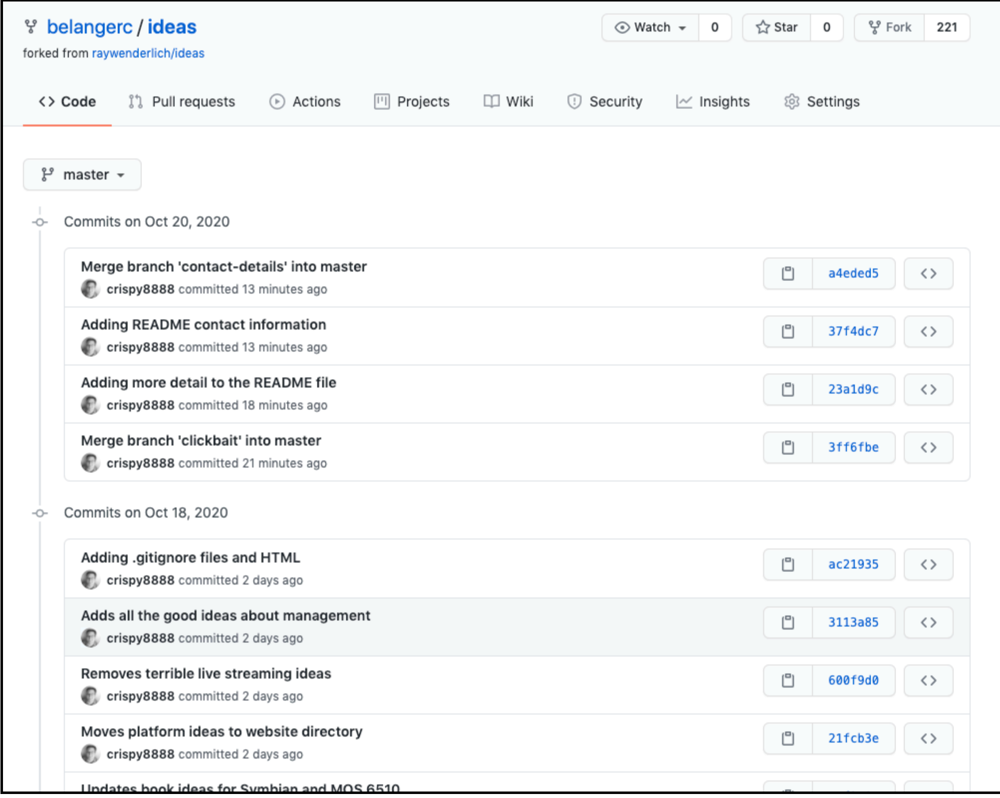
Git's given you a lot of output in this message, but essentially it's telling you some high-level information about what it's done, here: It's synchronized 12 changed items from your local repository on the remote repository.

> **Note**: Wondering why Git didn't prompt you for a commit message, here? That's because a push is not really *committing* anything; what you're doing is asking Git to take your changes and synchronize them onto the remote repository. You're combining your commits with those already on the remote, not creating a new commit on top of what's already on the remote.

Want to see the effect of your changes? Head over to the URL for your repository on GitHub. If you've forgotten what that is, you can find it in the output of your `git push` command. In my case, it's `https://www.github.com/belangerc/ideas`, but yours will have a different username in there.

Once there, click the **25 commits** link near the top of your page:

You’ll be taken to a list of all of your synchronized changes in your remote repository, and you should recognize the commits that you’ve made in your local repository:



That’s one half of the synchronization dance. And the yin to `git push`’s yang is, unsurprisingly. `git pull`.

Pulling changes

Pulling changes is pretty much the reverse scenario of pushing; Git takes the commits on the remote repo, and it integrates them all with your local commits.

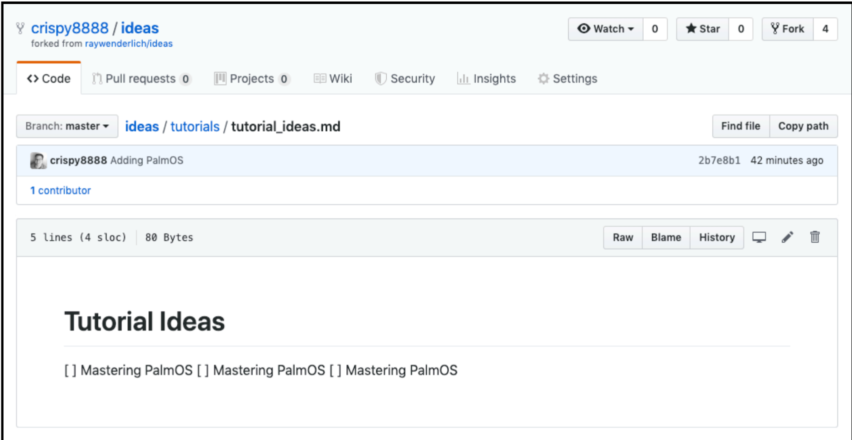
That operation is pretty straightforward when you're working by yourself on a project; you pull the latest changes from the repository, and, most likely, the remote will always be synchronized with your local, since there's no one else but you to make any changes.

But the more common scenario is that you'll be working with others in the same repository, and they will be their own pushing changes to the repository. So most of the time, you won't have the luxury of pushing your changes onto an untouched repository, and you'll have to integrate the changes on the remote by pulling them into your repository before you can push your local changes.

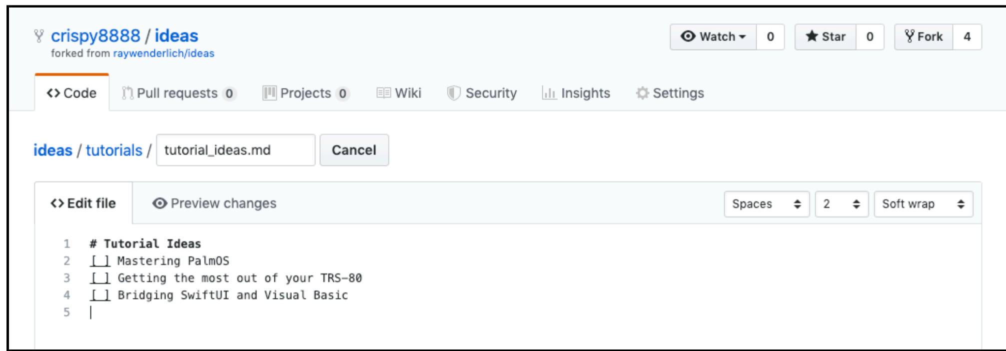
To illustrate how this works, and to illustrate what `git pull` actually does to your repository, you'll simulate a scenario wherein someone else has made a change to the `master` branch and pushed their changes before you had a chance to push yours. You'll see how Git responds to this scenario, and you'll learn the steps required to solve this issue see how to solve this issue.

Moving the remote ahead

First, you have to simulate someone else making a change on the remote. Navigate to the main page on GitHub for your repository: `https://github.com/<username>/ideas`. Once there, click on the **tutorials** directory link of your project, and then click on **tutorial_ideas.md** to view it in your browser.



Click the **edit** icon on the page (the little pencil icon), and GitHub will open a basic editor for you.



Add the following idea to **tutorial_ideas.md** in the editor:

```
[ ] Blockchains with BASIC
```

Then, scroll down to the **Commit changes** section below the editor, add a commit message of your choice in the first field of that section, leave the radio button selection as **Commit directly to the `master` branch**, and click **Commit changes**.

This creates a new commit on top of the existing `master` branch on the remote repository, just as if someone else on your development team had pushed the commits from their local system.

Now, you'll create a change to a different file in your **local** repository.

Return to your terminal program, and edit **books/book_ideas.md** and add the following line to the bottom of the file:

```
- [ ] Debugging with the Grace Hopper Method
```

Save your changes and exit.

Stage the change:

```
git add books/book_ideas.md
```

Now, create a commit on your local repository:

```
git commit -m "Adding debugging book idea"
```

You now have a commit on the head of your local `master` branch, and you also have a different commit on the head of your remote `master` branch. Now you want to push this change up to the remote. Well, that's easy. Just execute the `git push` command as you normally would:

```
git push origin master
```

Git balks, and returns the following information to you:

```
! [rejected]        master -> master (fetch first)
error: failed to push some refs to 'https://www.github.com/
belangerc/ideas'
hint: Updates were rejected because the remote contains work
that you do
hint: not have locally. This is usually caused by another
repository pushing
hint: to the same ref. You may want to first integrate the
remote changes
hint: (e.g., 'git pull ...') before pushing again.
hint: See the 'Note about fast-forwards' in 'git push --help'
for details.
```

Well, that didn't work as expected. Git is quite helpful sometimes in the hints it gives; in this case, it's telling you that it detected changes on the remote that you don't have locally. Since you'd probably want to make sure that your local changes meshed properly with the changes on the remote before you push, you'll want to pull those changes down to your local system.

Execute the following to pull the changes from the remote into your local:

```
git pull origin
```

Oh, heck, Git has opened up Vim, which means that it's creating a commit; in this case, it's creating a merge commit. Why, Git, why?

```
Merge branch 'master' of https://github.com/belangerc/ideas into
master
# Please enter a commit message to explain why this merge is
necessary,
# especially if it merges an updated upstream into a topic
branch.
#
# Lines starting with '#' will be ignored, and an empty message
aborts
# the commit.
```

You'll explore what Git is doing shortly, but finish this commit first and let Git get on with whatever it's doing. Git has already auto-created a commit message for you, so you might as well accept that and try and figure this mess out later. Press **:**, then type **wq** and then press **Enter** to save this commit message and exit out of Vim.

You're taken back to the command prompt, so execute the following to see what Git has done for you:

```
git log --oneline --graph
```

You'll see something similar to the following:

```
*   b495cc8 (HEAD -> master) Merge branch 'master' of https://
github.com/belangerc/ideas into master
|\
| * 35054cc (origin/master, origin/HEAD) Update
tutorial_ideas.md
* | 8648645 Adding debugging book idea
|/
*   a4eded5 Merge branch 'contact-details' into master
.
.
.
```

> **Note**: Wondering what those asterisks (*) mean in the graphical representation of your tree? Since commits from different branches are shown stacked one on top of the other, the asterisks simply show you on which branch this commit was made. In this case, you can see the book idea was committed on one branch (your local `master` branch), and the other commit was created on the remote origin branch.

Working up the tree, you have a common ancestor of `a4eded5 Merge branch 'contact-details' into master`. Then you have commit `8648645`, which is the commit you made on your local repository, followed by `35054cc`, your remote commit on the GitHub repository page. And *also*, there's this `b495cc8 Merge branch 'master'` stuff at the top. And *also also*, Git shows your remote "Update tutorial_ideas.md" on a branch. But you didn't create a branch. You chose the option on the GitHub edit page to commit directly to `master`. Where did that come from?

> **Note**: It's seemingly simple scenarios like this — non-conflicting changes to distinct files resulting in a merge commit — that causes newcomers to Git to throw up their hands and say, "What the heck, Git?"
>
> This is why learning Git on the command line can be instructive, as opposed to using a Git GUI client that hides details like this. Seeing what Git is doing under the hood, and, more importantly, understanding *why*, is what will help you navigate these types of scenarios like a pro.

To understand what Git's doing, you need to dissect the `git pull` command first, since `git pull` is not one, but *two* commands in disguise.

Press **Q** to exit out of the git log viewer.

First step: Git fetch

`git pull` is really *two* commands in one: `git fetch`, followed by `git merge`.

You haven't run across `git fetch` yet. Fetching updates your local repository's hidden **.git** directory with all of the commits for this repository, both local and remote. Then, Git can figure out what to do with what it's fetched from the remote; maybe it can fast-forward merge it, maybe it can't, or maybe there's a conflict preventing Git from going any further until you fix the conflict.

Generally, it's a good idea to execute `git fetch` before pushing your changes to the remote, if you suspect that someone else may have been committing changes to that same particular branch on the remote, and you want to check out what they've done before you integrate it with your work.

When Git fetches the remote commits and brings them down to your local system, it creates a temporary reference to the tip of the remote repository's branch. Think back to when you explored a little of the Git internal file structure, and you found the file **.git/refs/heads/master** that simply contained a reference to the hash of the commit that was at the tip of the current branch (i.e., `HEAD`).

You can see this reference in your own local hidden **.git** directory.

Execute the following command:

```
ls .git
```

In the results, you should see a file named **FETCH_HEAD**. That's the temporary reference to the tip of your remote branches. Want to see what's inside? Sure thing!

Execute the following command to see the contents of **FETCH_HEAD**:

```
cat .git/FETCH_HEAD
```

You'll see a hash, along with a note of where this commit came from. In my case, I see the following at the top of that file:

```
8909ec5feb674be351d99f19c51a6981930ba285        branch 'master'
of https://github.com/belangerc/ideas
```

Second step: Git merge

So once Git has fetched all of the commits to your local system, you're essentially in a position in which you have a commit from one source — your local commit — that Git needs to combine with another commit: the remote commit. Sounds like merging a branch, doesn't it?

In fact, that's pretty much how Git views the situation. Take a look back at the state of the repository graph before you merged, reproduced here:

```
| * 35054cc (origin/master, origin/HEAD) Update
tutorial_ideas.md
* | 8648645 Adding debugging book idea
|/
*   a4eded5 Merge branch 'contact-details' into master
.
.
.
```

Merging two commits, regardless of where they came from, is essentially what you did when you merged your branches back to `master` in the previous chapter. The difference here is that Git creates a virtual "branch" that points to the commit from the remote repository, as you can see in the graphical representation of the repository tree above.

There is a way around creating a messy merge commit, that involves the Git mechanism of **rebasing**. You'll cover that method of merging in later sections of this book, but, for now, you'll simply push your changes to the remote and live with the merge commit for now.

Execute the following command to push your changes up to the remote:

```
git push origin master
```

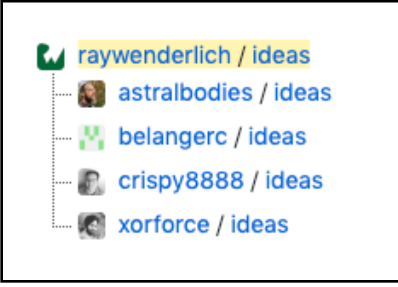
Head over to the main GitHub page for your repository, click on the **28 commits** link, and you'll see your changes up there on the remote.

Dealing with multiple remotes

There's another somewhat common synchronization scenario in which you have not one, but *two* remotes to deal with.

You've been working on your own fork of the **ideas** repository for some time, but what if there were a few changes in someone else's forked repository that you wanted to pull down to your own local system, and merge from whatever branch that user has them in, into your `master` branch?

Head over to the original **ideas** repository at https://github.com/raywenderlich/ideas. Click on the number next to the **Fork** button, and you'll see a list of all the forks that have been created from this repository:



This mysterious `crispy8888` user has created an update on his copy of the repository that you'd like to pull down and incorporate into your local repository. Click on the **ideas** link next to the `crispy8888` username, and you'll be taken to the `crispy8888` fork. Get the URL of this fork using the **Clone or Download** button.

Back in your terminal program, execute the following to add a new remote to your repository:

```
git remote add crispy8888 https://github.com/crispy8888/
ideas.git
```

This creates a new remote reference in your repository, named `crispy8888`, that points to the `crispy8888`'s fork at the above URL.

Execute the following command to see that your local repository now has another remote added to it:

```
git remote -v
```

You'll see something similar to the following:

```
crispy8888  https://github.com/crispy8888/ideas.git (fetch)
crispy8888  https://github.com/crispy8888/ideas.git (push)
origin  https://www.github.com/belangerc/ideas (fetch)
origin  https://www.github.com/belangerc/ideas (push)
```

There you are: another remote that points to someone else's fork. Now you can work with that remote, just as you did with `origin`. Remember, the name of your first remote, `origin`, is nothing more than a convention. There's nothing special about `origin`; it's just another remote, no different than the `crispy8888` one you just created. And you don't have to name your new remote the same as the account that created it; I could easily have named that remote `whatshisname` instead of `crispy8888` and things would have worked just as well.

At this point, you only have a *reference* to the remote in your local repository; you don't actually have any of the new remote's content yet. To see this, execute the following command to see the graphical view of your repository:

```
git log --oneline --graph --all
```

It looks the same as before. But didn't you just add a remote, and then use the `--all` switch above?

Even though you've instructed Git to look at all of the branches, you still can't see the changes on the `crispy8888` remote. That's because you haven't **fetched** any of the content yet from that fork; it's all still up on the server.

Execute the following command to pull down the contents of the `crispy8888` remote:

```
git fetch crispy8888
```

At the end of the output from that command, you'll see the following two lines:

```
 * [new branch]      clickbait  -> crispy8888/clickbait
 * [new branch]      master     -> crispy8888/master
```

Now you can look at the graphical representation of this repository with the following command:

```
git log --oneline --graph --all
```

Scroll down until you find the most recent entry that references the `crispy8888` remote, and , you'll see where this remote has diverged from the original:

```
*   3ff6fbe Merge branch 'clickbait' into master
|\
| | * fbe86a2 (crispy8888/clickbait) Added another clickbait
idea
| |/
| * e69a76a (origin/clickbait, clickbait) Adding suggestions
from Mic
| * 5096c54 Adding first batch of clickbait ideas
| | *   22d9abd (crispy8888/master) Merge branch 'master' of
https://github.com/crispy8888/ideas into master
| | |\
| | | * f550fed Update tutorial_ideas.md
| |_|/
|/| |
| | * f9278e6 Adding debugging book idea
| |/
|/|
```

ASCII graphing tools have their limitations, to be sure! But you get the point: there is a commit on `crispy8888/clickbait` that you'd like to pull into your own repository.

To be diligent, you should probably follow a branching workflow here so your actions are easily traceable in the log. Move to your own `clickbait` branch:

```
git checkout clickbait
```

Now you'd like to merge those two changes into your new branch. That's done in just the same way that you merge any other branch. The only difference is that you have to explicitly specify the remote that you want to merge from:

```
git merge crispy8888/clickbait
```

Git narrates every step of what it's doing like any good, modern YouTube star:

```
Updating e69a76a..9ff4582
Fast-forward
 articles/clickbait_ideas.md | 1 +
 1 file changed, 1 insertion(+)
```

Oh, that's nice — Git performed a clean fast-forward merge for you, since there were no other changes on the forked `clickbait` branch since you created your own fork. That's quite a change from your previous attempt, where you ended up with a merge commit for a simple change.

To check that Git actually created a fast-forward merge, check the first few lines of `git log --oneline --graph` (don't use the `--all` switch, so you'll just see your current branch):

```
* fbe86a2 (HEAD -> clickbait, crispy8888/clickbait) Added
another clickbait idea
* e69a76a (origin/clickbait) Adding suggestions from Mic
* 5096c54 Adding first batch of clickbait ideas
```

Are you done, yet? No, you've only merged this into your local `clickbait` branch. You still need to merge this into `master`.

First, switch to the branch you'd like to merge into:

```
git checkout master
```

Now, merge in your local `clickbait` branch as follows:

```
git merge clickbait
```

Vim opens up, so either accept the default merge message, or press **I** to enter Insert mode to improve it yourself. When done, **Escape** + **Colon** + **w** + **q** will get you out of there.

Pull up the log again, with `git log --oneline --graph` to see the current state of affairs:

```
*   72670be (HEAD -> master) Merge branch 'clickbait' into
master
|\
| * fbe86a2 (crispy8888/clickbait, clickbait) Added another
clickbait idea
* |   b495cc8 (origin/master, origin/HEAD) Merge branch 'master'
of https://github.com/belangerc/ideas into master
|\ \
| * | 35054cc Update tutorial_ideas.md
* | | 8648645 Adding debugging book idea
|/ /
.
.
.
```

At the top is your merge commit, and below that is your work done merging from the `crispy8888` remote. You can tell that Git is pushing its ASCII art graphing skills to the limit here with just three branches at play, but `git log` does nicely in a pinch when you don't have access to your usual GUI tools.

You're done, here, so all that's left is to push this merge to `origin`. Do that as you normally would with the following command:

```
git push origin master
```

You've done a *tremendous* amount in this chapter, so there's no challenge for you. You've covered more here than any average developer would likely see in the course of a few years' worth of simple pushing, pulling, branching and merging.

Key points

- Git has two mechanisms for synchronization: **pushing** and **pulling**.
- `git push` takes your local commits and synchronizes the remote repository with those commits.
- `git pull` brings the commits from the remote repository and merges them with your local commits.
- `git pull` is actually two commands in disguise: `git fetch` and `git merge`.
- `git fetch` pulls all of the commits down from the remote repository to your local one.
- `git merge` merges the commits from the remote into your local repository.
- You can't push to a remote that has any commits that you don't have locally, and that Git can't fast-forward merge.
- You can pull commits from multiple remotes into your local repository and merge them as you would commits from any other branch or remote.

Where to go from here?

You've accomplished quite a bit, here, so now that you know how to work in a powerful fashion with Git repositories, it's time to loop back around and answer two questions:

- "How do I create a Git repository from scratch?"
- "How to I create a remote repository from a local one?"

You'll answer those two questions in the next two chapters that will close out this Beginning Git section of the book, and lead you nicely into the Intermediate Git chapters to come.

Chapter 10: Creating a Repository

You’ve come a long way in your Git journey, all the way from your first commit, to learning about what Git does behind the scenes, to managing some rather complicated merge scenarios. But in all your work with repositories, you haven’t yet learned exactly *where* a repository comes from. Sure, you’ve cloned a repository, and you’ve forked repositories and worked with remotes, but how do you create a repository and a remote *from scratch*?

This chapter shows you how to create a brand-new repository on your local machine, and how to create a remote to host your brand-new repository for all to see.

Getting started

Many people will blindly tell you that the easiest way to create a repository is to "Go to GitHub, click 'New Repository', and then clone it locally." But, in most cases, you'll have a small project built up on disk before you ever think about turning it into a full-fledged repository. So this chapter will put you right into the middle of your project development and walk you through turning a simple project directory into a full-fledged repository.

But, first, you'll need a project! Check the **starter** folder for this chapter; inside, you'll find a small starter project that is the starting webpage for the sales page for this book.

Copy the entire **git-apprentice-web** directory from the **starter** folder into your main **GitApprentice** folder.

Now, open up your terminal program and navigate into the **git-apprentice-web** directory. If you've been following along with the book so far, you're likely still in the **GitApprentice/ideas** folder, so execute the following command to get into the **git-apprentice-web** subdirectory:

```
cd ../git-apprentice-web/
```

Once there, execute the following command to tell Git to set this directory up as a new repository:

```
git init
```

Git tells you that it has set up an empty repository:

```
Initialized empty Git repository in /Users/chrisbelanger/
GitApprentice/git-apprentice-web/.git/
```

Why does Git tell you it's an empty repository, when there are files in that directory? Think back to how you staged files to add to a repository: You have to use the `git add` command to tell Git what to include in the repository; Git wouldn't just assume it should pick up any old file lying around. And the same is true, here; Git has created an empty repository, just waiting for you to add some files.

Now, before you add any files, you'll want to get two things in your repository that are good hygiene for any repository that's designed to be shared online: a **LICENSE** file, and a **README** file.

Creating a LICENSE file

It's worth understanding why you need a license file, before you go and create one blindly.

Having a license file in your repository makes it clear how others may, or may not, use your code. In this modern, digital age, some people believe that copying/stealing/borrowing/reusing anything is fair game, but most people will want to respect your license terms, even though you may be providing the code freely online.

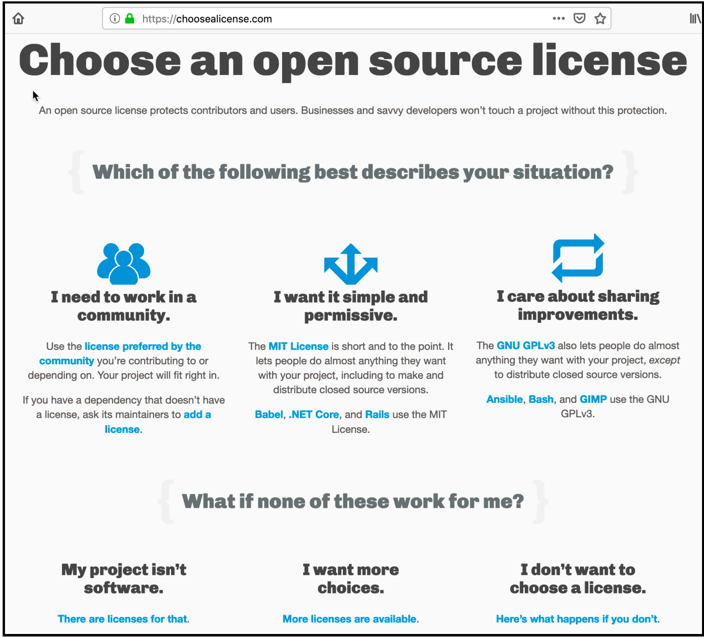
Having a license outlines how others may contribute to your project and what their rights are. The interesting bit comes in when you *don't* include a license to your work. If you create a project and stick it up on GitHub, without a license, you're stating that *no one* has the license to use your code in *any* situation — they can look at it, but that's about it.

That's all well and good if "look but don't touch" is truly what you want, but if you're inviting others to collaborate with you, then having no license means that once someone else touches the code *it's not clear who owns the copyright anymore*. Having a license file included with your code makes it clear where the ownership of this code lies.

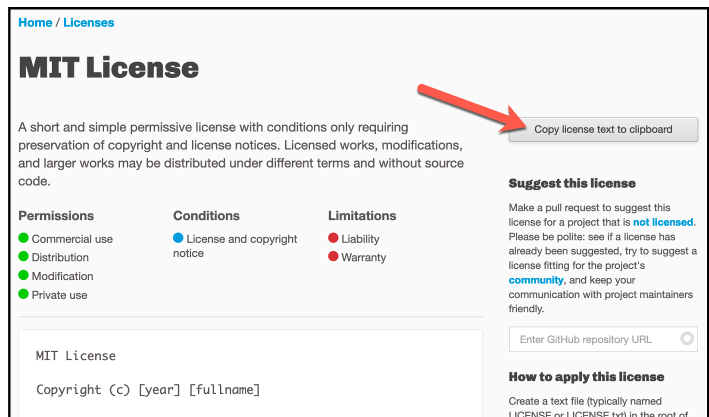
True, having a license included with your project won't protect you from code burglars who just want to take your work and use it without your permission. But what it *does* do is indicate the terms of use and reuse of your project to anyone who wants to collaborate in a fair manner, or use your work in any other manner. It's a live-and-let-live kind of thing.

Now, with that said, what kind of license should you choose? That's not always an easy question to answer. Most of the time, your projects will have just code in them, but what if they contain images? What if they contain hardware designs? 3D printing files? Your open-source book manuscript? Fonts you designed and want to open-source? What if your project is a mix of these or more?

There's a great site out there that will help you navigate the ins and outs of your project, and help you choose a license for your new project. Navigate to https://choosealicense.com/, and you'll see a lot of options:



You can explore the site at your leisure, but, in this case, I am happy for others to learn from and reuse my work in any way they like as I build up my webpage. So select the **MIT License** link, and you'll be taken to the main license page for the MIT License, which is one of the most common and most permissive licenses.



Click the **Copy license text to clipboard** button to copy the text of the MIT license to your clipboard.

Now, return to your terminal program, create a new file named **LICENSE** (yes, uppercase, and no extension required) in the root folder, and populate it with the contents of the clipboard. Save your work when you're done.

That takes care of the license file. Now, it's time to turn your attention to the README file.

Creating a README file

The README is much more straightforward than the license file. Inside the README, you can put whatever details you want people to know about you, your project, and anything that will help them get started using your project.

The common convention is to craft README files in Markdown, primarily so that they can be rendered in an easy-to-read format on the front page of your repository on GitHub, GitLab or other cloud hosts.

Create a new file in the root directory of your project named **README.md** and populate it with the following information (changing whatever you like to suit):

```
# git-apprentice-web

This is the main website for the Git Apprentice book, from
raywenderlich.com.

contact: @crispytwit
```

Save your changes and exit out of the editor.

You've got your current project, LICENSE file, and the README file — looks like you're ready to commit your files to the repository.

To see what's outstanding for your first commit, execute `git status` to see what Git's view of your working area looks like:

```
~GitApprentice/git-apprentice-web $ git status
On branch master

No commits yet

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

    LICENSE
    README.md
    css/
    images/
    index.html

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

That looks as you'd expect: The basic files for the project are there, along with the new LICENSE and README.md file.

By this point, you should be able to stage and commit this collection of files to your new repository. Try to stage and commit the complete set of files on your own first, before following the instructions below. Remember: If you mess things up, you can simply use `git reset` to revert your changes.

Stage the files for commit with the following command:

```
git add .
```

This adds everything in the current directory and subdirectories.

Now, commit your changes to the repository, providing a sensible commit message:

```
git commit -m "Initial commit of the web site, README and
LICENSE"
```

Since this is your very first commit into the repository, Git shows you a bit of different output:

```
[master (root-commit) 443f9b3] Initial commit of the web site,
README and LICENSE
 5 files changed, 111 insertions(+)
 create mode 100644 LICENSE
 create mode 100644 README.md
 create mode 100644 css/style.css
 create mode 100644 images/SFR_b+w_-_penguin.jpg
 create mode 100644 index.html
```

The very first commit to the repository is a bit special, since it doesn't have *any* parents. Recall earlier when you learned that every commit in Git has at least one parent? Well, this is a special case in which Git creates a root commit for the repository, upon which all future commits will be based.

And that's it! You've made your first commit to your repository. But you're not done — you want to get this repository pushed up to a remote for the world to *ooh* and *ahh* over. You'll do that in the second half of this chapter.

Create mode

That `create mode` is something you've seen before in the output from `git commit`, and have probably wondered about. It's of academic interest only at this point; it really doesn't affect you much at this stage of your interaction with repositories.

But in the interest of being obsessively thorough, here's what that number with `commit mode` means:

- The number after `create mode` is an octal (base 8) representation of the type of file you're creating, along with the read/write/execute permissions of that file.
- The first part of that binary number is a 4-bit value that indicates the *kind* of file you're creating. In this case, you're creating a regular file, which Git labels with `1000` in binary. There are other types, including symlinks and gitlinks, which you aren't using yet in your Git career.

- The next part of that binary number is three unused bits: `000`.
- The last part of that binary number is made of nine bits, and represents the UNIX-style permissions of this file. The first three bits hold the owner's read/write/execute permission bits, the next three bits hold the group's read/write/execute bits, and the final three bits hold the global read/write/execute bits.
- So since you own the file, Git sets the first three bits to `110` (read, write, but no execution since this isn't an executable binary or script file).
- To allow anyone in your group to read but not write to this file, Git assigns `100` (read, no write, no execute).
- To allow anyone in the world to read but not write to this file, Git assigns `100` (read, no write, no execute).
- When all of that binary is concatenated together, you have `1000` with `000` with `110100100` = `1000000110100100` as the full binary string.
- Convert `1000000110100100` to octal (base 8), and you have `100644` as a compact way to indicate the type and permissions of this file.

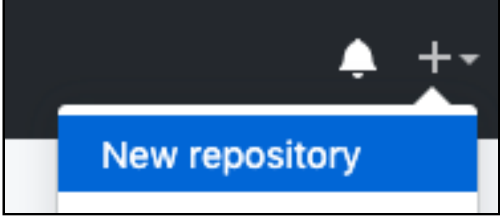
See? I *told* you it was of academic interest only.

Creating and syncing a remote

At the moment, you have your own repository on your local system. But that's a bit like practicing your guitar in your room your whole life and never jamming out at a party so you can wow your guests with a performance of "Wonderwall." You need to get this project out where others can see and potentially collaborate on it.

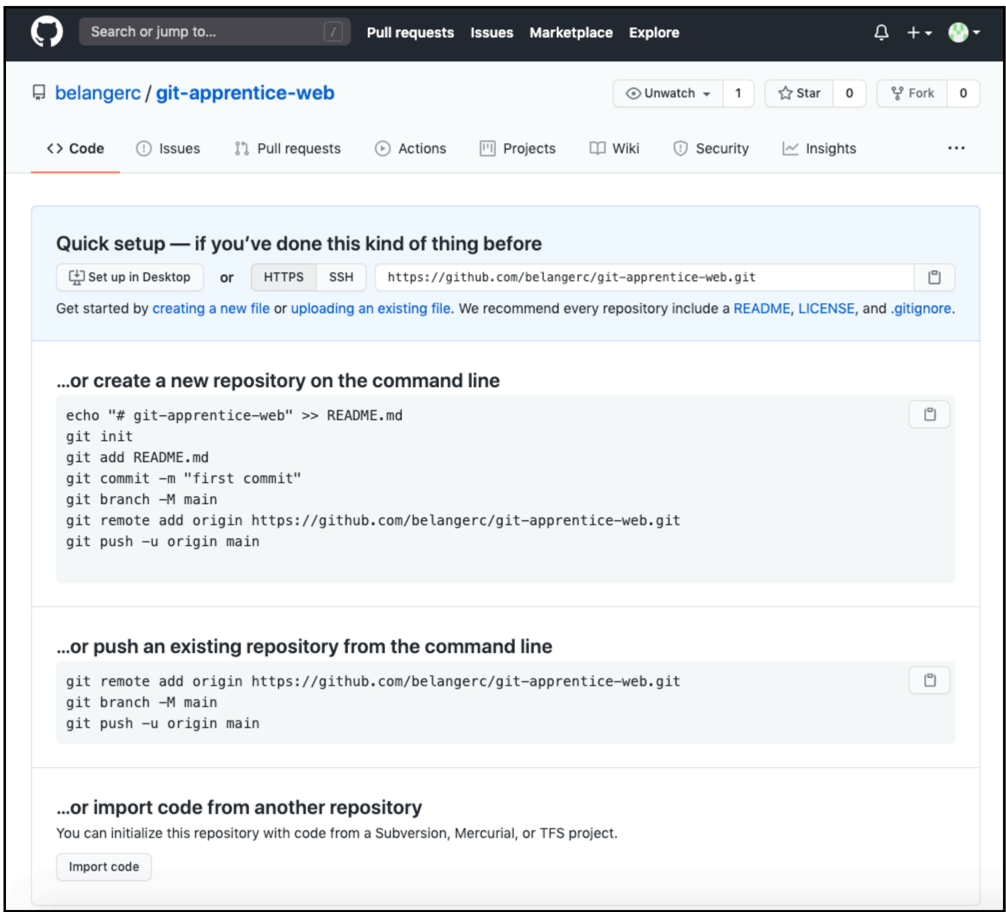
Head over to GitHub to create a new remote repository for your project, and log in to your account.

Click the + sign at the top right-hand corner of the screen, and select **New repository**.



A few details to follow, here:

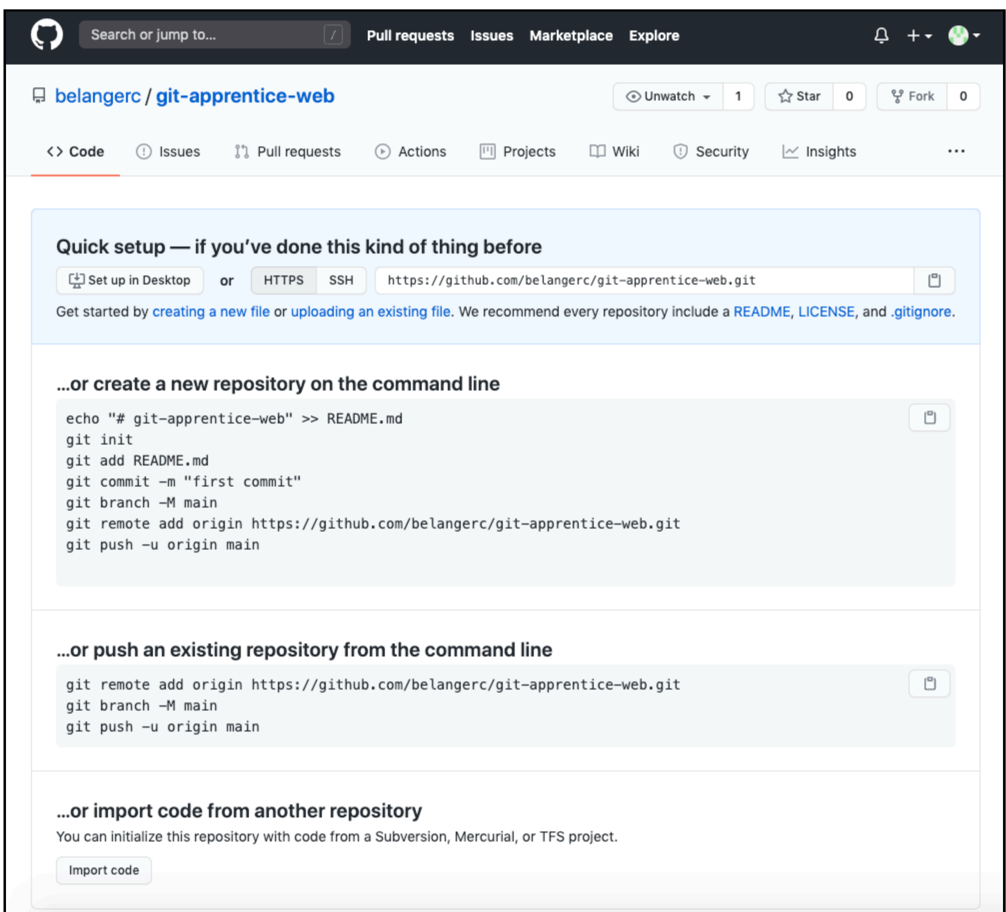
- Give your repository a good name; in this case, I'm going to use the same name as my project's directory name, `git-apprentice-web`, although this isn't strictly necessary.
- Leave the repository set to **Public**, so that anyone can see it.
- Finally, leave everything in the **Initialize this repository with:** section unchecked, since you will be importing the repository from your local workstation, which already exists and already has a LICENSE and a README.



This gives you several instructions on how to get some content into your repository. In your case, you already have an existing repository, so you can use the instructions under **...or push an existing repository from the command line**. Because you're all about that command line Git mastery, right?

Ensure the **HTTPS** option is selected in the top section of this page, next to the repository's URL. Copy the URL provided to your clipboard.

- Click the **Create repository** button and Git will shortly bring you to the **Quick setup** page.



Return to your terminal program, and execute the following to add a new remote to your local repository, substituting in the copied URL of your own repository where necessary:

```
git remote add origin https://github.com/<your-repo-name>/git-
apprentice-web.git
```

Git gives you no output from that command, but you can verify that you've added a remote, using the following command:

```
git remote -v
```

You should see your remote shown in the output:

```
origin  https://github.com/<your-username>/git-apprentice-
web.git (fetch)
origin  https://github.com/<your-username>/git-apprentice-
web.git (push)
```

So this is where many people get tripped up. As of late 2020, GitHub now uses `main` as the default branch name for all new repositories. But if you have a plain vanilla install of Git on your local workstation, you're likely configured with `master` as your default branch name. To check this, simply execute the following to see what `git init` set as your first branch name:

```
git branch
```

In my case, Git responds with the following:

```
* master
```

You have a disconnect here; your local workstation has `master` as the default branch, but your new GitHub repo is expecting `main`.

To fix this, execute the second command as prompted on the Quick setup page:

```
git branch -M main
```

Although Git gives you no output, this command changes the local name of your branch from `master` to `main`. Again, it pays to be paranoid with Git, so execute `git branch` again to confirm that your branch has been renamed to `main`.

OK - so your local repository is ready to be pushed to the remote. Now, execute the final command from the Quick setup page:

```
git push -u origin main
```

This pushes your changes, as you'd expect, with some corresponding output. The `-u` switch is the shorthand equivalent of `--push-upstream`, which ensures that every branch in your local repository tracks against the corresponding branch in the remote repository. Otherwise, Git won't automatically "know" to track your local branches against the remote ones.

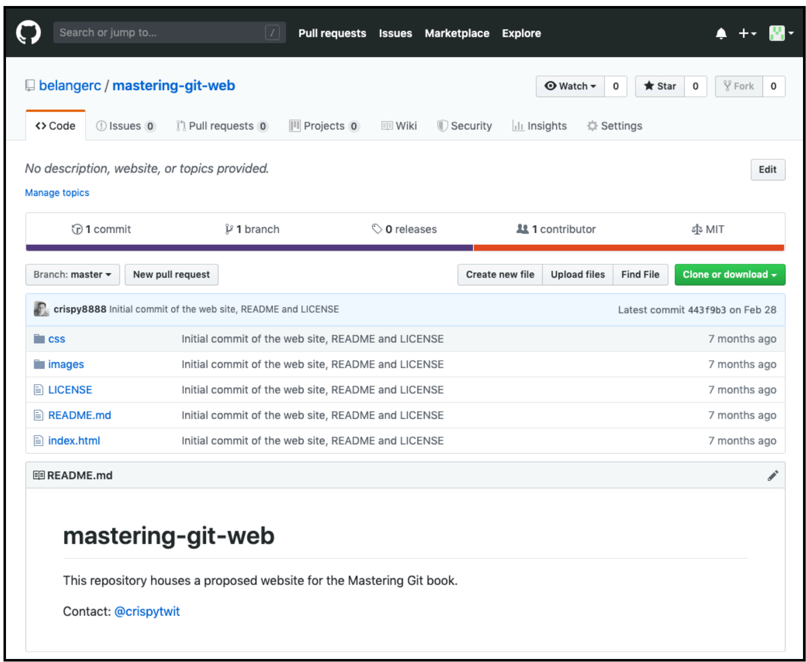
The `origin` option is simply the name of the remote to which you want to push; remember, `origin` is simply the conventional default name of the remote Git uses when it sets up your repository with `git init`, and not a standard.

`main` is the name of the local branch you want to push to your remote.

You can verify that Git has pushed and started tracking your local branch against the remote branch by looking at the final lines in the output from your `git push` command:

```
 * [new branch]      main -> main
Branch 'main' set up to track remote branch 'main' from
'origin'.
```

Head back to the homepage for your GitHub repository, and refresh the page to see your new repository there in all its glory:



At this point, your repository is ready for you, or anyone else, to view, clone, and contribute to.

Key points

- Use `git init` to set up a Git repository.
- It's accepted practice to have a **LICENSE** file and a **README.md** file in your repository.
- Use `git add` followed by `git commit` to create the first commit on your new repository.
- `create mode` is simply Git telling you what file permissions it's setting on the files added to the repository.
- You can create an empty remote on GitHub to host your repository, and you can choose to not have GitHub populate your remote with a LICENSE and README.md by default.
- Use `git remote add origin <remote-url>` to add a remote to your local repository.
- Use `git remote -v` to see the remotes associated with your local repository.
- If your Git installation uses `master` as the default branch in new repositories and you want to push to a newly created GitHub repository with `main` as the default branch, you'll need to execute `git branch -M main` to rename the local `master` branch to `main` to match your remote.
- Use `git push --set-upstream origin main` or `git push -u origin main` to push the local commits in your repository to your remote, and to start tracking your local branch against the remote branch.

Where to go from here?

You've come full circle with your introduction to Git! You started out with cloning someone else's repo, made a significant amount of changes to it, learned how to stage and commit your changes, how to view the log, how to branch, how to pull and push changes, and now you're back where you started, except that *you* are the creator of your very own repository. That feels good, doesn't it?

If you're an inquisitive sort, you probably have a lot of unanswered questions about Git, especially how it works under the hood, what merge conflicts are, how to deal with partially complete workfiles, and how to do things that you've heard about online, such as squashing commits, rewriting history, and using rebasing as an alternative to merging.

The next book in the Git Series is called Advanced Git (https://www.raywenderlich.com/books/advanced-git). That book takes you further under the hood of Git, shows you a little more about the internals of Git, and walks you through some scenarios that scare a lot of developers off of using Git in an advanced way. But you'll soon see that the elegance and relative simplicity of Git let you do some *amazing* things that can greatly improve the life of you and your distributed development team.

Conclusion

We hope this book has helped you get up to speed with Git! You know everything you need to know to effectively use Git on any sized project and team.

You're now ready to move on to advancing your Git skills! Check out our *Advanced Git* book in the Raywenderlich.com library at https://library.raywenderlich.com/.

If you have any questions or comments as you continue to use Git, please stop by our forums at http://forums.raywenderlich.com.

Thank you again for purchasing this book. Your continued support is what makes the tutorials, books, videos and other things we do at raywenderlich.com possible — we truly appreciate it!

– Chris, Jawwad, Bhagat, Cesare, Manda, Sandra and Aaron

The *Git Apprentice* team

Section II: Appendices

Appendix A: Installing & Configuring Git

Installing Git is relatively straightforward, but putting a little care in your initial setup and configuration will go a long way to ensuring that your work with Git is as hassle-free as possible.

Installing on Windows

To remain as platform-agnostic as possible, you'll install Git using one of the official standalone installers. While you can use the Chocolatey Package Manager for Windows, or even download and install GitHub Desktop (which installs Git on its own), you'll install and configure the plain-vanilla version of Git for Windows.

These instructions were tested on Windows 10, but the concepts should be similar across Windows versions.

1. Download the official release of Git for Windows at the following link:

- https://github.com/git-for-windows/git/releases/

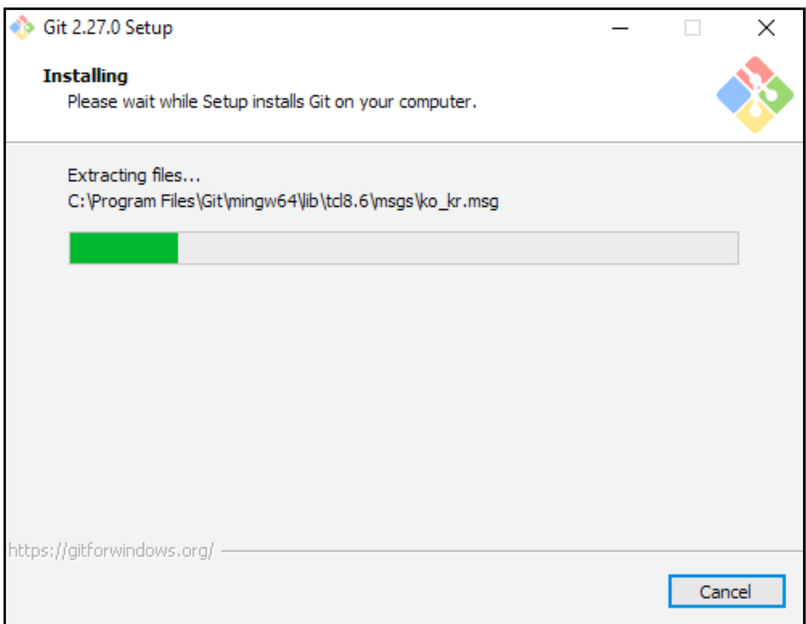
This book uses the 2.27.0 release, available here:

- https://github.com/git-for-windows/git/releases/download/v2.27.0.windows.1/Git-2.27.0-64-bit.exe

2. Execute the self-contained EXE file once it downloads fully.

3. If prompted to allow changes to your system, click **OK**.

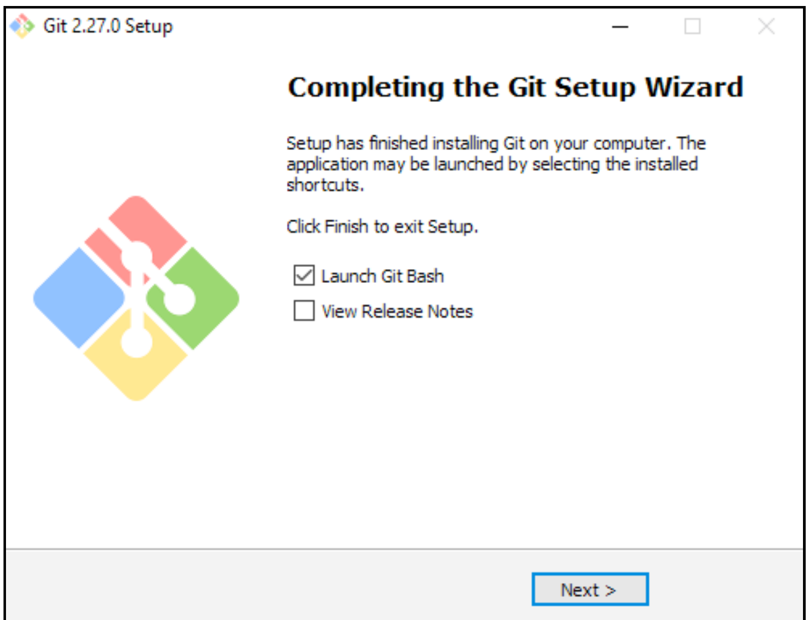
4. Click through the installer, accepting all defaults along the way. One thing you might want to change, depending on your system, is the install location of Git, which by default is **C:\Program Files\Git**. If you usually install everything into **C:\Program Files**, then you can leave this option alone.



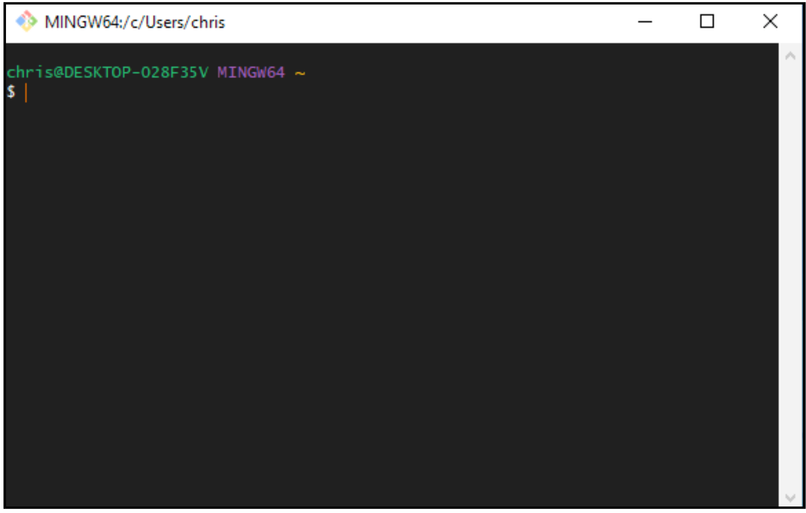
> **Note**: To best follow along with this book, leave the default editor option as **Vim**.
>
> Although you can select from a list of arguably excellent and more user-friendly editors as part of the setup process, you'll likely get lost when you try to use another text editor to create your commit messages or do other tasks. But if you feel compelled to choose another option, do so now. *Dulce periculum*. :]

5. The app will install Git and a host of helper libraries. This will only take a Microsoft minute.

6. When the installer finishes, you'll be presented with the completion dialog. Unselect **View Release Notes** (you have this book, so who needs release notes anyway?) and select **Launch Git Bash** so you can start the configuration process once the installer closes. Then click **Next**.



7. You'll see a console that looks similar to the following:



This is **Git Bash**. It's similar to the familiar Windows Command shell, but it's a version of the Bourne Again Shell (**bash**) that's a common way for people to interact with Linux, macOS and other platforms.

If you use the Git Bash shell to interact with your directories and projects, you'll be able to follow along with this book pretty much verbatim. This includes using the command line tools in this book, such as `nano`.

However, if you choose to use Git CMD (which lets you use the familiar Windows path structure, among other things), you'll have to adapt some of the commands and/or tools that you'll use in this book to their Windows equivalents.

Installing on macOS

There are a few ways to install Git on macOS. There *is* a standalone installer for Git, but it's unfortunately quite out of date and isn't recommended anymore. Installing GitHub Desktop will set up Git for you. However, the two recommended methods for maximum control are to either install with Xcode's command-line tools or use the Homebrew package manager to install Git on your system.

Installing Xcode's command-line tools

Chances are you're using Xcode if you're developing on a Mac. Since Xcode has some really good Git integration, you might as well just let Xcode do what it wants and manage the Git installation itself.

1. If you have Xcode installed on your Mac, simply execute the following command from Terminal to install the Xcode command-line tools:

```
xcode-select --install
```

You'll see a prompt to install the Xcode command-line tools, which include Git. Simply wait for the installer to run and finish.

2. Run the following command to verify which version of Git installed:

```
git --version
```

If Git responds with the current version or greater, which is 2.27.0 as of this writing, then you're good. If the version number is older, verify that you have the most recent version of Xcode installed.

Installing with Homebrew

Homebrew is a useful package manager for macOS. With Homebrew, you can install and update hundreds, if not thousands, of pieces of software right from your command line.

To install Homebrew, execute the following command in Terminal:

```
/bin/bash -c "$(curl -fsSL https://raw.githubusercontent.com/
Homebrew/install/master/install.sh)"
```

Sit back and wait while Homebrew installs itself. This might take a while.

Once Homebrew has finished installing, you can get down to installing Git.

1. Execute the following command in Terminal to install Git using Homebrew:

```
brew install git
```

Once again, sit back and wait for Homebrew to do its thing.

2. Once Homebrew has finished installing Git, you'll need to check that Homebrew has actually been able to overwrite any existing installations of Git on your machine.

To check which version of Git is being pointed to on your system, execute the following command in Terminal:

```
git --version
```

If Git responds with the current version, which is 2.27.0 as of this writing, then you're good.

If Git responds with an older version, then the symlinks that point to the Homebrew-installed copy of Git haven't been updated properly. Force Homebrew to rebuild those symlinks with the following command:

```
brew link --overwrite git
```

Execute `git --version` once more and Git should now report the correct version.

Configuring credentials

There are a few things you can do once you've installed Git to make your life a tiny bit easier; they're optional but highly recommended. One of those things is to set up your GitHub credentials in Git so they stick, saving you from having to re-enter them frequently.

Setting your username and email

1. To persist your GitHub username so you don't have to type it in every time you push your changes to a remote repository, execute the following command, enclosing your name in quotes:

```
git config --global user.name "your-username-here"
```

2. To persist the email you use for commits, which will appear alongside your commit history, enter the following command, enclosing your email in quotes:

```
git config --global user.email "youremail@domain.com"
```

Persisting your password

If you're on macOS, authenticating against GitHub or other repositories from the command line will store your password on the macOS Keychain, so you won't have to enter your credentials each time you want to interact with a remote repository.

If you're on Windows, you'll need to install the Git Credential Manager for Windows to get the same functionality. Find the instructions for installing that helper tool here:

- https://github.com/Microsoft/Git-Credential-Manager-for-Windows