Up to date for Git 2.21

Mastering Git

FIRST EDITION

Understanding Git Internals and Commands

By the raywenderlich Tutorial Team

Jawwad **Ahmad** & Chris **Belanger**

Mastering Git

Jawwad Ahmad and Chris Belanger

Notice of Rights

Notice of Liability

Trademarks

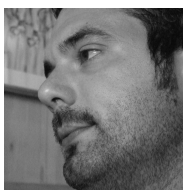
About the Author

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.

Jawwad Ahmad is an author of this book. He is an iOS Developer that spends way too much time using the power of Git to attempt to craft the most ideal commits. He currently works as a Software Engineer at a technology company in the San Francisco Bay Area.

About the Editors

Bhagat Singh is the 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. She's 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

Table of Contents: Extended

Book License

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

- You are allowed to use and/or modify the source code in *Mastering Git* 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 *Mastering Git* in as many apps as you want, but must include this attribution line somewhere inside your app: "Artwork/images/designs: from *Mastering Git*, available at www.raywenderlich.com".
- The source code included in *Mastering Git* is for your personal use only. You are NOT allowed to distribute or sell the source code in *Mastering Git* 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.

What You Need

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

- **Git 2.21.0 or later.** Git is the software package that you'll use for all of 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.

B Book Source Code & Forums

If you bought the digital edition

The digital edition of this book comes with the source code for the starter and completed projects for each chapter. These resources are included with the digital edition you downloaded from store.raywenderlich.com.

If you bought the print version

You can get the source code for the print edition of the book here:

https://store.raywenderlich.com/products/mastering-git-source-code

Forums

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

Digital book editions

We have a digital edition of this book available in both ePUB and PDF, which can be handy if you want a soft copy to take with you, or you want to quickly search for a specific term within the book.

Buying the digital edition version of the book also has a few extra benefits: free updates each time we update the book, access to older versions of the book, and you can download the digital editions from anywhere, at anytime.

Visit our book store page here:

- https://store.raywenderlich.com/products/mastering-git.

And if you purchased the print version of this book, you're eligible to upgrade to the digital editions at a significant discount! Simply email support@razeware.com with your receipt for the physical copy and we'll get you set up with the discounted digital edition version of the book.

Early Access Edition

You're reading an an early access edition of *Mastering Git*. This edition contains a sample of the chapters that will be contained in the final release.

We hope you enjoy the preview of this book, and that you'll come back to help us celebrate the full launch of *Mastering Git* early in 2020!

The best way to get update notifications is to sign up for our monthly newsletter. This includes a list of the tutorials that came out on raywenderlich.com that month, any important news like book updates or new books, and a list of our favorite development links for that month. You can sign up here:

- www.raywenderlich.com/newsletter

Section I: Beginning Git

This first 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.

Specifically, you'll cover:

1. **Crash Course in Git**: Learn how to get started with Git, the differences between platforms, and a quick overview of the typical Git workflow.
2. **Cloning a Repo**: It's quite common to start by creating a copy of somebody else's repository. Discover how to clone a remote repo to your local machine, and what constitutes "forking" a repository.
3. **Committing Your Changes**: A Git repo is made up of a sequence of commits—each representing the state of your code at a point in time. Discover how to create these commits to track the changes you make in your code.
4. **The Staging Area**: Before you can create a Git commit, you have to use the "add" command. What does it do? Discover how to use the staging area to great effect through the interactive git add command.
5. **Ignoring Files in Git**: Sometimes, there are things that you really don't want to store in your source code repository.
6. **Git Log & History**: There's very little point in creating a nice history of your source code if you can't explore it. You'll discover the versatility of the git log command—displaying branches, graphs and even filtering the history.
7. **Branching**: The real power in Git comes from its branching and merging model. This allows you to work on multiple things simultaneously. Discover how to manage branches, and exactly what they are in this chapter.

8. **Syncing with a Remote**: You've been working hard on your local copy of the Git repository, and now you want to share this with your friends. See how you can share through using remotes, and how you can use multiple remotes at the same time.
9. **Creating a Repository**: If you are starting a new project, and want to use Git for source control, you first need to create a new repository.
10. **Merging**: Branches in Git without merging would be like basketball without the hoop—fun, sure, but with very little point. In this chapter you'll learn how you can use merging to combine the work on multiple branches back into one.
11. **Stashes**: Git stashes offer a great way for you to create a temporary snapshot of what you're working on, without having to create a full-blown commit. Discover when that might be useful, and how to go about it.

Chapter 1: Crash Course in Git

This is an early access release of this book. Stay tuned for this chapter in a future release!

Chapter 2: Cloning a Repo

By Chris Belanger

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 choose to self-host repositories, and you'll learn about that later in this book. But, 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:

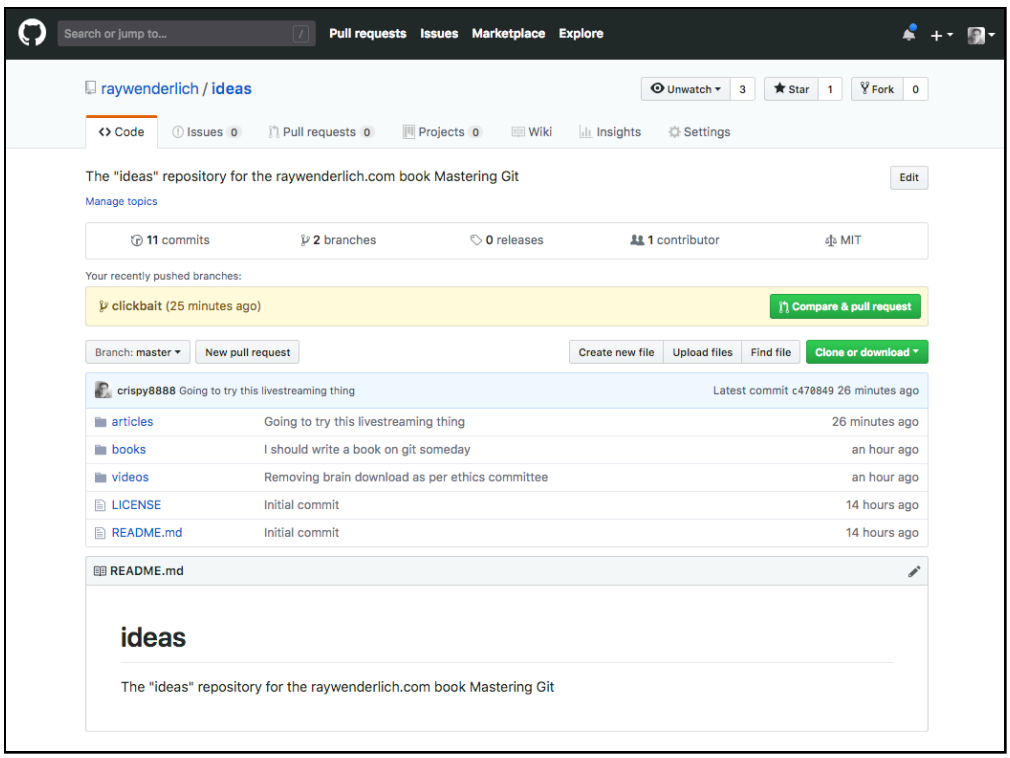
- A working installation of git on your local system.
- The remote URL of the repository you want to clone.
- Any credentials for the online host.

> **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.

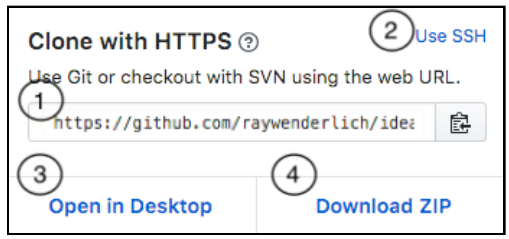
But where do you find the remote URL of the repository to clone it? Like many things in git (and with computers, in general), there are multiple ways to clone a repository. In this chapter, you'll use the easiest and most common cloning method, which starts on the GitHub repository homepage.

Finding the repository clone URL

Look for and click on the **Clone or download** 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. Leave this at (the rather unintuitive) **Use SSH** for now. You'll cover SSH later in this book.
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 **MasteringGit** where I would like to locally store all of the repos for this book.

Execute the following command to create the new directory:

```
mkdir MasteringGit
```

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 **MasteringGit** directory:

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

Execute the following command to navigate into the new directory:

```
cd MasteringGit
```

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 **MasteringGit** directory:

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

Use the `cd` command, followed by the `ls` command, to navigate into the new 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:

```
~/MasteringGit/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 — or a **fork** — 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 **MasteringGit** directory:

```
~/MasteringGit $
```

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:

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

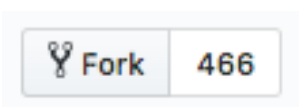
Looks good. You're ready to create a fork of the raywenderlich **ideas** repository... which leads you to your challenge for this chapter!

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 the **challenges** folder.

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

By Chris Belanger

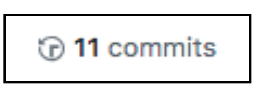
The previous chapter showed you how to clone remote repositories down to your local system. At this point, you're ready to start making changes to your repository. That's great!

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.

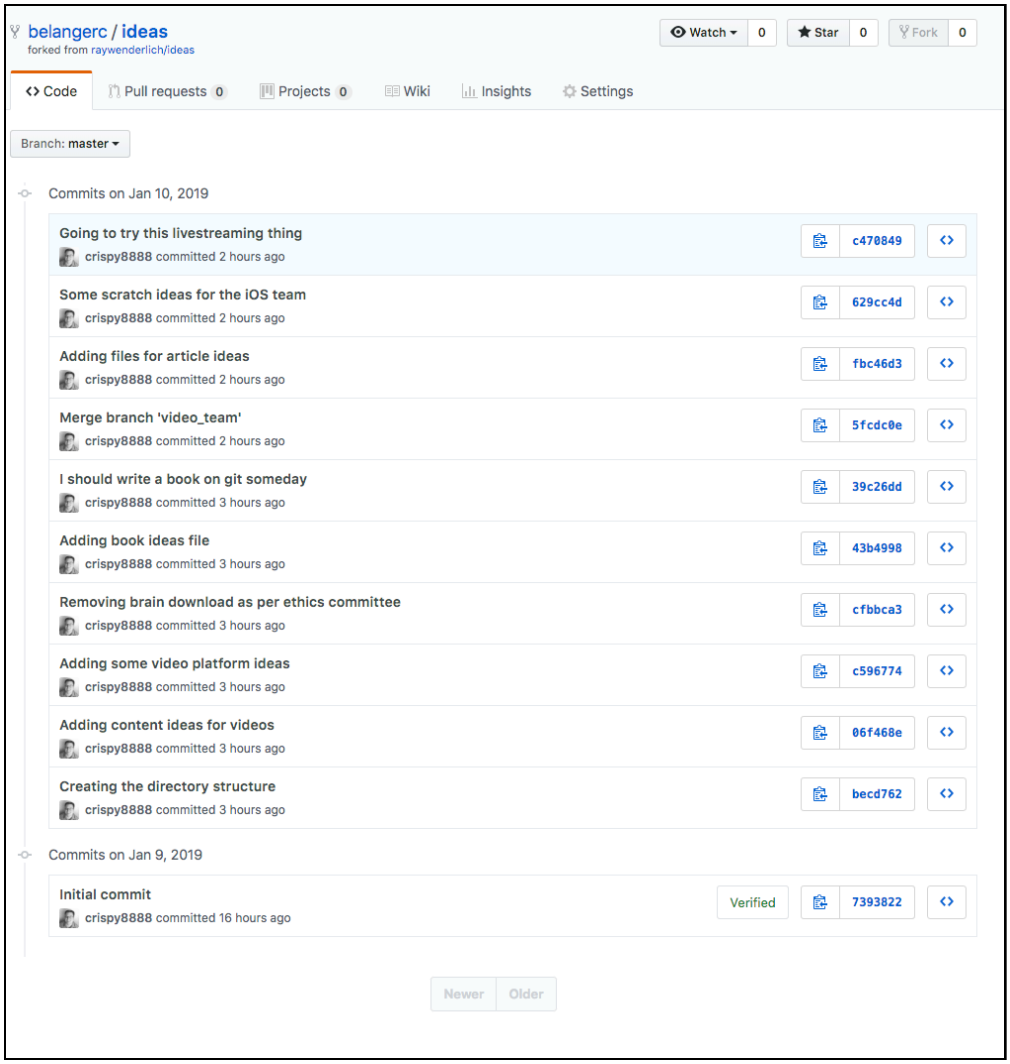
What is a commit?

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:



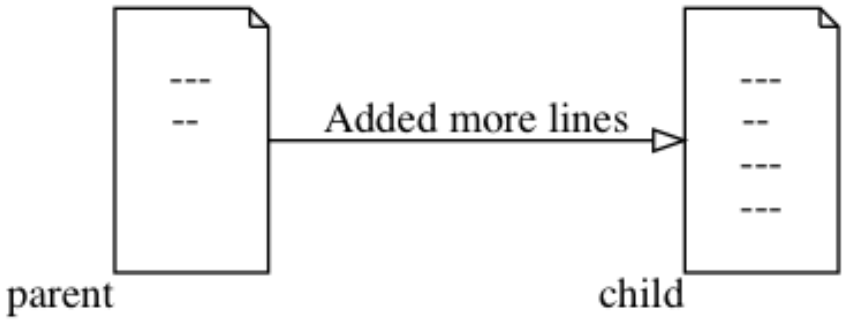
Click that link, and you'll see a bit of the history of this repository:



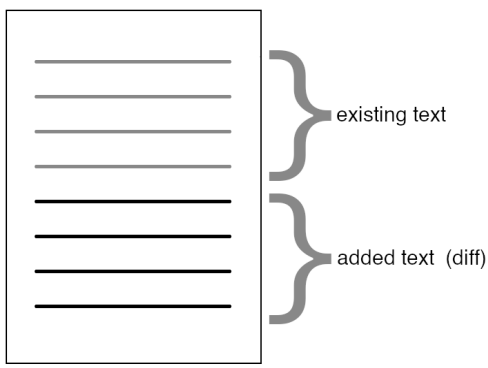
Each of those entries is a **commit**, which is essentially a snapshot of the particular state of the set of files in the repository at a point in time.

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:



In this example, you can see that the parent commit is X, and the child commit is Y. The diff between them are the changes I made to a single file:



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 inside of the **MasteringGit** 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:

```
~/MasteringGit/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:

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

Changes to be committed:
  (use "git reset HEAD <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:

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

Changes to be committed:
  (use "git reset HEAD <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 checkout -- <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:

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

Changes to be committed:
  (use "git reset HEAD <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:

```
~/MasteringGit/ideas $ git diff
~/MasteringGit/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:

```
~/MasteringGit/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
```

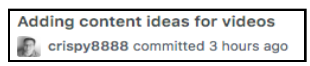
```
#
~
~
~
~
"~/MasteringGit/ideas/.git/COMMIT_EDITMSG" 10L, 272C
```

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:



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:

```
~/MasteringGit/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:

```
~/MasteringGit/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:

```
~/MasteringGit/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 files as string that point to entities 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 and 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:

```
~/MasteringGit/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:

```
~/MasteringGit/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/*
```

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:

```
~/MasteringGit/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 reset HEAD <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:

```
~/MasteringGit/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, in the advanced sections of this book, you'll investigate 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:

```
~/MasteringGit/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, "Viewing Git 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: 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, or want to check your solution, you can always find the answer to this challenge under the **challenges** folder for this chapter.

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

By Chris Belanger

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.

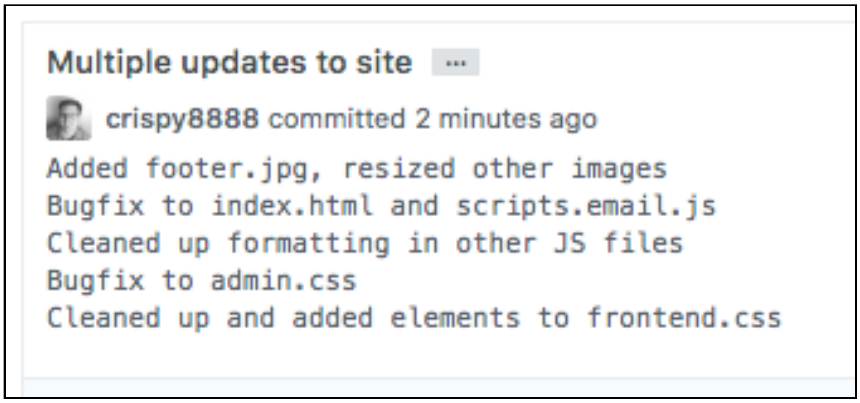
Git was built to compensate for this messy, non-linear approach to development. It's possible to work on *lots* of things at once, and selectively choose what you want to stage and commit to the repository. The general philosophy is that a commit should be a logical collection of changes *that make sense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

A simple staging example

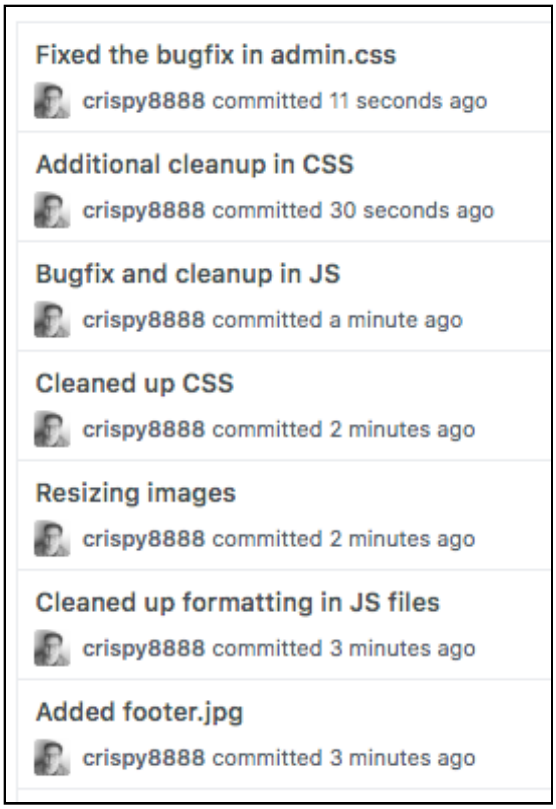
In the example below, I'm working on a website, and I want my design guru to review my CSS changes. I've changed the following files in the course of my work:

```
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:

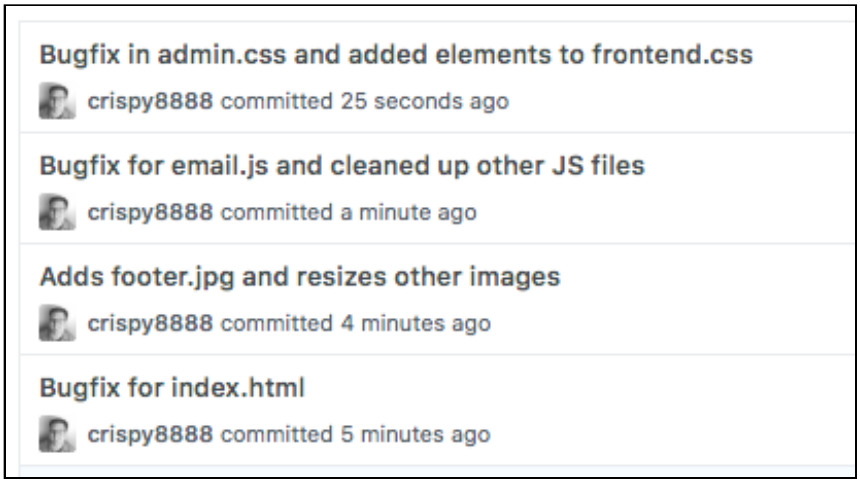


And if I committed each little change as I made it, my commit history might look like the following:



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:

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

Changes to be committed:
  (use "git reset HEAD <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, execute `git status` once again:

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

Changes to be committed:
  (use "git reset HEAD <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 checkout -- <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 reset HEAD <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 checkout -- <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 reset HEAD <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: 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 **challenges** 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

By Chris Belanger

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**:

```
*.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 that file 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?

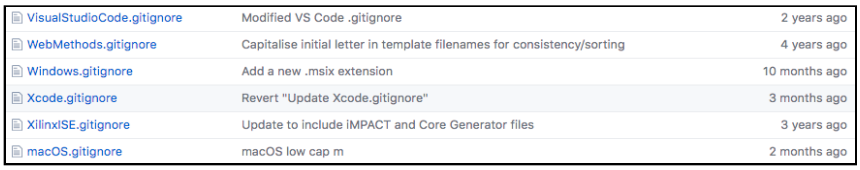
Finding sample .gitignore files

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: Populate your local .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 challenges 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 ~/.gitignore_global`.
- 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. You'll also learn about Git aliases, which will help you create some "shortcut" commands to make your life on the Git command line a whole lot easier!

Chapter 6: Git Log & History

By Chris Belanger

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.

Viewing Git history

Git keeps track of pretty much everything you do in your repository, and you've already seen this in action, in a brief manner, in previous chapters, through your use of the `git log` command. But there's 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

You can open up 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 the **ancestral commits** — that is, the set of commits that form the history of the current **head**, which in this case, is the most recent commit in the `master` branch of your repository.

Press **Q** to exit out of this view.

The basic `git log` command shows you *all* of 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`:

```
~/MasteringGit/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, and 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` that you've used before, shows the diffs of your commits. But there is 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 more clear 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:

```
~/MasteringGit/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.

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 is made up 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: the initial adding of the book ideas file. 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

By Chris Belanger

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` is nothing but a 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 *main* branch."

There's nothing special about `master`; again, Git simply knows that the `master` 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` 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 1: 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 challenges 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 main branch of a repository.
- 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: Syncing with a Remote

By Chris Belanger

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
```

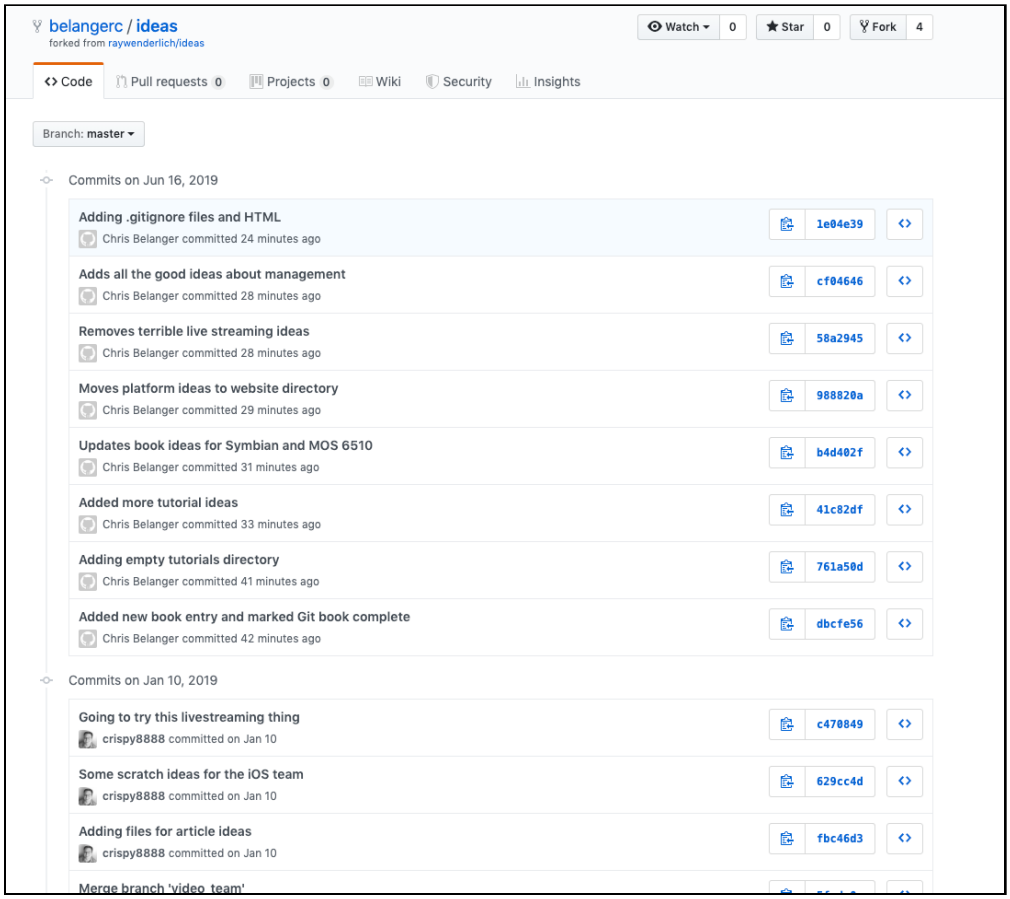
Git's given you a lot of output in this message, but essentially it's telling you some high-level information about what it's done, here: It's synchronized 12 changed items from your local repository on the remote repository.

> **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 **19 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.

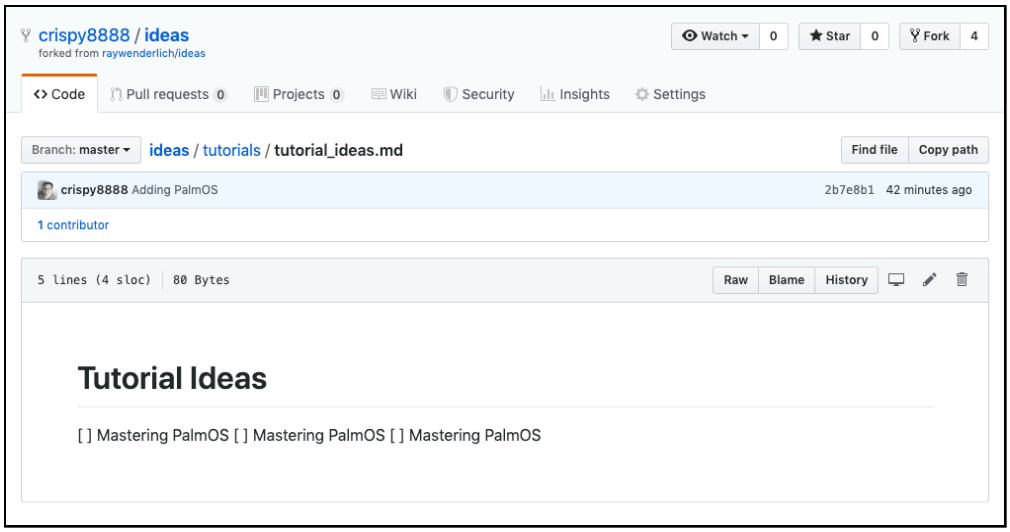
That operation is pretty straightforward when you're working by yourself on a project; you pull the latest changes from the repository, and, most likely, the remote will always be synchronized with your local, since there's no one else but you to make any changes.

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.

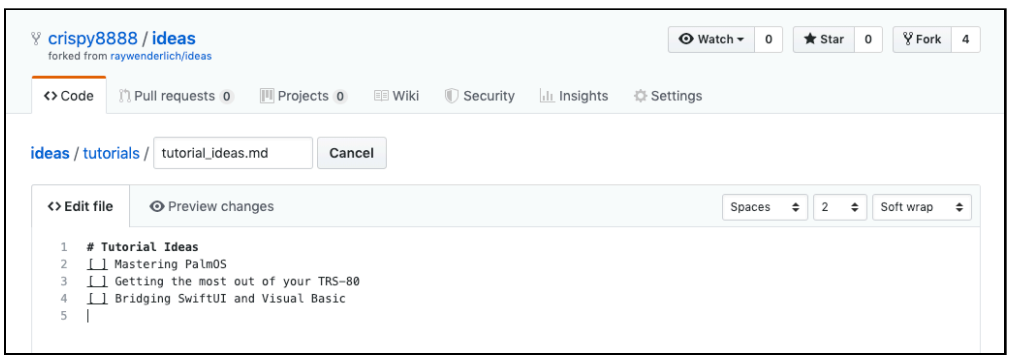
To illustrate how this works, and to illustrate what `git pull` actually does to your repository, you'll simulate a scenario wherein someone else has made a change to the `master` branch and pushed their changes before you had a chance to push yours. You'll see how Git responds to this scenario, and 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, 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 (non-fast-forward)
error: failed to push some refs to 'https://www.github.com/
belangerc/ideas'
hint: Updates were rejected because the tip of your current
branch is behind
hint: its remote counterpart. Integrate the remote changes (e.g.
hint: '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

# 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 --graph --oneline
```

You'll see something similar to the following:

```
*   a3ee3c2 (HEAD -> master) Merge branch 'master' of https://
github.com/belangerc/ideas
|\
| * 8909ec5 (origin/master, origin/HEAD) Added killer blockchain
idea
* | c7f4e7f Adding debugging book idea
|/
* 1e04e39 Adding .gitignore files and HTML
.
.
.
```

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 `1e04e39 Adding .gitignore files and HTML`. Then you have commit `c7f4e7f`, which is the commit you made on your local repository, followed by `8909ec5`, your remote commit on the GitHub repository page. And *also*, there's this `a3ee3c2 Merge branch 'master'` stuff at the top. And *also also*, Git shows your remote blockchain commit 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.

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:

```
* c7f4e7f (HEAD -> master) Adding debugging book idea
| * 8909ec5 (origin/master, origin/HEAD) Added killer blockchain
idea
|/
*   1e04e39 Adding .gitignore files and HTML
.
.
.
```

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
```

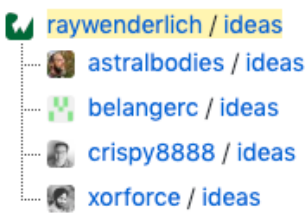
Head over to the main GitHub page for your repository, click on the **22 commits** link, and you'll see your changes up there on the remote.

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 --graph --oneline --all
```

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 --graph --oneline --all
```

At the top of the resulting graph, you'll see where this remote has diverged from the original:

```
* 9ff4582 (crispy8888/clickbait) Added another clickbait idea
* e69a76a (HEAD -> clickbait, origin/clickbait) Adding
suggestions from Mic
* 5096c54 Adding first batch of clickbait ideas
| *   a3ee3c2 (origin/master, origin/HEAD, master) Merge branch
'master' of https://github.com/belangerc/ideas
| |\
| | * 8909ec5 Added killer blockchain idea
| * | c7f4e7f Adding debugging book idea
| |/
| * 1e04e39 Adding .gitignore files and HTML
| * cf04646 Adds all the good ideas about management
| * 58a2945 Removes terrible live streaming ideas
| * 988820a Moves platform ideas to website directory
| * b4d402f Updates book ideas for Symbian and MOS 6510
| * 41c82df Added more tutorial ideas
| * 761a50d Adding empty tutorials directory
| * dbcfe56 Added new book entry and marked Git book complete
| * c470849 (crispy8888/master) Going to try this livestreaming
thing
| * 629cc4d Some scratch ideas for the iOS team
|/
* fbc46d3 Adding files for article ideas
```

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 --graph --oneline`:

```
* 9ff4582 (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:

```
*   58b5b43 (HEAD -> master) Merge branch 'clickbait'
|\
| * 9ff4582 (crispy8888/clickbait, clickbait) Added another
clickbait idea
| * e69a76a (origin/clickbait) Adding suggestions from Mic
| * 5096c54 Adding first batch of clickbait ideas
* |   a3ee3c2 (origin/master, origin/HEAD) Merge branch 'master'
of https://github.com/belangerc/ideas
|\ \
| * | 8909ec5 Added killer blockchain idea
* | | c7f4e7f 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. Later in the book, you'll see a few nicer alternatives to the Git command line graph analysis, 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 9: Creating a Repository

By Chris Belanger

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 **mastering-git-web** directory from the **starter** folder into your main **MasteringGit** folder.

Now, open up your terminal program and navigate into the **mastering-git-web** directory. If you've been following along with the book so far, you're likely still in the **MasteringGit/ideas** folder, so execute the following command to get into the **mastering-git-web** subdirectory:

```
cd ../mastering-git-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/
MasteringGit/mastering-git-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.

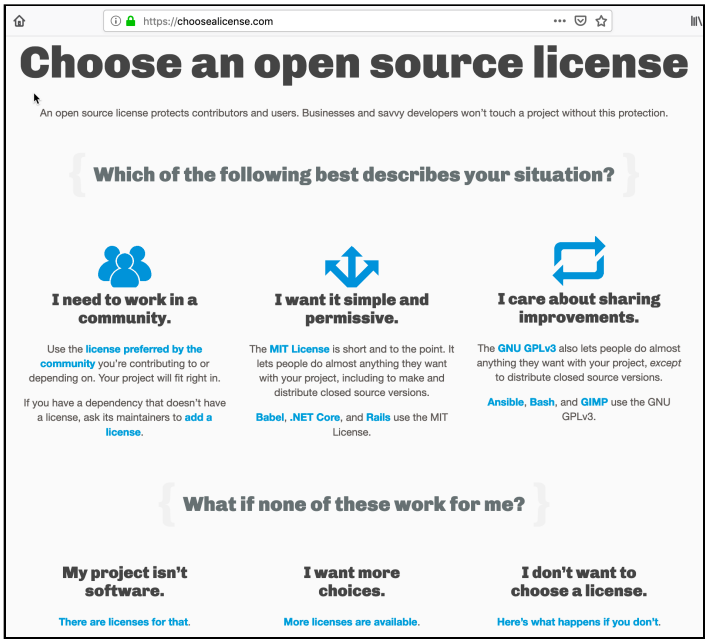
Having a license outlines how others may contribute to your project and what their rights are. The interesting bit comes in when you *don't* include a license to your work. If you create a project and stick it up on GitHub, without a license, you're stating that *no one* has the license to use your code 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.

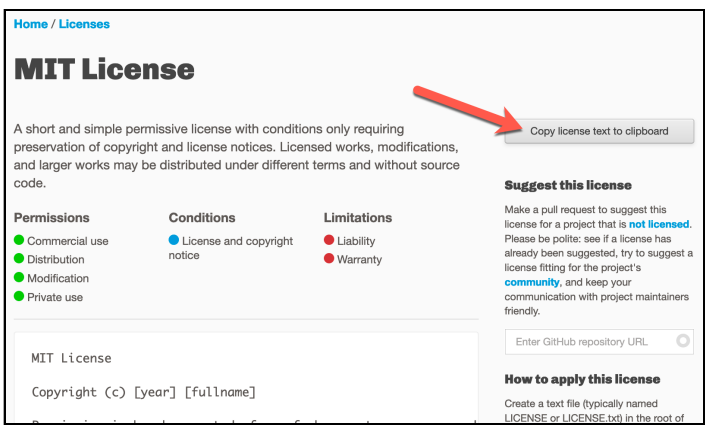
True, having a license included with your project won't protect you from code burglars who just want to take your work and use it without your permission. But what it *does* do is indicate the terms of use and reuse of your project to anyone who wants to collaborate in a fair manner, or use your work 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.

In my case, I used `nano` to create the file:

```
nano LICENSE
```

Then, I pasted in the text I copied from https://choosealicense.com/, updated `[YEAR]` with the current year, updated `[fullname]` with the name of my organization, and saved my changes.

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):

```
# mastering-git-web

This is the main website for the Mastering Git 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:

```
~MasteringGit/mastering-git-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.

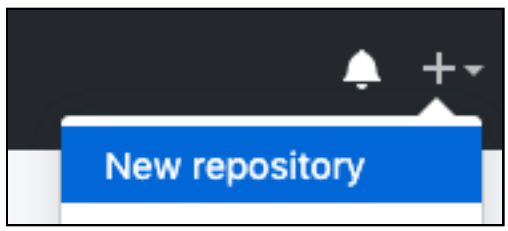
See? I *told* you it was of academic interest only.

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.

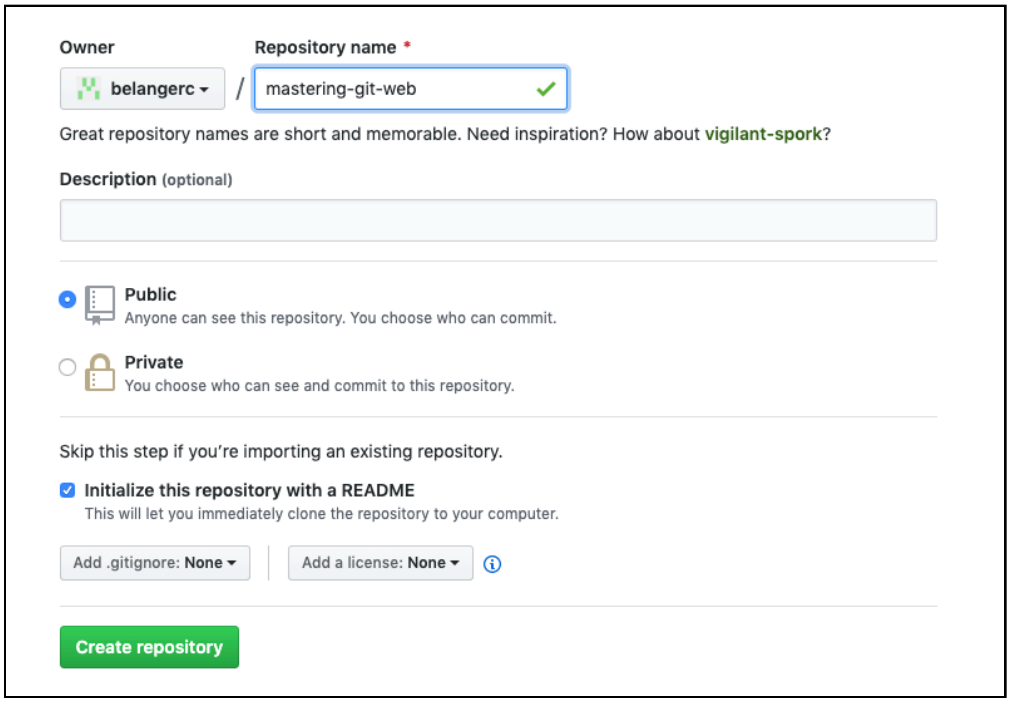
Click the + sign at the top right-hand corner of the screen, and select **New repository**.



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, `mastering-git-web`, although this isn't strictly necessary.
- Leave the repository set to **Public**, so that anyone can see it.
- Finally, leave the **Initialize this repository with a README** unchecked, since your local repository already exists and already has a README.
- Leave **Add .gitignore** and **Add a license** to their default **None** settings, since you don't need those either, and they can be added or changed later on.

- Click the **Create repository** button and Git will shortly bring you to the **Quick setup** page.



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.

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/belangerc/mastering-
git-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/belangerc/mastering-git-web.git
(fetch)
origin  https://github.com/belangerc/mastering-git-web.git
(push)
```

Now, you simply need to push the commits on your local repository to your remote. Do that with the following command:

```
git push --set-upstream origin master
```

This pushes your changes, as you'd expect. The `--set-upstream` 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.

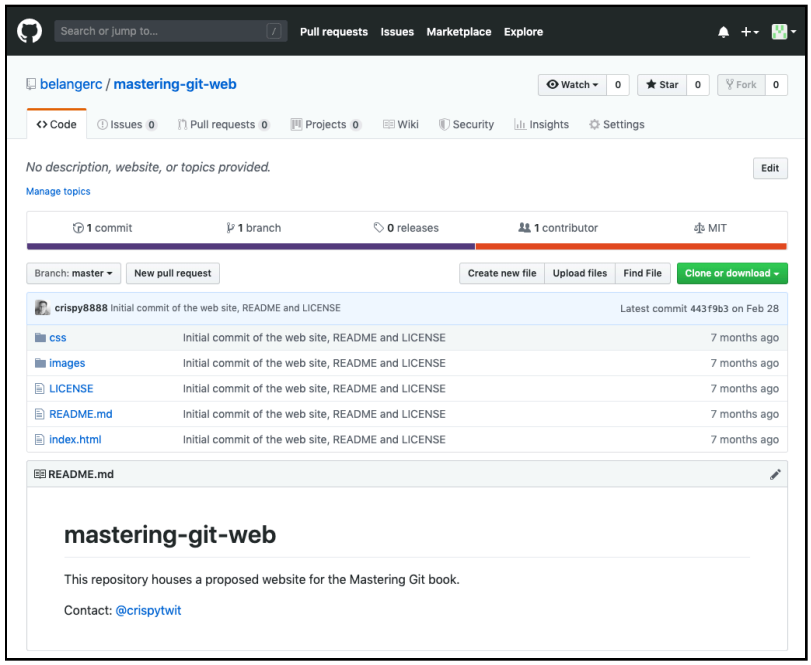
`master` is the name of the local branch you want to push to your remote. Again, Git assumes the default name of `master` for the first branch of your repository.

> **Note**: You can also use the shorter `git push -u origin master` to accomplish the same thing. `-u` and `--set-upstream` are aliases.

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]      master -> master
Branch 'master' set up to track remote branch 'master' from
'origin'.
```

Head back to the homepage for your GitHub repo, and refresh the page to see your new repo 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.
- Use `git push --set-upstream origin master` or `git push -u origin master` 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 merge, 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, though, 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, cherry-picking commits, rewriting history, and using rebasing as an alternative to merging.

The next section of this 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.

Chapter 10: Merging

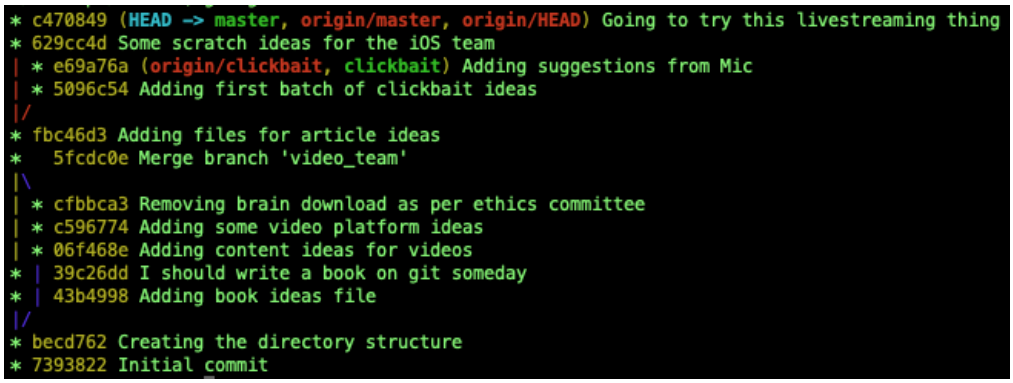
By Chris Belanger

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.

A look at your branches

If you were to visualize the branching history of your current **ideas** repository, it would look something like this:



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 most recent branch you switched to (in the previous chapter), 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.

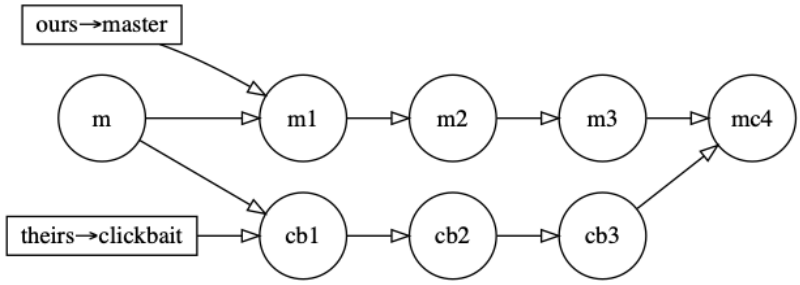
This is a fairly common development workflow; in a small team, `master` can effectively serve as the main development line, and developers make branches off of `master` to work on features or bug fixes, without messing with what's in the main development line. Many teams consider `master` to represent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

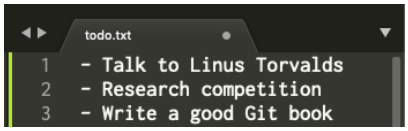
Let's say you want to merge the `clickbait` branch back into `master`. In this case, as shown in the diagram below, `master` is **ours** and the `clickbait` branch would be **theirs**. Keeping this distinction straight will help you immeasurably in your merging career.



Three-way merges

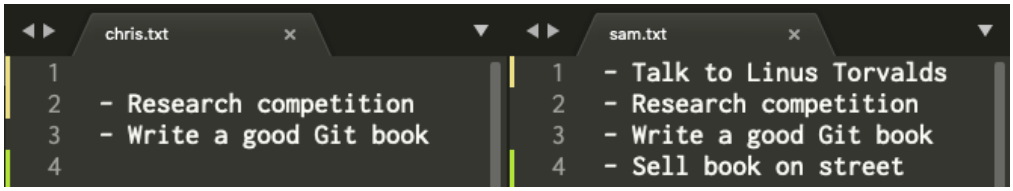
You might think that merging is really just taking two revisions, one on each branch, and mashing them together in a logical manner. This would be a **two-way** merge, and it's the way most of us think about the world: a new element formed by two existing elements is simply the union of the unique and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



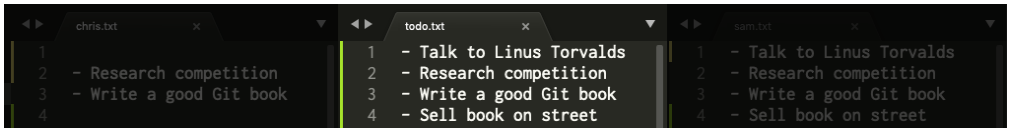
The original file.

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.

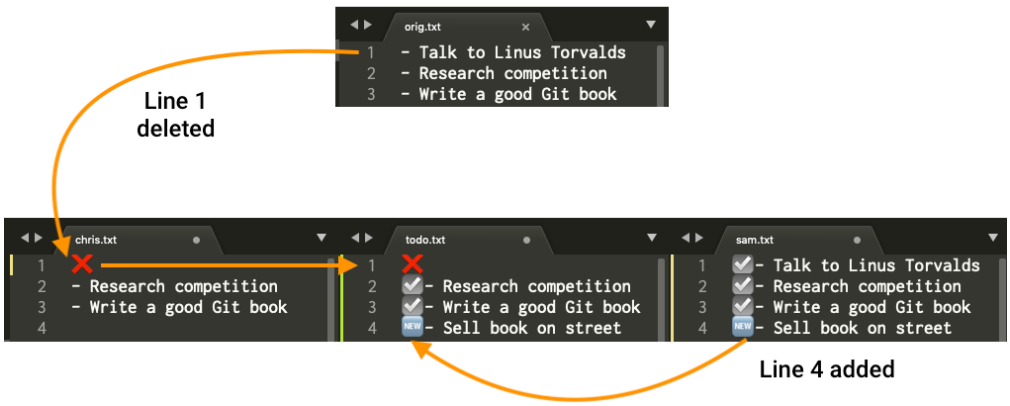


With no background of what the starting point was, the person responsible to merge tries to preserve as many lines as possible in common to both files.

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.

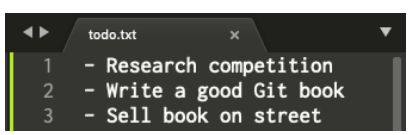
To perform an educated merge of these two files, your impartial third party has to know about the **common ancestor** of both of these files. This common ancestor is the third revision that comes in to play with a three-way merge.

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:

```
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, where is that **articles/clickbait_ideas.md** you looked at in the other branch? Execute that same command, again:

```
~/MasteringGit/ideas $ cat articles/clickbait_ideas.md
cat: articles/clickbait_ideas.md: No such file or directory
```

It's not there. That makes sense, since you haven't yet merged that file into the `master` branch, so it's not going to be there when you switch back to `master`.

You're now back on the `master` branch, ready to pull in the changes from the `clickbait` branch. But just 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'`. As nice as that is, you'll probably want a bit more detail in there. But that's OK - you know what you're doing by now, don't you? If not, here is a quick cheatsheet for you:

- Press **I** to enter Insert mode.
- Cursor down to the line below the provided merge message.
- Press Enter to create a blank line.
- Add some details to your commit message. I suggest "These are some clickbait ideas… whether anyone wants them or not."
- Press Escape to exit out of Insert mode.
- 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:

```
*   55fb2dc (HEAD -> master) Merge branch 'clickbait'
|\
| * 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:

```
~/MasteringGit/ideas $ cat articles/clickbait_ideas.md
```

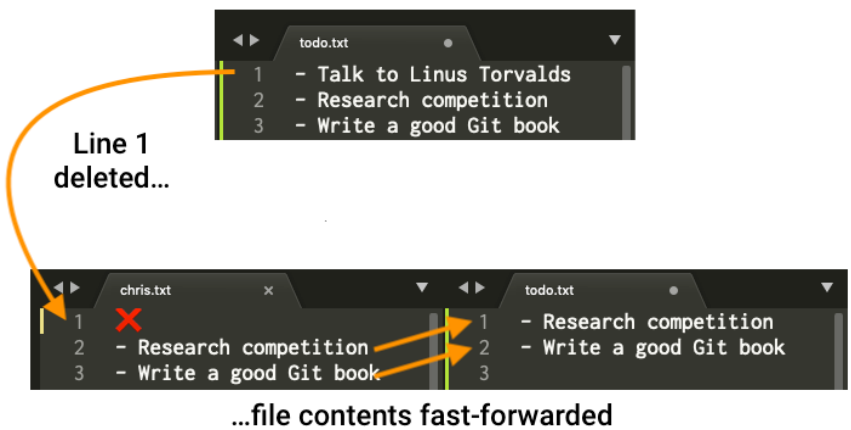
You'll see the contents of the file spat out to the console.

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 --graph --oneline --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'
```

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:

```
~/MasteringGit/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 --graph --oneline --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'
```

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 1: 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 challenges 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?

To this point, you’ve been doing pretty much everything on your local repository. But you want the rest of your team to see the amazing work you’ve been doing, don’t you? In the next chapter, you’ll learn how to synchronize your repository with a remote repository with `git push`, `git pull`, `git remote` and much more.

Chapter 11: Stashes

This is an early access release of this book. Stay tuned for this chapter in a future release!

Section II: Advanced Git

This section dives deeper into the inner workings of Git, what particular Git operations actually do, and will walk you through some interesting problem-solving scenarios when Git gets cranky. You'll build up some mental models to understand what's going on when Git complains about things to help you solve similar issues on your own in the future.

Specifically, you'll cover:

12. **How Does Git Actually Work?**: If you've been using Git for a while, you might be wondering how it actually works. Discover how Git is built on top of a simple key-value store-based file system, and what power this provides to you.

13. **Merge Conflicts**: Merging isn't always as simple as it might first appear. In this chapter you will learn how to handle merge conflicts — which occur when Git cannot work out how to automatically combine changes.

14. **Demystifying Rebasing**: Rebasing is poorly understood, although it can be an incredibly powerful tool. In this chapter, we'll cover what happens behind the scenes when you rebase and set you up for some useful applications of rebasing in the coming chapters.

15. **Rebasing to Rewrite History**: Rebase is a whole lot more powerful than just as a replacement for merge. It offers the ability to completely rewrite the history of your git repo.

16. **Gitignore After the Fact**: Gitignore is easy right? If you've been using it for a while you'll know that isn't always true. Discover how you can fix problems with gitignore such as handling files that have been accidentally committed to the repository.

17. **Cherry Picking**: Cherry picking provides a way to grab single commits from other branches, and apply them to your own branch.

18. **Many Faces of Undo**: One of the common questions associated with git is "how can I get out of this mess?" In this chapter you'll learn about the different "undo"

commands that git provides — what they are and when to use them.

Chapter 12: How Does Git Actually Work?

By Chris Belanger

Git is one of those wonderful, elegant tools that does an amazing job of abstracting the underlying mechanism from the front-end workings. To pull changes from the remote down to the local, you execute `git pull`. To commit your changes in your local repository, you execute `git commit`. To push commits from your local repository to the remote repository, you execute `git push`. The front end does an excellent job of mirroring the mental model of what's happening to your code.

But as you would expect, a lot is going on underneath. The nice thing about Git is that you could spend your entire career not knowing how the Git internals work, and you'd get along quite well. But being aware of how Git manages your repository will help cement that mental model and give a little more insight into why Git does what it does.

Everything is a hash

Well, not *everything* is a hash, to be honest. But it's a useful point to start when you want to know how Git works.

Git refers to all commits by their SHA-1 hashes. You've seen that many times over, both in this book and in your personal and professional work with Git. The hash is the key that points to a particular commit in the repository, and it's pretty clear to see that it's just a type of unique ID. One ID references one commit. There's no ambiguity there.

But if you dig down a little bit, the commit hash doesn't reference *everything* that has to do with a commit. In fact, a lot of what Git does is create references to references in a tree-like structure to store and retrieve your data, and its metadata, as quickly and efficiently as possible.

To see this in action, you'll dissect the "secret" files underneath the **.git** directory and see what's inside of each.

Dissecting the commit

Since the atomic particle of Git workflow is the commit, it makes sense to start there. You'll start walking down the tree to see how Git stores and tracks your work.

> **Note**: The commit hashes I'll use will be different than the ones in your repository. Simply follow the steps below, substituting in your hashes for the ones I have in my repository.

I'm going to pick one of my most recent commits that has a change that I made, as opposed to a merge, just to narrow down the set of changes I want to look at.

To get the list of the most recent five commits, execute the `git log` command as below:

```
git log -5 --oneline
```

My log result looks like the following:

```
f8098fa (HEAD -> master, origin/master, origin/HEAD) Merge
branch 'clickbait' with changes from crispy8888/clickbait
d83ab2b (crispy8888/clickbait, clickbait) Ticked off the last
item added
5415c13 More clickbait ideas
fed347d (from-crispy8888) Merge branch 'master' of https://
www.github.com/belangerc/ideas
ace7251 Adding debugging book idea
```

I'll select the commit with the short hash `d83ab2b` to start stepping through the tree structure. First, though, you'll need to get the long hash for this, instead of the short one. You'll see why this is in a moment.

You *could* simply run `git log` again without the `--oneline` option to get the long hash, but there's an easier way.

Converting short hash into long

Execute the command below to convert a short hash into its long equivalent:

```
git rev-parse d83ab2b
```

Git responds with the long hash equivalent:
`d83ab2b104e4addd03947ed3b1ca57b2e68dfc85`.

Now, you need to start crawling through the Git tree to find out what this commit *looks* like on disk.

The inner workings of Git

Change to your terminal program and navigate to the main directory of your repository. Once you're there, navigate into the **.git** directory of your repository:

```
cd .git
```

Now, pull up a directory listing of what's in the **.git** directory, and have a look at the directories there. You should, at a minimum, see the following directories:

```
info/
objects/
hooks/
logs/
refs/
```

The directory you're interested in is the **objects** directory. In Git, the most common objects are:

- **Commits**: Structures that hold metadata about your commit, as well as the pointers to the parent commit and the files underneath.
- **Trees**: Tree structures of all the files contained in a commit.
- **Blobs**: Compressed collections of files in the tree.

Start by navigating into the **objects** directory:

```
cd objects
```

Pull up a directory listing to see what's inside, and you'll be greeted with the following puzzling list of directories:

```
02    14    39    55    6e    84    ad    c5    db    f8
05    19    3a    56    72    88    b4    c8    e0    f9
06    1a    3b    57    73    8b    b5    ca    e6    fb
0a    1c    3d    59    75    99    b8    ce    e7    fe
0b    24    3e    5d    76    9d    b9    cf    eb    ff
0c    29    43    5f    78    9f    ba    d2    ec    info
0d    2c    45    62    7a    a0    bb    d3    ed    pack
0e    33    47    65    7d    a1    be    d7    ee
0f    35    4e    67    7f    a4    bf    d8    f1
11    36    50    69    81    ab    c0    d9    f4
12    37    54    6c    83    ac    c4    da    f5
```

It's clear that this is a lookup system of some sort, but what does that two-character directory name mean?

The Git object repository structure

When Git stores objects, instead of dumping them all into a single directory, which would get unwieldy in rather short order, it structures them neatly into a tree. Git takes the first two characters of your object's hash, uses that as the directory name, and then uses the remaining 38 characters as the object identifier.

Here's an example of the Git **object** directory structure, from my repository, that shows this hierarchy:

```
objects
├── 02
│   ├── 1f10a861cb8a8b904aac751226c67e42fadbf5
│   └── 8f2d5e0a0f99902638039794149dfa0126bede
├── 05
│   └── 66b505b18787bbc710aeef2c8981b0e13810f9
├── 06
│   └── f468e662b25687de078df86cbc9b67654d938b
├── 0a
│   └── 795bccdec0f85ebd9411e176a90b1b4dfe2002
├── 0b
│   └── 2d0890591a57393dc40e2155bff8901acafbb6
├── 0c
│   └── 66fedfeb176b467885ccd1a1ec70849299eeac
├── 0d
│   └── dfac290832b19d1cf78284226179a596bf5825
├── 0e
│   └── 066e61ce93bf5dfaa9a6eba812aa62038d7875
├── 0f
```

```
│   └── a80ee6442e459c501c6da30bf99a07c0f5624e
├── 11
│   ├── 06774ed5ad653594a848631f1f2786a76a776f
│   ├── 92339da7c0831ba4448cb46d40e1b8c2bed12c
│   └── c1a7373df5a0fbea20fa8611f41b4a032b846f
.
.
.
```

To find the object associated with a commit, simply take the commit hash you found above:

```
d83ab2b104e4addd03947ed3b1ca57b2e68dfc85
```

Decompose that into a directory name and an object identifier:

- **Directory**: `d8`
- **Object identifier**: `3ab2b104e4addd03947ed3b1ca57b2e68dfc85`

Now you know that the object you want to look at is inside the **d8** directory. Navigate into that directory and pull up another listing to see the files inside:

```
.
.
.
d7
├── c33fdd7d35372cba78386dfe5928f1ba8dfb70
└── e92f9daeec6cd217fda01c6b726cb07866728c
d8
└── 3ab2b104e4addd03947ed3b1ca57b2e68dfc85
d9
└── 809bc1dafdec03f0d60f41f6c7f6cfc3228c80
da
├── 967ae1f60e59d2a223e37301f63050dca0cf6f
└── fe823560ecc5694151c37187f978b5cf3d5cf1
.
.
```

In my case, I only see one file: **3ab2b104e4addd03947ed3b1ca57b2e68dfc85**. You may see other files in there, and that's to be expected in a moderately busy repository.

You can't take a look at this object directly, though, as objects in Git are compressed. If you tried to look at it using `cat 3ab2b104e4addd03947ed3b1ca57b2e68dfc85` or similar, you'll probably see a pile of gibberish like so, along with a few chirps from your computer as it tries to read control characters from the binary object:

```
xu?Ko?0???51??Л
yB
    ??f?y?cBшo?{ɛ?|ЬFL?:?@??_?0Td5?D2Br?D$??f?B??b?5W?HÁ?H*?&??
(fbd

dC!DV%?????D@?(???u0??8{?w????0?IULC1????@(<?s '
mO????????ƶe?S????>?K8                    89_vxm(#?jxOs?u?b?5m????
=w\l?
%?O??[V?t]?^??????G6.n?Mu?%
                              ??X??X̌v??x?EX???:sys???G2?y??={X?ᑎe?
X?4u???????4o'G??^"qِ???$?Ccu?ml???vB_)?I?6?$?(?E9?z??nUmV?Em]?
p??3?`??????q?Ţqjw????VR?O? q?.r???e|lN?p??Gq?)?????#???85V?
W6?????
)|Wc*??8?1a?b?=?f*??pSvx3??;??3??^??O?S}??Z4?/?%J?
                                                    F?of??Oی*??`
```

Viewing Git objects

Git provides a way to look at the contents of a compressed Git object: `git cat-file`. This decompresses the object and writes it out to your console in a human-readable form. You can simply pass it a short or long hash, and Git will write out the contents of that object in a human-readable form.

So take a look at the uncompressed form of the object file with the following command, substituting in the short or long hash from the commit that you want to look at:

```
git cat-file -p d83ab2b
```

The `-p` option here tells Git to figure out what type of object it's dealing with and to provide appropriately formatted output.

The commit object

In my case, Git tells me the details about my chosen commit object:

```
tree c0425d3b2aa2bfbbc0a08efda69ed00286dec6e4
parent 5415c13d2449f9719a8a8e84ee25105a1a587c5f
author cripsy8888 <chris@razeware.com> 1549849076 -0400
committer GitHub <noreply@github.com> 1549849076 -0400
gpgsig -----BEGIN PGP SIGNATURE-----

 wsBcBAABCAAQBQJcYNH0CRBK7hj4Ov3rIwAAdHIIABLgrn6UmK0fzh/
jqaIg7ax2

kie1Grd4EqLA+kuNT0jR+qTbc6x+OwlYt2PWZXOzfyOwY3UNKByHWhJDrhgzjLjB
 65CT7GGmMOKlGi7gis3W6jZetka+Lnauoeg9e/VnAu6q/
9JOv6ZyRN4j13wYpnK1

9wyooTbV2ipKMRFBs56DjL+6LkJcuIdD98rqluUzugGIvjFnGmIUCKF485l1bN3Q
 eZ+PsFGeqqIFHdWnXOyvBhzjVogoumR8K7WtQ8tGMXnAnwlBo0s+sikJa4tTmO/
o

feVt0ln+frS+j6zhnC1RHRPkucPDBV9DuVdrSiA4w1xmXCXmVZ26bCEHQkaf1Z0=
 =QrF9
 -----END PGP SIGNATURE-----


Ticked off last item added

No one would believe you could skew election results...
```

There's a wealth of information here, but what you're interested in is the **`tree`** hash.

The tree object

The tree object is a pointer to another object that holds the collection of files for this commit.

So execute `git cat-file` again to see what's inside *that* object, substituting your particular hash:

```
git cat-file -p c0425d3b2aa2bfbbc0a08efda69ed00286dec6e4
```

I get the following information about the tree object:

```
100644 blob 8b23445f4a55ae5f9e38055dec94b27ef2b14150    LICENSE
100644 blob f5c651739ff232f6226d686724f3c9618dd9f840
README.md
040000 tree d27f2eb006fff5b83fdc5d6639c7cfabdcf9fc37    articles
040000 tree 0b2d0890591a57393dc40e2155bff8901acafbb6    books
040000 tree 028f2d5e0a0f99902638039794149dfa0126bede    videos
```

Ah — that looks a *lot* like the working tree of the project from the first part of this book, doesn't it? That's because that's precisely what this *is*: a compressed representation of your file structure inside the repository.

Now, again, this object is simply a pointer to other objects. But you can keep unwrapping objects as you go.

The blob object

For instance, you can see the state of the **LICENSE** file in this commit with `git cat-file`:

```
git cat-file -p 8b23445f4a55ae5f9e38055dec94b27ef2b14150
```

I see all that glorious legalese of the MIT license I added to my repository so many chapters ago:

```
MIT License

Copyright (c) 2019

Permission is hereby granted, free of charge, to any person
obtaining a copy
of this software and associated documentation files (the
"Software"), to deal
in the Software without restriction, including without
limitation the rights
to use, copy, modify, merge, publish, distribute, sublicense,
and/or sell...
<snip>
```

You can dig further into the tree by following the references down. What's inside the **articles** directory in this commit? The following command will tell you that:

```
git cat-file -p d27f2eb006fff5b83fdc5d6639c7cfabdcf9fc37
```

I see the following files inside that directory:

```
100644 blob e69de29bb2d1d6434b8b29ae775ad8c2e48c5391    .keep
100644 blob f8a69b62146eceef1b9078fed8788fbb6089f14f
clickbait_ideas.md
100644 blob e69de29bb2d1d6434b8b29ae775ad8c2e48c5391
ios_article_ideas.md
```

Looking inside **clickbait_ideas.md** with `git cat-file` again, I'll see the full contents of that file as I committed it:

```
# Clickbait Article Ideas

These articles shouldn't really have any content but need
irresistible titles.

- [ ] Top 10 iOS interview questions
- [ ] 8 hottest rumors about Swift 5 - EXPOSED
- [ ] Try these five weird Xcode tips to reduce app bloat
- [ ] Apple to skip iOS 13, eyes a piece of Android's pie
- [ ] 15 ways Android beats iOS into the ground and 7 ways it
doesn't
- [ ] I migrated my entire IT department back to Windows XP -
and then this happened
- [ ] The Apple announcement that should worry Swift developers
- [ ] iOS 13 to bring back skeuomorphism amidst falling iPhone
sales
- [x] Machine Learning to blame for skewed election results
```

You could keep digging further, but I'm sure you've seen enough to get an understanding of how Git stores commits, trees and the objects that represent the files in your project. It's turtles all the way down, man.

So you can see how easily Git can reconstruct a branch, based on a single commit hash:

1. You switch to a named branch, which is a label that references a commit hash.
2. Git finds that commit object by its hash, then it gets the tree hash from the commit object.
3. Git then recurses down the tree object, uncompressing file objects as it goes.
4. Your working directory now represents the state of that branch as it is stored in the repo.

That's enough mucking about under the hood of Git; navigate back up to the root directory of your project and let Git take care of its own business. You have more important things to attend to.

Key points

- Git uses the SHA-1 hash of content to create references to commits, trees and blobs.
- A commit object stores the metadata about a commit, such as the parent, the author, timestamps and references to the file tree of this commit.
- A tree object is a collection of references to either child trees or blob objects.
- Blob objects are compressed collections of files; usually, the set of files in a particular directory inside the tree.
- `git rev-parse`, among other things, will translate a short hash into a long hash.
- `git cat-file`, among other things, will show you the pertinent metadata about an object.

Where to go from here?

Git has quite an elegant and powerful design when you think about it. And the wonderful thing is that all of this is abstracted away from you at the command line, so you don't need to know *anything* about the mechanisms underneath if you're the type who thinks ignorance is bliss.

But for those of you who *do* want to know how things work, and who want to be able to fix things when they go awry (and in Git, they often do), then this entire section will be a treat for you. The next chapter deals with a very common scenario that will (and should) occur with some regularity if you're doing any level of distributed development: merge conflicts.

Chapter 13: Merge Conflicts

By Chris Belanger

The reality of development is that it's a messy business; on the surface, it's simply a linear progression of logic, a smattering of frameworks, a bit of testing — and you're done. If you're a solo developer, then this may very well be your reality. But for the rest of us who work on code that's been touched by several, if not hundreds, if not thousands of other hands, it's inevitable that you'll eventually want to change the same bit of code that someone else has recently changed.

Imagine that your team's project contains the following bit of HTML:

```
<p>Head over to the following link to learn how to get started
with Git:</p>
<a href="http://guides.github.com/activities/hello-
world/">link</a>
```

You've been tasked with updating all of the text of the links to something more descriptive, while your teammate has been tasked with changing HTTP URLs in this particular project to HTTPS.

At 9:00 a.m., your teammate pushes the following change to the piece of code to the project repository, to update `http` to `https`:

```
<p>Head over to the following link to learn how to get started
with Git:</p>
<a href="https://guides.github.com/activities/hello-
world/">link</a>
```

At 9:01 a.m. (because you were a little farther back in the coffee lineup that morning), you attempt to push the following change to the repository:

```
<p>Head over to the following link to learn how to get started
with Git:</p>
<a href="http://guides.github.com/activities/hello-
world/">GitHub's Hello World project</a>
```

But, instead of Git committing your changes to the repository, you receive the following message instead:

```
 ! [rejected]        master -> master (fetch first)
error: failed to push some refs to 'https://github.com/
supersites/git-er-done.git'
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.
```

That's something you've seen before, especially if you worked through Chapter 8, "Syncing with a Remote." The remote has your teammate's changes that you just haven't yet pulled down to your local system. "Easy fix," you think to yourself, so you execute `git pull` as suggested, and…

```
From https://github.com/supersites/git-er-done
   7588a5f..328aa94  master     -> origin/master
Auto-merging index.html
CONFLICT (content): Merge conflict in index.html
Automatic merge failed; fix conflicts and then commit the
result.
```

Well, that didn't go as planned. You were expecting Git to be smart to merge the contents of the remote, which contains the commits from your teammate, with your local changes. But, in this case, you and your teammate have changed the **same line**. And since Git, by design, doesn't know anything about the language you're working with, it doesn't know that your changes won't impact your teammate's changes — and vice-versa. So Git plays it safe and bails, and asks *you* to do the work to merge the two files manually.

Welcome to the wonderful world of **merge conflicts**.

What is a merge conflict?

As a human, it's fairly easy to see how two people modifying the same line of code in two separate branches could result in a conflict, and you could even argue that a halfway intelligent developer could easily work around that situation with a minimum of fuss. But Git can't reason about these things in a rational manner as you or I would. Instead, Git uses an algorithm to determine what bits of a file have changed and if any of those bits could represent a conflict.

For simple text files, Git uses an approach known as the **longest common subsequence** algorithm to perform merges and to detect merge conflicts. In its simplest form, Git find the longest set of lines in common between your changed file and the common ancestor. It then finds the longest set of lines in common between your teammate's changed file and the common ancestor.

Git aligns each pair of files along its longest common subsequence and then asks, for each pair of files, "What has changed between the common ancestor and this new file?" Git then takes those differences, looks again, and asks, "Now, of those changes in each pair of files, are there any sets of lines that have changed *differently* between each pair?" And if the answer is "Yes," then you have a merge conflict.

To see this in action, you'll start working through the sample project for this section of the book, and you'll merge in some of your team's branches in order to see that resolving merge conflicts isn't *quite* as scary or frustrating as it looks on the surface.

Handling your first merge conflict

To get started, you'll need to clone the magicSquareApp repository that's used in this section of the book.

You can do this by way of the `git clone` command:

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

Once that's done, navigate into the directory into which you cloned it.

Now, here's the situation: Zach has been working on the front-end HTML of the magic square application to make it work with the back-end JavaScript. Zach isn't a designer, so Yasmin has offered to lend her design skills to the project UI and style the front end so that it looks presentable.

As the project lead, you're responsible for merging the various bits together and testing out the project. So, at this point, you'd like to verify that Zach's HTML works properly with Yasmin's UI. To do this, you'll have to merge Zach's work with Yasmin's work, and then test the project locally.

Merging from another branch

Zach has been doing his work in the **zIntegration** branch, while Yasmin has been working in the **yUI** branch. Your job is to merge Yasmin's branch with Zach's branch and resolve any conflicts.

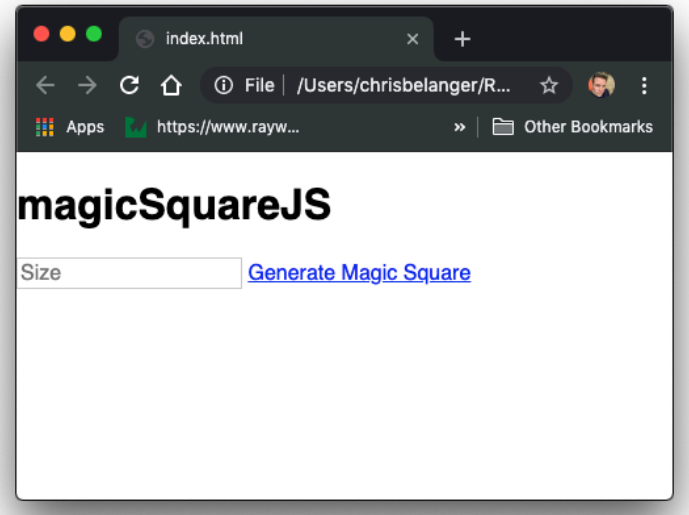
Pull down Yasmin's branch so you have a local, tracked copy of the branch:

```
git checkout yUI
```

First, switch to Zach's branch:

```
git checkout zIntegration
```

Open up **index.html** in a browser, to see what things look like in their current, pre-Yasminified state:



Well, it's clear that Zach is no designer. Good thing we have Yasmin.

Now you need to merge in Yasmin's UI branch:

```
git merge yUI
```

It appears that Zach and Yasmin's work wasn't *completely* decoupled, though, since Git indicates you have a merge conflict:

```
Auto-merging index.html
CONFLICT (content): Merge conflict in index.html
Automatic merge failed; fix conflicts and then commit the
result.
```

Helpfully, Git tells you above what file or files contain the merge conflicts. Open up **index.html** and find the following section:

```
  <body>
    <h1>magicSquareJS</h1>
<<<<<<< HEAD
    <section>
      <input type="text" placeholder="Size"
        id="magic-square-size" />
      <a href="#"
        id="magic-square-generate-button">
          Generate Magic Square</a>
      <pre id="magic-square-display">
=======
    <section class="box">
      <input type="text" class="flex-item" placeholder="Size"/>
      <a href="#"
        class="flex-item btn" >Generate Magic Square</a>
      <pre class="flex-item" >
>>>>>>> yUI
      </pre>
```

OK, you admit that your HTML is a *little* rusty, but you're pretty sure that `<<<<<<< HEAD` stuff isn't valid HTML. What on earth did Git do to your file?

Understanding Git conflict markers

What you're seeing here is Git's representation of the conflict in your working copy. Git compared Yasmin's file to the common ancestor, then compared Zach's file to the common ancestor and found this block of code that had changed differently in each case.

In this case, Git is telling you that the HEAD revision (i.e., the latest commit on Zach's branch) looks like the block between the `<<< HEAD` marker, and the `===` marker. The latest revision on Yasmin's branch is the block contained between the `===` line and the `>>> yUI` marker.

Git puts both revisions into the file in your working copy, since it expects you to do the work yourself to resolve this conflict. If you were intimately familiar with the code in question, you might know exactly how to combine Zach's and Yasmin's code to get the desired result. But you skipped a few too many project design meetings, didn't you?

No matter; you can ask Git to give you a few more clues as to what's happened here. Remember that a merge in Git is a three-way merge, but by default Git only shows you the two child revisions in a merge conflict; in this case, Yasmin and Zach's changes. It would be quite instructional to see the common ancestor for both of these child revisions, to figure out the intent behind each change.

Resolving merge conflicts

First, you need to return to the previous state of your working environment. Right now, you're mid-merge, and you only have two choices at this point: Either go forward and resolve the merge, or roll back and start over. Since you want to look at this merge conflict from a different angle, you'll roll back this merge and start over.

Reset your working environment with the following command:

```
git reset --hard HEAD
```

This reverts your working environment back to match HEAD, which, in this case, is the latest commit of your current branch, `zIntegration`.

A better way to view merge conflicts

Now, you can configure Git to show you the three-way merge data with the following command:

```
git config merge.conflictstyle diff3
```

> **Note**: If you ever wanted to change back to the default merge conflict tagging, simply execute `git config merge.conflictstyle merge` to get rid of the common ancestor tagging.

To see the difference in the merge conflict output, run the merge again:

```
git merge yUI
```

Git explains patiently that yes, there's still a conflict. In fact, this is a good time to see what Git's view of your working tree looks like, before you go in and fix everything up. Execute the `git status` command, and Git shows you its understanding of the current state of the merge:

```
On branch zIntegration
Your branch is up to date with 'origin/zIntegration'.

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

Changes to be committed:

  new file:   css/main.css

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

  both modified:   index.html
```

Most of that output makes sense, but the last bit is rather odd: `both modified: index.html`. But there's only one **index.html**, isn't there? Why does Git think there is more than one?

A consolidated git status

Remember that Git doesn't always think about files, per se. In this case, Git is talking about both *branches* that are modified. To see this in a bit more detail, you can add the `-s` (`--short`) and `-b` (`--branch`) options to `git status` to get a consolidated view of the situation:

```
git status -sb
```

Git responds with the following:

```
## zIntegration...origin/zIntegration
A  css/main.css
UU index.html
```

The first two columns (showing `A` and `UU`) represent the "ours" versus "theirs" view of the code. The left column is your local branch, which currently is the mid-merge state of the original `zIntegration` branch mixed with the changes from the `yUI` branch. The right column is the remote branch. So this abbreviated `git status` command shows the following:

- You have one file added (`A`) on your local branch; this is **css/main.css** that Yasmin must have added in her work. But it's not in conflict with your work.
- On the other hand, you have not one, but *two* revisions of a file that are unmerged (`U`) in your branch. This is the original **index.html** from the `zIntegration` branch, and the **index.html** from your `yUI` branch.

These files are considered unmerged because Git has halted partway through a merge, and put the onus on you to fix things up. Once you've fixed them up, committing those changes will continue the merge.

Editing conflicts

Open up **index.html** and have a look at the conflicted block of code now, with the new `diff3` conflict style:

```
  <body>
    <h1>magicSquareJS</h1>
<<<<<<< HEAD
    <section>
      <input type="text" placeholder="Size" id="magic-square-
size" />
      <a href="#" id="magic-square-generate-button">Generate
Magic Square</a>
      <pre id="magic-square-display">
||||||| merged common ancestors
    <section>
      <input type="text" placeholder="Size"/>
      <a href="#">Generate Magic Square</a>
      <pre>
=======
    <section class="box">
      <input type="text" class="flex-item" placeholder="Size"/>
      <a href="#" class="flex-item btn" >Generate Magic Square</
```

```
a>
      <pre class="flex-item" >
>>>>>>>yUI
      </pre>
```

There's a new section in there: `||| merged common ancestors`. This shows you the common ancestor of both Yasmin and Zach's changes; that is, what the code looked like before each created their own branch. A visual comparison of HEAD (which is Zach's branch) against the common ancestry shows the following changes:

- Added `id="magic-square-size"` to the `input` tag
- Added `id="magic-square-generate-button"` to the `a` (anchor) tag
- Added `id="magic-square-display"` to the `pre` tag

A quick visual comparison of Yasmin's changes against the common ancestor shows the following changes:

- Added `class="box"` to the `section` tag
- Added `class="flex-item"` to the `input` tag
- Added `class="flex-item btn"` to the `a` tag
- Added `class="flex-item"` to the `pre` tag

It looks like it will be less work to migrate Zach's changes into Yasmin's code. So edit **index.html** by hand, moving Zach's new `id` attributes, from the first block of code in the conflicted section, into the third block in the conflicted section, which is Yasmin's code.

When you've moved those three `id` attributes into Yasmin's code, you can now delete the entire first two blocks from the conflicted section, from `<<< HEAD` all the way to `===`. Then, delete the `>>> yUI` line as well. When you're done, this section of code should look like the following:

```
  <body>
    <h1>magicSquareJS</h1>
    <section class="box">
      <input type="text" id="magic-square-size" class="flex-
item" placeholder="Size"/>
      <a href="#" id="magic-square-generate-button" class="flex-
item btn" >Generate Magic Square</a>
        <pre id="magic-square-display" class="flex-item" >
        </pre>
```

Save your work and return to the command line.

Completing the merge operation

You've finished resolving the conflict, so you can stage your changes with the following:

```
git add index.html
```

Execute the condensed version of `git status -sb` to see what Git thinks about your merge attempt:

```
On branch zIntegration
Your branch is up to date with 'origin/zIntegration'.

All conflicts fixed but you are still merging.
  (use "git commit" to conclude merge)

Changes to be committed:

  new file:   css/main.css
  modified:   index.html
```

There you are; one new file and one modified file. Git's noticed that you've resolved the outstanding conflicts, so all that's left to do to complete the merge is to commit your staged changes.

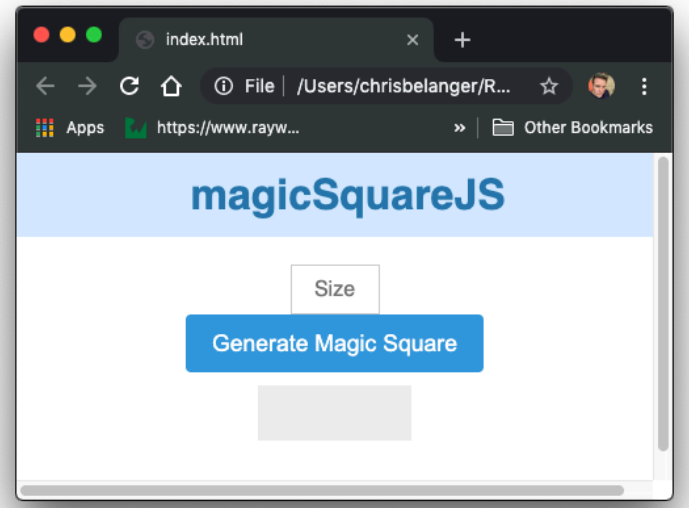
Commit those changes now, this time letting Git provide the merge message via Vim:

```
git commit
```

Type **:wq** inside of Vim to accept the preconfigured merge commit message, and Git dumps you back to the command line with a brief status, showing you that the merge succeeded:

```
[zIntegration af33aaa] Merge local branch 'yUI' into
zIntegration
```

Now, open **index.html** in a browser to see the changes:



That looks quite good. It's not fully functional at the moment, but you can see that Yasmin's styling changes are working. You're free to delete her branch and merge this work into `master`.

First, delete the `yUI` branch:

```
git branch -d yUI
```

Switch to the `master` branch:

```
git checkout master
```

Now, attempt a merge of the `zIntegration` branch:

```
git merge zIntegration
```

Git takes you straight into Vim, which means the merge had no conflicts. Type **:wq** to save this commit message and complete the merge. Git responds with the results of the merge:

```
Merge made by the 'recursive' strategy.
 css/main.css | 268 ++++++++++++++++++++++++++++++++++++++++++++++++
```

```
++++++++++++++++++++++
 index.html   |  16 +++++++----
 js/main.js   |  85 ++++++++++++++++++++++++++++++++++++++++++++++++
++++++++++++++
 3 files changed, 363 insertions(+), 6 deletions(-)
 create mode 100644 css/main.css
```

You're now able to delete the `zIntegration` branch, so do that now:

```
git delete -b zIntegration
```

Challenge: Resolve another merge conflict

The challenge for this chapter is straightforward: resolve another merge conflict.

Xanthe has an old branch with some updates to the documentation; this work is in the `xReadmeUpdates` branch. You want to merge that work to `master`.

The steps are as follows:

- Check out the `xReadmeUpdates` branch and look at the README.md file to see Xanthe's version.
- Check out `master`, since this is the destination for your merge.
- Resolve any merge conflicts by hand.
- Stage your changes.
- Commit your changes.
- Delete the `xReadmeUpdates` branch.

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

Key points

- Merge conflicts occur when you attempt to merge one set of changes with another, conflicting set of changes.
- Git uses a simple three-way algorithm to detect merge conflicts.
- If Git detects a conflict when merging, it halts the merge and asks for manual intervention to resolve the conflict.
- `git config merge.conflictstyle diff3` provides a three-way view of the conflict, with the common ancestor, "their" change, and "our" change.
- `git status -sb` gives a concise view of the state of your working tree.
- To complete a merge that's been paused due to a conflict, you need to manually fix the conflict, add your changes, and then commit those changes to your branch.

Where to go from here?

In practice, merge conflicts can get pretty messy. And it might seem that, with a bit of intelligence, Git could detect that adding HTML attributes to a tag is not really a conflict. And there are, in fact, lots of tools, such as IDEs and their plugins, that are language-aware and can resolve conflicts like this easily, without making you make all the edits by hand. But no tool can ever replace the insight that you have as a developer, nor can it replace your intimate understanding of your code and its intent. So even though you may come across tools that seem to do most of the work of resolving merge conflicts for you, at some point you'll find that there is no other way to resolve a merge conflict except by manual code surgery, so learning this skill now will serve you well in the future.

Up to now, your workflow has been constrained to the "happy path": you can create commits, switch between branches, and generally get along quite well without being interrupted. But real life isn't like that; you'll more often than not be partway through working on a feature or a fix, when you want to switch your local branch to take a look at something else. But because Git works at the atomic level of the commit, it doesn't like leaving things in an uncommitted state. So you need to stash the current state of your work somewhere, before you switch branches. And `git stash`, covered in the next chapter, does just that for you.

Chapter 14: Demystifying Rebasing

By Chris Belanger

Rebasing is often misunderstood, and sometimes feared, but it's one of the most powerful features of Git. Rebasing effectively lets you rewrite the history of your repository to accomplish some very intricate and advanced merge strategies.

Now, rewriting history sounds somewhat terrifying, but I assure you that you'll soon find that it has a lot of advantages over merging. You just have to be sure to rebase responsibly.

Why would you rebase?

Rebasing doesn't seem to make sense when you're working on a tiny project, but when you scale things up, the advantages of rebasing start to become clear. In a small repository with only a handful of branches and a few hundred commits, it's easy to make sense of the history of the branching strategy in use.

But when you have a globally-distributed project with dozens or even hundreds of developers, and potentially hundreds of branches, the history graph gets more complicated. It's especially challenging when you need to use your repository commit history to identify when and how a particular piece of code changed. For example, when you're troubleshooting a previously-working feature that's somehow regressed.

Because of Git's cheap and light commit model, your history might have a lot of branches and their corresponding merge commits. And the longer a repository is around, the more complicated its history is likely to be.

The issue with merge commits becomes more apparent as the number of branches off of a feature branch grows. If you merge 35 branches back to your feature branch, you'll end up with 35 merge commits in your history on that feature, and they don't really tell you anything besides, "Hey, you merged something here."

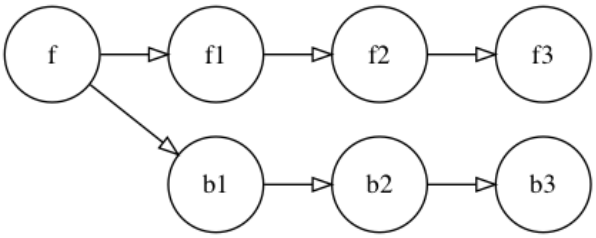
While that can often be useful, if the development workflow of your team results in fast, furious and short-lived branches, you might benefit from limiting merge commits and rebasing limited-scope changes instead. Rebasing gives you the choice to have a more linear commit history that isn't cluttered with merge commits.

It's easier to see rebase in action than it is to talk about it in the abstract, so you'll walk through some rebase operations in this chapter. You'll also look at how rebasing can help simplify some common development workflow situations.

What is rebasing?

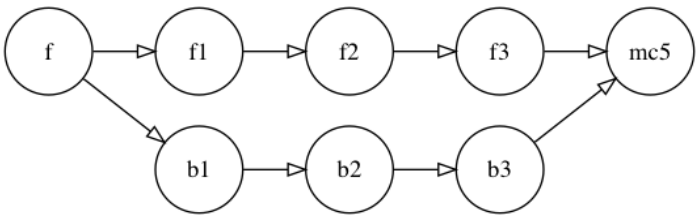
Rebasing is essentially just replaying a commit or a series of commits from history on top of a different commit in the repository. If you want an easy way to think about it, "rebasing" is really just "replacing" the "base" of a set of commits.

Take a look at the following scenario: The **f** commits denote a random **feature** branch, and the **b** commits denote a **bugfix** branch you created in order to correct or improve something inside the feature branch, without impeding work on the feature development. You've made a few commits along the way, and now it's time to merge the work in the **bugfix** branch back to **feature**.



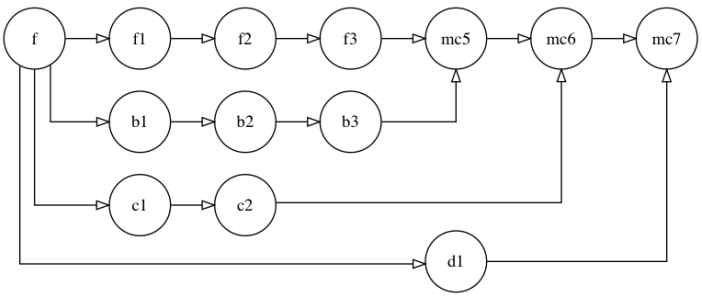
A simple branch (b) off of feature (f).

If you were to simply merge the **bugfix** branch back to **feature**, as you would normally tend to do, then the resulting history graph would look like this:



A simple branch (b) off of feature (f), merged back to feature with merge commit mc5.

The **mc5** commit is your merge commit. Merge commits are a familiar sight, and merging is a mechanism that you and most everyone else who uses Git understands well. But as the size and activity of your repository grow, you can end up with a very complicated graph.

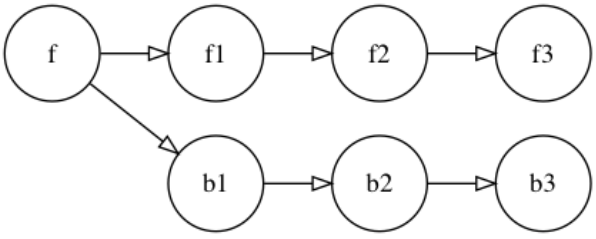


A more complex set of multiple branches and merge commits, with merge commits mc5, mc6 and mc7.

Depending on the kind of work you're doing, you may not want to have your repository history show that you branched off, did some work and merged the changes back in. That little bit of extra cognitive overhead starts to add up as you try to make sense of months or even years of history graphs. Especially with small or trivial changes, you might prefer a linear history seeing them branched off and merged in again.

Rebasing gives you the freedom to avoid retaining branch history and merge commits. Instead, you can recreate your work as a linear commit progression.

Go back to the original branching scenario, with your **bugfix** branch off of **feature**:

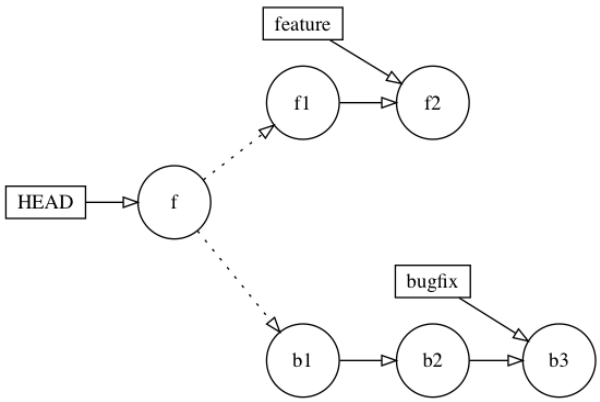


A simple branch (b) off of feature (f).

Rebasing uses a series of standard Git operations under the hood to accomplish rebasing. It isn't quite as straightforward as just moving commits as you'd move nodes in a tree data structure, for instance.

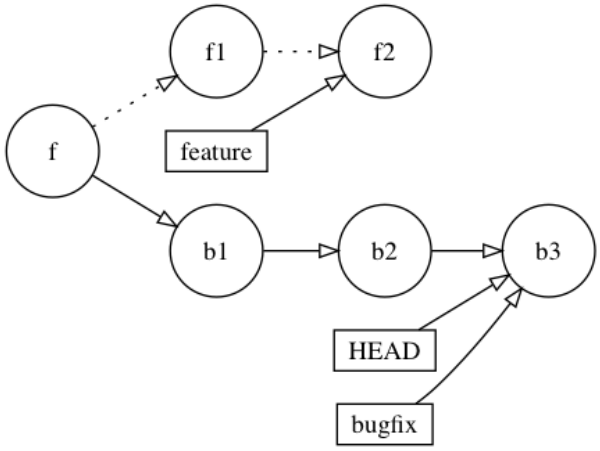
Let's presume you wanted it to appear as though the work performed on **feature** followed the work you did on **bugfix**. In Git parlance, you'd be rebasing **feature** on top of **bugfix**.

Git first rewinds the branch that you're rebasing – in this case, **feature** – back to its common ancestor with **bugfix**. The common ancestor is commit **f**:



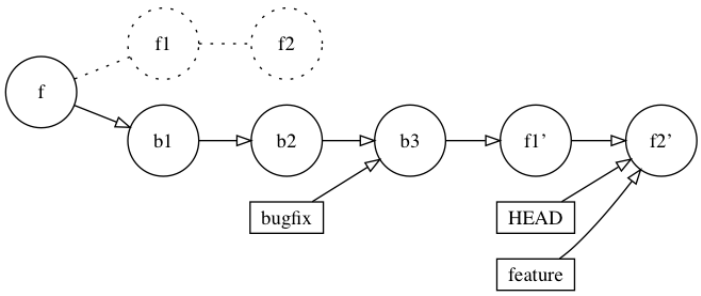
Rewinding HEAD to the common ancestor of the feature and bugfix branches

Git then replays the patch of each commit from the branch you're rebasing on top of, in this case, **bugfix**, and moves the `HEAD` and `bugfix` labels along:



Applying b1, b2 and b3 patches on top of the common ancestor and moving labels along.

Finally, one at a time, Git applies the patch of each commit from the branch you're rebasing, in this case, **feature**, and moves the `HEAD` and `feature` labels along:



Applying f1, and f2 patches on top of the new base branch and moving labels along

At this point, you no longer have any real reference to those original commits from the **feature** branch. Git will eventually just collect those orphaned commits (or "loose" commits, as they're known) and clean them up in a regular garbage collection process. Although the loose commits still "know" who their parent is, you won't see these commits show up in the history graph, so I haven't shown them as an edge on the graph above.

It's like those commits never even existed. That's where the whole idea that Git rebase is, quite literally, rewriting history comes from. For all intents and purposes, anyone looking at the repository has no reason to believe that you didn't just make those commits on top of **feature** in the first place.

It's important to understand that Git is not just *moving* commits here; it's actually creating a *brand new* commit based on the contents of the patch it calculated at each commit in your branch.

> **Note**: Choose to rebase only when the branch you're on is not shared with anyone else because, once again, you're rewriting the history of the repository. If you *must* rebase a shared branch, you'll have to coordinate with your team to make sure that everyone has pushed any and all changes to the branch and deleted it locally before you begin your work. Otherwise, *you're gonna have a bad time*.

Creating your first rebase operation

To start, find the starting repository for this chapter in the **starter** folder and unzip it to a working location.

You'll create an extremely trivial branch off of **wValidator**, make a change on that branch, and then rebase **wValidator** on top of your branch.

First, check that you're on the correct branch for your repo:

```
git branch
```

You should be on the **wValidator** branch.

Create a new branch named **cValidator** from **wValidator**:

```
git checkout -b cValidator
```

Next, open up **README.md** and add your name to the end of the `# Maintainers` section:

```
# Maintainers

This project is maintained by teamWYXZ:
- Will
- Yasmin
- Xanthe
- Zack
- Chris
```

Save your changes and exit the editor.

Add your changes:

```
git add .
```

Commit your changes with an appropriate message:

```
git commit -m "Added new maintainer to README.md"
```

At this point, you have a branch **cValidator** with a commit containing changes to README.md. Now, you want to simulate someone creating more commits on the **wValidator** branch.

Switch back to the **wValidator** branch:

```
git checkout wValidator
```

Open **README.md** and add your initial to the end of the team name in the `# Maintainers` section:

```
# Maintainers

This project is maintained by teamWYXZC:
```

(A bit of alphabet soup, that is: "teamWXYZC". You should petition the team to get a really cool name someday. But that's for later.)

Save your changes, exit the editor and stage your changes:

```
git add .
```

Now create a commit with that change, using an appropriate message:

```
git commit -m "Updated team acronym"
```

Take a quick look at the current state of the repository in graphical form:

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

The top three lines show you what's what:

```
* c628929 (HEAD -> wValidator) Updated team acronym
| * 2eb17a2 (cValidator) Added new maintainer to README.md
|/
* 3574ab3 Whoops - didn't need to call that one twice
```

You have a commit in a separate branch, **cValidator**, that you'd like to rebase **wValidator** on top of. While you could merge this in as usual with a merge commit, there's really no need, since the change is so small and the changes in each branch are trivial and related to each other.

To rebase **wValidator** on top of **cValidator**, you need to be on the **wValidator** branch (you're there now), and tell Git to execute the rebase with the following command:

```
git rebase cValidator
```

Git shows a bit of output, telling you what it's doing:

```
First, rewinding head to replay your work on top of it...
Applying: Updated team acronym
```

As expected, Git rewinds `HEAD` to the common ancestor — commit `3574ab3` in the graph shown above. It then applies each commit from the *branch you are on* — i.e., the branch that's being rebased — on top of the end of the *branch you are rebasing onto*. In this case, the only commit from **wValidator** Git has to apply is `78c60c3 – Updated team acronym`.

Take a look at the history graph to see the end result by executing the following:

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

You'll see the following linear activity at the top of the graph:

```
* 17771e6 (HEAD -> wValidator) Updated team acronym
* 2eb17a2 (cValidator) Added new maintainer to README.md
* 3574ab3 Whoops – didn't need to call that one twice
```

For a bit of perspective, you can look at the simple graphs at the start of the chapter for a visual reference to what's happened here. But here's the play-by-play to show you each of the steps:

- Git rewound back to the common ancestor (`3574ab3`).
- Git then replayed the **cValidator** branch commits (in this case, just `3f7969b`) on top of the common ancestor.
- Git left the branch label `cValidator` attached to `3f7969b`.
- Git then replayed the patches from each commit in **wValidator** on top of the commits from **cValidator** and moved the `HEAD` and `wValidator` labels to the tip of this branch.

You don't need that **cValidator** branch anymore, nor is instructive to keep that label hanging around in the repository, so clean up after yourself with the following command:

```
git branch -d cValidator
```

As an aside, did you notice the difference in the commit hashes?

- Old commit for `Updated team acronym`: `78c60c3`

- New commit for `Updated team acronym`: `f76b62c`

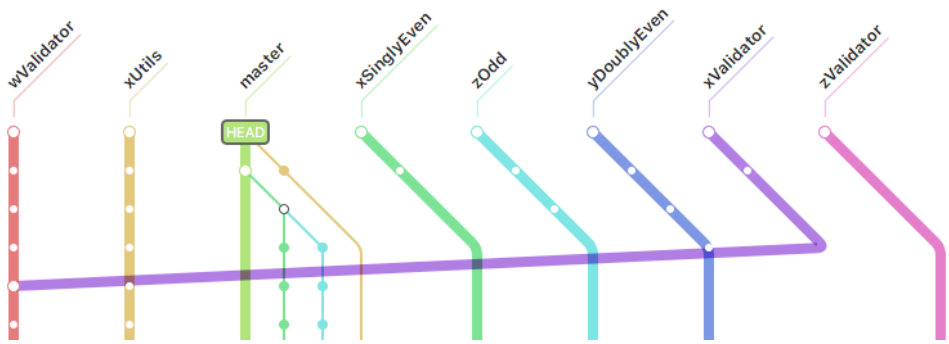
They're different because what you have at the tip of **wValidator** is *a brand-new commit* — not just the old commit tacked onto the end of the branch.

You may be wondering where that old commit went, and you'll dig into those details just a little further into this chapter as you investigate a more common scenario where you'll encounter and resolve rebase conflicts.

A more complex rebase

Let's go back to our Magic Square development team. Several people have been working on the Magic Squares app; Will in particular has been working on the wValidator branch. Xanthe has also been busy refactoring on the xValidator branch.

Here's what the repository history looks like at this point:



The partial GitUp view of the repository, including the branches wValidator and xValidator.

Xanthe has branched off of Will's original branch to work on some refactoring, and it's now time to bring everything back into the wValidator branch. Because branching is cheap and easy in Git, these types of scenarios where developers branch off of existing branches is fairly common. Again, there's nothing saying that you always have to branch off of master — you can support any branching scheme you like, as long as you can keep track of things!

Although you could just merge all of Xanthe's work into Will's branch, you'd end up with a merge commit and clutter the history a little. And, conceptually, it makes sense to rebase in this situation, because the refactoring that Xanthe has done is within the logical context of Will's work, so you might as well make it appear that the work has all taken place on a common branch.

First, check out the commits made since the common ancestor of wValidator and xValidator:

```
git log --oneline bf3753e~..
```

That last bit is new. What `bf3753e~..` means is: "limit `git log` to just this particular commit (inclusive) up to `HEAD`." Not providing the end commit hash indicates `HEAD`.

You'll see the following:

```
f76b62c (HEAD -> wValidator) Updated team acronym
3f7969b Added new maintainer to README.md
3574ab3 Whoops — didn't need to call that one twice
43d6f24 check05: Finally, we can return true
bf3753e check04: Checking diagonal sums
```

Those are the most recent commits on wValidator. Now, you know that xValidator branched from wValidator, so is it possible to view just what's changed on xValidator?

Absolutely. Execute the following to see what's happened since you branched xValidator from wValidator:

```
git log --oneline bf3753e~..xValidator
```

You'll see the following:

```
8ef01ac (xValidator) Refactoring the main check function
5fea71e Removing TODO
bf3753e check04: Checking diagonal sums
```

Your goal is to rebase the changes from wValidator on top of xValidator.

To begin your rebase operation, go to the command line, and navigate into your repository's directory.

To start, check out the branch you want to merge your changes into with the following command:

```
git checkout wValidator
```

Git tells you you're already on that branch — no worries. It always pays to be sure.

Now, begin the rebase operation with the `git rebase` command, where you indicate which branch you want to rebase on top of your current branch:

```
git rebase xValidator
```

Resolving errors

`git rebase` provides quite a lot of verbose output, but if you look carefully through the output of your command, you'll see that there's a conflict you have to resolve in **js/magic_square/validator.js**.

Open up **js/magic_square/validator.js** and you'll see the conflict that you need to resolve. In this case, you want to keep the bits marked as `<<< HEAD`, since these are Xanthe's refactored changes that you want to keep.

> **Note**: You might be confused here. Why are you keeping the `HEAD` changes, if `HEAD` is the tip of the branch you're on — in this case, **wValidator**?
>
> In a rebase situation, `HEAD` refers to the tip of the branch you're rebasing on top of. As Git replays each commit onto this branch, `HEAD` moves along with each replayed commit.

Resolve the commit manually, removing the bits from the common ancestors and the original bits from Will's code. Save your work when you're done.

> **Note**: Did you notice the final separator line of the conflict?
>
> `>>>>>>> check05: Finally, we can return true`
>
> Because rebasing works by replaying the commits of the other branch one by one on the current branch, Git helpfully tells you in which commit the conflict occurred. For complex merge conflicts, this little bit of extra information can be quite useful.

When you're done, return to the command line and continue the rebase with the following command:

```
git rebase --continue
```

Oh, but Git won't let you continue. It gives you the following message:

```
js/magic_square/validator.js: needs merge
You must edit all merge conflicts and then
mark them as resolved using git add
```

Again, because you're working within the context of a single commit, you need to stage those changes. Git rebases each of the original commits one at a time, so you need to deal with and add the changes from each commit resolution one at a time.

Execute the following command to stage those changes to continue:

```
git add .
```

Then continue with the rebase:

```
git rebase --continue
```

But, frustratingly, Git *still* won't let you continue:

```
Applying: check05: Finally, we can return true
No changes - did you forget to use 'git add'?
If there is nothing left to stage, chances are that something
else
already introduced the same changes; you might want to skip this
patch.

Resolve all conflicts manually, mark them as resolved with
"git add/rm <conflicted_files>", then run "git rebase --
continue".
You can instead skip this commit: run "git rebase --skip".
To abort and get back to the state before "git rebase", run "git
rebase --abort".
```

This is one of those instances where Git can appear to be *completely* dense. Doesn't Git know that you *just* ran `git add`? Doesn't it *see* that you *just* resolved those commits? Gitdammit.

Feel free to vent for a second, and then consider the situation from Git's perspective.

What you did above was to keep everything from the commit you are rebasing from xValidator. But Git is effectively expecting that there should be *some* change in the commit you're rebasing from xValidator, as this is the most likely case you'll encounter when rebasing work.

Imagine if Git just assumed that taking the commit verbatim was a completely normal situation; if you weren't paying attention to your commit resolution, you'd likely hit a point where you'd unwittingly clobber some work on the branch on which you're rebasing on top of. And then you'd spend countless hours, late at night, with too much coffee, trying to figure out where your rebase went so horribly wrong. In this situation, Git's error message would actually help you out.

But in this case, since you *are* taking the commit verbatim with no changes, you can simply execute the following command to carry on:

```
git rebase --skip
```

> **Note**: This may or may not be a great time to tell you that you didn't actually *need* to perform that conflict resolution in **validator.js**, since you took that commit as-is from xValidator. You could've just executed `git rebase --skip` straight away to tell Git that you had no interest in resolving the commit and to rebase the commit unchanged.

Git then carries on and attempts to apply the second commit, but again halts on a conflict:

```
error: Failed to merge in the changes.
Patch failed at 0002 Whoops - didn't need to call that one twice
```

Open **js/magic_square/validator.js** and you'll see a conflict situation nearly identical to the one before. There's a change in the wValidator branch that doesn't make sense anymore (removing a duplicate line) since Xanthe's refactoring changes in xValidator make that change moot.

So, how do you tell Git to take the commit from xValidator as-is? That's right — you just skip the rebasing of this commit with the following:

```
git rebase --skip
```

Git then carries on and happily rebases the remaining two commits without conflict:

```
Applying: Added new maintainer to README.md
Applying: Updated team acronym
```

And you're done that frustrating, yet enlightening journey through Git rebasing.

To see the result of your work from the perspective of Git, have a look at your history graph again since that common ancestor:

```
git log --oneline bf3753e~..
```

You'll see that the two original commits Will made at the end of wValidator are gone (the commits with short hash `3574ab3` and `43d6f24`), and Xanthe's commits are now neatly tucked in between the common ancestor and your updates to README.md, the xValidator branch label points to what was the tip of xValidator, and the wValidator branch label points to the tip as expected:

```
57f62b0 (HEAD -> wValidator) Updated team acronym
b14948d Added new maintainer to README.md
8ef01ac (xValidator) Refactoring the main check function
5fea71e Removing TODO
bf3753e check04: Checking diagonal sums
```

Stop just for a moment and consider what you've done here. Where did those commits from Will go? If you'd just done a simple merge, as you're used to doing, you would have still seen them in the history of the repo.

Even more confusingly, you can still find these commits in the logs. Execute the following command to see the three logged commits, starting at the "Whoops — didn't need to call that one twice" commit of `3574ab3`:

```
git log --oneline -3 3574ab3
```

That shows the history of the wValidator branch, from `3574ab3` back, as you understood it before you started rebasing:

```
3574ab3 Whoops — didn't need to call that one twice
43d6f24 check05: Finally, we can return true
bf3753e check04: Checking diagonal sums
```

But where are those commits? Essentially, those commits are orphaned, or "loose" as Git refers to them. They are no longer referenced from any part of the repository tree, except for their mention in the Git internal logs.

You can see that the object still exists inside the .git directory:

```
git cat-file -p 3574ab3
```

Git returns with the commit metadata:

```
tree 1b4c07023270ed26167d322c6e7d9b63125320ef
parent 43d6f24d140fa63721bd67fb3ad3aafa8232ca97
author Will <will@example.com> 1499074126 +0700
committer Sam Davies <sam@razeware.com> 1499074126 +0700

Whoops - didn't need to call that one twice
```

But as you saw from the repository history tree above, that actual commit is no longer referenced anywhere. It's just sitting there until Git does its usual garbage collection, at which point Git will physically delete any loose objects that have been hanging around too long.

> **Note**: Git generally tries to be as paranoid as possible when running garbage collection. It doesn't clean up every single loose object it finds, because there *might* be a chance that you made a mistake and really need the code from that commit.
>
> In fact, even though the commit isn't referenced anywhere, as long as you know the hash of that commit from the logs, you can still check it out and work with the code inside. So Git, like any good developer, will keep those files hanging around for a while…*juuuuust* in case you need them later. Thanks, Git!

Just for comparison purposes, check out what this entire scenario would have looked like from a merge perspective, as opposed to a rebase perspective:

```
*   96f42e3 (HEAD -> wValidator) Merge branch 'xValidator' into
wValidator
|\
| * 8ef01ac (xValidator) Refactoring the main check function
| * 5fea71e Removing TODO
* |   b567a15 Merge branch 'cValidator' into wValidator
|\ \
| * | 9443e8d (cValidator) Added new maintainer to README.md
* | | 76bacc5 Updated team acronym
|/ /
* | 3574ab3 Whoops - didn't need to call that one twice
* | 43d6f24 check05: Finally, we can return true
|/
* bf3753e check04: Checking diagonal sums
```

You can see that the merge commit would result in the branch actions remaining in the repository history. Instead, the rebase action streamlined the commit history and gathered those changes as a cohesive linear operation. This is, arguably, clearer to the casual observer of your repository's history.

Although the politics and goals of your development team will dictate your approach to merging and rebasing, here are some pragmatic tips on when rebasing might be more appropriate over merging, and vice versa:

- Choose to rebase when grouping the changes in a linear fashion makes contextual sense, such as Will's and Xanthe's work above that's contained to the same file.
- Choose to merge when you've created major changes, such as adding a new feature in a pull request, where the branching strategy will give context to the history graph. A merge commit will have the history of both common ancestors, while rebasing removes this bit of contextual information.
- Choose to rebase when you have a messy local commit or local branching history and you want to clean things up before you push. This touches on what's known as **squashing**, which you'll cover in a later chapter.
- Choose to merge when having a complex history graph doesn't affect the day-to-day functions of your team.
- Choose to rebase when your team frequently has to work through the history graph to figure out who changed what and when. Those merge commits add up over time!

There's a long, political history surrounding rebasing in Git, but hopefully, you've seen that it's simply another tool in your arsenal. Rebasing is most useful in your local, unpushed branches, to clean up the unavoidably messy business of coding.

But you've only begun your journey with rebasing. In the next chapter, you'll learn about **interactive rebasing**, where you can literally rewrite the history of the entire repository, one commit at a time.

Challenge

You've discovered that Zach has also been doing a bit of refactoring on the **zValidator** branch with the range checking function:

```
  | * 136dc26 (zValidator) Refactoring the range checking function
  |/
  * 665575c util02: Adding function to check the range of values
```

Your challenge is to rebase the work you've done on the wValidator branch on top of the zValidator branch. Again, the shared context here and the limited scope of the changes mean you don't need a merge commit.

Once you've rebased wValidator on top of zValidator, delete both the zValidator and xValidator branches, as you're done with them. Git might complain when you try to delete the branches. Explain why this is, and then figure out how to force Git to do it anyway.

As always, if you need help, or want to be sure that you've done it properly, you can always find the solution under the **challenges** folder for this chapter.

Key points

- Rebasing "replays" commits from one branch on top of another.
- Rebasing is a great technique over merging when you want to keep the repository history linear and as free from merge commits as possible.
- To rebase your current branch on top of another one, execute `git rebase <rebase-branch-name>`.
- You can resolve rebase conflicts just as you do merge conflicts.
- To resume a rebase operation after resolving conflicts and staging your changes, execute `git rebase --continue`.
- To skip rebasing a commit on top of the current branch, execute `git rebase --skip`.

Chapter 15: Rebasing to Rewrite History

By Chris Belanger

As you saw in the previous chapter, rebasing provides you with an excellent alternative to merging. But rebasing also gives you the ability to reorganize your repository's history. You can reorder, rewrite commit messages and even squash multiple commits into a single, tidy commit if you like.

Just as you'd tidy up your code before pushing your local branch to a remote repository, rebasing lets you clean up your commit history before you push to remote. This gives you the freedom to commit locally as you see fit, then rearrange and combine your commits into a handful of semantically-meaningful commits. These will have much more value to someone (even yourself!) who has to comb through the repository history at some point in the future.

> **Note**: Again, a warning: Rebasing in this manner is best used for branches that you haven't shared with anyone else. If you must rebase a branch that you've shared with others, then you must work out an arrangement with everyone who's cloned that repository to ensure that they all get the rebased copy of your branch. Otherwise, you're going to end up with a very complicated repository cleanup exercise at the end of the day.

To start, extract the compressed repository from the **starter** directory to a convenient location on your machine then navigate into that directory from the command line.

Reordering commits

You'll start by taking a look at Will's `wValidator` branch. Execute the following to see what the current history looks like:

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

You'll see the following at the top of your history graph:

```
* 45f5b4f (HEAD -> wValidator) Updated team acronym
* 15233a5 Added new maintainer to README.md
* 783031e Refactoring the main check function
* 6396aa8 Removing TODO
* 8e39599 check04: Checking diagonal sums
* 199e71d util06: Adding a function to check diagonals
* a28b9e3 check03: Checking row and column sums
* bdc8bc7 util05: Fixing comment indentation
* a4d6221 util04: Adding a function to check column sums
* 59fd06e util03: Adding function to check row sums
* 5f53302 check02: Checking the array contains the correct
values
* 136dc26 Refactoring the range checking function
* 665575c util02: Adding function to check the range of values
* 0fc1a91 check01: checking that the 2D array is square
* 5ec1ccf util01: Adding the checkSqaure function
```

It's not *terrible*, but this could definitely use some cleaning up. Your task is to combine those two trivial updates to **README.md** into one commit. You'll then reorder the `util*` commits and the `check*` commits together and, finally, to combine those related commits into two separate, tidy commits.

Interactive rebasing

First up: Combine the two top commits into one, inside the current branch. You're familiar with rebasing branches on top of other branches, but in this chapter, you'll rebase commits on top of other commits in the same branch.

In fact, since a branch is simply a label to a commit, rebasing branches on top of other branches really *is* just rebasing commits on top of one another.

But since you want to manipulate your repository's history along the way, you don't want Git to just replay commits on top of other commits. Instead, you'll use **interactive rebase** to get the job done.

First, get your game plan together. You want to combine, or **squash** those top two commits into one commit, give that new commit a clear message, and rebase that new squashed commit on top of the ancestor of the original commits. So your plan looks a little like the following:

- Squash `45f5b4f` and `15233a5`.
- Create a new commit message for this squashed commit.
- Rebase the resulting new commit on top of `783031e`.

To start an interactive rebase, you need to use the `-i` (`--interactive`) flag. Just as before, you need to tell Git where you want to rebase on top of; in this case, `783031e`.

So, execute the following to start your first Git interactive rebase:

```
git rebase -i 783031e
```

Git opens up the default editor on your system, likely Vim, and shows you the following:

```
pick 15233a5 Added new maintainer to README.md
pick 45f5b4f Updated team acronym

# Rebase 783031e..45f5b4f onto 783031e (2 commands)
#
# Commands:
# p, pick <commit> = use commit
# r, reword <commit> = use commit, but edit the commit message
# e, edit <commit> = use commit, but stop for amending
# s, squash <commit> = use commit, but meld into previous commit
# f, fixup <commit> = like "squash", but discard this commit's
log message
# x, exec <command> = run command (the rest of the line) using
shell
# b, break = stop here (continue rebase later with 'git rebase
--continue')
# d, drop <commit> = remove commit
# l, label <label> = label current HEAD with a name
# t, reset <label> = reset HEAD to a label
# m, merge [-C <commit> | -c <commit>] <label> [# <oneline>]
# .       create a merge commit using the original merge
commit's
# .       message (or the oneline, if no original merge commit
was
# .       specified). Use -c <commit> to reword the commit
message.
#
# These lines can be re-ordered; they are executed from top to
bottom.
```

```
#
# If you remove a line here THAT COMMIT WILL BE LOST.
#
# However, if you remove everything, the rebase will be aborted.
#
# Note that empty commits are commented out
```

Well, that's different! You've seen Vim in action before to create commit messages, but this is something new. What's going on?

Squashing in an interactive rebase

Here, Git's taken all of the commits past your rebase point, `15233a5` and `45f5b4f`, and put them at the top of the file with some rather helpful comments down below.

What you're doing at this step is effectively creating a script of commands for Git to follow when it rebases. Git will start at the top of this file and work downwards, applying each action it finds, in order.

To perform a squash of commits, you simply put the `squash` command on the line with the commit you wish to squash into the previous one. In this case, you want to squash `45f5b4f`, the last commit, into `15233a5`.

> **Note**: Git interactive rebase shows all commits in ascending commit order. This is a different order than what you're used to seeing in with `git log`, so be careful that you're squashing things in the correct direction!

Since you're back in Vim, you'll have to use Vim commands to edit the file. Cursor to the start of the `45f5b4f` line and press the `C` key, followed by the `W` key — this is the "change word" command, and it essentially deletes the word your cursor is on and puts you into insert mode.

So type **squash** right there. The top few lines of your file should now look as follows:

```
pick 15233a5 Added new maintainer to README.md
squash 45f5b4f Updated team acronym
```

That's all you need to do, so write your changes and quit with the familiar **Escape** + **:wq** + **Enter** combination.

Git then throws you straight back into *another* Vim editor, this one a little more familiar:

```
# This is a combination of 2 commits.
# This is the 1st commit message:

Added new maintainer to README.md

# This is the commit message #2:

Updated team acronym

# Please enter the commit message for your changes. Lines
starting
# with '#' will be ignored, and an empty message aborts the
commit.
#
# Date:      Sun Jun 9 07:28:08 2019 -0300
#
# interactive rebase in progress; onto 783031e
# Last commands done (2 commands done):
#    pick 15233a5 Added new maintainer to README.md
#    squash 45f5b4f Updated team acronym
# No commands remaining.
# You are currently rebasing branch 'wValidator' on '783031e'.
#
# Changes to be committed:
#       modified:   README.md
#
```

Okay, this is a commit message editor, which you've seen before. Here, Git helpfully shares the messages of all commits affected by this rebase operation. You can choose to keep or edit any one of those commit messages, or you can choose to create your own.

Creating the squash commit message

In this case, you'll just create your own. Clear this file as follows:

1. Type **gg** to ensure you're on the first line of the file.
2. Type **dG** (that's a capital "G") to delete all of the following lines from the file.

You now have a nice, clean file for a commit message. Press **i** to enter **insert mode** and then add the following message:

```
Updates to README.md
```

Then save your changes with **Escape** + **:wq** + **Enter** to continue the rebase operation.

Git carries on, emitting a little output with the success message at the end:

```
Successfully rebased and updated refs/heads/wValidator.
```

Execute the following to see what the repository history looks like now:

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

Look at the top two lines and you'll see the following (your hashes will be different, of course):

```
* 2492536 (HEAD -> wValidator) Updates to README.md
* 783031e Refactoring the main check function
```

Git has done just what you asked; it's created a new commit from the two old commits and rebased that new commit on top of the ancestor. To see the combined effect of squashing those two commits into one, check out the patch Git created for `2492536` with the following command:

```
git log -p -1
```

Take a look at the bottom of that output and you'll see the following:

```
-This project is maintained by teamWYXZ:
+This project is maintained by teamWYXZC:
 - Will
 - Yasmin
 - Xanthe
 - Zack
+- Chris
```

There's the combined effect of merging those two patches into one and rebasing that change on top of the ancestor commit.

Reordering commits

The asynchronous and messy nature of development means that sometimes you'll need to reorder commits to make it easier to squash a set of commits later on. Interactive rebase lets you literally rearrange the order of commits within a branch. You can do this as often as you need, to keep your repository history clean.

Execute the following to see the latest commits in your repository:

```
git log --oneline
```

Take a look at the order of the last dozen commits or so:

```
2492536 (HEAD -> wValidator) Updates to README.md
783031e Refactoring the main check function
6396aa8 Removing TODO
8e39599 check04: Checking diagonal sums
199e71d util06: Adding a function to check diagonals
a28b9e3 check03: Checking row and column sums
bdc8bc7 util05: Fixing comment indentation
a4d6221 util04: Adding a function to check column sums
59fd06e util03: Adding function to check row sums
5f53302 check02: Checking the array contains the correct values
136dc26 Refactoring the range checking function
665575c util02: Adding function to check the range of values
0fc1a91 check01: checking that the 2D array is square
5ec1ccf util01: Adding the checkSqaure function
69670e7 Adding a new secret
```

There's a collection of commits there that would make more sense if you arranged them contiguously. There's one set of check functions commits (the `check0x` commits) and another set of utility functions (the `util0x` commits). Before you merge these to `master`, you'd like to squash these related sets of commits into two commits to keep your repository history neat and tidy.

First, you'll need to start with the common ancestor of all of these commits. In this case, the base ancestor commit of the commits you're concerned with is `69670e7`. That commit will be the base for your interactive rebase.

Execute the following to start the interactive rebase on top of that base commit:

```
git rebase -i 69670e7
```

Once again, you'll be launched into Vim to edit the rebase script for the rebase operation:

```
pick 5ec1ccf util01: Adding the checkSqaure function
pick 0fc1a91 check01: checking that the 2D array is square
pick 665575c util02: Adding function to check the range of
values
pick 136dc26 Refactoring the range checking function
pick 5f53302 check02: Checking the array contains the correct
values
pick 59fd06e util03: Adding function to check row sums
pick a4d6221 util04: Adding a function to check column sums
```

```
pick bdc8bc7 util05: Fixing comment indentation
pick a28b9e3 check03: Checking row and column sums
pick 199e71d util06: Adding a function to check diagonals
pick 8e39599 check04: Checking diagonal sums
pick 6396aa8 Removing TODO
pick 783031e Refactoring the main check function
pick 2492536 Updates to README.md

# Rebase 69670e7..2492536 onto 69670e7 (14 commands)
```

Since Git starts at the top of the file and works its way down in order, you simply need to rearrange the lines in this file in contiguous order to rearrange the commits.

Since you're in Vim, you might as well use the handy Vim shortcuts to move lines around:

- To "cut" a line into the clipboard buffer, type **dd**.
- To "paste" a line into the edit buffer underneath the current line, type **p**.

Use these two key combinations to do the following:

1. Move the `util01` line to just above the `util02` line.
2. Leave the `Refactoring the range checking function` after `util02`.
3. Move the `util03` through `util06` lines, in order, to follow the `Refactoring the range checking function` commit.

When you're done, your rebase script should look as follows:

```
pick 0fc1a91 check01: checking that the 2D array is square
pick 5ec1ccf util01: Adding the checkSqaure function
pick 665575c util02: Adding function to check the range of
values
pick 136dc26 Refactoring the range checking function
pick 59fd06e util03: Adding function to check row sums
pick a4d6221 util04: Adding a function to check column sums
pick bdc8bc7 util05: Fixing comment indentation
pick 199e71d util06: Adding a function to check diagonals
pick 5f53302 check02: Checking the array contains the correct
values
pick a28b9e3 check03: Checking row and column sums
pick 8e39599 check04: Checking diagonal sums
pick 6396aa8 Removing TODO
pick 783031e Refactoring the main check function
pick 2492536 Updates to README.md
```

Then, save your changes with **Escape** + **:wq** + **Enter** to continue the rebase operation.

Git continues with a little bit of output to let you know things have succeeded:

```
Successfully rebased and updated refs/heads/wValidator.
```

Now, take a look at the log with `git log --oneline` and you'll see that Git has neatly reordered your commits, and added new hashes as well:

```
35aab2b (HEAD -> wValidator) Updates to README.md
3899829 Refactoring the main check function
c8d5335 Removing TODO
5d16107 check04: Checking diagonal sums
5c9e64d check03: Checking row and column sums
4018013 check02: Checking the array contains the correct values
f7a31a0 util06: Adding a function to check diagonals
851663d util05: Fixing comment indentation
6c857e4 util04: Adding a function to check column sums
5ad299c util03: Adding function to check row sums
2575920 Refactoring the range checking function
96fb378 util02: Adding function to check the range of values
55d4ded util01: Adding the checkSqaure function
ded7caa check01: checking that the 2D array is square
69670e7 Adding a new secret
```

I want to stress once again that these are *new* commits, not simply the old commits moved around. And it's not just the commits you moved around inside the instruction file that have new hashes: Every single commit from your rebase script has a new hash — *because they are new commits*.

Rewording commit messages

If you take a look at the `util01` commit message, you'll notice that it's misspelled as "Sqaure" instead of "Square". As a word nerd, I can't leave that the way it is. But I can quickly use interactive rebase to change that commit message.

Note: In my repo, the commit has the hash of `55d4ded`, while in your system, it will likely be different. Simply replace the hash below with the hash of the commit you want to rebase on top of — that is, the commit just before the one you want to change, and things will work just fine.

Execute the following to start another interactive rebase, indicating the commit you want to rebase on top of. In this instance, you want to rebase on top of the `check01: checking that the 2D array is square` commit:

```
git rebase -i ded7caa
```

When Vim comes up, you'll see the commit you'd like to change at the top of the list:

```
pick 55d4ded util01: Adding the checkSqaure function
```

Ensure your cursor is on that line, and type **cw** to cut the word `pick` and change to insert mode in Vim. In place of `pick`, type `reword` there, which tells Git to prompt you to reword this commit as it runs the rebase script.

When you're done, the very first line in the script should look as follows:

```
reword 55d4ded util01: Adding the checkSqaure function
```

Note: You're not fixing the commit message in this step; rather, you'll wait for Git to prompt you to do it when the rebase script runs.

Save your work with **Escape** + **:wq** + **Enter** and you'll immediately be put back into Vim. This time, Git's asking you to actually modify the commit message.

Press **i** to enter insert mode, cursor over to that egregious misspelling, change the word `checkSqaure` to `checkSquare`, and save your work with **Escape** + **:wq** + **Enter**.

Git completes the rebase and drops you back at the command line.

You can see that Git has changed the commit message for you by executing `git log --oneline` and scrolling down to find your new, rebased commit:

```
4f4e308 util01: Adding the checkSquare function
```

It's a small thing, to be sure, but it's a *nice* thing.

Squashing multiple commits

Now that you have your utility functions all arranged contiguously, you can proceed to squash these commits into one.

Again, you'll launch an interactive rebase session with the hash of the commit you want to rebase on top of. You want to rebase on top of the `Adding a new secret` commit, which is still `69670e7`. Remember: When you rebase *on top* of a commit, that commit doesn't change, so it still has the same hash as before. It's just the commits that follow that will get new hashes as each is rebased.

To start your adventure in squashing, execute the following to kick off another interactive rebase:

```
git rebase -i 69670e7
```

Once you're back in Vim, find the list of contiguous commits for the utility functions.

To squash a list of commits, find the first commit in the sequence you'd like to squash and leave that commit as it is. Then, on every subsequent line, change `pick` to `squash`. As Git executes this rebase script, each time it encounters `squash`, it will meld that commit with the commit on the previous line.

That's why you need to leave that first line unchanged: Otherwise, Git will squash *that* first commit into the previous commit, which isn't what you want. You want to squash this set of changes relevant to the utility functions as a nice tidy unit, not squash them into some random commit preceding them.

Note: You can use a bit of Vim-fu to speed things along here.

- Type **cw** on the first commit you want to squash (the `util02` one) and change `pick` to `squash`.
- Then press **Escape** to get back to command mode.
- Cursor down to the start of the next commit you want to squash, and type **.** - a period. This tells Git "Do that same thing again, only on this line instead."

Continue on this way for all of the utility function commits. When you're done, your rebase script should look like the following:

```
pick ded7caa check01: checking that the 2D array is square
pick 4f4e308 util01: Adding the checkSquare function
squash 421c298 util02: Adding function to check the range of
values
squash 96dc840 Refactoring the range checking function
squash 19e90e9 util03: Adding function to check row sums
squash c9d8aa3 util04: Adding a function to check column sums
squash 30f164a util05: Fixing comment indentation
squash 0bda95b util06: Adding a function to check diagonals
pick d34c59b check02: Checking the array contains the correct
values
pick d235bf9 check03: Checking row and column sums
pick 00212f3 check04: Checking diagonal sums
pick ca6f8df Removing TODO
pick a4a05c0 Refactoring the main check function
pick a351e8a Updates to README.md
```

Save your changes with **Escape** + **:wq** + **Enter** and you'll be brought into another instance of Vim. This is your chance to provide a single, clean commit message for your squash operation.

Vim helpfully gives you a bit of context here, as it lists the collection of commit messages from the squash operation for context:

```
# This is a combination of 7 commits.
# This is the 1st commit message:

util01: Adding the checkSquare function

# This is the commit message #2:

util02: Adding function to check the range of values

# This is the commit message #3:

Refactoring the range checking function

# This is the commit message #4:

util03: Adding function to check row sums

# This is the commit message #5:

util04: Adding a function to check column sums

# This is the commit message #6:

util05: Fixing comment indentation

# This is the commit message #7:

util06: Adding a function to check diagonals
```

You could choose to reuse some of the above content for the squash commit message, but in this case, simply type **gg** to ensure you're on the first line and **dG** to clear the edit buffer entirely.

Press **i** to enter insert mode, and add the following commit message, to sum up your squash effort:

```
Creating utility functions for Magic Square validation
```

Save your changes with **Escape** + **:wq** + **Enter** and Git will respond with a bit of output to let you know it's done. Execute `git log --oneline` to see the result of your actions:

```
858e215 (HEAD -> wValidator) Updates to README.md
d42dd03 Refactoring the main check function
499c6ac Removing TODO
7530a8f check04: Checking diagonal sums
c98bb17 check03: Checking row and column sums
eec2df9 check02: Checking the array contains the correct values
2207949 Creating utility functions for magic square validation
ded7caa check01: checking that the 2D array is square
69670e7 Adding a new secret
```

Nice! You've now squashed all of the `util` commits into a single commit with a concise message.

But there's still a bit of work to do here: You also want to rearrange and squash the `check0x` commits in the same manner. And that, dear reader, is the challenge for this chapter!

Challenge 1: More squashing

You'd like to squash all of the `check0x` commits into one tidy commit. And you *could* follow the pattern above, where you first rearrange the commits in one rebase and then perform the squash in a separate rebase.

But you can do this all in one rebase pass:

1. Figure out what your base ancestor is for the rebase.
2. Start an interactive rebase operation.
3. Reorder the `check0x` commits.
4. Change the `pick` rebase script command for `squash` on all commits from the `check02` commit, down to and including the `Refactoring the main check function` commit.
5. Save your work in Vim and exit.
6. Create a commit message in Vim for the squash operation.
7. Take a look at your Git log to see the changes you've made.

Challenge 2: Rebase your changes onto master

Now that you've squashed your work down to just a few commits, it's time to get wValidator back into the `master` branch. It's likely your first instinct is to merge `wValidator` back to `master`. However, you're a rebase guru by this point, so you'll rebase those commits on top of `master` instead:

1. Ensure you're on the wValidator branch.
2. Execute `git rebase` with `master` as your rebase target.
3. Crud — a conflict. Open README.md and resolve the conflict to preserve your changes, and move the changes to the `## Contact` section.
4. Save your work.
5. Stage those changes with `git add README.md`.
6. Continue the rebase with `git rebase --continue`.
7. Check the log to see where `master` points and where `wValidator` points.
8. Check out the `master` branch.
9. Execute `git merge` for `wValidator`. What's special about this merge that lets you avoid a merge commit?
10. Delete the `wValidator` branch.

If you get stuck or need any assistance, you can find the solution for these challenges inside the **challenges** folder for this chapter.

Key points

- `git rebase -i <hash>` starts an interactive rebase operation.
- Interactive rebases in Git let you create a "script" to tell Git how to perform the rebase operation
- The `pick` command means to keep a commit in the rebase.
- The `squash` command means to merge this commit with the previous one in the rebase.
- The `reword` command lets you reword a particular commit message.
- You can move lines around in the rebase script to reorder commits.
- Rebasing creates new commits for each original commit in the rebase script.
- Squashing lets you combine multiple commits into a single commit with a new commit message. This helps keep your commit history clean.

Where to go from here?

Interactive rebase is one of the most powerful features of Git because it forces you to think logically about the changes you've made, and how those changes appear to others. Just as you'd appreciate cloning a repo and seeing a nice, illustrative history of the project, so will the developers that come after you.

In the following chapter, you'll continue to use rebase to solve a terribly common problem: What do you do when you've already committed files that you want Git to ignore? If you haven't hit this situation in your development career yet, trust me, you will. And it's a beast to solve without knowing how to rebase!

Chapter 16: Gitignore After the Fact

By Chris Belanger

When you start a new software project, you might think that the prefab **.gitignore** you started with will cover every possible situation. But more often than not, you'll realize that you've committed files to the repository that you shouldn't have. While it seems that all you have to do to correct this is to reference that file in **.gitignore**, you'll find that this doesn't solve the problem as you thought it would.

In this chapter, you'll cover a few common scenarios where you need to go back and tell Git to ignore those types of mistakes. You're going to look at two scenarios to fix this locally: Forcing Git to assume a file is unchanged and removing a file from Git's internal index.

Getting started

To start, extract the repository contained in the starter **.zip** file inside the **starter** directory from this chapter's materials. Or, if you completed all of the challenges from the previous chapter, feel free to continue with that instead.

.gitignore across branches

Git's easy and cheap branching strategy is amazing, isn't it? But there are times when flipping between branches without a little forethought can get you into a mess.

Here's a common scenario to illustrate this.

Inside the **magicSquareJS** project, ensure you're on `master` with `git checkout master`.

Pull up a listing of the top-level directory with `ls` and you'll see a file named **IGNORE_ME**. Print the contents of the file to the command line with `cat IGNORE_ME` and you'll see the following:

```
Please ignore this file. It's unimportant.
```

Now assume you have some work to do on another branch. Switch to the `yDoublyEven` branch with the following command:

```
git checkout yDoublyEven
```

Pull up a complete directory listing with the following command:

```
ls -la
```

You'll see that there's a **.gitignore** there, but there's no sight of the IGNORE_ME file. Looks like things are working properly so far.

Open up the **.gitignore** file in an editor and you'll see the following:

```
IGNORE_ME*
```

It looks like you're all set up to ignore that **IGNORE_ME** file. Therefore, if you create an **IGNORE_ME** file, Git should completely ignore it, right? Let's find out.

Create a file named **IGNORE_ME** in the current directory, and add the following text to that file:

```
Please don't look in here
```

Save your changes and exit.

You can check that Git is ignoring the file by executing `git status`:

```
nothing to commit, working tree clean
```

So far so good. It looks like everything is working as planned.

Now switch back to `master` with the following command:

```
git checkout master
```

And at this point, Git shouldn't have anything to complain about, since it's ignoring that **IGNORE_ME** file. But open up that **IGNORE_ME** file and see what's inside:

```
Please ignore this file. It's unimportant.
```

Wait — shouldn't Git have ignored the change to that file and preserved the original `Please don't look in here` text you added on the other branch? Why did Git overwrite your changes, if it should have been ignoring any changes to this file?

Sounds like you should have a look at the **.gitignore** file in `master` to see what's going on. There *is* a **.gitignore** file in `master`, right?

Pull up a full directory listing with `ls -la` and you'll see that, in fact, there *is* no **.gitignore** on the `master` branch:

```
.
├── .DS_Store
├── .git
├── .tools-version
├── IGNORE_ME
├── LICENSE
├── README.md
```

```
├── SECRETS
├── css
├── img
├── index.html
└── js
```

Oh. Well, that seems easy to fix. You'll just add a reference to **IGNORE_ME** to the **.gitignore** on `master` and everything should just sort itself out.

Create a **.gitignore** file in the current directory, and add the following to it:

```
IGNORE_ME*
```

Save your changes and exit. So Git should start ignoring any changes to **IGNORE_ME** now, right? It seems like you're safe to put your original change back in place.

Open up **IGNORE_ME** in an editor, and replace the contents of that file with the original content you wanted in there in the first place:

```
Please don't look in here
```

Save your changes and exit. Execute a quick `git status` to check that Git is actually ignoring that file, as you'd hoped:

```
git status
```

You'll see the following in your console, showing that Git is absolutely *not* ignoring that file:

```
On branch 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:   IGNORE_ME

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

    .gitignore

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

Wait, what? You told Git to ignore that file, yet Git is obviously still tracking it. What's going on here? Doesn't putting something in **.gitignore**, gee, I don't know, tell Git to *ignore* it?

This is one of the more frustrating things about Git; however, once you build a mental model of what's happening, you'll see that Git's doing exactly what it's supposed to. And you'll also find a way to fix the situation you've gotten yourself into.

How Git tracking works

When you stage a change to your repository, you're adding the information about that file to Git's **index**, or **cache**. This is a binary structure on disk that tracks everything you've added to your repository.

When Git has to figure out what's changed between your working tree and the staged area, it simply compares the contents of the index to your working tree to determine what's changed. This is how Git "knows" what's unstaged and what's been modified.

But if you *first* add a file to the index, and *later* add a rule in your **.gitignore** file to ignore this file, this won't affect Git's comparison of the index to your working tree. The file exists in the index and it also exists in your working tree, so Git won't bother checking to see if it should ignore this file. Git only performs **.gitignore** filtering when a file is in your working tree, but *not yet* in your index.

This is what's happening above: You added the **IGNORE_ME** file to your index in `master` *before* you got around to adding it to the **.gitignore**. So that's why Git continues to operate on **IGNORE_ME**, even though you've referenced it in the **.gitignore**.

In fact, there's a handy command you can use to see what Git is currently ignoring in your repository. You've already used it quite a lot in this book, believe it or not! It's simply `git status`, but with the `--ignored` flag added to the end.

Execute this now to see what Git is ignoring in your repository:

```
git status --ignored
```

My output looks like the following:

```
On branch 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:   IGNORE_ME

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

    .gitignore

Ignored files:
  (use "git add -f <file>..." to include in what will be
committed)

    .DS_Store
    js/.DS_Store

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

So Git is ignoring .DS_Store files, as per my global **.gitignore**, but it's not ignoring **IGNORE_ME**. Fortunately, there are a few ways to tell Git to start ignoring files that you've already added to your index.

Updating the index manually

If all you want is for Git to ignore this file, you can update the index yourself to tell Git to assume that this file will never, ever change again. That's a cheap and easy workaround.

Execute the following command to update the index and indicate that Git should assume that when it does a comparison of this file, the file hasn't changed:

```
git update-index --assume-unchanged IGNORE_ME
```

Git won't give you any feedback on what it's done with this command, but run `git status --ignored` again and you'll see the difference:

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

    .gitignore

Ignored files:
  (use "git add -f <file>..." to include in what will be
```

```
committed)

    .DS_Store
    js/.DS_Store

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

Git isn't ignoring it, *technically*, but for all intents and purposes, this method has the same effect. Git won't ever consider this file changed for tracking purposes.

To prove this to yourself, modify **IGNORE_ME** and add some text to the end of it, like below:

```
Please don't look in here. I mean it.
```

Save your changes, exit out of the editor, and then run `git status --ignored` again. You'll see that Git continues to assume that that file is unchanged.

This is useful for situations where you've added placeholders or temporary files to the repository, but you don't want Git tracking the changes to those temporary files during development. Or maybe you just want Git to ignore that file for now, until you get around to fixing it in a refactoring sprint later.

The issue with this workaround is that it's only a *local* solution. If you are working on a distributed repository, everyone else would have to do the same thing in their own clone if they want to ignore that file. Telling Git to assume a file is unchanged only updates the index on your local system. This means these file changes won't make it into a commit — but it also means that anyone else cloning this repo will still run into the same issues you did.

In fact, you might prefer to remove this file from the index entirely, instead of just asking Git to turn a blind eye to it.

Removing files from the index

When you implicitly or explicitly ask Git to start tracking a file, Git dutifully places that file in your index and starts watching for changes. If you're quite certain that you don't want Git to track this file anymore, you can remove this file from the index yourself.

After you remove a file from the index, Git follows the natural progression of checking the working tree against the index for changes, then looking to the **.gitignore** to see if it should exclude anything from the changeset.

You've already run across a command to remove files from Git's index: `git rm`. By default, `git rm` will remove files from *both* the index and your working tree. But in this case, you don't want to remove the file in your working tree — you want to keep it.

To remove a file from the index but leave it in your working tree, you can use the `--cached` option to tell Git to remove this file from the index only.

Execute the following command to instruct Git to remove **IGNORE_ME** from the index. Git will, therefore, stop tracking it:

```
git rm --cached IGNORE_ME
```

Git responds with a simple confirmation:

```
rm 'IGNORE_ME'
```

To see that this has worked, run `git status --ignored` again:

```
On branch master
Changes to be committed:
  (use "git reset HEAD <file>..." to unstage)

    deleted:    IGNORE_ME

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

    .gitignore

Ignored files:
  (use "git add -f <file>..." to include in what will be
committed)

    .DS_Store
    IGNORE_ME
    js/.DS_Store
```

IGNORE_ME now shows that it's both `deleted` and `ignored`. How can that be?

If you think about Git's perspective for a moment, this makes sense: `git status` compares the staging area, or index, to `HEAD` to see what the next commit should be. Git sees that **IGNORE_ME** is no longer in the index. Whether this file exists on disk is irrelevant to Git at this moment. So it sees that the next commit would delete **IGNORE_ME** from the repository.

Besides that, including the `--ignored` option on `git status` builds a list of what Git now knows to ignore, based on any files in your working tree that match any filters in the **.gitignore**.

IGNORE_ME is not in your current index, so when Git runs its ignore filter, it sees that you have a file named **IGNORE_ME** on disk and that file isn't present in your current index.

However, you've put this filter in your **.gitignore**, so Git adds this file to its list of files to ignore. Hence, **IGNORE_ME** is both in deleted status (as far as the index is concerned) and ignored status (as far as your **.gitignore** is concerned).

Since this seems to have cleared up the situation, you can now create your next commit. But wait — aren't you forgetting something? Something that got you into this mess in the first place?

Ah right — **.gitignore** is still untracked. Stage that file now:

```
git add .gitignore
```

And commit this change before you forget again:

```
git commit -m "Added .gitignore and removed unnecessary file"
```

Just remember that if someone else clones the repo after you've pushed this commit, they'll also lose that file in their clone. As long as that's your intent, that's fine.

Now, remember that this doesn't remove *all* traces of your file — there's still a whole history of commits in your repository that have this file fully intact. If someone really wanted to, they could go back in history and find what's inside that file.

To see this, run `git log` on the file in question:

```
git log -- IGNORE_ME
```

The second entry in that log shows the following:

```
commit 7ba2a1012e69c83c4642c56ec630cf383cc9c62b
Author: Yasmin <yasmin@example.com>
```

```
Date:   Mon Jul 3 17:34:22 2017 +0700
an
    Adding the IGNORE_ME file
```

Well, that doesn't seem to be a *huge* deal. So what if people can see that you added a file you later removed from the repository?

In this case, it's not that important. But often, people commit massive zip or binary files to a repo, and don't realize it until people complain about how long it takes to clone a repo to their local system.

More critically, what if you'd accidentally committed a file with API keys, passwords or other secrets inside? Then you *absolutely do care* about making sure you've purged the repository of any history about this file. If someone were to get your API keys or other secrets, they potentially have unlimited, unfettered access to some of your systems. Whoops.

Rebasing isn't always the solution

Assume you don't want anyone to know about the existence of **IGNORE_ME**. You've already learned one way to rewrite the history of your repository: Rebasing. But will this solve your current issue?

To see why rebasing isn't a great way to solve this problem, you'll work through an interactive rebase on the current repository. This will show you the situations where `git rebase` might not be the best choice to rewrite history.

You know that Yasmin added **IGNORE_ME** back in commit hash `7ba2a1012e69c83c4642c56ec630cf383cc9c62b`, as you saw above. So all you have to do is drop that particular commit, rebase everything else on top of the ancestor commit, and everything would be *just fine*, right?

But first: Did that commit *only* add **IGNORE_ME**? Or did it add any other files? You need to know that before you commit. You can't always trust someone's commit message.

Have a look at the patch for this commit to see what it actually contains:

```
git log -p -1 7ba2a10
```

You should see the following:

```
commit 7ba2a1012e69c83c4642c56ec630cf383cc9c62b
Author: Yasmin <yasmin@example.com>
Date:   Mon Jul 3 17:34:22 2017 +0700

    Adding the IGNORE_ME file

diff --git a/IGNORE_ME b/IGNORE_ME
new file mode 100644
index 0000000..28c0f4f
--- /dev/null
+++ b/IGNORE_ME
@@ -0,0 +1 @@
+Please ignore this file. It's unimportant.
```

OK, it seems that commit only added that file, as it said in the commit. Theoretically, you should be able to drop that commit from the history of the repo and everything should be *just* fine.

Start an interactive rebase with the following:

```
git rebase -i 7ba2a10^
```

The caret ^ at the end of the commit hash means "start the rebase operation at the commit just prior to this one."

Git presents you with the interactive script for this rebase:

```
pick 7ba2a10 Adding the IGNORE_ME file
pick 883eb6f Adding methods to allow editing of the magic square
pick e632550 Adding ID to <pre> tag
pick f28af7a Adding ability to validate the inline square
pick c2cf184 Wiring up the square editing and validation
.
.
.
pick 5d026f0 Added .gitignore and removed unnecessary file
```

All you need to do is drop that first commit, right? Using your git-fu skills, type **cw** to cut the `pick` command on that first line, and in its place, put `drop`. Your rebase script should look like the following:

```
drop 7ba2a10 Adding the IGNORE_ME file
pick 883eb6f Adding methods to allow editing of the magic square
pick e632550 Adding ID to <pre> tag
pick f28af7a Adding ability to validate the inline square
pick c2cf184 Wiring up the square editing and validation
.
```

```
.
.
pick 5d026f0 Added .gitignore and removed unnecessary file
```

Press **Escape** to exit out of insert mode, and type **:wq** followed by **Enter** to save your work and carry on with the interactive rebase.

…and, of course, nothing is ever as simple as it seems. You've run into a merge conflict already, on **index.html**:

```
Auto-merging index.html
CONFLICT (content): Merge conflict in index.html
error: could not apply f985ed1... Centre align everything
```

Oh, right. Because Git is actually replaying all of the other commits as part of the rebase, you'll encounter merge conflicts in files that aren't related to **IGNORE_ME**.

What you failed to take into consideration is the ancestor of `7ba2a10 Adding the IGNORE_ME file` — and what's happened in the repository since then.

Execute the following command to see the full gory details of the origins of this commit:

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

Scroll *way* down and you'll see commit `69670e7 Adding a new secret`:

```
.
.
.
| * | | e632550 Adding ID to <pre> tag
| * | | 883eb6f Adding methods to allow editing of the magic
square
| |/ /
* | | 7ba2a10 Adding the IGNORE_ME file
* | | 32067b8 Adding the structure to the generator
|/ /
* | 69670e7 Adding a new secret
.
.
.
```

`69670e7` is the ancestor of `7ba2a10`. And a lot has happened in the repository since that point. So when Git rewinds the history of the repository, it has to go all the way back to that ancestor and replay *every* commit that's a descendant of that ancestor and rebase it on top of `69670e7` — even commits that you've already merged back to `master`. Ugh. This really isn't what you bargained for, is it?

You *could* go through each of these commits and resolve them, but that's a tremendous amount of work, and quite a bit of risk, just to get rid of a single file.

Abort this rebase in progress with the following command:

```
git rebase --abort
```

This resets your staging and working environment back to where you were before.

> **Note**: For the purists out there, your working and staging area never actually changed during the rebase. Rebasing happens in a temporary detached HEAD space, which you can think of as a "virtual" branch that isn't spliced onto your repo until the rebase is complete. Aborting a rebase simply throws away that temporary space and puts you back into your unchanged working and staging area.

This isn't a scalable solution — not in the least. There's a better way to do this, and it's known as `git filter-branch`.

Using filter-branch to rewrite history

Let's put the issue with IGNORE_ME aside for the moment; you'll come back to it at the end of the chapter. Right now, you'll work through an issue with a similar file, **SECRETS**, that plays out the dreaded scenario above where you've committed files or other information that you never wanted to be public.

Print out the contents of the **SECRETS** file with the following command:

```
cat SECRETS
```

You'll see the following:

```
DEPLOY_KEY=THIS_IS_REALLY_SECRET
RAYS_HOTTUB_NUMBER=012-555-6789
```

Can you *imagine* the chaos if those two pieces of information hit the streets? You'll need to clean up the repository to remove all traces of that file — and also make sure the repository has been rewritten to remove any indication that this file was *ever* there in the first place.

The `filter-branch` command in Git lets you programmatically rewrite your repository. It's similar to what you tried to do with the interactive rebase, but it's far more flexible and powerful than trying to tweak things manually during an interactive rebase.

Although there are lots of ways to run `filter-branch`, you'll take the most direct route to remove this file: Rewrite your repository's staging area, or index.

A quick review, first: Do you recall how to remove a file from the index? That's right — `git rm --cached` removes the file from your staging area, as opposed to your working area. Remember this; you'll need it in just a moment.

There's another option to `git rm` that you'll need to know: `--ignore-unmatch`.

To see why you need this option, execute the following command at the command line to try to remove a non-existent file from the index:

```
git rm --cached -- NoFileHere
```

Git will respond with a `fatal` error:

```
fatal: pathspec 'NoFileHere' did not match any files
```

Since this is a fatal error, Git stops in its tracks and returns with what's known as a non-zero exit status; in other words, it errors out.

To prove this even further, execute the following stacked Bash command, which will print `success!` if the first command succeeds:

```
git rm --cached -- NoFileHere && echo 'success!'
```

Git again responds with the single fatal error and halts; `echo 'success!'` is never executed. It's clear that if `git rm` doesn't match on a filename, it's done and halts execution immediately.

To get around this, `--ignore-unmatch` will tell Git to report a zero exit status — that is, a successful completion — even if it doesn't find any files to operate on. To see this in action, execute the following stacked Bash command:

```
git rm --cached --ignore-unmatch NoFileHere && echo 'success!'
```

You'll see `success!` printed to the console, showing that `git rm` exited successfully.

Now — to put this knowledge to work.

Execute the following command to run `git filter-branch` to remove the offending file:

```
git filter-branch -f --index-filter 'git rm --cached --ignore-
unmatch -- SECRETS' HEAD
```

Taking that long command one bit at a time:

- You execute `git filter-branch` to tell Git to start rewriting the repository history.
- The `-f` option means "force"; this tells Git to ignore any internally-cached backups from previous operations. If you routinely use `filter-branch`, you'll want to use the `-f` option to avoid Git reminding you every time you run `filter-branch` that you have an existing backup from a previous operation.
- You next specify the `--index-history` option to tell Git to rewrite the index, instead of rewriting your working tree directly (more on that later).
- You then specify the filter, or command, you want to run on each matching commit as Git rewrites history. In this case, you're performing `git rm --cached` to look up files in the index. `--ignore-unmatched` prevents Git from bailing out of `filter-branch` if it doesn't match any files. Finally, you indicate you want to remove the **SECRETS** file.
- The final option indicates the revision list to operate on. Providing a single value here, in this case, `HEAD`, tells Git to apply `filter-branch` to all revisions from `HEAD` to as far back in history as Git can go with this commit's ancestors.

Git spits out multiple lines of output that tell you what it's doing. Here's one line from my output; yours may be slightly different:

```
Rewrite f28af7aad4f77da8deb28f1e0eb93b85ee755b43 (20/38) (1
seconds passed, remaining 0 predicted)    rm 'SECRETS'
```

Git has stepped through every commit from `HEAD` back in time, performed the specified `git rm` command, and then re-committed the change. To prove this, look for the **SECRETS** file:

```
git log -- SECRETS
```

You'll get nothing back, telling you that Git's log knows nothing about this **SECRETS** file you're asking for.

Now, it seems like you've removed every single trace of this file, but there's one small clue that might tell someone you've removed something from the repository.

The original commit that added this file is still around. Execute `git log --oneline --graph --decorate`, scroll down, and you'll see the original commit that added this secret file:

```
dcbdf0c Adding a new secret
```

Then, look at the patch of the commit using the following command:

```
git log -p -1 dcbdf0c
```

Git shows you the metadata, but the patch itself is empty:

```
commit dcbdf0c2b3b5cf06eafd5dc6e441c8ab3a1d2ed5
Author: Will <will@example.com>
Date:   Mon Jul 3 14:10:59 2017 +0700

    Adding a new secret
```

Although no one can tell what the secret *was*, it would be nice to get rid of that commit entirely since it's empty.

That's as simple as using another option to `filter-branch`: `--prune-empty`. If your author had had the foresight to tell you to use it in the first place, then you could have just tacked this on as an option to your original command.

But, Git is not a vengeful deity; you can run `filter-branch` again to clean things up. Execute the following command to run through your repository again and remove any "empty" commits:

```
git filter-branch --prune-empty -f HEAD
```

This simply runs through your repository, removing any commits that have an empty patch. Again, the `-f` command forces Git to perform `filter-branch`, disregarding any previous backups it may have saved from previous `filter-branch` operations.

Pull up your log again with `git log --oneline --decorate --graph` and scroll around; the commit is now gone.

Now that you're an expert on rewriting the history of your repository, it's time for your challenge for this chapter. It will bring things full circle and deal with that poor little **IGNORE_ME** file you were working with earlier.

Challenge: Remove IGNORE_ME from the repository

Now that you've learned how to eradicate any trace of a file from a repository, you can go back and remove all traces of **IGNORE_ME** from your repository.

You previously removed all traces of **SECRETS** from your repository, but that took you two steps. The challenge here is to do the same in one single command:

- Use `git filter-branch`.
- Use `--index-filter` to rewrite the index.
- You can use a similar `git rm` command, but remember, you're filtering on a different file this time.
- Use `--prune-empty` to remove any empty commits.
- Remember that you want to apply this to all commits, starting at `HEAD` and going back.
- You'll need to use `-f` to force this `filter-branch` operation, since you've already done a `filter-branch` and Git has stored a backup of that operation for you.

> **Note**: If Git balks, check that the positioning of your options is correct in your command.

If you want to check your answer, or need a bit of help, you can find the answer to this challenge in the **challenge** folder included with this chapter.

Key points

- **.gitignore** works by comparing files in the staging area, or index, to what's in your working tree.
- **.gitignore** won't filter out any files already present in the index.
- `git status --ignored` shows you the files that Git is currently ignoring.
- `git update-index --assume-unchanged <filename>` tells Git to always assume that the file contained in the index will never change. This is a quick way to work around a file that isn't being ignored.
- `git rm --cached <filename>` removes a file from the index but leaves the original file in your working tree.
- `git rm --cached --ignore-unmatch <filename>` will succeed, returning an exit code of `0`, if `git rm` doesn't match on a file in the index. This is important when you use this command in conjunction with `filter-branch`.
- `git filter-branch -f --index-filter 'git rm --cached --ignore-unmatch -- <filename>' HEAD` will modify any matching commits in the repository to remove `<filename>` from their contents.
- The `--prune-empty` option will remove any commits from the repository that are empty after your `filter-branch`.

Where to go from here?

What you've learned in this chapter will usually serve you well when you've committed something to your repository that you didn't intend to be there.

The reverse case is fairly common, as well: You don't have something in your repository, but you know that bit of code or that file exists in another branch or even in another repository.

Just as you can selectively remove changes from your repository with `filter-branch`, you can also pull in very specific changes to your current branch with `git cherry-pick`, which is covered in the next chapter.

Chapter 17: Cherry Picking

This is an early access release of this book. Stay tuned for this chapter in a future release!

Chapter 18: The Many Faces of Undo

By Chris Belanger

One of the best aspects of Git is the fact that it remembers *everything*. You can always go back through the history of your commits with `git log`, see the history of your team's activities and cherry-pick commits from other places.

But one of the most frustrating aspects of Git is also that it remembers *everything*. At some point, you'll inevitably create a commit that you didn't want or that contains something you didn't intend to include.

While you can't rewrite shared history, you *can* get your repository back into working order without a lot of hassle.

In this chapter, you'll learn how to use the `reset`, `reflog` and `revert` commands to undo mistakes in your repository. While doing so, you'll find a new appreciation for Git's infallible memory.

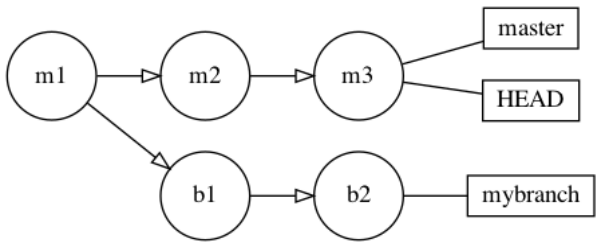
Working with git reset

Developers quickly become familiar with the `git reset` command, usually out of frustration. Most people see `git reset` as a "scorched earth" approach to fix a repository that's messed up beyond repair. But when you delve deeper into how the command works, you'll find that `reset` can be useful for more than a last-ditch effort to get things working again.

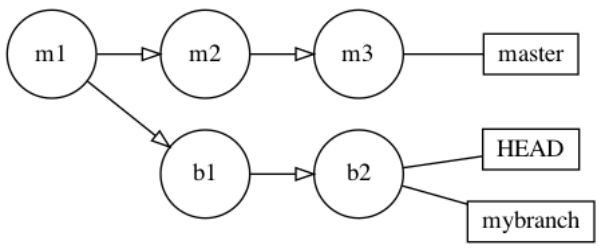
To learn how `reset` works, it's worth revisiting another command you're intimately familiar with: `checkout`.

Comparing reset with checkout

Take the example below, where the branch `mybranch` is a straightforward branch off of `master`:

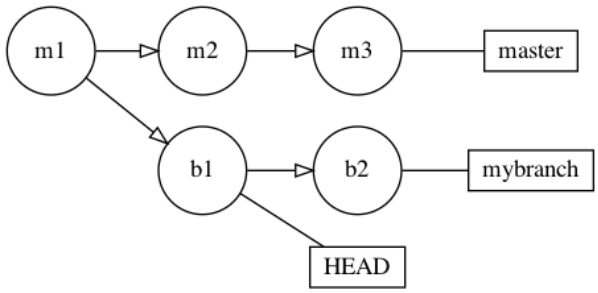


In this case, you're working on `master`, and `HEAD` is pointing to the hash of the last commit on the `master` branch. When you check out a branch with `checkout mybranch`, Git moves the `HEAD` label to point to the most recent commit on the branch:



So `checkout` simply moves the `HEAD` label between commits. But instead of specifying a branch label, you can also specify the hash of a commit.

For example, assume that instead of `checkout mybranch`, you wanted to check out the commit just before the one referenced by `HEAD` — in this case, `b1`:

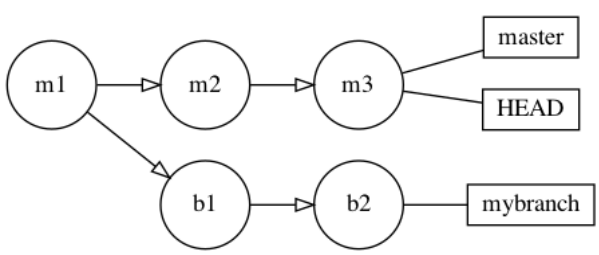


So your working directory now reflects the state of the repository represented by commit `b1`. This is a **detached `HEAD` state**, which simply means that `HEAD` now points to a commit that has no other label pointing to it.

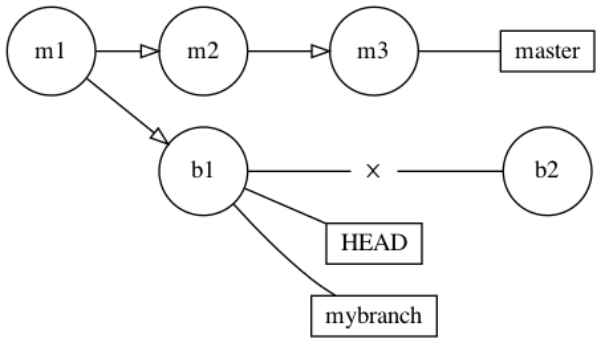
> **Note**: This is a weird (but totally permissible) state from Git's perspective. Git's normal workflow is to either work from the tip of a branch, denoted by the branch's name label, or to work from some other named label in the repository. A detached HEAD state is useful when you want to view the state of the repository at some earlier point in time, but it's not a state you'd be in as part of a normal workflow.

You've seen that `checkout` simply moves `HEAD` to a particular commit. `reset` is similar, but it also takes care of moving the branch's label to the *same* commit instead of leaving the branch label where it was. `reset`, in effect, returns your working environment — including your branch label — to the state a particular commit represents.

Consider again the example above, with a simple branch, `mybranch`, off of `master`:



This time, you execute a `reset` command with a target commit of `b1`:



Both `HEAD` and `mybranch` have now moved back to `b1`. This means you've effectively discarded the original tip commits of the `mybranch` branch, and stepped back to the `b1` commit.

But what happens to the tip commit that's now hanging from the `b1` label?

From Git's perspective, it doesn't exist anymore. Git will collect it with its regular garbage collection cycle, and any commits you make on `mybranch` will now stem from the `b1` commit as their ancestor.

In this way, `reset` is quite useful when you're trying to "roll back" commits you've made, to get to an earlier point in your repository history. `reset` has a lot of different use cases for it, and with it several options to learn about that change its behavior.

Working with the three flavors of reset

Remember that Git needs to track three things: your working directory, the staging area and the internal repository index. Depending on your needs, you can provide parameters to `reset` to roll back either all those things or just a selection:

- **soft**: Leaves your working directory and staging area untouched. It simply moves the reference in the index back to the specified commit.
- **mixed**: Leaves your working directory untouched, but rolls back the staging area and the reference in the index.
- **hard**: Leaves nothing untouched. This rolls your working directory, your staging area and the reference in the index back to the specified commit.

To understand `reset` more fully, you'll work through a few scenarios in your repository to see how it affects each of the three areas above.

Start by extracting the compressed repository from the **starter** directory to a convenient location on your machine, then navigating into that directory from the command line.

Testing git reset --hard

`git reset --hard` is most people's first introduction to "undoing" things in Git. The `--hard` option says to Git, "Please forget about the grievous things I've done to my repository and restore my entire environment to the commit I've specified".

The default commit for `git reset` is `HEAD`, so executing `git reset --hard` is the equivalent of saying `git reset HEAD --hard`.

To see how this can get you out of a sticky situation, you'll make some rather ill-considered changes to your repository, check the state of the index and staging area then execute `git reset --hard` to see how Git can "undo" that mess for you.

Removing an utterly useless directory

Start by going to the command line and navigating to the root directory of your repository. Execute the following command to get rid of that pesky **js** directory, which doesn't look very important:

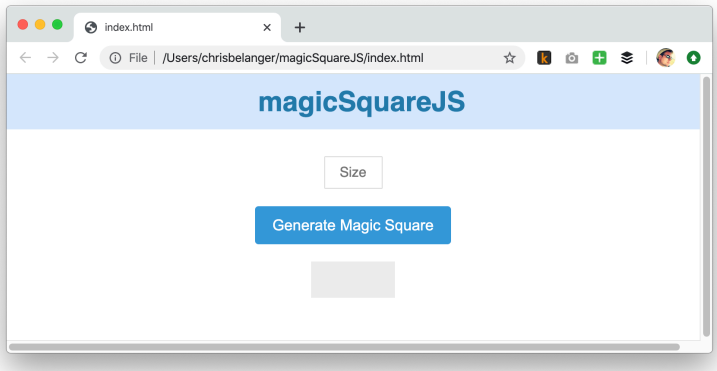
```
git rm -r js
```

This uses the `git rm` command to not only delete the directory, but also to update Git's staging area with the files deleted as well.

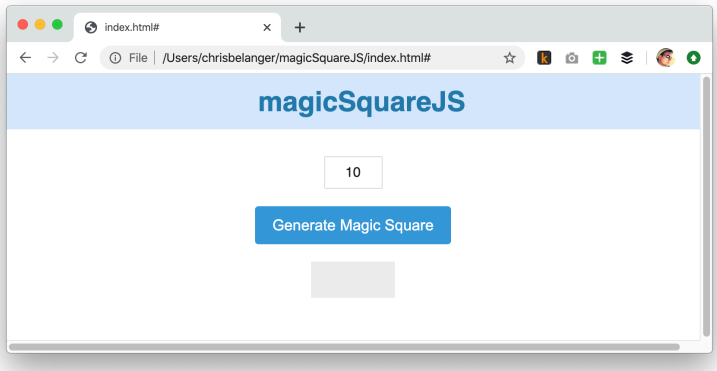
You're ultra-confident you don't need that directory, nor do you even need to test your changes (does anyone even *use* JavaScript anymore?), so you also commit your changes to the repository:

```
git commit -m "Deletes the pesky js directory"
```

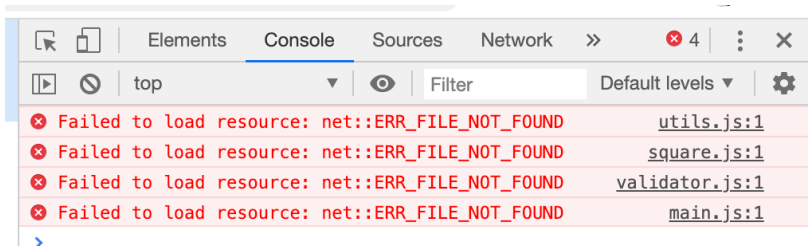
Now, open **index.html** in a browser and you'll find that the site still looks great, despite the loss of the **js** directory:



But enter a number in the field and click the **Generate Magic Square** button and you'll find that nothing happens at all:



Even worse, take a look at the developer console of the browser and you'll see the following JavaScript errors:



Whoops! Looks like you needed that directory after all. But you've gone and committed your work, haven't you? Yes, unfortunately, you have.

Execute the following command to see the commit history of your repository:

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

Sadly, you see your ill-advised commit sitting proudly at the tip of `master`:

```
* 6c5ecf1 (HEAD -> master) Deletes the pesky js directory
```

Oh no, no, no, no, no. How will you get that directory back now?

The first option is to panic, delete everything you're working on, and re-clone the repository.

Luckily, there's really no need to go to those lengths. Git remembers *everything*, so it's easy to get back to a previous state.

Restoring your directory

In this case, you want to return to the last commit before you made your blunder. You don't even need to know the commit hash; you can provide *relative* references to `git reset` instead.

Press **Q** to exit the previous context. Then execute the following command to return your working directory, your staging area and your index to the previous commit:

```
git reset HEAD^ --hard
```

Here, the caret character, `^`, means "the first immediate ancestor commit just before `HEAD`".

Look at your working directory and you'll see that your **js** directory has reappeared. To be sure, open **index.html** in a browser and you'll see that your magic square generator now functions as it did before.

Whew! You dodged that bullet.

This situation allowed you to completely blow away everything in your working directory. But what if you had something in there that you wanted to keep?

That's where the other parameters for `git reset` come to the rescue.

Trying out git reset --mixed

Imagine that you're working on another software project. You're up late, the coffee ran out hours ago and you're tired. That never happens in real life, of course, but bear with me.

You want to create a temporary file to hold some login information. You're a responsible developer, so you'd *never* commit that sensitive information to the repository.

Create a file named **SECRETS** in your working directory with the command below:

```
touch SECRETS
```

Then add some ultra-secret information with the `echo` command:

```
echo 'password=correcthorsebatterystaple' >> SECRETS
```

Now, assume some time has passed and you've made lots of progress on your website. You want to get to bed as soon as you can, so you use the shortcut `git add` to add all your changes to the staging area:

```
git add .
```

And then you commit your changes, like the responsible developer you are:

```
git commit –m "Adds final styling for website"
```

No sooner have you pressed the **Enter** key, when you realize — with a start — that you committed **SECRETS**, too!

Well, you're fully awake now, thanks to that burst of adrenaline, and you're in quite a pickle.

Fortunately, you haven't pushed your changes to the remote yet, so that's one less mess to untangle. But you'd like to get **SECRETS** out of the commit history so you don't look like a total fool.

Removing your unwanted commit

You *could* use `git reset HEAD^ --hard`, as above, but that would blow away hours of hard work. Instead, use `git reset --mixed` to reset the commit index and the staging area, but leave your working directory alone.

This will let you add the **SECRETS** file to your .gitignore — which you should have done in the first place, silly — and preserve all your work.

Execute the following command to reset only the index and the staging area to the previous commit:

```
git reset HEAD^ --mixed
```

Now execute `git status` to see that **SECRETS** is now untracked:

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

There you are — you're back to the state right before you executed `git add .`. You're now free to add **SECRETS** to your .gitignore (lesson learned), then stage and commit all your hard work.

Execute the following commands to do just that:

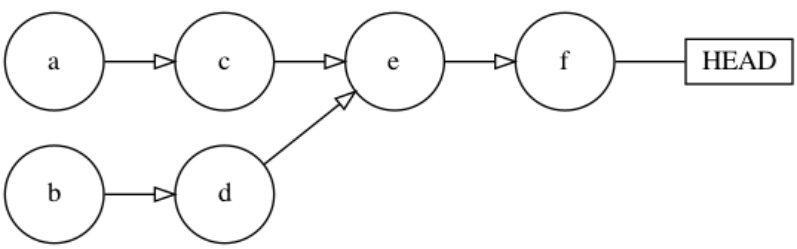
```
echo SECRETS >> .gitignore
git add .gitignore
git commit -m "Updates .gitignore"
```

Do you *always* have to use HEAD^; that is, do you always have to go to the previous commit?

No, not at all! It's what you'll use most of the time, in practice, but you can specify any commit in Git's history by using its commit hash.

There are other ways to specify a commit relative to `HEAD`. These are handy when you're going back farther than one level in history, or you're dealing with a commit that has more than one parent, like merge commits.

Take the image below, which shows a simple two-branch scenario that's been merged. In this diagram, from the perspective of time, commit `a` occurred first, commit `b` second, commit `c` happened third and so on.



Here's how you can use relative references to get to each point in this tree:

HEAD^1: References the first immediate ancestor of this commit in history: commit `e`. This is the simple case, since the commit referenced by `HEAD`, `f`, only has one ancestor. This is equivalent to the shorthand: `HEAD^`.

HEAD^^: References the immediate ancestor of the immediate ancestor of this commit in history: commit `c`. Because commit `e` has two ancestors — `c` and `d` — Git chooses the "oldest" or first ancestor of `e`: `c`.

HEAD^^2: References the second immediate ancestor of the immediate ancestor this commit in history: commit `d`. Because commit `e` has two ancestors — `c` and `d` — specifying `^2` (parent #2) chooses the "newer" or later ancestor of `e`: `d`.

HEAD^^^: References the first immediate ancestor, of the first immediate ancestor, of the first immediate ancestor of this commit in history: `a`. Here, you go back three generations, always following the "older" path first.

HEAD^^2^: References the first immediate ancestor, of the second immediate ancestor, of the first immediate ancestor of this commit in history: commit `b`. The first `^` tells Git to look back at the first ancestor of this commit. The next `^2` tells Git to look at the newer ancestor, and the final `^` looks to the ancestor of *that* commit: `b`.

If you'd like more information about trees and graph traversals, read up on the finer details of relative references, such as `HEAD~`, in the **Specifying Revisions** section of Git's `man` pages. This contains a more complex commit tree and instructions on how to traverse this tree with relative references.

Using git reset --soft

If you like to build up commits bit by bit, staging changes as you make them, then you may encounter a situation where you've staged various changes and committed them prematurely.

In that situation, use `git reset --soft` to roll back that commit while leaving your meticulously-built staging area intact.

You've dodged that bullet with the **SECRETS** file, but now you have a few more changes to make. You have two files to add as part of this commit: a configuration file and a change to README.md that explains how to set all the parameters in the configuration file.

Create the configuration file first:

```
touch setup.config
```

Now, stage that change:

```
git add setup.config
```

Next, execute the following command to add a line of text to the end of **README.md**:

```
echo "For configuration instructions, call Sam on 555-555-5309
any time" >> README.md
```

Making a mistake

Just before you add that to the staging area, Will and Xanthe call you excitedly with their plans for their next big project: to create a — wait for it — magic triangle generator. You humor them for a while, then turn your attention back to your project.

Did you add everything to the staging area? You're pretty sure you did, so you commit what's in the staging area:

```
git commit -m "Adds configuration file and instructions"
```

However, your keen eye notices the output message from Git:

```
[master c416751] Adds configuration file and instructions
 1 file changed, 0 insertions(+), 0 deletions(-)
 create mode 100644 setup.config
```

Zero insertions… that doesn't make sense. Wait, did you stage that change to **README.md**? No, you didn't, because you were distracted by Will and Xanthe.

Cleaning up your commit

So now, you need to clean up that commit so it includes both the change to **README.md** and the addition of **setup.config**.

All that's missing is that small change from **README.md**, so `git reset --soft` will roll back your commit and let you stage and commit that one change.

Execute the following to do a soft reset to the previous commit:

```
git reset HEAD^ --soft
```

Stage that small change from **README.md**:

```
git add README.md
```

Now, you can commit those changes again:

```
git commit -m "Adds configuration file and instructions"
```

Did it work? The output from Git confirms it did:

```
[master 297be58] Adds configuration file and instructions
 2 files changed, 2 insertions(+)
 create mode 100644 setup.config
```

You can use `git log` to see the actual contents of that commit with the following command:

```
git log -p -1
```

The output tells you that yes, you've committed both **config.setup** and that change to **README.md**:

```
diff --git a/README.md b/README.md
index 331487d..fb18f7c 100644
.
.
.
+For configuration instructions, call Sam on 555-555-5309 any
time
diff --git a/setup.config b/setup.config
new file mode 100644
index 0000000..e69de29
```

There you go. You were able to salvage your carefully-crafted staging area without having to start over. Nice!

So that wraps up the situations where you created a commit that you didn't want in the first place. But what about the reverse situation, where you got rid of a commit that you *didn't* want to lose?

Using git reflog

You know that Git remembers *everything*, but you probably don't realize just how deep Git's memory goes.

Head to the command line and execute the following command:

```
git reflog
```

You'll get a *ton* of output. Here are the top few lines of mine:

```
297be58 (HEAD -> master) HEAD@{0}: commit: Adds configuration
file and instructions
6b51dc9 HEAD@{1}: reset: moving to HEAD^
c416751 HEAD@{2}: commit: Adds configuration file and
instructions
6b51dc9 HEAD@{3}: reset: moving to HEAD^
9142192 HEAD@{4}: commit: Adds final styling for website
6b51dc9 HEAD@{5}: reset: moving to HEAD^
6c5ecf1 HEAD@{6}: commit: Deletes the pesky js directory
6b51dc9 HEAD@{7}: filter-branch: rewrite
1bc3d71 (refs/original/refs/heads/master) HEAD@{8}: filter-
branch: rewrite
32281cf HEAD@{9}: filter-branch: rewrite
fdb857a HEAD@{10}: rebase -i (abort): updating HEAD
59f601b HEAD@{11}: rebase -i (pick): Linking to the main CSS
file
e725307 HEAD@{12}: rebase -i (pick): Creating basic CSS file
.
.
.
```

It looks a bit like a stash file, doesn't it? The Git **ref log** is like the world's most detailed play-by-play sports commentator. It's a running historical record of absolutely everything that's happened in your repo, from commits, to resets, to rebases and more.

Think of it as Git's undo stack — you can use it to get back to a particular point in time.

Press **Q** to exit the ref log. It's time to see how to resurrect commits that you assumed were long gone.

Finding old commits

You've rethought your changes above. Putting configuration elements in a separate file in the repo along with instructions isn't the best way to go about things. It obviously makes more sense to put those settings, along with Sam's mobile number, on the main wiki page for this project.

That means you can delete that last commit, and you might as well use `git reset --hard` to reset your working directory as well, to keep things clean.

Execute the following command to roll back to the previous commit:

```
git reset HEAD^ --hard
```

Now, check your commit history with the following command. You'll see that `HEAD` now points to the previous commit in the tree. No sign of your commit with the **Adds configuration file and instructions** message remains:

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

Just then, Yasmin pings you via DM. "Hey," she says, "Can you share those two files that have the setup configuration and Sam's mobile number? I'll stick them in the wiki myself. Thanks!"

But — you've gotten rid of that commit with `git reset`. How do you get it back?

Well, use the following command to take a look at Git's ref log to see if you can recover that commit:

```
git reflog
```

Here are the first two lines of my ref log:

```
6b51dc9 (HEAD -> master) HEAD@{0}: reset: moving to HEAD^
297be58 HEAD@{1}: commit: Adds configuration file and
instructions
```

Looking at these two lines in order, the `HEAD@{0}` reference is the `git reset` action you just applied, while the `HEAD@{1}` reference is your previous commit.

So you'll want to go back to the state that `HEAD@{1}` references to get those changes back. To get there, you'll use the `git checkout` command.

Recovering your commit with git checkout

Even though you usually use `git checkout` to switch between branches, as you saw way back at the beginning of this chapter, you can use `git checkout` and specify a commit hash, or in this case, a reflog entry, to create a detached `HEAD` state. You'll do that now.

First, execute `git checkout` with the reflog reference of the commit you want:

```
git checkout HEAD@{1}
```

Git will notify you that you're in a detached `HEAD` state:

```
Note: checking out 'HEAD@{1}'.

You are in 'detached HEAD' state. You can look around, make
experimental
changes and commit them, and you can discard any commits you
make in this
state without impacting any branches by performing another
checkout.

If you want to create a new branch to retain commits you create,
you may
do so (now or later) by using -b with the checkout command
again. Example:

  git checkout -b <new-branch-name>

HEAD is now at 297be58 Adds configuration file and instructions
```

Read through that above output before you go any further. Git has some excellent advice about what you might want to do in a detached `HEAD` state.

You definitely *do* want to retain any commits you create, so you'll need to create a branch to hold them, then merge those changes into `master`.

To see just how this looks in your commit tree, use `git log` to look at the top two commits of the tree:

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

You'll see some output, similar to the following:

```
* 297be58 (HEAD) Adds configuration file and instructions
* 2401800 (master) Updates .gitignore
```

You can tell from `git log` that this is a detached `HEAD` state, since it's not referencing any branch — it's just hanging out there on its own. If `HEAD` referenced a branch, you'd see it in the git log as `6c5ecf1 (HEAD -> master)` or similar.

To save your work in a detached `HEAD` state, use `git checkout` like this:

```
git checkout -b temp
```

That command creates a new branch, `temp`, based on what `HEAD` was pointing to. In this case, that's the detached commit you retrieved from Git's ref log. The command then updates `HEAD` to point to the tip of the `temp` branch.

Checking that your changes worked

Look at your commit tree again with `git log --all --oneline --graph` and you'll see something like this:

```
* 297be58 (HEAD -> temp) Adds configuration file and
instructions
* 2401800 (master) Updates .gitignore
```

`HEAD` now points to the `temp` branch, as you expected. If you pull a directory listing with `ls`, you'll notice that **setup.config** and your one-line change to **README.md** have both been preserved.

To prove that your changes are actually on a proper branch, switch back to `master` with `git checkout master`. Then, execute the following command to see what the tree looks like:

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

Git shows that your resurrected commit is on the `temp` branch and you're safely back on `master`:

```
* 297be58 (temp) Adds configuration file and instructions
* 2401800 (HEAD -> master) Updates .gitignore
```

Using git revert

In all of this work with `git reset` and `git reflog`, you haven't pushed anything to a remote repository. That's by design. Remember, you can't change *shared* history. Once you've pushed something, it's a lot harder to get rid of a commit since you have to synchronize with everyone else.

However, there's one final way to *mostly* undo what you've done in a commit. `git revert` takes the patchset of changes you applied in a specified commit, rolls back those changes, and then creates an entirely new commit on the tip of your branch.

To see this in action — and to learn why I say it can *mostly* undo your changes — you'll merge in the `temp` branch you created above, revert those changes then take a look at your commit history to see what you've done.

Setting up your merge

First, merge in that branch. Ensure you're on `master` to start:

```
git checkout master
```

Then, merge in the `temp` branch like so:

```
git merge temp
```

Git responds with what it's done:

```
Updating 6b51dc9..297be58
Fast-forward
 README.md    | 1 +
 setup.config | 0
 2 files changed, 1 insertion(+)
 create mode 100644 setup.config
```

OK, a fast-forward merge. That makes sense, since `temp` was a direct descendant of the changes on `master`.

Look at your commit history with `git log --all --oneline --graph` and you'll see something like the following:

```
* 297be58 (HEAD -> master, temp) Adds configuration file and
instructions
* 2401800 Updates .gitignore
```

There's `temp`, `master` and `HEAD`. Looks like your merge went fine.

You merrily push those changes to the remote… but then have second thoughts. You decide you *don't* want those changes, after all.

However, you just pushed those changes — and on `master`, of all places — so they're shared with everyone else.

Reverting your changes

While you can't change shared history, you can at least revert the changes you've made here to get back to the previous commit.

`git revert`, like most other Git commands, accepts a target commit: a label, a commit hash or other reference.

Again, you can use relative references to specify the commit you want to revert. In this case, however, you're simply reverting the last changes you made, so you'll use `HEAD` as a reference.

Execute the following command to revert the last change you made to `master`. `git revert` creates a new commit as the result of its actions. To avoid having to go into Vim and edit the message, you'll use the `--no-edit` switch to just accept the default revert message that Git provides:

```
git revert HEAD --no-edit
```

Git tells you what it's doing:

```
[master 82cfe6d] Revert "Adds configuration file and
instructions"
 2 files changed, 1 deletion(-)
 delete mode 100644 setup.config
```

If you compare that with the previous commit from earlier in this chapter that added these changes, you'll see that it's the exact inverse operation:

```
[master 297be58] Adds configuration file and instructions
 2 files changed, 1 insertion(+)
 create mode 100644 setup.config
```

Now, take a look at your commit history with `git log --all --oneline --graph` to see what happened:

```
* 82cfe6d (HEAD -> master) Revert "Adds configuration file and
instructions"
* 297be58 (temp) Adds configuration file and instructions
* 2401800 Updates .gitignore
```

You can see that `git revert` created a new commit at the tip of the `master` branch: `82cfe6d`. If you're still a little unsure what that commit actually did, use the `git log -p -1` command to see the contents of the patch for that commit:

```
diff --git a/README.md b/README.md
index fb18f7c..331487d 100644
--- a/README.md
+++ b/README.md
.
.
.
-For configuration instructions, call Sam on 555-555-5309 at
anytime
diff --git a/setup.config b/setup.config
deleted file mode 100644
index e69de29..0000000
```

The reason it *mostly* undoes your changes is that you still have the original commit that added these undesired changes in history.

If it offends you that the original commit is still in the history, use the techniques in Chapter 15, "Rebasing to Rewrite History" to fix that problem.

And with that, you've seen most of the ways you can undo your work in Git. Hopefully, you've learned some techniques to help you avoid relying on `git reset HEAD --hard` as a scorched earth technique to get your repository back in working order.

Key points

Congratulations on finishing this chapter! Here's a quick recap of what you've covered:

- A **detached HEAD** situation occurs when you check out a commit that no other branch or labeled reference points to.
- **git reset** updates your local system to reflect the state represented by `<commit>`. It also moves `HEAD` to `<commit>`, unlike `git checkout <commit>`.
- Git's regular garbage collection process will eventually clean up any commits left **unreferenced** due to `git reset`.
- **git reset --soft** leaves your working directory and staging area untouched, and simply moves the reference in the index back to the specified commit.
- **git reset --mixed** leaves your working directory untouched, but rolls back the staging area and the reference in the index.
- **git reset --hard** leaves nothing untouched. It rolls your working directory, your staging area and the reference in the index back to the specified commit.
- Use **relative references** to specify a commit, such as `HEAD^` and `HEAD~`.
- **git reflog** shows the entire history of all actions on your local repository and lets you pick a target point to revert to.
- **git revert** applies the inverse of the patch of the target commit to your working directory and creates a new commit.
- **git revert --no-edit** bypasses the need to edit the commit message in Vim.

Where to go from here?

You've already covered quite a lot in this chapter, but I recommend reading a bit more about how relative references work in Git.

Here are two good resources on relative references. In particular, they'll show you the difference between relative addressing using `HEAD~` and `HEAD^`.

Knowing the difference will save you a *lot* of grief in the future when you're trying to fix a repo that seems beyond repair.

- https://stackoverflow.com/questions/2221658/whats-the-difference-between-head-and-head-in-git
- https://git-scm.com/docs/git-rev-parse#_specifying_revisions

This brings an end to the in-depth exploration of the ins and outs of Git internals and the various commands you can use to achieve mastery over your repository.

However, Git is rarely used in isolation. You'll usually use Git in a team setting, so your team will have to collaborate and agree about which workflows to use to avoid stepping on each others' toes.

The next section of the book covers Git development workflows, so if you're struggling to figure out just how to implement Git across your teams, you'll find the upcoming chapters useful.

Section III: Git Workflows

Now that you understand how Git works and how to use some of the advanced features, you need to learn how to incorporate Git into your software development lifecycle. There are established best practices and several formal Git workflows out there.

Those formal Git workflows, well, they're all good, and in some cases, they're all bad. It depends what you want to accomplish in your repo, and how your own team works. GitFlow is one of the most popular branching strategies, but there are alternative models that work well in many situations. This section will introduce you to these workflows and branching models, and explain what problems they solve and what problems they create.

Specifically, you'll cover:

19. **Centralized Workflow**: This model means you work in master all the time. Although this might seem terrifying, it actually works rather well for small teams with infrequent commits.

20. **Feature Branch Workflow**: Feature branches are used to create new features in your code and then merged to master when they're done.

21. **Gitflow Workflow**: A popular method to manage your team's development workflow. In fact, there are even plugins for IDEs that support this Git workflow.

22. **Forking Workflow**: Not all teams have to work out of a single online repository with local clones. When you work at the scale of open-source projects, making each contributor work out of their own fork can be quite advantageous.

Chapter 19: Centralized Workflow

Jawwad Ahmad

A centralized workflow is the simplest way to start with Git. With this system, you work directly on master instead of working in a branch and merging it with master when you're done.

Creating branches in Git is extremely easy, so you should only skip them when they're absolutely unnecessary.

In this chapter, you'll learn about scenarios where the centralized workflow is a good fit. You'll also learn how to handle common situations that arise when multiple developers are committing directly to master.

When to use the centralized workflow

One of the main reasons to first commit and push your code to a branch is to allow other developers to review your code before you push it to master. If the code doesn't need to be reviewed, the overhead of creating and pushing a separate branch is unnecessary. That's where the centralized workflow is a great fit.

Here are a few scenarios where a code review may not be necessary.

1. When working alone

If you're the sole developer on a project, you don't need the overhead of creating branches since there are no other developers to review your code.

Consider the commands you'd run if you were committing your feature to a branch before merging it to master:

```
git checkout -b my-new-feature # 1: Create and switch to branch
# Write the code
git add . && git commit -m "Adding my new feature"
git checkout master            # 2: Switch back to master
git merge my-new-feature       # 3: Merge branch into master
git branch -d my-new-feature   # 4: Delete branch
git push master
```

Compare that to how you'd handle the same update using a centralized workflow. You'd skip the four numbered commands above and end up with only:

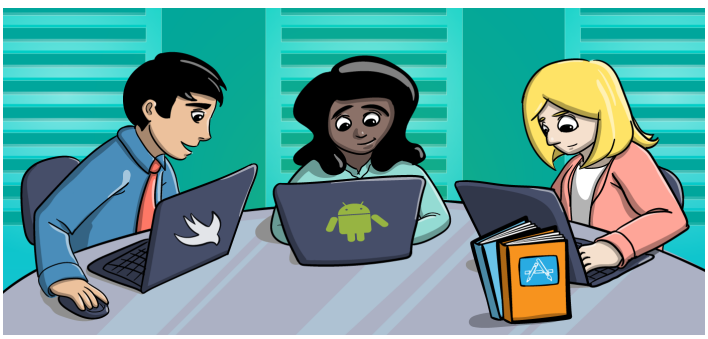
```
# Write the code
git add . && git commit -m "Adding my new feature"
git push master
```

Even when using the centralized workflow, there are still valid reasons to create branches. For example, if you have experimental or incomplete code that you aren't ready to commit to master, you can commit it to a branch and revisit it later.

In the centralized workflow creating branches is optional since you're allowed to push your commits directly to the master branch. This isn't the case in the feature branch workflow which you'll learn about in the next chapter. In that workflow creating branches is required since pushing to master directly is not allowed.

2. When working on a small team

If you're part of a small team where each team member has a specialized area of knowledge, a centralized workflow is a good choice. For example, if one developer works on backend code using one programming language and another works on front-end code in a different language, it's not always useful or practical for those team members to review code outside of their area of expertise.



Small team with non-overlapping expertise or code ownership

In another common scenario, each developer owns a specific area of the code. For example, in an iPhone app, one developer works on the search flow while another works on settings and account preferences. In this scenario, each member of the team is completely responsible for making the changes they need and ensuring their changes work correctly.

3. When optimizing for speed

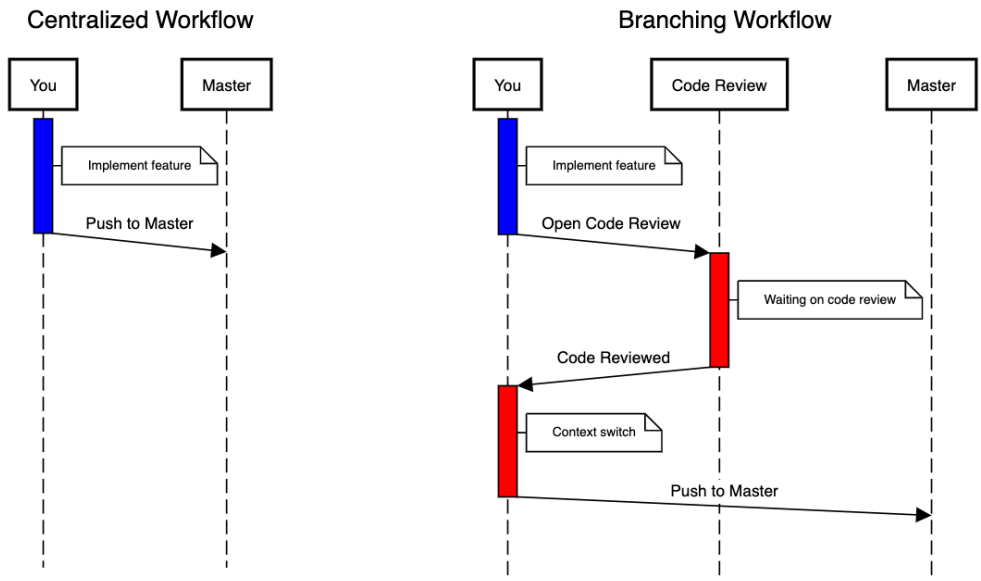
Code reviews are a great way to improve the code's quality before pushing it to the central repository, but every code review has some overhead.

After the author commits their change, they need to wait for someone to review it, which can block them from moving forward.

Furthermore, emails and alerts about code reviews are disruptive. Some team members might stop what they're doing to take a quick look at the code review request to see if they can review it immediately. If not, they need to devote time to do it later. Context switching is especially expensive when performing focused work, such as software development.

Any code that's pending review creates a mental burden for both the author and the rest of the review team.

The following sequence diagram illustrates some of the extra time and overhead required.



Centralized Workflow vs Branching Workflow

The first red section illustrates the overhead of waiting for your code to be reviewed. The second shows the context switch you have to make when you interrupt what you are currently working on to go back and merge the original code into master.

The longer a code review takes, the more likely it is that other people introduce conflicts that you'll have to resolve manually.

If you want to optimize for speed and reduce interruptions, your team can adopt a strategy where code doesn't have to be reviewed before the author pushes it to the remote master branch.

Keep in mind that not reviewing code before pushing it to master doesn't mean that the team can't review the code afterward. It just means that the code on master might not be clean and perfect the first time around.

On the other hand, even well-reviewed code is far from perfect. When optimizing for speed, it might make sense to allow for a bit more entropy for the sake of expediency.

This doesn't mean that you can't have your code reviewed. You can always create a branch to request an ad-hoc code review on a new or complex feature. It just means that there isn't a blanket policy to require a code review for every new feature.

4. When working on a new project

The need for expediency is often stronger when you're working on a new project with tight deadlines. In this case, the inconvenience of waiting for a code review may be especially high.

While bugs are undesirable in any context, unreleased projects have a higher tolerance for them since their impact is low. Thus, you don't have to scrutinize each commit as thoroughly before you push it to master.

Drop dead launch date! Must ship by the 8th!

Even if your new project doesn't start off using a centralized workflow, don't be surprised if your team lets you commit and push directly to master once the deadline approaches!

Centralized workflow best practices

Here are some best practices you can adopt to make using the centralized workflow easier. These are especially important when working in teams where multiple developers are committing to master.

Two important things to keep in mind are to rebase early and often and to prefer rebasing over creating merge commits. If you do accidentally create a merge commit, you can undo it as long as you haven't pushed it to the remote repository.

Rebase early and often

When using the centralized workflow in a team, you often have to rebase before pushing to master to avoid merge commits.

Even before you're ready to push your locally-committed code to the remote repository, you'll benefit from rebasing your work onto any newly-committed code that's available in master. You might pull in a bug fix or code you need for features that you're building upon.

The earlier you resolve conflicts and integrate your work-in-progress with the code on master, the easier it is to do. For example, if you're using a variable that was recently renamed, you'll have fewer updates to make if you pull it in sooner.

Remember, you want to use the `--rebase` option with the `git pull` command so you rebase any commits on your local `master` branch on `origin/master` instead of creating a merge commit. You'll work through an example of this shortly.

Undo accidental merge commits

At times, your local `master` branch may diverge from the remote `origin/master` branch. For example, when you have local commits that you haven't pushed yet, and the remote `origin/master` has newer commits pushed by others.

In this case, executing a simple `git pull` will create a merge commit. Merge commits are undesirable since they add an extra unnecessary commit and make it more challenging to review the Git history.

If you've accidentally created a merge commit, you can easily undo it as long as you haven't pushed it to master.

In the project, you'll work through an example to demonstrate this workflow and how to handle some of the issues you'll encounter when working directly on `master`.

Getting started

To simulate working on a team, you'll play the role of two developers, Alex and Beth!

Alex and Beth are working on an HTML version of a TODO list app called Checklists. They've just started work on the project, so there isn't much code.

Start by unzipping the **repos.zip** file from the **starter** folder for this chapter. You'll see the following unzipped directories within **starter**:

```
starter
└── repos
    ├── alex
    │   └── checklists
    ├── beth
    │   └── checklists
    └── checklists.git
```

At the top level, there are three directories: **alex**, **beth** and **checklists.git**. Within the **alex** and **beth** directories are checked-out copies of the **checklists** project.

What's unique about this setup is that **checklists.git** is configured as the remote origin repository for both Alex's and Beth's checked-out Git repositories. So when you push or pull from within Alex's or Beth's **checklists** repository, it will push to and pull from the local **checklists.git** directory instead of a repository on the internet.

The easiest way to work on the project is to have three separate terminal tabs open. Open your favorite terminal program, then open two additional tabs.

> **Note**: If you're on a Mac, **Command-T** opens a new tab in both **Terminal.app** and **iTerm2.app**, and **Command-Number** switches to the appropriate tab. For example, **Command-2** switches to the second tab.

Once you have three tabs open, `cd` to the **starter** folder and then to **repos/alex/checklists** in the first tab, **repos/beth/checklists** in the second tab and **repos/checklists.git** in the third tab.

```
cd path/to/starter/repos/alex/checklists # 1st Tab
cd path/to/starter/repos/beth/checklists # 2nd Tab
cd path/to/starter/repos/checklists.git  # 3rd Tab
```

To check what the remote origin repository is configured as, run the following command within **alex/checklists** or **beth/checklists**:

```
git config --get remote.origin.url # Note: The --get is optional
```

You'll see the following relative path, which indicates that the remote origin repository is the **checklists.git** directory:

```
../../checklists.git
```

If the remote repository were on GitHub, this URL would have started with either **https://github.com** or **git@github.com** instead of being a local path.

Alex's and Beth's respective projects have been configured with their name and email, so when you commit from within their checklists folder, the commit author will show as Alex or Beth.

While you could run `git config user.name`, and `git config user.email` to verify this, sometimes it's easier to just peek at the local `.git/config` file.

Run the following from within **alex/checklists** or **beth/checklists**:

```
cat .git/config
```

At the end of the file, you'll see their **user.name** and **user.email** settings:

```
...
[user]
    name = Alex Appleseed
    email = alex@example.com
```

Note: Your own name and email should already be configured in your global .gitconfig file. You can run `cat ~/.gitconfig` to verify this.

State of the project

The remote origin repository, **checklists.git**, contains four commits, which we'll refer to as A1, B1, A2 and B2 instead of with their commit hashes. Alex's and Beth's projects also have local commits that have not yet been pushed to the remote. Alex has one additional commit, A3, and Beth has two, B3 and B4.

In your terminal, switch to the **checklists.git** tab and run **`git log --oneline`**:

```
824f3c7 (HEAD -> master) B2: Added empty head and body tags
3a9e970 A2: Added empty html tags
b7c58f4 B1: Added index.html with <!DOCTYPE html> tag
a04ae7f A1: Initial Commit: Added LICENSE and README.md
```

You can see the four commits on `origin`: A1, B1, A2 and B2.

> **Note**: The **checklists.git** repository is a *bare* repo, which means that it only contains the history without a working copy of the code. You can run commands that show you the history, like `git log`, but commands that give you information about the state of the working copy, such as `git status`, will fail with an error, `fatal: this operation must be run in a work tree`.

Next switch to the **alex/checklists** tab and run **`git log --oneline`**:

```
865202c (HEAD -> master) A3: Added Checklists title within head
824f3c7 (origin/master, origin/HEAD) B2: Added empty head and...
3a9e970 A2: Added empty html tags
b7c58f4 B1: Added index.html with <!DOCTYPE html> tag
a04ae7f A1: Initial Commit: Added LICENSE and README.md
```

You can see A3 in addition to the four commits already on `origin/master`.

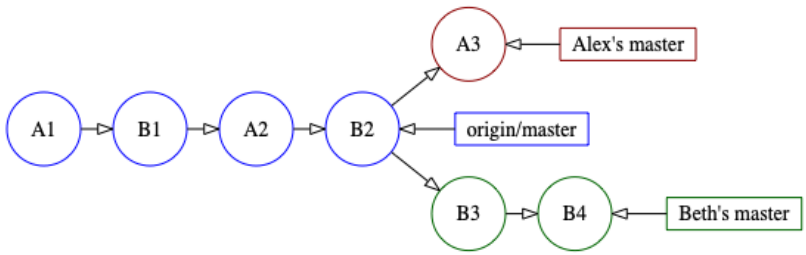
> **Note**: Some commit messages, such as for B2 above, will be shortened to end with an ellipsis (`...`) to fit them on a single line.

Finally, switch to the **beth/checklists** tab and run **`git log --oneline`**:

```
4da1174 (HEAD -> master) B4: Added "Welcome to Checklists!" w...
ed17ce4 B3: Added "Checklists" heading within body
824f3c7 (origin/master, origin/HEAD) B2: Added empty head and...
3a9e970 A2: Added empty html tags
b7c58f4 B1: Added index.html with <!DOCTYPE html> tag
a04ae7f A1: Initial Commit: Added LICENSE and README.md
```

You can see B3 and B4 in addition to the four commits already on `origin/master`.

Here is a combined view of the commits in the three repositories:



Relationship between origin/master and Alex and Beth's master branches

So while Alex and Beth are both working on `master`, their branches have diverged.

At this point, either Alex or Beth could push their commits to `origin`, but once one of them does, the other won't be able to.

For the remote to accept a push, it needs to result in a fast-forward merge of `master` on the remote. In other words, the pushed commits need to be direct descendants of the latest commit on `origin/master`, i.e. of B2.

Currently, both Alex's and Beth's commits qualify to be pushed. But once the remote's `master` branch is updated with one person's commits, the other won't be able to push without rebasing or creating a merge commit.

You'll have Beth push her commits to `origin` first.

Pushing Beth's commits to master

Switch to **beth/checklists** in your terminal and run **`git status`**. It should show the following to verify that it's ahead of `origin/master` by two commits:

```
On branch master
Your branch is ahead of 'origin/master' by 2 commits.
...
```

Now, run **`git push`** to push Beth's commits to the remote `master` branch.

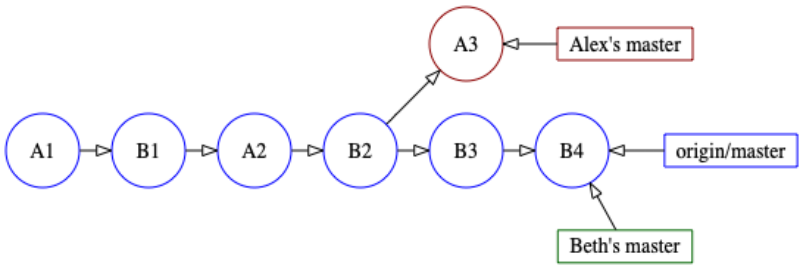
It'll successfully push both commits to the remote repository, i.e. to **checklists.git**.

Switch to the **checklists.git** tab and run **`git log --oneline`**:

```
4da1174 (HEAD -> master) B4: Added "Welcome to Checklists!" w...
ed17ce4 B3: Added "Checklists" heading within body
824f3c7 B2: Added empty head and body tags
...
```

You can see Beth's two additional commits B3 and B4, ahead of B2.

This is what it looks after Beth's push:



Relationship between origin/master and local master branches after Beth's push

Next, you'll attempt to push Alex's A3 commit to `master`.

Pushing Alex's commit to master

Switch to **alex/checklists** and run **`git status`**:

```
On branch master
Your branch is ahead of 'origin/master' by 1 commit.
...
```

Alex's repository still thinks it's one commit ahead of `origin/master`. This is because he hasn't yet run a `git fetch` after Beth's push.

You'll run `git fetch` in a moment, but first, run **`git push`** to see what happens:

```
To ../../checklists.git
 ! [rejected]        master -> master (fetch first)
error: failed to push some refs to '../../checklists.git'
hint: Updates were rejected because the remote contains work
hint: that you do not have locally. This is usually caused by
hint: another repository pushing to the same ref. You may want
hint: to first integrate the remote changes (e.g.,
hint: 'git pull ...') before pushing again.
...
```

Uh oh. Take a look at the hint message piece by piece.

First, it says:

```
Updates were rejected because the remote contains work that you
do not have locally.
```

That's right, since it now contains the two additional commits from Beth: B3 and B4.

Then it says:

```
This is usually caused by another repository pushing to the same
ref.
```

Yes, that's exactly what Beth just did.

And finally, it suggests:

```
You may want to first integrate the remote changes (e.g., 'git
pull ...') before pushing again.
```

That's what you'll do next. But first, run **`git status`** again; you'll see that it still thinks Alex's branch is ahead of `origin/master` by one commit:

```
On branch master
Your branch is ahead of 'origin/master' by 1 commit.
...
```

Although the origin repository rejected the changes, the local repository still hasn't fetched updates from `origin`.

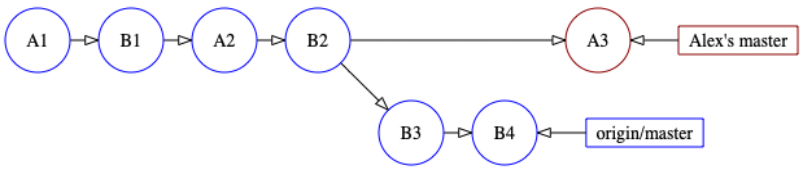
Run **`git fetch`** to fetch updates from the remote. When you run **`git status`** now, it will correctly show that your local `master` branch has diverged from `origin/master`:

```
On branch master
Your branch and 'origin/master' have diverged,
and have 1 and 2 different commits each, respectively.
  (use "git pull" to merge the remote branch into yours)
```

Run **`git log --oneline --graph --all`** to see the log in graph format:

```
* 865202c (HEAD -> master) A3: Added Checklists title within ...
| * 4da1174 (origin/master, origin/HEAD) B4: Added "Welcome t...
| * ed17ce4 B3: Added "Checklists" heading within body
|/
* 824f3c7 B2: Added empty head and body tags
* 3a9e970 A2: Added empty html tags
* b7c58f4 B1: Added index.html with <!DOCTYPE html> tag
* a04ae7f A1: Initial Commit: Added LICENSE and README.md
```

Which is just a textual representation of the following:



Visual representation of the previous git log --oneline --graph --all command

> **Note**: Without the `--graph` option, it would have looked like the commit history was all on one branch. Without the `--all` option, it would only have shown you the commits on your current branch — that is, on `master` but not on `origin/master`. Try running the command without each of the options for comparison.

You can see that your local master has diverged from `origin/master`. You can't push to the remote repository in this state.

There are two ways you can resolve this issue:

1. The first and recommended way is to run `git pull` with the `--rebase` option to rebase any commits to your local master branch onto `origin/master`.
2. The second way is to create a merge commit by running `git pull`, committing the merge and pushing the merge commit to the remote.

Since it's easy to forget the `--rebase` option and simply run `git pull`, you'll use the non-recommended way first so you can also learn how to undo an accidentally-created merge commit.

Undoing a merge commit

Since Alex's `master` branch has diverged from `origin/master`, running a `git pull` will result in a merge commit.

This is because **`git pull`** is actually the combination of two separate commands: **`git fetch`** and **`git merge origin/master`**.

If Alex didn't have any local commits, then the implicit `git merge` part of the command would perform a fast-forward merge. This means that Alex's `master` branch pointer would simply move forward to where `origin/master` is pointing to. However, since `master` has diverged, this creates a merge commit.

1. Abort the merge commit

The easiest way to prevent a merge commit is to short-circuit the process by leaving the commit message empty.

From **alex/checklists**, run **`git pull`**. Vim will open with the following:

```
Merge branch 'master' of ../../checklists
# 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.
```

Take a look at the last line of the commit message template. It says:

```
Lines starting with '#' will be ignored, and an empty message
aborts the commit.
```

This means that you can enter **dd** to delete the first line and leave the remaining lines since they all start with a **#**.

However, there's something reassuring about clearing the complete commit message. Since it takes the same number of keystrokes, you'll do that instead. Enter **dG** to delete everything until the end and then **:wq** to exit.

Now, you'll see the following:

```
Auto-merging index.html
error: Empty commit message.
Not committing merge; use 'git commit' to complete the merge.
```

As the last line above indicates, you aborted the commit of the merge, but not the merge itself.

You can verify this by running a `git status`:

```
...
All conflicts fixed but you are still merging.
...
```

Run the following command to abort the merge itself:

```
git merge --abort
```

Congratulations, merge commit averted!

2. Hard reset to ORIG_HEAD

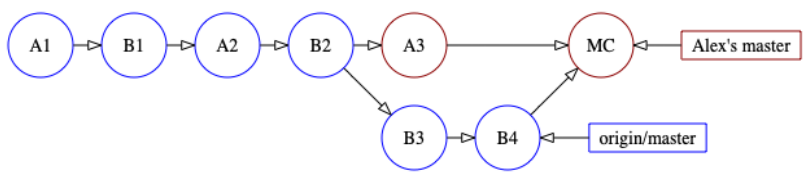
So what can you do if you accidentally created the merge commit? As long as you haven't pushed it yet, you can reset your branch to its original commit hash before the merge.

Run **`git pull`** again to trigger the merge. When Vim opens, type **`:wq`** to accept the default message and commit the merge.

Now run **`git log --oneline --graph`**:

```
*   fc15106 (HEAD -> master) Merge branch 'master' of ../../c...
|\
| * 4da1174 (origin/master, origin/HEAD) B4: Added "Welcome t...
| * ed17ce4 B3: Added "Checklists" heading within body
* | 865202c A3: Added Checklists title within head
|/
* 824f3c7 B2: Added empty head and body tags
* 3a9e970 A2: Added empty html tags
* b7c58f4 B1: Added index.html with <!DOCTYPE html> tag
* a04ae7f A1: Initial Commit: Added LICENSE and README.md
```

Visually, your repository is in the following state:



Now you have a merge commit, MC, that is a combination of all of the contents of `origin/master` that weren't in your branch yet. In this case, MC would contain the code from Beth's B3 and B4 commits.

As long as you haven't pushed the merge commit to master, you can undo it. First, however, you have to determine what the commit hash of Alex's `master` branch was before the merge, and then run `git reset --hard` using that commit hash.

One way to identify the commit hash is by looking at the commit log. You can visually see that **`865202c`** is the commit hash for the **A3** commit, which is where **`master`** was before the merge, so you could run `git reset --hard 865202c`.

There's also an easier way to identify the commit hash before the merge. When Git commits a merge operation, it saves the original commit hash before the merge into `ORIG_HEAD`.

If you're curious, you can run either of the following commands to see what the commit hash is for `ORIG_HEAD`:

```
git rev-parse ORIG_HEAD
```

or

```
cat .git/ORIG_HEAD
```

This shows the following:

```
865202c4bc2a12cc2fbb94f5980b00457d270113
```

Run the following command to perform the reset:

```
git reset --hard ORIG_HEAD
```

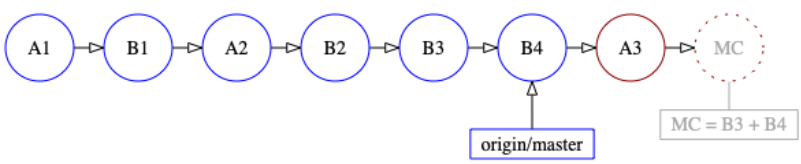
You should see the following confirmation message:

```
HEAD is now at 865202c A3: Added Checklists title within head
```

You're back to where you started, which is exactly what you wanted!

3. Rebase the merge commit

Another strategy you can adopt is to rebase the merge commit onto `origin/master`. This applies A3 and the merge commit on top of B4. Since `origin/master` already has B3 and B4, i.e., the contents of the merge commit, this removes the merge commit entirely.



Create the merge commit again by running `git pull` and then **:wq** to save the commit message.

Now run the following:

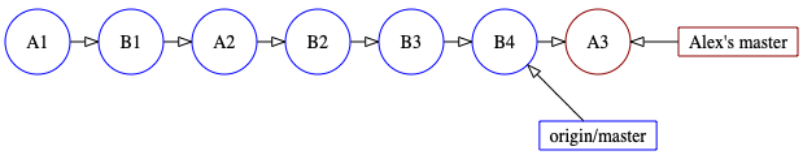
```
git rebase origin/master
```

Then run **`git log --oneline --graph`** to take a look at the commit history:

```
* 7988360 (HEAD -> master) A3: Added Checklists title within ...
* 4da1174 (origin/master, origin/HEAD) B4: Added "Welcome to ...
* ed17ce4 B3: Added "Checklists" heading within body
* 824f3c7 B2: Added empty head and body tags
* 3a9e970 A2: Added empty html tags
* b7c58f4 B1: Added index.html with <!DOCTYPE html> tag
* a04ae7f A1: Initial Commit: Added LICENSE and README.md
```

You can see that you rebased A3 on top of B4, and the merge commit has disappeared!

Visually, your repository is now in the following state:



This is the same outcome that you would have had with `git pull --rebase`, which is what you'll try next.

You could push at this point, but instead, you'll reset your branch again so you can try `git pull --rebase`. Since you rebased after the merge, you can no longer use `ORIG_HEAD`, so you'll reset to the commit hash directly. Resetting to `ORIG_HEAD` would have taken you back to the merge commit before the rebase.

Run the following:

```
git reset --hard 865202c
```

Then run **`git log --oneline --graph --all`** to verify that you've reset `master`.

Using git pull --rebase

You previously learned that **`git pull`** is the combination of two separate commands: **`git fetch`**, and **`git merge origin/master`**.

Adding the **`--rebase`** option to **`git pull`** essentially changes the second **`git merge origin/master`** command to **`git rebase origin/master`**.

Run **`git pull --rebase`**. You'll see the following:

```
First, rewinding head to replay your work on top of it...
Applying: A3: Added Checklists title within head
```

Then run **`git log --oneline --graph`** to take a look at the commit history:

```
* 4742353 (HEAD -> master) A3: Added Checklists title within ...
* 4da1174 (origin/master, origin/HEAD) B4: Added "Welcome to ...
* ed17ce4 B3: Added "Checklists" heading within body
...
```

You can see that you've now rebased your local A3 commit onto `origin/master`.

Reset your master branch one final time for the next exercise:

```
git reset --hard 865202c
```

Setting up automatic rebase

You may occasionally forget that you have local commits on `master` before you run `git pull`, resulting in a merge commit. Of course, this is no longer a terrible issue since you now know how to abort and undo merge commits.

But wouldn't it be swell if Git could automatically take care of this for you? And it can! By setting the **`pull.rebase`** option to **`true`** in your Git configuration, you can do just that.

Run the following command to set Git up to always rebase when you run `git pull`:

```
git config pull.rebase true
```

Now run **`git pull`**. It will automatically rebase your commit on top of `origin/master` instead of creating a merge commit.

Now, finally, the moment you've been working toward! Run **`git push`** to push Alex's newly rebased commit to the `master` branch of the remote.

```
git push
```

Voila! You can now `git pull` without having to remember to add the `--rebase` option.

One final point to keep in mind is that each developer on your team would have to configure this option for themselves. If there are common configuration options like this that would be useful for everyone on the team, consider adding them to something like a `setup_git_config.sh` file that you'd commit to the repository.

Key points

- The centralized workflow is a good fit when working alone or on small teams, when optimizing for speed or when working on a new, unpublished project.
- You can still create branches for in-progress code or for ad-hoc code reviews.
- Rebase frequently to incorporate upstream changes and resolve conflicts sooner.
- Prefer **`git pull --rebase`** instead of **`git pull`** to avoid creating merge commits.
- Set the **`pull.rebase`** option to **`true`** in your Git config to automatically rebase when pulling.
- There are multiple ways to undo accidental merge commits as long as you haven't pushed them to the remote repository.

Now that you have a good handle on using the centralized workflow, the next step in your Git journey is to branch towards the branching workflow. Proceed to the next chapter to get started!

Chapter 20: Feature Branch Workflow

This is an early access release of this book. Stay tuned for this chapter in a future release!

Chapter 21: Gitflow Workflow

This is an early access release of this book. Stay tuned for this chapter in a future release!

Chapter 22: Forking Workflow

This is an early access release of this book. Stay tuned for this chapter in a future release!

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.

Version control systems like Git are incredibly important to coordinate and collaborate with file-based projects. Git, at its core, is very simple once you understand those fundamental pieces of what is going on when you commit changes. When things go wrong it is important to know how to step through resolving those issues, which you now know how to do.

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, and Aaron

The *Mastering Git* team