LEARN ENOUGH

GIT

TO BE DANGEROUS

FUNDAMENTALS **03**

TUTORIAL BY

MICHAEL HARTL

Learn Enough Git to Be Dangerous

An introduction to version control with Git

Michael Hartl

Contents

1 Getting started **1**

1.1 Installation and setup 6

1.1.1 Exercises 7

1.2 Initializing the repo 8

1.2.1 Exercises 9

1.3 Our first commit 10

1.3.1 Exercises 13

1.4 Viewing the diff 14

1.4.1 Exercises 15

1.5 Adding an HTML tag 16

1.5.1 Exercises 23

1.6 Adding HTML structure 23

1.6.1 Exercises 26

1.7 Summary 28

2 Backing up and sharing **31**

2.1 Signing up for GitHub 31

2.2 Remote repo 33

2.2.1 Exercises 38

2.3 Adding a README 38

2.3.1 Exercises 46

2.4 Summary 46

3 Intermediate workflow **47**
3.1 Commit, push, repeat . . . 47
3.1.1 Exercises . . . 52
3.2 Ignoring files . . . 54
3.2.1 Exercises . . . 56
3.3 Branching and merging . . . 56
3.3.1 Rebasing . . . 66
3.3.2 Exercises . . . 67
3.4 Recovering from errors . . . 67
3.4.1 Exercises . . . 72
3.5 Summary . . . 73

4 Collaborating **75**
4.1 Clone, push, pull . . . 76
4.1.1 Exercises . . . 88
4.2 Pulling and merge conflicts . . . 89
4.2.1 Non-conflicting changes . . . 89
4.2.2 Conflicting changes . . . 96
4.2.3 Exercises . . . 102
4.3 Pushing branches . . . 103
4.3.1 Exercises . . . 114
4.4 A surprise bonus . . . 115
4.4.1 Exercises . . . 120
4.5 Summary . . . 121
4.6 Advanced setup . . . 121
4.6.1 A checkout alias . . . 121
4.6.2 Prompt branches and tab completion . . . 122
4.6.3 Exercises . . . 124
4.7 Conclusion . . . 126

About the author

Michael Hartl is the creator of the *Ruby on Rails Tutorial*, one of the leading introductions to web development, and is cofounder and principal author at Learn Enough. Previously, he was a physics instructor at the California Institute of Technology (Caltech), where he received a Lifetime Achievement Award for Excellence in Teaching. He is a graduate of Harvard College, has a Ph.D. in Physics from Caltech, and is an alumnus of the Y Combinator entrepreneur program.

Chapter 1

Getting started

Learn Enough Git to Be Dangerous is the final installment in a trilogy of tutorials on *Developer Fundamentals* designed to teach three skills essential for software developers and those who work with them. Its only prerequisites are the first two tutorials in the trilogy, *Learn Enough Command Line to Be Dangerous* (covering the Unix command line) and *Learn Enough Text Editor to Be Dangerous* (covering text editors).

This tutorial covers a third essential skill: *version control*. As with its two predecessors, *Learn Enough Git to Be Dangerous* doesn't even assume you're familiar with the *category* of application, so if you're unsure about what "version control" is, you're in the right place. Even if you are already familiar with the subject, it's likely you'll still learn a lot from this tutorial. Either way, learning the material in *Learn Enough Git to Be Dangerous* prepares you for the other Learn Enough tutorials while enabling an astonishing variety of applications—including a special surprise bonus at the end (Box 1.1).

Box 1.1. Real artists ship

As legendary Apple cofounder Steve Jobs once said: *Real artists ship.* What he meant was that, as tempting as it is to privately polish in perpetuity, makers must *ship* their work—that is, actually finish it and get it out into the world. This can

be scary, because shipping means exposing your work not only to fans but also to critics. "What if people don't like what I've made?" *Real artists ship.*

It's important to understand that shipping is a separate skill from making. Many makers get good at making things but never learn to ship. To keep this from happening to us, starting in *Learn Enough Git to Be Dangerous* we're going to ship at least one thing in every Learn Enough tutorial. In fact, in this tutorial we'll actually ship *two* things—a public *Git repository* and a surprise bonus that will give you bragging rights with all of your friends.

Version control solves a problem that might look familiar if you've ever seen Word documents or Excel spreadsheets with names like **`Report_2014_-1.doc`**, **`Report_2014_2.doc`**, **`Report_2014_3.doc`**, or **`budget-v7.xls`**. These cumbersome names indicate how annoying it can be to track different versions of documents. Nowadays, applications like Word do sometimes offer built-in version tracking, but such features are tightly coupled to the underlying application, and aren't useful for any other document types. Many technical applications (including most websites and programming projects) require a general solution to the problem of versions.

A version control system, or *VCS*, provides an automatic way to track changes in software projects, giving creators the power to view previous versions of files and directories, develop speculative features without disrupting the main development, securely back up the project and its history, and collaborate easily and conveniently with others. In addition, using version control also makes deploying production websites and web applications much easier. As a result, fluency in at least one version control system is an essential component of *technical sophistication* (Box 1.2), a useful skill for developer, designer, and manager alike. This applies especially to the version control system covered in this tutorial, called *Git*.

Box 1.2. Technical sophistication

A principal theme of the Learn Enough tutorials is the development of *technical sophistication*, the combination of hard and soft skills that make it seem like you can magically solve any technical problem (as illustrated in "Tech Support Cheat Sheet" from xkcd). *Learn Enough Git to Be Dangerous* is important for developing these skills because being able to use at least one modern version control system is an essential component of technical sophistication.

In the context of Git, technical sophistication includes several things. Many Git commands print various details to the terminal screen; technical sophistication lets you figure out which ones to pay attention to and which to ignore. There are also many Git-related resources on the web, which among other things means that Google searches are often useful for figuring out the exact command you need at a particular time. Technical sophistication lets you figure out the best search terms for finding the answer you're looking for; e.g., if you need to delete a remote branch (Section 4.3.1), Googling for "git delete remote branch" is a good bet to turn up something useful. Finally, repository hosting sites like GitHub, GitLab, and Bitbucket typically include commands to help guide you through various setup tasks, and technical sophistication gives you the confidence to follow the steps even if you don't understand every detail.

One helpful command for learning Git is **`git help`**, which by itself gives general guidelines on Git usage, and when applied to a specific command gives further information on that command. For example, **`git help add`** shows details about the **`git add`** command. The output of **`git help`** is similar to the man pages covered in *Learn Enough Command Line to Be Dangerous*: full of useful but often obscure information. As always, use your technical sophistication to help make sense of it.

Version control has evolved considerably over the years. The family line leading to Git includes programs called RCS, CVS, and Subversion, and there are many current alternatives as well, including Perforce, Bazaar, and Mercurial. I mention these examples not because you need to know what they are, but only to show what a bewildering variety there is. What's worse, when you

choose a version control system, you really *commit* to it,[1] and it is often difficult to switch from one to another. Happily, in the last few years an undisputed winner has emerged in the open-source VCS wars: Git. This victory is the main reason this tutorial is called *Learn Enough Git to Be Dangerous* rather than *Learn Enough Version Control to Be Dangerous*. Nevertheless, many of the ideas here are quite general, and if by some chance you need to use a different VCS, this tutorial will still provide a useful introduction to the subject.

Originally developed by Linux creator Linus Torvalds[2] to host the Linux kernel, Git is a command-line program that is designed in the Unix tradition (which is why a familiarity with the Unix command line is an important prerequisite). Git has a combination of power, speed, and community adoption that leave it few rivals, but it can be tricky to learn, and other Git tutorials have a tendency to introduce lots of heavy theory, which can be interesting to learn but in practice is really only understood by a tiny handful of Git users (as illustrated in "Git" via the webcomic xkcd). The good news is that the set of Git commands needed to be productive is relatively small; there are some pointers to more advanced and theory-oriented resources listed in Section 4.7, but in this tutorial we focus on the essential commands needed to be *dangerous*.

Note: If you're using macOS, you should follow the instructions in Box 1.3 at this time.

Box 1.3. Switching macOS to Bash

If you're using macOS, at this point you should make sure you're using the right shell program for this tutorial. The default shell as of macOS Catalina is *Z shell* (Zsh), but to get results consistent with this tutorial you should switch to the shell known as *Bash*.

The first step is to determine which shell your system is running, which you can do using the `echo` command:

[1] Pun intended. If you don't get it, don't worry—by the end of this tutorial, you will.

[2] *Git* is a mildly insulting British slang term for a stupid or annoying person, and Linus likes to joke that he named both Linux and Git after himself.

```
$ echo $SHELL
/bin/bash
```

This prints out the `$SHELL` environment variable. If you see the result shown above, indicating that you're already using Bash, you're done and can proceed with the rest of the tutorial. (In rare cases, `$SHELL` may differ from the current shell, but the procedure below will still correctly change from one shell to another.) For more information, including how to switch to and use Z shell with this tutorial, see the Learn Enough blog post "Using Z Shell on Macs with the Learn Enough Tutorials".

The other possible result of `echo` is this:

```
$ echo $SHELL
/bin/zsh
```

If that's the result you get, you should use the `chsh` ("change shell") command as follows:

```
$ chsh -s /bin/bash
```

You'll almost certainly be prompted to type your system password at this point, which you should do. Then completely exit your shell program using Command-Q and relaunch it.

You can confirm that the change succeeded using `echo`:

```
$ echo $SHELL
/bin/bash
```

At this point, you will probably start seeing the following alert, which you should ignore:

```
The default interactive shell is now zsh.
To update your account to use zsh, please run `chsh -s /bin/
For more details, please visit https://support.apple.com/kb/

[~]$
```

Note that the procedure above is entirely reversible, so there is no need to be concerned about damaging your system. See "Using Z Shell on Macs with the Learn Enough Tutorials" for more information.

1.1 Installation and setup

The most common way to use Git is via a command-line program called **`git`**, which lets us transform an ordinary Unix directory into a *repository* (or *repo* for short) that enables us to track changes to our project.[3] In this section, we'll begin by installing Git (if necessary) and doing some one-time configuration.

Before doing anything else, we first need to check to see if Git is installed on the current system. As a reminder, we're working in the Unix tradition, so it is strongly recommended that you use macOS or Linux (possibly via a virtual machine (Box 1.4)).

Box 1.4. Using Unix

This tutorial, as with the others in the Learn Enough to Be Dangerous series, assumes you have access to a computer running some variant of Unix. If you already run macOS or Linux, you're good to go, but if you're on Windows you should either use a *cloud IDE* or install a *Linux virtual machine*. Both options are covered in *Learn Enough Dev Environment to Be Dangerous*.

The easiest way to check for Git is to start a terminal window and use **`which`**[4] at the command line to see if the **`git`** executable is already present:

[3] As we'll see in Section 1.2, Git uses a special hidden directory called **`.git`** to track changes, but at the level of this tutorial these details aren't important.

[4] The **`which`** command is discussed in *Learn Enough Command Line to Be Dangerous*.

```
$ which git
/usr/local/bin/git
```

If the result is empty or if it says the command is not found, it means you have to install Git manually. To do this, follow the instructions at "Getting Started – Installing Git" in the official Git documentation. (This will likely give you an opportunity to apply some technical sophistication (Box 1.2).)

After installing Git but before starting a project, we need to perform a couple of one-time setup steps, as shown in Listing 1.1. These are *global* setups, meaning you only have to do them once per computer. (Don't worry about the meaning or structure of these commands at this stage.)

Listing 1.1: One-time global configuration settings.

```
$ git config --global user.name "Your Name"
$ git config --global user.email your.email@example.com
```

These configuration settings allow Git to identify your changes by name and email address, which is especially helpful when collaborating with others (Chapter 4). Note that the name and email you use in Listing 1.1 will be viewable in any projects you make public, so don't expose any information you'd rather keep private.

In addition to this required configuration, *Learn Enough Git to Be Dangerous* includes some optional advanced setup (Section 4.6) that I recommend you complete at some point. If you already have some familiarity with Git, or if you're an experienced user of the Unix command line, I recommend completing the steps in Section 4.6 at this time, but otherwise I recommend deferring the advanced setup until later.

1.1.1 Exercises

1. Run `git help` at the command line. What is the first command listed?
2. There's a chance that the full output of `git help` was too big to fit in your terminal, with most of it just scrolling by. What's the command to

let us navigate the output of `git help` interactively? (On some systems, you can use the mouse to scroll back in the terminal window, but it's unwise to rely on this fact.) *Hint*: Pipe the output to `less`.

3. Git stores global configuration settings in a hidden text file located in your home directory. By inspecting the file `~/.gitconfig` with a tool of your choice (`cat`, `less`, a text editor, etc.), confirm that the configuration set up by Listing 1.1 corresponds to simple text entries in this file.

1.2 Initializing the repo

Now it's time to start creating a project and put it under version control with Git. To see how Git works and what benefits it brings, it helps to have a concrete application in mind, so we'll be tracking changes in a simple project consisting of a small website consisting of two pages, a Home page and an About page.[5] We'll begin by making a directory with the generic name `website` inside a repositories directory called `repos`:

```
[~]$ mkdir -p repos/website
```

Here we've used the "make directory" command `mkdir` covered in *Learn Enough Command Line to Be Dangerous*, together with the `-p` option, which arranges for `mkdir` to create intermediate directories as required (in this case, `repos`). Note also that I've included the current directory in the prompt (in this case, `[~]`) as arranged by the configuration in Listing 4.15.

After making the directory, we can `cd` into it as follows:

```
[~]$ cd repos/website/
[website]$
```

[5] *Learn Enough HTML to Be Dangerous* and *Learn Enough CSS & Layout to Be Dangerous* build on this foundation to make more complicated sites.

(Recall that you can use tab completion when changing directories, so in real life I would probably type something like `cd re`→|`w`→|.)

Even though the `website` directory is empty, we can already convert it to a *repository*, which you can think of as a sort of enhanced Unix directory with the additional ability to track changes to every file and subdirectory. The way to create a new repository with Git is with the `init` command (short for "initialize"), which creates a special hidden directory called `.git` where Git stores the information it needs to track our project's changes. (It's the presence of a properly configured `.git` directory that distinguishes a Git repository from a regular directory.)

All Git commands consist of the command-line program `git` followed by the name of the command, so the full command to initialize a repository is `git init`, as shown in Listing 1.2.

Listing 1.2: Initializing a Git repository.

```
[website]$ git init
Initialized empty Git repository in /Users/mhartl/repos/website/.git/
[website (master)]$
```

The prompt shown in Listing 1.2 reflects both the Bash customization from *Learn Enough Text Editor to Be Dangerous* and the advanced setup in Section 4.6.2, so your prompt may differ.[6] In particular, Listing 1.2 shows the name of the default Git *branch*, called `master`.[7] Don't worry about what this means now; we'll discuss branches starting in Section 3.3.

1.2.1 Exercises

1. By running `ls -a` (discussed in *Learn Enough Command Line to Be Dangerous*), list **a**ll the files and directories in your `website` directory.

[6] To learn how to set up this same custom prompt using Z shell, see the Learn Enough blog post "Using Z Shell on Macs with the Learn Enough Tutorials".

[7] The `master` branch is created automatically by `git init`, but some developers prefer to use a different name for the default branch. Unfortunately, there is no universally agreed-upon alternative; for one possibility, see the post "Easily rename your Git default branch from master to main" by Scott Hanselman.

What is the name of the hidden directory used by the Git repository? (There is one such hidden directory per project.)

2. Using the result of the previous exercise, run `ls` on the hidden directory and guess the name of the main Git configuration file. Use `cat` to dump its contents to the screen.

1.3 Our first commit

Git won't let us complete the initialization of the repository while it's empty, so we need to make a change to the current directory. We'll make a more substantive change in a moment, but for now we'll follow a common convention and simply use `touch` to create an empty file (as mentioned in *Learn Enough Command Line to Be Dangerous*). In this case, we're making a simple website, and the near-universal convention is to call the main page `index.html`:

```
[website (master)]$ touch index.html
```

Having created this first file, we can use the `git status` command to see the result:

```
[website (master)]$ git status
On branch master

Initial commit

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

  index.html

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

We see here that the `index.html` file is "untracked", which means Git doesn't yet know about it. We can *add* it using the `git add` command:

```
[website (master)]$ git add -A
```

Here the **-A** option tells Git to add *all* untracked files, even though in this case there's only one. In my experience, 99% of the time you add files you'll want to add them all, so this is a good habit to cultivate, and learning how to add individual files is left as an exercise (Section 1.3.1). (By the way, the nearly equivalent command **git add .**, where the dot refers to the current directory, is also common.)[8]

We can see the result of **git add -A** by running **git status** again:

```
[website (master)]$ git status
On branch master

Initial commit

Changes to be committed:
  (use "git rm --cached <file>..." to unstage)

  new file:   index.html
```

As implied by the word "unstage", the status of the file has been promoted from *untracked* to *staged*, which means the file is ready to be added to the repository. *Untracked/unstaged* and *staged* are two of the four states commonly used by Git, as shown in Figure 1.1. (Technically, untracked and unstaged are different states, but the distinction is rarely important because **git add** tracks and stages files at the same time.)

As shown in Figure 1.1, after putting changes in the staging area we can make them part of the local repository by *committing* them using **git commit**. (We'll cover the final step from Figure 1.1, **git push**, in Section 2.3.) Most uses of **git commit** use the command-line option **-m** to include a *message* indicating the purpose of the commit (Box 1.5). In this case, the purpose is to initialize the new repository, which we can indicate as follows:

[8] In the rare cases where the two differ, what you usually want is **git add -A**, and this is what's used in the official Git documentation, so that's what we go with here.

Figure 1.1: The main Git status sequence for a changing file.

```
[website (master)]$ git commit -m "Initialize repository"
[master (root-commit) 879392a] Initialize repository
 1 file changed, 0 insertions(+), 0 deletions(-)
 create mode 100644 index.html
```

(I've shown my output here for completeness, but your details will vary.)

Box 1.5. Committing to Git

By design, Git requires every commit to include a *commit message* describing the purpose of the commit. Typically, this takes the form of a single line, usually limited to around 72 characters, with an optional longer message if desired (Section 4.2.3). Although conventions for commit messages vary (as humorously depicted in the xkcd comic strip "Git Commit"), the style adopted in this tutorial is to write commit messages in the *present tense* using the *imperative mood*, as in "Initialize repository" rather than "Initializes repository" or "Initialized repository". The reason for this convention is that Git models commits as a series of text transformations, and in this context it makes sense to describe what each commit *does* instead of what it did. Moreover, this usage agrees with the convention followed by the commit messages generated by Git commands themselves (e.g., "merge" rather than "merges" or "merged"). For more information, see the GitHub article "Shiny new commit styles".

At this point, we can use **`git log`** to see a record of our commit:

```
[website (master)]$ git log
commit 879392a6bd8dd505f21876869de99d73f40299cc
Author: Michael Hartl <michael@michaelhartl.com>
Date:   Thu Dec 17 20:00:34 2015 -0800

    Initialize repository
```

The commit is identified by a *hash*, which is a unique string of letters and numbers that Git uses to label the commit and which lets Git retrieve the commit's changes. In my case, the hash appears as

```
879392a6bd8dd505f21876869de99d73f40299cc
```

but since each hash is unique your result will differ. The hash is often referred to as a "SHA" (pronounced *shah*) because of the acronym for the Secure Hash Algorithm used to generate it. We'll put these SHAs to use in Section 3.4, and several more advanced Git operations require them as well.

1.3.1 Exercises

1. Using the `touch` command, create empty files called `foo` and `bar` in your repository directory.
2. By using `git add foo`, add `foo` to the staging area. Confirm with `git status` that it worked.
3. Using `git commit -m` and an appropriate message, add `foo` to the repository.
4. By using `git add bar`, add `bar` to staging area. Confirm with `git status` that it worked.
5. Now run `git commit` *without* the `-m` option. Use your Vim knowledge to add the message "Add bar", save, and quit.
6. Using `git log`, confirm that the commits made in the previous exercises worked correctly.

1.4 Viewing the diff

It's often useful to be able to view the changes represented by a potential commit before making it. To see how this works, let's add a little bit of content to **`index.html`** by redirecting the output of **`echo`** to make a "hello, world" page:

```
[website (master)]$ echo "hello, world" > index.html
```

Recall from *Learn Enough Command Line to Be Dangerous* that the Unix **`diff`** utility lets us compare two files **`foo`** and **`bar`** by typing

```
$ diff foo bar
```

Git has a similar function, **`git diff`**, which by default just shows the difference between the last commit and unstaged changes in the current project:

```
[website (master)]$ git diff
diff --git a/index.html b/index.html
index e69de29..4b5fa63 100644
--- a/index.html
+++ b/index.html
@@ -0,0 +1 @@
+hello, world
```

Because the content added in Section 1.3 was empty, here the diff appears simply as an addition:

```
+hello, world
```

We can commit this change by passing the **`-a`** option (for "all") to **`git commit`**, which arranges to commit all the changes in currently existing files (Listing 1.3).

Listing 1.3: Committing changes to all modified files.

```
[website (master)]$ git commit -a -m "Add content to index.html"
[master 03aff34] Add content to index.html
 1 file changed, 1 insertion(+)
```

Note that the `-a` option includes changes only to files already added to the repository, so when there are new files it's important to run `git add -A` as in Section 1.3 to make sure they're added properly. It's easy to get in the habit of running `git commit -a` and forget to add new files explicitly; learning how to deal with this situation is left as an exercise (Section 1.4.1).

Having added and committed the changes, there's now no diff:

```
[website (master)]$ git diff
[website (master)]$
```

(In fact, simply adding the changes is sufficient; running `git add -A` would also lead to there being no diff. To see the difference between staged changes and the previous version of the repo, use `git diff --staged`.)

We can confirm that the change went through by running `git log`:

```
[website (master)]$ git log
commit 03aff34ec4f9690228e057a4252bcca169a868b4
Author: Michael Hartl <michael@michaelhartl.com>
Date:   Thu Dec 17 20:03:33 2015 -0800

    Add content to index.html

commit 879392a6bd8dd505f21876869de99d73f40299cc
Author: Michael Hartl <michael@michaelhartl.com>
Date:   Thu Dec 17 20:00:34 2015 -0800

    Initialize repository
```

1.4.1 Exercises

1. Use `touch` to create an empty file called `baz`. What happens if you run `git commit -am "Add baz"`?

2. Add `baz` to the staging area using `git add -A`, then commit with the message `"Add bazz"`.
3. Realizing there's a typo in your commit message, change `bazz` to `baz` using `git commit --amend`.
4. Run `git log` to get the SHA of the last commit, then view the diff using `git show <SHA>` to verify that the message was amended properly.

1.5 Adding an HTML tag

We've now seen all of the major elements involved in the simplest Git workflow, so in this section and the next we'll review what we've done and see how everything fits together. We'll err on the side of making more frequent commits, representing relatively modest changes, but this isn't necessarily how you should work in real life (Box 1.6). Still, it's an excellent foundation, and it will give you a solid base on which to build your own workflow and development practices.

Box 1.6. Commitment issues

One common issue when learning Git involves figuring out when to make a commit. Unfortunately, there's no simple answer, and real-life usage varies considerably (as illustrated in the xkcd comic strip "Git Commit"). My best advice is to make a commit whenever you've reached a natural stopping point, or when you've made enough changes that you're starting to worry about losing them. In practice, this can lead to inconsistent results, and it's common to work for a while and make a large commit and then make a minor unrelated change with a small commit. This mismatch between commit sizes can seem a little weird, but it's a difficult situation to avoid.

Many teams (including most open-source projects) have their own conventions for commits, including the practice of *squashing* commits to combine them all into one commit for convenience. (Per Box 1.2, this is exactly the kind of thing you can

learn about by Googling for it.) In these circumstances, I recommend following the conventions adopted by the project in question.

More than anything, don't worry about it too much. "Git Commit" is a only a slight exaggeration, and in any case deciding when to commit is the kind of thing that you'll invariably get better at with time and experience.

As in previous sections, we'll be working on the main **index.html** file. Let's start by opening this file in both a text editor and in a web browser. My preferred method for doing this is at the command line using the **atom** and **open** commands (though the latter works only on macOS):

```
[website (master)]$ atom index.html    # or open index.html
```

If you're not on a Mac (or even if you are), you can open the directory using a graphical file browser and double-clicking the file to open it in the default browser (Figure 1.2). However you open the file, the results should appear approximately as shown in Figure 1.3 and Figure 1.4.

At this point, we're ready to make a change, which is to promote "hello, world" from ordinary text to a top-level (Level 1) heading. In HTML, the language of the World Wide Web, the way to do this is with a *tag*—in this case, the Level 1 header tag **h1**. Most browsers set **h1** tags in a large font, so the text **hello, world** should look bigger when we're done. To make the change, replace the current contents of **index.html** with the contents shown in Listing 1.4. (In this and all other examples of editing text, you'll learn more if you type in everything by hand instead of copying and pasting.)

Listing 1.4: A top-level heading.

```
<h1>hello, world</h1>
```

Listing 1.4 shows the basic structure used by most HTML tags. First, there's an *opening tag* that looks like **<h1>**, where the angle brackets **<** and **>** surround

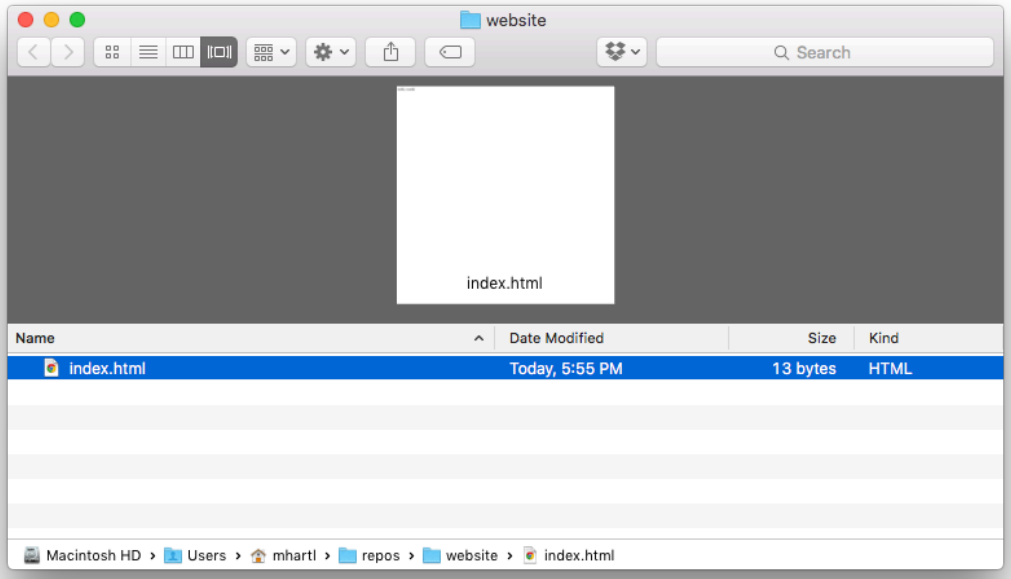

Figure 1.2: Viewing **`index.html`** in a filesystem browser.

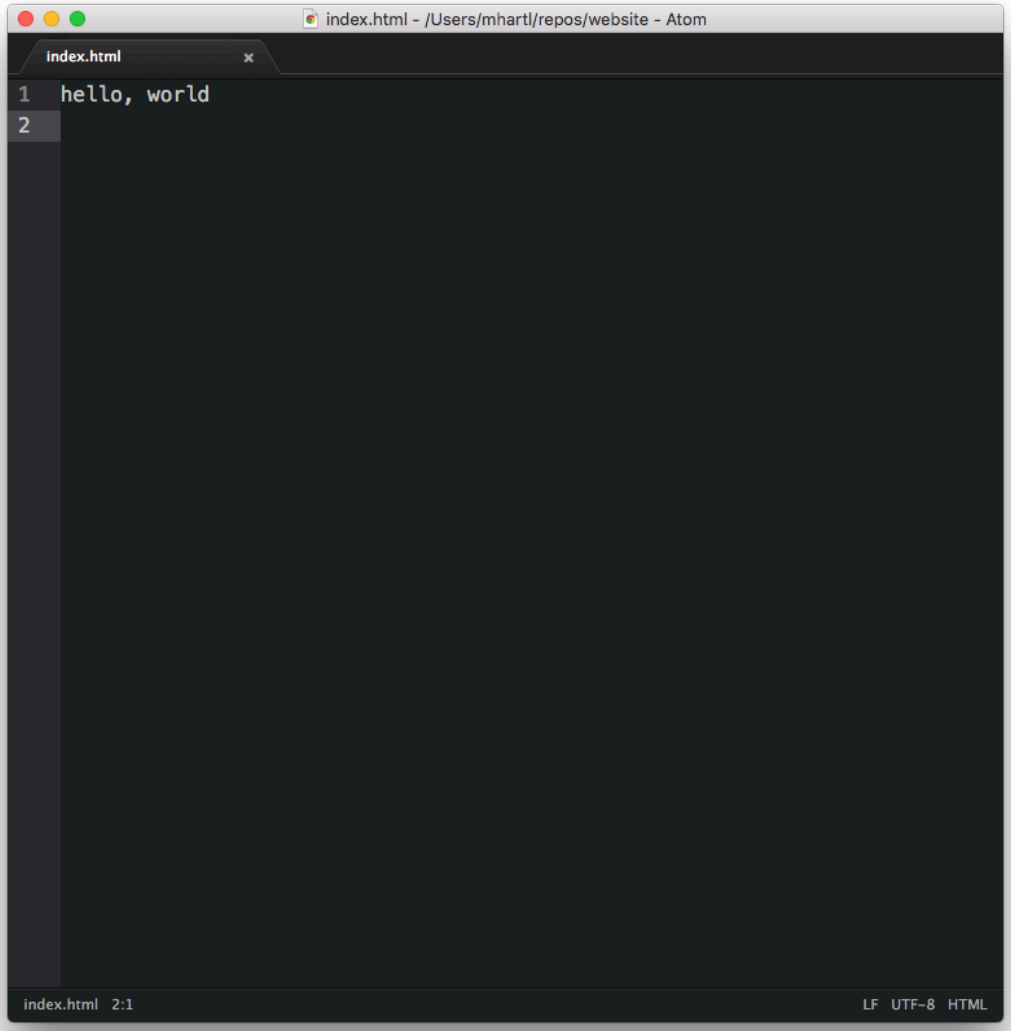

Figure 1.3: The initial HTML file opened in Atom.

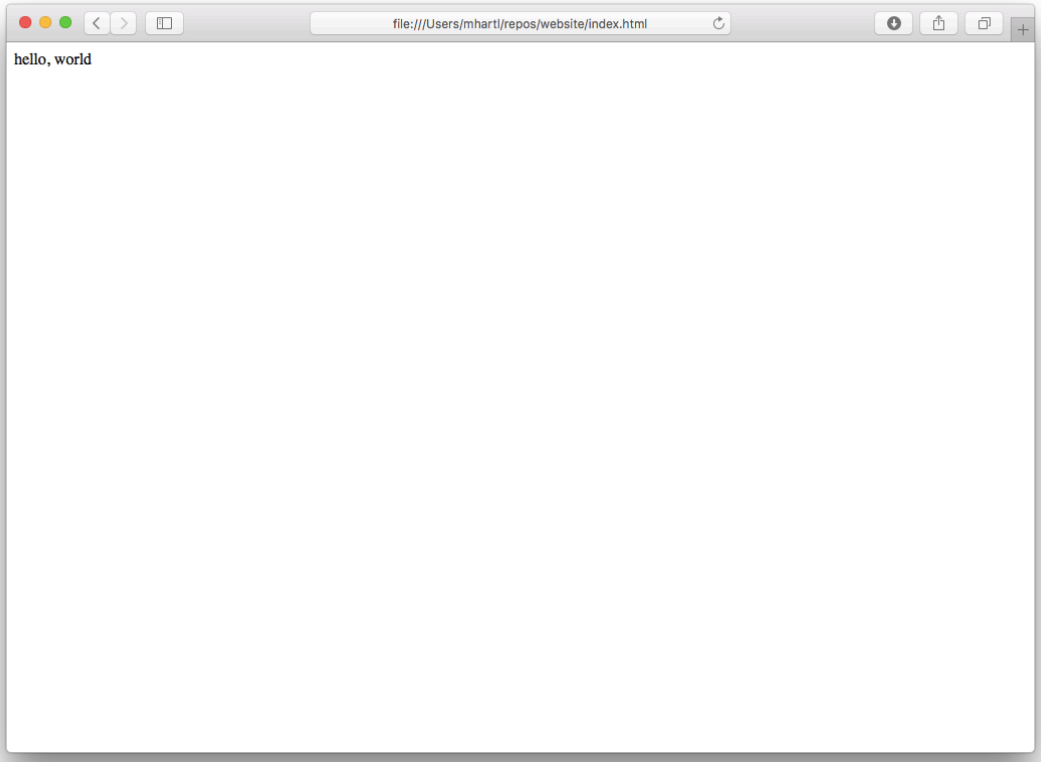

Figure 1.4: The initial HTML file viewed in a web browser.

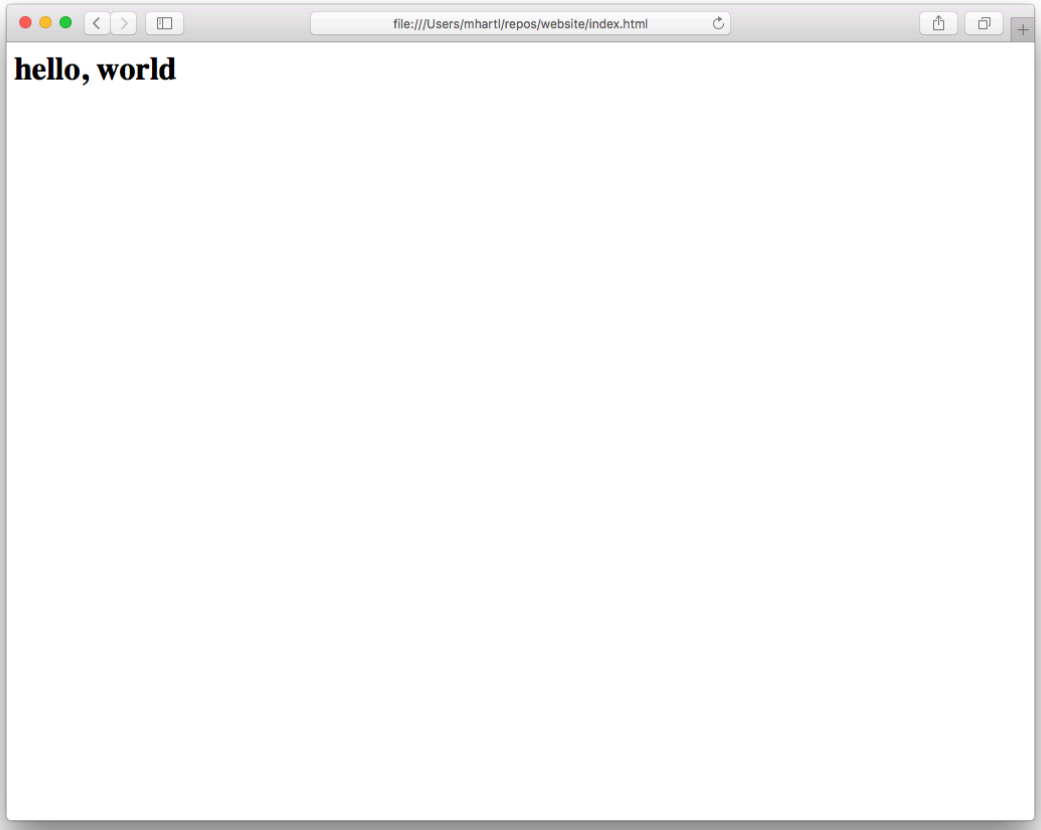

Figure 1.5: The result of adding an `h1` tag.

the tag name (in this case, `h1`). After the content, there's a *closing tag* that's the same as the opening tag, except with an extra slash after the opening angle bracket: `</h1>`. (Note that, as with addresses on the World Wide Web, this is a *slash*, not a *backslash* (a common confusion humourously referenced in the xkcd comic strip "Trade expert").)

Upon refreshing the web browser, the index page should appear something like Figure 1.5. As promised, the font size of the text for the top-level heading is bigger (and bolder, too).

As before, we'll run `git status` and `git diff` to learn more about what

we're going to commit to Git, though with experience you'll come to run these commands only when necessary. The status simply indicates that `index.html` has been modified:

```
[website (master)]$ git status
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:   index.html

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

Meanwhile, the diff shows that one line has been deleted (indicated with `-`) and another added (indicated with `+`):

```
[website (master)]$ git diff
diff --git a/index.html b/index.html
index 4b5fa63..45d754a 100644
--- a/index.html
+++ b/index.html
@@ -1 +1 @@
-hello, world
+<h1>hello, world</h1>
```

As with the Unix `diff` utility, modified sections of code or markup are shown as close to each other as possible so that it's clear at a glance what changed.[9]

At this point, we're ready to commit our changes. In Listing 1.3 we used both the `-a` and `-m` options to commit **a**ll pending changes while adding a commit **m**essage, but in fact the two can be combined as `-am` (Listing 1.5).

Listing 1.5: Committing with `-am`.

```
[website (master)]$ git commit -am "Add an h1 tag"
```

Using the `-am` combination as in Listing 1.5 is common in idiomatic Git usage.

[9] When viewing small diffs, particularly in prose, the `--color-words` option is especially useful, so if the regular diff is hard to read I recommend trying `git diff --color-words` to see the effect. (This option also works with the regular Unix `diff` program.)

1.5.1 Exercises

1. The `git log` command shows only the commit messages, which makes for a compact display but isn't particularly detailed. Verify by running `git log -p` that the `-p` option shows the full diffs represented by each commit.

2. Under the `h1` tag in Listing 1.4, use the `p` tag to add a *paragraph* consisting of the line "Call me Ishmael." The result should appear as in Figure 1.6. (Don't worry if you get stuck; we'll incorporate the answer to this exercise in Section 1.6 (Listing 1.6).)

1.6 Adding HTML structure

Although the web browser correctly rendered the `h1` tag in Figure 1.5, properly formatted HTML pages have more structure than just bare `h1` or `p` tags. In particular, each page should have an `html` tag consisting of a *head* and a *body* (identified with `head` and `body` tags, respectively), as well as a "doctype" identifying the document type, which in this case is a particular version of HTML called HTML5. (Don't worry about these details now; we'll cover them in more depth in *Learn Enough HTML to Be Dangerous*.)

Applying these general considerations to `index.html` leads to the full HTML structure shown in Listing 1.6. This includes the `h1` tag from Listing 1.4 and the paragraph tag from Figure 1.6. (The `title` tag, included inside the `head` tag, is empty, but in general every page should have a title, and adding one for `index.html` is left as an exercise (Section 1.6.1).)

Listing 1.6: The HTML page with added structure.

```
<!DOCTYPE html>
<html>
  <head>
    <title></title>
  </head>
  <body>
```

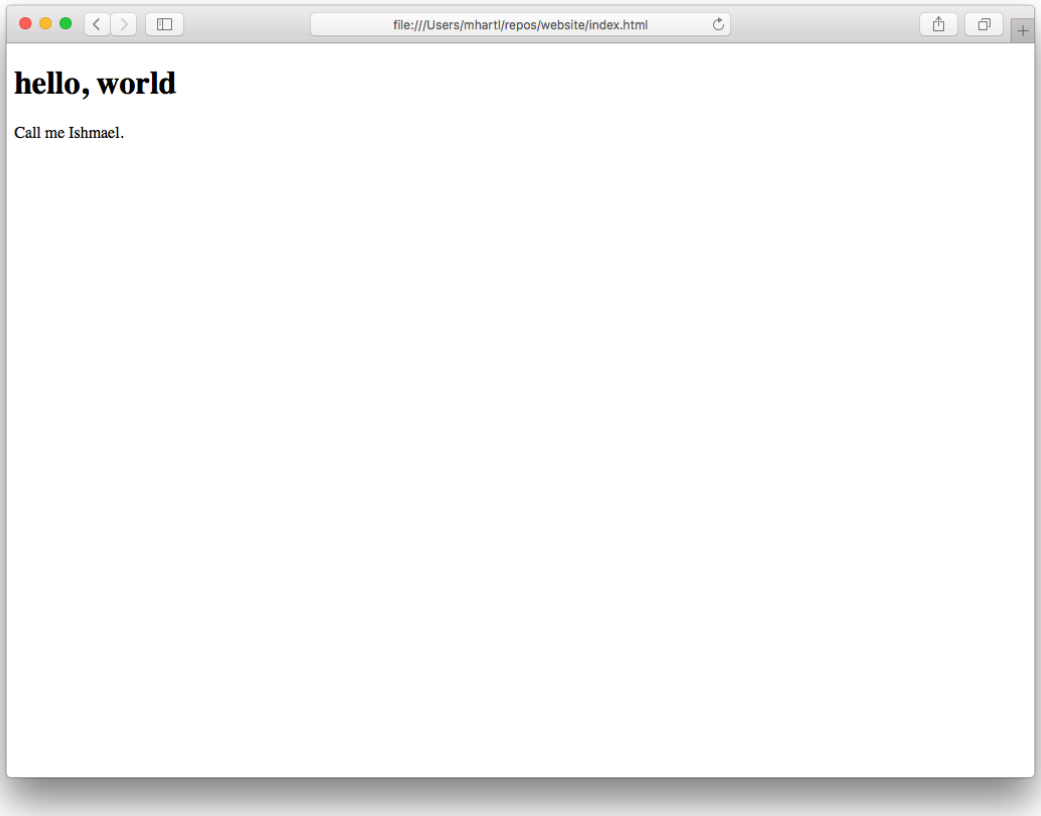

Figure 1.6: The result of adding a short paragraph.

```
7       <h1>hello, world</h1>
8       <p>Call me Ishmael.</p>
9     </body>
10  </html>
```

Because this is a lot more content than our previous iteration (Listing 1.4), it's a good idea to go through it line by line:

1. The document type declaration
2. Opening **`html`** tag
3. Opening **`head`** tag
4. Opening and closing **`title`** tags
5. Closing **`head`** tag
6. Opening **`body`** tag
7. Top-level heading
8. Paragraph from the exercises (Section 1.5.1)
9. Closing **`body`** tag
10. Closing **`html`** tag

As usual, we can see the changes represented by our addition using **`git diff`** (Listing 1.7).

Listing 1.7: The diff for adding HTML structure.

```
[website (master)]$ git diff
diff --git a/index.html b/index.html
index 4b5fa63..afcd202 100644
--- a/index.html
+++ b/index.html
@@ -1 +1,10 @@
-<h1>hello, world</h1>
```

```
+<!DOCTYPE html>
+<html>
+  <head>
+    <title></title>
+  </head>
+  <body>
+    <h1>hello, world</h1>
+    <p>Call me Ishmael.</p>
+  </body>
+</html>
```

Despite the extensive diffs in Listing 1.7, there are hardly any user-visible differences (Figure 1.7); the only change from Figure 1.6 is a small amount of space above the top-level heading. The structure is much better, though, and brings our page nearly into compliance with the HTML5 standard. (It's not quite valid, because a nonblank page title is required by the standard; fixing this issue is left as an exercise (Section 1.6.1).)

Since we haven't added any files, using **`git commit -am`** suffices to commit all the changes (Listing 1.8).

Listing 1.8: The commit to add the HTML structure.

```
[website (master)]$ git commit -am "Add some HTML structure"
```

1.6.1 Exercises

1. Add the title "A whale of a greeting" to **`index.html`**. Browsers differ in how they display titles; the result in Google Chrome is shown in Figure 1.8.

2. Commit the new title with a commit message of your choice. Verify using **`git log -p`** that the change was committed as expected.

3. By pasting the contents of Listing 1.6 into an HTML validator, verify that it is *not* (quite) a valid web page.

4. Using the validator, verify that the current **`index.html`** (with nonblank page title) *is* valid.

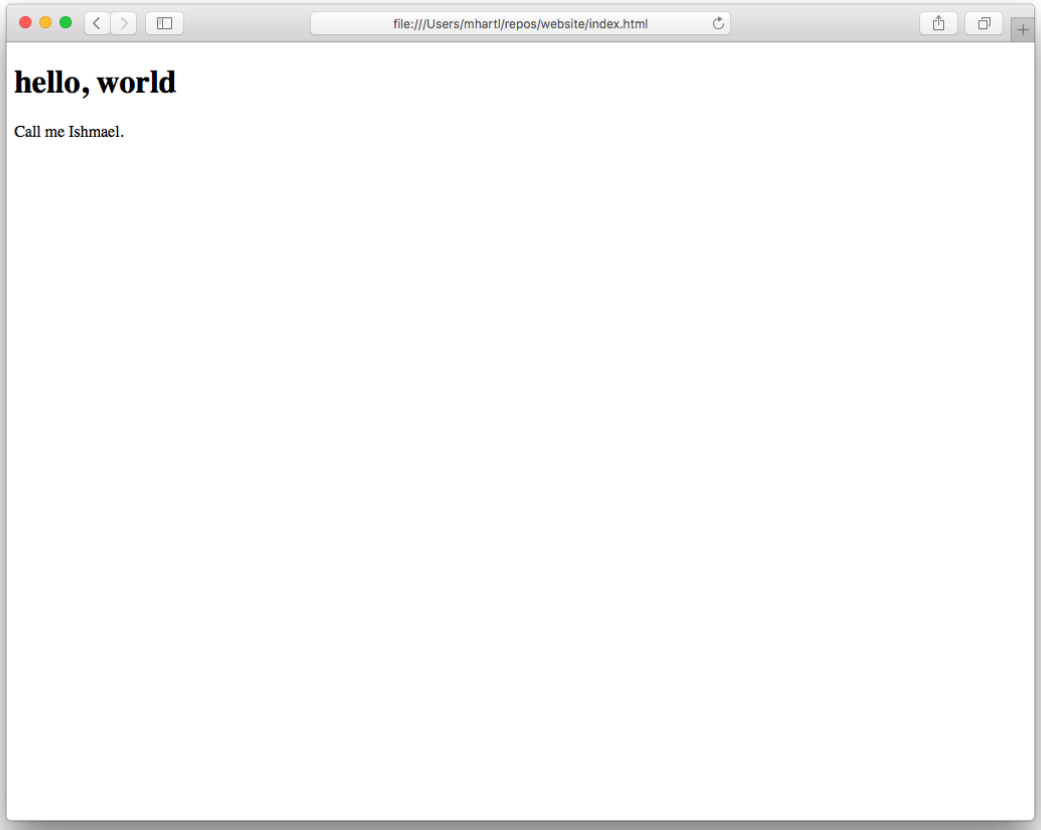

Figure 1.7: Adding HTML structure makes hardly any difference in the appearance.

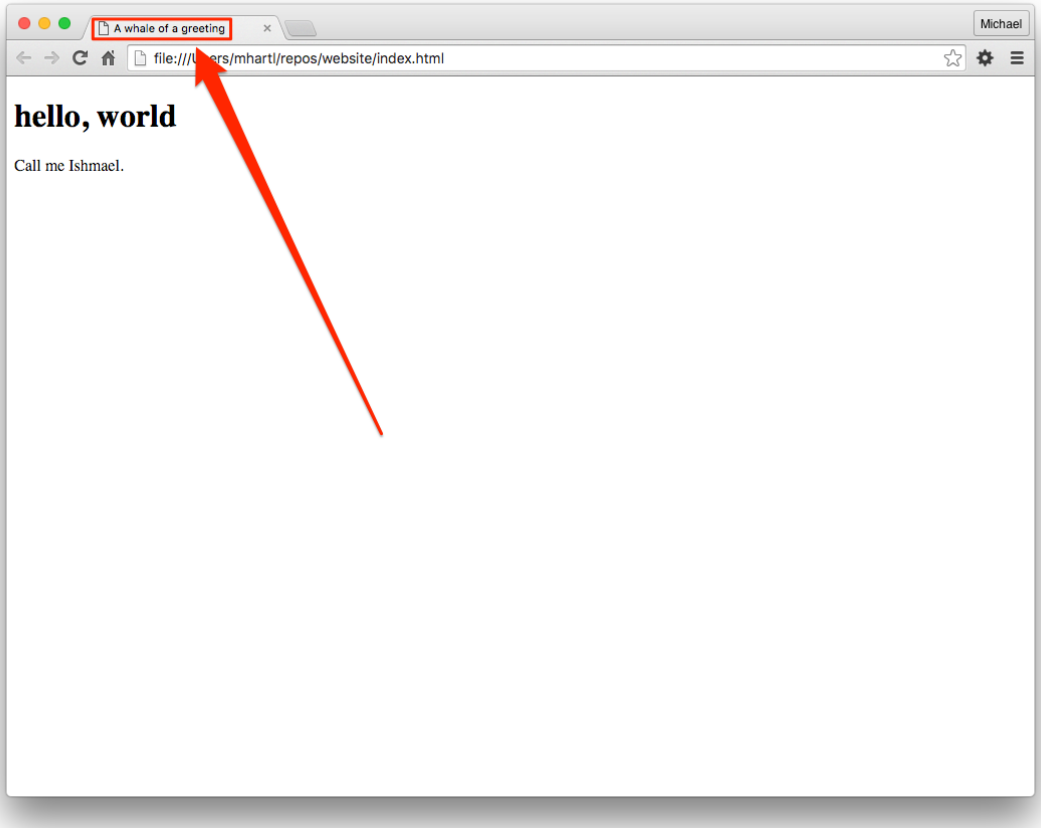

Figure 1.8: The page title displayed in a browser.

1.7 Summary

Important commands from this section are summarized in Table 1.1.

Command	Description	Example
`git help`	Get help on a command	`$ git help push`
`git config`	Configure Git	`$ git config --global …`
`source <file>`	Activate shell changes	`$ source ~/.bash_profile`
`mkdir -p`	Make intermediate directories as necessary	`$ mkdir -p repos/website`
`git status`	Show the status of the repository	`$ git status`
`touch <name>`	Create empty file	`$ touch foo`
`git add -A`	Add all files or directories to staging area	`$ git add -A`
`git add <name>`	Add given file or directory to staging area	`$ git add foo`
`git commit -m`	Commit staged changes with a message	`$ git commit -m "Add thing"`
`git commit -am`	Stage and commit changes with a message	`$ git commit -am "Add thing"`
`git diff`	Show diffs between commits, branches, etc.	`$ git diff`
`git commit --amend`	Amend the last commit	`$ git commit --amend`
`git show <SHA>`	Show diff vs. the SHA	`$ git show fb738e…`

Table 1.1: Important commands from Chapter 1.

Chapter 2

Backing up and sharing

With the changes made in Chapter 1, we're now ready to push a copy of our project to a *remote repository*. This will serve as a backup of our project and its history, and will also make it easier for collaborators to work with us on our site.

We'll start by pushing our project up to *GitHub*, a site designed to facilitate collaboration with Git repositories. For repositories that are publicly available, GitHub has always been free, so we'll plan to make our website's repo public to take advantage of this. (When this tutorial was first written, GitHub charged for private repositories; in Section 4.4.1, we'll discuss an alternative that has always allowed unlimited free private repos.) Over time, releasing projects publicly on GitHub serves to build up a portfolio, which is one good reason to make as much work public as possible. There's also a Secret Reason™ for adding our repo to GitHub, which we'll get to in Section 4.4.

For reference, important commands from this section are summarized in Section 2.4.

2.1 Signing up for GitHub

If you don't already have a GitHub account, you can get started by visiting the GitHub signup page (Figure 2.1) and following the instructions. Use your technical sophistication (Box 1.2) if you get stuck.

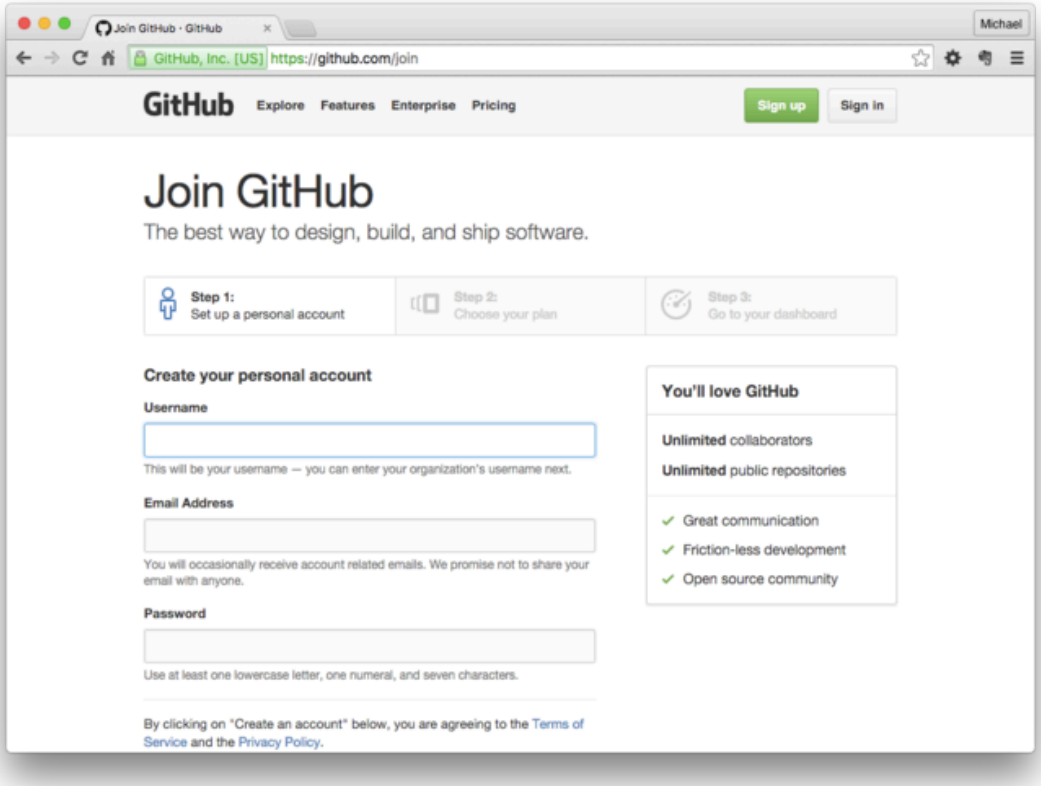

Figure 2.1: Joining GitHub.

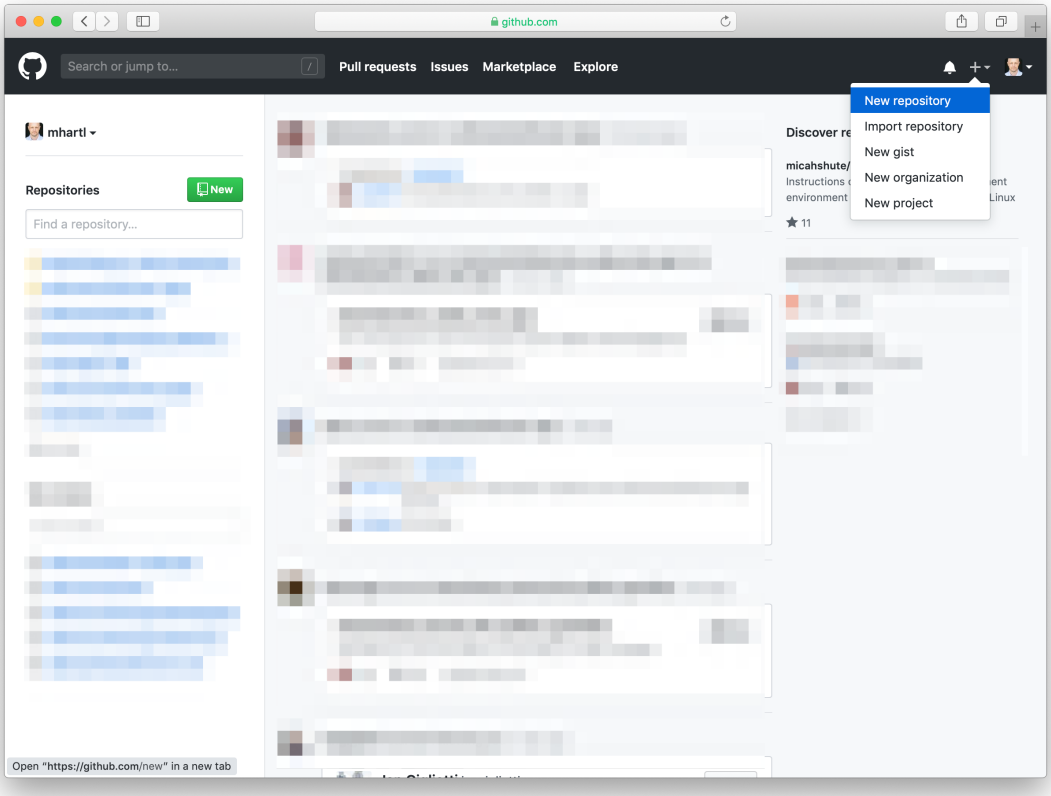

Figure 2.2: Adding a new repository at GitHub.

2.2 Remote repo

After signing up for a GitHub account, the next step is to create a remote repository. Start by selecting the menu item for adding a new repository, as shown in Figure 2.2, and then fill in the repository name ("website") and description ("A sample website for Learn Enough Git to Be Dangerous") as shown in Figure 2.3. GitHub actively develops its user interface, so Figure 2.2, Figure 2.3, and other GitHub screenshots may not match your results exactly, but this is no cause for concern. As usual, apply your technical sophistication (Box 1.2) to resolve any discrepancies.

After clicking the green "Create repository" button seen in Figure 2.3, you

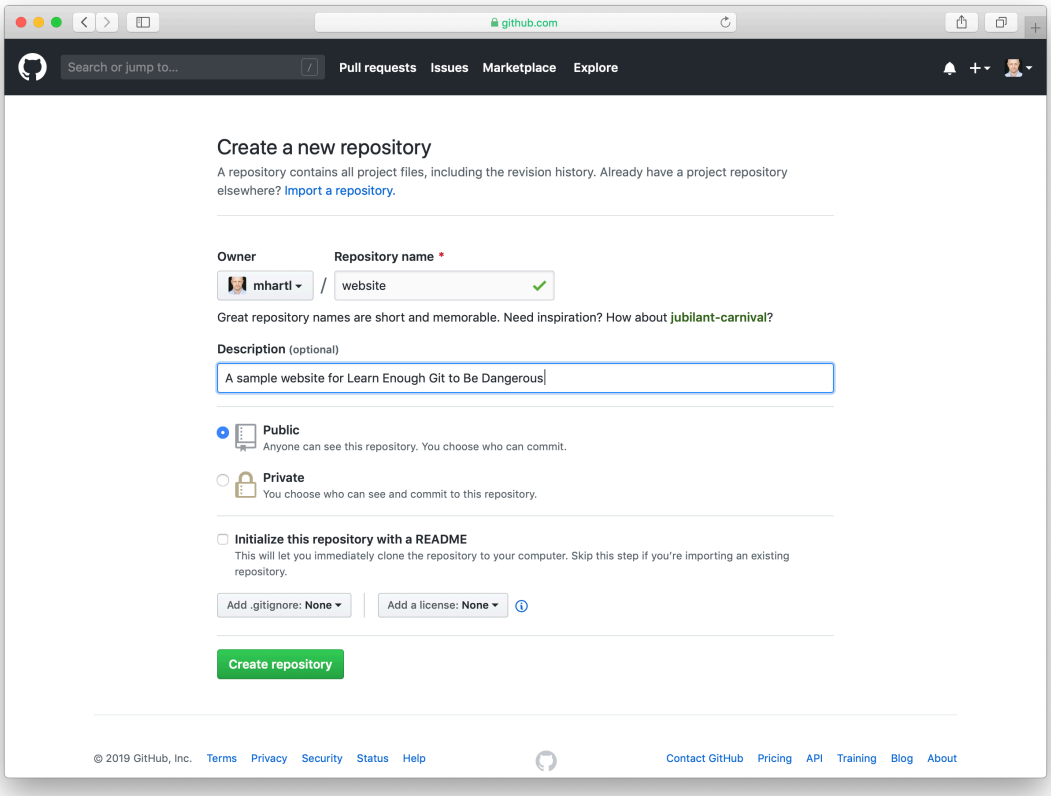

Figure 2.3: Creating a new repository.

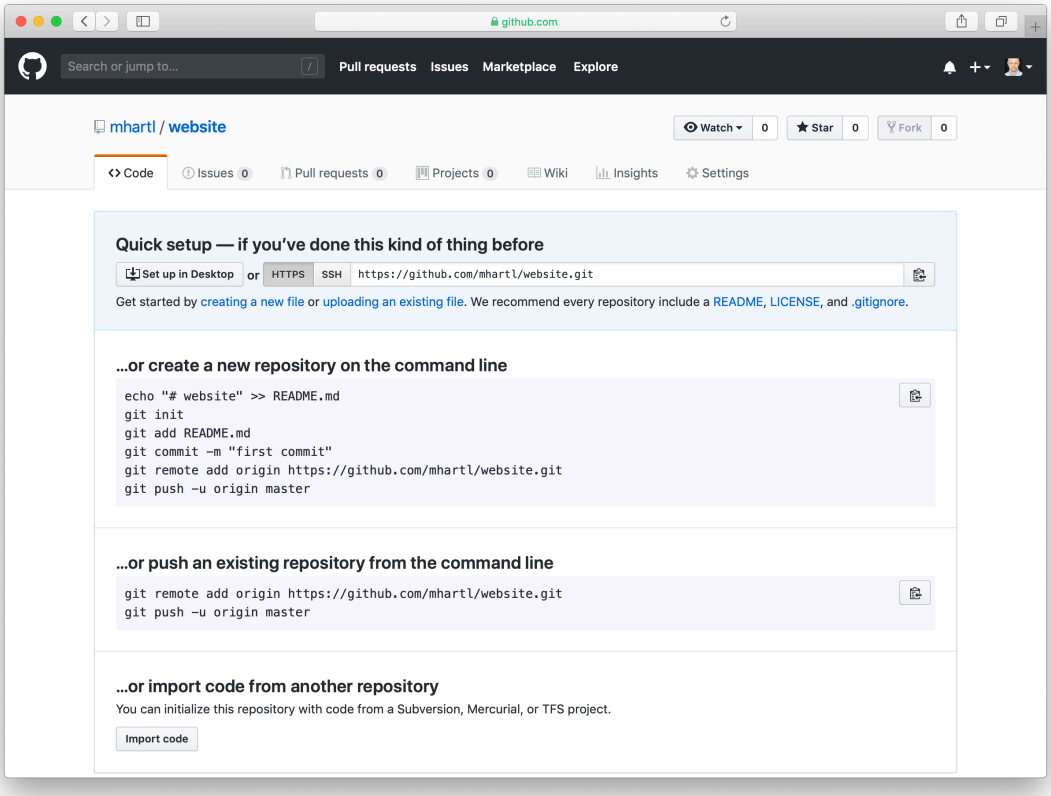

Figure 2.4: Instructions for pushing up the repo.

should see a page like Figure 2.4 containing instructions for how to *push* your local repository up to GitHub.

The exact commands in Figure 2.4 will be tailored to your personal account name, with a template that looks like Listing 2.1. (It is not important to understand these commands at this time.)

Listing 2.1: A template for the first push to GitHub.

```
[website (master)]$ git remote add origin https://github.com/<name>/website.git
[website (master)]$ git push -u origin master
```

Of course, you should replace **`<name>`** with your actual username. For example,

Figure 2.5: The browser reload page button.

the commands for my username, which is **mhartl**, look like this:

```
[website (master)]$ git remote add origin https://github.com/mhartl/website.git
[website (master)]$ git push -u origin master
```

The two commands in Listing 2.1 first set GitHub as the *remote origin* and then *push* the full repository. The **-u** option to **git push** sets GitHub as the *upstream repository*, which means we'll be able to download any changes automatically when we run **git pull** starting in Section 4.1. Don't worry about these details, though; you will almost always copy such commands from GitHub and probably won't ever have to figure them out on your own.

It's important to note that Figure 2.4 has a repository URL that uses HTTPS, the secure version of the HyperText Transfer Protocol (HTTP). This is the current GitHub default, but there's another version that uses so-called SSH Keys, which has the advantage of remembering your password automatically. We'll stick with HTTPS for now, since it's simpler to use and configure. The biggest downside is that you have to type your password every time you want to push any changes, which can be inconvenient. Luckily, there are ways to get your computer to remember, or *cache*, your password; see the article "Caching your GitHub password in Git" for information on how to set this up on your system.

After executing the first **git push** as shown in Listing 2.1, you should reload the current page (using, e.g., ⌘R or the icon shown in Figure 2.5). The result should look something like Figure 2.6. If it does, you have officially shipped your first Git repository!

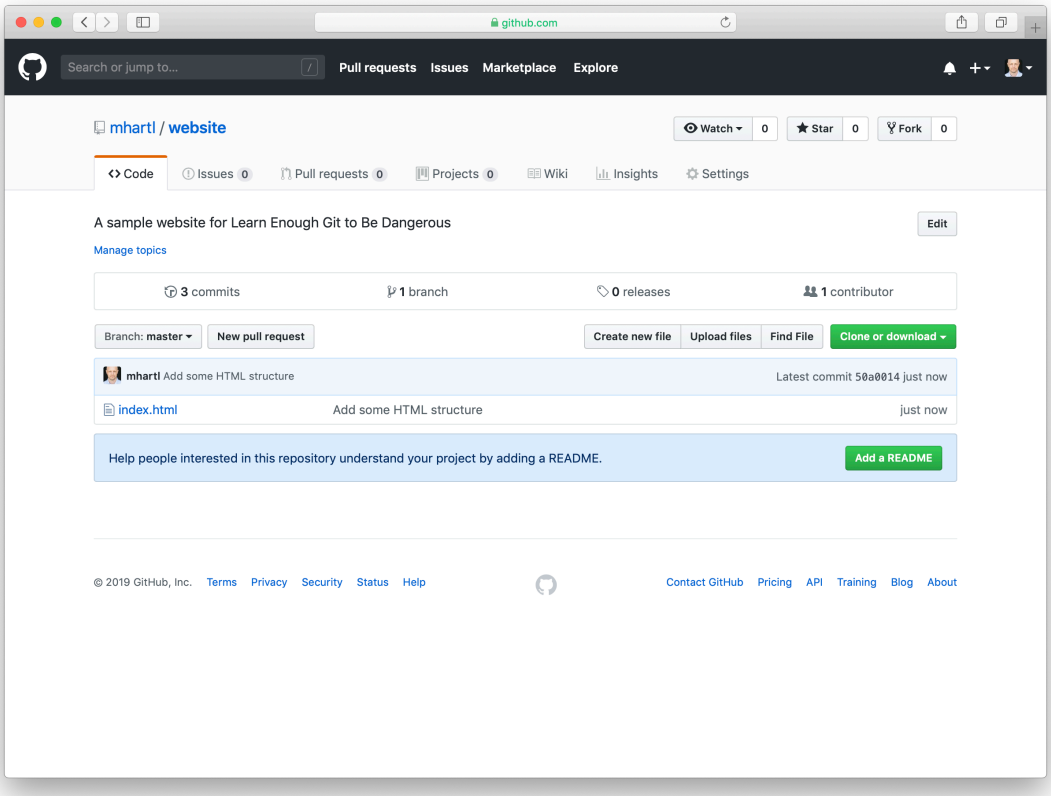

Figure 2.6: The remote repository at GitHub.

2.2.1 Exercises

1. On the GitHub page for your repo, click on "Commits" to see a list of your commits. Confirm that they match the results of running `git log` on your local system.

2. At GitHub, click on the commit for adding HTML structure (Listing 1.8). Verify that the diff for the commit agrees with the one shown in Listing 1.7.

3. In honor of shipping your first Git repo, drink a celebratory beverage of your choice (Figure 2.7).[1]

2.3 Adding a README

Now that we've pushed up our repository, let's add a second file and practice the `add`, `commit`, and `push` sequence shown in Figure 1.1. You may have noticed in Figure 2.6 that GitHub encourages the presence of a README file via the note "Help people interested in this repository understand your project by adding a README." Such a file literally asks the viewer to "READ ME", *à la* the DRINK ME bottle from *Alice's Adventures in Wonderland* (Figure 2.8),[2] and it's a good practice to include one.

Figure 2.6 shows a green `Add a README` button that GitHub includes to make it easy to add a README file through the web interface, but we'll follow the common (and more instructive) practice of adding it by hand locally and then pushing it up. When it comes to rendering and displaying READMEs, GitHub supports several common formats, but my favorite format for short documents like READMEs is Markdown, a lightweight markup language discussed before in *Learn Enough Text Editor to Be Dangerous*.

[1] Image retrieved from http://retrovintage-vector.blogspot.com/2011/09/retro-women-vector-drinks-coffee.html on 2019-02-15 and is in the public domain.

[2] *Alice's Adventures in Wonderland* original illustrations by John Tenniel. Image retrieved from https://cellcode.us/quotes/colored-book-alice-wonderland-original-drawings.html on 2019-02-15. Copyright © 1890, now in the public domain.

Figure 2.7: Shipping a project often calls for a celebratory beverage.

Figure 2.8: Alice would know to read a README file.

We can get started by opening **README.md** in Atom (or any other text editor), where the **.md** extension identifies the file as Markdown:

```
[website (master)]$ atom README.md
```

We can then fill it with the content shown in Listing 2.2.

Listing 2.2: The contents of the README file.
~/repos/website/README.md

```
# Sample Website

This is a sample website made as part of [*Learn Enough™ Git to Be
Dangerous*](https://www.learnenough.com/git-tutorial), possibly the greatest
beginner Git tutorial in the history of the Universe. You should totally [
check it out](https://www.learnenough.com/git-tutorial), and be sure to [join
the email list](https://www.learnenough.com/#email_list) and
[follow @learnenough](http://twitter.com/learnenough) on Twitter.

After finishing *Learn Enough™ Git to Be Dangerous*, you'll know enough Git
to be *dangerous*. This means you'll be able to use Git to track changes in
your projects, back up data, share your work with others, and collaborate
with programmers and other users of Git.
```

The result in Atom appears as shown in Figure 2.9. As mentioned in *Learn Enough Text Editor to Be Dangerous*, Atom includes a Markdown previewer via the Packages menu item shown in Figure 2.10, which (after resizing the window) results in the preview shown in Figure 2.11.[3]

Now that we've created the **README.md** file, we're ready to add it to our Git repository and push it up. We can't just run **git commit -am** because **README.md** isn't currently in the repository, so we have to add it first:

```
[website (master)]$ git add -A
```

(As noted in Section 1.3.1, we could also run **git add README.md**, but in most cases we want to add all the new files, so I suggest getting in the habit of

[3] Atom comes with a built-in Markdown previewer, but recall from *Learn Enough Text Editor to Be Dangerous* that editors such as Sublime Text often have installable Markdown Preview packages as well.

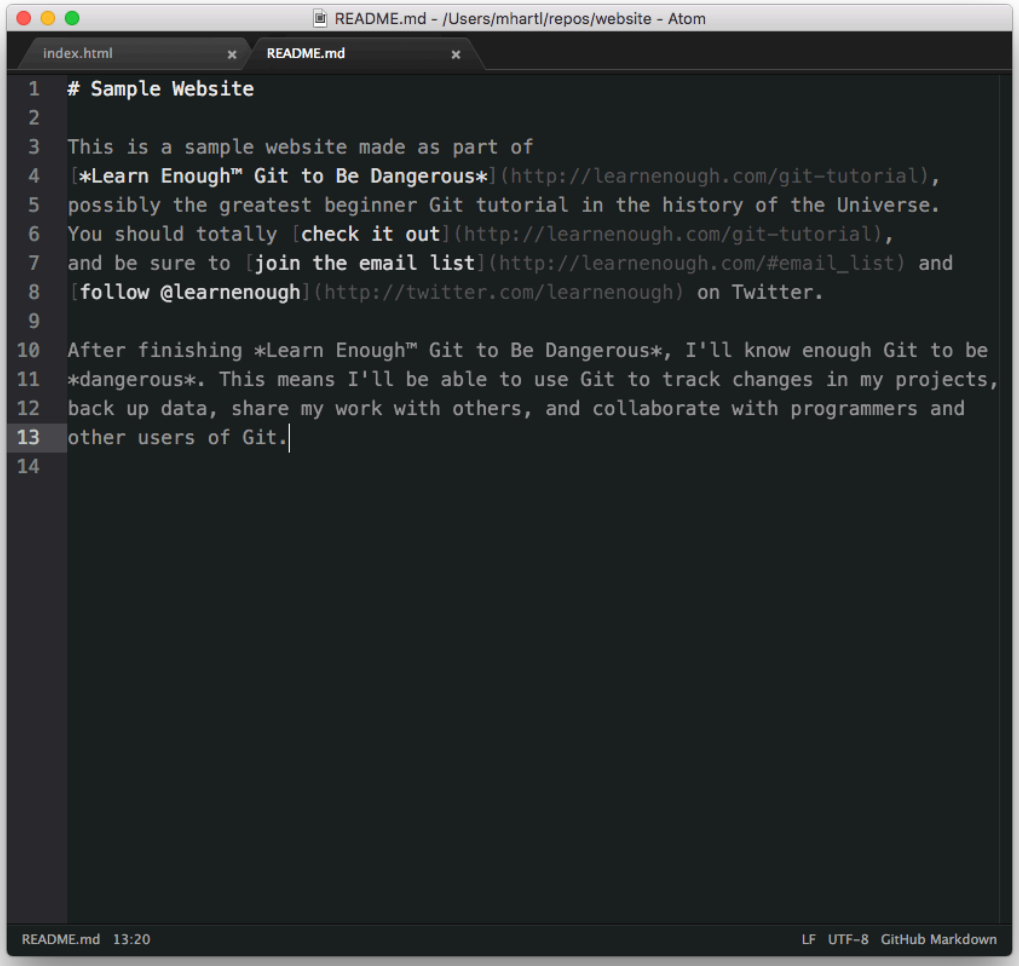

Figure 2.9: The README file viewed in Atom.

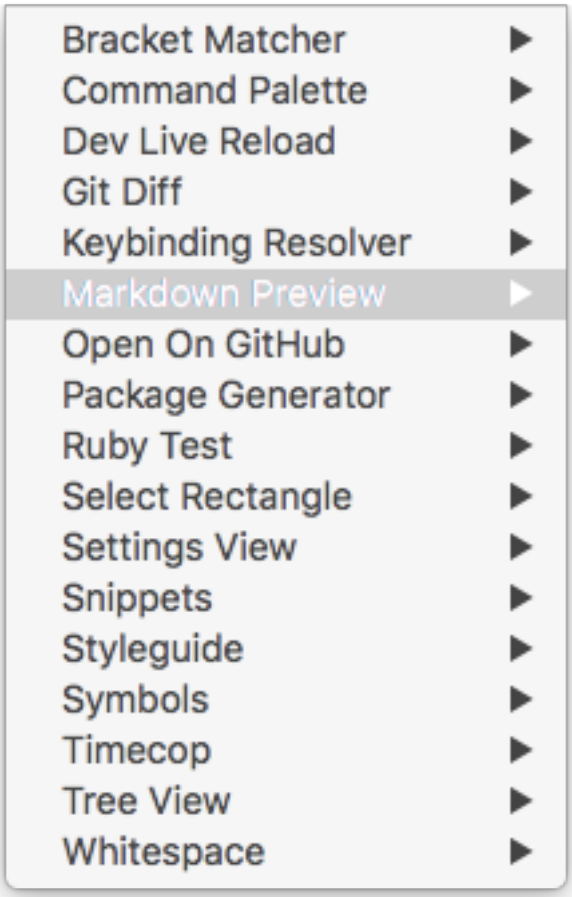

Figure 2.10: The Packages menu item for toggling the Markdown preview.

Figure 2.11: The resized Atom window with a Markdown preview.

running **`git add -A`** unless there's a specific reason not to.) Then we commit as usual:

```
[website (master)]$ git commit -m "Add README file"
```

By the way, there's no harm in including **`-a`** via the **`-am`** combination shown in Listing 1.5 (and despite the redundancy I often do so out of habit), so this could just as easily read **`git commit -am "Add a README file"`**. (The call to **`git add`** is still necessary, though; recall from Section 1.4 that **`git commit -a`** by itself commits changes only to files that Git is already tracking and have been modified.)

Having added the file to the repository and made a commit, we're now ready to push up to GitHub. Recall from Listing 2.1 that the first occurrence of **`git push`** included the "set upstream" option **`-u`**, the destination **`origin`**, and **`master`**, but once these are set up we can omit all those details and just **`push`**, like this:

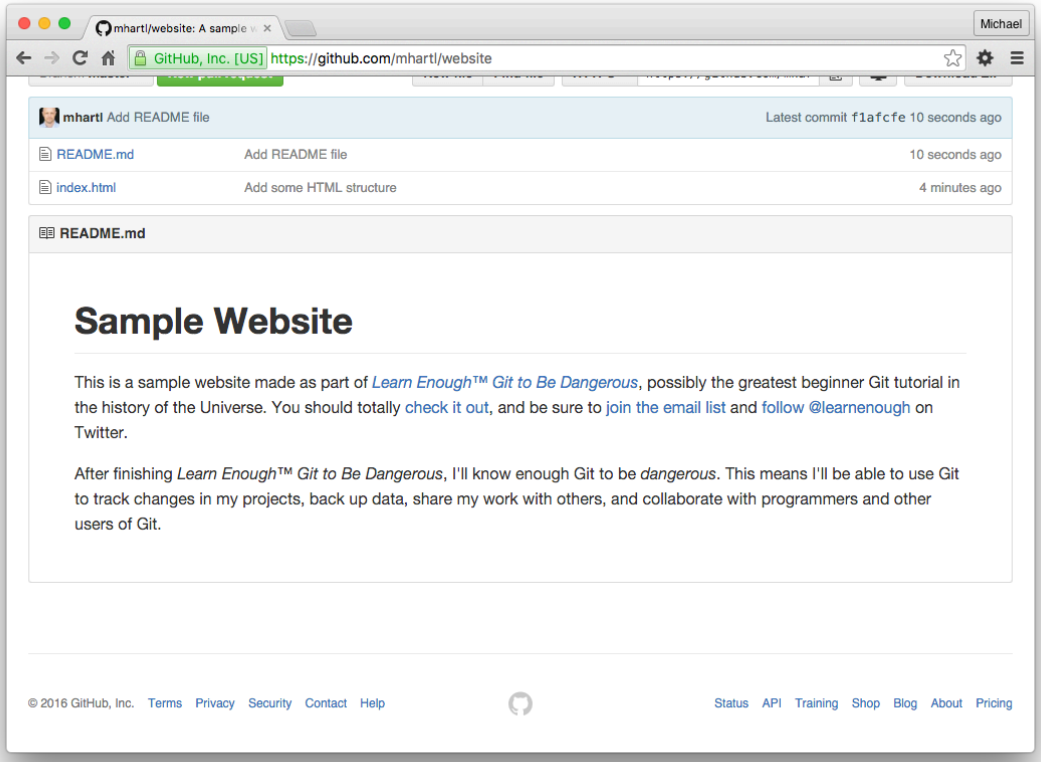

Figure 2.12: The README file at GitHub.

```
[website (master)]$ git push
```

The result of this is to push up the new README to the remote repository, which means that we've completed the full sequence shown in Figure 1.1. In this case, GitHub uses the `.md` extension to identify the file as Markdown, converting it to HTML for easy viewing,[4] as shown in Figure 2.12.

[4] This involves converting the `#` in Listing 2.2 to a top-level heading (the `h1` we first saw in Section 1.5) and converting each Markdown link of the form `[content](address)` into an HTML *anchor* tag `a`, which we'll meet in Section 3.3.

Command	Description	Example
`git remote add`	Add remote repo	`$ git remote add origin`
`git push -u <loc>  `	Push branch to remote	`$ git push -u origin master`
`git push`	Push to default remote	`$ git push`

Table 2.1: Important commands from Chapter 2.

2.3.1 Exercises

1. Using the Markdown shown in Listing 2.3, add a line at the end of the README with a link to the official Git documentation.

2. Commit your change with an appropriate message (Box 1.5). You don't have to run `git add`. Why not?

3. Push your change to GitHub. By refreshing your browser, confirm that the new line has been added to the rendered README. Click on the "official Git documentation" link to verify that it works.

Listing 2.3: Markdown code for adding a link to the official Git documentation.
~/repos/website/README.md

```
For more information on Git, see the
[official Git documentation](https://git-scm.com/).
```

2.4 Summary

Important commands from this section are summarized in Table 2.1.

Chapter 3

Intermediate workflow

In this chapter, we'll practice and extend the basic workflow introduced in Section 2.3. This will include adding a new directory to our project, learning how to tell Git to ignore certain files, how to *branch* and *merge*, and how to recover from errors. Rather than providing an encyclopedic coverage of Git's many commands, our focus is on covering practical techniques used every day by software developers and other users of Git.

For reference, important commands from this chapter are summarized in Section 3.5.

3.1 Commit, push, repeat

We'll start by adding an image to our site, which involves making a change to an existing file (`index.html`) while adding a new file in a new directory. The first step is to make a directory for images:

```
[website (master)]$ mkdir images
```

Next, download the image shown in Figure 3.1[1] to the local directory using `curl`:

[1] Image retrieved from https://www.flickr.com/photos/28883788@N04/10097824543 on 2015-12-26. Copyright © 2013 by Denis Hawkins and used unaltered under the terms of the Creative Commons Attribution-NoDerivs 2.0 Generic license.

```
$ curl -o images/breaching_whale.jpg \
>      -OL https://cdn.learnenough.com/breaching_whale.jpg
```

We're now ready to include the image in our index page using the *image tag* **img**. This is a new kind of HTML tag; before we had opening and closing tags like

```
<p>content</p>
```

but the image tag is different. Unlike tags like **h1** and **p**, the **img** tag is a *void element* (also called a *self-closing tag*), which means that it starts with **<img** and ends with **>**:

```
<img src="path/to/file" alt="Description">
```

Note that **img** has no content between tags because there's no "between"; instead, it has a path to the *source* of the image, indicated by **src**. An alternate syntax uses **/>** instead of **>** in order to conform to constraints of XML, a markup language related to HTML:

```
<img src="path/to/file" alt="Description" />
```

You might sometimes see this syntax instead of the plain **>**, but in HTML5 the two are exactly equivalent.

By the way, in the example above the path **path/to/file** is *meta*, meaning that it talks *about* the path rather than referring to the literal path itself. In such cases, it's important to use the actual path to the file. (Successfully navigating such meta usage is a good sign of increasing technical sophistication (Box 1.2).) In this case, the path is **images/breaching_whale.jpg**, so the **img** tag in **index.html** should appear as shown in Listing 3.1. (This image tag is actually missing something important, which we'll add in Section 4.2.)

Figure 3.1: An image to include in our website.

Listing 3.1: Adding an image to the index page.
~/repos/website/index.html

```
<!DOCTYPE html>
<html>
  <head>
    <title>A whale of a greeting</title>
  </head>
  <body>
    <h1>hello, world</h1>
    <p>Call me Ishmael.</p>
    <img src="images/breaching_whale.jpg">
  </body>
</html>
```

Refreshing the browser then gives the result shown in Figure 3.2. (Note that Listing 3.1 includes the **`title`** tag content, thereby incorporating the solution to an exercise in Section 1.6.1.)

At this point, **`git diff`** confirms that the image addition is ready to go:

```
[website (master)]$ git diff index.html
diff --git a/index.html b/index.html
index 706a1be..74043f7 100644
--- a/index.html
+++ b/index.html
@@ -6,5 +6,6 @@
   <body>
     <h1>hello, world</h1>
     <p>Call me Ishmael.</p>
+    <img src="images/breaching_whale.jpg">
   </body>
 </html>
```

On the other hand, running **`git status`** shows that the entire **`images/`** directory is untracked:

```
[website (master)]$ 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)
```

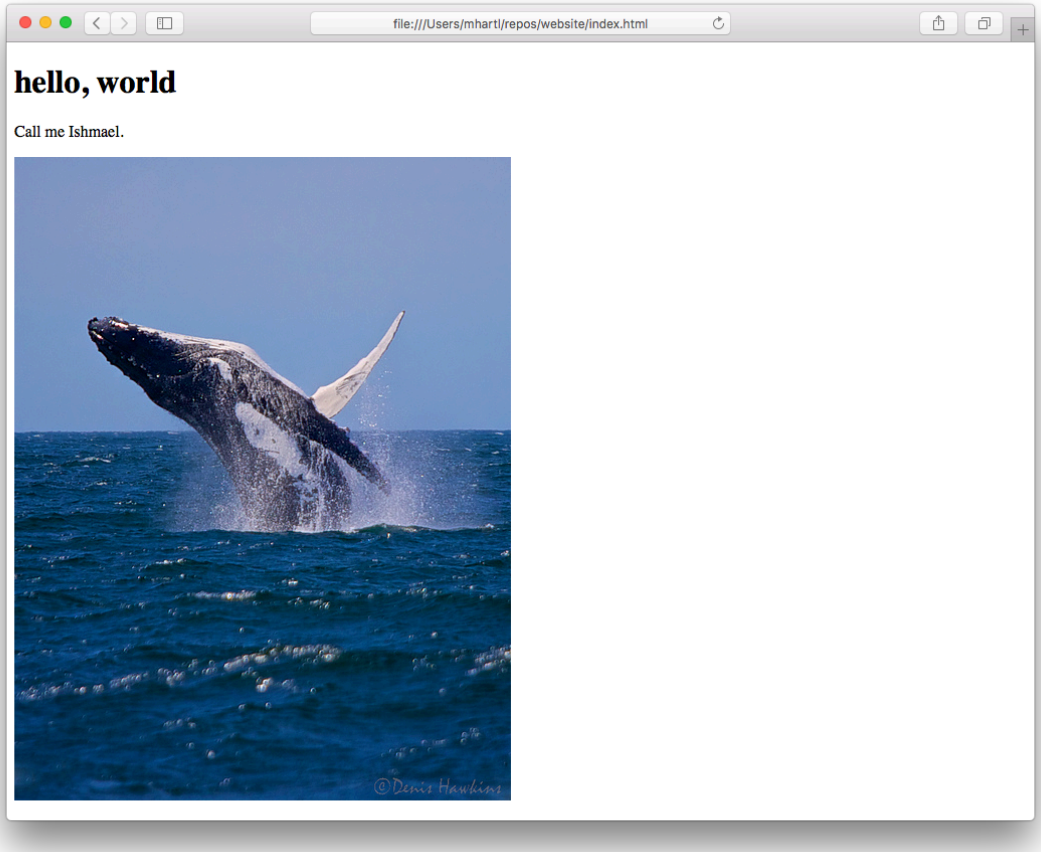

Figure 3.2: Our website with an added image.

```
  modified:   index.html

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

  images/

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

As you might guess, **`git add -A`** adds all untracked *directories* in addition to adding all untracked files, so we can add the image and its directory with a single command:[2]

```
[website (master)]$ git add -A
```

We then commit and push as usual:

```
[website (master)]$ git commit -m "Add an image"
[website (master)]$ git push
```

It's a good idea to get in the habit of pushing up to the remote repository frequently, as it serves as a guaranteed backup of the project while also allowing collaborators to pull in any changes (Chapter 4).

After refreshing the GitHub repository in your browser, you should be able to confirm the presence of the new file by clicking on the **`images`** directory link, with the results as shown in Figure 3.3.

3.1.1 Exercises

1. Click on the image link at GitHub to verify that the **`git push`** succeeded.

[2] Technically, Git tracks only files, not directories; in fact, it won't track empty directories at all, so if you want to track an otherwise empty directory you need to put a file in it. One common convention is to use a hidden file called **`.gitkeep`**; to create this file in an empty directory called **`foo`**, you can use the command **`touch foo/.gitkeep`**. Then **`git add -A`** will add the **`foo`** directory as desired.

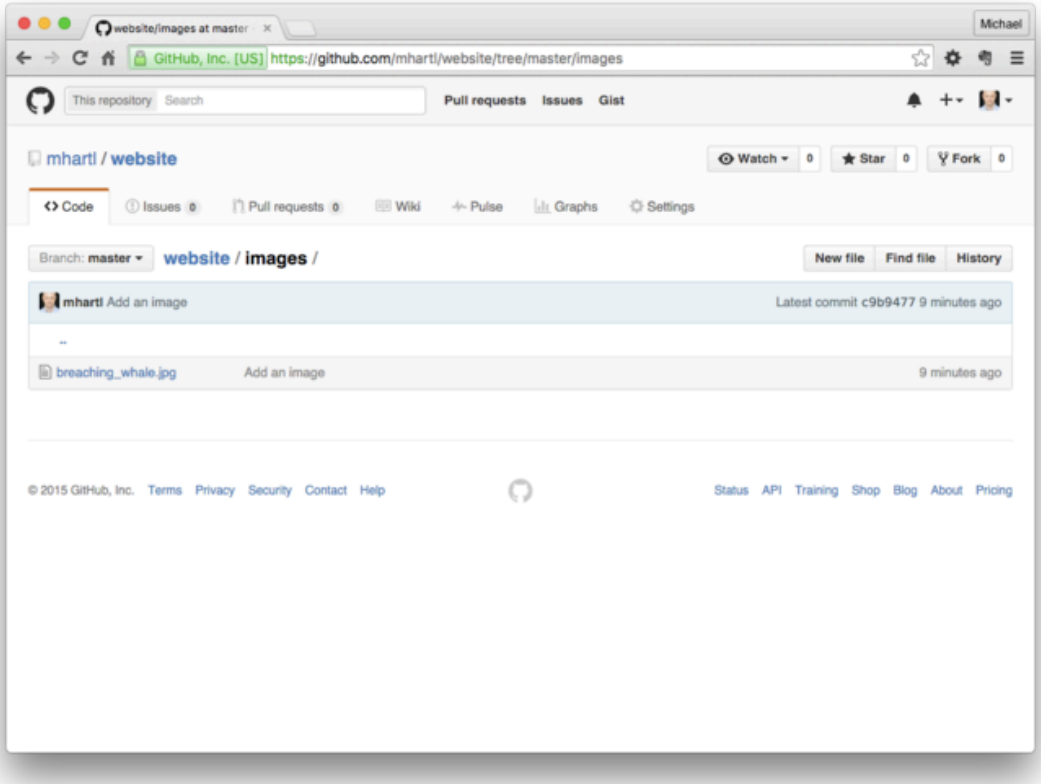

Figure 3.3: The new images directory on GitHub.

2. At this point, the number of commits is large enough that the output of `git log -p` is probably too big to fit in your terminal window. Confirm that running `git log -p` drops you into a `less` interface for easier navigation.

3. Use your knowledge of `less` commands to search for the commit that added the HTML `DOCTYPE`. What is the SHA of the commit?

3.2 Ignoring files

A frequent issue when dealing with Git repositories is coming across files you *don't* want to commit. These include files containing secret credentials, configuration files that aren't shared across computers, temporary files, log files, etc.

For example, on macOS a common side-effect of using the Finder to open directories is the creation of a hidden file called `.DS_Store`.[3] In case you haven't run into it yourself, we can simulate such a side-effect by using `touch` to create a sample `.DS_Store` file as follows:

```
[website (master)]$ touch .DS_Store
```

This file now shows up in the status:

```
[website (master)]$ git status
On branch master
Your branch is up-to-date with 'origin/master'.
Untracked files:
  (use "git add <file>..." to include in what will be committed)

  .DS_Store

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

[3] This happened to me when I ran `open images/` while writing Section 3.1, which is what reminded me I should cover it here.

This is annoying, as we have no need to track this file, and indeed when collaborating with other users it could easily cause conflicts (Section 4.2) down the line.

In order to avoid this annoyance, Git lets us *ignore* such files using a special hidden configuration file called `.gitignore`. To ignore `.DS_Store`, create a file called `.gitignore` using your favorite text editor and then fill it with the contents shown in Listing 3.2.

Listing 3.2: Configuring Git to ignore a file.
~/repos/website/.gitignore

```
.DS_Store
```

After saving the contents of Listing 3.2, the status now picks up the newly added `.gitignore` file, but it *doesn't* list the `.DS_Store` file, thereby confirming that it's being ignored:

```
[website (master)]$ git status
On branch master
Your branch is up-to-date with 'origin/master'.
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)
```

This is an excellent start, but it would be inconvenient if we had to add the name of every file we want to ignore. For instance, the Vim text editor (covered briefly in *Learn Enough Command Line to Be Dangerous*) sometimes creates *temporary files* whose names involve appending a tilde `~` to the end of the normal filename, so you might be editing a file called `foo` and end up with a file called `foo~` in your directory. In such a case, we would want to ignore *all* files ending in a tilde. To support this case, the `.gitignore` file also lets us use *wildcards*, where the asterisk `*` represents "anything":[4]

[4] Wildcards are discussed in *Learn Enough Command Line to Be Dangerous*, as in the command `ls *.txt`.

```
*~
```

Adding the line above to `.gitignore` would cause all temporary Vim files to be ignored by Git. We can also add directories to `.gitignore`, so that, e.g.,

```
tmp/
```

would arrange to ignore all files in the `tmp/` directory.

Git ignore files can get quite complicated, but in practice you can build them up over time by running `git status` and looking for any files or directories you don't want to track, and then adding a corresponding pattern to the `.gitignore` file. In addition, many systems (such as the Ruby on Rails web framework and the Softcover publishing platform) generate a good starting `.gitignore` file for you.[5] See Chapter 1 of the *Ruby on Rails Tutorial* for more information.

3.2.1 Exercises

1. Commit the `.gitignore` file to your repository. *Hint*: Running `git commit -am` isn't enough. Why not?

2. Push your commit up to GitHub and confirm using the web interface that the push succeeded.

3.3 Branching and merging

One of the most powerful features of Git is its ability to make *branches*, which are effectively complete self-contained copies of the project source, together with the ability to *merge* one branch into another, thereby incorporating the changes into the original branch. The best thing about a branch is that you can

[5] This common practice is further evidence of the ubiquity of Git—at this point, many projects simply assume you're using it.

make your changes to the project in isolation from the master copy of the code, and then merge your changes in only when they're done. This is especially helpful when collaborating with other users (Chapter 4); having a separate branch lets you make changes independently from other developers, reducing the risk of accidental conflicts.

We'll use the addition of a second HTML page, an "About page", as an example of how to use Git branches. Our first step is to use **`git checkout`** with the **`-b`** option, which makes a new branch called **`about-page`** and checks it out at the same time, as shown in Listing 3.3.

Listing 3.3: Checking out and creating the **`about-page`** branch.

```
[website (master)]$ git checkout -b about-page
[website (about-page)]$
```

The prompt in Listing 3.3 includes the new branch name for convenience, which is a result of the optional advanced setup in Section 4.6.2, so your prompt may differ.

Now that we've checked out the new **`about-page`** branch, we can visualize our repository as shown in Figure 3.4. The main repository evolution is a series of commits, and the branch effectively represents a copy of the repo at the time the branch was made.[6] Our plan is to make a series of changes on the **`about-page`** branch, and then incorporate the changes back into the **`master`** branch using **`git merge`**.

We can view the current branches using the **`git branch`** command:

```
[website (about-page)]$ git branch
* about-page
master
```

This lists all the branches currently defined on the local machine, with an asterisk **`*`** indicating the currently checked-out branch. (We'll learn how to list *remote* branches in Section 4.3.)

[6] Of course, it would be potentially inefficient to copy all the files over to the new branch, since there's usually a lot overlap with the old one. To avoid any unnecessary duplication, Git tracks diffs rather than actually making full copies of all files.

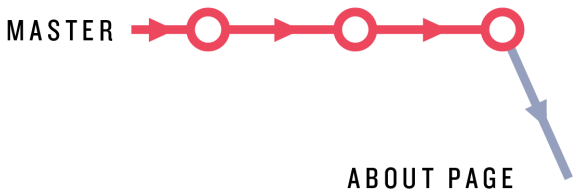

Figure 3.4: Branching off the **master** branch.

Having checked out the branch for the About page, we're now ready to start making some changes to our working directory. We'll start by making a new file called **about.html** to include some information about our project. Because we want the new page to have the full HTML structure (as in Listing 1.6), we'll copy over the **index.html** file and then edit it as necessary:

```
[website (about-page)]$ cp index.html about.html
```

If this duplication seems a little unclean, it is. For example, what if there were an error in the HTML structure of **index.html**? Having copied it over to **about.html**, we'd have to make the correction in both places. As we'll see in Section 4.3, in fact there *is* an error, and we *will* have to make the correction twice. This sort of situation is annoying, and it's far better to use a *site template* that avoids unnecessary duplication. We'll start learning how to do that in *Learn Enough CSS & Layout to Be Dangerous*.

Throughout the rest of the tutorial, we'll be editing both **index.html** and **about.html**, so this is a good opportunity to use the preferred technique for opening a full project in a text editor (as covered in *Learn Enough Text Editor to Be Dangerous*). I suggest closing all current editor windows and re-opening the project as follows:

```
[website (about-page)]$ atom .
```

By doing this, we can use "fuzzy opening" to open the files of our choice. In particular, in Atom we can use ⌘P to open **about.html** and start making the necessary changes.

After opening **about.html**, fill it with the contents shown in Listing 3.4. As always, I recommend typing in everything by hand, which will make it easier to see the diffs relative to Listing 3.1. (The only possible exception is the trademark character ™, added to highlight character encoding issues, which you might have to copy and paste. On a Mac, you can get ™ using Option-2.)

Listing 3.4: The initial HTML for the About page.
~/repos/website/about.html

```
<!DOCTYPE html>
<html>
  <head>
    <title>About Us</title>
  </head>
  <body>
    <h1>About</h1>
    <p>
      This site is a sample project for the <strong>awesome</strong> Git
      tutorial <em>Learn Enough™ Git to Be Dangerous</em>.
    </p>
  </body>
</html>
```

Listing 3.4 introduces two new tags: **strong** (which most browsers render as **boldface** text) and **em** for emphasis (which most browsers render as *italicized* text).

We're now ready to commit the initial version of the About page. Because **about.html** is a new file, we have to add it and then commit, and I sometimes like to combine these two steps using **&&** as described in *Learn Enough Command Line to Be Dangerous*:

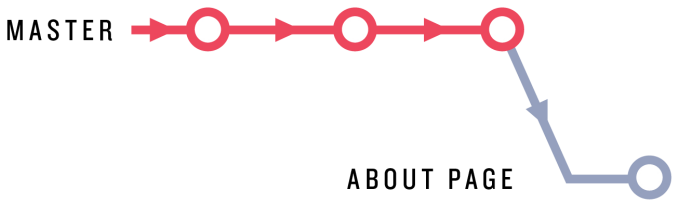

Figure 3.5: The **about-page** branch with a diff from **master**.

```
[website (about-page)]$ git add -A && git commit -m "Add About page"
```

At this point, the **about-page** branch has diverged from **master**, as shown in Figure 3.5.

Before merging **about-page** back in to the **master** branch, we'll make one more change. In the editor, use ⌘P or the equivalent to open **index.html** and add a *link* to the About page, as shown in Listing 3.5.

Listing 3.5: Adding a link to the About page.
~/repos/website/index.html

```
<!DOCTYPE html>
<html>
  <head>
    <title>A whale of a greeting</title>
  </head>
  <body>
    <h1>hello, world</h1>
    <a href="about.html">About this project</a>
    <p>Call me Ishmael.</p>
    <img src="images/breaching_whale.jpg">
  </body>
</html>
```

Listing 3.5 uses the important (if confusingly named) *anchor tag* **a**, which is the HTML tag for making links. This tag contains both content ("About

this project") and a *hypertext reference*, or **href**, which in this case is the **about.html** file we just created. (Because **about.html** is on the same site as **index.html**, we can link to it directly, but when linking to external sites the href should be a fully qualified URL,[7] such as http://example.com/.)[8]

After saving the change and refreshing **index.html** in our browser, the result should appear as shown in Figure 3.6. Following the link should lead us to the About page, as seen in Figure 3.7. Note that the trademark character ™ doesn't display properly in Figure 3.7; this behavior is browser-dependent—as of this writing, the ™ symbol displays properly in Firefox and Chrome but not in Safari. We'll add code to ensure consistent behavior across all browsers in Section 4.3.

Having finished with the changes to **index.html**, we can make a commit as usual with **git commit -am**:

```
[website (about-page)]$ git commit -am "Add a link to the About page"
```

With this commit, the **about-page** branch now appears as in Figure 3.8.

We're done making changes for now, so we're ready to merge the About page topic branch back into the **master** branch. We can get a handle on which changes we'll be merging in by using **git diff**; we saw in Section 1.4 that this command can be used by itself to see the difference between unstaged changes and our last commit, but the same command can be used to show diffs between branches. This can take the form **git diff branch-1 branch-2**, but if you leave the branch unspecified Git automatically diffs against the current branch. This means we can diff **about-page** vs. **master** as follows:

```
[website (about-page)]$ git diff master
```

The result in my terminal program appears as shown in Figure 3.9. On my system, the diff is too long to fit on one screen, but (as we saw with **git log** in Section 3.1.1) the output of **git diff** uses the **less** program in this case.

[7] Recall that URL is short for Uniform Resource Locator, and in practice usually just means "web address".

[8] Fun fact: As you can verify by visiting it, example.com is a special domain reserved for examples just like this one.

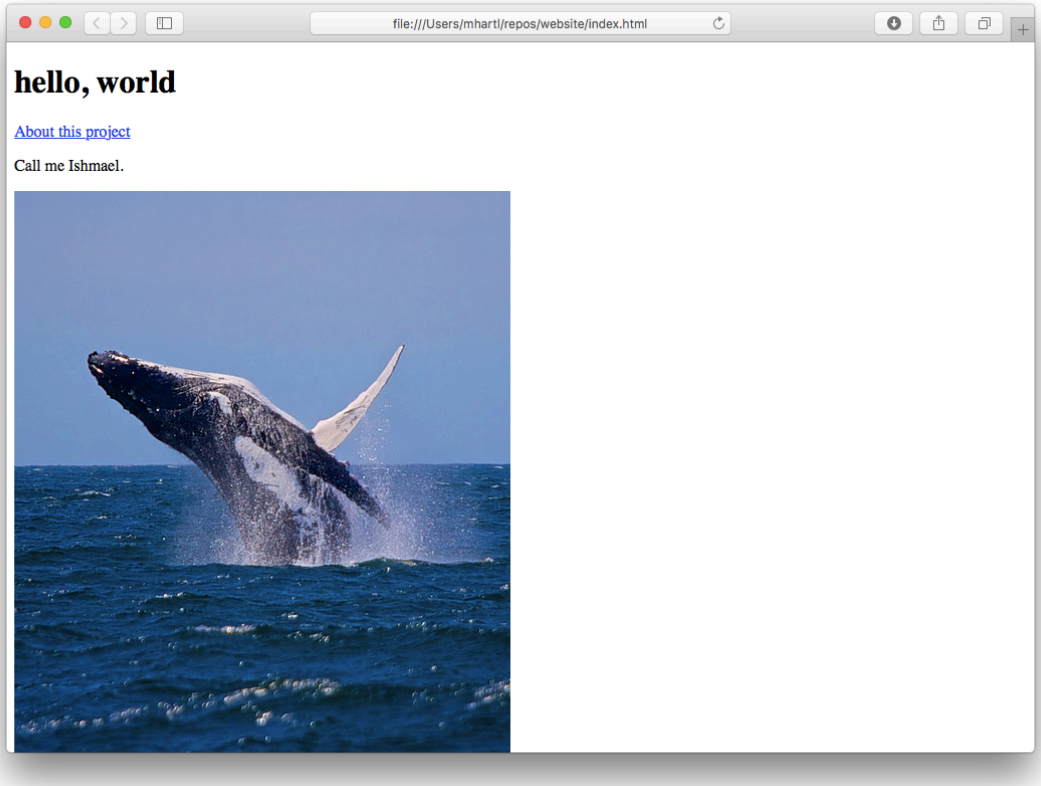

Figure 3.6: The index page with an added link.

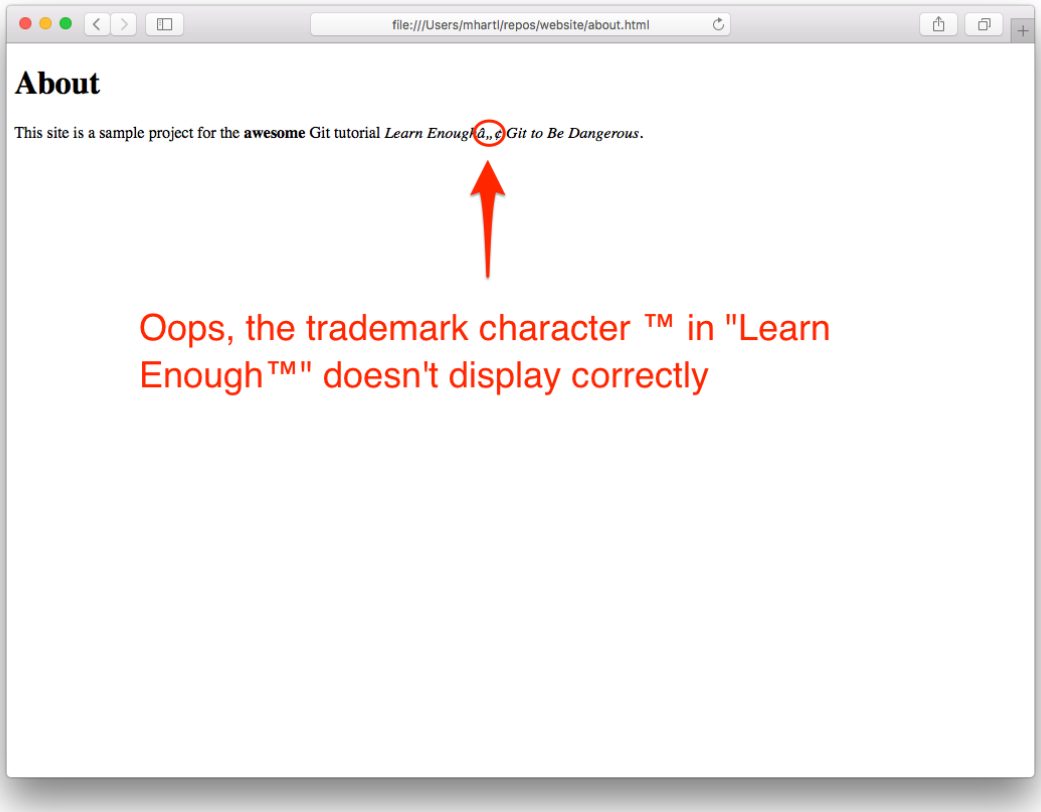

Figure 3.7: A slightly broken About page.

Figure 3.8: The current state of the `about-page` branch relative to `master`.

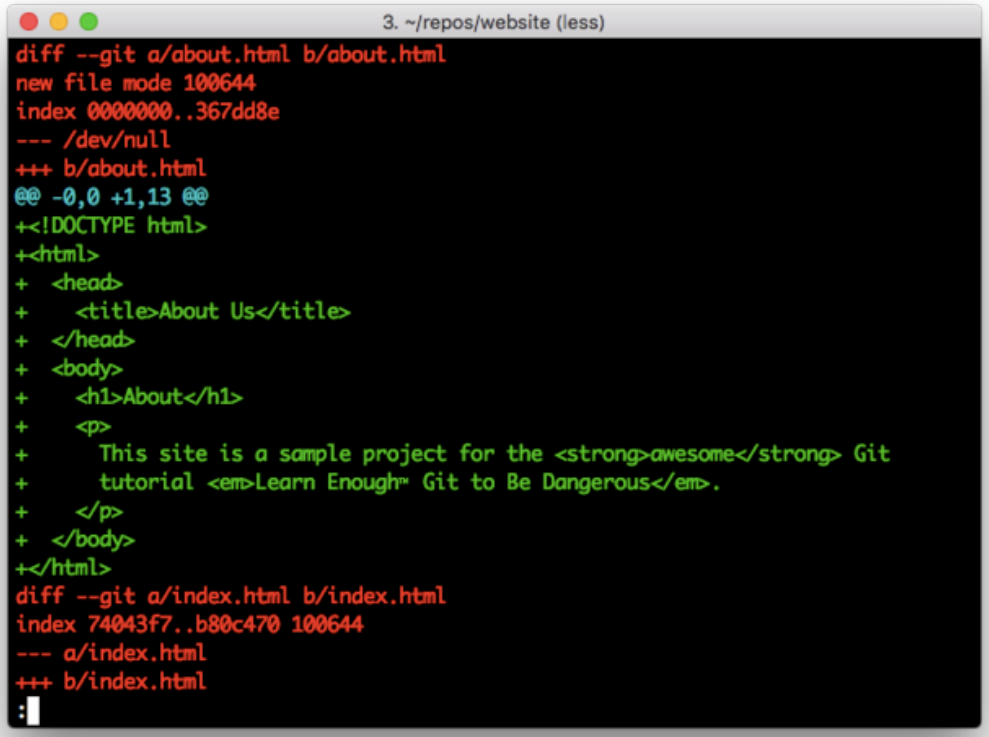

Figure 3.9: Diffing two branches.

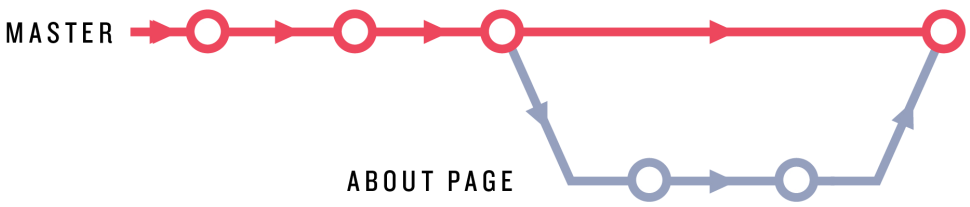

Figure 3.10: The branches after merging **about-page** into **master**.

To incorporate the changes on **about-page** into **master**, the first step is to check out the **master** branch:

```
[website (about-page)]$ git checkout master
[website (master)]$
```

Note that, unlike the **checkout** command in Listing 3.3, here we omit the **-b** option because the **master** branch already exists.

The next step is to merge in the changes on the other branch, which we can do with **git merge**:

```
[website (master)]$ git merge about-page
```

At this point, our branch structure appears as in Figure 3.10.

In the present case, the **master** branch didn't change while we were working on the **about-page** branch, but Git excels even when the original branch has changed in the interim. This situation is especially common when collaborating with others (Chapter 4), but can happen even when working alone. Suppose, for example, that we discovered a typo on **master** and wanted to fix it and push up immediately. In that case the **master** branch would change (Figure 3.11), but we could still merge in the topic branch as usual. There is a possibility that changes on **master** would *conflict* with the merged changes,

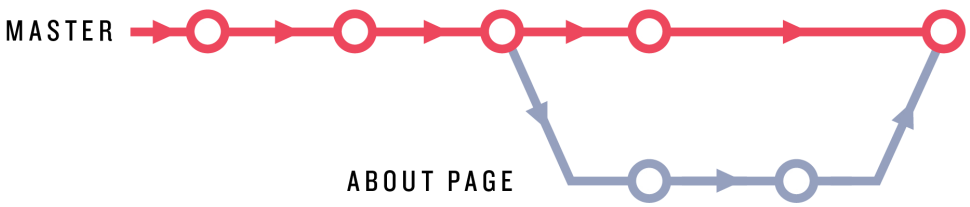

Figure 3.11: The tree structure if we made a change to **master**.

but Git is good at automatically merging content. Even when conflict is unavoidable, Git is good at marking conflicts explicitly so that we can resolve them by hand. We'll see a concrete example of this in Section 4.2.

Having merged in the changes, we can sync up the local **master** branch with the version at GitHub (called **origin/master**) as usual:

```
[website (master)]$ git push
```

Since we probably don't need the **about-page** branch any longer, we can optionally delete it, which is left as an exercise (Section 3.3.2).

3.3.1 Rebasing

The most common way to combine branches is **git merge**, but there's a second method called **git rebase** that you're likely to encounter at some point. My advice for now is: *ignore **git rebase***. The differences between merging and rebasing are subtle, and conventions for using **rebase** differ, so I recommend using **git rebase** only when an advanced Git user tells you to; otherwise, use **git merge** to combine the contents of two branches.

3.3.2 Exercises

1. Use the command `git branch -d about-page` to delete the topic branch. Confirm by running `git branch` that only the `master` branch is left.

2. In Listing 3.3, we used `git checkout -b` to create a branch and check it out at the same time, but it's also possible to break this into two steps. As a first step, use `git branch` to make a branch with the name `test-branch`. (This involves passing an argument to `git branch`, as in `git branch <branch name>`.) Then confirm that the new branch exists but isn't currently checked out by running `git branch` without an argument.

3. Check out `test-branch` and use `touch` to add a file with a name of your choice, then add and commit it to the repository.

4. Check out the `master` branch and try deleting the test branch using `git branch -d` to confirm that it doesn't work. The reason is that, in contrast to the `about-page` branch, the test branch hasn't been merged into `master`, and by design `-d` doesn't work in this case. Because we don't actually want its changes, delete the test by using the related `-D` option, which deletes the branch in question even if its changes are unmerged.

3.4 Recovering from errors

One of the most useful features of Git is its ability to let us recover from errors that would otherwise be catastrophic. The error-recovery techniques themselves can be dangerous, though, so they should always be implemented with care.

Let's consider a common scenario where we make an unintentional change to a project and want to get back to the state of the repository as of the most recent commit (a state known as `HEAD`). For example, it's a good practice to include a newline at the end of a file so that, e.g., running `tail`[9] gives

[9] The `tail` command is covered in *Learn Enough Command Line to Be Dangerous*.

```
[website (master)]$ tail about.html
  .
  .
  .
  </body>
</html>
[website (master)]$
```

instead of

```
[website (master)]$ tail about.html
  .
  .
  .
  </body>
</html>[website (master)]$
```

Of course, we could add such a newline using a text editor, but a common Unix idiom to accomplish the same thing is to use **`echo`** with the append operator **`>>`**:

```
[website (master)]$ echo >> about.html    # Appends a newline to about.html
```

Unfortunately, in this context it's easy to accidentally leave off one of the angle brackets and inadvertently use the *redirect* operator **`>`** instead:[10]

```
[website (master)]$ echo > about.html
```

Go ahead and try the command above; you will discover that the result is to overwrite **`about.html`** with a newline, thereby effectively wiping out its contents, as we can verify with **`cat`**:

[10] Redirecting and appending are covered in *Learn Enough Command Line to Be Dangerous*.

```
[website (master)]$ cat about.html

[website (master)]$
```

In a regular Unix directory, there would be no hope of recovering the contents of **about.html**, but in a Git repository we can undo the changes by forcing the system to check out the most recently committed version. We start by confirming that **about.html** has changed by running **git status**:

```
[website (master)]$ 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:   about.html

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

This doesn't indicate the scope of the damage, though, which we can inspect using **git diff**:

```
diff --git a/about.html b/about.html
index 6278cf6..8b13789 100644
--- a/about.html
+++ b/about.html
@@ -1,13 +1 @@
-<!DOCTYPE html>
-<html>
-  <head>
-    <title>About Us</title>
-  </head>
-  <body>
-    <h1>About</h1>
-    <p>
-      This site is a sample project for the <strong>awesome</strong> Git
-      tutorial <em>Learn Enough™ Git to Be Dangerous</em>.
-    </p>
-  </body>
-</html>
```

Those minus signs indicate that all of the lines of content are now gone. Happily, we can undo these changes by passing the `-f` (force) option to `checkout`, which forces Git to check out `HEAD`:[11]

```
[website (master)]$ git checkout -f
```

We can then confirm that the About page has been restored:

```
[website (master)]$ git status
On branch master
Your branch is up-to-date with 'origin/master'.
nothing to commit, working directory clean
```

The status "working directory clean" indicates that there are no changes, and you can verify by running `cat about.html` that its contents have been restored. Phew! That was a close one. (It's worth noting that `git checkout -f` itself is potentially dangerous, as it wipes out *all* the changes you've made, so use this trick only when you're 100% sure you want to revert to `HEAD`.)

Another source of robustness against error is using branches, as described in Section 3.3. Because changes made on one branch are isolated from other branches, you can always just delete the branch if things go horribly wrong. For example, suppose we made the same `echo` mistake on a `test-branch`:

```
[website (master)]$ git checkout -b test-branch
[website (test-branch)]$ echo > about.html
```

We can fix this by committing the changes and then deleting the branch:

```
[website (test-branch)]$ git commit -am "Oops"
[website (test-branch)]$ git checkout master
[website (master)]$ git branch -D test-branch
```

[11] The command `git reset --hard HEAD` is equivalent, but I find the version with `checkout` to be easier to remember.

Note here that we need to use `-D` instead of `-d` to delete the branch because `test-branch` is unmerged (Section 3.3.2).

A final example of recovering from error involves the common case of a bug or other defect that makes its way into a project, origins unknown. In such a case, it's convenient to be able to check out an earlier version of the repository.[12] The way to do this is to use the SHAs from the Git log (Section 1.3). For example, to restore the website project to the state right after the second commit, we would run `git log` and navigate to the beginning of the log. Because `git log` uses the `less` interface, we can do this by typing `G` to go to the last line of the log.[13] The result on my system is shown in Listing 3.6. (Because SHAs are by design unique identifiers, your values will differ.)

Listing 3.6: Viewing the SHAs in the Git log.

```
[website (master)]$ git log
commit 8c19674468a67720b9ba61a783e81f97062874bf
Author: Michael Hartl <michael@michaelhartl.com>
Date:   Mon Dec 21 21:27:56 2015 -0800

    Add a README

commit 69b955490caf12552e83d476820d29475fa35010
Author: Michael Hartl <michael@michaelhartl.com>
Date:   Mon Dec 21 21:02:20 2015 -0800

    Add some HTML structure

commit 03aff34ec4f9690228e057a4252bcca169a868b4
Author: Michael Hartl <michael@michaelhartl.com>
Date:   Thu Dec 17 20:03:33 2015 -0800

    Add content to index.html

commit 879392a6bd8dd505f21876869de99d73f40299cc
Author: Michael Hartl <michael@michaelhartl.com>
Date:   Thu Dec 17 20:00:34 2015 -0800

    Initialize repository
```

To check out the commit with the message "Add content to index.html",

[12] The most powerful way to track down such errors is `git bisect`. This advanced technique is covered in the Git documentation.

[13] This `less` navigation trick is described in *Learn Enough Command Line to Be Dangerous*.

simply copy the SHA and check it out:

```
[website (master)]$ git checkout 03aff34ec4f9690228e057a4252bcca169a868b4
Note: checking out '03aff34ec4f9690228e057a4252bcca169a868b4'.

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 03aff34... Add content to index.html
[website ((03aff34...))]$
```

Note that the branch name in the last line has changed to reflect the value of the SHA, and Git has issued a warning that we are in a 'detached HEAD' state. I recommend using this technique to inspect the state of the project and figure out any necessary changes, then check out the **`master`** branch to apply them:

```
[website ((03aff34...))]$ git checkout master
[website (master)]$
```

At this point, you could switch to your text editor and make any necessary changes (such as fixing a bug discovered on the earlier commit).

If all this seems a little abstract, don't worry. The main takeaways are (1) it's possible to "go back in history" to view the project at an earlier state and (2) it's tricky to make changes, so if you find yourself doing anything complicated you should ask a more experienced Git user what to do. (In particular, the exact practices in such a case could be team-dependent.)

3.4.1 Exercises

1. The **`git checkout -f`** trick works only with files that are staged for commit or are already part of the repository, but sometimes you want to get rid of new files as well. Using **`touch`**, create a file with a name of

File/Command	Description	Example
`.gitignore`	Tell Git which things to ignore	`$ echo .DS_store >> .gitignore`
`git checkout  `	Check out a branch	`$ git checkout master`
`git checkout -b  `	Check out & create a branch	`$ git checkout -b about-page`
`git branch`	Display local branches	`$ git branch`
`git merge  `	Merge in a branch	`$ git merge about-page`
`git rebase`	Do something possibly weird & confusing	See "Git Commit"
`git branch -d  `	Delete branch (if merged)	`$ git branch -d about-page`
`git branch -D  `	Delete branch (even if unmerged) **(dangerous)**	`$ git branch -D other-branch`
`git checkout -f`	Force checkout, discarding changes **(dangerous)**	`$ git add -A && git checkout -f`

Table 3.1: Important commands from Chapter 3.

your choice, then **`git add`** it. Verify that running **`git checkout -f`** gets rid of it.

2. Like many other Unix programs, **`git`** accepts both "short form" and "long form" options. Repeat the previous exercise with **`git checkout --force`** to confirm that the effects of **`-f`** and **`--force`** are identical. *Extra credit*: Double-check this conclusion by finding the "force" option in the output of **`git help checkout`**.

3.5 Summary

Important files and commands from this chapter are summarized in Table 3.1.

Chapter 4

Collaborating

Now that we've covered some of the tools needed to use Git effectively on solo projects, it's time to learn about what is perhaps Git's greatest strength: making it easier to collaborate with other people. This is especially the case when using repository hosts like GitHub or Bitbucket, but it is also possible to host Git repositories on private servers (sometimes using software like GitLab to get many GitHub-like benefits).

Because this tutorial is designed for individual readers, we won't actually be able to collaborate with others, but this chapter will explain how you can practice "collaborating" with yourself. There are many different collaboration scenarios, and they vary significantly by team and by project, so we'll focus on the important case of multiple collaborators who all have *commit rights* to a particular repo. This model is appropriate for teams where everyone can make changes without explicit approval from a project maintainer.

Open-source projects typically use a different flow involving *forking* and *pull requests*, but the details differ enough that it's best to defer to the collaboration instructions of each particular project. Consider, for example, the instructions for contributing to Ruby on Rails. With the commands from this tutorial and your technical sophistication (Box 1.2), you'll be in a good position to understand and follow such instructions if you decide to get involved in contributing to open-source software or other projects under version control with Git.

For reference, important commands from this chapter are summarized in

Section 4.5.

4.1 Clone, push, pull

As an example of a common collaboration workflow, we'll simulate the case of two developers working on the same project, in this case the simple website developed in this tutorial. We'll start with Alice (Figure 4.1)[1] working in the original **`website`** directory, and we'll create a second directory (**`website-copy`**) for her collaborator Bob (Figure 4.2).[2]

As a first step, Alice runs **`git push`** just to make sure all her changes are on the remote repository:

```
[website (master)]$ git push
```

In real life, Alice would now need to add Bob as a collaborator on the **`website`** repository, which she could do at GitHub by clicking on **Settings > Collaborators** and then put Bob's GitHub username in the Add collaborator box (Figure 4.3). Because we're collaborating with ourselves, we can skip this step.

Once Bob gets the notification that he's been added to the **`website`** repository, he can go to GitHub to get the *clone URL*, as shown in Figure 4.4. This URL lets Bob make a full copy of the repository (including its history) using **`git clone`**.

Ordinarily, Bob would probably use his own **`repos`** directory, with a project called **`website`** as in Alice's original, but because we're only simulating the

[1] *Alice's Adventures in Wonderland* original illustrations by John Tenniel. Image retrieved from https://cellcode.us/quotes/colored-book-alice-wonderland-original-drawings.html on 2019-02-15. Copyright © 1890, now in the public domain.

[2] Image retrieved from https://www.pinterest.com/pin/508766089125966253/ 2019-02-15. Copyright © 1912 by Jessie Wilcox Smith, now in the public domain.

Figure 4.1: Alice, working on **`website`**.

Figure 4.2: Bob (with son Tim), working on **`website-copy`**.

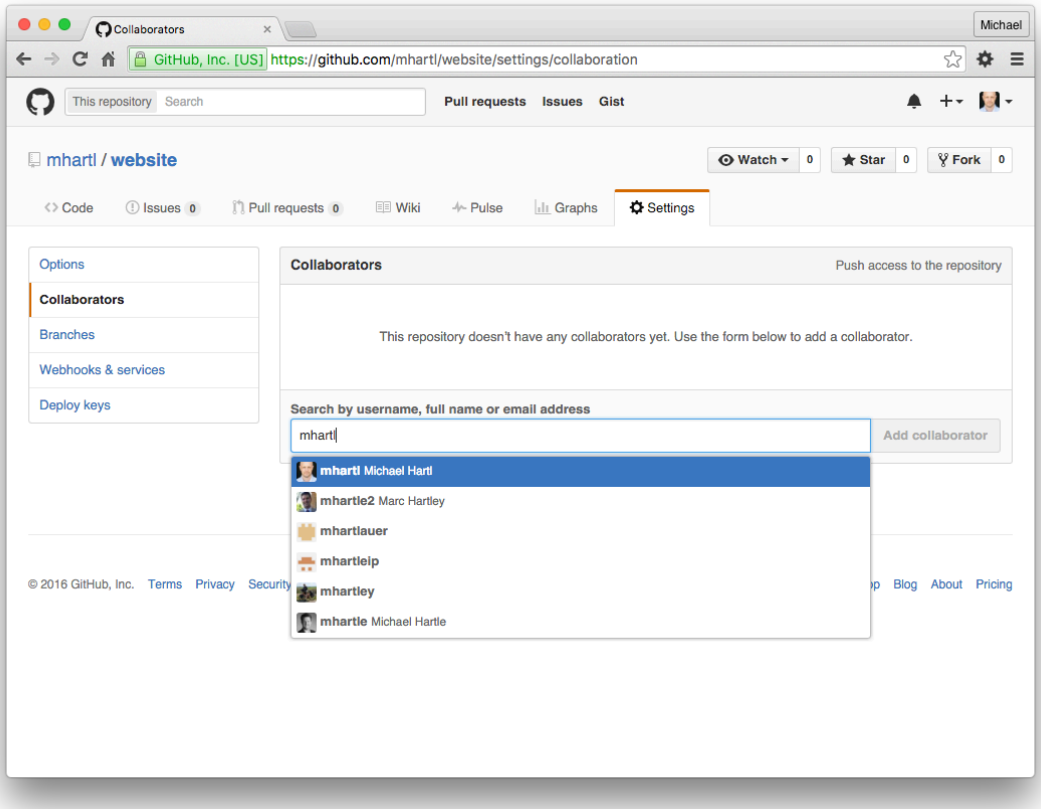

Figure 4.3: The GitHub page to add collaborators.

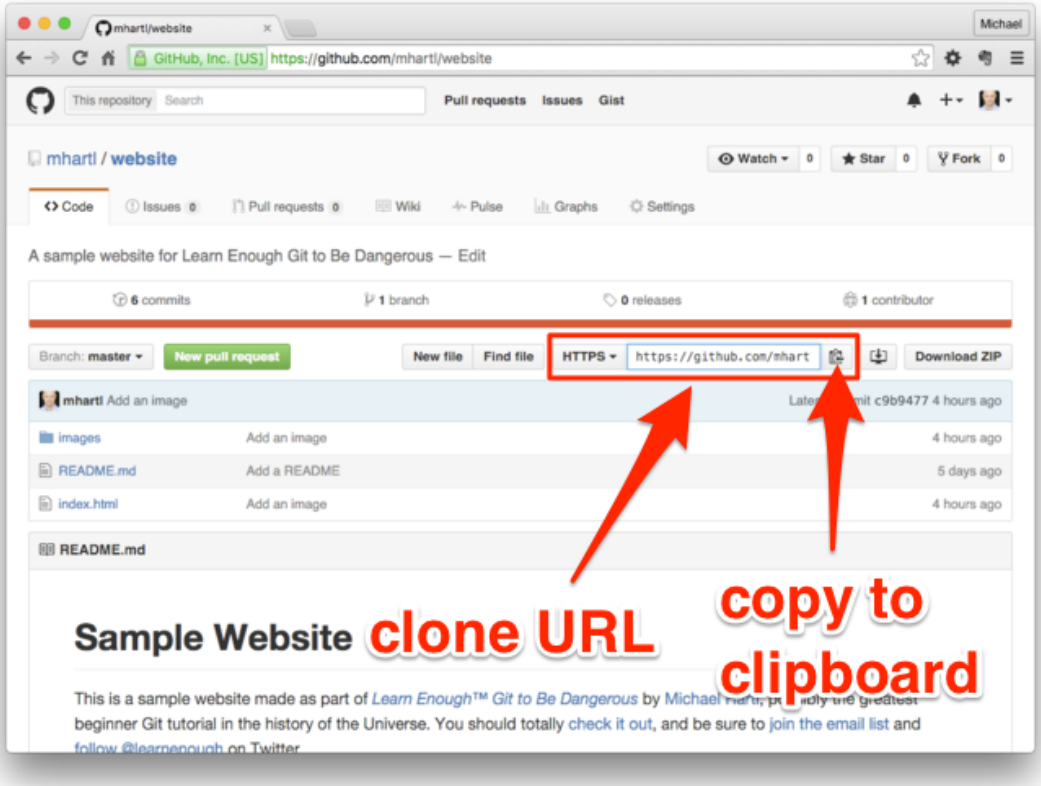

Figure 4.4: Finding the clone URL at GitHub.

collaboration we'll use the name **website-copy** for clarity. In addition, when doing something a little artificial like this, I like to use a temp directory called **~/tmp**,[3] so create this directory if it doesn't already exist on your system:

```
$ cd
$ mkdir tmp
```

Then **cd** to it and clone the repo to the local directory:

```
[~]$ cd tmp/
[tmp]$ git clone <clone URL> website-copy
Cloning into 'website-copy'...
[tmp]$ cd website-copy/
```

Here we've included the argument **website-copy** to **git clone**, thereby showing how to use a different name than the original repo, but usually you just run **git clone <clone URL>**, which uses the default repo name (in this case, **website**).

Now we're ready to open the copy of the project and start making edits:

```
[website-copy (master)]$ atom .
```

For the purposes of this exercise, I recommend placing the editor windows for **website** and **website-copy** side by side, as shown in Figure 4.5.

[3] The idea behind a temp directory is to have a place to put temporary files that won't necessarily persist for long. Many operating systems have a system-wide temp directory (often called **/tmp**), but I also like to have one under my home directory for personal use.

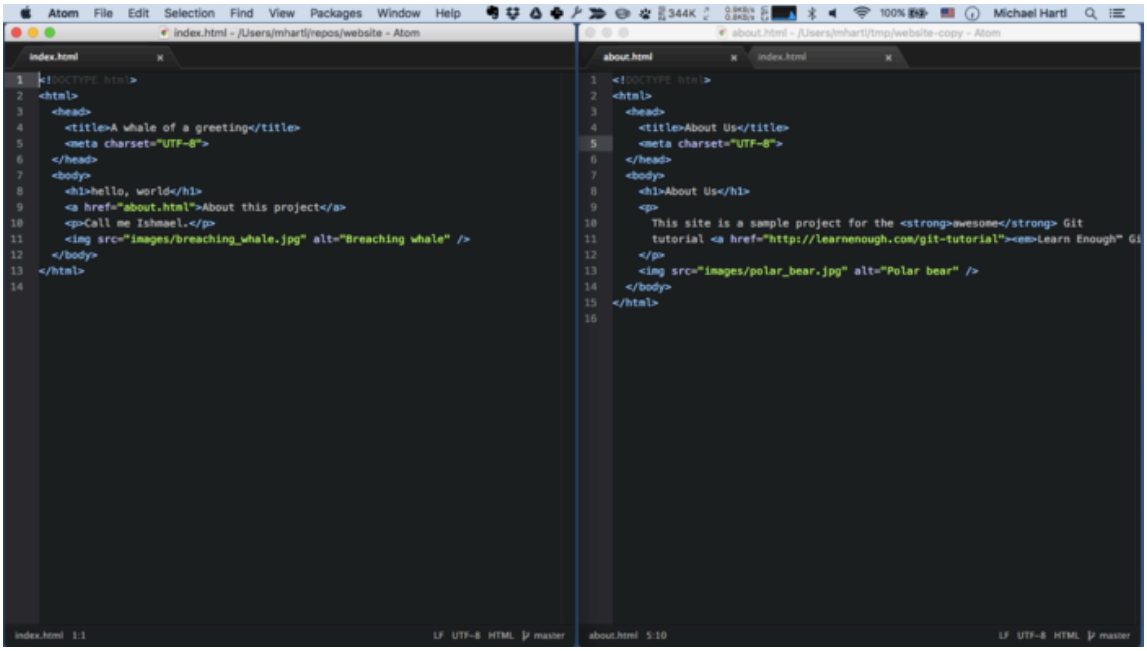

Figure 4.5: The **website** and **website-copy** editors running side by side.

To begin the collaboration, we'll have Bob make a change to the site by wrapping the tutorial title on the About page in a link, like this:

```
<a href="https://www.learnenough.com/git-tutorial">…</a>
```

Here the ellipsis … represents the full title of the tutorial, *Learn Enough Git to Be Dangerous*. The resulting line is too long to display here, but we can wrap it, as shown in Figure 4.6, with the result as shown in Figure 4.7.

If we look at the diff using **git diff**, we see the wrapped line (Figure 4.8), which appears in a browser as shown in Figure 4.9.

Having added the link, Bob can commit his changes and push up to the remote repository:

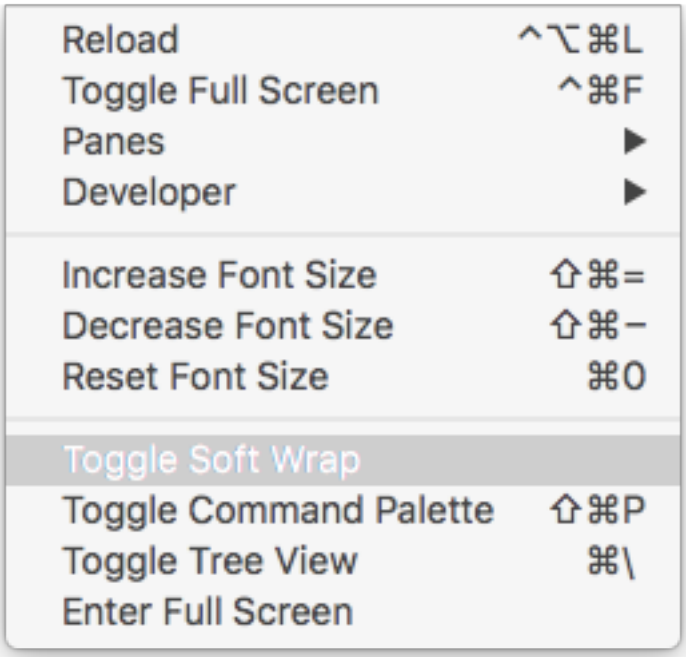

Figure 4.6: Toggling soft wrap in Atom.

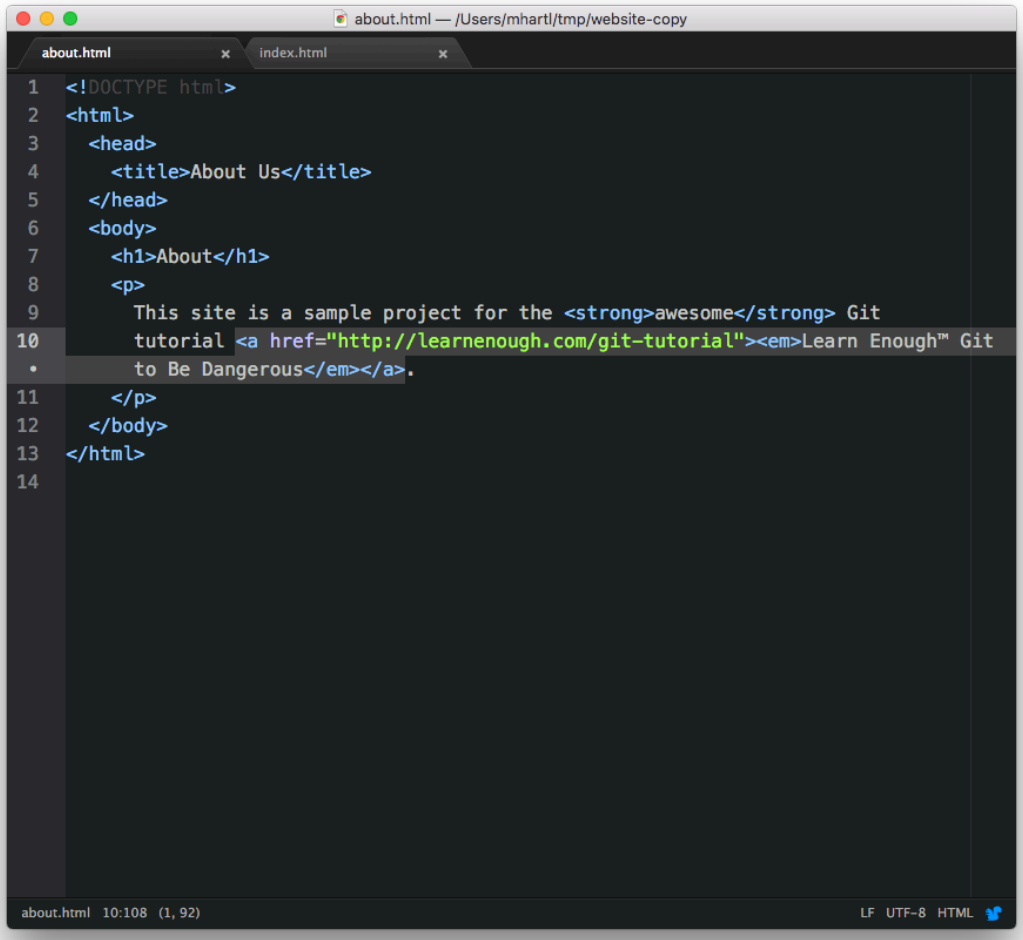

Figure 4.7: The About page with soft wrap activated.

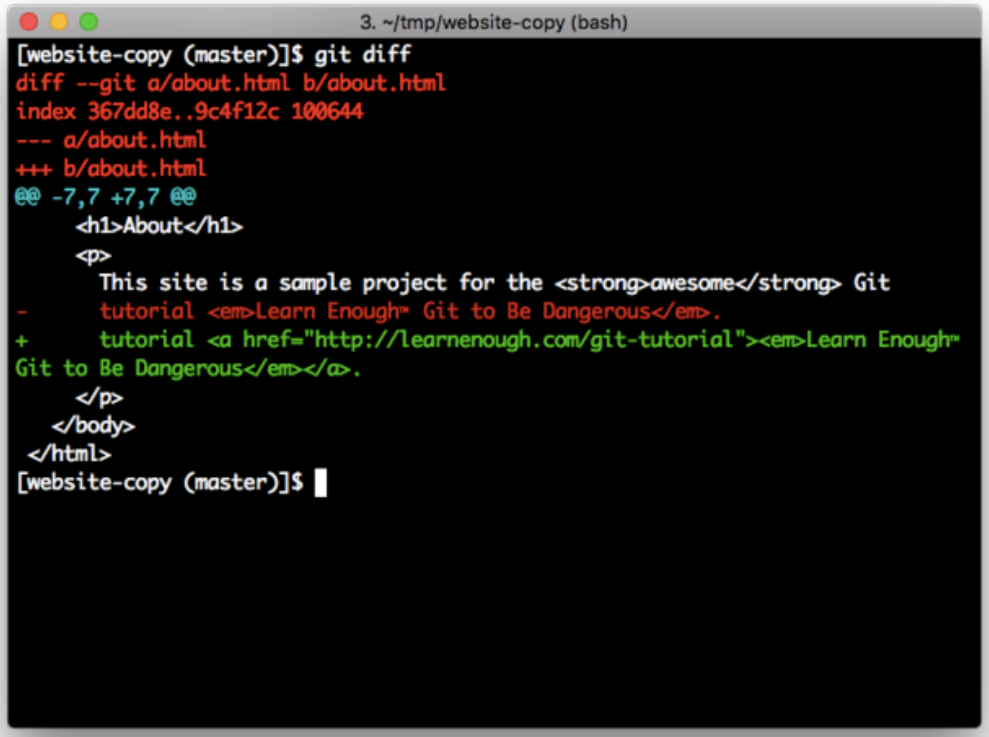

Figure 4.8: The diff with a wrapped line.

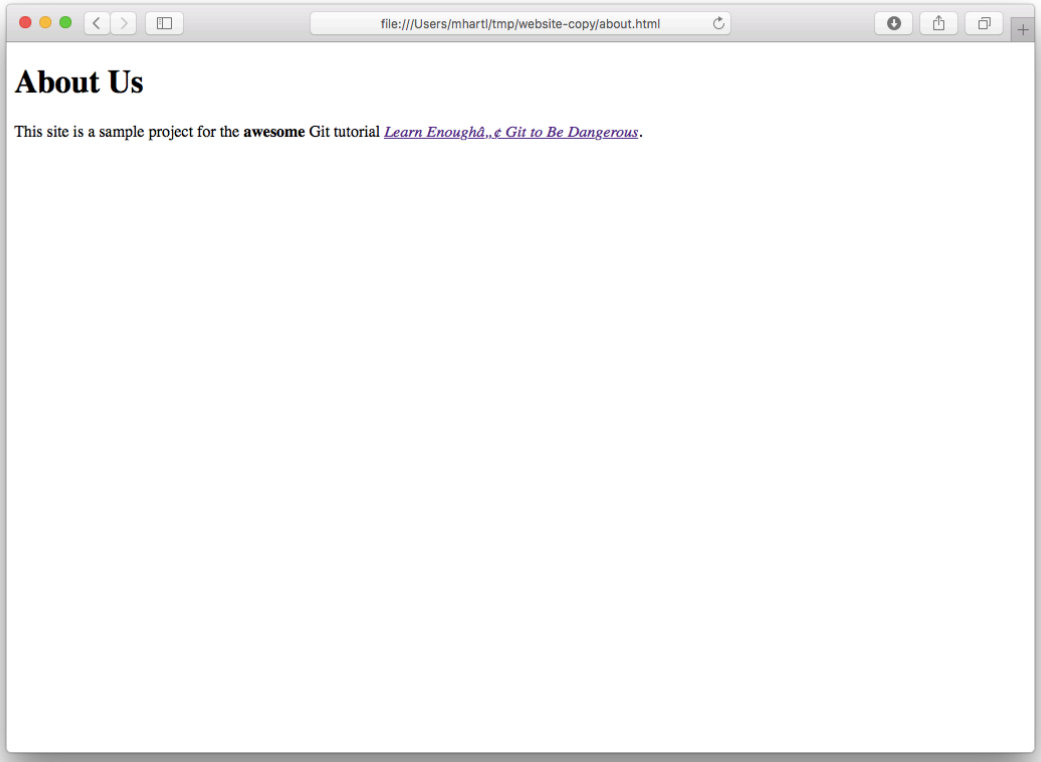

Figure 4.9: Linking the Git tutorial title on the About page.

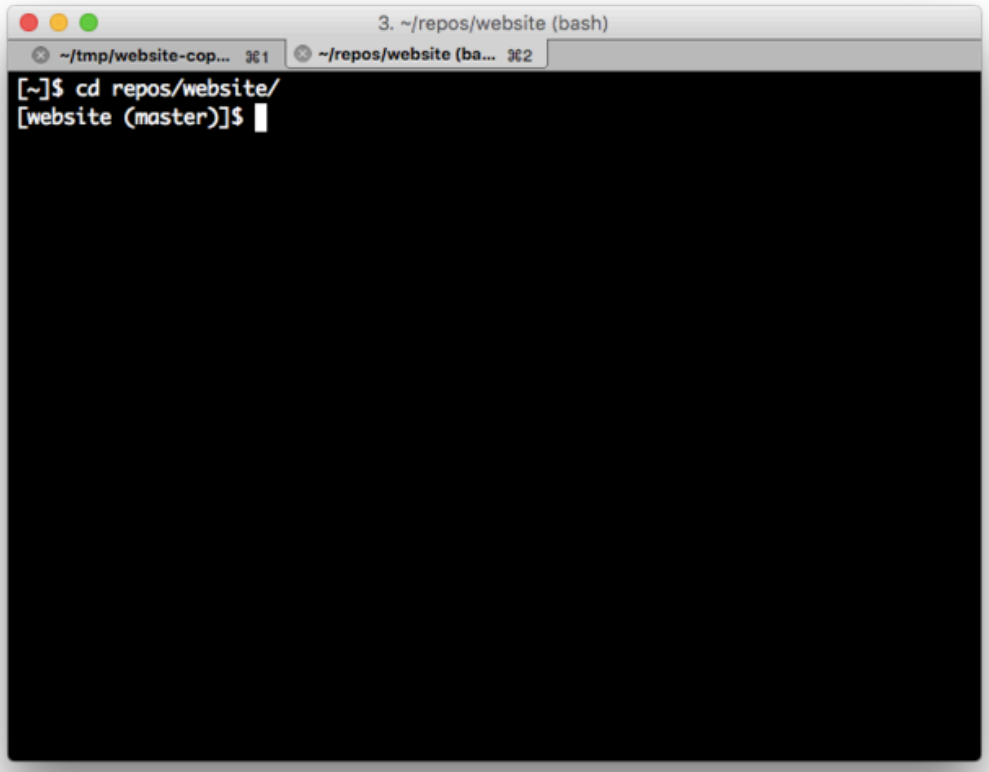

Figure 4.10: Using a new terminal tab for the original directory.

```
[website-copy (master)]$ git commit -am "Add link to tutorial title"
[website-copy (master)]$ git push
```

At this point, Bob might send Alice a notification that there's a change ready, or Alice might just be diligent about checking for changes. In either case, Alice can get the changes from the remote origin by running **`git pull`**. I suggest opening up a new tab in your terminal window for Alice's directory (as shown in Figure 4.10) and then pull as follows:

```
[website (master)]$ git pull
remote: Counting objects: 3, done.
remote: Compressing objects: 100% (1/1), done.
remote: Total 3 (delta 2), reused 3 (delta 2), pack-reused 0
Unpacking objects: 100% (3/3), done.
From https://github.com/mhartl/website
   42db83e..986a487  master     -> origin/master
Updating 42db83e..986a487
Fast-forward
 about.html | 2 +-
 1 file changed, 1 insertion(+), 1 deletion(-)
```

With that, Alice's project should have Bob's commit, and her copy of the About page should be identical to Figure 4.9. (Checking that Bob's commit is present in the log is left as an exercise.)

4.1.1 Exercises

1. As Alice, run `git log` to verify that the commit was pulled down correctly. Double-check the details using `git log -p`.

2. The whale picture added in Listing 3.1 (Figure 3.1) requires attribution under the Creative Commons Attribution-NoDerivs 2.0 Generic license. As Alice, link the image to the original attribution page, as shown in Listing 4.1. Commit the result and push to GitHub.

3. As Bob, pull in the changes from the previous exercise. Verify by refreshing the browser and by running `git log -p` that Bob's repo has been properly updated.

Listing 4.1: Linking to the whale image's attribution page.
~/repos/website/index.html

```
.
.
.
<a href="https://www.flickr.com/photos/28883788@N04/10097824543">
  <img src="images/breaching_whale.jpg">
</a>
.
.
.
```

4.2 Pulling and merge conflicts

In Section 4.1, Alice didn't make any changes while Bob was making his commit, so there was no chance of conflict, but this is not always the case. In particular, when two collaborators edit the same file, it is possible that the changes might be irreconcilable. Git is pretty smart about merging in changes, and in general conflicts are surprisingly rare, but it's important to be able to handle them when they occur. In this section, we'll consider both non-conflicting and conflicting changes in turn.

4.2.1 Non-conflicting changes

We'll start by having Alice and Bob make *non*-conflicting changes in the same file. Suppose Alice decides to change the top-level heading on the About page from "About" to "About Us", as shown in Listing 4.2.

Listing 4.2: Alice's change to the About page's `h1`.
~/repos/website/about.html

```
<!DOCTYPE html>
<html>
    .
    .
    .
    <h1>About Us</h1>
    .
    .
    .
  </body>
</html>
```

After making this change, Alice commits and pushes as usual:

```
[website (master)]$ git commit -am "Change page heading"
[website (master)]$ git push
```

Meanwhile, Bob decides to add a new image (Figure 4.11)[4] to the About page. He first downloads it with **curl** as follows:

```
[website-copy (master)]$ curl -o images/polar_bear.jpg \
>                             -OL https://cdn.learnenough.com/polar_bear.jpg
```

(Note here that you should type the backslash character **** in the first line, but you *shouldn't* type the literal angle bracket **>** in the second line. The **** is used for a *line continuation*, and after hitting return the **>** will be added automatically

[4]Image retrieved from https://www.flickr.com/photos/puliarfanita/22959238329 on 2015-12-28. Copyright © 2015 by Anita Ritenour and used unaltered under the terms of the Creative Commons Attribution 2.0 Generic license.

Figure 4.11: An image for Bob to add to the About page.

by your shell program.) He then adds it to **about.html** using the **img** tag, as shown in Listing 4.3, with the result shown in Figure 4.12.

Listing 4.3: Adding an image to the About page.
~/tmp/website-copy/about.html

```
<!DOCTYPE html>
<html>
    .
    .
    .
    <img src="images/polar_bear.jpg" alt="Polar bear">
  </body>
</html>
```

Note that Bob has included an **alt** attribute in Listing 4.3, which is a text al-

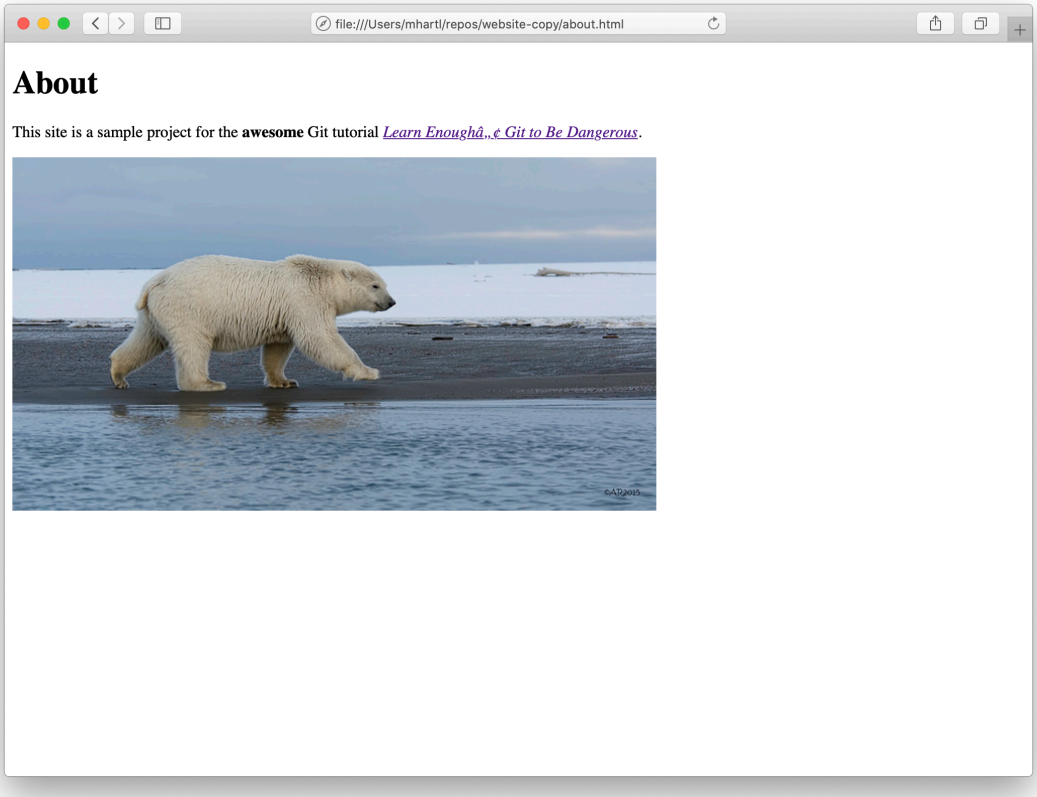

Figure 4.12: The About page with an added image.

ternative to the image. The `alt` attribute is actually required by the HTML5 standard, and including it is a good practice because it's used by web spiders and by screen readers for the visually impaired.

Having made his change, Bob commits as usual:

```
[website-copy (master)]$ git add -A
[website-copy (master)]$ git commit -m "Add an image"
```

When he tries to push, though, something unexpected happens, as shown in Listing 4.4.

Listing 4.4: Bob's push, rejected.

```
[website-copy (master)]$ git push
To https://github.com/mhartl/website.git
 ! [rejected]        master -> master (fetch first)
error: failed to push some refs to 'https://github.com/mhartl/website.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.
```

Because of the changes Alice already pushed, Git won't let Bob's push go through: as indicated by the first highlighted line in Listing 4.4, the push was rejected by GitHub. As indicated by the second highlighted line, the solution to this is for Bob to **`pull`**:

```
[website-copy (master)]$ git pull
```

Even though Alice made changes to **`about.html`**, there is no conflict because Git figures out how to combine the diffs. In particular, **`git pull`** brings in the changes from the remote repo and uses **`merge`** to combine them automatically,

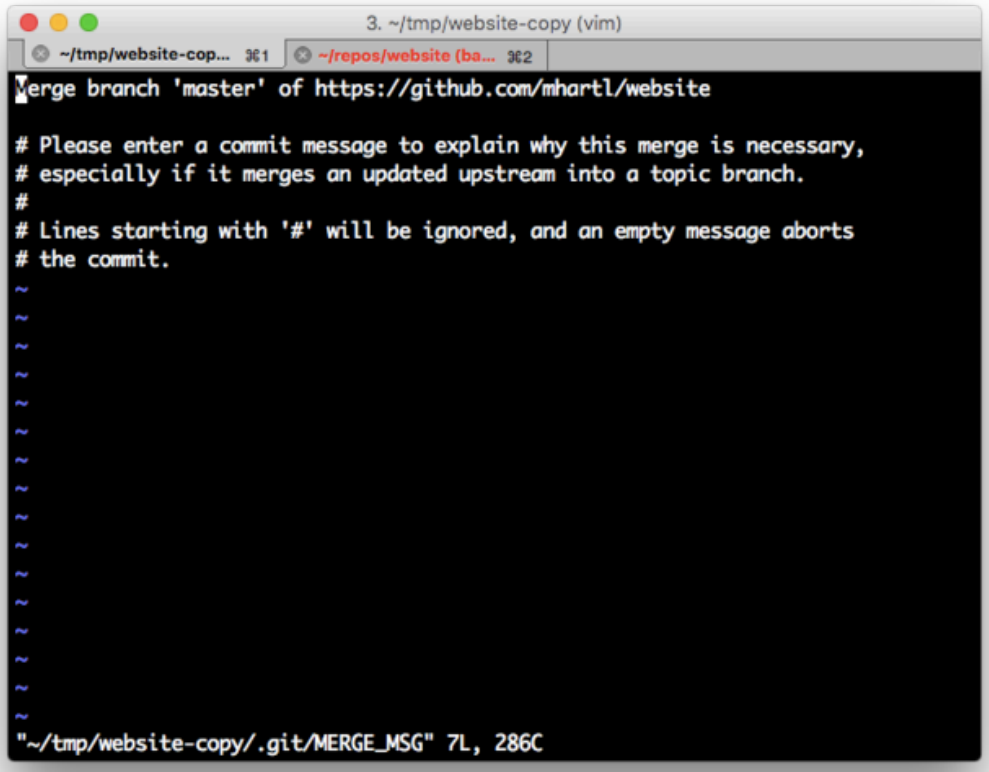

Figure 4.13: The default editor for merging from a **`git pull`**.

adding the option to add a commit message by dropping Bob into the default editor, which on most systems is Vim (Figure 4.13). (This is just one of many reasons why *Learn Enough Text Editor to Be Dangerous* covers Minimum Viable Vim.) To get the merge to go through, you can simply quit out of Vim using **`:q`**.

We can confirm that this worked by checking the log, which shows both the merge commit and Alice's commit from the original copy (Listing 4.5).

Listing 4.5: The Git log after Bob merges in Alice's changes. (Exact results will differ.)

```
[website-copy (master)]$ git log
commit 86dccde63ac15331a068ce79fa9c83d8b784b28b
Merge: 9b7eda1 5ca69e4
Author: Michael Hartl <michael@michaelhartl.com>
Date:   Mon Dec 28 13:14:44 2015 -0800

    Merge branch 'master' of https://github.com/mhartl/website

commit 9b7eda1b0a95740a241684b82d4474aa8f16ae45
Author: Michael Hartl <michael@michaelhartl.com>
Date:   Mon Dec 28 13:13:37 2015 -0800

    Add an image

commit 5ca69e4dca9487b5cd7e1be52222c5389392527d
Author: Michael Hartl <michael@michaelhartl.com>
Date:   Mon Dec 28 13:02:42 2015 -0800

    Change page heading
```

If Bob now pushes, it should go through as expected:

```
$ git push
```

This puts Bob's changes on the remote repo, which means Alice can pull them in:

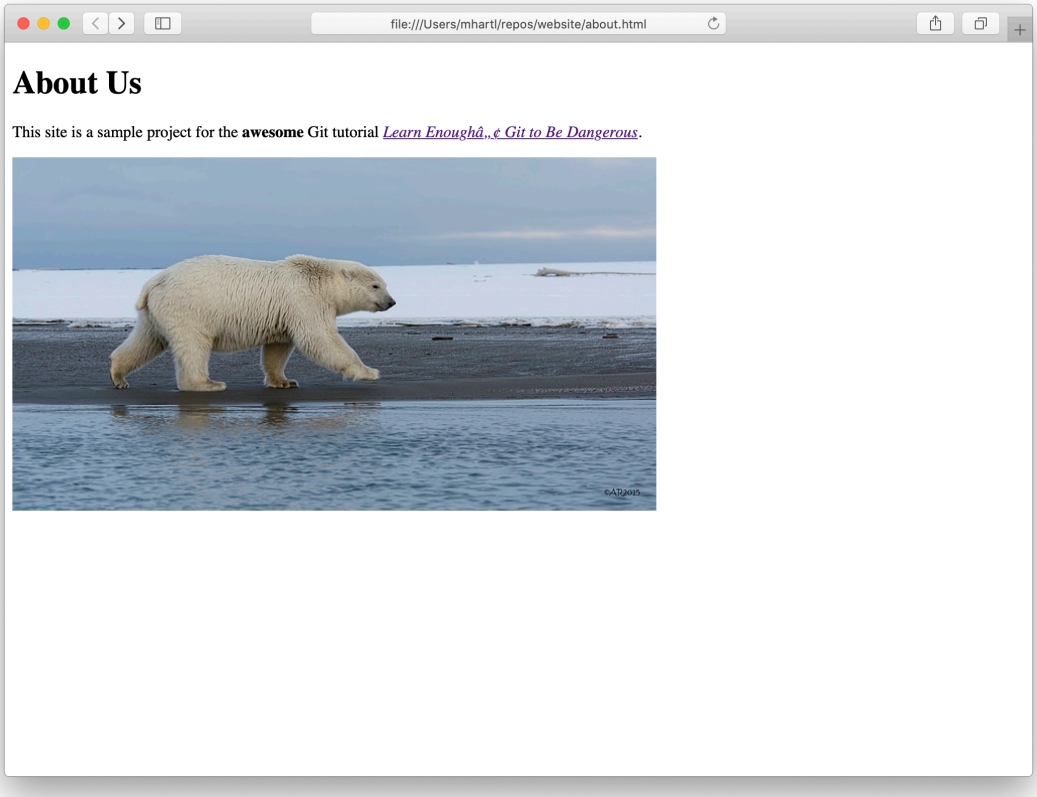

Figure 4.14: Confirming that Alice's repo includes Bob's added image.

```
$ git pull
```

Alice can confirm that her repo now includes Bob's changes by inspecting the Git log, which should match the results you got in Listing 4.5. Meanwhile, she can refresh her browser to see Bob's cool new ursine addition (Figure 4.14).

4.2.2 Conflicting changes

Even though Git's merge algorithms can often figure out how to combine changes from different collaborators, sometimes there's no avoiding a conflict.

For example, suppose both Alice and Bob notice that the required `alt` attribute is missing from the whale image included in Listing 3.1 and decide to correct the issue by adding one.

First, Alice adds the `alt` attribute "Breaching whale" (Listing 4.6).

Listing 4.6: Alice's image `alt`.
~/repos/website/index.html

```
<!DOCTYPE html>
<html>
    .
    .
    .
    <a href="https://www.flickr.com/photos/28883788@N04/10097824543">
      <img src="images/breaching_whale.jpg" alt="Breaching whale">
    </a>
  </body>
</html>
```

She then commits and pushes her change:[5]

```
[website (master)]$ git commit -am "Add necessary image alt"
[website (master)]$ git push
```

[5] Listing 4.6 and Listing 4.7 include the attribution link added in the Section 4.1.1 exercises.

Listing 4.7: Bob's image `alt`.
~/tmp/website-copy/index.html

```
<!DOCTYPE html>
<html>
    .
    .
    .
    <a href="https://www.flickr.com/photos/28883788@N04/10097824543">
      <img src="images/breaching_whale.jpg" alt="Whale">
    </a>
  </body>
</html>
```

Meanwhile, Bob adds his own `alt` attribute, "Whale" (Listing 4.7), and commits his change:

```
[website-copy (master)]$ git commit -am "Add an alt attribute"
```

If Bob tries to `push`, he'll be met with the same rejection message shown in Listing 4.4, which means he should pull—but that comes at a cost:

```
[website-copy (master)]$ git pull
remote: Counting objects: 3, done.
remote: Compressing objects: 100% (1/1), done.
remote: Total 3 (delta 2), reused 3 (delta 2), pack-reused 0
Unpacking objects: 100% (3/3), done.
From https://github.com/mhartl/website
   5ca69e4..7ada3b5  master     -> origin/master
Auto-merging index.html
CONFLICT (content): Merge conflict in index.html
Automatic merge failed; fix conflicts and then commit the result.
[website-copy (master|MERGING)]$
```

As indicated in the second highlighted line, Git has detected a merge conflict from Bob's pull, and his working copy has been put into a special branch state called **master|MERGING**.

Bob can see the effect of this conflict by viewing **index.html** in his text editor, as shown in Figure 4.15. Supposing Bob prefers Alice's more descriptive **alt** text, he can resolve the conflict by deleting all but the line with **alt="Breaching whale"**, as seen in Figure 4.16.

After saving the file, Bob can commit his change, which causes the prompt to revert back to displaying the **master** branch, and at that point he's ready to **push**:

```
[website-copy (master|MERGING)]$ git commit -am "Use longer alt attribute"
[website-copy (master)]$ git push
```

Alice's and Bob's repos now have the same content, but it's still a good idea for Alice to pull in Bob's merge commit:

```
[website (master)]$ git pull
```

Because of the potential for conflict, it's a good idea to do a **git pull** before making any changes on a project with multiple collaborators (or even just being edited by the same person on different machines). Even then, on a long enough timeline some conflicts are inevitable, and with the techniques in this section you're now in a position to handle them.

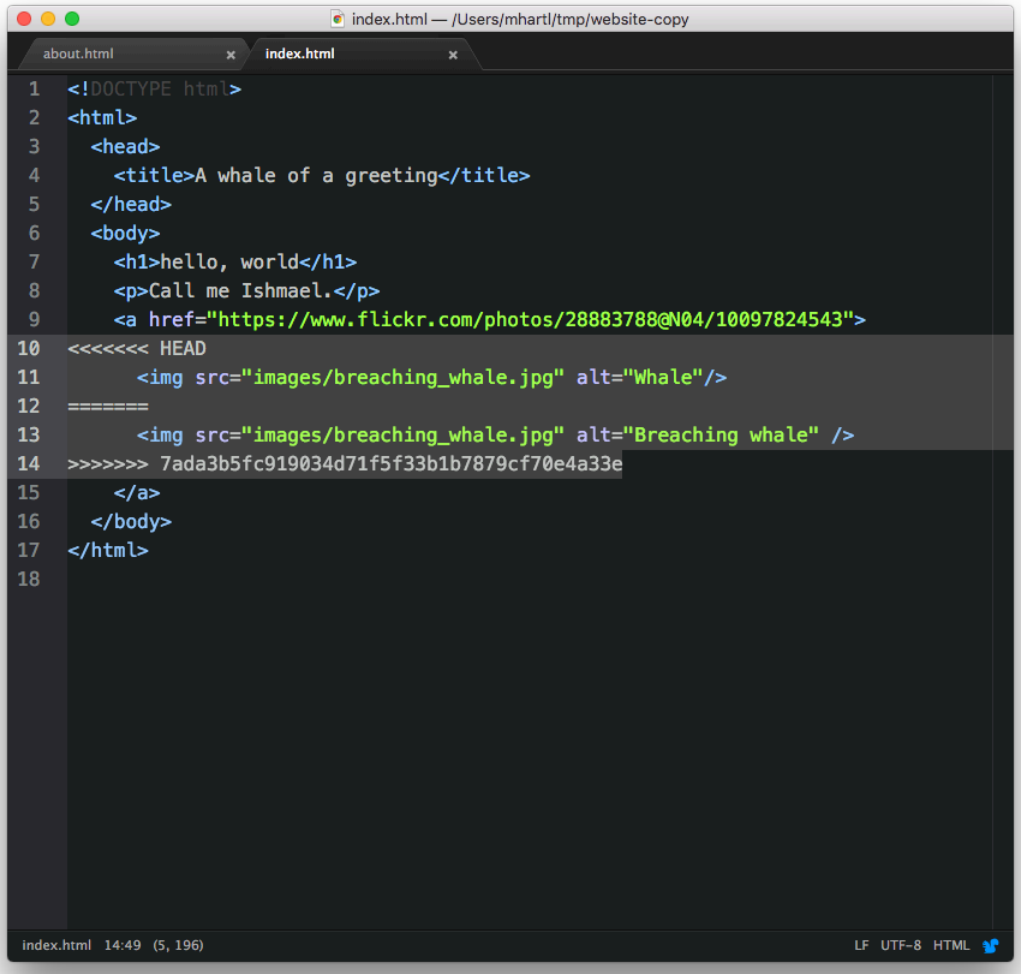

Figure 4.15: A file with a merge conflict.

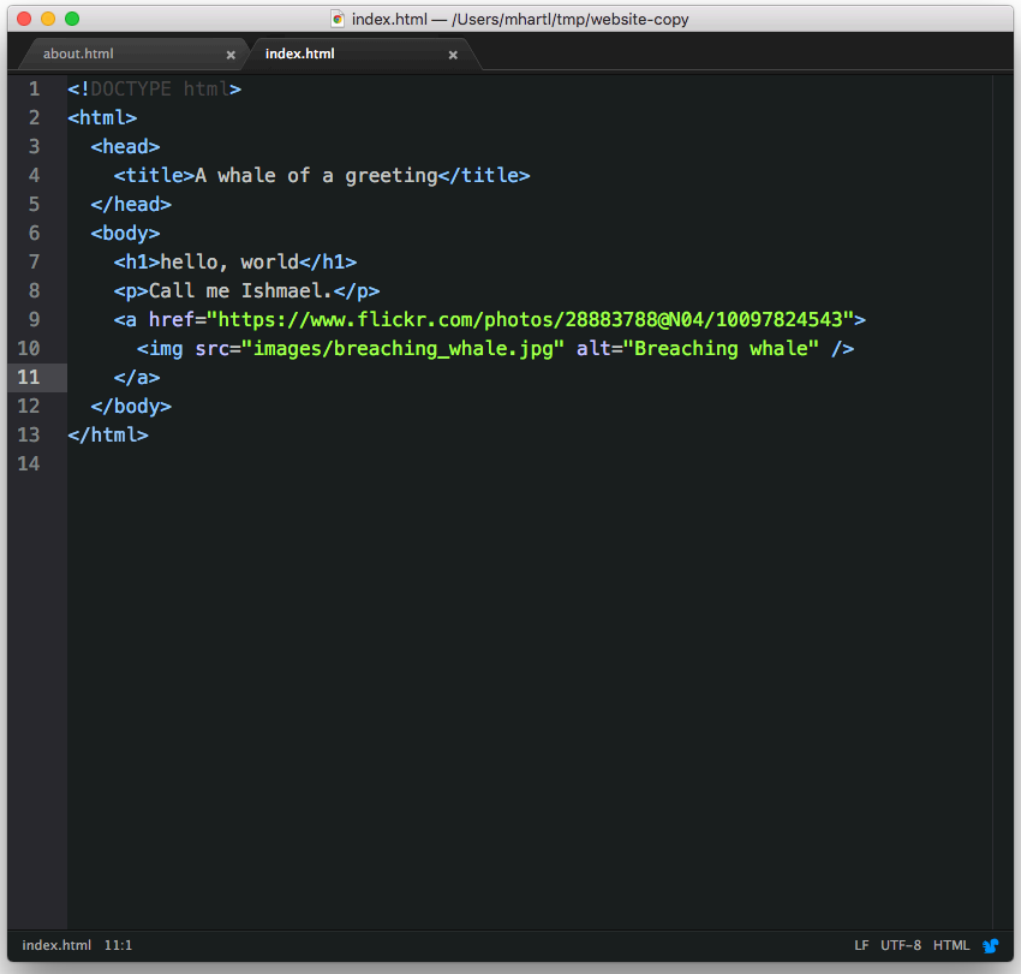

Figure 4.16: The HTML file edited to remove the merge conflict.

4.2.3 Exercises

1. Change your default Git editor from Vim to Atom. *Hint*: Google for it. (This is an absolutely *classic* application of technical sophistication (Box 1.2): With a well-chosen Google search, you can often go from "I have no idea how to do this" to "It's done" in under 30 seconds.)

2. The polar bear picture added in Listing 4.3 (Figure 4.11) requires attribution under the Creative Commons Attribution 2.0 Generic license. As Alice, link the image to the original attribution page, as shown in Listing 4.8. Then run `git commit -a` *without* including `-m` and a command-line message. This should drop you into the default Git editor. Quit the editor *without* including a message, which cancels the commit.

3. Run `git commit -a` again, but this time add the commit message "Add polar bear attribution link". Then hit return a couple of times and add a longer message of your choice. (One example appears in Figure 4.17.) Save the message and exit the editor.

4. Run `git log` to confirm that both the short and longer messages correctly appear. After pushing the changes to GitHub, navigate to the page for the commit to confirm that both the short and longer messages correctly appear.

5. As Bob, pull in the changes to the About page. Verify by refreshing the browser and by running `git log -p` that Bob's repo has been properly updated.

Listing 4.8: Linking to the polar bear image's attribution page.
~/repos/website/about.html

```
.
.
.
<a href="https://www.flickr.com/photos/puliarfanita/22959238329">
  <img src="images/polar_bear.jpg" alt="Polar bear">
</a>
.
.
.
```

4.3 Pushing branches

In this section, we'll apply our newfound collaboration skills to get Alice to request a bugfix from Bob, who will make the correction and then share the result with Alice. In the process, we'll learn how to collaborate on branches other than **`master`**, thereby applying the material from Section 3.3 as well.

Recall from Section 3.3 that the trademark character ™ is currently broken on the About page (Figure 3.7). Alice suspects the fix for this involves adding some markup to the HTML template for the website's pages, but she's already agreed to attend a tea party (Figure 4.18),[6] so she only has time to add a couple of *HTML comments* requesting for Bob to add the relevant fix, as shown in Listing 4.9 and Listing 4.10. (We'll cover HTML comments further in *Learn Enough HTML to Be Dangerous*.)

Listing 4.9: A stub for the fix to the ™ problem.
~/repos/website/about.html

```
<!DOCTYPE html>
<html>
  <head>
```

[6] *Alice's Adventures in Wonderland* original illustrations by John Tenniel. Image retrieved from Wikimedia on 2019-02-15. Copyright © 1890, now in the public domain.

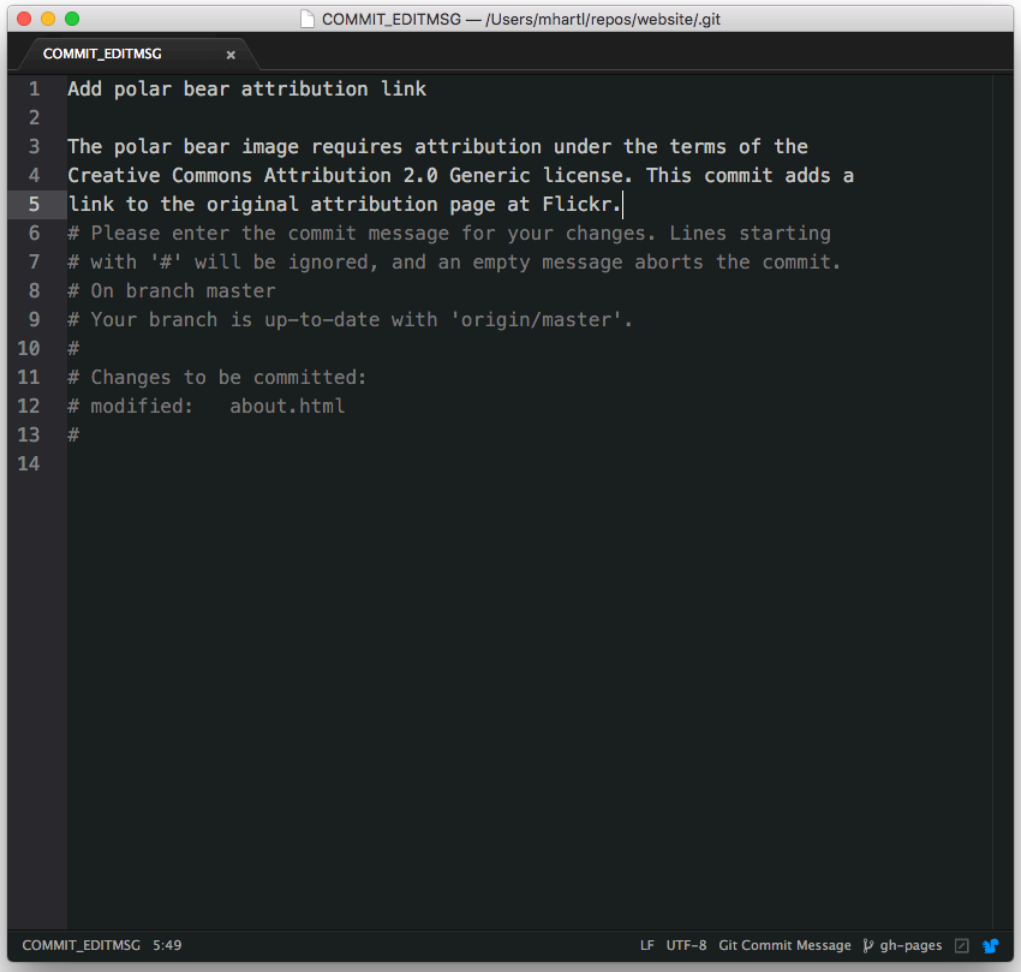

Figure 4.17: Adding a longer message in a text editor.

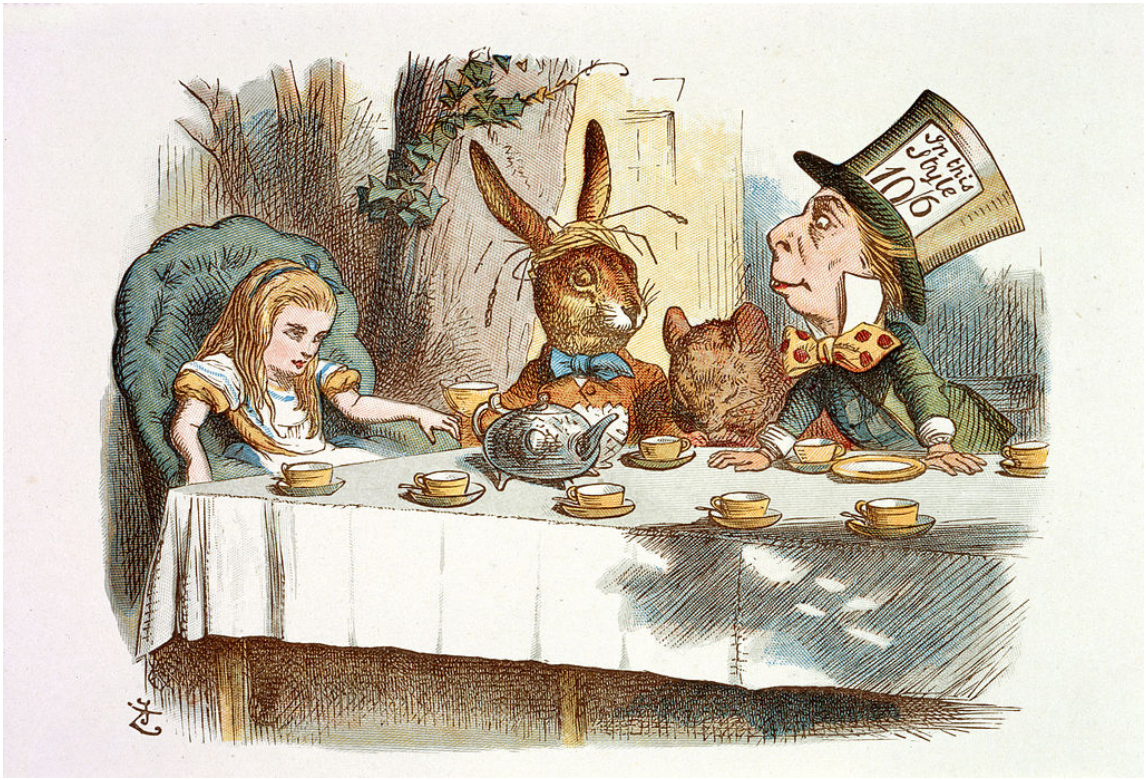

Figure 4.18: Alice has a tea party to attend and so asks Bob to fix the website.

```
    <title>About Us</title>
    <!-- Add something here to fix trademark -->
  </head>
  .
  .
  .
</html>
```

Listing 4.10: A stub to add the ™ fix to the index page.
~/repos/website/index.html

```
<!DOCTYPE html>
<html>
  <head>
    <title>A whale of a greeting</title>
    <!-- Add something here to fix trademark -->
  </head>
  .
  .
  .
</html>
```

Notice that Alice has wisely asked Bob to fix the index page as well (Listing 4.10) even though the current error only occurs on the About page. This way, any ™ or similar characters added to **`index.html`** will automatically work in the future. (As noted in Section 3.3, having to make such changes in multiple places is annoying, and it's also brittle and error-prone. The correct solution is to use *templates*, which we'll cover starting in *Learn Enough CSS & Layout to Be Dangerous*.)

Alice has decided to follow a common convention and use a separate branch for the bugfix, which in this case she calls **`fix-trademark`**:

```
[website (master)]$ git checkout -b fix-trademark
[website (fix-trademark)]$
```

This shows something important: it's possible to make changes to the working directory (in this case, the additions from Listing 4.9 and Listing 4.10) *before* creating a new branch, as long as those changes haven't yet been committed.

Having made the new branch for the fix, Alice can make a commit and push up the branch using **`git push`**:

```
[website (fix-trademark)]$ git commit -am "Add placeholders for the TM fix"
[website (fix-trademark)]$ git push -u origin fix-trademark
```

Here Alice has used exactly the same **`push`** syntax used in Listing 2.1 to push the repo up to GitHub in the first place, with **`fix-trademark`** in place of **`master`**.

If Alice sends Bob a note before she heads off to her tea party, Bob will know to do a **`git pull`** to pull in Alice's changes:

```
[website-copy (master)]$ git pull
remote: Counting objects: 4, done.
remote: Compressing objects: 100% (1/1), done.
remote: Total 4 (delta 3), reused 4 (delta 3), pack-reused 0
Unpacking objects: 100% (4/4), done.
From https://github.com/mhartl/website
 * [new branch]      fix-trademark -> origin/fix-trademark
Already up-to-date.
```

Bob can check his local working directory for the **`fix-trademark`** branch that Alice created and pushed, but it isn't there:

```
[website-copy (master)]$ git branch
* master
```

The reason is that the branch is associated with the remote **origin**, and such branches aren't displayed by default. To see it, Bob can use the **-a** option (for "all"):[7]

```
[website-copy (master)]$ git branch -a
* master
  remotes/origin/HEAD -> origin/master
  remotes/origin/fix-trademark
  remotes/origin/master
```

To start work on **fix-trademark** on his local copy, Bob just needs to check it out. By using the same name (i.e., **fix-trademark**), he arranges for it to be associated with the upstream branch on GitHub, which means that **git push** will automatically push up his changes:

```
[website-copy (master)]$ git checkout fix-trademark
Branch fix-trademark set up to track remote branch fix-trademark from origin.
Switched to a new branch 'fix-trademark'
[website-copy (fix-trademark)]$
```

At this point, Bob can **diff** against **master** to see what he's dealing with:

[7] In fact, **git branch --all** works, but when using Git at the command line it's more common to use the short forms of the options.

```
[website-copy (fix-trademark)]$ git diff master
diff --git a/about.html b/about.html
index 8a879f5..3d567eb 100644
--- a/about.html
+++ b/about.html
@@ -2,6 +2,7 @@
 <html>
   <head>
     <title>About Us</title>
+    <!-- Add something here to fix trademark -->
   </head>
   <body>
     <h1>About Us</h1>
diff --git a/index.html b/index.html
index fcb80f4..c4920c0 100644
--- a/index.html
+++ b/index.html
@@ -2,6 +2,7 @@
 <html>
   <head>
     <title>A whale of a greeting</title>
+    <!-- Add something here to fix trademark -->
   </head>
   <body>
     <h1>hello, world</h1>
```

Now all Bob has to do is actually implement the fix. If you'd like a challenging exercise in technical sophistication, try Googling around to see if you can figure out what the problem might be, and also how you might fix it. In case you'd like to do this, I'll wait here while you look…

All right, the problem is that the page doesn't have the right *character encoding* to display non-ASCII characters like ™, ®, or £. The fix involves using a tag called **`meta`** to tell browsers to use a character set (or **`charset`** for short) called UTF-8, which will let our page display anything that's part of the enormous set of Unicode characters. The result, which you would not necessarily be able to guess, appears in Listing 4.11 and Listing 4.12.

Listing 4.11: A fix for the ™ problem.
~/tmp/website-copy/about.html

```
<!DOCTYPE html>
<html>
  <head>
    <title>About Us</title>
    <meta charset="utf-8">
  </head>
  .
  .
  .
</html>
```

Listing 4.12: Adding the ™ fix to the index page.
~/tmp/website-copy/index.html

```
<!DOCTYPE html>
<html>
  <head>
    <title>A whale of a greeting</title>
    <meta charset="utf-8">
  </head>
  .
  .
  .
</html>
```

Like the `img` tag introduced in Section 3.1, `meta` is a void element (self-closing) and so has no closing tag.

Having made the change, Bob can confirm the fix by reloading the page in his browser, as shown in Figure 4.19.

Confident that his solution is correct, Bob can now make a commit and push the fix up to the remote server:

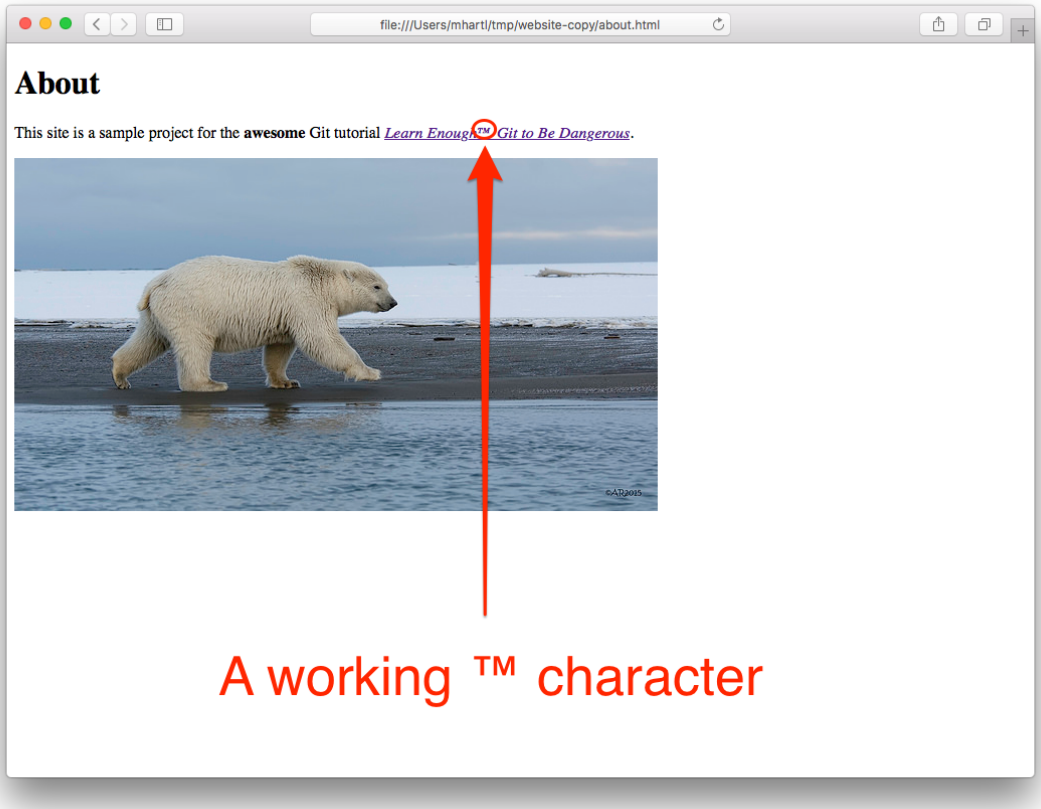

Figure 4.19: Confirming a working trademark character.

Figure 4.20: Bob's reward for a job well-done.

```
[website-copy (fix-trademark)]$ git commit -am "Fix trademark character display"
[website-copy (fix-trademark)]$ git push
```

With that, Bob sends a note to Alice that the fix is pushed, and heads out for some well-deserved rest (Figure 4.20).[8]

Alice, now back from her tea party, gets Bob's note and pulls in his fix:

[8]Image retrieved from https://www.flickr.com/photos/rtadlock/2716877199 on 2016-01-06. Copyright © 2008 by Robert Tadlock and used unaltered under the terms of the Creative Commons Attribution 2.0 Generic license.

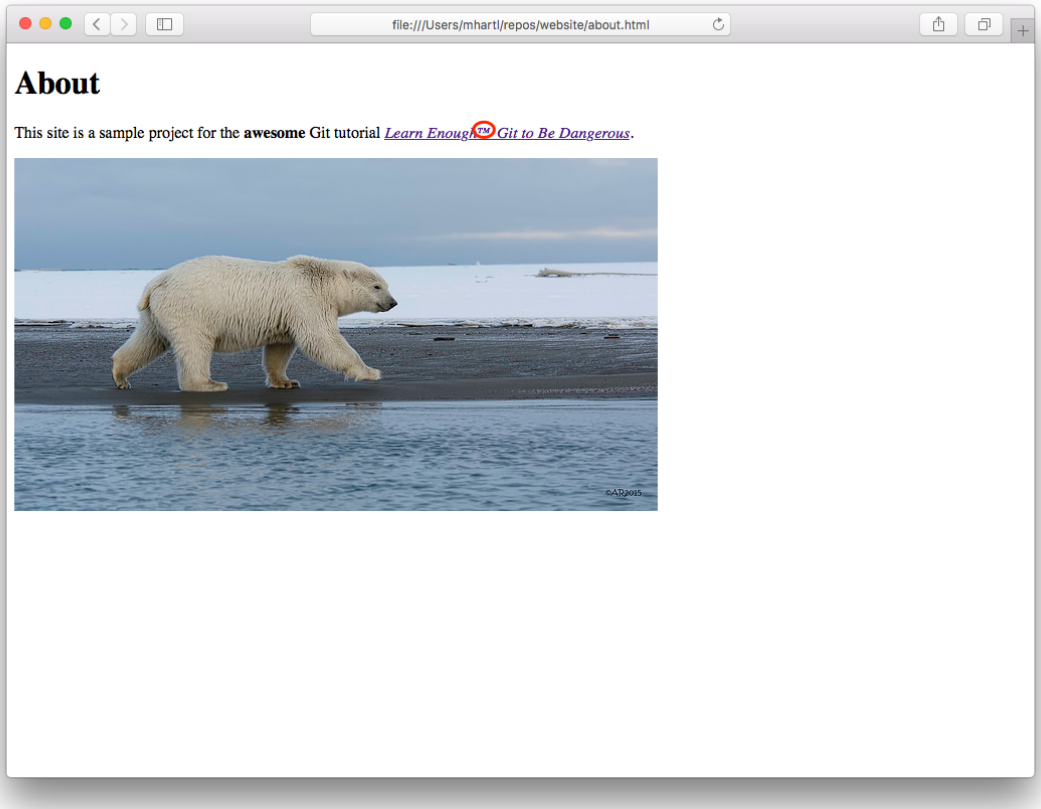

Figure 4.21: Reconfirming the trademark fix before merging.

```
[website (fix-trademark)]$ git pull
```

She refreshes her browser to confirm that the ™ character displays properly on her end of things (Figure 4.21), and then merges the changes into **`master`**:

```
[website (fix-trademark)]$ git checkout master
[website (master)]$ git merge fix-trademark
[website (master)]$ git push
```

With the final **`git push`**, Alice arranges for the remote **`master`** branch on GitHub to get the fix. (Syncing up Bob's **`master`** branch is left as an exercise (Section 4.3.1).)

Of course, **`git push`** publishes the change only to a remote Git repository. Wouldn't it be nice if there were a way to confirm that the ™ character—and the rest of the website—displays correctly on the live Web?

4.3.1 Exercises

1. Bob's **`master`** branch doesn't currently have Alice's merge, so check out **`master`** as Bob and do a **`git pull`**. Confirm using **`git log`** that Alice's merge commit is now present.

2. Delete the **`fix-trademark`** branch locally. Do you need to use the **`-D`** option (Section 3.3.2), or is **`-d`** sufficient?

3. Delete the remote **`fix-trademark`** branch on GitHub. *Hint*: If you get stuck, Google for it.

4.4 A surprise bonus

As hinted at the end of the last section, it would be nice to be able to confirm that the new character encoding works on a live web page. But this requires knowing how to deploy a live site to the Web, and that's beyond the scope of a humble Git tutorial, right? Amazingly, the answer is no. The reason is that GitHub offers a free service called *GitHub Pages*, and *any* repository at GitHub containing static HTML is automatically available as a live website.

There is one minor prerequisite to using GitHub Pages, which is that you have to verify your email address with GitHub. Once you've done that, though, all you need to do is configure your repository to use GitHub Pages on the **`master`** branch, which you can do by going to the settings (Figure 4.22) and then selecting the **`master`** and saving the changes (Figure 4.23).

That's it! Our website is now available at the URL

```
https://<name>.github.io/website/
```

where **`<name>`** is your GitHub username. Since my username is **`mhartl`**, my copy of the this tutorial's website is at mhartl.github.io/website/, as shown in Figure 4.24.

Note that the URL **`https://<name>.github.io/website/`** automatically displays **`index.html`**, which is the usual convention on the web: the index page is understood to be the default, so there's no need to type it in. This is not the case with other pages, though, and if you follow the link to the About page you'll see that the filename appears in the address bar (Figure 4.25). You'll also see in Figure 4.25 that the trademark character ™ renders correctly on a live website, just as we hoped it would.

Because static HTML pages by definition don't change from one page view to the next, GitHub can *cache* them efficiently, storing them for the next user who visits the site. This makes GitHub Pages sites both fast and cheap to serve (which is why GitHub can afford to offer them for free). You can even configure GitHub Pages to work with a custom domain, letting you replace <name>.github.io with something like www.example.com; see the free tutorial *Learn Enough Custom Domains to Be Dangerous* to learn how to do it.

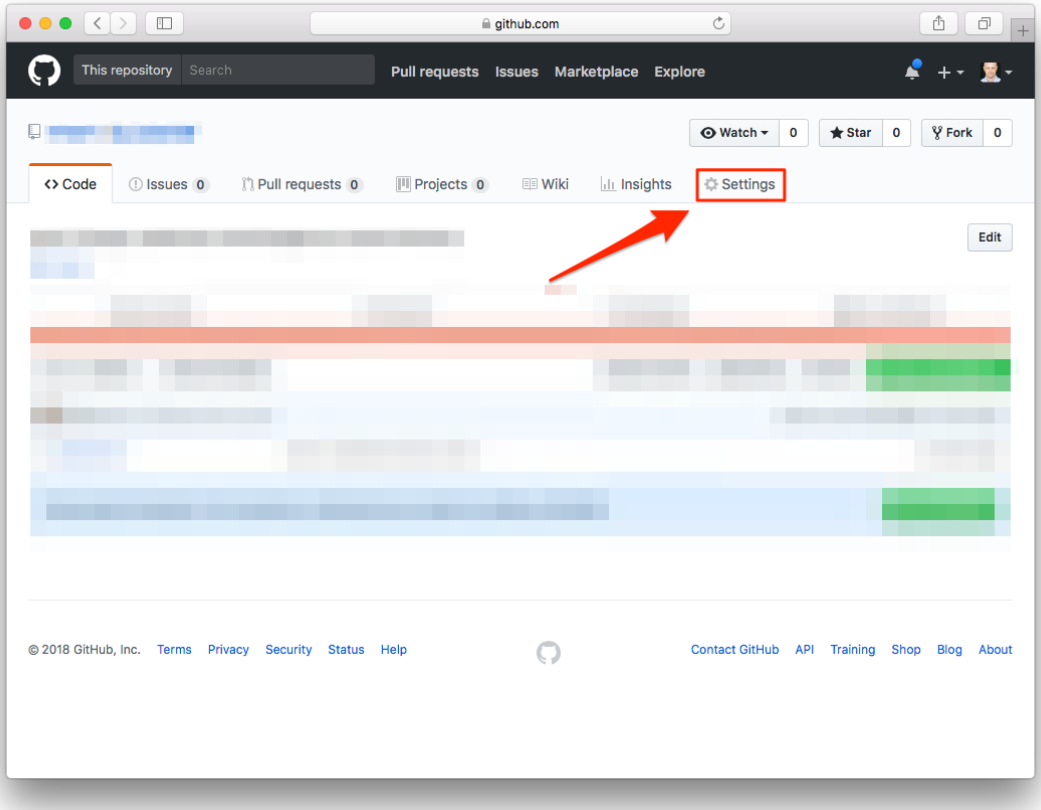

Figure 4.22: The settings for a GitHub repository.

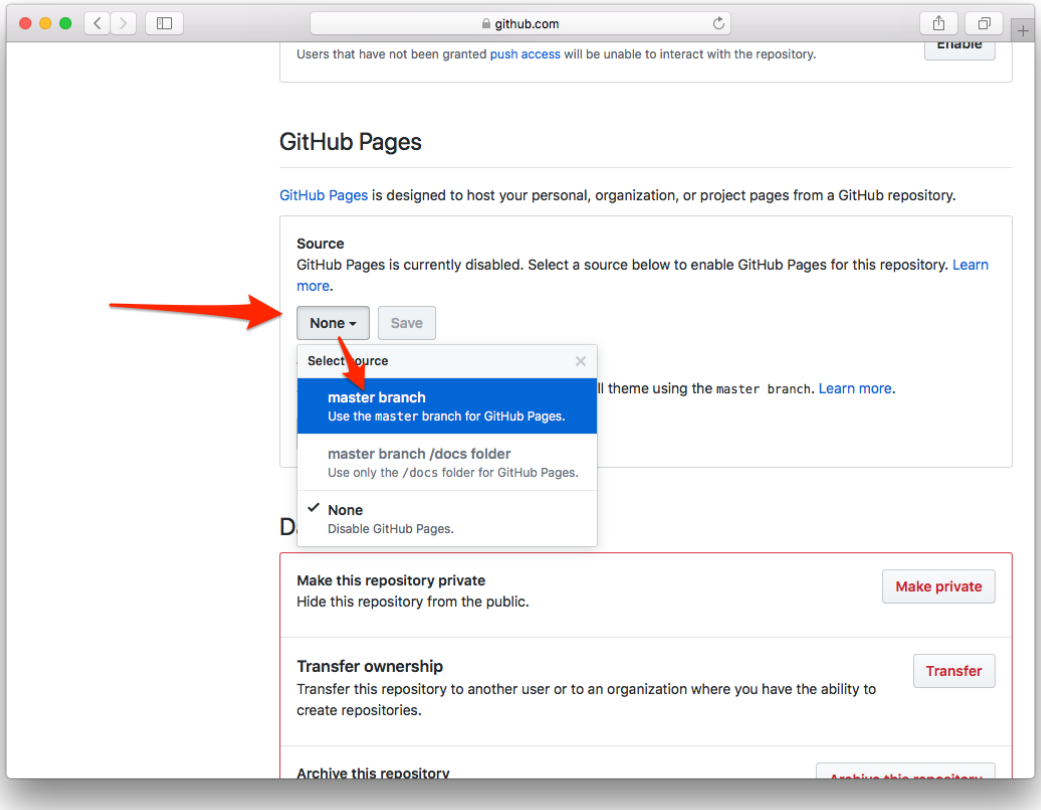

Figure 4.23: Serving our website from the **`master`** branch.

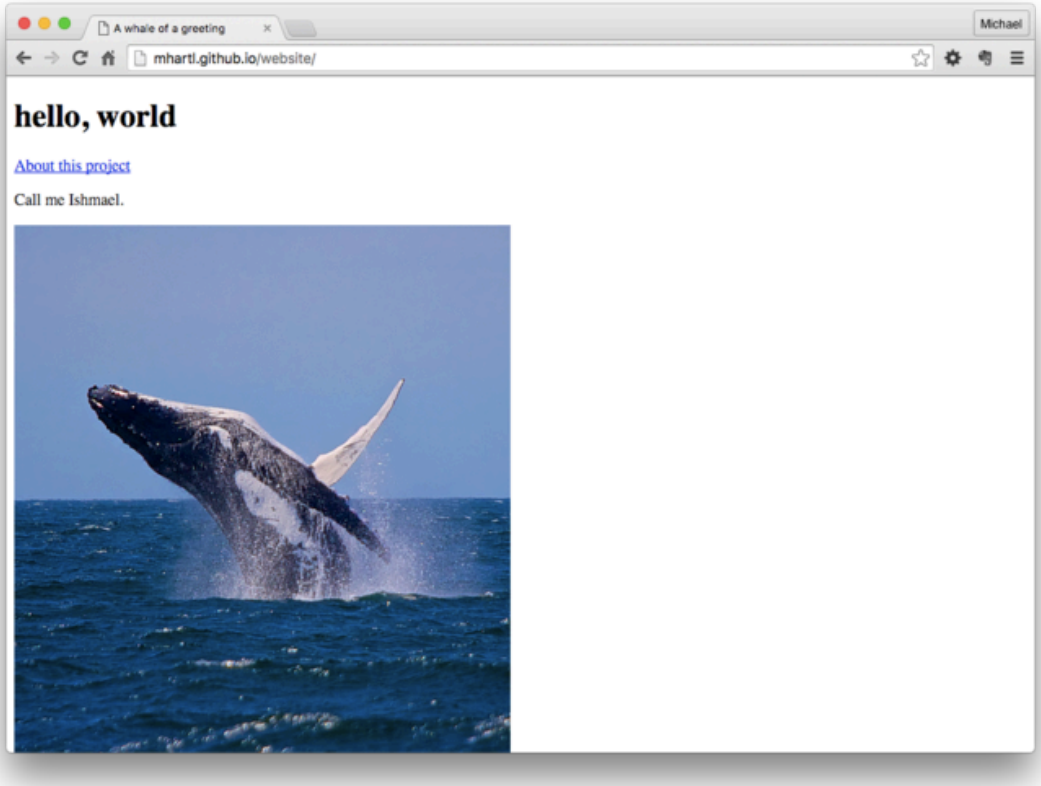

Figure 4.24: A production website at GitHub Pages.

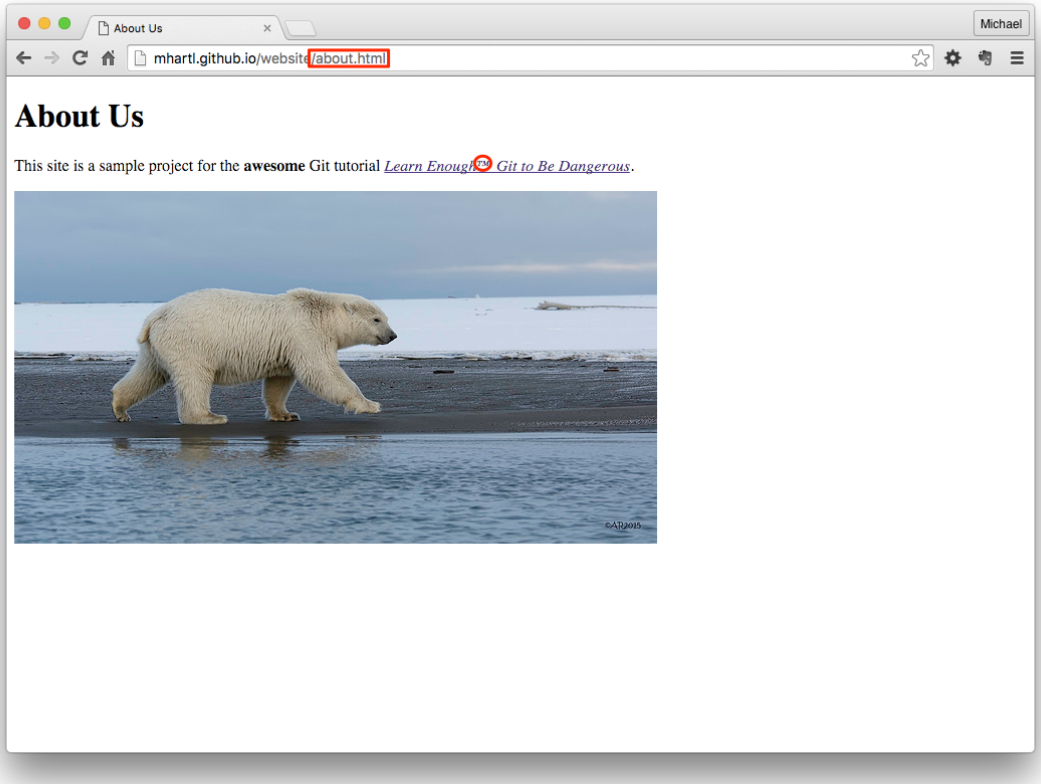

Figure 4.25: The About page in production.

This combination of high performance and support for custom domains makes GitHub Pages suitable for production websites—for example, the Learn Enough blog is a static website running on a custom domain at GitHub Pages.

The example website in this tutorial is really just a toy, but it's a great start, and we'll build on this foundation to make a nearly industrial-grade website in *Learn Enough HTML to Be Dangerous* and a fully industrial-grade site in *Learn Enough CSS & Layout to Be Dangerous*.

4.4.1 Exercises

1. On the About page, add a link back to `index.html`. Commit and push your change and verify that the link works on the production site.

2. As covered in *Learn Enough Command Line to Be Dangerous*, two of the most important Unix commands are `mv` and `rm`. Git provides analogues of these commands, which have the same effect on local files while also arranging to track the changes. Experiment with these commands via the following sequence: Create a file with some *lorem ipsum* text, add & commit it, rename it with `git mv` & commit, then remove it with `git rm` and commit again. Examine the results of `git log -p` to see how Git handled the operations.

3. To practice the process of making a new Git repository, make a second project called `second_website` in the `repos` directory. Create an `index.html` file with the content "hello, again!" and follow the steps (starting in Section 1.2) needed to deploy it to the live Web.

4. Make a third, secret project called `secret_project`. Touch files called `foo`, `bar`, and `baz` in the main project directory, and then follow the steps to initialize the repository and commit the initial results. Then, to practice using a service other than GitHub, create a free private repository at Bitbucket.

Command	Description	Example
`git clone <URL>`	Copy repo (incl. full history) to local disk	`$ git clone https://ex.co/repo.git`
`git pull`	Pull in changes from remote repository	`$ git pull`
`git branch -a`	List all branches	`$ git branch -a`
`git checkout  `	Check out remote branch and configure for push	`$ git checkout fix-trademark`

Table 4.1: Important commands from Chapter 4.

4.5 Summary

Important commands from this chapter are summarized in Table 4.1.

4.6 Advanced setup

This section contains some optional advanced Git setup. The main features are adding an alias for checking out branches, adding the branch name to the Unix prompt, and enabling branch name tab completion. Following the steps in this section should be within your capabilities if you completed *Learn Enough Command Line to Be Dangerous* and *Learn Enough Text Editor to Be Dangerous*, but they can be tricky, so use your technical sophistication (Box 1.2) if you get stuck. If you'd rather skip these steps for now, you can proceed directly to the conclusion (Section 4.7).

Note for Mac users: The instructions below assume you are using Bash, as described in Box 1.3. To learn how to set up your Git system using Z shell instead, see the Learn Enough blog post "Using Z Shell on Macs with the Learn Enough Tutorials".

4.6.1 A checkout alias

In Chapter 1, we added global configuration settings for the name and email address (Listing 1.1) to be included automatically when making commits. Now we'll add a third config setting, an *alias* to make it easier to check out branches.

Throughout this tutorial, we've used `git checkout` to check out branches (e.g., Listing 3.3), but most experienced Git users configure their systems to use the shorter command `git co`.[9] The way to do this is with a Git *alias*: much as the Bash aliases covered in *Learn Enough Text Editor to Be Dangerous* let us add commands to our Bash shell, Git aliases let us add commands to our Git system. In particular, the way to add the `co` alias is to run the command shown in Listing 4.13.

Listing 4.13: Adding an alias for `git co`.

```
$ git config --global alias.co checkout
```

In effect, this adds `co` as a new Git command, and running Listing 4.13 allows us to replace `checkout` in commands like

```
$ git checkout master
```

with the more compact `co` command, as follows:

```
$ git co master
```

For maximum compatibility with systems that don't have `co` configured, this tutorial has always used the full `checkout` command, but in real life I nearly always use `git co`.

4.6.2 Prompt branches and tab completion

In this section, we'll add two final advanced customizations. First, we'll arrange for the command-line prompt to include the name of the current branch. Second, we'll add the ability to fill in Git branch names using *tab completion*, which is especially convenient when dealing with longer branch names. Both of these

[9] This choice is no doubt influenced by the analogous command `svn co` used by Subversion, one of Git's main predecessors.

features come as shell scripts with the Git source code distribution, which can be downloaded as shown in Listing 4.14.

Listing 4.14: Downloading scripts for branch display and tab completion.

```
$ curl -o ~/.git-prompt.sh -OL https://cdn.learnenough.com/git-prompt.sh
$ curl -o ~/.git-completion.bash \
>      -OL https://cdn.learnenough.com/git-completion.bash
```

Here the `-o` flag arranges to save the files locally under slightly different names from the ones on the server, prepending a dot `.` so that the files are hidden and saving them in the home directory `~`.

After downloading the scripts as in Listing 4.14, on some systems we need to make them executable, which we can do with the `chmod` command (mentioned before in *Learn Enough Text Editor to Be Dangerous*):

```
$ chmod +x ~/.git-prompt.sh
$ chmod +x ~/.git-completion.bash
```

Next, we need to tell the shell about the new commands, so open up the Bash profile file in your favorite editor (which for simplicity I'll assume is Atom):

```
$ atom ~/.bash_profile
```

Then add the configuration shown in Listing 4.15 to the bottom of the file. Also, make sure to delete any other lines starting with `PS1` (which you'll have to do if you modified `.bash_profile` as shown in *Learn Enough Text Editor to Be Dangerous*).

Listing 4.15: Git configuration in the `.bash_profile` file.
~/.bash_profile

```
.
.
.
```

```
# Git configuration
# Branch name in prompt
source ~/.git-prompt.sh
PS1='[\W$(__git_ps1 " (%s)")]\$ '
export PROMPT_COMMAND='echo -ne "\033]0;${PWD/#$HOME/~}\007"'
# Tab completion for branch names
source ~/.git-completion.bash
```

Note: The vertical dots in Listing 4.15 indicate omitted content and should not be copied literally. This is the sort of thing you can figure out using your technical sophistication (Box 1.2). Speaking of which, I have hardly any idea of what most of the code in Listing 4.15 means; part of having technical sophistication means be able to copy things from the Internet and get them to work even when you have no idea what you're doing (Figure 4.26).

Once we've saved the result of editing **`.bash_profile`**, we have to *source* it to make the changes active (as discussed in *Learn Enough Text Editor to Be Dangerous*):

```
$ source ~/.bash_profile
```

At this point, the prompt for a Git repository's default **`master`** branch should look something like this:

```
[website (master)]$
```

If you skipped ahead from Section 1.1 to complete this section, you'll have to wait until Section 1.2 to see this effect. Checking that tab completion is working is left as an exercise (Section 4.6.3).

4.6.3 Exercises

1. Create a branch called **`really-long-branch-name`** using **`git co -b`**.
2. Switch back to the **`master`** branch using **`git co`**.

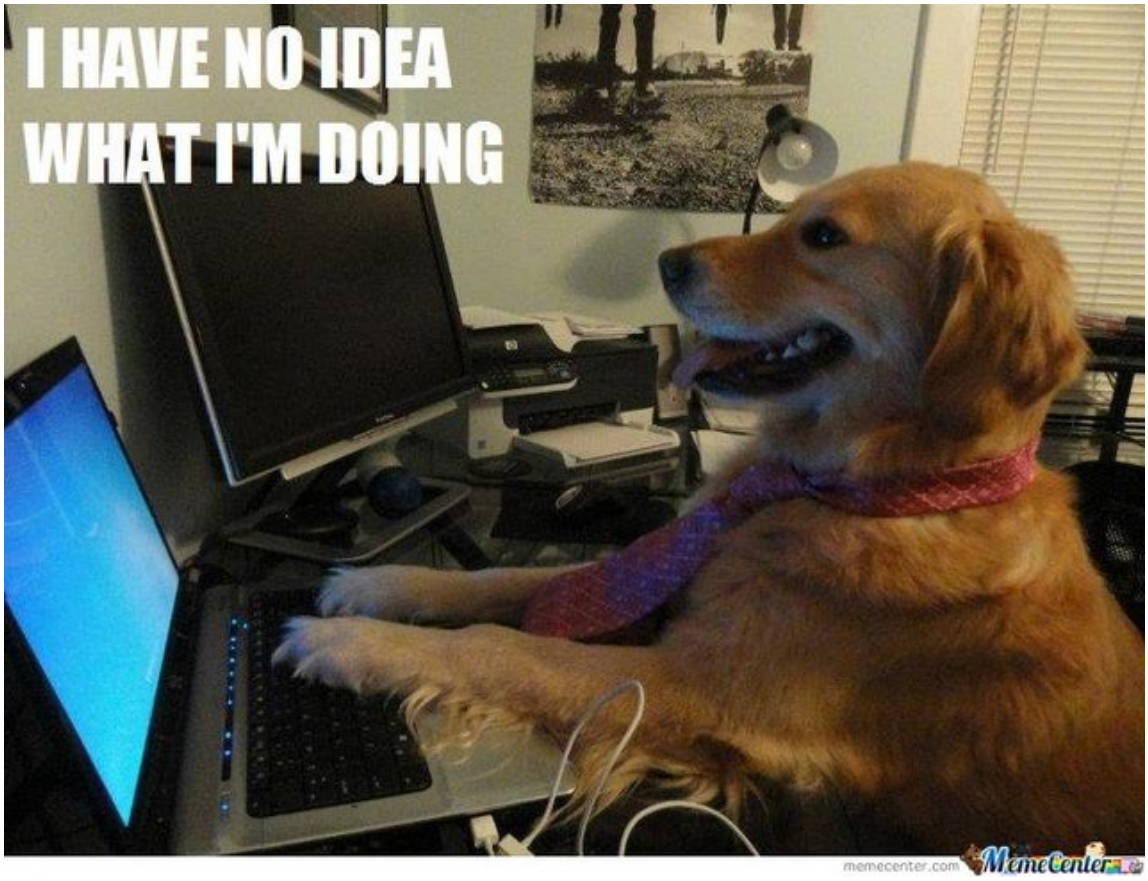

Figure 4.26: It's OK—neither does anyone else.

3. Check out the branch `really-long-branch-name` using tab completion by typing `git checkout r`→| at the command-line prompt.
4. What does your prompt look like? Verify that the correct branch name appears in the prompt.
5. Check out the `master` branch using `git co m`→|. (This shows that tab completion works with the `co` alias set up in Listing 4.13.) What does the prompt look like now?
6. Use `git branch -d r`→| to delete `really-long-branch-name`, thus verifying that tab completion works with `git branch` as well as with `git checkout`. (In fact, tab completion works with most relevant Git commands.)

4.7 Conclusion

Congratulations! You now know enough Git to be *dangerous*. There's a lot more to learn, and if you continue down this technical path you'll keep getting better at using Git for years to come, but with the material in this tutorial you've got a great start. For now, you're probably best off working with what you've got, applying your technical sophistication (Box 1.2) when necessary. Once you've gotten a little more experience under your belt, I recommend seeking out additional resources. Here are some suggestions for getting started:

- *Pro Git* by Scott Chacon and Ben Straub
- Learn Git at Codecademy
- Git tutorials by Atlassian (makers of Bitbucket)
- Tower Git tutorials

At this point, you have completed the **Learn Enough Developer Fundamentals** and are in an excellent position to collaborate with millions of software developers around the world. You are also well on your way to becoming

a developer yourself. Regardless of your ultimate goals, you can continue improving your dev skills with the rest of the core Learn Enough sequence:

1. **Developer Fundamentals**
   (a) *Learn Enough Command Line to Be Dangerous*
   (b) *Learn Enough Text Editor to Be Dangerous*
   (c) *Learn Enough Git to Be Dangerous* (you are here)
2. **Web Basics**
   (a) *Learn Enough HTML to Be Dangerous*
   (b) *Learn Enough CSS & Layout to Be Dangerous*
   (c) *Learn Enough JavaScript to Be Dangerous*
3. **Application Development**
   (a) *Learn Enough Ruby to Be Dangerous*
   (b) *The Ruby on Rails Tutorial*
   (c) *Learn Enough Action Cable to Be Dangerous* (optional)

Good luck!