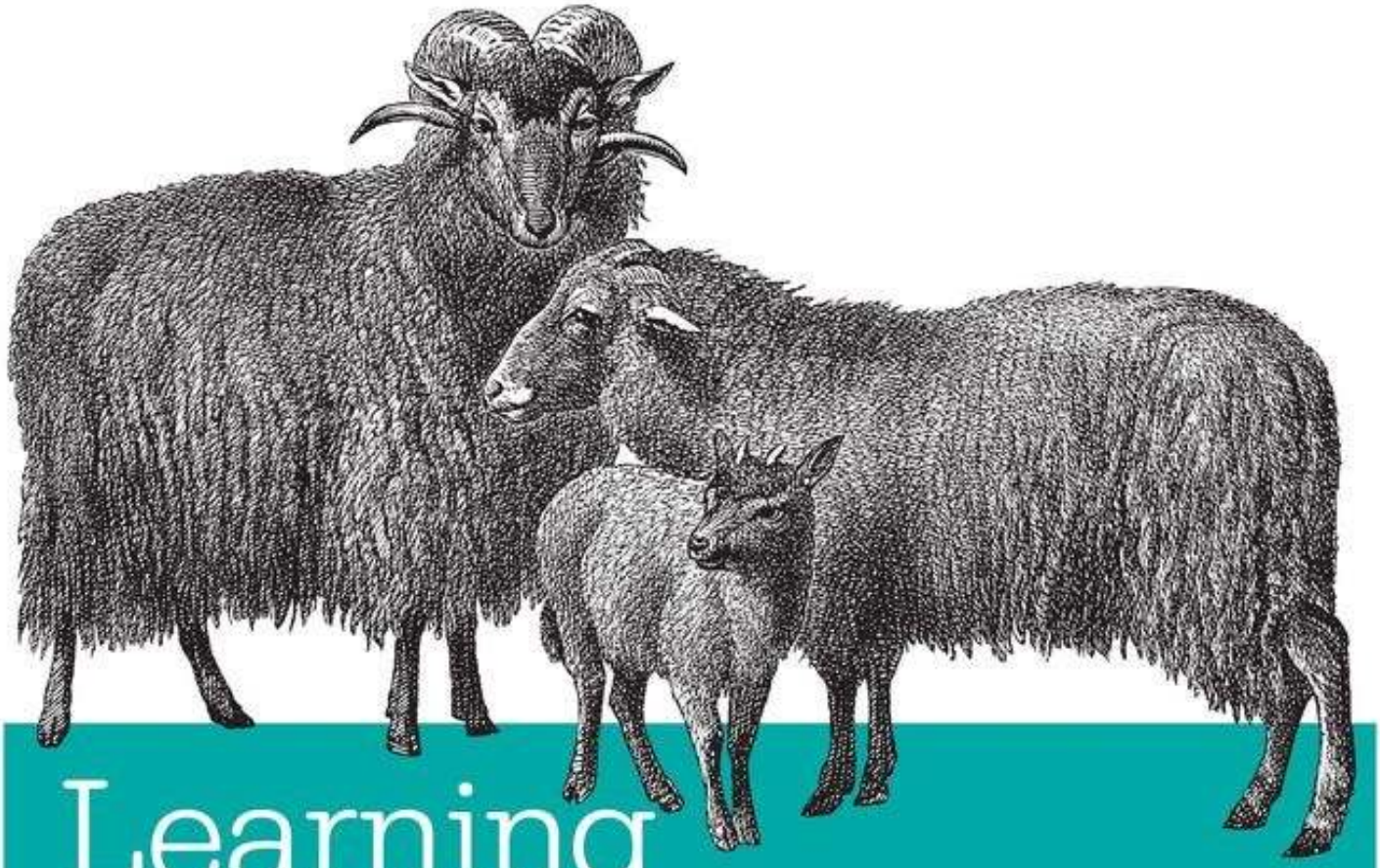
O'REILLY®



Learning Web App Development

BUILD QUICKLY WITH PROVEN JAVASCRIPT TECHNIQUES

Semmy Purewal

Learning Web Application Development

Semmy Purewal

DEDICATION

To my parents.

Thanks for all of your support and encouragement over the years!

Preface

In early 2008, after about six years of grad school and teaching part-time, I found myself hoping to land a job as a full-time computer science professor. It didn't take me long to realize that professor jobs are really hard to come by, and obtaining a good one has almost as much to do with luck as it ha to do with anything else. So I did what any self-respecting academic does when faced with a scary academic job market: I decided to make myself employable by learning how to develop web applications.

This may sound a little strange. After all, I had been studying computer science for about nine years that point, and had been teaching students how to develop software for about six years. Shouldn't I have *already known* how to build web applications? It turns out that there's a pretty large gap betwee practical, everyday software engineering and programming as taught by computer science departmen at colleges and universities. In fact, my knowledge of web development was limited to HTML and a little CSS that I had taught myself at the time.

Fortunately, I had several friends who were actively working in the web development world, and mos of them seemed to be talking about a (relatively) new framework called Ruby on Rails. It seemed lik a good place to focus my efforts. So I purchased several books on the topic and started reading online tutorials to get up to speed.

And after a couple months of really trying to get it, I nearly gave up.

Why? Because most of the books and tutorials started out with the assumption that I had already bee developing web apps for years! And even though I had a pretty solid background in computer programming, I found all of the material extremely terse and difficult to follow. For example, it turn out you can take an awful lot of computer science classes without ever coming across the Model-View-Controller design pattern, and some of the books assumed you understood that in the first chapter!

Nevertheless, I managed to learn enough about web app development to get a few consulting gigs to support me until I managed to land a professor job. And through that, I realized I enjoyed the practic aspects of the field so much that I continued consulting outside of teaching.

After a few years of doing both, I was offered the opportunity to teach my first class in Web Application Development at the University of North Carolina at Asheville. My initial inclination was to *start* with Ruby on Rails, but when I started reading the latest books and tutorials, I realized that they hadn't improved much over the years. This isn't to say that they aren't good resources for peopl who already have a background in the basics, it's just that they didn't seem suitable for the students I was teaching.

Sadly, but not surprisingly, the academic books on web development are far worse! Many of them contain outdated concepts and idioms, and don't cover the topics in a way that make platforms like Ruby on Rails more accessible. I even reviewed one book that was updated in 2011 and still used

table-based layouts and the `<font>` tag!

I didn't have much of a choice but to develop my course from scratch, creating all the material myself. I had done a little work in some consulting gigs with Node.js (server-side JavaScript) at the time, so I thought it would be interesting to try to teach a course that covered the same language on the client and server. Furthermore, I made it my goal to give the students enough background to launch into the self-study of Ruby on Rails if they decided to continue.

This book consists largely of the material that I created while I was teaching this course at UNCA. It shows how to build a basic database-backed web application from scratch using JavaScript. This includes a basic web-development workflow (using a text editor and version control), the basics of client-side technologies (HTML, CSS, jQuery, JavaScript), the basics of server-side technologies (Node.js, HTTP, Databases), the basics of cloud deployment (Cloud Foundry), and some essential good code practices (functions, MVC, DRY). Along the way we'll get to explore some of the fundamentals of the JavaScript language, how to program using arrays and objects, and the mental models that come along with this type of software development.

Technology Choices

For version control, I picked Git, because—well—it's Git and it's awesome. Plus, it gave my student the opportunity to learn to use GitHub, which is becoming immensely popular. Although I don't cove GitHub in this book, it's pretty easy to pick up once you get the hang of Git.

I decided to use jQuery on the client because it's still relatively popular and I enjoy working with it. didn't use any other frameworks on the client, although I do mention Twitter Bootstrap and Zurb Foundation in Chapter 3. I chose to stay away from modern client-side frameworks like Backbone or Ember, because I think they are confusing for people who are just starting out. Like Rails, however, you should be able to easily dive into them after reading this book.

On the server-side, I chose Express because it's (relatively) lightweight and unopinionated. I decided against covering client- and server-side templating, because I think it's essential to learn to do things by hand first.

I decided against relational databases because it seemed like I couldn't give a meaningful overview o the topic in the time I allotted to that aspect of the course. Instead, I chose MongoDB because it is widely used in the Node.js community and uses JavaScript as a query language. I also just happen to really like Redis so I provided coverage of that as well.

I selected Cloud Foundry as the deployment platform because it was the only one of the three that I considered (including Heroku and Nodejitsu) that offered a free trial and didn't require a credit card set up external services. That said, the differences between the platforms aren't huge, and going from one to another shouldn't be too hard.

Is This Book for You?

This book is not designed to make you a “ninja” or a “rock star” or even a particularly good compute programmer. It won’t prepare you for immediate employment, nor can I promise that it will show yo “the right way” to do things.

On the other hand, it will give you a solid foundation in the essential topics that you’ll need in order understand how the pieces of a modern web app fit together, and it will provide a launching point to further study on the topic. If you work your way through this book, you’ll know everything that I wis I had known when I was first starting out with Rails.

You’ll get the most out of this book if you have a little experience programming and no previous experience with web development. At minimum, you probably should have seen basic programming constructs like `if-else` statements, loops, variables, and data types. That said, I won’t assume that you have any experience with object-oriented programming, nor any particular programming language. You can obtain the necessary background by following tutorials on Khan Academy or Code Academy, or by taking a programming course at your local community college.

In addition to being used for self-study, I hope that this book can serve as a textbook for community classes in web application development, or perhaps a one-semester (14-week) college-level course.

Learning with This Book

Developing web applications is definitely a skill that you’ll need to learn by doing. With that in mind I’ve written this book to be read actively. What this means is that you’ll get the most out of it if you’re sitting at a computer while reading it, and if you actually type in all the examples.

Of course, this particular approach is fraught with peril—there is a danger that the code examples wi not work if you don’t type them exactly as they appear. To alleviate that risk, I’ve created a GitHub repository with all of the examples in this book in working order. You can view them on the Web at http://www.github.com/semmypurewal/LearningWebAppDev. Because the full examples live there, I try to avoid redundantly including full code listings throughout.

In addition, I leave big portions of the projects open-ended. When I do that, it’s because I want you t try to finish them on your own. I encourage you to do that before looking at the full examples I’ve posted online. Every chapter concludes with a set of practice problems and pointers to more information, so I encourage you to complete those as well.

Teaching with This Book

When I teach this material in a 14-week class, I usually spend about 2–3 weeks on the material in the first three chapters, and 3–4 weeks on the material in the last three. That means I spend the majority the time on the middle three chapters, which cover JavaScript programming, jQuery, AJAX, and Node.js. The students that I teach seem to struggle the most with arrays and objects, so I spend extra time on those because I think they are so essential to computer programming in general.

I definitely cover things in a more *computer-sciency* way than most books on this topic, so it might b

a good fit for a course in computer science programs. Specifically, I cover mental models such as trees and hierarchical systems, and I try to emphasize functional programming approaches where the make sense (although I try not to draw attention to this in the narrative). If you find yourself teaching in a computer science program, you might choose to focus more clearly on these aspects of the material.

I currently have no plans to post solutions to the practice problems (although that may change if I ge a lot of requests), so you can feel comfortable assigning them as homework and out-of-class projects

Where to Go for Help

As mentioned before, there is a GitHub repository with all of the code samples contained in this book In addition, you can check out http://learningwebappdev.com for errata and other updates as they are necessary.

I also try to stay pretty accessible and would be glad to help if you need it. Feel free to tweet at me (*@semmypurewal*) with quick questions/comments, or email me any time (me@semmy.me) with longer questions. I also encourage you to use the "issues" feature of our GitHub repository to ask questions. I'll do my best to respond as quickly as I can.

General Comments on Code

I've done my best to stay idiomatic and clear wherever possible. That said, those two goals are sometimes in conflict with each other. Therefore, there are times when I didn't do things "the right way" for pedagogical reasons. I hope that those places are self-evident to experienced developers, an that they don't cause any grief for novice developers in the long run.

All of the code should work fine in modern web browsers, and I've tested everything in Chrome. Obviously, I can't guarantee things will work in older versions of Internet Explorer. Please let me know if you find any browser compatibility issues in the Internet Explorer 10+ or modern versions of any other browser.

For the most part, I've followed idiomatic JavaScript, but there are a few places I've strayed. For example, I preferred double quotes instead of single quotes for strings, primarily because I've been working under the assumption that students may be coming from a Java/C++ background. I choose to use quotes around property names in object literals so that JSON doesn't look too different from JavaScript objects. I also use `$` as the first character in variables that are pointing to jQuery objects. I find that it maintains clarity and makes the code a little more readable for novices.

Conventions Used in This Book

The following typographical conventions are used in this book:

Italic
: Indicates new terms, URLs, email addresses, filenames, and file extensions.

`Constant width`
: Used for program listings, as well as within paragraphs to refer to program elements such as variable or function names, databases, data types, environment variables, statements, and keywords.

`Constant width bold`
: Shows commands or other text that should be typed literally by the user.

`Constant width italic`
: Shows text that should be replaced with user-supplied values or by values determined by context.

> **TIP**
>
> This element signifies a tip or suggestion.

> **NOTE**
>
> This element signifies a general note.

> **WARNING**
>
> This element indicates a warning or caution.

Using Code Examples

Supplemental material (code examples, exercises, etc.) is available for download at http://www.github.com/semmypurewal/LearningWebAppDev.

This book is here to help you get your job done. In general, if example code is offered with this book you may use it in your programs and documentation. You do not need to contact us for permission unless you're reproducing a significant portion of the code. For example, writing a program that uses several chunks of code from this book does not require permission. Selling or distributing a CD-ROM of examples from O'Reilly books does require permission. Answering a question by citing this book and quoting example code does not require permission. Incorporating a significant amount of example code from this book into your product's documentation does require permission.

We appreciate, but do not require, attribution. An attribution usually includes the title, author, publisher, and ISBN. For example: "*Learning Web App Development* by Semmy Purewal (O'Reilly). Copyright 2014 Semmy Purewal, 978-1-449-37019-0."

If you feel your use of code examples falls outside fair use or the permission given above, feel free to contact us at permissions@oreilly.com.

Safari® Books Online

Safari Books Online is an on-demand digital library that delivers expert content in both book and video form from the world's leading authors in technology and business.

Technology professionals, software developers, web designers, and business and creative professionals use Safari Books Online as their primary resource for research, problem solving, learning, and certification training.

Safari Books Online offers a range of product mixes and pricing programs for organizations, government agencies, and individuals. Subscribers have access to thousands of books, training videos and prepublication manuscripts in one fully searchable database from publishers like O'Reilly Media Prentice Hall Professional, Addison-Wesley Professional, Microsoft Press, Sams, Que, Peachpit Pres Focal Press, Cisco Press, John Wiley & Sons, Syngress, Morgan Kaufmann, IBM Redbooks, Packt, Adobe Press, FT Press, Apress, Manning, New Riders, McGraw-Hill, Jones & Bartlett, Course Technology, and dozens more. For more information about Safari Books Online, please visit us onlin

How to Contact Us

Please address comments and questions concerning this book to the publisher:

O'Reilly Media, Inc.

1005 Gravenstein Highway North

Sebastopol, CA 95472

800-998-9938 (in the United States or Canada)

707-829-0515 (international or local)

707-829-0104 (fax)

We have a web page for this book, where we list errata, examples, and any additional information. You can access this page at http://oreil.ly/learning-web-app.

To comment or ask technical questions about this book, send email to bookquestions@oreilly.com.

For more information about our books, courses, conferences, and news, see our website at http://www.oreilly.com.

Find us on Facebook: http://facebook.com/oreilly

Follow us on Twitter: http://twitter.com/oreillymedia

Watch us on YouTube: http://www.youtube.com/oreillymedia

Acknowledgments

Thanks to the nice folks in the Computer Science department at UNC Asheville for letting me teach this class twice. And, of course, thanks to the students who took the class for being patient with me a this material evolved.

Thanks to my editor Meg Blanchette for doing her best to keep me on track and—of course—her

constant patience with missed deadlines. I'm going to miss our weekly email exchanges!

Thanks to Simon St. Laurent for offering lots of advice early on and helping me get the idea approve by O'Reilly.

Sylvan Kavanaugh and Mark Philips both did a very careful reading of every chapter and gave lots of very helpful feedback along the way. Emily Watson read the first four chapters and gave lots of thoughtful suggestions for improvements. Mike Wilson read the last four chapters and gave invaluab technical advice. I owe you all a debt of gratitude and hope I can repay the favor one day.

Bob Benites, Will Blasko, David Brown, Rebekah David, Andrea Fey, Eric Haughee, Bruce Hauman, John Maxwell, Susan Reiser, Ben Rosen, and Val Scarlata read various revisions of the material and provided helpful suggestions. I sincerely appreciate the time and effort they put in. You rock!

Despite the all-around-excellence of the reviewers and friends who looked at the material, it's nearly impossible to write a book like this without some technical errors, typos, and bad practices slipping through the cracks. I take full responsibility for all of them.

Errata Reporters

Several readers were kind enough to report errors in the first printing of the book. Thanks to all of th people who took the time to do so: Olusola Akapo, Micheal Beatty, David Boles, Matthew Brockway Dan Candela, Gilbert Desport, Douglas Eichelberger, Stephen Fickas, James FitzGibbon, Michael Hennessy, Ken Hommel, Nick Litwin, Daniel Overton, Michael Rasmussen, and Marco Vaccari.

Chapter 1. The Workflow

Creating web applications is a complicated task involving lots of moving parts and interacting components. In order to learn how to do it, we have to break down these parts into manageable chunk and try to understand how they all fit together. Surprisingly, it turns out that the component we interact with most often doesn't even involve code!

In this chapter, we'll explore the web application development workflow, which is the process that w use to build our applications. In doing so, we'll learn the basics of some of the tools that make it a manageable and (mostly) painless process.

These tools include a text editor, a version control system, and a web browser. We won't study any o these in depth, but we'll learn enough to get us started with client-side web programming. In Chapter 2, we'll actually see this workflow in action as we're studying HTML.

If you're familiar with these tools, you may want to scan the summary and the exercises at the end of the chapter and then move on.

Text Editors

The tool that you'll interact with most often is your text editor. This essential, and sometimes overlooked, piece of technology is really the most important tool in your toolbox, because it is the program that you use to interact with your code. Because your code forms the concrete building bloc of your application, it's really important that creating and modifying it is as easy as possible. In addition, you'll usually be editing several files simultaneously, so it's important that your text editor provide the ability to quickly navigate your filesystem.

In the past, you may have spent a good deal of time writing papers or editing text documents with programs like Microsoft Word or Google Docs. These are not the types of editors that we're talking about. These editors focus more on formatting text than making it easy to edit text. The text editor th we'll use has very few features that allow us to format text, but has an abundance of features that hel us efficiently manipulate it.

At the other end of the spectrum are Integrated Development Environments (IDEs) like Eclipse, Visual Studio, and XCode. These products usually have features that make it easy to manipulate code but also have features that are important in enterprise software development. We won't have the occasion to use any of those features in this book, so we're going to keep it simple.

So what kinds of text editors should we explore? Two primary categories of text editors are common used in modern web application development. The first are Graphical User Interface (GUI) editors. Because I'm assuming that you have some background in programming and computing, you've most likely experienced a Desktop GUI environment. Therefore, these editors should be relatively comfortable for you. They respond well to the mouse as an input device and they have familiar menu

that allow you to interact with your filesystem as you would any other program. Examples of GUI tex editors include TextMate, Sublime Text, and Coda.

The other category of text editors are terminal editors. These editors were designed before GUIs or mice even existed, so learning them can be challenging for people who are used to interacting with a computer via a GUI and a mouse. On the other hand, these editors can be much more efficient if you're willing to take the time to learn one of them. The most commonly used editors that fall into this category are Emacs (shown in Figure 1-1) and Vim (shown in Figure 1-2).

In this book, we'll focus on using a GUI text editor called Sublime Text, but I encourage everyone to get some experience in either Emacs or Vim. If you continue on your web application development journey, it's highly likely you'll meet another developer who uses one of these editors.

Installing Sublime Text

Sublime Text (or Sublime, for short) is a popular text editor with several features that make it great for web development. In addition, it has the advantage that it's cross-platform, which means it should work roughly the same whether you're using Windows, Linux, or Mac OS. It's not free, but you can download an evaluation copy for free and use it for as long as you like. If you do like the editor and find that you're using it a lot, I encourage you to purchase a license.

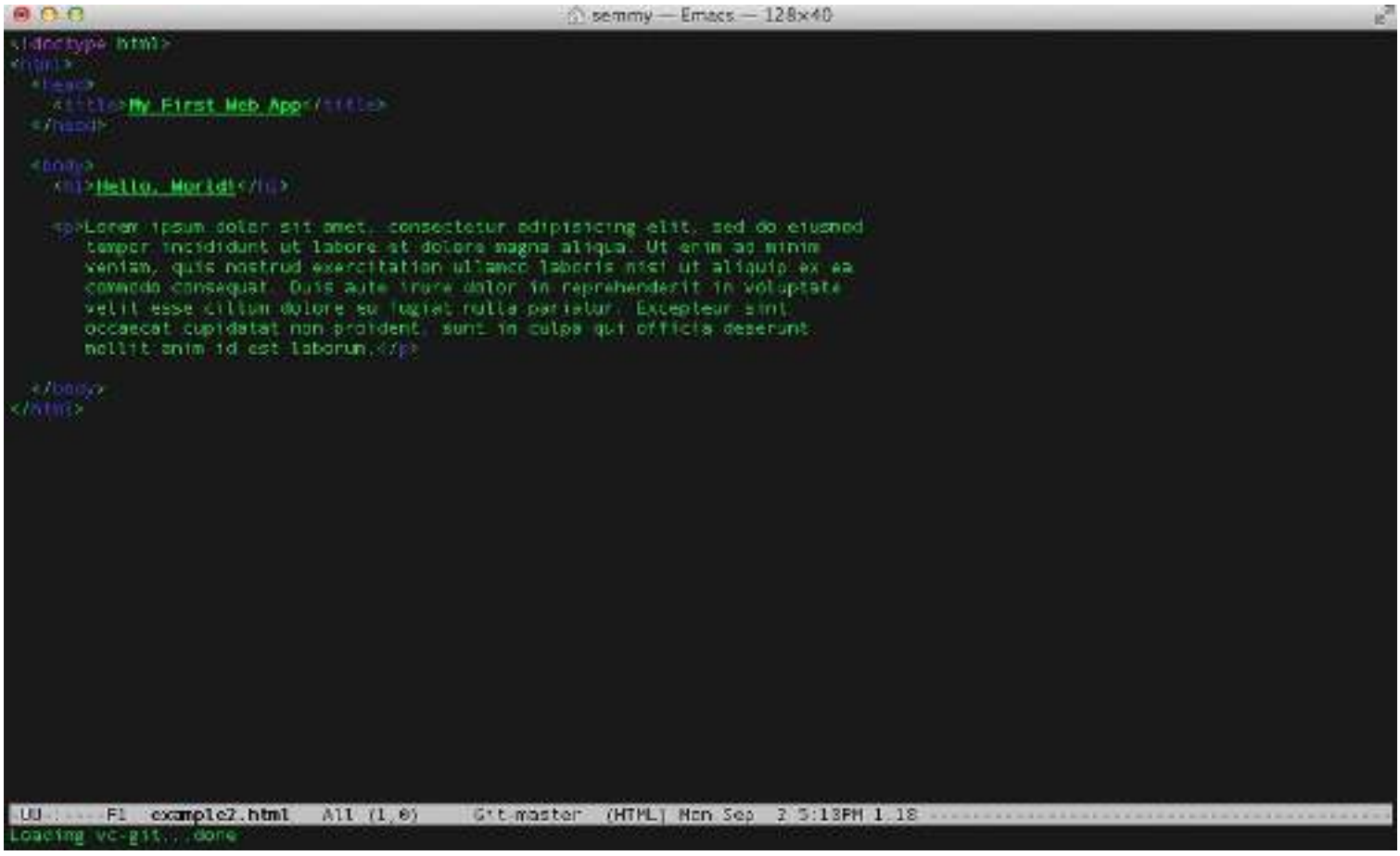


Figure 1-1. An HTML document opened in Emacs

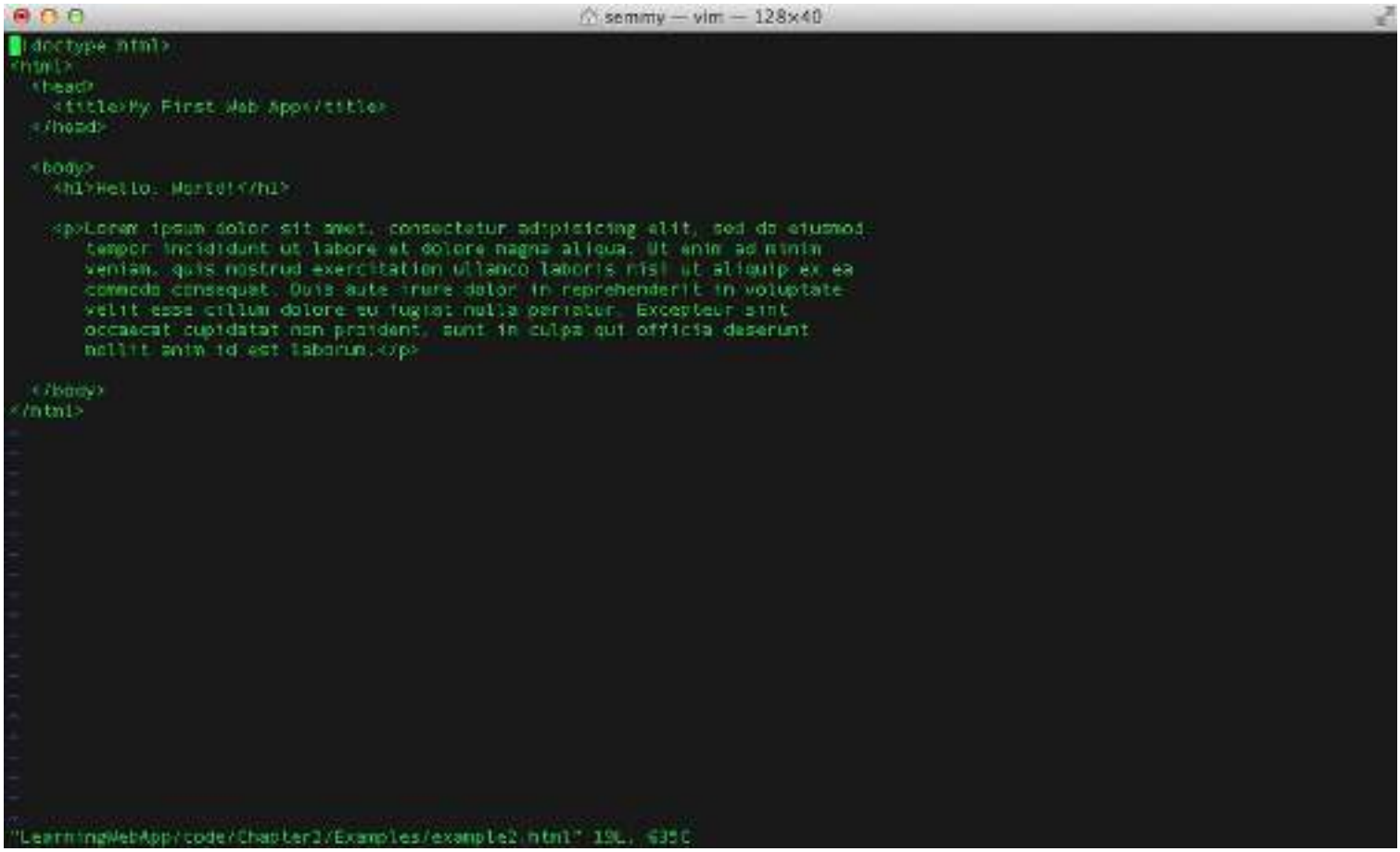


Figure 1-2. An HTML document opened in Vim

To install Sublime, visit http://www.sublimetext.com and click the Download link at the top. There you'll find installers for all major platforms. Even though Sublime Text 3 is in beta testing (at the time of this writing), I encourage you to give it a try. I used it for all the examples and screenshots in this book.

Sublime Text Basics

Once you have Sublime installed and run it, you'll be presented with a screen that looks something like Figure 1-3.

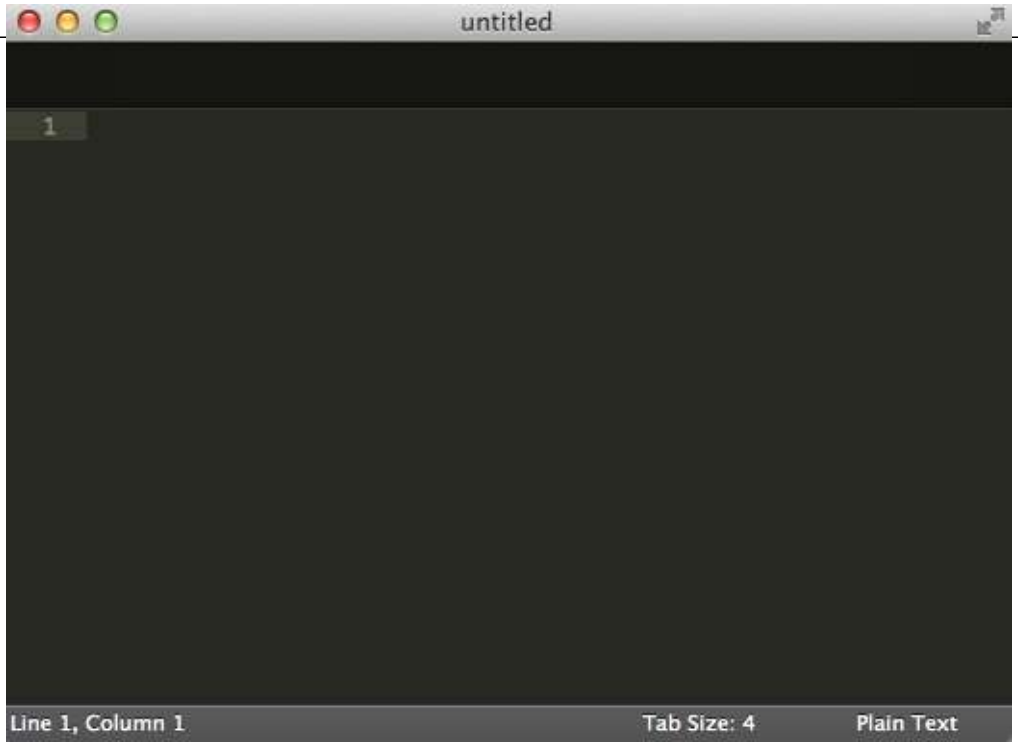


Figure 1-3. Sublime Text, after being opened for the first time

The first thing you'll want to do is create a new file. Do that by going to the File menu and clicking New. You can also do that by typing Ctrl-N in Windows and Linux or using Command-N in Mac OS. Now type **`Hello World!`** into the editor. The editor will look similar to Figure 1-4.

You can change the appearance of the Sublime environment by going to the Sublime Text menu and following Preferences → Color Scheme. Try out a few different color schemes and find one that is comfortable for your eyes. It's probably a good idea to spend some time exploring the theme options because you'll spend a lot of time looking at your text editor. Note that you can also change the font size from the Font submenu under Preferences to make the text more readable.

Figure 1-4. Sublime after a new file is opened and Hello World! is typed into the file

You probably noticed that Sublime changed the tab name from “untitled” to “Hello World!” as you typed. When you actually save, the default filename will be the text that appears in the tab name, but you’ll probably want to change it so that it doesn’t include any spaces. Once saved with a different name, the tab at the top will change to the actual filename. Notice that when you subsequently make any changes you’ll see the X on the right side of the tab change to a green circle—this means you hav unsaved changes.

After you’ve changed your theme and saved your file as *hello*, the editor will look similar to Figure 1 5.

> **WARNING**
>
> Because we’ll be working from the command line, it’s a good idea to avoid spaces or special characters in filenames. We’ll occasionally save files using the underscore (_) character instead of a space, but try not to use any other nonnumeric or nonalphabetic characters.

We’ll spend a lot of time editing code in Sublime, so we’ll obviously want to make sure we’re saving our changes from time to time. Because I expect that everyone has a little experience with code, I’ll assume that you’ve seen the edit-save-edit process before. On the other hand, there’s a related essential process that many new programmers don’t have experience with, and that’s called version control.

Figure 1-5. Sublime after the theme has been changed to Solarized (light) and the file has been saved as hello

Version Control

Imagine that you're writing a long piece of fiction with a word processor. You're periodically saving your work to avert disaster. But all of the sudden you reach a very important plot point in your story and you realize that there is a significant part of your protagonist's backstory that is missing. You decide to fill in some details, way back near the beginning of your story. So you go back to the beginning, but realize that there are two possibilities for the character. Because you don't have your story completely outlined, you decide to draft both possibilities to see where they go. So you copy your file into two places and save one as a file called *StoryA* and one as a file called *StoryB*. You draf out the two options of your story in each file.

Believe it or not, this happens with computer programs far more often than it happens with novels. In fact, as you continue on you'll find that a good portion of your coding time is spent doing something that is referred to as *exploratory coding*. This means that you're just trying to figure out what you have to do to make a particular feature work the way it's supposed to before you actually start coding it. Sometimes, the exploratory coding phase can spawn changes that span multiple lines in various code files of your application. Even beginning programmers will realize this sooner rather than later, and they will often implement a solution similar to the one just described. For example, beginners might copy their current code directory to another directory, change the name slightly, and continue on. If they realize that they've made a mistake, they can always revert back to the previous copy.

This is a rudimentary approach to *version control*. Version control is a process that allows you to kee labeled checkpoints in your code so you can always refer back to them (or even revert back to them) it becomes necessary. In addition to that, version control is an essential tool for collaborating with other developers. We won't emphasize that as often in this book, but it's a good idea to keep it in

mind.

Many professional version control tools are available and they all have their own set of features and nuances. Some common examples include Subversion, Mercurial, Perforce, and CVS. In the web development community, however, the most popular version control system is called Git.

Installing Git

Git has straightforward installers in both Mac OS and Windows. For Windows, we'll use the msysgit project, which is available on GitHub as shown in Figure 1-6. The installers are still available on Google Code and are linked from the GitHub page. Once you download the installer, double-click it and follow the instructions to get Git on your system.

Figure 1-6. The msysgit home page

For Mac OS, I prefer using the Git OS X installer shown in Figure 1-7. You simply download the prepackaged disk image, mount it, and then double-click the installer. At the time of this writing, the installer says that it is for Mac OS Snow Leopard (10.5), but it worked fine for me on my Mountain Lion (10.8) system.

Figure 1-7. The Git for OS X home page

If you're using Linux, you can install Git through your package management system.

Unix Command-Line Basics

There are graphical user interfaces to Git, but it's much more efficient to learn to use it through the command line. Before you learn to do that, however, you'll have to learn to navigate your filesystem using some basic Unix commands.

Like I mentioned before, I am assuming you have a background in computing and programming so you've most likely interacted with a desktop GUI environment. This means that you've had to use th desktop environment to explore the files and folders stored on your machine. You typically do this through a filesystem navigator such as Finder for Mac OS or Windows Explorer in Windows.

Navigating your computer's filesystem from the command line is almost the same as navigating it using your system's file browser. There are still files, and those files are organized into folders, but we refer to folders as *directories*. You can easily accomplish all the same tasks that you can accomplish in the file browser: you can move into a directory or out of a directory, see the files that are contained in a directory, and even open and edit files if you're familiar with Emacs or Vim. The only difference is that there is no continuous visual feedback from the GUI, nor are you able to interact via a mouse.

If you're in Windows, you'll do the following in the *Git Bash* prompt that you installed with the msysgit project described in the previous section. Git Bash is a program that simulates a Unix

terminal in Windows and gives you access to Git commands. To fire up the Git Bash prompt, you'll navigate there via your Start menu. If you're running Mac OS, you'll use the Terminal program, whic you can find in the *Utilities* directory of your *Applications* folder. If you're using Linux, it depends a bit on the particular flavor you're using, but there is usually an easily available Terminal program in your applications. The default Mac OS terminal window is shown in Figure 1-8.

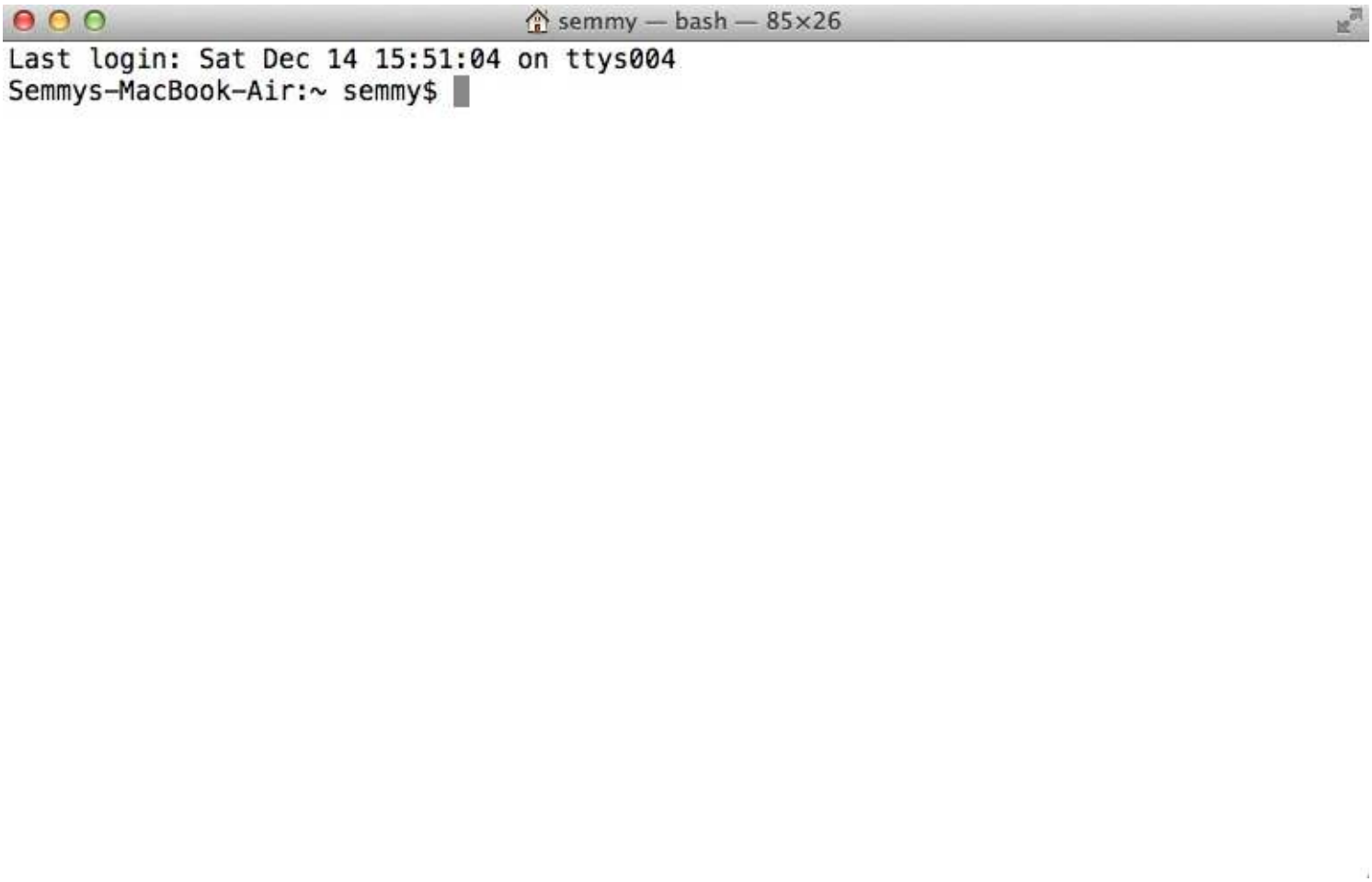


Figure 1-8. A default terminal window in Mac OS

Once you open the terminal, you'll be greeted with a command prompt. It may look different depending on whether you're using Windows or Mac OS, but it usually contains some information about your working environment. For instance, it may include your current directory, or maybe your username. In Mac OS, mine looks like this:

```
Last login: Tue May 14 15:23:59 on ttys002
hostname $ _
```

Where am I?

An important thing to keep in mind is that whenever you are at a terminal prompt, you are always in directory. The first question you should ask yourself when presented with a command-line interface i "Which directory am I in?" There are two ways to answer this question from the command line. The first way is to use the `pwd` command, which stands for *print working directory*. The output will look something like this:

```
hostname $ pwd
/Users/semmy
```

Although I do use `pwd` on occasion, I definitely prefer to use the command `ls`, which roughly translates to *list the contents of the current directory*. This gives me more visual cues about where I am. In Mac OS, the output of `ls` looks something like this:

```
hostname $ ls
Desktop    Downloads  Movies  Pictures
Documents  Library    Music
```

So `ls` is similar to opening a Finder or Explorer window in your home folder. The result of this command clues me in that I'm in my home directory because I see all of its subdirectories printed to the screen. If I don't recognize the subdirectories contained in the directory, I'll use `pwd` to get more information.

Changing directories

The next thing that you'll want to do is navigate to a different directory than the one you're currently in. If you're in a GUI file browser, you can do this by simply double-clicking the current directory.

It's not any harder from the command line; you just have to remember the name of the command. It's `cd`, which stands for *change directory*. So if you want to go into your *Documents* folder, you simply type:

```
hostname $ cd Documents
```

And now if you want to get some visual feedback on where you are, you can use `ls`:

```
hostname $ ls
Projects
```

This tells you that there's one subdirectory in your *Documents* directory, and that subdirectory is called *Projects*. Note that you may not have a *Projects* directory in your *Documents* directory unless you've previously created one. You may also see other files or directories listed if you've used your *Documents* directory to store other things in the past. Now that you've changed directories, running `pwd` will tell you your new location:

```
hostname $ pwd
/Users/semmy/Documents
```

What happens if you want to go back to your home directory? In the GUI file browser, there is typically a back button that allows you to move to a new directory. In the terminal there is no such button. But you can still use the `cd` command with a minor change: use two periods (`..`) instead of a directory name to move back one directory:

```
hostname $ cd ..
```

```
hostname $ pwd
/Users/semmy
hostname $ ls
Desktop    Downloads  Movies  Pictures
Documents  Library    Music
```

Creating directories

Finally, you'll want to make a directory to store all of your projects for this book. To do this, you'll use the `mkdir` command, which stands for *make directory*:

```
hostname $ ls
Desktop    Downloads  Movies  Pictures
Documents  Library    Music
hostname $ mkdir Projects
hostname $ ls
Desktop    Downloads  Movies  Pictures
Documents  Library    Music   Projects
hostname $ cd Projects
hostname $ ls
hostname $ pwd
/Users/semmy/Projects
```

In this interaction with the terminal, you first look at the contents of your home directory to make su you know where you are with the `ls` command. After that, you use `mkdir` to create the *Projects* directory. Then you use `ls` to confirm that the directory has been created. Next, you use `cd` to enter th *Projects* directory, and then `ls` to list the contents. Note that the directory is currently empty, so `ls` has no output. Last, but not least, you use `pwd` to confirm that you are actually in the *Projects* directory.

These four basic Unix commands are enough to get you started, but you'll learn more as we move forward. I've included a handy table at the end of this chapter that describes and summarizes them. It's a good idea to try to memorize them.

Filesystems and trees

Web development (and programming in general) is a very abstract art form. This roughly means that to do it effectively and efficiently, you'll need to improve your abstract thinking skills. A big part of thinking abstractly is being able to quickly attach mental models to new ideas and structures. And on of the best mental models that can be applied in a wide variety of situations is a tree diagram.

A tree diagram is simply a way of visualizing any kind of hierarchical structure. And because the Un filesystem is a hierarchical structure, it's a good idea to start practicing our mental visualizations on it. For example, consider a directory called *Home* that contains three other directories: *Documents*, *Pictures*, and *Music*. Inside the *Pictures* directory are five images. Inside the *Documents* directory is another directory called *Projects*.

A tree diagram for this structure might look something like Figure 1-9.

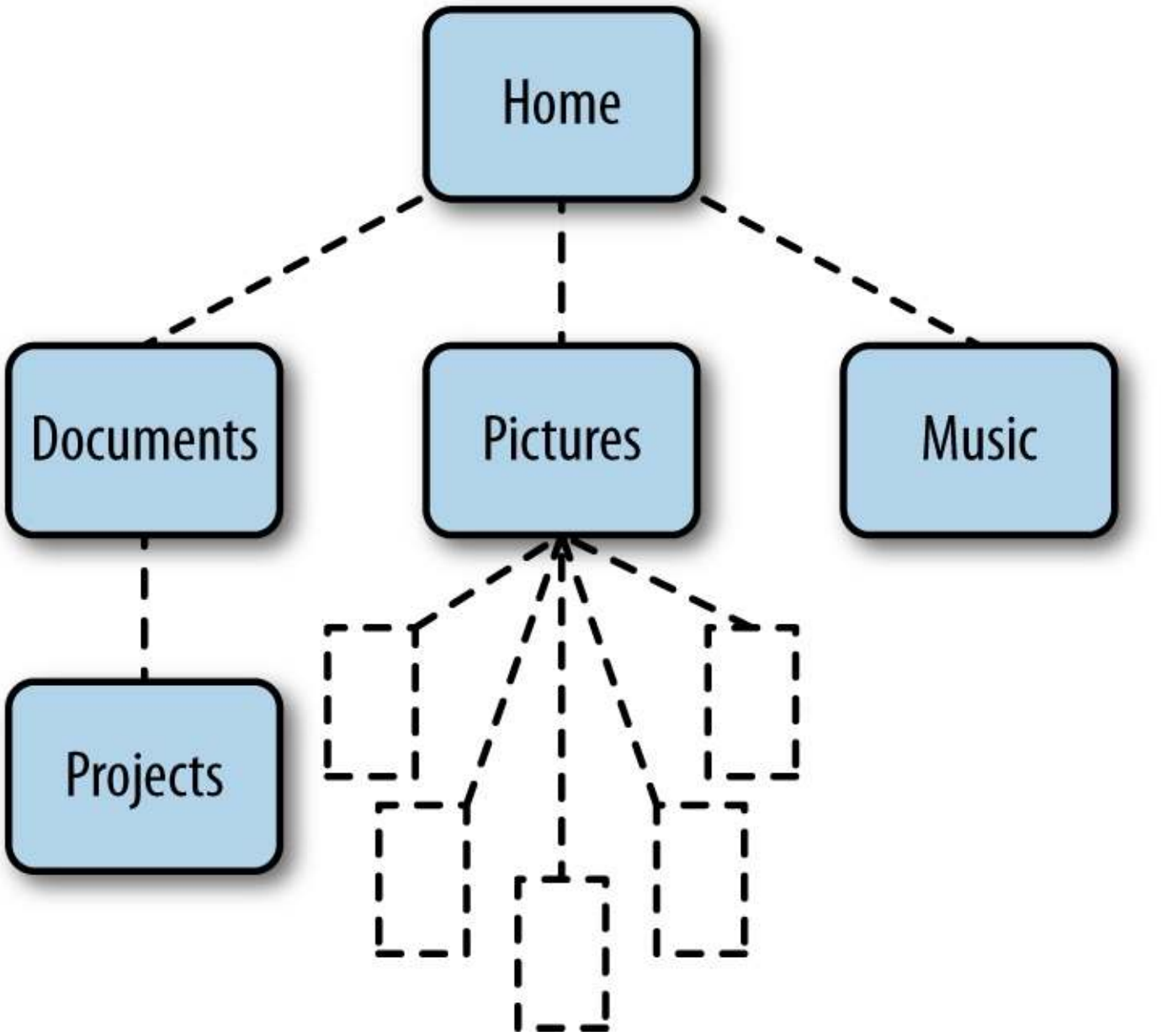


Figure 1-9. A tree diagram representing a file hierarchy

It’s a good idea to keep this mental model in your head while you’re navigating the filesystem. In fac I would recommend adding an asterisk (or something similar) that denotes your current directory and have that move as you’re moving through the filesystem.

More generally speaking, if you try to attach a tree diagram to any hierarchical structure you’ll most likely find that it’s easier to understand and analyze. Because a large part of being an effective programmer comes from the programmer’s ability to quickly build mental models, it’s a good idea to practice attaching these tree diagrams to real-world hierarchical systems whenever they make sense. We’ll do that in a few instances throughout the rest of the book.

Git Basics

Now that we can navigate the command line, we’re ready to learn how to keep our project under version control with Git.

Configuring Git for the first time

Like I mentioned before, Git is actually designed for large-scale collaboration among many programmers. Even though we're going to use it for our personal projects, it will need to be configured so that it can track our changes with some identifying information, specifically our name and email address. Open your terminal and type the following commands (changing my name and email address to yours, of course):

```
hostname $ git config --global user.name "Semmy Purewal"
hostname $ git config --global user.email "semmy@semmy.me"
```

We'll only need to do this once on our system! In other words, we don't need to do this every time w want to create a project that we're tracking with Git.

Now we're ready to start tracking a project with Git. We'll begin by navigating to our *Projects* folde if we're not already there:

```
hostname $ pwd
/Users/semmy
hostname $ cd Projects
hostname $ pwd
/Users/semmy/Projects
```

Next we'll create a directory called *Chapter1*, and we'll list the contents of the directory to confirm that it's there. Then we'll enter the directory:

```
hostname $ mkdir Chapter1
hostname $ ls
Chapter1
hostname $ cd Chapter1
hostname $ pwd
/Users/semmy/Projects/Chapter1
```

Initializing a Git repository

Now we can put the *Chapter1* directory under version control by initializing a Git repository with the `git init` command. Git will respond by telling us that it created an empty repository:

```
hostname $ pwd
/Users/semmy/Projects/Chapter1
hostname $ git init
Initialized empty Git repository in /Users/semmy/Projects/Chapter1/.git/
```

Now try typing the `ls` command again to see the files that Git has created in the directory, and you'll find there's still nothing there! That's not completely true—the *.git* directory is there, but we can't se it because files prepended by a dot (`.`) are considered hidden files. To solve this, we can use `ls` with the `-a` (all) flag turned on by typing the following:

```
hostname $ ls -a
```

```
.   ..   .git
```

This lists all of the directory contents, including the files prepended with a dot. You'll even see the current directory listed (which is a single dot) and the parent directory (which is the two dots).

If you're interested, you can list the contents of the *.git* directory and you'll see the filesystem that G prepares for you:

```
hostname $ ls .git
HEAD        config         hooks    objects
branches    description    info     refs
```

We won't have the occasion to do anything in this directory, so we can safely ignore it for now. But we will have the opportunity to interact with hidden files again, so it's helpful to remember the `-a` fl on the `ls` command.

Determining the status of our repository

Let's open Sublime Text (if it's still open from the previous section, close it and reopen it). Next, open the directory that we've put under version control. To do this, we simply select the directory in Sublime's Open dialog box instead of a specific file. When you open an entire directory, a file navigation pane will open on the left side of the editor window—it should look similar to Figure 1-1

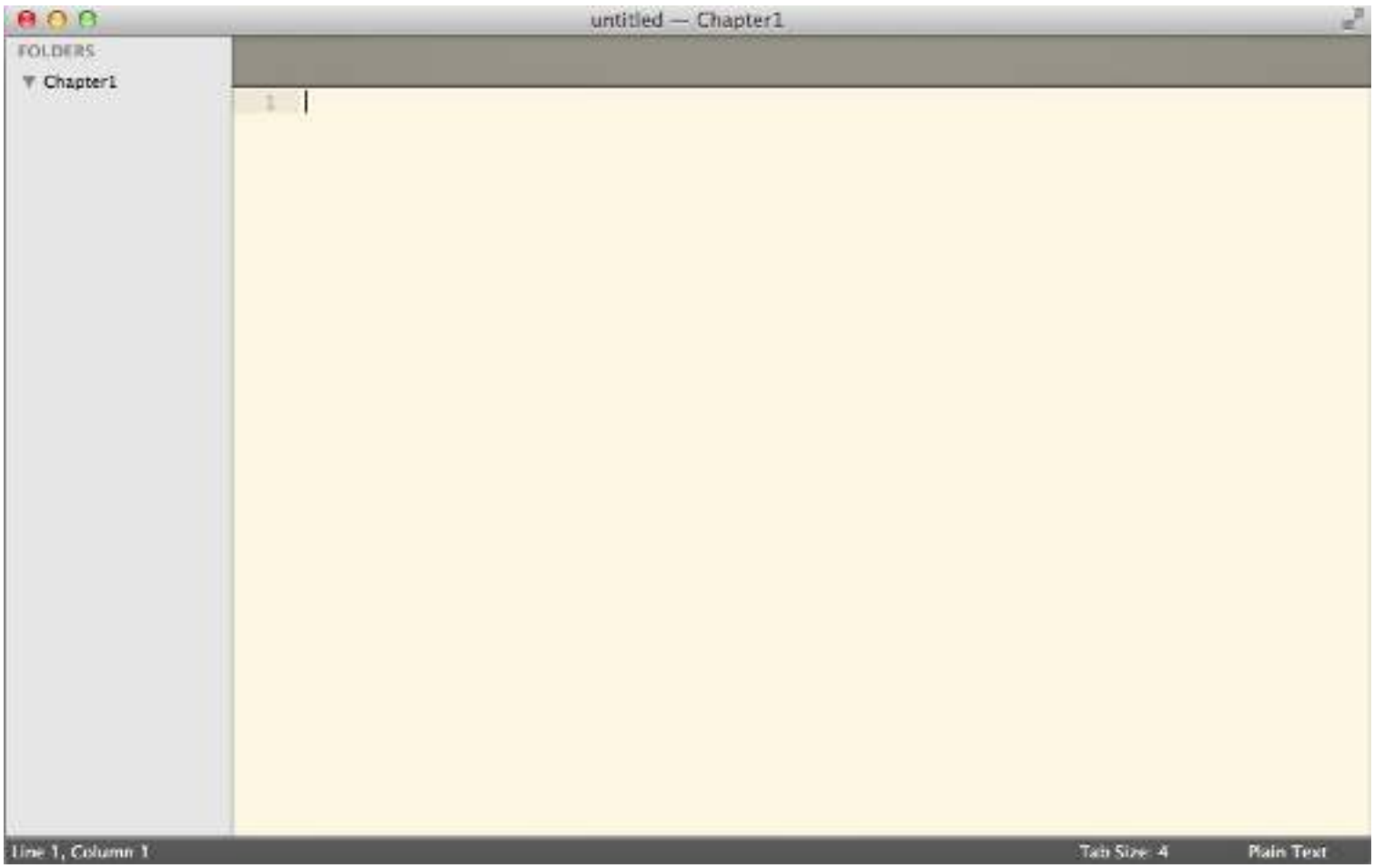


Figure 1-10. Sublime with the Chapter1 directory opened