Learn Programming through Web Development

Andrew Pikler

AHS Capstone, Spring 2012

Preface

The goal of this book is to teach you the basics of programming.

Of the many (perhaps infinite) ways in which this could be done, this text exposes you to beginning programming concepts while walking you through the making of a simple webpage. The idea is that as you're reading the chapters, you pick up programming experience, and when you're done, you will know enough about web development to immediately apply your newly learned skills.

Read on to find out more about this book and whether it is right for you.

What is programming?

In short, programming is the art of telling a computer what to do. It consists of writing **source code**, or line-by-line instructions that are precise enough for a computer to follow.

Every program that you use on your computer is simply the result of the computer executing, one line at a time, the instructions contained in the program's source code. These instructions can be as simple as outputting some text onto the screen, or as complicated as defining a large set of user interactions (as in the case of your web browser, for example).

In this book the code you write will be responsible for telling the computer to display some text and graphics on a webpage, as well as for allowing the user to interact with that webpage in certain ways.

What is web development?

Web development is a term that refers to the creation of a website. Just like programs that run on your computer—for example your web browser, any games that you might play, or your word processor—websites have code behind them that makes them work. The people who write this code are called web developers, and that's exactly what you'll be learning to do as you follow along with the text.

How much programming experience should I have before reading this book?

This text is primarily aimed at people with no programming background whatsoever. If you've never written a line of code in your life but are curious to learn what it's all about, then this book might be for you. If you have some coding experience but want to learn how to make a webpage, then the book could still work; just skim the parts that explain concepts you already know and focus on the parts that are new. On the other hand, if you've been coding for years and have also dabbled in web development, then you should probably find another reference that delves more deeply into whatever it is that you want to learn.

While no programming knowledge is required to read this book, this text does assume that you are familiar and comfortable with using a computer. This includes skills like typing, using the mouse, browsing and searching the Internet, saving and loading files, and installing and uninstalling programs.

How should I read this book?

The chapters of this text walk you through making a complete, functional webpage from scratch. In other words, at the beginning of the book, you'll have a blank page, and at the end, you'll have a working webpage. Since this one example spans multiple chapters, the best way to read the book is in order from front to back so that you can keep up with the various parts of the website as they are added. The less programming experience you have, the more sense it makes to read the text this way, since I start out with the simplest concepts in the first chapters and then move on to more complicated concepts that build on the earlier material.

However, for convenience if you should choose to skip around instead, the start of every chapter provides the complete code of the webpage as it stands at that point.

It is also essential that you read this book while sitting in front of a computer and following along on-screen with the examples (meaning that you should actively follow the directions in the examples). You'll get much more out of the chapters if you try things out for yourself than if you read them passively. Each chapter contains exercises that are meant to guide you in exploring things yourself; struggling with getting a piece of code to do what you want is likely to teach you much more than just reading what's written on the pages of this book. Also, the examples are written with good programming practices in mind—aspects of code that make it easier for other people to read—and the more time you spend with them, the more you will pick up on those practices.

What kind of learner is this book best for?

There is no course or instructor (officially, at least) that goes along with this text; you are meant to read it and follow along with it primarily by yourself. This means that you'll likely do well with the book if you prefer to work through tutorials without outside pressure or deadlines.

Much of this text is set up as a *guided lab*, which means it gives you detailed instructions on how to write a piece of code that exhibits a certain behavior, and you follow along on your computer and get the code working yourself. If you've ever followed along with tutorials of any kind (for example, Photoshop tutorials, or recipes from cookbooks) and enjoyed it, this book might be for you.

I've tried to be mindful of various learning styles throughout the book. For example, the beginning of each chapter begins with a short summary of what that chapter will cover to help those who prefer to see the big picture first before delving into details; and I've included diagrams and screenshots wherever possible to help those with a more visual learning style.

Nevertheless, this text will work best for those who learn mainly by reading and following along. If you learn primarily by having discussions with other people, or by being set loose on large, open-ended projects, this book probably won't work as well; no part of the book involves interacting with others, and much of it consists of specific instructions.

Why web development?

Many intro programming books teach programming without any context. They pick a programming language to teach, and walk you through some chapters after which you become quite good at solving the exercises they throw at you; but once you're done with the book, you have nothing to which you can apply your newly learned knowledge. In other words, there's a separation between what you've learned—how to solve contrived programming exercises—and how programming is used in the real world.

This book is an attempt at addressing this problem. By teaching web development as you're learning how to program, you are immediately exposed to a real-world application of programming. This has two main benefits: first, when you've finished the last chapter, you'll be more prepared to learn more about web development, as you've already been exposed to the basics and will know where and how to find more information. Second, what you learn will be less theoretical and more applicable, which means that you are likely to be able to use more of it right away.

What this book is not

This book is not:

- A complete reference for programming concepts;
- A complete reference for web development;
- A complete reference for anything at all.

This text makes no attempt to teach you everything there is to know about either programming or web development; in fact, some important concepts are left out in both of those areas because they are beyond the scope of the material this book aims to cover. (See the last chapter for a quick mention of those, and how you can find out more about them.) Rather, this text serves as a good starting point to both programming and web development, so that by the time you finish it you'll be well prepared to learn more about either topic.

About the exercises

Each chapter has a few exercises at the end. Some of the exercises test your grasp of the material covered in the corresponding chapter, while others allow you to explore an extension of the material that wasn't covered in detail. I highly recommend them as a way to see how you're progressing and to cover more material, but don't panic if you can't solve some—at no point are the exercises required to go on in the book. Also, I've provided solutions for them in the back of the book.

If this sounds good to you...

...then get out your computer and turn the page to Chapter 1, which walks you through setting up your machine to start making your webpage.

Chapter 1

Setting up your "Environment"

Overview

In this chapter, you'll set up your computer to allow you to start making a website. Rather than configuring your computer directly, however, you'll download and use a Virtual Machine (VM), a sort of computer within your computer, that I've made and pre-configured with everything that you'll need. I'll cover what webservers are and how your Internet browser interacts with them, and you'll use the VM to make your very first webpage.

Note: The rest of the book assumes that you are using the VM that you install in this chapter.[1] If you already have a setup that allows you to create webpages, you're welcome to use that, but you may have to do some creative thinking to figure out how some of the examples in the book translate to your setup. If you are at all in doubt, I recommend following the instructions in this chapter.

1.1 What is a webserver?

To understand why I'm going through all the steps in this chapter, first you need to understand what happens when you navigate to a webpage in your browser.

Say you want to go to Google's home page, located at the address http://www.google.

[1] While in this book I have you do all your web development in a VM, I do this mainly to make instructions easy to follow on any operating system that you might be using. Web development in the real world is not typically done on a VM; the last chapter of the book prepares you to move past a VM by helping you set up a more practical environment.

com; this address is also known as the **URL**, or **U**niversal **R**esource **L**ocator of the page. URLs always begin with `http://`. You open your favorite web (Internet) browser (such as Firefox, Google Chrome, or Internet Explorer), known as the **client** in web-speak, and type in the **URL** at the top. Here's what happens: your browser sends what is known as a **request** to a computer far away that contains all of Google's website. This computer is known as a **webserver**, or **server** for short. The server receives your request, and after some processing, sends back a **response** to your computer that contains the webpage you wanted to see (in this example, Google's home page). Your browser, upon receiving the response, displays the page for you.

Here's what this interaction looks like:

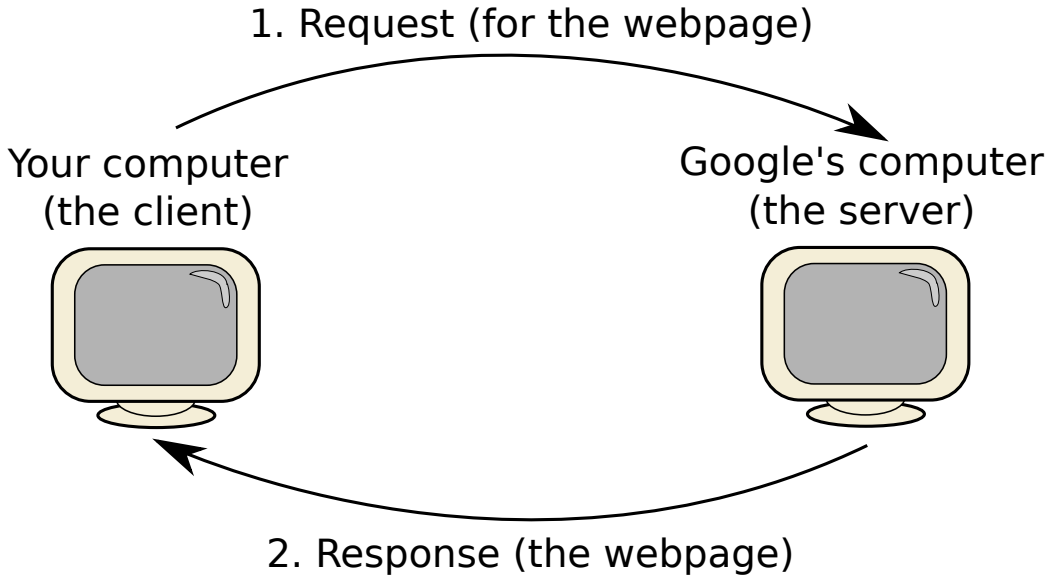

Figure 1.1: Requesting a webpage from Google's server. (Computer image from http://openclipart.org.)

This is a rather simplified version of what actually goes on in this process, but I'll revisit this graphic later in the book in more detail when those details become relevant. By the way, this entire process is known as **HTTP**—the **H**yper**t**ext **T**ransfer **P**rotocol.

The code governing the behavior of a website lives on the webserver—in the case of the above example, on Google's server, at some undetermined location away from your computer. In 1.1, this is represented by the computer on the right. In order to make a website, you'll need to edit code on such a webserver, but the problem is that you (most likely) don't have access to one.

So, how do you solve this problem? By using a modified setup where your computer will act as both the client *and* the server. It'll look something like this:

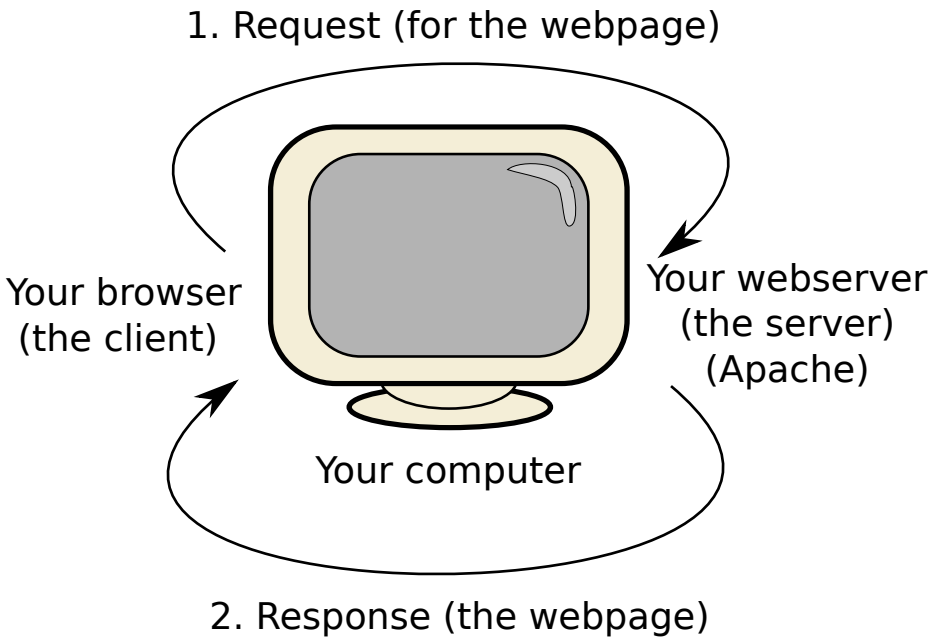

Figure 1.2: Your computer will act as both the client and the server.

The way this will work is that your computer (more specifically your VM, explained in the next section) will run a program called Apache (labeled "your webserver" in Figure 1.1) that will do everything that a remote server (such as Google's servers) usually do when you request a webpage. Once you make a page for your site, you'll enter a special URL into your browser, `http://localhost`, which will tell your browser to request the page from Apache running on your very same computer. Apache will receive the request and will send back your page to your browser, which will, in turn, display it for you.

Setting up all of this on your computer is not easy, though, and the way you do it varies wildly depending on what operating system (Windows, Mac, or Linux) you're using. So instead of setting this up yourself, you'll use a Virtual Machine I've made that comes ready to use out of the box with all of this set up for you. The webserver and the browser you will use will both be inside the VM.

1.2 What is a Virtual Machine?

A **Virtual Machine**, or **VM** for short, is essentially an entire "fake" computer that runs on your machine like any other program. To start a VM, you simply launch it, at which point you'll see a window pop up that shows your VM starting up much in the same way in which your real computer boots when you power it on. Once the VM has finished starting, you can interact with it using your mouse and keyboard, but anything you do

inside it will be *completely separate* from your real computer.

This last point deserves to be repeated. A VM can run any operating system (OS, such as Windows, Mac OS, or Linux, that serves as a platform on which programs can run), independently of what OS your computer happens to be using. You can install programs inside the VM all you want, but these programs won't be available on your actual computer (outside the VM), just as the programs that you have installed on your computer won't be available inside the VM. If you so please, you can even delete some of the VM's critical system files or allow it to get infected by viruses (if the VM is running Windows), and while this will break the VM, your real machine will be unaffected. These properties make VMs ideal for exactly the sort of experimentation that you'll be doing throughout this book.

In this book, you'll be using a VM that runs Linux—more specifically, a version of Linux known as Xubuntu. I chose for you to use Xubuntu for several reasons:

- Xubuntu is free, so unlike Windows and Mac OS, you can download it without paying for it.
- Much in the same way that most personal computers in the world run Windows, most webservers in the world run a version of Linux.
- Xubuntu has relatively low system requirements (in terms of processing power and the amount of memory it needs).

In the following section, you'll be downloading and setting up a Xubuntu VM that I've made specifically for this book. Once you launch it, everything will be configured as needed for the rest of the text.

1.3 Downloading and installing VirtualBox

Since your computer doesn't come pre-configured with the ability to run VMs, you'll need to set up a program called VirtualBox to run the VM. VirtualBox is developed by Oracle Corporation, and you can download it for free from their website:

https://www.virtualbox.org/wiki/Downloads

If you're using Windows or Mac OS X, click on the appropriate download link near the top of the page (under the heading "VirtualBox Binaries"). If you're using Linux, the

installation instructions are slightly more complicated.[2]

The file you just downloaded is the VirtualBox **installer**. It will have a filename that looks something like this:

`VirtualBox-4.1.12-77245-Win.exe` on Windows, or
`VirtualBox-4.1.12-77245-OSX.dmg` on Mac OS X, or
`virtualbox-4.1_4.1.12-77245~Ubuntu~oneiric_amd64.deb` on Linux.

The numbers, which represent the version of VirtualBox that you'll be installing, might be different for you than in the above examples if you download a later version of VirtualBox than the one I used when writing this book. This is OK.

Double-click the installer to launch it. Follow the instructions and install VirtualBox like you'd install any other program on your system. If the installer offers you any choices (such as the location, on your computer, where VirtualBox will be installed), just accept the default option.

Once the installation has finished, you can delete the installer from your computer. You won't be needing it anymore.

Now, launch VirtualBox. You can find it in the default place where installed programs go on your computer—the Start menu on Windows, or the Applications folder on Mac as well as many Linux distributions. When you launch VirtualBox for the first time, it should look something like this:

[2] If you are using Linux: to download VirtualBox, click on "VirtualBox *[version]* for Linux hosts", and then select the appropriate download for your distribution. Next you'll need to find out whether your system is 32-bit or 64-bit in order to download the correct file. To do this, open a terminal window and type:

`getconf LONG_BIT`

If the number it gives you is 64, you need the AMD64 version of VirtualBox; otherwise, you need the i386 version.

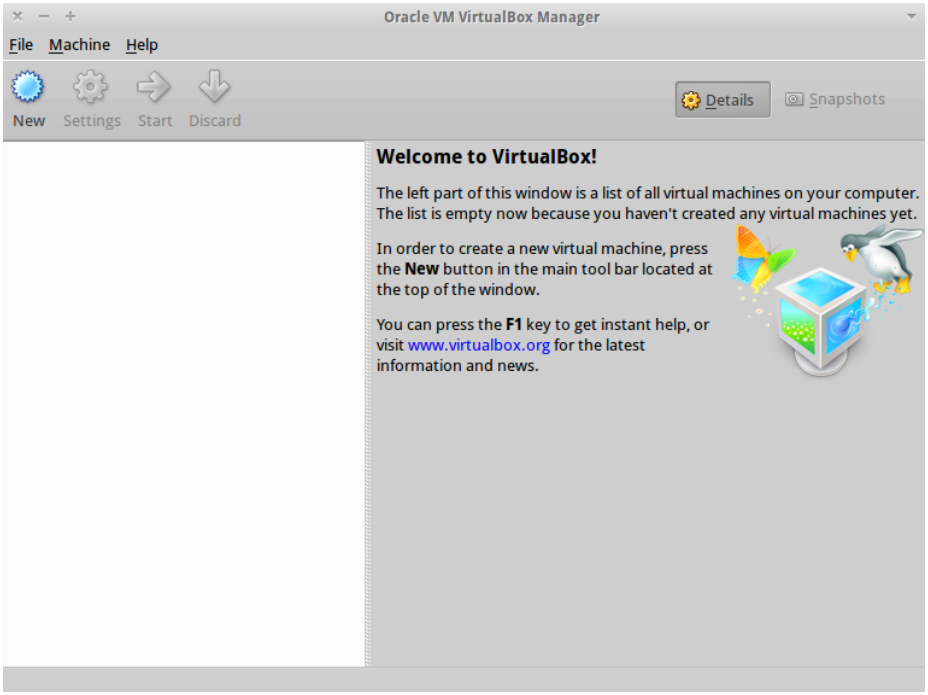

Figure 1.3: The appearance of VirtualBox when you first launch it.

At this point, you've installed VirtualBox, but you haven't actually configured it to use any VMs. That's what you'll be doing next.

1.4 Downloading and importing the VM into VirtualBox

Next you'll need to download the VM itself. You can get it here:

http://dl.dropbox.com/u/2394723/webdev/webdev.zip

Download this file. Note that it is quite large (~1.2 GB; you are downloading an entire operating system, after all) and can take more than an hour to download even with relatively fast connections.

The file you just downloaded, `webdev.zip`, is a .zip archive, meaning that it can contain other files. In this case, it contains exactly one file—`webdev.ova`, which is the VM itself—and you need to extract that from the .zip archive. On all major operating systems, you can do this by double-clicking on the .zip archive, which will open up a window that shows you its contents (which should consist of `webdev.ova` and nothing else). Then you can

click and drag `webdev.ova` out of the window. Once you've done this, and `webdev.ova` appears outside the .zip archive, you can delete the `webdev.zip`.

Now, you'll import `webdev.ova`, the VM, into VirtualBox. In the VirtualBox window, go to File ->Import Appliance..., which will open up the "Appliance Import Wizard" (VirtualBox uses the word "Appliance" in this case to mean VM):

Figure 1.4: The Appliance Import Wizard in VirtualBox.

Click on "Choose...", and find and select `webdev.ova` on your hard drive. Click Next. Now you'll get a screen with many settings—leave them as is and click Import. The import process, which converts the information packed within `webdev.ova` into a format that your computer can read more easily, will most likely take several minutes.

After successfully importing the VM, it'll appear under VirtualBox's main screen as "Xubuntu - WebDev":

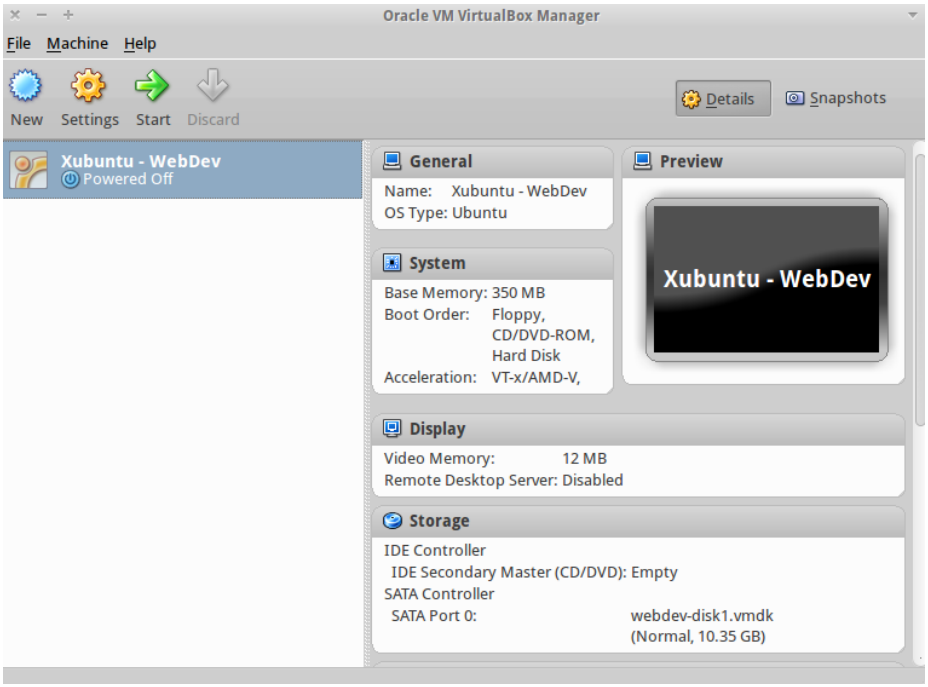

Figure 1.5: The VM shows up in VirtualBox after you import it.

Launch it by selecting it and then clicking the green Start arrow. It might help to make sure all your other programs are closed before you start the VM to ensure that you don't run into performance problems (such as your computer running out of memory, which has the effect of slowing everything down a lot).

Note that while the VM boots up for the first time, VirtualBox might display a variety of annoying notices that are similar to this one:

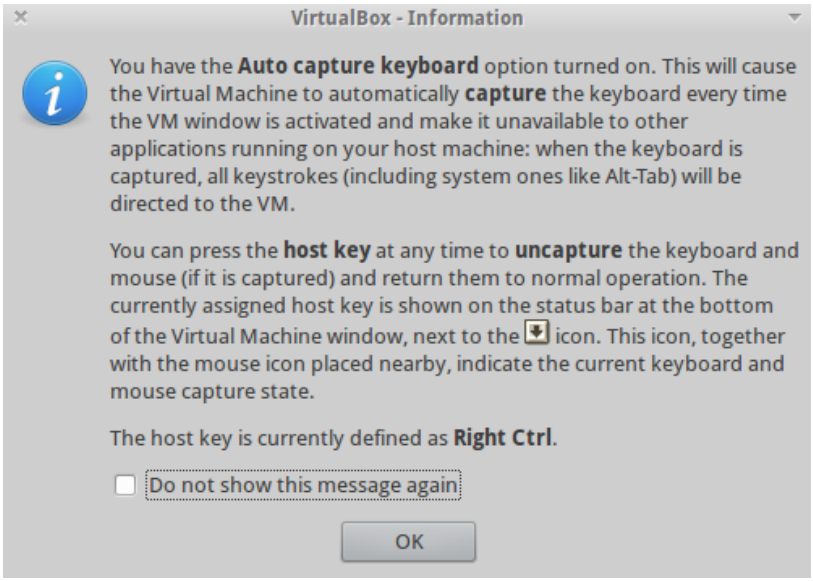

Figure 1.6: One of VirtualBox's many notice dialogs.

If you don't want to see it again, check the "Do not show this message again" box, then click OK. It's okay if you don't fully understand the contents; most boxes just alert you about VM settings that are beyond the scope of this book.

Once you get past the dialogs, you should see the VM boot up, after which you'll be presented with a login screen that looks like this:

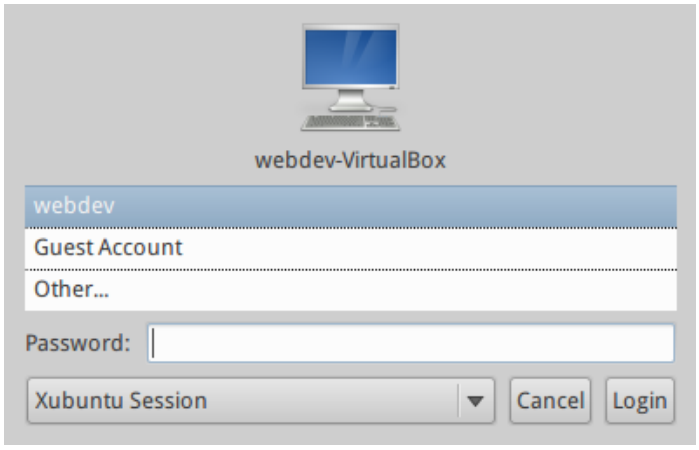

Figure 1.7: The Xubuntu login screen.

Select the "webdev" user and enter the password, which is `password` This is a mindnumbingly horrible password that you should never use for anything important. I'm simply using

it in this case because the VM is unimportant—remember, even if it is damaged, your real computer will be fine—and it is easy to remember and type.

Now, you should see the desktop:

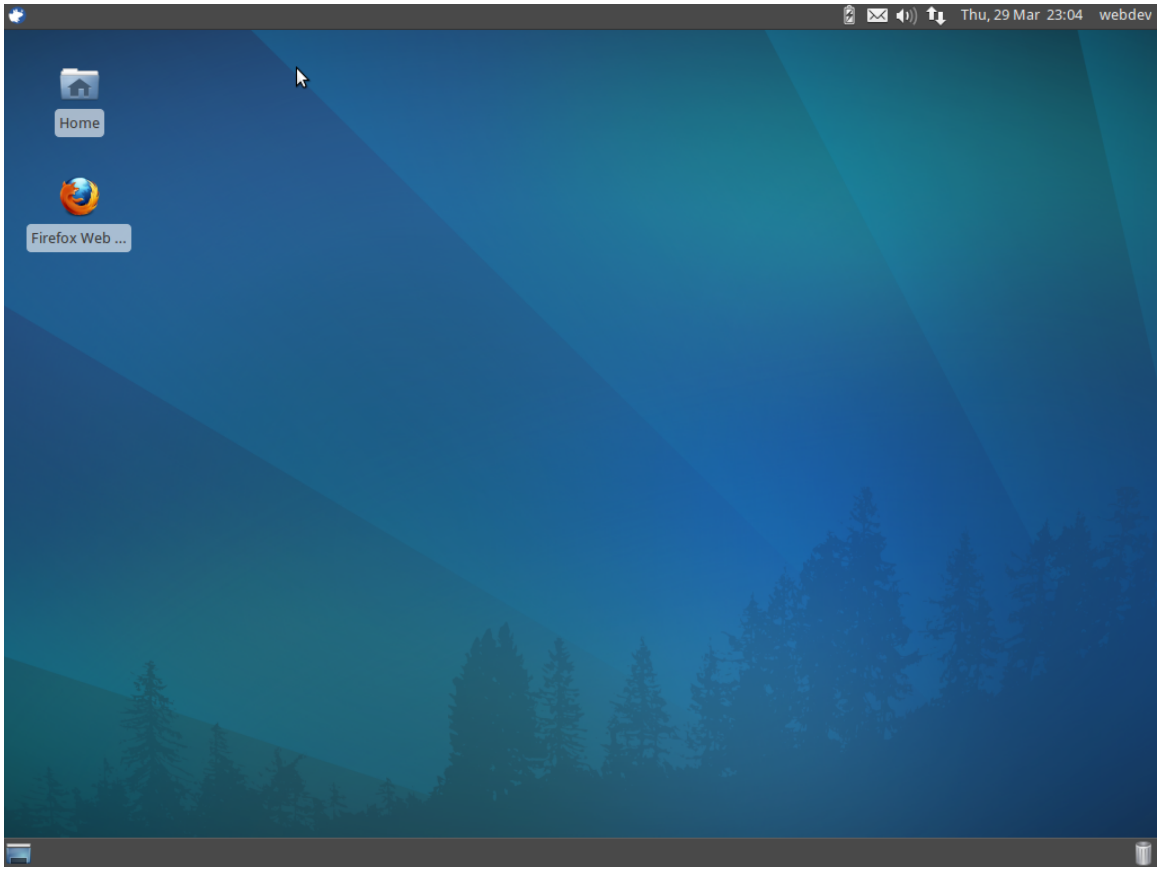

Figure 1.8: The Xubuntu desktop in the VM.

Congratulations! Everything is set up now.

To shut down the VM, select "webdev" in the upper right corner of the screen, then click "Shut down". Alternatively, select "Close..." from VirtualBox's "Machine" menu (in the upper left of the VirtualBox window).

1.5 Familiarizing yourself with the new interface

Xubuntu's interface is most likely new to you. Don't panic; it's probably not very different from the operating system that you're used to, and most simple features that you know

are likely to have analogues under Xubuntu.

Let me go over some of the basic features visible on the desktop:

- The mouse icon in the upper left corner of the screen is a drop-down menu that contains shortcuts (or **launchers**, to use the proper Linux term) to various applications on the VM. You won't be using most of these, but feel free to explore.

- The upper right corner of the screen contains controls for shutting down the VM, controlling the sound volume, as well as a display of the time.

- The Trash can be found in the lower right corner. This works exactly like the Trash under Mac OS X or the Recycle Bin under Windows; files that you delete go here until you "clear" the Trash, at which point they go away permanently.

- The lower left corner has a "Show Desktop" button, which has the effect of minimizing all open windows.

The desktop itself contains two launchers: one of them launches Firefox, the web browser. (Don't worry if you're used to a different browser; for most of the book, you'll just be using Firefox to load a single webpage, and nothing else.)

The other launcher opens the **File Manager**, a program that you can use to view all the folders and files inside the VM. Double-clicking it will get you this window:

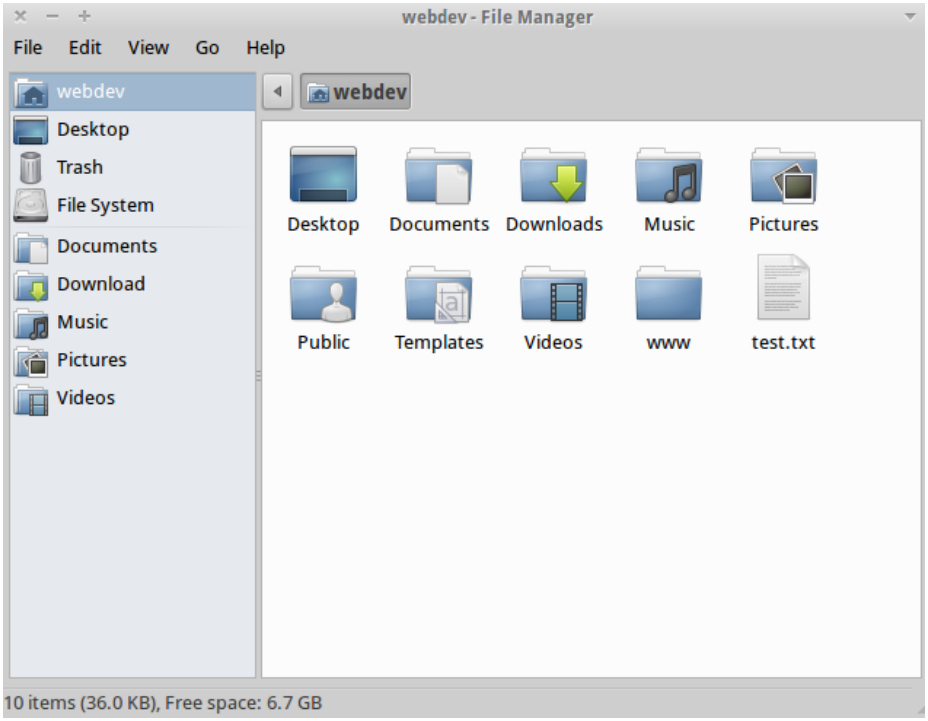

Figure 1.9: The File Manager, showing your home folder.

This is showing a folder that is known as your **home folder**. It is meant for containing your personal files, such as your pictures, documents, music, etc. Right now it contains several empty folders and a text file.

1.6 The filesystem

Now I'll cover what you need to know about the VM's filesystem in order to get started with your first webpage.

The filesystem, which is very similar to the filesystem found on Mac and Windows systems, can contain two kinds of objects: **folders** and **files**. Folders can contain other folders and files, whereas files are things like text documents, movie clips, audio recordings, and the like. Files and folders all have **names**, which identify the given file or folder inside its parent folder. No two files or folders within any one folder can have the same name.

Figure 1.5 shows your home folder, whose name is `webdev`, along with the folders it contains, which appear in bold. (`Desktop`, `Documents`, `Downloads`, `www`, etc.), as well as the only file within it (`test.txt`).

If it helps, you can think of the filesystem like this:

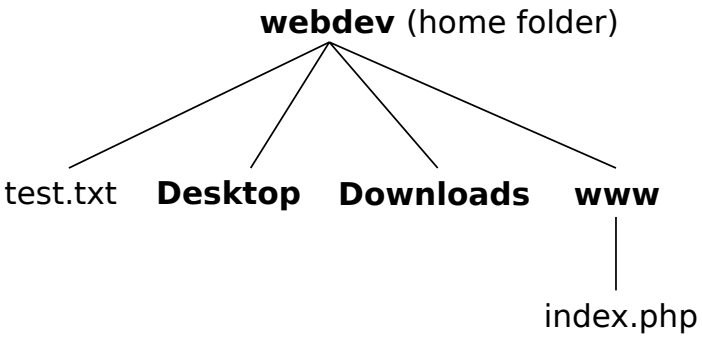

Figure 1.10: Your home folder, represented visually. Bold names are folders; `test.txt` and `index.php` are files. Note that to keep the image simple, several folders (for example, Desktop, Music, Pictures, among others) inside `webdev` are not shown.

In the above Figure 1.6, a line connects your home folder, `webdev`, to all the files and folders within it. `index.php` is a text file inside the folder called `www` that you will make in the next section of this chapter. All the work you will be doing for the website in this text will take place inside `www`.

In the File Manager, to enter a folder (i.e., "go down a level" if you are following Figure 1.6), double-click a folder icon. To go back up a level, click on the appropriate folder name near the top middle of the File Manager.

This hierarchical system extends above your home folder as well, meaning that there is another folder that contains inside it `webdev`. These folders won't be used in this book, but you're welcome to browse.

Now you know enough about the filesystem to create your first webpage.

1.7 Your first webpage

In the VM, open Firefox (using the launcher on the desktop). Navigate to the following URL: `http://localhost`. This is a special URL that brings about the actions depicted in Figure 1.1—your browser sends a request to Apache (also running in the VM), which in turn sends back your webpage to your browser.

You should now see a page titled "Index of /". This is a general placeholder page for a webpage that doesn't exist, which makes sense because you haven't made any pages yet. You will do that now.

Using the File Manager, navigate to the `www` folder in your home folder. (Double-click the "Home" icon on the desktop to open the file manager if it isn't open already, then double click the "www" icon to enter the `www` folder.) Right-click (on a Mac, if you don't have a right mouse button, click while holding the Ctrl key), and create a new document:

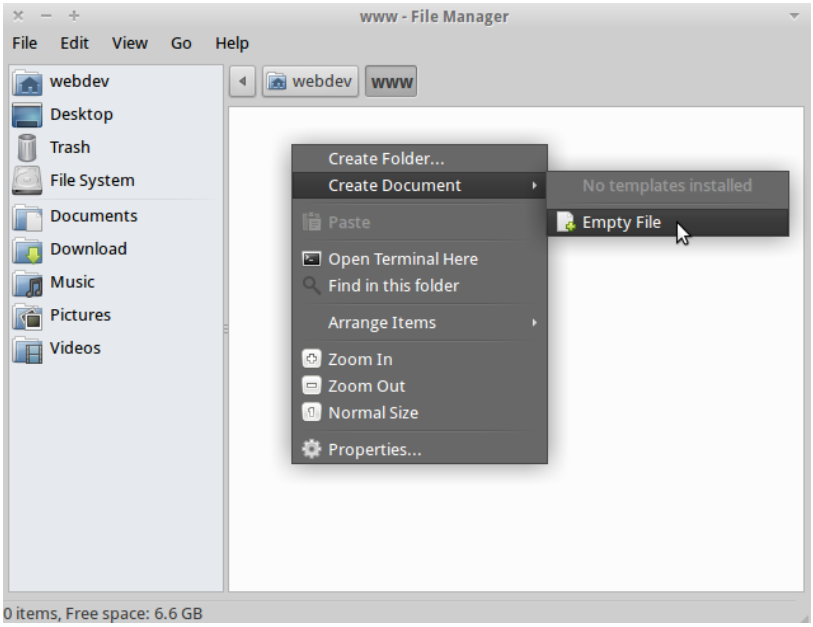

Figure 1.11: Creating an empty file.

Call the new file `index.php`. This name is important—if you pick another name, this won't work.

Right-click on the newly made file, and select "Open with gedit." `gedit` is a text editor that you'll be using to edit files throughout this book.

Once the file is open, simply enter the words "Hello World" into the empty file, then save the file (File ->Save, or Ctrl+S). You can close `gedit` if you want.

Now go back to Firefox, and refresh the page. You should see "Hello World" displayed on a blank page. Yay! It's your first webpage.

Here's how it works: the `www` folder is known as the **root** of your website. When you navigate to `http://localhost`, Apache looks in this folder for a file called `index.php`. If it finds that file, it returns the file's contents (in this case, "Hello World") to your browser, which displays the page.

In the next chapter, you'll build on this by making Apache return something more interesting than just "Hello World".

1.8 Exercises

1. In the VM, create a new empty document in your home folder. Then, using the File Manager, cut and paste it to the `Documents` folder in your home folder. Finally, delete it and clear the Trash.

2. In the root of your website (the `www` folder), create a new file called `newfile.php`. Enter something into it, save it, then go to `http://localhost/newfile.php` in Firefox. What happens? Also try going to `http://localhost/index.php`.

3. Now try going to the URL of a file that doesn't exist; for example, `http://localhost/badfile.php`.

4. If you had a file called `myfile.php` in the `www` folder, what URL would you enter into Firefox to view it? Try creating said file and then viewing it in Firefox.

Chapter 2

HTML

2.1 Overview

In the last chapter, you made a simple webpage consisting only of the words "Hello World." In this chapter, you'll expand on that by making a more substantial page that has text of different sizes and styles, as well as text input areas and buttons that users can interact with.

You'll accomplish this by learning HTML, the language web developers use to tell browsers how a webpage should look. First, you'll learn how to create various page elements in HTML—headings, paragraphs, and input forms, among others—and then you'll put these elements together into a single, coherent page. This will be a "guestbook": a page where users can leave their names and email addresses along with short comments.

2.2 What is HTML?

HTML stands for **H**yper**t**ext **M**arkup **L**anguage and, as the name implies, is a *markup language*—a way to describe the appearance of a webpage. (It is not a *programming language*, which you will meet in the next chapter on PHP, at which point the differences between markup and programming languages will become clear.)

How does HTML factor into how the Web works? Revisit Figure 1.1, this time with slightly more detail:

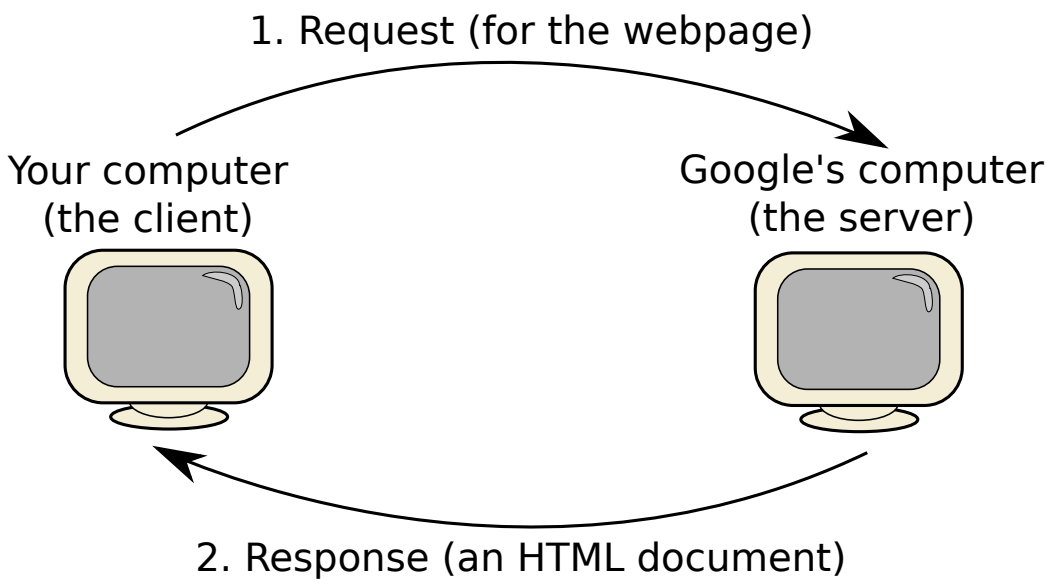

Figure 2.1: Requesting a webpage from Google's server. The response that comes back from Google is actually an HTML document.

To review, this picture depicts what happens when you go to `http://google.com` in your browser. Your browser sends a request to Google's webserver, which responds to your request.

Before, all I said about this response was that it contains information that allows your browser to display Google's home page, the page that you requested. But now I'll be more specific and say that this information comes in the form of an HTML document. This document is **plain-text**, meaning it contains only text and nothing fancy like pictures or video clips. Your browser reads the HTML and, based on the instructions it contains, displays Google's home page for you. (If the page contains images or anything besides text, the HTML will contain instructions for the browser about how to obtain those and display them on the page). In more technical terms, your browser **renders** the HTML document into a form suitable for human viewing.

Now you'll see this in action by writing a simple HTML page yourself.

2.3 A simple HTML page

In the VM, find the `index.php` file that you made in Section 1.7. It should be contained within a folder called `www`, which in turn should be inside your home folder (`webdev`). Replace its contents with the following:

```
<html>
<head>
        <title>A simple page</title>
</head>
<body>
        <h1>Hello world!</h1>
        <p>Some other text.</p>
</body>
</html>
```

Now go to http://localhost in Firefox. You should see it display the following page:

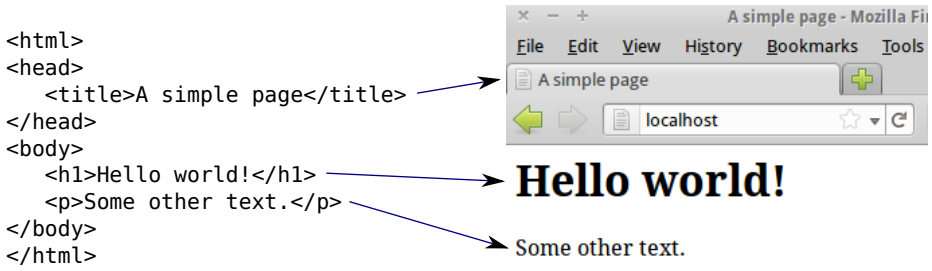

Figure 2.2: The simple page's HTML (left) rendered in Firefox (right).

Notice that the various parts of your HTML document correspond to what Firefox displays. More specifically, the text of the page is enclosed within various **tags**—these are the bits of HTML surrounded by angle brackets, such as `<head>` and `<title>`. Tags come in pairs, with an opening tag, such as `<title>`, followed by the body of the tag (in this case the title of the page, "A simple page"), and finished by a closing tag, which looks like the opening tag but with an extra slash: `</title>`. Tags have different behaviors depending on their names.

In the above example, here is a list of all the tags I used and their meanings:

- `<html></html>`: These tags enclose the entire contents of the HTML document. Every *well-formed* HTML file (a term that just means the HTML file is properly written) must start with `<html>` and end with `</html>`.
- `<head></head>`: Various information about the page (for example the page's title or information about how the page looks) goes in between these tags.
- `<title></title>`: The actual title of the page.

- <body></body>: Everything displayed in the browser window (as opposed to the title bar of the window, for example) goes between these tags.
- <h1></h1>: A heading (i.e., large text). This is part of a family of tags that goes from <h1> to <h6>; the bigger the number, the smaller the heading. For example, <h1>Hello</h1> would appear much bigger than <h6>Hello</h6>.
- : A paragraph. Suitable for a long body of text.

These are, in my experience, some of the most prevalent tags, and they show up in most HTML documents. But there are other important tags that you should be familiar with.

2.4 More HTML tags

As you explore more HTML tags, I recommend that you insert each of them into `index.php` (between <body> and </body>) and experiment with how they look in Firefox.

Images: <img />

A page that only contains text is boring, so inevitably you'll get to a point where you want to put an image somewhere. You do this using the <img /> tag. For example, if you want the Firefox logo, located at this URL:

http://www.mozilla.org/media/img/firefox/template/header-logo.png

to appear on your page, you would do it like this:

```
<img src="http://www.mozilla.org/media/img/firefox/template/header-logo.png"/>
```

There are two new things going on here. First, the way you specify the image URL is by using an **attribute** named `src`. An attribute is a way to give a tag additional information; all attributes are written like so:

```
name="value"
```

In the above case, the name of the attribute is `src`, and the value is the long URL that begins with http://www.mozilla.org. Attributes always go in the opening HTML tag; never in the closing tag or in the tag **body**, the stuff between the opening and closing tags.

Second, the `<img />` tag is *self-closing*. It doesn't have a body, so leaving space for one by writing the opening and closing tags separately (`<img></img>`) is unnecessary. Instead, you combine the opening and closing tags into one tag that ends with a slash: `/>`. There are many other self-closing tags (ones for which it doesn't make sense to have a body)—the break tag, `<br />`, coming up in the next section, for example.

By the way, at this point you've covered all of HTML's syntax: there is nothing else to HTML besides opening tags, closing tags, self-closing tags, and attributes. All the rest is learning the various tags and their corresponding attributes and recognizing when to use them.

Line breaks: `<br />`

HTML documents are not sensitive to whitespace, which means that if your document contains the following:

```
<p>I want this to be
on two separate lines!</p>
```

...it will appear all on one line in your browser.

To force it to appear on two lines, you can use `<br />`, which is another self-closing tag:

```
<p>I want this to be<br />
on two separate lines!</p>
```

Input forms: `<form>`, `<input />`, and `<textarea>`

Sometimes you want users to be able to enter information into your page; that's where forms and input fields come in.

The following produces a simple text field where a user can enter one line of text:

```
<input type="text" />
```

Note that like the `<img />` tag, `<input />` is self-closing. The `type` attribute specifies the type of input, which in this case is a line of text; `submit` is another valid type that displays a clickable button.

The following produces a larger text field that allows users enter multiple lines of text:

```
<textarea style="width:400px; height:200px"></textarea>
```

The `style` attribute can be applied to lots of different tags. You can use it to control the appearance of the elements the tags represent as they show up in the browser—for example, their size (like in this case, where you make the text area be 400 pixels tall by 200 pixels wide), their color, and other properties. The body of the `textarea` is the default text that appears in it when the page loads. In this case, it's blank.

All input fields on a page must be inside `<form>` tags. So, for example, here is a form that allows users to enter their name, along with some text:

```
<form>
Name: <input type="text" /><br />
Comment: <br />
<textarea style="width:400px; height:200px"></textarea> <br />
<input type="submit" value="Post Comment" />
</form>
```

The second `input` in this example, of type `submit`, is a button. The `value` attribute specifies the text that appears on the button; in this case, it's "Post Comment".

This entire setup will look like this in a browser:

Figure 2.3: Your input form.

Horizontal rules: `<hr />`

The `<hr />` tag simply produces a long horizontal line that's useful for separating discrete bits of content from one another—for example, multiple comments on a page (which is

exactly what you'll use this tag for later). An example that uses a horizontal rule is coming on the next page.

A container for other tags: `<div>`

The `<div>` tag is useful for grouping several other tags into one unit. One use, for example, is to restrict several tags to a certain width. For example, if you wanted to display user-generated comments on a page, you might do it like this:

```
<div style="width:400px">
<hr />
Some comment.
<hr />
Another comment.
<hr />
</div>
```

Without the `<div>`, all the elements here, including the horizontal rules, would stretch horizontally all the way across the browser window. This would make long paragraphs hard to read, and the `<hr />`s ugly. However, the `<div>` constrains everything to a width of 400 pixels.

Bold and italics: `<b>` and `<i>`

These tags are used to add styling to text. For example, in the above scenario, where you're displaying user-generated comments, you might use the `<b>` tag to emphasize the posters' names:

```
<div style="width:400px">
<hr />
<b>Name:</b> Bob <br />
Some comment.
<hr />
<b>Name:</b> Jane <br />
Another comment.
<hr />
</div>
```

In a browser, this would look like so:

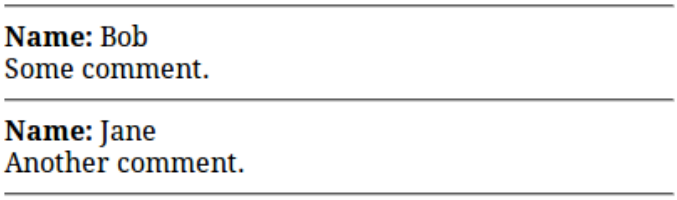

Figure 2.4: User comments.

Links: `<a>`

The <a> tag is used to make hyperlinks. For example:

```
<a href="http://google.com">Google</a>
```

The target of the link (i.e., the page where you end up if you click the link) is the value of the `href` attribute. The body of the tag is the text that gets displayed on the page. So the above would create a link that looks like this:

Google

Other tags

Of course, I've only covered a small fraction of the tags available in HTML, but these are all the tags you'll be using for a large part of this book. You can find a complete (and very intimidating!) list of tags here: http://www.w3schools.com/tags/. Don't worry, though, you'll never end up using most of those tags.

2.5 Putting it together: a guestbook

In the previous section, you created several small snippets of HTML. If you put these snippets together, you can make a page that resembles a "guestbook"—a page where users can come and post comments, and all the previously posted comments are displayed for public viewing.

First, I'll present the HTML, which is more or less the above snippets combined into one, with a few small modifications for readability. Then, I'll show how the HTML maps onto

the output you see in the browser, like I did in Figure 2.3. Without further ado, here is the HTML code of the guestbook page. Replace the contents of `index.php` in your `www` folder with this:

```
<html>
<head>
    <title>Guestbook</title>
</head>
<body>
    <h2>Guestbook</h2>
    <p>Welcome! Please post a comment using the form below.</p>
    <form>
        Name: <input type="text" /><br />
        Comment: <br />
        <textarea style="width:400px; height:200px"></textarea> <br />
        <input type="submit" value="Post Comment" />
    </form>

    <p>There have been <b>2</b> comments posted:</p>

    <div style="width:400px">
        <hr />
            <b>Name:</b> Bob <br />
            Some comment.
        <hr />
            <b>Name:</b> Jane <br />
            Another comment.
        <hr />
    </div>
</body>
</html>
```

And here are the mappings between the HTML and the browser's rendering of the various elements:

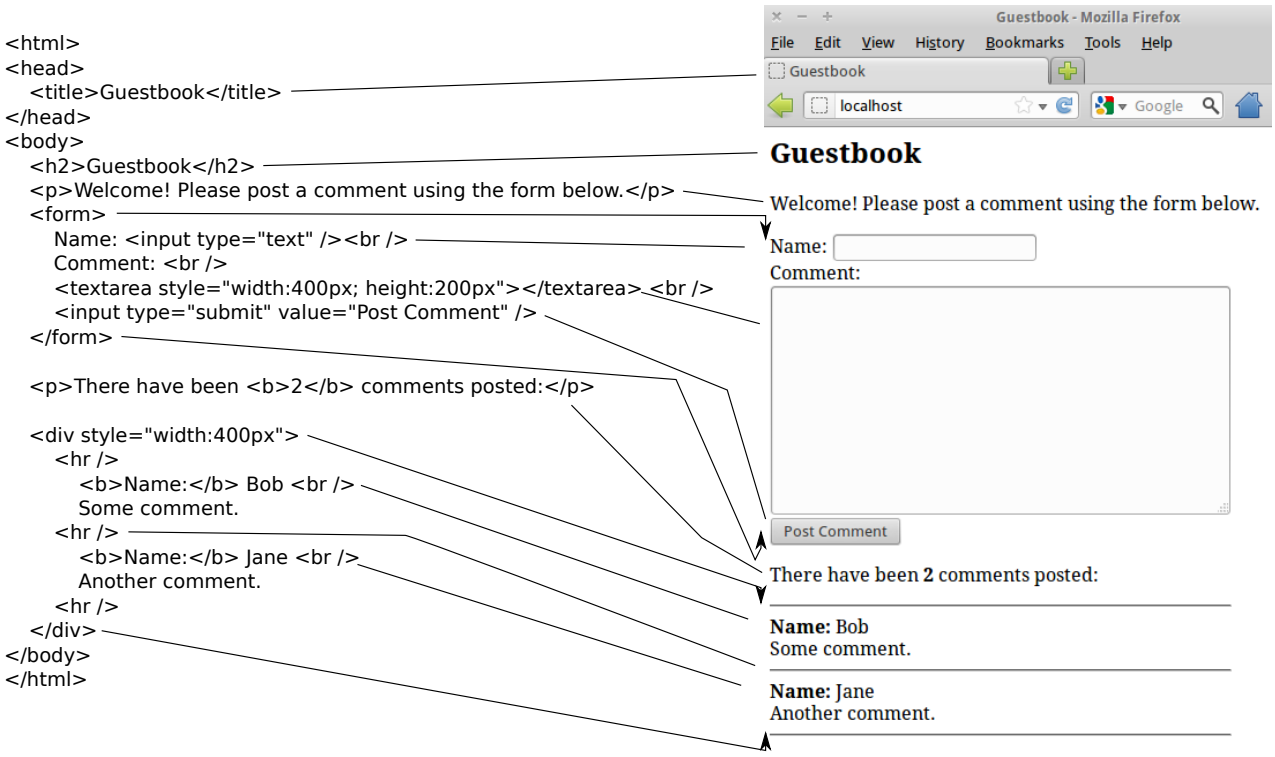

Figure 2.5: Mappings between the HTML and the browser rendering.

At this point, you have a page that displays a guestbook—but it's not interactive. Clicking the button only refreshes the page, and the text you enter into either of the input fields doesn't get saved; the comments that are displayed are always the same two boring sample comments. You'll revisit this example later in the book, where you make it functional. But for that you need PHP, which is the topic of the next chapter.

2.6 Exercises

1. Modify the HTML for the guestbook by adding an input field (between the "Name" and "Comment" fields) where the user can enter their email address.

2. Add an `<img>` tag on the guestbook page that displays an image from a URL somewhere on the Internet.

3. A frequently used tag that I didn't cover in this chapter is `<table>`. You can read about it here: http://www.w3schools.com/html/html_tables.asp. Use a table to format the input fields for the user's name and email address nicely so that

the two input fields line up with each other. The end result should look something like in this picture:

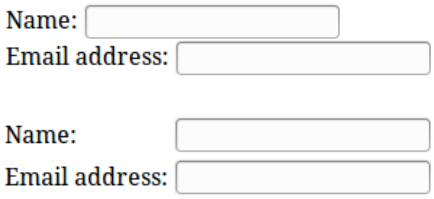

Figure 2.6: Input fields without using a table (top) and with a table (bottom). The table forces the input fields to line up with each other.

Appendix A

Solutions to Exercises

A.1 Chapter 1

2. When you go to http://localhost/newfile.php, your browser should display the contents of `newfile.php`, the file you just put in your `www` folder. When you go to http://localhost/index.php in your browser, you should see the contents of `index.php`; in other words, the same thing as if you went to plain old http://localhost.

What this means is that the part of the URL that comes after `localhost/` tells Apache which file you want to view in the `www` folder. `index.php` is a special name because it's the file you see if you don't put anything after `localhost/`.

3. You should see a page that says something along the lines of "404 File Not Found."

4. To view `myfile.php`, you would go to http://localhost/myfile.php.

A.2 Chapter 2

1. The new input field should go below the "Name" field and above the "Comment" field.

```
<!-- snip -->
Name: <input type="text" />
Email address: <input type="text" />
Comment: <br />
```

```
<!-- snip -->
```

By the way, the text inside <!-- and --> is what's known as a **comment**. It is ignored by browsers and is meant only for human readers. In this case, I'm using the comments to denote the presence of additional HTML that I haven't copied onto this page.

2. This tag would add the Google logo to the page:

```
<img src="https://www.google.com/images/srpr/logo3w.png" />
```

You can put it anywhere between the opening and closing body tags.

3. The table looks like this:

```
<table>
<tr><td>Name:</td><td><input type="text" /></td></tr>
<tr><td>Email address:</td><td><input type="text" /></td></tr>
</table>
```

The <tr> tags denote the start of a row, whereas the <td> tags denote the start of a column (i.e., a new cell within a row).

Appendix B

Additional Context

Author's Note: This section further elaborates and justifies some of the points I make in the Preface.

What separates my book from most other introductory books is that those books teach programming without giving significant greater context for its use. By the end of the book, readers might become good at solving the book's abstract exercises using the skills they learned, but they won't know how to use said skills for anything "real." As Brown et al. (1989) put it, "...it is common for students to acquire algorithms, routines, and decontextualized definitions that they cannot use and that, therefore, lie inert" (p. 33). Brown et al. also say, "Recent investigations of learning, however, challenge this separating of what is learned from how it is learned and used. The activity in which knowledge is developed and deployed, it is now argued, is not separable from or ancillary to learning and cognition. Nor is it neutral. Rather, it is an integral part of what is learned. Situations might be said to co-produce knowledge through activity. Learning and cognition, it is now possible to argue, are fundamentally situated" (p. 32). That is to say, most programming texts separate what is being learned—basic programming concepts—from how they might be used in the real world. My teaching of web development along with coding skills is an attempt to eliminate that separation.

Using a real-world context for learning programming can also help increase a reader's situational interest, which, in turn, increases their motivation for continuing with the book. Eccles and Wigfield (2002) write, "The following text features have been found to arouse situational interest and promote text comprehension and recall: personal relevance, novelty, activity level, and comprehensibility" (p. 115). As my target audience is familiar with the internet, my hope is that learning how to make a webpage will be personally relevant to them. My book will also meet Eccles and Wigfield's three other suggestions for arousing situational interest: activity level, because the text contains exercises and examples the reader is meant to follow along with; novelty, because web development and

coding will be new to the reader; and comprehensibility, because I will take care to write at the reader's level and make liberal use of the 1st and 2nd person in my narratives. Dalton (1988) also speaks to the importance of writing in the reader's language: "So a smart textbook can be defined as a book that does what the human teacher attempts to do, i.e., anticipate learner characteristics and the learning environment and thus enhance and encourage student learning" (p. 230). My "anticipation" of my readers in the book will involve me giving the same kinds of explanations that I might give if I were actually having conversations with them.

My use of one overarching example throughout the text is meant to be motivating to my readers in the sense that they will be able to actually see the results of their work (an "artifact") at the end of the book. Blumenfeld (1991) writes, "Students' freedom to generate artifacts is critical, because it is through this process of generation that students construct their knowledge the doing and the learning are inextricable" (p. 372). Furthermore, my use of a longer example by the end of the book will allow me to highlight a larger number of good general coding principles—ones that only become apparent with lengthier code (such as the need for comments). This is important according to Borstler et al. (2011): "The transfer literature suggests that the most effective transfer may come from a balance of specific examples and general principles, not from either one alone" (p. 3).

As my book is set up as a tutorial that walks readers through the creation of various pieces of code, it can be classified as a type of exercise that Nilson (1997) refers to as a *guided lab* (p. 81). Nilson claims that this kind of exercise is best suited for a type of learner known as a *converger*: "Convergers rely primarily on their skill of abstract conceptualization and active experimentation in their learning. They are often characterized as asocial and unemotional, preferring to work with things rather than people. What grabs their attention is how things work. They enjoy assignments that require practical applications, experimentation, and, in the end, precise, concrete answers" (p. 81). I am comfortable favoring this learning style over others for two reasons: first, it is my preferred learning style and as such I feel that I have good insight into the needs of this kind of reader, and second, it is also the learning style that many online tutorials target, and I feel that being exposed to this style of writing is useful for would-be programmers.

While convergers (especially ones that are also digital learners) are the people for whom my book will most likely be successful, Nilson gives suggestions for helping other learning styles that I see no reason not to implement. For example, to help visual learners, Nilson recommends "illustrations, pictures, diagrams, flow charts, graphs, graphic models and organizers (conceptual "tree" and "bubble diagrams), and graphic metaphors" (p. 83). Additionally, some readers might be *intuitive thinkers*, who "learn not by building up from details and examples but by first understanding the overarching global principle" (p. 85). To help these learners, I will summarize each chapter's contents at the beginning of the chapter.

Works Cited

Blumenfeld, P. "Motivating Project-Based Learning: Sustaining the Doing, Supporting the Learning." In Educational Psychologist, 26, No. 3 & 4, 1991, 369-398.

Borstler, J., Nordstrom, M., and Paterson, J.H. "On the quality of examples in introductory Java textbooks." ACM Trans. Comput. Educ. 11, 1, Article 3, (2011), 1-21.

Brown, A. S., Collins, A., and Duguid, P. "Situated Cognition and the Culture of Learning." In Educational Researcher, 18, No. 1, (1989), 32-41.

Dalton, W.K. "Smart ET textbooks: Why bind them at all." In Frontiers in Education Conference, 1988., Proceedings, vol., no., (1988), 229-233, 22-25

Eccles, J. and Wigfield, A. "Motivational Beliefs, Values, and Goals. In *Annual Review of Psychology*, 53, (2002), 109 132.

Nilson, Linda. "Teaching to Different Styles." In Teaching at Its Best, Bolton: Anker Publishing Company, 1997, 79 - 86.