Introducing Cloud-Based Mobile Apps

WHAT YOU WILL LEARN IN THIS CHAPTER:

➤ Using your existing skills as a web developer to build mobile apps
➤ Understanding how HTML5 will be used as an app-development standard
➤ Learning how to dynamically create JavaScript functions
➤ Using the WebKit browser engine for app development
➤ Creating a mobile web app that responds to touch
➤ Installing and using the nginx web server

This book is for web developers who want to build mobile apps and cloud services. If you know HTML, CSS, and JavaScript, you already have the skills to build not only mobile apps but also the cloud services that power them.

The code examples in this book show you how to build complete apps. You are never left to put together the pieces yourself. The code is simple and includes error-handling logic, so you’ll learn how to build production-ready apps and systems.

Over the course of this book, you will build three complete applications. You’ll learn how to put together all the elements of the technology stack, and you’ll learn about a wide range of technologies and services. This book will enable you to get to work but avoids unnecessary detail and theory.

This book is an accelerator for your skills. You can use it to efficiently make the leap into mobile and cloud development. Rather than attempting to be a reference for all the details, which you can find on the web anyway, it is a stepping stone for your skills.
CHAPTER 1
INTRODUCING CLOUD-BASED MOBILE APPS

HOW TO BUILD MOBILE APPS IN THE CLOUD

This book describes how to build apps that run on the new generation of smart mobile devices. It also shows how to build out the business logic behind these apps, and how to run that business logic in a cloud hosting environment, such as that provided by Amazon.

This book focuses on the two leading platforms: iPhone and Android. These two, between them, cover the vast majority of smartphones and provide access to the largest market.

NOTE In this book, the term iPhone should be taken as shorthand for any iOS-based device, including iPad and the iPod Touch devices. Similarly, the term Android refers to any device running Android version 2.1 or higher, including any of the Android tablets that are competing with the iPad.

It’s important to understand the types of apps that can run on mobile devices:

- **Mobile web apps** — These apps are really just websites, designed to function in an app-like way. They run in a web browser on a device.

- **Purely native apps** — These apps are written in a device-specific language, using a device-specific programming interface: Objective-C for iPhone apps or Java for Android apps. Native apps can access all the capabilities of the device and can take many forms, from simple utility apps to advanced 3-D games.

- **Hybrid native apps** — For these apps, you use HTML to build the user interface but wrap the HTML in a native container. Such apps can access some of the native capabilities of the device but can still be developed using HTML.

In this book you will learn how to build mobile web apps and hybrid apps.

The other component that many apps have is not something that lives on the mobile device at all. It is the business logic, data storage, and service integration that support the operation of the code on the mobile device. This element of app development is just as important as the visible part that you install on your device. Placing this code in a cloud-hosting environment is the best approach to developing a robust support system for your app, and this book shows you how to build the server elements of your app. You will learn how to do this by using your existing JavaScript skills. You’ll run your code on the server using Node.js, a high-speed, high-capacity JavaScript server engine.

Using Your Existing Skill Set

As a web developer, you already possess all the skills you need to be a mobile app developer as well. If you can build websites, you can build mobile apps. If you are a web developer wanting to build mobile apps, you do not need to learn new languages such as Objective-C or Java. You do not even need to learn new languages to build the code for servers that support your app.

All you need to know is HTML, CSS, and JavaScript. This book assumes that you have a working knowledge of these three basic web languages. Even if you are more comfortable with design and
graphics and are stronger in HTML and CSS than in JavaScript, you will still be able to follow the examples in this book and build your own apps.

This book takes a practical approach and shows you how to build real applications. The examples stick to common language features and avoid anything esoteric. The first set of code examples in this chapter lay the JavaScript groundwork that will see you through to the end of the book.

You will use your existing skill set to build mobile web apps. You will then support those apps by using some server-side JavaScript, running on cloud servers, and you'll see all the steps needed to set this up. Then you'll learn how to create hybrid native apps using HTML, CSS, and JavaScript.

**Determining What Tools You Need**

You'll need some development tools in order to fully explore the examples in this book. You will certainly find a physical iPhone or Android device very useful for testing and development. You need to run mobile apps on an actual device to really understand how they will behave once users get hold of them.

To a certain extent, you can develop the apps and code examples in this book on any of the three major operating systems: Mac, Windows, or Linux. However, you will find that a Mac is the best choice, simply because the iPhone development tools from Apple can only run on a Mac. Your Mac can also run the server code quite easily. One thing you should do is upgrade to the latest version of the Mac OS X operating system, as this will support the most up-to-date versions of the iPhone development tools.

Windows and Linux are also acceptable, although you will have to do a little more configuration and setup work. In particular, on Windows, you will need to install the Cygwin UNIX environment so that you can run Node.js. Cygwin is available from [www.cygwin.com](http://www.cygwin.com). As discussed in Chapter 11, you will also have to rely on third-party services to build hybrid native iPhone apps.

You can build mobile web apps and the necessary server code using your existing development tools. All you need is a good code editor, and I'm sure you've already chosen a favorite for coding websites. You'll also be using the command line quite a bit, especially for the server code. But don't worry if you're not comfortable with the command line; this book gives you exactly the commands you need to run.

Later in this book, you'll need to download and install the software development kits (SDKs) for iPhone and Android development. These SDKs are provided as part of the Xcode (for iPhone) and Eclipse (for Android) development environments. Xcode runs only on a Mac, but you can run Eclipse on all three operating systems.

The final development tool you'll use is the Safari web browser. You can download this directly from the Apple website: [www.apple.com/safari](http://www.apple.com/safari). You will use Safari as a test and deployment tool because Safari is built with the open source WebKit browser engine. This browser engine is used on both iPhone and Android, and it is the web browser environment for which you need to develop. In your coding work cycle, you will use the desktop Safari browser as your test system.
The Skills You’ll Learn

As you work through this book, you’ll learn and enhance a wide range of skills. These skills will cover the entire mobile app technology stack, from the device, to the server, to the database. These will be practical skills, and theory will be kept to a minimum. All the code examples in this book create complete, fully working apps. You’ll be able to use them as foundations for your own work.

You’ll make good use of the new features in HTML5 when you build mobile web apps and HTML5-based native apps. The Safari browser and the WebKit engine have good support for many of the features of HTML5. You’ll be able to use local on-device storage, app caching, geolocation, and even audio and video support.

You’ll also learn about the special metatags and design considerations needed for mobile web app development. These allow you to deal with different screen sizes and device capabilities. They also allow you to define home screen icons and loading screens for your app.

To make the transition from a mobile web app to a hybrid native app, you’ll use the open source PhoneGap project. This project provides a framework that allows you to embed your HTML in a native container. This is how you will build native apps using JavaScript. PhoneGap also provides you with an extended set of JavaScript functions that let you access the device camera and use the device accelerometer to detect movement.

The app on a device is only part of the story. The aim of this book is to teach you how to create mobile apps that provide complete services. This means you’ll need to write server code that handles some of your business logic, such as user account management, user data storage, and integration with third-party cloud services. You’ll also learn how to provide social media logins using Facebook and Twitter.

You will use JavaScript to build the server-side logic. You’ll run your JavaScript in an application server known as Node.js (or more commonly, just “Node”), a JavaScript server created using the Google JavaScript engine from the Chrome web browser. Node is amazingly fast and designed to handle thousands of concurrent clients, so it’s perfect for running a popular app.

You’ll need a place to put your server-side code, so you’ll learn how to host your code in the cloud, using Amazon Web Services (AWS). AWS is a full cloud-hosting system provided by Amazon that lets you create your servers and store images and files; it provides a nearly infinite database. You’ll learn how to set up and configure an Amazon server and how to access the Amazon services over the web.

Finally, you’ll learn how to work with next-generation databases that go beyond traditional tables, columns, and rows. You’ll learn how to work with these schemaless database systems and how to synchronize them with the data on your mobile device.

TWO BIG IDEAS ABOUT THE APP INDUSTRY

This book is based on two predictions about where the app industry is going:

➤ Cloud computing will be the primary way to build the service infrastructure for apps — Most app developers accept this idea.

➤ Using HTML5 is a great way to build apps and will only become better — This view is still quite controversial.
The following sections take a closer look at these predictions and how they affect you, as a web developer and as a mobile app developer.

**Web Apps and the Future**

There are two types of web apps: mobile web apps and hybrid native apps. A mobile web app is delivered as a website and runs in a web browser on a mobile device. You can bookmark web apps on the home screen of the device, and you can give them icons and loading screens (as you’ll see in Chapter 4). The user experience with mobile web apps is essentially different from that of native apps, but you can remove many of the differences.

The other type of web app, the hybrid native app, is actually a native app that runs a web browser inside itself. The entire app user interface is actually a web page that runs your HTML5 code. You use wrapper systems, such as the open source PhoneGap, to create a native wrapper. The native wrapper is a native app, but it only really does two things: create a WebView control (the native element that displays HTML) and provide access to device capabilities such as the camera via a JavaScript API.

This book will show you how to create both types of apps. You should carefully consider which approach is the best choice. The huge advantage of mobile web apps is that they are ultimately just websites, which means they do not have to be submitted to any app stores for approval. You can update them as frequently as you like, and they can be accessed from almost all smartphones (although functionality may be limited outside the iPhone and Android platforms).

However, hybrid native apps have the advantage of being proper apps that are listed in the app stores. They can use a device to its full potential, and you can extended these apps with native code, if necessary. You might be surprised to learn that a great many “purely” native apps actually make extensive use of HTML5 WebView controls. This is the main way to display rich text interfaces, even for purely native apps.

The following are the main reasons to develop an app using HTML5, whether as a mobile web app or within a hybrid native wrapper:

- **Cross-platform** — Your app is automatically cross-platform, and you have to develop it only once. The minor debugging required to handle device browser differences is inconsequential compared to the huge effort required to port an entire app from one platform language to another.

- **Standards compliant** — Long after the current set of mobile platforms have become historical entries in Wikipedia, your HTML5 app will still be running. HTML, as a technology choice, is a completely safe bet. It is quickly becoming the primary means of building user interfaces on any device, including tablet computers.

- **Lower-cost rapid development** — Developing your app with HTML, CSS, and JavaScript means that you can build and iterate extremely quickly. You have the advantage of many tools that support HTML, along with a wide pool of developer talent. Even if your ultimate goal is to create a native hybrid app, you can still do most of your development using a web browser, which means a much faster work cycle.

- **Low-friction deployment** — You can launch and update mobile web apps immediately, without waiting for a third-party approval process. You have complete control over content, user base, and commercial activities. No vendor is powerful enough to control the web.
Easy to learn — You already have a good knowledge of web languages, so you can start building apps right away just by learning to deal with the particularities of mobile web app development. You do not need to invest any time in learning a new language such as Objective-C, which has limited use outside its own ecosystem.

JavaScript — One important reason to build apps with HTML5 is that you will use JavaScript as your development language. Long-neglected as a toy scripting language and useful only for form validation and drop-down menus, JavaScript is emerging as the next big industry language. This change is happening because it is now possible to use JavaScript not only for websites but also for mobile apps and for server code. JavaScript is one of the few languages that can cover the entire technology stack.

JavaScript has certain weaknesses, including an unfortunate syntax inherited from the C language. But it is also capable of supporting advanced functional and object-oriented programming styles. You’ll see examples of this power in many of the code examples in this book. With JavaScript, you need relatively few lines of code to quickly build complex apps, and you can easily debug those apps interactively in your web browser. As a result, when you adopt JavaScript as your primary language for mobile and cloud development, you will experience a huge increase in your software development productivity.

The Cloud as the Future

Cloud computing means many things. For mobile app developers, it provides the ability to build apps that millions of people can use. Cloud computing makes it easier to handle large and growing numbers of users.

This book shows you how to build your own cloud-based system from the ground up, using the Amazon cloud. You can use the same basic approach with other cloud vendors, such as Rackspace or Joyent. Chapter 11 covers the use of higher-level services that completely remove the need for any server configuration and just run your code for you.

You do not need to use traditional server-side languages such as Java or C# to build the cloud element of your app. You do not even need to know any of the existing scripting languages, such as Ruby, Python, or PHP. By using the Node server, you can run JavaScript on the server side. This means you can use a single language for all your development. You can stay focused on what you need to do, without being distracted by the differences between programming languages.

Using the cloud to build your server-side business logic has the following advantages:

➤ **Low cost** — It is easy to get started, as you pay only for what you use. There is no need to buy a server or sign up for a fixed monthly fee.

➤ **High capacity** — The cloud can provide you with as much bandwidth and storage as you need. You can grow your service easily as your user base grows.

➤ **Flexibility** — You can add and remove servers and databases very quickly, you can add capacity in the geographic regions where needed, and you can easily integrate with third-party services.

➤ **Low maintenance** — You do not need to worry about system administration or configuration. You can use prebuilt and preconfigured machines. You can use cloud databases that do not have to be tuned for performance.

By the end of Chapter 2, you’ll be running a cloud-based mobile web app.
GETTING STARTED

This book contains many code examples. For the most part, each example is contained within its own folder, and you will be instructed to create these folders when needed. You do, however, need to decide on a place to keep all the sample code. The best thing to do is to create a Projects folder inside your home folder and store everything there. Of course, you are free to use a different folder structure if it suits you better.

All the code in this book is available for download. If you expand the downloadable zip file for each chapter into your Projects folder, you will end up with the full set of code examples in one easy-to-access location.

For the purposes of learning the material in this book, you may find it helpful to create some of the code by hand. The examples assume that you are doing everything manually, even if you are just running the downloadable code. The precise manual steps are explained so that you can follow along and understand not just the code but also how to put the mobile apps and cloud systems together end-to-end.

The instructions assume basic knowledge of the command line. They also assume that you are familiar with your code editor and know how to create folders and manipulate files.

Let’s begin!

Using JavaScript Functions

In this book, you will use JavaScript as the primary language, on both the client and the server. JavaScript is a strange language, with some really great parts and some truly awful parts. If you’ve only used JavaScript for form validation or doing some HTML manipulation, then reading this book will help you get to the next level with JavaScript.

The most important part of JavaScript to understand is the way that functions work. This book makes liberal use of such things as dynamic functions, closures, and callbacks. This section is a primer on these aspects of JavaScript functions. If you already know your way around these concepts, feel free to skip to the next section. If not, try out these simple exercises to get a feel for some of the more advanced uses of functions in JavaScript.

NOTE Each of the JavaScript examples in this section is a self-contained HTML page that you can load directly into your web browser from the file system on your computer. The code in the examples inserts some output text into the blank HTML page so you can see what you are doing. These examples will work in all major browsers, but you should really use Safari or Chrome when doing mobile web app development, as these browsers use the same WebKit engine as iPhone and Android devices.

JavaScript Literal Data Structures

One of the best things about JavaScript is that it is really easy to define data structures directly in code. The JavaScript syntax for doing this is so easy that it has inspired a data exchange format
called JavaScript Object Notation, otherwise known as JSON. You'll be doing a lot of work with JSON in later chapters.

The following example shows the basic syntax for arrays and objects in JavaScript. If your JavaScript is a little rusty, or if you've just picked it up by copying random code snippets, then you'll find this example useful. If you already know your way around JSON, you might want to skip ahead to the next example.

The word *literal* used in this example means that you can type in the data structure directly, without executing any code to build it. As in many other languages, a literal JavaScript array is a comma-separated list of values, enclosed in square brackets:

```
["a", "b", "c"]
```

A literal JavaScript object is a list of key/value pairs, enclosed in braces:

```
{ ... }
```

The keys are strings, and the values can be anything from numbers to strings to literal arrays and objects themselves:

```
{"a":100, "b":"BBB", "c":["a", "b", "c"]}
```

The key/value pairs are separated by a : character, with the key on the left. As shorthand, you can remove the quotation marks from keys if they contain only letters and numbers:

```
{a:100, b:"BBB", c:["a", "b", "c"]}
```

---

**TRY IT OUT**  Using Literal Data Structures in JavaScript

The code in this example shows you how to set and get values from arrays and objects, as well as how to print them in JSON format. Take a look:

1. Using your code editor, create a new file called `js-literals.html` in your Projects folder.
2. Insert the following HTML code into the new file and save it:

```html
<DOCTYPE html>
<html><head></head><body id="main"><script>
var main = document.getElementById('main');

var myarray = ['a', 'b'];
myarray.push('c');

var myobject = {
  a: 'AAA',
  b: 'BBB'
};
myobject.c = 'CCC';

main.innerHTML =
```
"<pre> +
"myarray[0] is " + myarray[0] + "<br>" +
"myarray[1] is " + myarray[1] + "<br>" +
"myobject.a is " + myobject.a + "<br>" +
"myobject.b is " + myobject.b + "<br>" +
"myobject.c is " + myobject.c + "<br>" +
"myobject['a'] is " + myobject['a'] + "<br>" +
"myobject['b'] is " + myobject['b'] + "<br>" +
"myobject['c'] is " + myobject['c'] + "<br>" +
"myarray is " + JSON.stringify(myarray) + "<br>" +
"myobject is " + JSON.stringify(myobject) + "<br>" +
"<pre>
</script></body></html>

code snippet js-literals.html

To avoid typing all this code in by hand, you can instead use the downloadable code examples.

3. Open the js-literals.html file in your web browser. You can do this by double-clicking the file in your operating system’s file explorer application. Or you can select File ➪ Open in your browser. Your web browser should display the following text:

myarray[0] is a
myarray[1] is b
myarray[2] is c
myobject.a is AAA
myobject.b is BBB
myobject.c is CCC
myobject['a'] is AAA
myobject['b'] is BBB
myobject['c'] is CCC
myarray is ['a','b','c']
myobject is {"a":"AAA","b":"BBB","c":"CCC"}

**WARNING** If you are using the Chrome browser, you’ll find that there is no File ➪ Open command. Instead, you have to enter the full folder path to the js-literals.html file in the address bar, prefixed with file:///. Here’s an example: file:///home/richard/Projects/js-literals.html.

**How It Works**

For most of the examples in this book, the How It Works section goes through the code from top to bottom, showing how each feature is implemented. As you go through this book, you’ll be able to understand larger sections of code in one go.
This example involves an HTML file. In fact, it is a HTML version 5 file, because you used the HTML5 document type at the start of the file:

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

Next, you define some boilerplate HTML to create an empty document. You’ll reuse this boilerplate code in the other examples in this chapter. The boilerplate contains the standard HTML document tags: the `html` tag containing `head` and `body` tags. The `body` tag has an `id` attribute so that you can refer to it in the JavaScript code. Everything is squashed onto one line to keep it out of the way of the JavaScript code, which is the star of this show. At the end of this second line, an opening `script` tag means you can start to write some JavaScript:

```html
<html><head></head><body id="main"><script>
```

The first line of JavaScript is also boilerplate. You need to get a reference to the `body` tag so that you can output some text to demonstrate various features of JavaScript literals. This line of code uses the built-in JavaScript document API provided by all browsers. This API, known as the Document Object Model (DOM), is standardized. You store the `body` tag reference in a variable called `main`:

```javascript
var main = document.getElementById('main);
```

Now on to the actual sample code. First, you define a literal array:

```javascript
var myarray = ["a","b"];
```

This array-building syntax is common to many languages and should feel quite comfortable to you. You can use either single or double quotes in JavaScript when you are providing literal string values. If you use single quotes, you do not need to escape double quotes (by using the `\` character), and the same goes for single quotes inside double quotes. Thus the following are both acceptable:

```javascript
var single = "quoted"
var double = '"quoted"'
```

To set an array value, you use the common square bracket syntax. For example, `anarray[2] = "foo"` sets the third element (array indexes start at 0) to the string value "foo".

In this book and in your own apps, you will often need to append values to the end of an array. You can do this by using the built-in `push` function, like so:

```javascript
myarray.push('c');
```

In the code example, this means that the `myarray` array variable now contains three values, “a”, “b”, and “c”. Every array object has a `push` function that you can use in this way.

The next lines of code define an object literal. The keys in the key/value pairs are referred to as the `properties` of the object. This object has two property keys, `a` and `b`. The `a` property key has the value `AAA`, and the `b` property key has the value `BBB`:

```javascript
var myobject = {
    a: 'AAA',
    b: 'BBB'
};
```
b: 'BBB'
];

The literal object syntax always follows this pattern:

{ <key1>:<value1>, <key2>:<value2>, ...}

Once the object is created, you can also set further values. In this line of code, you set the value of the c property key to CCC:

myobject.c = 'CCC';

This syntax, known as *dot notation*, lets you specify the properties of an object in an abbreviated manner. Dot notation is really just a shorthand for the full square bracket notation: myobject['d'] = 'DDD'. You can use dot notation when you already know the name of the property, and you can use the square bracket notation when you are using a variable whose value is the name of the property or when the property name contains non-alphanumeric characters that need to be inside quotes.

**NOTE** The examples in this book use the literal object notation a great deal, and it will become second nature to you very quickly.

The final statement in the sample code is a multiline statement that displays a selection of syntax examples for accessing the values of array and object literals. The statement builds a string containing HTML code and then uses the innerHTML DOM property of the body tag (referenced by the main variable), to set the contents of the HTML page:

```javascript
main.innerHTML = "<pre>" +
"myarray[0] is " + myarray[0] + "<br>" +
"myarray[1] is " + myarray[1] + "<br>" +
"myobject.a is " + myobject.a + "<br>" +
"myobject.b is " + myobject.b + "<br>" +
"myobject.c is " + myobject.c + "<br>" +
"myobject['a'] is " + myobject['a'] + "<br>" +
"myobject['b'] is " + myobject['b'] + "<br>" +
"myobject['c'] is " + myobject['c'] + "<br>" +
```

The `pre` tag is used to display the output as monospaced source code.

The first group of lines shows you how to access array values (no surprises here):

The second group of lines shows how dot notation works:

The next group of lines shows the same values, this time using square brackets to access the property values:
Finally, the last two lines display the `myarray` and `myobject` variables, using the JSON data format. Modern web browsers include a special JSON utility object. The `JSON.stringify` function converts a variable into a string containing the textual representation of the JSON encoding of the value of that variable. Of course, this looks almost exactly like the original object literal syntax you used at the top of the file:

```
"myarray is " + JSON.stringify(myarray) + "<br>
"myobject is " + JSON.stringify(myobject) + "<br>
```

The last line of the statement closes the `pre` tag:

```
"</pre>
```

At the end are the closing HTML tags:

```
</script></body></html>
```

Notice that the `script` tag in this example is not inside the `head` tag but rather inside the `body` tag. This means that the `script` tag code will run after the browser has prepared the DOM. And this means that the `document.getElementById` function will return an actual element rather than null. You can also use the `onload` event handler to the same end, but this example keeps the boilerplate code to a minimum.

The rest of the examples focus on the workings of the JavaScript code, and the boilerplate works in the same way for each example.

You now have enough knowledge of the JavaScript literal syntax to follow the rest of the examples in this book.

### JavaScript Functions

JavaScript functions are both the same as and different from functions in other languages. You can create them before your code runs or while your code is running, and you can even create them without names. In this book, because you focus on Safari, you do not need to worry about cross-browser differences in the way that function definitions are handled. The examples in this book also stick to an easy-to-understand subset of all the ways you can use functions. This example covers the main things you need to know.

#### TRY IT OUT Writing JavaScript Functions

There are two main ways to create functions in JavaScript: by using function declarations or function expressions. You can use function declarations to create your functions ahead of time, before your code runs. You can use function expressions to create your functions on demand when your code is running. To see how these two methods are used, follow these steps:

1. Using your code editor, create a new file called `js-functions.html` in your Projects folder.
2. Insert the following HTML code into the new file and save it:
<html><head></head><body id="main"><script>
var main = document.getElementById('main');
var log = [];
log.push('<pre>');

log.push( declaration() );

function declaration() {
  return "declaration";
}

var expression = function() {
  return "expression";
};

log.push( expression() );

var myobject = {
  property: function() {
    return "property";
  }
}

log.push( myobject.property() );

log.push('</pre>');
main.innerHTML = log.join('<br>');
</script></body></html>

code snippet js-literals.html

If you prefer, you can use the downloadable code.

3. Open the js-functions.html file in your web browser by either double-clicking the file or using the File ➪ Open menu command in your browser. Your web browser will display the following text:

declaration
expression
property

How It Works

As in the previous example, this HTML file uses boilerplate code. The HTML tags are the same, as is the reference to the body tag stored in the main variable.

This time, you use the JavaScript array variable log to store the results of the code execution. log contains a list of strings that you build up over time by recording what happens in the code. You start with an empty array:

var log = [];
You use the `push` function to append strings to the array. Because you will want to display the output in a monospaced font when you are finished, you’ll enclose the logging strings in a `pre` tag. The first `push` sets this up:

    log.push('<pre>');

Next, you push the result returned from calling the `declaration` function onto the log:

    log.push( declaration() );

But the `declaration` function has not been written yet! How can this work? It works because the `declaration` function is written using the syntax for function `declaration`. In this syntax, the name of the function appears after the `function` keyword, as highlighted below:

    function declaration() {
        return "declaration";
    }

This means that the `declaration` function is created before any code runs. By the time your code does run, the `declaration` function is already defined and is ready and waiting to be used.

The next function is defined using a function `expression`. You can tell it’s an expression because the function is being treated like the value of a variable:

    var expression = function() {
        return "expression";
    }

Here, the `expression` variable is assigned a function as its value. This function is anonymous: There is no name after the `function` keyword. But you can still call the function by using the name of the variable that points at the function:

    log.push( expression() );

In this line of code, you are again pushing the string value returned by the function onto the log array. You call the function by using the ordinary syntax:

    <function_name>()

You call functions created using expressions the same way you call declared functions, except you use the name of the variable.

The other important point about function expressions is that you must define a function before you call it. In this example, the `expression()` call can work only after the `var expression = function()` `{ ... }` lines.

There is a special case of function expressions that you will use quite a bit in this book. You can assign anonymous functions to the properties of an object. The function expression is the value of the property. The value just happens to be a function, and not a string or number:
var myobject = {
    property: function() {
        return "property";
    }
};

You call functions defined in this way by referencing them as object properties:

    log.push( myobject.property() );

This line uses dot notation, but you could also write this:

    log.push( myobject["property"]() );

The main reason for placing functions inside an object in this way is to put them together into logical
groups. This makes object-oriented programming easier.

The last thing you need to do is display the results of all this function calling. First, you close the <pre> tag:

    log.push('</pre>');

Then you use the innerHTML property of the body tag to set the textual content of the page. To turn
the array into a string, you use the join function. This function is available on all arrays, and it joins
up the values into one long string. You can provide an argument, such as "<br>", to the join function,
and it will place the string value of the argument between the values in the final string. The br tag
places each log entry on a new line.

This is not all there is to JavaScript functions. At the end of this section are some links to more detailed
resources. However, the code examples in this book almost always use one of these three ways of
creating functions, so you now know enough to follow the sample code.

---

Mastering the Power of Callback Functions

The most common function code pattern you will see in this book is the callback pattern. The idea
is that when you ask a function to do something for you, instead of getting a result right away as a
return value, you'll get a result later. This is useful when the function takes time to finish its job or
only registers your interest in future events.

Consider the event-handling functions that web browsers provide, such as onclick or onkeypress.
These can't return a click or keypress event to you as a return value because the click or key
press only happens later, when the user performs those actions. If your code had to wait for a return
value, it would be blocked, unable to proceed any further, and none of the rest of your code
would run.

The solution that event handlers use is a callback function. You provide a function to the event
handler when you set it up, and later, when the event happens, your callback function itself is called.
This pattern is very easy to set up in JavaScript because you can create dynamic functions easily
by using function expressions. In the most common form of this pattern, you pass an anonymous callback function directly as an argument to an event handler:

```javascript
function eventHandler( function() {
    // body of your anonymous callback function
})
```

This pattern is also useful for things like database queries. You’ll learn all about writing server-side JavaScript using the Node server later in this book, and you’ll see many examples of this pattern in action. Because a database query takes some time to complete — you have to send data over the network between yourself and the database, and the database has to perform its own work — your query result will take time to come back to you. So you use the callback pattern and receive the query result via your callback function.

### TRY IT OUT Using Callback Functions

Callback functions are just like normal functions in that they have parameters that are used to receive the data they work on. In the case of callbacks, the parameters contain the results of the calling function's work. Here’s a simple example to show how this works:

1. Create a new file called `js-callbacks.html` in your Projects folder.
2. Insert the following HTML into this file:

```html
<!DOCTYPE html>
<html><head></head><body id="main"><script>
var main = document.getElementById('main');
var log = ['<pre>']
function waitForSomething( callback ) {
    callback('called')
}
waitForSomething( function( someText ) {
    log.push( someText )
})
log.push('</pre>')
main.innerHTML = log.join('<br>')
</script></body></html>
```

If you prefer, you can use the downloadable code.

3. View the `js-callbacks.html` file in your web browser. Your web browser should display the following:

```
called
```
How It Works

The boilerplate in this example is the same as the boilerplate in the preceding example. This example also uses the log variable as a string array to collect the output. The innerHTML element property and the array join function are used to display the logged output, as in the previous example.

In this example, the waitForSomething function is the function that will call the callback function. This code, because it is an example, just performs the call immediately and does not actually wait for anything:

```javascript
function waitForSomething( callback ) {
    callback('called')
}
```

The callback parameter is a variable that points to your callback function. It is passed in as a variable name, callback, and called as a function, callback(). This callback function is passed an argument when it is called, a string with the value 'called'. In the real world, this argument would be a mouse click event or database result set.

Now you use the waitForSomething function and pass in your callback function. To do this, you dynamically create an anonymous function and pass it in place as the first argument to waitForSomething. This is shown highlighted here:

```javascript
waitForSomething( function( someText ) {
    log.push( someText )
})
```

The anonymous callback function takes in a single parameter, someText, and pushes this parameter onto the output log right away. To help you understand how this works, here is a list of the lines of code, in the order in which they are executed:

A: waitForSomething( callback )
B: callback('called')
C: function( someText ) // someText == 'called'
D: log.push( someText )

You'll get used to callbacks very quickly. The syntax is quite convenient because you can define the actions to take when an event happens right at the place in the code where you indicate your interest in that event.

NOTE  Callback functions introduce a lot of brackets and braces into your code. Make sure your code editor helps you keep track of them (with some form of highlighting) and that you stick to a consistent coding and indentation style. It's easy to lose track of which closing bracket or brace matches which opening bracket or brace.
Dynamic Functions

In JavaScript, you can create new functions at any time, using function expressions. This means you can create new functions when your code is running. This can be very useful for things like error handling, and you’ll see examples of it in later chapters.

The other useful thing you can do with dynamic functions is to create functions that have some of their variables already set. This is useful because it means you have to write less code: You can drop any logic that has to figure out what the values should be.

Let’s say you have a photo gallery app. Each time the user clicks on a photo, you pop up an expanded version of the photo. You write an onclick event-handling function, which you attach to each photo that first inspects the click event to determine which photo was clicked on and expands that photo.

But you could also create the photo click event handler dynamically. For each photo, you create a new event-handler function that already knows which photo it is meant to expand. Now you only have to write the photo expander code, and you can drop the event inspection code.

How does the dynamically created function know which photo to expand? It references a variable outside itself that has a reference to the right photo. Of course, this outside variable still has to be within the scope of the function; it could be a local variable or function parameter of the code that creates the dynamic function.

**TRY IT OUT  Using Dynamic Functions**

This simple example shows dynamic functions in action. Follow these steps:

1. Create a new file called *js-dynamic.html* in your Projects folder.
2. Insert the following HTML into this file:

```html
<!DOCTYPE html>
<html><head></head><body id="main"><script>
var main = document.getElementById('main');
var log = ['<pre>

function make( color ) {
  var dynamic = function( thing ) {
    return thing + ' is ' + color
  }
  return dynamic
}

var greenify  = make('green')
var blueify = make('blue')

log.push( greenify('grass') )
log.push( blueify('sky') )
log.push('</pre>)
main.innerHTML = log.join('<br>')
</script></body></html>
```

*Code snippet js-callbacks.html*
If you prefer, you can use the downloadable code.

3. View the js-dynamics.html file in your web browser. Your web browser should display the following:

   grass is green
   sky is blue

How It Works

The boilerplate in this example is the same as in the preceding examples. The make function is where the example begins. The make function creates another function inside itself. It first assigns the dynamically created function to the variable dynamic and then returns this variable. The dynamic function is shown highlighted:

   function make( color ) {
       var dynamic = function( thing ) {
           return thing + ' is ' + color
       }
       return dynamic
   }

The dynamic function creates a string that tells the color of something. The name of the thing is passed in as the thing parameter. However, the color of the thing is obtained from outside the dynamic function. It happens to be the value of the color parameter passed to the make function.

To aid your understanding of what happens when a JavaScript function is called, you need to be familiar with the concept of scope. The scope of a function is simply all the variables that the function can reach. A function can reach all its local variables and parameters, of course. But it can also reach any variables that were defined outside the function at a higher level, such as global variables or variables in any higher-level functions that the function is itself inside.

In this case, the function assigned to the dynamic variable is inside the make function and so can access the variables and parameters of the make function. When you return the newly created dynamic function, the value of the color parameter can still be accessed by the new dynamic function. You can think of the color variable as being carried around by the dynamic function.

You can now use the make function to create some functions with preconfigured colors. You call the make function, it returns the dynamic function, and you store a reference to the dynamic function in a local variable. In the sample code, greenify and blueify reference separate dynamic functions where the color variable has the value 'green' or 'blue', respectively:

   var greenify  = make('green')
   var blueify = make('blue')

In the case of the photo expander app, instead of a color, you would pass in the HTML img element that corresponds to each photo.
Finally, you use the dynamic functions to create the desired output:

```javascript
log.push( greenify('grass') )
log.push( blueify('sky') )
```

There is a computer science term for dynamic functions (such as `greenify` and `blueify`) created in this way: They are known as *closures*.

### Learning More About JavaScript


### The WebKit Browser Engine

The built-in web browsers on the iPhone and Android use the open-source WebKit browser engine to display HTML. The WebKit project was launched in 2005 by Apple and has developers from many companies, including Google. The WebKit projects is descended from the KHTML project, which was one of the early Linux web browser engines, renowned for its small and clean code base. If you are interested, find more details at [http://webkit.org](http://webkit.org).

In this book, you will use WebKit-based browsers on both your desktop development machine and your mobile device. On the Mac operating system, you will use the desktop version, in the form of the desktop Safari browser, which uses WebKit as its HTML engine. A version of Safari is also available for Windows, downloadable from [http://www.apple.com/safari](http://www.apple.com/safari). On Linux, your easiest option is to use Chrome, which has some implementation differences from Safari but also uses WebKit — and it has virtually the same developer tools as Safari.

On your iPhone, you'll use the mobile version of WebKit, known as mobile Safari, and on Android, you'll also use WebKit, as the built-in browser is based on it.

The fact that your mobile apps will run on the WebKit engine on both desktop and mobile devices makes your life much easier. It means you can develop an app using the desktop version of Safari and see almost the same behavior there as you will see on a mobile device. There are differences, of course, and you will still need to test on a physical device.

You'll find that your development cycle will involve spending time testing in the desktop Safari browser, testing on the Safari running on the device simulators, and testing on the actual device using mobile Safari. Your code–test–debug cycle will be very fast when you are working with desktop Safari because you are just reloading a website. You'll find this is a great aid to productivity.

The other great thing about Safari is that the built-in developer tools are very good. If you have used the Firebug extension on the Firefox browser, you will feel right at home. To enable the developer
tools, you open the Safari Preferences window, select the Advanced tab, and select the Show Develop Menu option. When a menu item titled Develop appears in the menu bar, you choose the Show Web Inspector menu item. The Web Inspector window should appear at the bottom of the page.

Figure 1-1 shows the Web Inspector with the previous code example file, js-dynamic.html, open. The tabs show various types of information about the page. The Elements tab is particularly useful; you can use it to review the HTML structure of the document when you make dynamic changes to it as you animate or alter the user interface of an app.

Apple provides copious volumes of documentation on every aspect of the Safari browser. You can find everything at the Safari Dev Center: http://developer.apple.com/devcenter/safari. In this book, you learn how to use Safari to develop and debug the various features in your app, such as HTML5 local storage.

A Colorful Little App

Now that you know some basics, in this section, you’ll actually build an app! In this example, you will build an app that displays a box with a random color. Each time you click or tap the box, it will change to a new random color. Figure 1-2 shows the app in action, running on desktop Safari, with the developer console open.
What do you need to do to create this app? You'll put everything in one HTML page. You'll need a square div tag to change color. You'll need some JavaScript to generate random colors and change the color of the div tag when the user click or taps it.

For now, you'll just build the desktop version of this app. In the following section, you'll learn how to view it on a mobile device.

**TRY IT OUT**

**Developing a Mobile Web App**

You need to use the desktop Safari browser to view this example. You’ll use Safari so that you can be sure you’ll see the same results on your mobile device. As with all the other examples in this book, the code you see here is a full working example. Follow these steps:

1. Create a new subfolder called view in your Projects folder.
2. Create a new file called view.html in your Projects/view subfolder.
3. Insert the following HTML in the view.html file:

```html
<!DOCTYPE html>
<html>
<head>
  <meta name="viewport" content="user-scalable=no,initial-scale=1.0,maximum-scale=1.0" />

  <style>
    body { margin: 0px; }
  
```
Getting Started

If you prefer, you can use the downloadable code.

4. Open the view.html file in your desktop Safari browser. You should see 300-by-300-pixel square, filled with a random color.

5. Click the square several times and verify that the color changes to another random color each time you click.

How It Works

This HTML file introduces some of the boilerplate code that you will use throughout this book to define the user interface for your HTML5 mobile apps. The first line indicates to the browsers that this page uses HTML5:

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

Using this document type declaration is the standard way to start an HTML5 web page.

Next, you open the HTML file in the usual way, with the standard html and head tags. You use the viewport metatag to define the screen dimensions of the app:

```html
<meta name="viewport" content="user-scalable=no,initial-scale=1.0,maximum-scale=1.0" />
```
The mobile device’s browser uses the viewport metatag to scale the page. When you visit a normal website on your mobile device, you use a two-finger pinch gesture to zoom in and out. This is not something you want to happen with your mobile app because it is not a website. To prevent this zooming behavior, you use the viewport metatag to specify a series of special settings for the mobile device. In this case, the user-scalable=no setting disables the zoom, the initial-scale=1.0 setting makes your app occupy the entire width of the mobile device screen, and maximum-scale=1.0 prevents any automatic scaling. These settings have no effect at the moment because you are working on the desktop version of Safari.

You define the user interface of this app by using HTML, so you can style it with CSS in the normal manner. The style tag sets the size and position of the square div that holds the color, which is initially set to red (#f00):

```html
<style>
  body { margin: 0px; }
  #tapper {
    margin: 10px;
    width: 300px;
    height: 300px;
    background-color: #f00;
  }
</style>
```

Next comes the script tag. First, you need a little utility function to help generate random colors. The hex function generates a random number between 0 and 15 and then returns the hexadecimal digit for that number — one of the characters in the string “0123456789abcdef”:

```javascript
function hex() {
  var hexchars = "0123456789abcdef";
  var hexval = Math.floor(16 * Math.random());
  return hexchars[hexval];
}
```

The hex function uses the simple trick of listing the hex digits in ascending order in a string and using the random number as an index for the character position in that string. So if the random number is 8, the character at position 8, namely "8", is returned. If the random is 15, the character at position 15, namely "f", is returned.

Now you need to react to the click or tap on the color square. You use the window.onload event handler to make sure the page has fully loaded before you try to find any elements in it:

```javascript
window.onload = function() {

Then you use the document.getElementById function to get the square color div, using the identifier you have given it: 'tapper':

```javascript
var tapper = document.getElementById("tapper")
```

Once you have the color div in the tapper variable, you use the onclick event handler to detect clicks on the div:

```javascript
tapper.onclick = function() {
```
Now you change the color. You use the `hex` function three times (once for each of the red, green, and blue color components) to generate a new random color and then set the background of the `div` to this new color:

```javascript
  tapper.style.backgroundColor = "#" + hex() + hex() + hex();
```

This is all the code logic you need. Now you can close your functions, the `script` tag, and the `head` tag:

```
  }
  }
</script>
</head>
```

The body of the HTML page defines the color `div`, setting the identifier of the `div` to "tapper" so that you can find it by using `document.getElementById("tapper")`.

```
<body>
  <div id="tapper"></div>
</body>
```

Finally, you close the HTML document:

```
</html>
```

Apart from the `viewport` metatag, this is all very standard HTML, CSS, and JavaScript. To view the app by using desktop Safari, you just load the `view.html` file directly. You can’t do this with your mobile device because there is no way to load the file. You have to instead access the file by using a web server. The next section shows you how to do this.

---

**Introducing the nginx Web Server**

As you develop mobile apps, you’ll need to test them on your mobile device. When you build HTML5-based apps, the device-testing process can actually be faster than the process of developing native apps. You can reload an app in the web browser on the device, which takes just a few seconds!

You can access an app on your mobile device by requesting it from a local web server running on your desktop development machine. To do this, you need to make sure that both your mobile device and your desktop machine are on the same local network. The easiest way to ensure this is to connect both to the same Wi-Fi router.

To deliver the HTML files for your app, you have to run a local web server. The web server used in this book is the nginx web server, available from [http://nginx.org](http://nginx.org). This server is extremely fast and lightweight — perfect for cloud servers. It is also designed to handle large numbers of users. The nginx configuration file is also very easy to work with and has a simple syntax.
Installing nginx on a Mac

To install nginx on a Mac, you need to use the MacPorts installer system. The MacPorts installer lets you install UNIX command-line tools and servers on your Mac by using simple one-line commands. You can go to the www.macports.org site and click the Installing Mac Ports link on the left. Then you download the dmg file for your version of Mac OS X: Lion, Snow Leopard, or Leopard. Then you open the dmg file and install MacPorts by double-clicking its icon and following the onscreen instructions.

Next, you open the Terminal app, which you can find in the Utilities subfolder of the Applications folder. You need to make sure the MacPorts installer system knows about the latest version of nginx. To do this, you run this command (and then enter your password):

```
sudo port -d selfupdate
```

You see many lines of output, indicating the status of the update process. Then you can install nginx by running this command:

```
sudo port install nginx
```

Again, you see some lines of output that provide feedback on the installation process. When it completes, you are ready to start using nginx, and you can skip ahead to the “Starting and Stopping nginx” section.

Installing nginx on Windows

The nginx website, http://nginx.org, makes Windows binaries available for download. All you have to do is download the zip file from the site and unzip it into a convenient location. Then you can skip ahead to the “Starting and Stopping nginx” section.

Installing nginx on Linux

There are a number of different Linux distributions and a number of different ways to install server software on Linux. This book shows you how to use one of the most popular distributions, Ubuntu, which is available from www.ubuntu.com. When you set up your Amazon cloud servers, you will also use Ubuntu.

On Ubuntu, installing nginx is very simple: You use the built-in package manager. You run this command (and enter your root password):

```
sudo apt-get install nginx
```

Depending on the configuration of your machine, this installation process may start nginx for you. If it does not, you can follow the instructions in the next section.

Starting and Stopping nginx

Before you start nginx, you need to make sure there are no other web servers running. If there are, they will be occupying port 80 on your machine, the network port number used for HTTP,
and nginx will not be able to run. To test whether there is a web server running on your desktop machine, you open a web browser (such as Safari) and visit http://localhost. If a web page appears, then you know you have another web server running, and you'll need to stop that web server first. If not, you can proceed to start nginx, as described below.

If the web server already running on your machine is Apache, then you can stop it on the command line by using this:

    apachectl stop

If you are on a Windows machine, you can usually find an Apache administration application on the Start menu, and it should have a Stop button.

If you are using Windows, you might find that the Microsoft web server, known as Internet Information Server (IIS), is running. You can stop IIS by opening its administration application, which is in the Control Panel.

If you have previously installed a different web server, you should refer to its documentation to find the procedure to shut it down. You can always start it again after you have finished your app development.

To start nginx on your Mac or Windows machine, you go to the command line (on Windows you need to \( cd \) to the folder containing nginx) and run this command:

    nginx

To stop nginx, you use the following command:

    nginx -s stop

On Linux, the commands are slightly different. This is the start command:

    sudo /etc/init.d/nginx start

And this is the stop command:

    sudo /etc/init.d/nginx stop

And that’s it! To verify that nginx has started, you should reload http://localhost in your web browser. You should see the text "Welcome to nginx!".

Using nginx

You’re going to use nginx to serve up the HTML files for your app. The nginx web server has a default html folder, where you can save HTML files. Any HTML files saved to this folder will be served by nginx. You’ll copy your view.html file into your nginx html folder so that the URL http://localhost/view.html will deliver your view.html file.

The location of the nginx html folder depends on your system. On a Mac, it will be /opt/local/share/nginx/html. On Windows, it will be C:\nginx\html if you installed nginx in your C: drive. On Linux, it will /var/www/nginx-default.
Once you have located the nginx html folder, you need to copy your view.html file into it. Then you can access this file in your web browser by using http://localhost/view.html. You should do this now to verify that everything is working.

There is one last step to viewing the app on your device. From your device, you cannot use localhost as the website address because that would refer to the mobile device itself. Instead, you must use the Internet Protocol (IP) address of your desktop development machine. You’ll need to determine your local IP address.

Your local IP address is not the same as your public address, which is the address your machine or network has on the public Internet. Rather, your local IP address is an address used internally on your local network. If you are on an IPv4 network, this IP address starts with 192.168., or 10., or 172.16. If you are on an IPv6 network, this address starts with fc00:.

The approach to finding your local IP address depends on your system. On a Mac, you click the apple icon on the top-left of the menu bar and select About This Mac. Then you click the More Info button in the small summary window that appears. This opens a larger window with detailed information. Next, you click the Network item on the left and look for your IP address on the local Wi-Fi network in the AirPort item description.

On Windows, you open the list of network connections from the Start menu and double-click the Wireless Network Connection item. Then you click the Support tab of the small window that appears. Your local IP address will be shown.

On Linux, you use the ifconfig command to print out quite a few lines of information. You should look for the IP addresses that begin with the prefix numbers mentioned above, such as 192.168.

Next, you load your app by using your local IP address. In your desktop Safari browser, you open the URL http://YOUR_IP_ADDRESS/view.html, replacing YOUR_IP_ADDRESS with your local IP address. You should see your app appear as before.

NOTE This book uses YOUR_IP_ADDRESS to refer to the local IP address of your desktop development machine.

Now it’s time to test the app on your device. On your iPhone or Android device, you open the built-in web browser app and enter http://YOUR_IP_ADDRESS/view.html. You should see your
app appear as in Figure 1-3. As you can see from the figure, my local IP address is 192.168.100.112.

Tap the color square with your finger. It will change to a new random color. Congratulations! You have created your first mobile web app!

SUMMARY

This chapter introduced you to hybrid mobile apps, which are native apps built with an HTML5 container as their entire user interface. You were also introduced to JavaScript object literal syntax and the hidden power of JavaScript functions. Using these new skills you built a simple but complete mobile web app and debugged and tested this app using the Safari web browser. Finally, you installed the nginx web browser and delivered your mobile web app direct to your iPhone or Android device. At this point you have almost covered the complete mobile web app development cycle.

In the next chapter, you’ll learn how to make your app respond to finger touch events, a critical feature of any mobile app. You’ll learn how to give your app a more native look and feel. You’ll also learn about “tracer bullets,” a great technique for speeding up your development of new code. Finally, you create your first cloud service, live, on the Amazon cloud.

EXERCISES

1. In the sample code for the simple app you built in this chapter, you listened for a click event to change the color. Although this works on a mobile device, it is less responsive than using touch events. Modify the code to use the `ontouchstart` event instead.

2. Some of the apps you develop will need to be able to detect whether the mobile device is in portrait (vertical) or landscape (horizontal) orientation. You can use the `window.onresize` event to detect orientation changes. Add a listener for this event and turn the square into a rectangle when the device is moved into landscape orientation. Turn it back into a square when the device returns to portrait orientation.

3. When a user first visits your mobile web app, the address bar of the mobile browser remains visible, reducing the amount of space available for your user interface. For this exercise, overcome this by scrolling the page up by one pixel.

4. For the most part, you’ll want your mobile apps to remain at the same fixed zoom level because this is how native apps behave. However, there are occasions when you might want to change this and take advantage of the native zooming behavior — for example, if you are displaying a large chart. Experiment with the scale setting in the `viewport` metatag to see if you can start the app zoomed out.

Answers to the Exercises can be found in the Appendix.
## WHAT YOU LEARNED IN THIS CHAPTER

<table>
<thead>
<tr>
<th>TOPIC</th>
<th>KEY CONCEPTS</th>
</tr>
</thead>
<tbody>
<tr>
<td>HTML5</td>
<td>The latest version of the HTML standard, produced by the World Wide Web consortium (W3C). This is the most important web standard to be released in recent years. It defines a new set of capabilities for applications built using HTML. You can now detect the location of your user, store data locally, and even generate complex graphics. The enhanced interactivity and dynamic interface features such as animation also enable you to build highly visual, user-friendly, and engaging user interfaces using HTML5.</td>
</tr>
<tr>
<td>Dynamic JavaScript Functions</td>
<td>The JavaScript language is more powerful than it might first appear. Because it supports a functional programming style through the use of dynamic functions, you can develop complex systems without code bloat. Learning the basics of the functional programming style can help you increase your code reuse significantly and enable you to become far more productive as a programmer.</td>
</tr>
<tr>
<td>WebKit</td>
<td>This open source browser engine has become the de facto standard for mobile web app development because it is used on both iPhone and Android, as well as other less popular platforms. The WebKit engine has string support for HTML5 and excellent support for debugging and app development. Because you can use the desktop version of WebKit, in the form of the Safari web browser, to test and develop your mobile app long before you reach for a physical device, your development work cycle is much faster and shorter.</td>
</tr>
<tr>
<td>nginx</td>
<td>This web server is specifically designed to handle large numbers of requests very efficiently. This makes it an excellent choice for cloud hosting. You’ll use the nginx web server throughout this book to host mobile web apps and to provide a front end for your service interfaces.</td>
</tr>
</tbody>
</table>