2
Setting up the Environment

Before we can start using some of the new JavaScript features of HTML5, we'll need to set up our environment. This is needed because of the security architecture built into HTML5, which restricts the way our code can run, what it can access, and so forth. After we have a local web server running, we'll be ready to jump right in, and start developing our games. However, we'll take one final stop before we get to that, and we'll develop a portal from which we can access our games.

While the first part of this chapter is absolutely essential for the rest of the work we'll cover in the book, the second part is completely optional. What we'll do is, use the semantic HTML5 tags to build a mini website. Since none of the games we'll build in this book have a real need for most of the semantic elements that make up a fundamental part of HTML5, this seems like the perfect opportunity to use them in a way that makes sense, and that is also fairly useful.

The purpose of this portal page that we'll build, is to centralize access to our games other than to better understand the new semantic elements and give us some practice with them. Each game will be described in this page, and will give you access to them. Put it simply, this page will be a catalog that shows off your games, and allows you to share them with, and impress your mates!

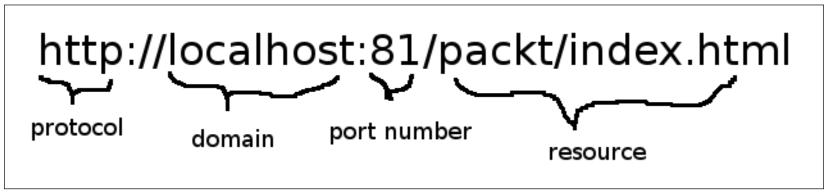
Using a web server

Although JavaScript is fairly a powerful programming language, given its role and purpose, it is somewhat limited in what it can and cannot do. For example, unlike Java or C++, you cannot write data directly to the file system (such as saving an image or a text file to a particular directory in the user's machine). However, just being able to read data from the user's machine could be a tremendous security breach. In order to mitigate this problem, JavaScript is not allowed to request data from the user's file system, which could be done using the `file://` protocol.

While most web browsers will properly render an HTML document independent of where its being served from, or with what protocol the file was fetched; many JavaScript API calls forbid requests to assets through the `file://` protocol. On top of this restriction, there is also a policy that forbids a web page to access assets from a different origin as itself. This policy, officially known as the Same-Origin Policy, basically states that the script requesting a resource needs to be in the same domain and port as the resource being requested, and the request must be made through the same protocol as it was loaded. Any requests that violates this policy will result in a `NS_ERROR_DOM_SECURITY_ERR` error.

The Same-Origin Policy checks that the request and the requester have the same protocol, domain, and port. The reason many JavaScript APIs fail even though they comply with the Same-Origin Policy is, because other policies may apply, such as allowed protocols. Thus, loading files with the `file://` protocol may limit your use of HTML5 features.

If a domain leaves off the port number, it is assumed by the browser to be the default port for websites, which is port 80.



The path to a server resource in many HTML5 APIs must match the protocol, domain, and port number of the script making the request using a web server

In order to make our games work properly and obey the Same-Origin Policy, we'll need to serve all our files from a web server. The good news is that there are many free, light-weight server that we can install locally and use for this purpose. If later you decide to publish your games so that others may access them from the internet, and play them, you'll need to copy the files to a web server that people can access. How to do that is beyond the scope of this book, but information about it is readily available online. For our purposes, we'll simply install an Apache web server that, by default, can only be accessed by you from your computer where the server is installed.

The following sections will guide you on how to install the Apache HTTP Server (httpd) on operating systems such as Microsoft Windows, Mac OS, and Linux. We'll use the popular *AMP stack, which is a solution stack using **open source software** (**OSS**) especially designed for web development. The stack includes an Apache server, a MySQL database system, and a server-side programming language, which nowadays is normally PHP. The first letter of the acronym (WAMP, MAMP, or LAMP) refers to the operating system in question (Windows, Mac, or Linux). The "P" can mean PHP, Python, or even Perl, although PHP is the most common of the three.

For Microsoft Windows developers, another commonly used server is **Internet Information Services** (**IIS**). In terms of raw power and performance, both IIS and Apache are very compelling servers, where neither is particularly better than the other. However, in order to stay consistent with other platforms, we will only refer to Apache servers in the sections to come.

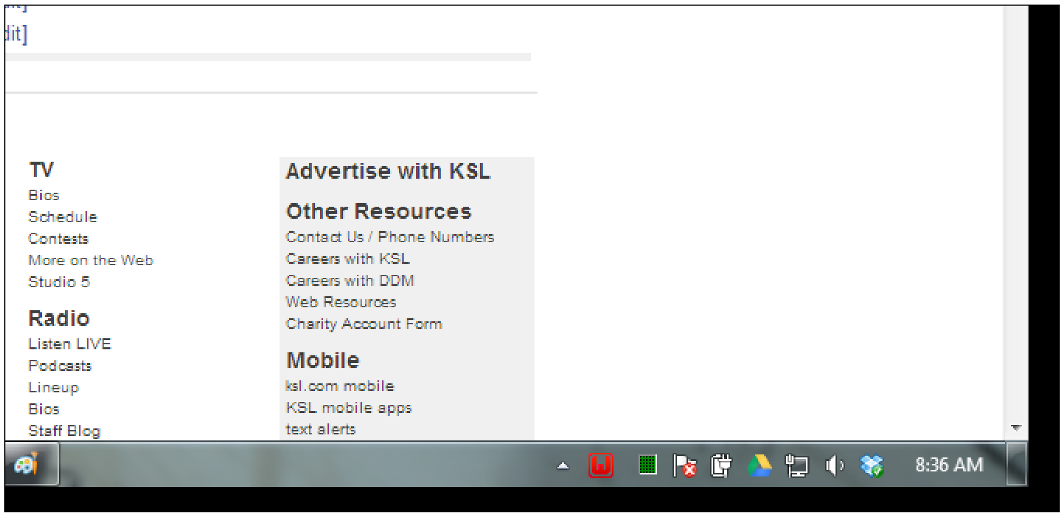
Keep in mind that any server-side programming is strictly beyond the scope of this book, although the default *AMP installation that we'll setup next will allow you to jump right in, and start writing PHP code.

Setting up WAMP in Windows

The best place to get the WAMP software bundle, is from its official website, that is, `http://www.wampserver.com/en/`. Look for the download section, and grab the version that is appropriate for your system. If there are options for different versions of Apache, MySQL, or PHP, just grab the latest one available. Since we won't make use of any features of Apache beyond its most crude functionality, it really doesn't matter one bit what you chose.

The installation process is pretty basic, as you will follow a visual wizard style installation.

1. First you'll be asked to accept a license agreement, then you'll be asked to select an installation destination, and that is it. I recommend you choose your primary drive as the installation destination (`c:` in my case), so it's easier to follow the rest of this guide.
2. Once the installation is complete, you'll see the WAMP icon in the system tray. This icon turns different colors indicating its status. If the icon is red, it means that the server is offline. If it's green, it means that all services are online and ready for use.

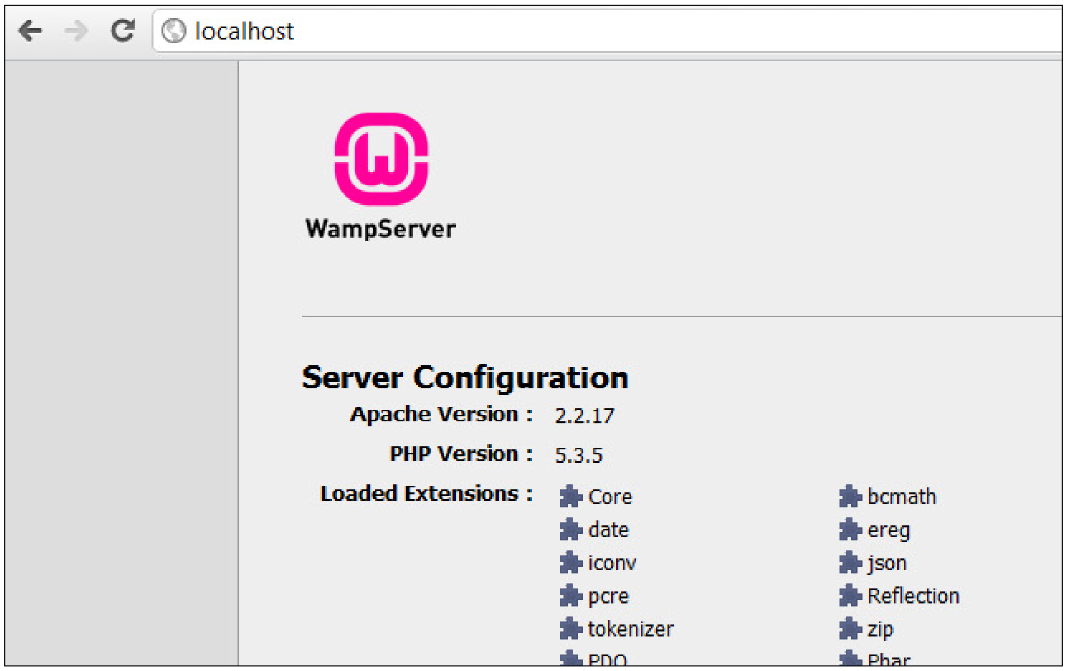


The red WAMP icon indicates that the server is not yet online. Be sure to click the icon and start the server before trying to use it.

Next, pull up your favorite modern browser and navigate to the following URL, just to make sure things are indeed working as they should:

```
http://localhost
```

The default message used by Apache to show that the server has been installed and configured properly is remarkably intuitive and easy to understand, as shown in the following screenshot:



Server Configuration

Apache Version : 2.2.17

PHP Version : 5.3.5

Loaded Extensions : Core, date, iconv, pcre, tokenizer, PDO, bcmath, ereg, json, Reflection, zip, Phar

The preceding screenshot shows the default page for a standard WAMP installation. This screen might differ based on the version of WAMP you use.

Finally, we'll need to set up our project files and directories, so that we can see more interesting things than just Apache's welcome message. To do so, go over to the following directory, which is where Apache looks for when you request the URL for localhost:

```
c:/www
```

Here is where we'll be placing all of our files. The first thing we'll do here is create a directory named `packt`, which will be the root of our local website. Any files we create will be placed inside this folder, as well as any sub projects we'll be creating. Your directory structure should now look similar to the following path:

```
c:/www
```

```
c:/www/packt
```

Now when you want to execute any files stored in our project directory, we can do so by accessing the following URL from a browser:

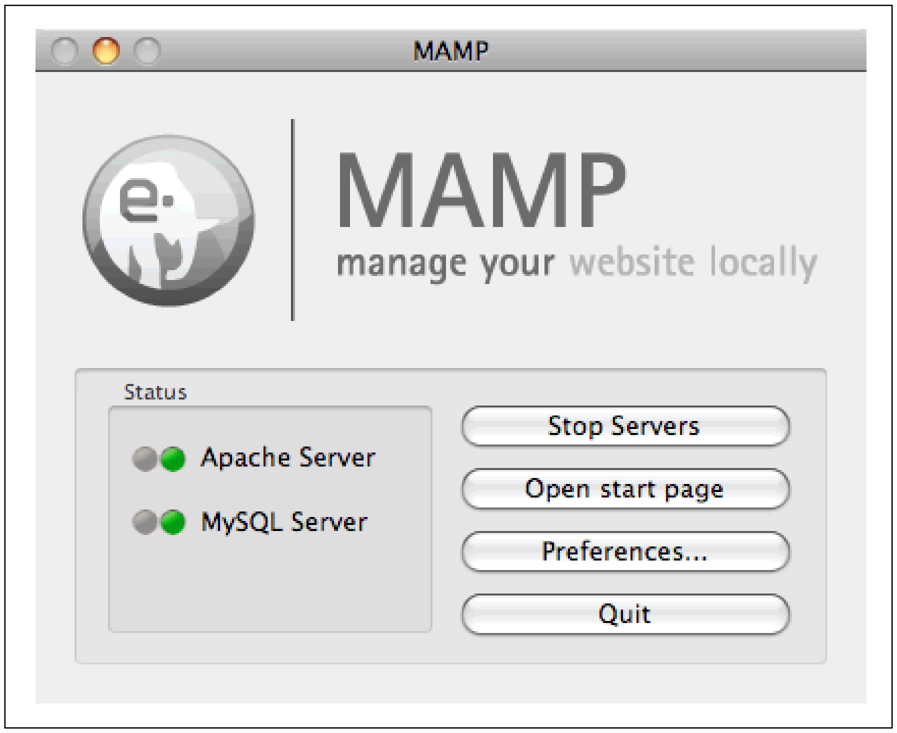
`http://localhost/packt`

Congratulations, you have successfully set up your development environment, and are now ready to start programming awesome things in HTML5!

Setting up MAMP in Mac

The best place to obtain the MAMP software is from the official website, `http://mamp.info/en/index.html`. Once you have downloaded the software into your hard drive, run the file and agree to the user agreement that accompanies the program. Follow any other instructions that show up on your screen, leaving any possible options as the default selection.

Once the installation is complete, running MAMP will bring up the following screen:

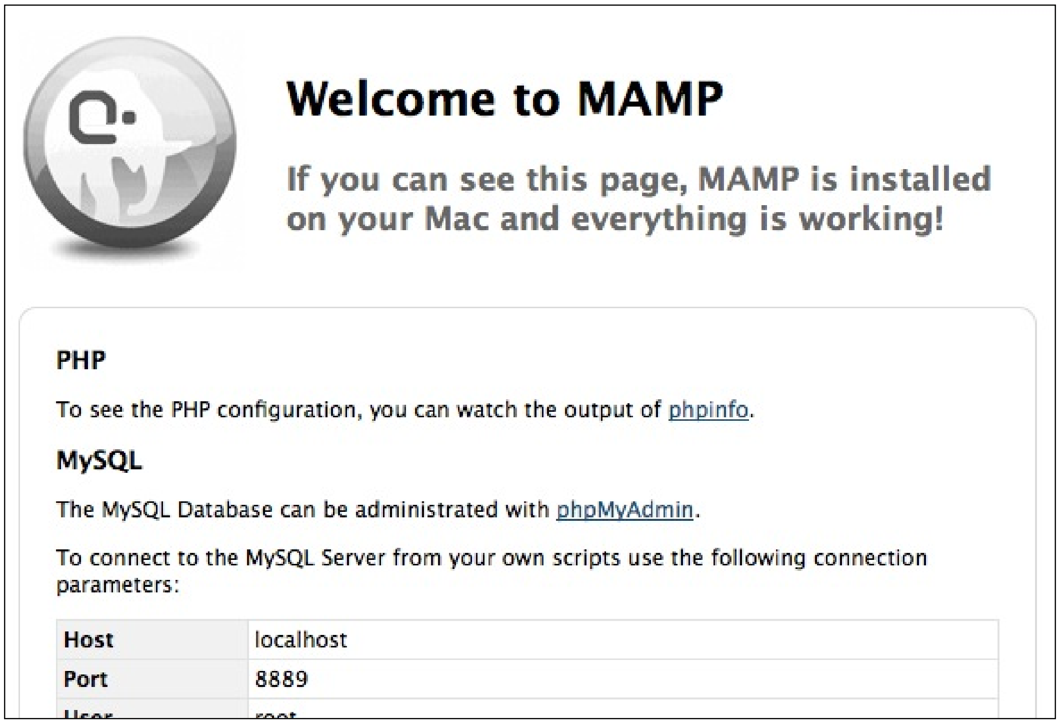


The preceding figure shows the main MAMP control, from where you can start and stop the server, and configure any settings related to the server environment.

Next, start the server and pull up your favorite modern browser and navigate to the following URL, just to make sure things are indeed working as they should:

```
http://localhost
```

The default message used by Apache to show that the server has been installed and configured properly is remarkably intuitive and easy to understand, as shown in the following screenshot:



Welcome to MAMP

If you can see this page, MAMP is installed on your Mac and everything is working!

PHP

To see the PHP configuration, you can watch the output of phpinfo.

MySQL

The MySQL Database can be administrated with phpMyAdmin.

To connect to the MySQL Server from your own scripts use the following connection parameters:

Host	localhost
Port	8889

This shows the default page for a standard MAMP installation. This screen might differ based on the version of WAMP you use.

Finally, we'll need to set up our project files and directories, so that we can see more interesting things than just Apache's welcome message. To do so, go over to the following directory, which is where Apache looks for when you request the URL for localhost:

```
/Applications/MAMP/htdocs
```

Here is where we'll be placing all of our files. The first thing we'll do here is create a directory named `packt`, which will be the root of our local website. Any files we create will be placed inside this folder, as well as any sub projects that we'll be creating. Your directory structure should now look similar to the following path:

```
/Applications/MAMP/htdocs
```

```
/Applications/MAMP/htdocs/packt
```

Now when you want to execute any files stored in our project directory, we can do so by accessing the following URL from a browser:

```
http://localhost/packt
```

Congratulations, you have successfully set up your development environment, and are now ready to start programming awesome things in HTML5!

Setting up LAMP in Linux

Installing software in Linux can be the easiest thing in the world, or it can be a terrible nightmare if you're not accustomed to the way Linux works. Since we won't be using a database system, nor any server-side programming, we'll simply install httpd so that we can run any JavaScript that we want, and the chances of you running into problems installing multiple packages will be decreased.

The easiest way to get Apache installed in Linux is through the package manager. However, chances are that you already have Apache HTTP Server installed in your machine unless you deliberately removed it. The easiest way to test if you need to perform the installation or not, is to actually attempt to install it. If your system already has it installed, it will simply let you know about it, and not install it.

To install httpd, simply open a terminal window and execute the following command (be sure to use whatever package manager your specific distribution of Linux uses. The following example shows Ubuntu's `apt-get` command):

```
sudo apt-get install apache2
```

If all goes well, you'll see no errors explaining why the installation didn't succeed. If that's the case, just follow whatever suggestions or instructions the messages direct you to do.

Once the installation is complete, or in case you already had httpd installed, we'll need to run the following command, just to make sure the server is in fact up and running in case it's not already:

```
sudo service apache2 start
```

Again, if the server was already running, then the system will simply inform you about it and continue its business as usual.

Beware of tasksel

There is a very simple way to install the entire LAMP server in Linux using a program called Tasksel. The way it works is really quite simple. You install tasksel with the following command:

```
sudo apt-get install tasksel
```

Once tasksel is installed, you can use it to install other packages. The benefit of using this tool is that it will install a group of tools together as one single unit for you. For example, instead of you manually installing Apache, MySQL, and PHP; tasksel would give you the option to install a LAMP server, where it would install all the individual components for you.

While that may seem like a convenient way to install LAMP, beware that really bad things can happen when you use tasksel. The reason for this is that the user interface for taskel is a classic definition of a poorly designed, very dangerous interface.

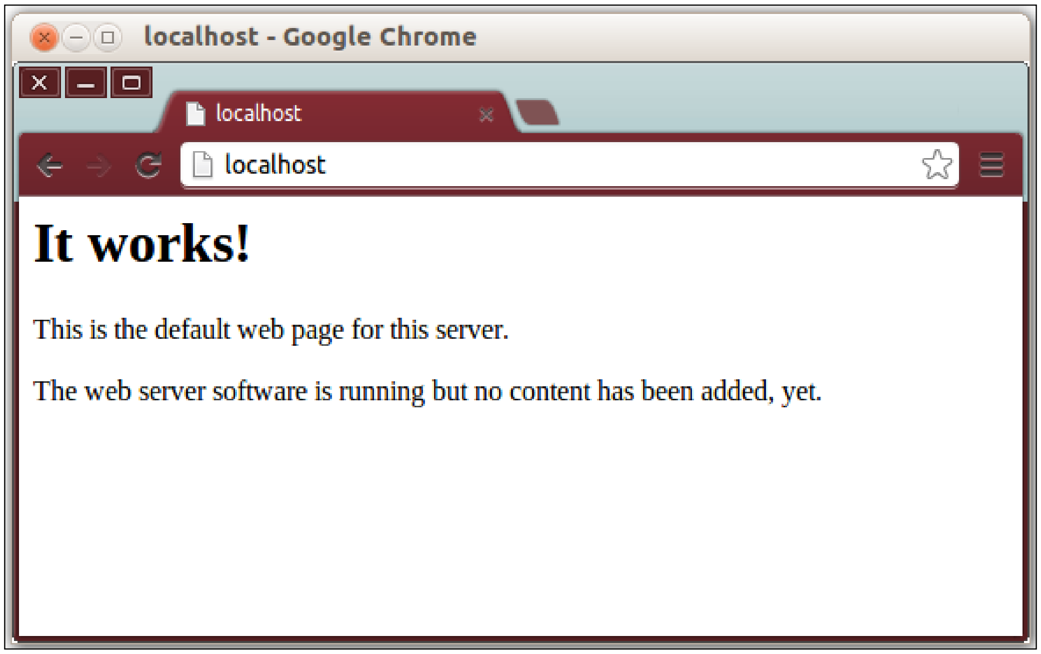
When you try to install a tool with tasksel, the interface will ask you which of the tool(s) you'd like to install. However, you will notice that some tools (such as Ubuntu desktop, for example) will already be checked, implying that those tools will be downloaded and installed. Sadly, if you make the common mistake of unchecking a tool from the user interface (thinking that by unchecking the tool from the list of tools to be installed, you'll simply be ignoring the download and installation of said tool), then tasksel will, without any warning or confirmation, proceed to uninstall those tools and all of their dependencies. Most of the time, this massive uninstallation will render your system unstable, since core components may be removed in the process.

Remember, in the tasksel interface, unchecking an item means that you want to uninstall an item.

Next, pull up your favorite modern browser and navigate to the following URL, just to make sure things are indeed working as they should:

```
http://localhost
```

The default message used by Apache to show that the server has been installed and configured properly is remarkably intuitive and easy to understand, as shown in the following screenshot:



It works!

This is the default web page for this server.

The web server software is running but no content has been added, yet.

This is the default page for Apache's httpd server.

Finally, we'll need to set up our project files and directories, so that we can see more interesting things than just Apache's welcome message. To do so, go over to the following directory, which is where Apache looks for when you request the URL for the localhost:

```
/var/www
```

Here is where we'll be placing all of our files. The first thing we'll do here is create a directory named `packt`, which will be the root of our local website. Any files we create will be placed inside this folder, as well as any sub projects we'll be creating. Your directory structure should now look similar to the following path:

```
/var/www
```

```
/var/www/packt
```

Now when you want to execute any files stored in our project directory, we can do so by accessing the following URL from a browser:

```
http://localhost/packt
```

Congratulations, you have successfully set up your development environment, and are now ready to start programming awesome things in HTML5!

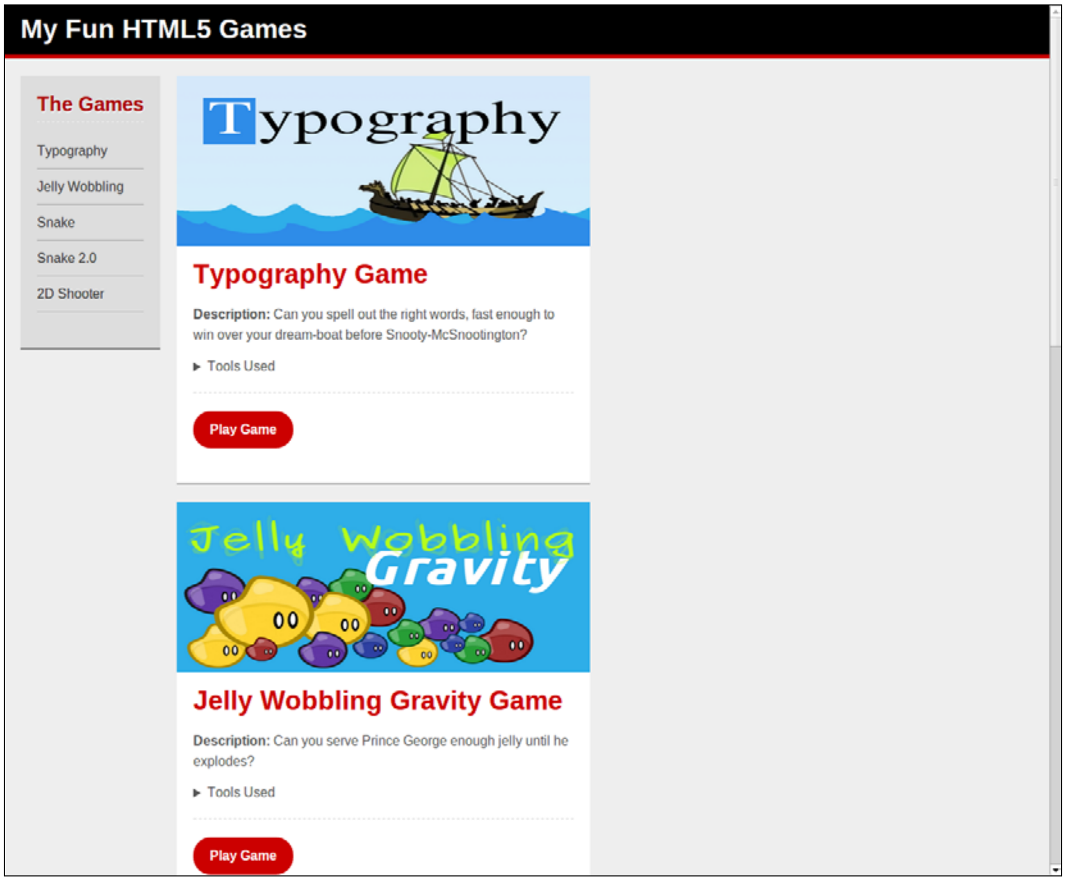
HTML5 semantic tags

As mentioned earlier in the chapter, this final section will focus on building a central portal type website, where we'll take the opportunity to learn more about, and get some practice using some of the new HTML5 semantic tags. The main idea of this book is to help you learn HTML5 by programming fun games, which just so happen to be a great concept where most of the new JavaScript features fit so well. However, we couldn't think of a whole lot of fun games that could make great use of these DOM elements, so that's why we decided to build this portal. All in all, a lot of HTML5 relies on these structural tags, which depending on how you use HTML in your day to day work, might be your bread and butter anyway.

Keep in mind, though that this will by no means be a lesson in web design. We won't be going over hardly any design theory or even best practices. If you would like some more guidance and direction on how to design beautiful websites, I suggest you continue your research after this book. The goal here is to show you how the new semantic elements are meant to be used within the context of the DOM, and nothing else.

The web portal

The following screenshot illustrates what the final product will look like. It'll be simple, yet arguably decent looking. If we had decided to make it really fancy, it probably would impress you more, but would also make it very difficult for you to follow along, and unlikely that we'd be able to efficiently copy it from the book alone. A standard two column layout, with a left-side navigation is shown in the following screenshot:



Typography Game

Description: Can you spell out the right words, fast enough to win over your dream-boat before Snooty-McSnootington?

▸ Tools Used

Jelly Wobbling Gravity Game

Description: Can you serve Prince George enough jelly until he explodes?

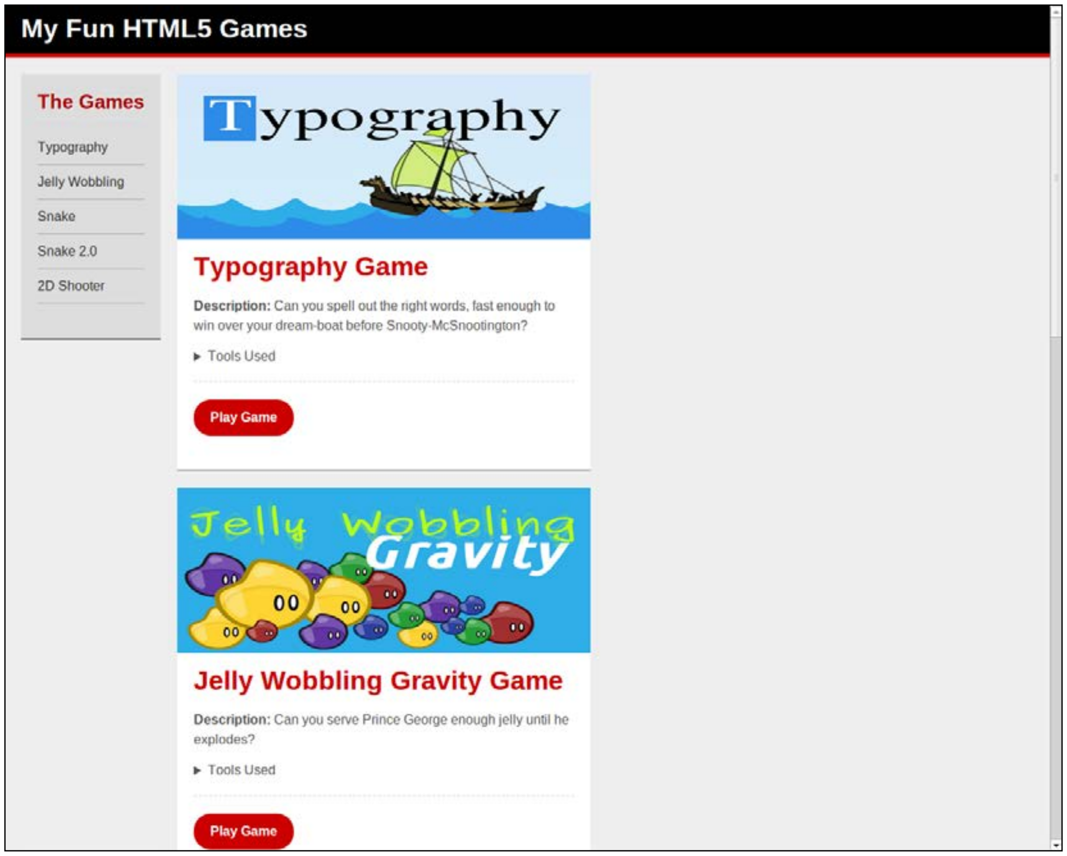
▸ Tools Used

We can break down the layout into four separate pieces, namely the top header, left navigation column, the main content section, and the footer. By dissecting the design into its most basic components, it becomes easier to understand and conceptualize the complete design, which also makes it easier to implement it, troubleshoot it, and maintain it.

First we'll go through each piece of the design, explaining what its purpose is and how it contributes to the rest of the website. Then we'll look at the tags being used in the layout, putting together the code that brings this design to life.

Top header

The purpose of the top header is to tell the user what this website is about, what it is called, and possibly what it does. The header is normally a very strong piece in establishing and supporting the website's brand identity. A high quality logo and a strong and direct tag line are common components usually found in a top header, but in our case, in the interest of simplicity, we simply included a generic text title meant to illustrate this point.



Typography Game

Description: Can you spell out the right words, fast enough to win over your dream-boat before Snooty-McSnootington?

Jelly Wobbling Gravity Game

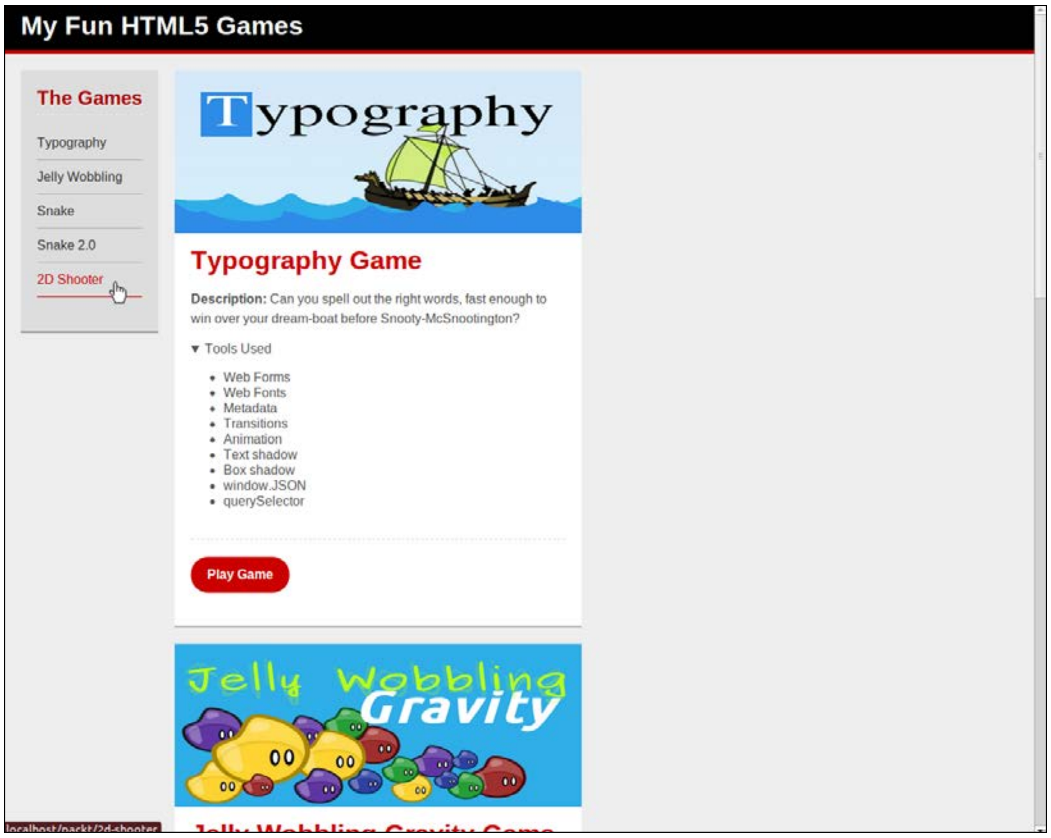
Description: Can you serve Prince George enough jelly until he explodes?

The top header tells the user about the website he or she is visiting.

Navigation column

There are many ways to implement a navigation column. Some times the navigation is found at the top of the page together with the top header, some times it's found on either the left or right side of the page. Either way, the purpose of the navigation container is to help the user know where he or she is within the website, where they can go next, and what they can expect from the website in terms of available content.

Our navigation column simply lists the games we'll be building, so that we can quickly get to the games and start playing right away. Clicking on a link from the navigation area, as expected will take us straight to the game. In order to make the presentation slightly more colorful and cheerful, we added a hover effect, which simply changes the color of the text and its bottom margin.



Description: Can you spell out the right words, fast enough to win over your dream-boat before Snooty-McSnootington?

▾ Tools Used

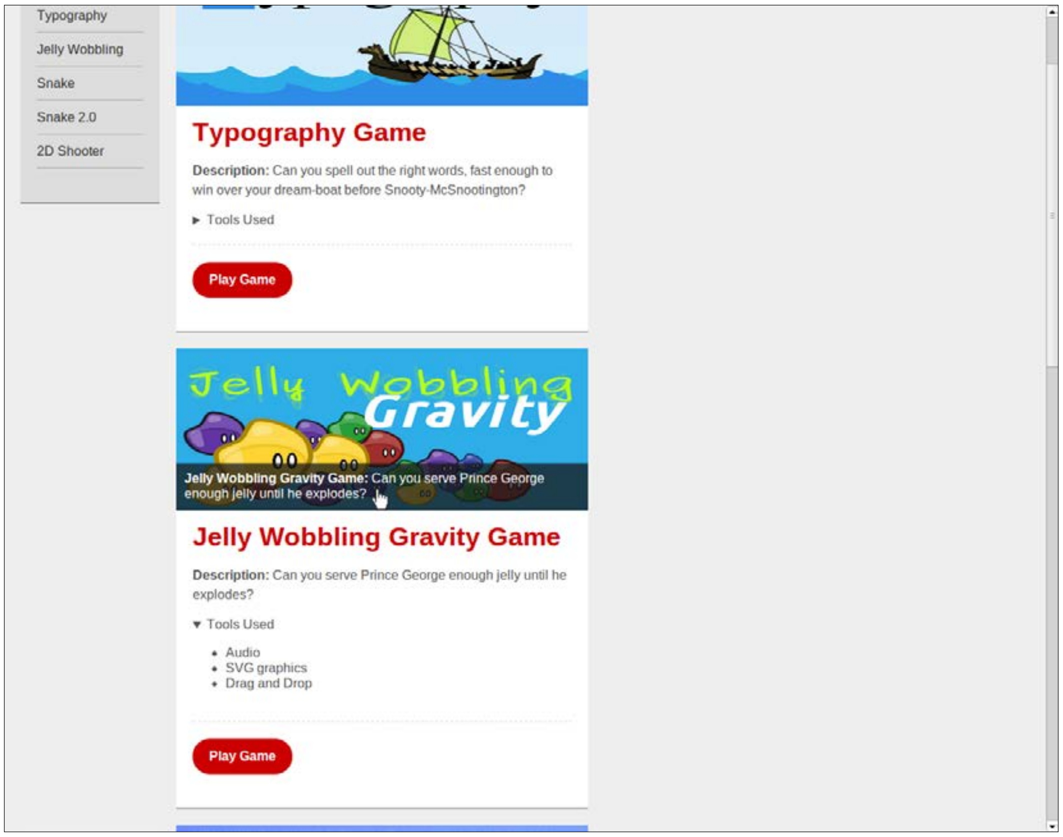
- Web Forms
- Web Fonts
- Metadata
- Transitions
- Animation
- Text shadow
- Box shadow
- window.JSON
- querySelector

Each link gets a high contrast highlight when hovered over by the mouse cursor, so the user has no doubts about where to click. If there were more pages than this one, the link corresponding to the current page should be highlighted in order to communicate to the user what the current page is.

Main content section

Here is where the meat of the page is found. Although our example is simple, the main content section is the reason that people do and will visit your website. With that in mind, HTML5 makes content publishing much more effective by the way it structures the main content area of a web page. This makes it easier for web spiders to crawl your content and determine what your page is about, as well as screen readers, and syndication software.

In our portal, the main section will be populated by several game cards, which describe the game in more detail than the navigation menu by adding a large, high quality screen shot or poster of the game, along with some description of what the game is, how it's played, and what tools were used to build the game. There is also a link that takes the user directly to the game. The main reason for the **Tools Used** section, as you'll see in a moment, is to show one of the most exciting tags added to HTML5, that is, `details`. This tag adds very useful behavior to the page, which up until this point, was only available through many lines of JavaScript.



Typography Game

Description: Can you spell out the right words, fast enough to win over your dream-boat before Snooty-McSnootington?

▸ Tools Used

Jelly Wobbling Gravity Game: Can you serve Prince George enough jelly until he explodes?

Jelly Wobbling Gravity Game

Description: Can you serve Prince George enough jelly until he explodes?

▾ Tools Used

- Audio
- SVG graphics
- Drag and Drop

Note the two subtle effects we have added here, all using native HTML5 functionality. The first is the collapsible details container, which shows and hides itself as it gets clicked subsequent times. The other is the figure caption, which shows itself when the figure is hovered, and hides itself away when the mouse leaves the figure that it belongs with. Although this particular feature is not necessarily new or unique to HTML5, it is exciting that the effect can be achieved with no more than two lines of CSS code.

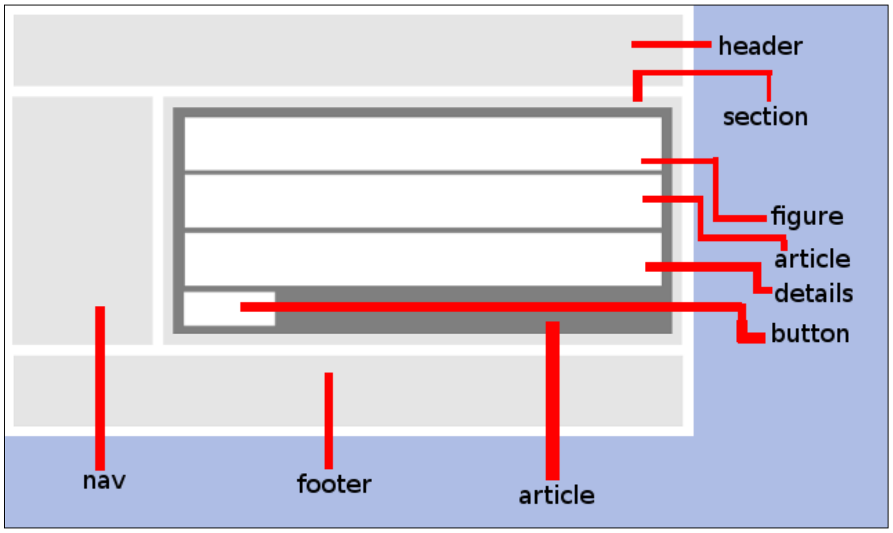
The footer

Another standard component in just about every website out there is the footer. It is meant to clearly show the end of the current web page, provide some copyright and contact information, as well as quick access to the main links pertaining to the site and some other important, yet not primary links, such as links to a privacy policy page, terms of use, and other legal documentation. In our case, we just display a standard copyright notice, indicating our ownership of the content we have created.

Again, notice that most of the content is symbolic, and only serves the purpose of illustrating the use of the new tags used to display said content.

Coding our portal

Now that we have seen a high level overview of what we'll be creating, along with a technical overview of each building block, we're ready to dive in and start coding. Let the fun begin!



This is a wireframe of the structure we'll develop. Note that the `article` element directly inside the `section` tag is representative of a single game card widget. Thus, the entire structure it represents will be repeated for each game that we'll be showing off in our portal page.

The doctype

The first thing you'll notice when looking at an HTML5 document is a new doctype. I have always found it interesting that the only purpose of the doctype declaration is to tell the browser how to interpret and render the document that follows. Since its purpose is so specific, it's always been hard to understand why the declaration was so verbose before HTML5. In previous versions of HTML, one had the following three options when it came to the doctype declaration, and none of it were easy to remember:

HTML 4.01 Transitional

This declaration was used when the document that followed it used any of the new elements and attributes, but also allowed the use of deprecated elements and attributes. This mode was also known as loose mode because the browser treaded it as any other HTML document.

```
<!DOCTYPE HTML PUBLIC "-//W3C//DTD HTML 4.01 Transitional//EN"
  "http://www.w3.org/TR/html4/loose.dtd">
```

HTML 4.01 Frameset

This declaration was used when the document that followed it complied with the transitional definition, but also included frames. These were actual frames that broke the page down into individual fragments, not merely the inline frames (iframes) still in use today. Framesets have been completely deprecated in HTML5.

```
<!DOCTYPE HTML PUBLIC "-//W3C//DTD HTML 4.01 Frameset//EN"  "http://
www.w3.org/TR/html4/frameset.dtd">
```

HTML 4.01 Strict

This declaration was used when the document that followed it only used current elements and attributes. Deservingly considered the most strict of the three options, documents in this mode did a great job at moving people towards better coding practices, since HTML had always been known for its very loose and forgiving nature.

```
<!DOCTYPE HTML PUBLIC "-//W3C//DTD HTML 4.01//EN"
  "http://www.w3.org/TR/html4/strict.dtd">
```

While the doctype's only real technical purpose as far as the browser is concerned, is to determine how each element should be rendered, there is also a very important reason for these doctype declarations, that is, HTML validators. Although their value is relative, and their importance is subjective, many people take validators to be a religion.

The only true purpose for a validator is to help you test the correctness of your HTML structure (proper tag nesting, proper formatting, and strict adherence to the doctype in use). Even if you declare a document with HTML 4.01 Strict doctype, the browser would still render your document just fine if you violated one or more of the rules declared by the doctype definition.

Some people (including companies such as Google, Apple, and Microsoft, to name a few) disregard any formatting complaints from the validator because since they receive such enormous traffic, it is beneficial for them to remove as much code as possible from their HTML files, while still making them render the same in as many different browsers as possible. The less code they have to send down to each request, the less time it takes for each request to complete.

Quirks mode

If you forget to provide a doctype declaration at the top of your documents, or provide an invalid declaration, your document will be in quirks mode. While quirks mode is not an actual standard, it is a way to provide backwards compatibility with older HTML documents. The downside to serving pages in quirks mode is that different browsers may render different elements differently, as well as interpret poorly written markup code differently. Thus, a page in quirks mode may provide unpredictable rendering. For this reason, it is recommended that you always include a valid doctype declaration, preferably the new HTML5 declaration. If a browser doesn't support HTML5 features, it will still switch the rendering mode to standards mode if it sees the new doctype declaration.

HTML5

This declaration is used when the document that follows it is awesome and sophisticated, or at the very least uses HTML5 elements and attributes. You may be surprised at the simplicity of the new declaration, especially because now it is possible for us to actually memorize the declaration.

```
<!doctype html>
```

That's it. By including those 15 characters at the top of your documents, the browser will know what you want to render, and respond accordingly. Also, if a browser does not yet support all of the latest HTML5 elements and attributes (or any element or attribute at all), it will still switch its rendering mode to standard mode if it finds this doctype definition declaration.

Simplified elements

Once we have the doctype definition in place, we'll need to declare the character encoding used in our page. This is done by using a `meta` tag element, which have also been greatly simplified in HTML5. In the previous versions of HTML, the character encoding declaration was done in a rather verbose and inconsistent manner.

```
<meta http-equiv="Content-Type" content="text/html; charset=UTF-8">
```

Luckily, this has been replaced with a very friendly 17 character string (excluding the actual encoding setting), which again we're likely to actually memorize.

```
<meta charset="UTF-8">
```

While the new character encoding declaration is much shorter and easier to type in, the previous style is still a valid syntax. For those of us out there going through lots of outdated HTML documents, and slowly converting them to HTML5, note that if updating the charset declaration is too painful of a task, you will be fine leaving it as the old version.

Other commonly used elements that were changed for simplicity include the `script` and the `link` tags, used to import CSS stylesheets and JavaScript scripts. The main difference is that the `type` attribute has been made optional, since CSS is the only stylesheet language currently in use with HTML, and JavaScript is the default scripting language.

The `script` tag requires us to specify the type of script it defines, such as the following syntax:

```
<script type="text/JavaScript"></script>
```

Now we can leave out the `type` attribute if the script is indeed JavaScript, as in the following syntax:

```
<script></script>
```

Note that the `type` attribute is only optional, not deprecated. If you provide a script type other than JavaScript, the browser will simply treat the contents of the tag as some script code, and not render it, nor attempt to interpret it or execute it.

A common practice today for providing content to be used by your JavaScript that is neither a text content to be displayed by the website, nor an actual JavaScript code is to include this content inside script tags, but provide a made-up `type` attribute.

This is very common in WebGL (3D graphics rendering in HTML5, which we discuss briefly in *Chapter 7*, *Adding Features to your Game*), where the JavaScript code uses shader code, which is a code in C-like language called GLSL. The GLSL code is embedded right on the HTML code inside the `script` tag, but because of the presence of the `type` attribute, with a value other than `text/JavaScript` (normally something such as `<script type="glsl/shader"></script>`), the browser leaves that code alone.

The `link` tag, used to import stylesheet files into the host HTML page, also required a `type` attribute, as used in the following syntax:

```
<link rel="stylesheet" type="text/css" href="path-to/stylesheet.css"
/>
```

Now we can leave out the `type` attribute since the `rel` attribute tells the browser that the relationship with the incoming file to the HTML document is that of a stylesheet, and no other stylesheet type (language) is currently available other than CSS. The new declaration looks such as the following syntax:

```
<link rel="stylesheet" href="path-to/stylesheet.css" />
```

The body of our portal

Finally, let's take a look at the elements and attributes inside the `body` tag of our web page. In total, there were 11 unique tags used (not counting common HTML elements such as the image or paragraph elements) to produce this document. While that's not all of the new semantic elements, I believe this does a good job at introducing and illustrating the point.

Header

The first element you'll note in this application is header. This element is a block level tag meant to represent the header of a section. Normally the contents of a `header` tag will include one or more of the heading tags (`h1` through `h6`), paragraph tags, images, or any other elements that contribute to the mission of representing a section header.

Restrictions that apply to the header tag that you need to keep in mind include the following three rules:

- Do not nest a `header` tag inside another `header` tag
- Do not nest a `header` tag inside an `address` tag
- Do not nest a `header` tag inside a `footer` tag

The main header of our web portal is as basic as it can get, as shown in the following code snippet:

```
<header>
  <h1>My Fun HTML5 Games</h1>
</header>
```

The corresponding CSS for this header is also very basic and straight forward as in the following snippet:

```
body > header {
  background: #000;
}

body > header h1 {
  margin: 0;
  padding: 10px 20px;
  color: #fff;
  border-bottom: 5px solid #c00;
}
```

This header could have been decorated with nice graphical logos and a nice tag line. Normally, the logo element, whether represented by an image or plain text, is decorated with a hyperlink that points at the home page of the website. This is by no means a hard rule, but most users are accustomed to that behavior, and thus expect the functionality to be present.

Nav

Next up is the main navigation widget, which uses the `nav` tag. The `nav` tag is a block level element, meant to be a section with links to other documents, or to sections within its host document. Most of the time, the links inside a `nav` element are implemented using some sort of list element (either ordered or unordered). You may have multiple separate nav elements in the same document, even if two or more of them contain the exact same content.

Often you'll find a duplicate nav at the bottom of the document (normally nested inside a footer element) with the same content as the main nav element found at the top of the same document. Again, this is by no mean a hard rule, but it does provide excellent user experience, and you will find that the payoff is well worth your time and effort in creating and maintaining this additional navigation controller.

The only restriction that applies to the `<nav>` tag is the that it cannot be nested inside an `address` tag.

The main navigation of our web portal is as basic as it can get, as shown in the following code snippet:

```
<nav>
  <h2>The Games</h2>
    <ul>
      <li><a href="/packt/typography">Typography</a></li>
      <li><a href="/packt/jelly-wobbling">Jelly Wobbling</a></li>
      <li><a href="/packt/snake">Snake</a></li>
      <li><a href="/packt/snake-2.0">Snake 2.0</a></li>
      <li><a href="/packt/2d-shooter">2D Shooter</a></li>
    </ul>
</nav>
```

The corresponding CSS for this `nav` is also very basic and straight forward as shown in the following code snippet:

```
nav {
  float: left;
  padding: 20px 20px 40px;
  margin: 20px 10px 10px 20px;
  background: #ddd;
  border-bottom: 2px solid #888;
}

nav h2 {
  margin: 0 0 15px;
  color: #a00;
```

```
    text-shadow: 1px 1px 1px #fff;
    padding: 0 0 5px;
    border-bottom: 1px dashed #eee;
  }

  nav ul {
    padding: 0;
    margin: 0;
    list-style: none;
  }

  nav a {
    text-decoration: none;
    display: block;
    padding: 10px 0;
    margin: 2px 0;
    border-bottom: 1px solid #aaa;
    color: #333;
  }

  nav a:hover {
    border-bottom: 1px solid #c00;
    color: #c00;
  }
```

Notice that it is fine to include other elements inside the `<nav>` tag, especially when doing so adds meaning to the structure. In this case I chose to add a header to describe the `nav` element, which I did by using an `h2` element. Here we could even have used a header tag to more explicitly accomplish the same thing, but in order to keep the structure flat, I opted for a single heading element instead.

Section

The section element is a block level element, representing a distinct section of the document, separated and grouped by theme, such as main content section, or comments section. A section usually has a title or header, such that it can be identified and distinguished from other sections. There are no restrictions to the types of elements that may be found within a `section` tag.

Section vs. Div

Before there was a `section` element, the only generic, block level element available was the `div` element. In a way, they both seem to serve a similar purpose, namely to section out areas of a document. Thus, one might wonder what the difference between the two is, if any.

The difference between the two is very clear. A section is a generic container that groups its child elements, forming a single theme. A div, on the other hand, has absolutely no meaning. It is as simple as that. In terms of semantics, a div is a dumb, block level tag with no meaning, and should only be used as a last resort.

The only restriction that applies to the section tag is the following is that it cannot be nested an address tag.

The only section used in our web portal was used to delineate the main content area, as shown in the following code snippet:

```
<section>
  <article>
    (...)
  </article>
</section>
```

The corresponding CSS for this section is also very basic and straight forward:

```
section {
  float: left;
  padding: 20px 10px 0;
}
```

Since this is the main content area, it is designed to contain a collection of articles, which in this case, represent a game card that describes a particular game. We could also have used a section element to mark some sort of left column, and distinguish it from this main section. The `nav` element would then have been a child of this second section, but again, this I chose to leave out for simplicity. Besides, just having the `nav` element on its own was descriptive enough that there was little need for a dedicated section to accompany it.

Article

The `article` element is a block level element that represents a standalone section of content. One key distinction between an `article` and a `section` element is that the article should be able to make sense all by itself, such as through syndication or some other means of publishing just that node and its children.

An `article` element can be the only one of its kind on a page, such as when a page only has one central content area (think of a generic about us page, for example, where the one block of content text is the only block of text on the page), or it can be repeated, such as when you have a blog roll, an RSS feed, or hey, maybe even a collection of game cards.

The only restriction that applies to the article tag is that it is not inside an address tag.

The game cards used in our web portal are somewhat complex, in that it encapsulates several other elements, including a second level of nested article tags, as shown in the following code snippet:

```
<article>
  <figure>
    <img src="img/typography-poster.png" alt="Typography Game Poster"
/>
      <figcaption>
        <p>
          <strong>Typography Game: </strong> Can you spell out the
right words, fast enough to win over your dream-boat before Snooty-
McSnootington?
        </p>
      </figcaption>
  </figure>
    <article>
      <header>
        <h1>Typography Game</h1>
      </header>
      <p>
        <strong>Description: </strong> Can you spell out the right
words, fast enough to win over your dream-boat before Snooty-
McSnootington?
      </p>
      <address><strong>Created by: </strong>Rodrigo Silveira</address>
      <time datetime="2013-02-07"><strong>Last Updated: </
strong>February 7, 2013</time>
      <details>
        <summary>Tools Used</summary>
```

```
            <ul>
              <li>Web Forms</li>
              <li>Web Fonts</li>
              <li>Metadata</li>
              <li>Transitions</li>
              <li>Animation</li>
              <li>Text shadow</li>
              <li>Box shadow</li>
              <li>window.JSON</li>
              <li>querySelector</li>
            </ul>
        </details>
        <p>
          <a href="/packt/#">Play Game</a>
        </p>
      </article>
  </article>
```

The corresponding CSS for this game card is also very basic and straight forward as shown in the following code snippet:

```
section > article {
  background: #fff;
  border-bottom: 2px solid #aaa;
  margin: 0 0 20px;
  padding: 0;
  width: 500px;
}
```

This may seem like a lot of code just to illustrate an `article` tag, but as you have noticed, there is a bit more than just an `article` tag there. The reason I'm showing that entire code snippet is because that outer article tag represents a game card widget, where everything inside that article represents one unit of information. That game card is repeated for each game described in the page, so it makes sense to keep everything together. Each individual element inside this `article` tag is styled separately, and explained in isolation in the following text.

Note, however, that while I keep referring to this widget as a game card, it is nothing more than an article element with a few sub nodes within it. That's the beauty of HTML5. We can use a mixture of elements and create new elements that make sense within the context of our application, but by itself, all the elements used still make sense to an outsider looking at our code without any knowledge about our objectives or specific meaning for the custom widget. We could have taken things a step further, and added a CSS class to our game cards so that its meaning would be further clarified, but it still worked out fine without it.

Optimizing CSS selectors

While we didn't use a single CSS class in the entire web portal design, in the real world we would probably have modified the HTML structure slightly in order to avoid more complex CSS selectors. This modification could be as simple as adding ID or CLASS attributes to certain DOM elements, so that we can target them through CSS with a single rule, rather than having to traverse through complex selectors.

The more complex a CSS selector, the more work the browser will have to do in order to access all of the nodes targeted by the selector. Thus, you must always watch the complexity of your CSS selectors, and look for ways to simplify them for better performance rendering. A simple way to optimize those selectors is to group similar elements with the same CSS class, or isolating single elements with a unique CSS ID.

Of course, the trick is to find that happy medium where you're not writing terribly complex CSS selectors, but at the same time you're not giving a unique ID to every element in your document. Ideally, the HTML structure has no idea about how it will be rendered, so the fewer classes and IDs you can possibly add to a document, the better. The key to always remember is balance.

Figure

The `figure` element is a block level element that, unlike some may say, does not replace the `image` element. Rather, a figure complements images by adding extra, optional benefits. First and foremost, an `img` tag includes the optional attributes `alt` and `title`, both of which are used to describe the image. However, these attributes are aimed more at machines (software) than humans, since the text in the `alt` attribute is not visible to the end user, and the text in the `title` attribute is not intuitive nor readily available to the user.

In contrast, a `figure` element allows you to provide an optional, yet powerful attribute that displays a caption for the entire figure, which belongs to the figure and no other element. This attribute is the element `figcaption`, which is nested inside the `figure` element that it belongs to.

The key takeaway here is that a `figure` element represents one standalone unit, although it may have a few other components inside of it.

The `figure` tag has no restrictions about where it can and can't appear.

The main usage of figures in our web portal is showing a poster representing a game, placed inside each game card, as shown in the following code snippet:

```
<figure>
  <img src="img/typography-poster.png" alt="Typography Game
    Poster" />
    <figcaption>
      <p>
        <strong>Typography Game: </strong> Can you spell out the
          right words, fast enough to win over your dream-boat
          before Snooty-McSnootington?
      </p>
    </figcaption>
</figure>
```

The corresponding CSS for this figure is a bit tricky, as it hides the figure caption until the image element is hovered over by the user. When the user hovers off the figure image, the figure caption belonging to that figure is hidden, clearing the user interface.

```
section article figure {
  margin: 0;
  position: relative;
}

section article figure:hover figcaption {
  display: block;
}

section article figure img {
  width: 500px;
  height: 200px;
}

section article figcaption {
  display: none;
  background: rgba(25, 25, 25, 0.75);
  position: absolute;
  bottom: 5px;
  width: 100%;
}

section article figcaption p {
  margin: 0;
  padding: 10px;
  color: #fff;
}
```

The way we're using the `figure` tag here tries to maximize the way CSS allows us to add some nice responsive effects, as well as make the most out of the figure caption element as well. What we do is very simple; we display the figure image with the caption position on top of it, with a semi transparent background to add that extra visual touch. By default the caption is hidden, but when the mouse hovers over the image, that caption block appears.

For an extra detail, we could have set one of the experimental CSS properties that would control the transition of that caption element, so that as it went from hidden (invisible) to visible, it would do so with a smooth fade in and fade out effect. However, for simplicity's sake, we left that out, and will touch on that later in the book. You will see, though, how easy it is to set that up, like most CSS properties are.

Figcaption

A `figcaption` (figure caption) element is a block level element that represents just a caption for a particular figure. This is different from the `title` element of an `image` tag in that the caption is visible by default, and it may be formatted with HTML and CSS. On the other hand, a `title` attribute can only be a string of text, which is decorated by the browser, and only shown to the user when the mouse cursor is placed over the image element for a brief moment. Thus, a `figcaption` element is especially useful for mobile devices, where the user doesn't have a cursor to place over the image, thus making it impossible for the title text to be displayed.

The only restriction that applies to the `figcaption` tag is that a `figcaption` tag may only be nested inside a `figure` tag.

Each figure is accompanied by a figure caption element, which simply contains a formatted string describing the figure that holds it, as shown in the following code snippet:

```
<figcaption>
  <p>
    <strong>Typography Game: </strong> Can you spell out the right
      words, fast enough to win over your dream-boat before
      Snooty McSnootington?
  </p>
</figcaption>
```

The corresponding CSS for this figure caption is pretty basic, with the exception of the `hover` attribute, which does two things; it tells which element's hover event triggers the rule, and it specifies which element needs the proceeding style applied to once the hover event is captured:

```
section article figure:hover figcaption {
  display: block;
}

section article figcaption {
  display: none;
  background: rgba(25, 25, 25, 0.75);
  position: absolute;
  bottom: 5px;
  width: 100%;
}
```

The only purpose for this address tag was to represent the author of the content shown on the page. The styling was the same as for other similar elements inside the game card widget, as to provide consistency to the user interface and user experience.

Address

The address element is a block level element that, unlike one may think, does not represent a physical mailing address. What an address element represents is simply any contact information, which can be anything from an author name, to an email address, to whatever piece of information that performs the task of identifying contact information.

You may have noticed that most structural elements are not allowed to be descendants (children nodes) of the address tag. However, any content elements (paragraphs, figures, hyperlinks, and so on) may be nested inside address tags.

The only restriction that applies to the `address` tag is that, we do not nest an `address` tag inside an `address` tag.

Each game card contains an address element that does nothing more than state the name of the creator of the game, as shown in the following code snippet:

```
<address>
  <strong>Created by: </strong>Rodrigo Silveira
</address>
```

The corresponding CSS for this address element is also very basic and straight forward as shown in the following code snippet:

```
section article article p,
section article article address,
section article article time {
  color: #555;
  line-height: 1.5em;
  margin: 1em 0;
  display: block;
}
```

The only purpose of this `address` tag was to represent the author of the content shown on the page. The styling was the same as for other similar elements inside the game card widget, as to provide consistency to the user interface and user experience.

Time

The `time` element represents a date and/or time, the context of which is independent of its data. For example, you can have a `time` element on a document representing the date and time it was first created, or when it was last updated. You can use a time element to represent your birthday, or the total amount of time it took you to complete a task, such as develop a new game in HTML5.

There are several attributes that can be applied to a time element, which specify the type of data the element represent. To be more precise, there is only one attribute; `datetime`, which can have any of the several possible values describing the attribute. Note that these attributes are intended for use primarily by the browser and not the user. The data to be consumed by the user is placed inside the opening and closing time tags.

Possible values you may use for the `datetime` attribute are as follows:

- Local date and time is formatted in yyyy-mm-ddThh:mm:ss. For example, 2012-03-21T15:33:00.
- Date is formatted in yyyy-mm-dd. For example, `2012-03-21`.
- Month is formatted in yyyy-mm. For example, `2012-03`.
- Week is formatted as yyyy-Wn (where n is the nth week of that year, which must be a two digit number between 1 and 53 inclusive, unless there are only 52 weeks on that particular year). For example, `2012-W07`.
- Time is formatted in hh:mm:ss.ms (where ms is an optional millisecond mark). For example, `15:16:24.42`.

The only restriction that applies to the time tag is that it cannot be nested inside another time tag.

Each game card contains a time element that represents the date when the game was last updated, as shown in the following code snippet:

```
<time datetime="2013-02-07">
  <strong>Last Updated: </strong>February 7, 2013
</time>
```

The corresponding CSS for this time element is the same as the one used in the address element.

The funny thing about the `time` element is that, although it allows us to enter a date or time in a very specific format that the browser can understand, the context that give those dates and times all the meaning in the world, are completely arbitrary. In our case, we use the `datetime` attribute to mean the date when the game was last updated, but the browser doesn't know any better.

If the browser doesn't know what the `datetime` attribute represents, what good is it, you may ask? The main reason is such that the values of the attribute are standardized, so any application wishing to parse the elements and use the `datetime` attribute (provided that the meaning of the attribute in its specific context is known beforehand) can easily parse the data into a useful format.

Details

The `details` element is a block level element that represents a block of additional information that the user can control. By default, the contents of this tag are hidden by the browser, except for the first nested pair of summary tags, which is how the user can naturally interact with the tag. Any additional sibling summary tags are simply treated as block level paragraph tags, though the semantic meaning breaks down at that point.

You may nest details elements inside other details, using subsequent summary elements for additional control over the inner details, but keep in mind that each additional level does not get its margin or padding automatically adjusted like list elements do. Every nested level of the `details` elements is left aligned (by default).

The position of the `summary` tag is relative to the rest of the contents inside the `details` tag does not control how the browser renders the `details` tag. Whenever you insert the `summary` tag within all other content tags inside the `details` element, the browser will only render the `summary` tag when the details are collapsed, and it will render the summary at the bottom of all the contents when the details are opened.

Finally, we can programmatically control the collapsing behavior of the `details` element through the special `open` attribute. Unlike some elements where the value of an attribute determine the state of that attribute, simply having the `open` attribute present will cause the attribute to be active. Setting the attribute to `open="false"`, for example, will not have the effect one might imagine. The element will become open (inspite of its value attempting to signify otherwise) because the attribute is there. The only possible value for the `open` attribute, if you decide to assign it one, as per the HTML5 spec, is the empty string, such as `open=""`.

One way to check for the attribute and control it, is to use the `setAttribute` and the `getAttribute` methods available in the DOM elements, as in the following snippet:

```
var details = document.getElementsByTagName("details")[0];
// If the open property is not present at all, this will return null.
// If it is present, but with no value set to it, this will return
"undefined"
// as a string, meaning that the value of the attribute is undefined,
but
// the attribute itself is defined. Any other value assigned to the
attribute will
// be return by this call.
var isOpen = details.getAttribute("open");
if (isOpen == null)
  details.setAttribute ("open");
else
  details.removeAttribute ("open");
```

Restrictions that apply to the `details` tag that you need to keep in mind include the following two rules:

- Do not nest a `details` tag inside an `a` tag.
- Do not nest a `details` tag inside a `button` tag.

Each game card contains a details element that summarizes each tool used on the game, as shown in the following code snippet:

```
<details>
  <summary>Tools Used</summary>
    <ul>
      <li>Web Forms</li>
      <li>Web Fonts</li>
      (...)
    </ul>
</details>
```

The corresponding CSS for this element is also very basic and straight forward as seen in the following code snippet:

```
section article article details {
  color: #555;
  border-bottom: 1px dashed #ddd;
  padding-bottom: 20px;
}
```

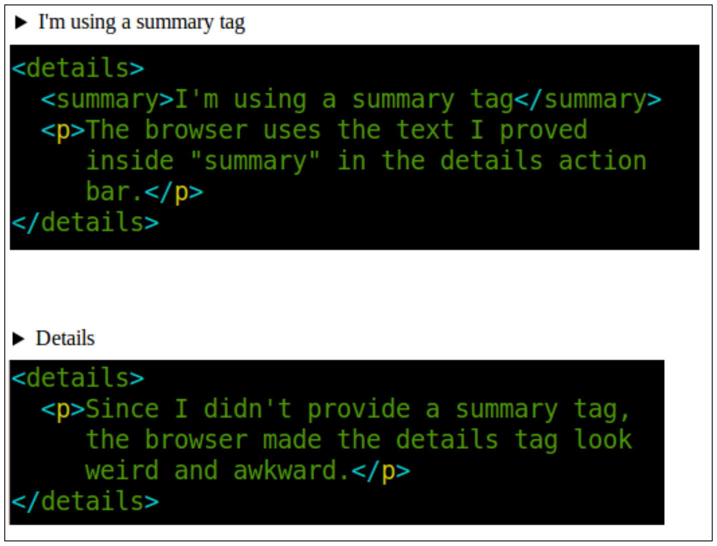
In the past, this behavior has been implemented by developers using a combination of JavaScript and CSS. You can simulate the toggle behavior by capturing the click event on some element, then checking if the contents you want to toggle have the CSS property `display` set to block or none. If it is currently set to `display="none"`, for example, then you'd simply set it to `display="block"`, and vice versa.

It is important to note here that this is not the way that browsers implement the behavior natively. Even if we assign CSS properties to force visibility of the contents of a `details` tag, the browser will still not render any of it, if the `details` element does not have the open attribute. The only way to force the contents to be shown programmatically is by opening the `details` tag though the `open` attribute. Of course, if you hide the contents of the tag through CSS, then the browser will not display those even when you open the `details` tag.

All we're doing with the `details` tag in the case of our web portal, is show a list of tools used in the creation of the particular game illustrated by the game card where the details are shown. In a real world example, we can use this tag to hide away less important information (secondary information) from the user, but still provide a way for the curious to read more (by using the `summary` tag, or through your own mechanism, using the `open` attribute, as discussed previously).

Summary

The `summary` element is a block level element that represents a summary or caption for a `details` element. Besides its semantic meaning, the summary element is used to customize the appearance of its parent details element. In other words, the text inside the opening and closing `summary` tag is set to the details action bar, through which the user may interact with the element. If you leave out a `summary` tag in the `details` elements, the browser will still display an action bar for the `details` tag, but the text shown inside this bar will be some default value determined by the browser.



```
<details>
  <summary>I'm using a summary tag</summary>
  <p>The browser uses the text I proved
     inside "summary" in the details action
     bar.</p>
</details>
```

```
<details>
  <p>Since I didn't provide a summary tag,
     the browser made the details tag look
     weird and awkward.</p>
</details>
```

Google Chrome defaults to the text Details if you leave out a `summary` tag. Remember, if you have multiple `summary` tags inside a `details` element, only the text of the first one is used in the details action bar.

The only restriction that applies to the `summary` tag is that a `summary` tag may only be nested inside a `details` tag.

The `summary` tag is used to tell the browser how to display the collapsible control, as shown in the following code snippet:

```
<details>
  <summary>Tools Used</summary>
    <ul>
      <li>Web Forms</li>
      <li>Web Fonts</li>
      (...)
    </ul>
</details>
```

The corresponding CSS for this summary is also very basic and straight forward as seen in the following snippet:

```
section article article summary {
  outline: 0;
  cursor: pointer;
}
```

The only thing we had to do to this element was to touch it up so that it didn't look like default because different browsers render the summary differently. All we did was remove the yellow outline that appears when the element is focused, and also added a pointer cursor to hint to the user that the element can be clicked and interacted with.

Like some other elements, such as file inputs, range inputs, check boxes, and radio boxes, the ability to style the summary element is fairly limited. For example, there's little you can do to customize the different states of the element, as well as the arrows that show to the left of the element. While there are some experimental, browser-specific CSS attributes that you can use, but as of this writing, there's really nothing we can do to universally style this element in its natural state. Fortunately, the element's default style doesn't look bad.

Footer

The `footer` element is a block level element that represents the footer section of a document, a section of the document, or an article. A footer is really just meant to be a place where you can display that final piece of information about whatever the footer belongs to, such as copyright information (when using a footer for the whole document), or author information (preferably nested within the `address` tags), related links, and so on, when applying a footer to an article.

An easy way of thinking about the purpose of the footer, is to think of a compliment/opposite tag to a `header` tag. While one introduces a document, section, or article, the other concludes it, adding extra details to further inform the user about that document, section, or article.

Restrictions that apply to the `footer` tag that you need to keep in mind include the following three rules:

- Do not nest a `footer` tag inside a `header` tag.
- Do not nest a `footer` tag inside an `address` tag.
- Do not nest a `footer` tag inside another `footer` tag.

The only footer element in our web portal describes the document, and is very basic and to the point, as shown in the following code snippet:

```
<footer>
  <p>Copyright &copy; 2013 My Fun HTML5 Games!</p>
</footer>
```

The corresponding CSS for this footer element is also very basic and straight forward as in the following code snippet:

```
footer {
  clear: both;
  background: #333;
}

footer p {
  margin: 0;
  padding: 10px 20px;
  color: #aaa;
}
```

The most significant CSS feature in this element is the clearing, which we use to make sure that any left or right floating of previous elements don't affect the rendering of the footer element. Beyond that, the only use for this footer is to display some standard copyrighting information, which in this particular case is more symbolic than anything.

To be more theoretically correct, we should probably have used other elements inside the footer to further describe the information there, such as an `address` tag. If we were to include some top level links, or some sort of navigation in the footer, as is common in most websites, a `nav` element would be very appropriate.

Depending on how granular you design your documents, you can also use a footer to describe smaller parts of your document. In our case, we could have used a footer inside each game card to show the end of each game card. The only advice here is that you consider the benefits of going with whatever level of abstraction and granularity you eventually decide to go with. What are you gaining by not using a footer at the end of every game card? Is the code easier to read without the extra footer element? A very wise quote has guided my personal HTML5 design when it comes to choosing whether or not to use elements, just because they could be used in a given situation: just because you can't, doesn't always mean you should.

Bringing it all together

Now that we've seen all the individual pieces and how they contribute to the complete whole, it is time to put all of the pieces together. Since the complete source code for the web portal is fairly lengthy for a book of this nature, we will not include the complete source code in the book, but you can download it all directly from the book's website.

The way the web portal was designed is pretty simple, and follows the file structure described in the following list. For the general environment setup, refer to the *Setting up* section earlier in this chapter. For the specific directory structure for this web portal project, we will need the following directories, remembering that the root directory of our project is named `packt`:

- `/packt/`
- `/packt/index.html`
- `/packt/css/`
- `/packt/css/style.css`

Index.html

Since this is the only HTML file in the web portal, we can just name it `index.html`, and have the server set it as the default welcome file that is returned when no file is specified for requests to the directory containing our project. As you already know, the name of the file is not as important as the contents thereof.

Style.css

Inside the `/css` directory of our project, create a file named `style.css`. This file will hold all of the styling for the portal. In subsequent projects covered throughout the rest of the book, we'll follow this same structure for stylesheets. All of the styles will be placed in one single file for ease of reading, understanding, and managing the file. Also, remember that browser-specific code may change over time. In order to avoid running into issues with browser compatibility, we'll be providing the most verbose possible version of each stylesheet file. Feel free to refactor any code provided in order to remove any superfluous CSS rules or other statements.

Summary

In this chapter we talked about the Same-Origin Policy, and why it's such an important aspect of the security model of HTML5. In order to comply with browser security and develop our games, we walked through the set up of our development environment, going through the steps required to set up an Apache server in our local system. We chose the *AMP stack because of its simplicity, and widespread use. We saw how to install the stack in the three most popular operating systems today, namely Windows, Mac, and Linux. If you are familiar with other servers, such as Microsoft's IIS, they are equally useful for our purposes in HTML5 development.

We also discussed some of the most commonly used HTML5 semantic tags, and worked through a hands-on example of how to use them in a real world situation. We then build an HTML5-based portal from which our games can be accessed and documented. If you would like to publish our HTML5 games portal (and the portal itself), the only step required will be to copy those files into a publicly accessible web server, or to configure our Apache installation to accept requests from any computer connected to the world wide web.

Next chapter will finally get us going with actual programming. We will build a typography game using web forms, web fonts, and CSS transitions and animations.