JUMP START SERIES

sitepoint

JUMP START Html5

By Tiffany Brown

GET UP TO SPEED WITH HTML5 IN A WEEKEND

Summary of Contents

sitepoint®

JUMP START HTML5 BASICS

BY **TIFFANY B. BROWN**

Jump Start HTML5 Basics

by Tiffany B. Brown

Product Manager: Simon Mackie
English Editor: Paul Fitzpatrick
Technical Editor: Craig Buckler
Cover Designer: Alex Walker

Notice of Rights

Notice of Liability

Trademark Notice

Published by SitePoint Pty. Ltd.

48 Cambridge Street Collingwood
VIC Australia 3066
Web: www.sitepoint.com
Email: business@sitepoint.com

ISBN 978-0-9922794-9-3 (print)

ISBN 978-0-9922794-8-6 (ebook)
Printed and bound in the United States of America

About Tiffany B. Brown

Tiffany B. Brown is a freelance web developer and technical writer based in Los Angeles. She has worked on the web for more than a decade at a mix of media companies and agencies. Before founding her consultancy, Webinista, Inc., she was part of the Opera Software Developer Relations & Tools team. Now she offers web development and consulting services to agencies and small design teams.

About SitePoint

SitePoint specializes in publishing fun, practical, and easy-to-understand content for web professionals. Visit http://www.sitepoint.com/ to access our blogs, books, newsletters, articles, and community forums. You'll find a stack of information on JavaScript, PHP, Ruby, mobile development, design, and more.

About Jump Start

Jump Start books provide you with a rapid and practical introduction to web development languages and technologies. Typically around 150 pages in length, they can be read in a weekend, giving you a solid grounding in the topic and the confidence to experiment on your own.

To my husband, Jason Toney, who made it possible for me to write this book by playing the role of a supportive wife. To my sisters by choice: Shoshana and Cheryl. To my unexpected cheerleaders: Divya Manian, Chris Mills, David Storey, and Jen Hanen. To my mom for being mom. And to my late father, who was more of an inspiration than I ever realized.

Table of Contents

Preface

HTML (HyperText Markup Language) is the predominant language of web pages. Whenever you read or interact with a page in your browser, chances are it's an HTML document. Originally developed as a way to describe and share scientific papers, HTML is now used to mark up all sorts of documents and create visual interfaces for browser-based software.

With HTML5, however, HTML has become as much an of **API** (Application Processing Interface) for developing browser-based software as it is a markup language. In this book, we'll talk about the history of HTML and HTML5 and explore some of its new features.

Who Should Read This Book

Although this book is meant for HTML5 beginners, its brevity means that it can't be comprehensive. As a result, we do assume some prior knowledge of HTML. If you are completely new to web development, SitePoint's *Build Your Own Website Using HTML and CSS* may be a better book for you.

Conventions Used

You'll notice that we've used certain typographic and layout styles throughout this book to signify different types of information. Look out for the following items.

Code Samples

Code in this book will be displayed using a fixed-width font, like so:

```
<h1>A Perfect Summer's Day</h1>
<p>It was a lovely day for a walk in the park. The birds
were singing and the kids were all back at school.</p>
```

If the code is to be found in the book's code archive, the name of the file will appear at the top of the program listing, like this:

example.css

```
.footer {
  background-color: #CCC;
  border-top: 1px solid #333;
}
```

If only part of the file is displayed, this is indicated by the word *excerpt*:

example.css ***(excerpt)***

```
  border-top: 1px solid #333;
```

If additional code is to be inserted into an existing example, the new code will be displayed in bold:

```
function animate() {
  new_variable = "Hello";
}
```

Also, where existing code is required for context, rather than repeat all the code, a ⋮ will be displayed:

```
function animate() {
  ⋮
  return new_variable;
}
```

Some lines of code are intended to be entered on one line, but we've had to wrap them because of page constraints. A ➥ indicates a line break that exists for formatting purposes only, and should be ignored.

```
URL.open("http://www.sitepoint.com/responsive-web-design-real-user-
➥testing/?responsive1");
```

Tips, Notes, and Warnings

Hey, You!

Tips will give you helpful little pointers.

Ahem, Excuse Me …

Notes are useful asides that are related—but not critical—to the topic at hand. Think of them as extra tidbits of information.

Make Sure You Always …

… pay attention to these important points.

Watch Out!

Warnings will highlight any gotchas that are likely to trip you up along the way.

Supplementary Materials

http://www.sitepoint.com/store/jump-start-html5-basics/
The book's website, containing links, updates, resources, and more.

https://github.com/spbooks/jshtml-basics1
The downloadable code archive for this book.

http://www.sitepoint.com/forums/
SitePoint's forums, for help on any tricky web problems.

`books@sitepoint.com`
Our email address, should you need to contact us for support, to report a problem, or for any other reason.

Tools You'll Need

All you'll need to develop HTML5 documents is a text editor for writing, and a browser for viewing your work. Don't use word processing software. Those programs are made for writing documents, not for programming. Instead, you'll need software that can read and write plain text.

If you’re a Windows user, try Notepad++[1], a free and open-source text editor. Mac OS X users may want to try TextWrangler[2] by Bare Bones software. It's free, but not open source. Brackets[3] is another option for Windows and Mac users. Linux users can use gEdit, which is bundled with Ubuntu Linux, or try the free and open source Bluefish[4]. Paid software options are also available, and are sometimes more refined than free and open-source options.

You'll also need at least one browser that supports HTML5 in order to make use of the examples in this book. Make sure you’re using the latest version of Google Chrome, Microsoft Internet Explorer, Apple Safari, Opera, or Mozilla Firefox available for your operating system. Internet Explorer and Safari are bundled with Microsoft Windows and Mac OS X, respectively. Other browsers may be downloaded from their company web sites.

Do You Want to Keep Learning?

[1] http://notepad-plus-plus.org/
[2] http://www.barebones.com/products/textwrangler/
[3] http://brackets.io/
[4] http://bluefish.openoffice.nl/
[5] https://learnable.com/

Chapter 1

What is HTML5?

The easy answer is that it's the latest version of HTML. But that doesn't tell us much. Specifically, HTML5:

- defines a parsing algorithm for generating a consistent **DOM** (Document Object Model) tree, even from ambiguous or poor-quality markup
- adds new elements to support multimedia and web applications
- redefines the rules and semantics of existing HTML elements

With HTML5, we can now embed audio and video natively within HTML documents. We can use inline **SVG** (Scalable Vector Graphics) markup. We can build more robust form experiences, complete with native error checking. We can create games, charts, and animations using the `canvas` element. Documents can communicate with each other using cross-document messaging. In other words, HTML5 is much more of *an application platform*, not just a markup language.

A Brief History of HTML5

The story of how and why HTML5 came to be is too long to adequately cover in this book. That said, a little historical context may help you understand some of how HTML5 came to be.

HTML has its roots in ***Standard General Markup Language***, or SGML. Think of SGML as a set of rules for defining and creating markup languages.

HTML, or HyperText Markup Language, began as an application of SGML. Created in the early 1990s, HTML was a standardized way to describe the structure of hypertext documents. "Hypertext" simply means that the text "contains links to other texts" and is not constrained by linearity[1].

By describing the structure of a document, we decouple it from how it looks, or how it's presented to the end user. This made it easier to share and redistribute. The associated Hypertext Transfer Protocol (HTTP) made sharing documents over the internet easy.

HTML: The Early Years

"HTML 1" defined a simple, tag-based syntax for explaining document structure—a very basic document structure. Paragraph (`p`) and list item (`li`) elements didn't require an end tag. The earliest version[2] didn't even include the `img` or `table` elements. Image support was added in version 1.2[3] of the specification.

HTML grammar changed only slightly with version 2.0[4]. Now we could use end tags for elements such as `p` and `li`, but these end tags were optional. The transition from HTML 2.0 to HTML 3.2, however, marked a huge leap.

With HTML 3.2, we could change type rendering with the `font` element. We could add robust interactivity with Java applets and the `applet` element. We could add tabular data with the `table`, `tr` and `td` elements. But perhaps the most significant feature introduced in HTML 3.2 was style sheets.

[1] http://www.w3.org/WhatIs.html
[2] http://info.cern.ch/hypertext/WWW/MarkUp/MarkUp.html
[3] http://www.w3.org/MarkUp/draft-ietf-iiir-html-01.txt
[4] http://www.w3.org/MarkUp/html-spec/

Most of the web, however, settled on HTML 4. With the advent of HTML 4, we could tell the browser how to parse our document by choosing a document type. HTML 4 offered three options:

- Transitional, which allowed for a mix of deprecated HTML 3.2 elements and HTML 4
- Strict, which only allowed HTML 4 elements
- Frameset, which allowed multiple documents to be embedded in one using the `frame` element

What HTML versions 1 through 4 didn't provide, however, were clear rules about how to parse HTML.

The W3C stopped working on HTML 4 in 1998, instead choosing to focus its efforts on a replacement: XHTML.

A Detour Through XHTML Land

XHTML 1.0[5] was created as "a reformulation of HTML 4 as an XML 1.0 application." **XML**, eXtensible Markup Language, was a web-friendly revision of SGML, offering stricter rules for writing and parsing markup.

XHTML, for example, required lower case tags while HTML allowed upper case tags, lower case tags, or a mix of the two. XHTML required end tags for all non-empty elements such as `p` and `li`. Empty elements such as `br` and `img` had to be closed with a `/>`. You had to quote all of your attributes, and escape your ampersands. All pages had to be served as `application/xml+xhtml` MIME type.

XHTML taught us how to write better-quality markup. But ultimately suffered from a lack of proper browser support.

XForms[6], on the other hand, was supposed to replace HTML forms. XForms introduced `upload` and `range` elements to provide richer ways to interact with web sites.

XForms didn't gain much traction, however. After all, why introduce a specific markup language for forms to the web? Why not enhance HTML instead?

[5] http://www.w3.org/TR/xhtml1/

[6] http://www.w3.org/TR/2007/REC-xforms-20071029/

The Battle for World DOM-ination

In 1996, Netscape released Netscape Navigator 2.0 with support for two separate, but related technologies: JavaScript and the Document Object Model. We usually talk about them as though they're one and the same thing, but DOM is an API for interacting with HTML documents. JavaScript is the primary language for interacting with that API.

Netscape Navigator's DOM interface turned each element of an HTML page into an object that could be created, moved, modified, or deleted using a scripting language. Now we could add animation or interactivity to our web pages, even if we had to wait ages for them to download over our super-slow, 14.4Kbps modems.

The DOM was such a brilliant addition to the web that other browsers quickly followed suit. But not every browser implemented the DOM in quite the same way. Netscape Navigator, for example, used `document.layers` objects to reference the entire collection of HTML nodes. Microsoft Internet Explorer went with `document.all`. And web developers everywhere spent years struggling to reconcile the two. Opera and WebKit, for what it's worth, followed Internet Explorer's lead. Both browsers adopted `document.all`.

Eventually "DOM0" went from being a standard-through-implementation to a standard-through-specification with the Document Object Model (DOM) Level 1 Specification[7]. Rather than `document.layers` and `document.all`, we could use `document.getElementById` and `document.getElementsByTagName`. Today, all browsers support the DOM.

Applets and Plugins

In the midst of all of this—the growth of HTML, the rise of the DOM, and the shift to XHTML—applets and browser plugins joined the party. To their credit, applets and plugins added functionality missing from HTML. For example, RealPlayer and Apple's QuickTime brought audio and video to the web. With Java applets, you could run a spreadsheet program in your browser. Macromedia (now Adobe) Flash and Shockwave let us add all of the above, plus animations.

Applets and plugins, however, suffered from three major problems:

[7] http://www.w3.org/TR/REC-DOM-Level-1/

1. Users who don’t have the plugin (or the applet environment) can’t see the content.
2. Applets and plugins expanded the surface for internet-based security breaches.
3. They were commercial products, and required developers to pay a license fee.

What’s more, plugins and applets sometimes caused their host environment—the browser—to slow or crash.

So what we had on the web was a scenario in which:

- Browsers didn’t parse HTML according to the same rules.
- New markup languages offered few clear advantages over HTML but added overhead to implement.
- Plugins and applets offered additional functionality, but created security and stability issues for browsers and licensing costs for developers.

These are the problems that HTML5 solves:

- It incorporates features and grammars introduced by XHTML and XForms.
- It almost eliminates the need for plugins and the stability and security issues they may introduce.

What HTML5 Isn’t

I admit that I’m taking a bit of a purist approach in this book. HTML5 has become a buzzword-y shorthand for "everything cool we can do in the browser that we couldn’t do before." In this book, however, we mean HTML elements and their Document Object Model (DOM) APIs.

We won’t talk much about features introduced with **CSS** (Cascading Style Sheets), Level 3 in these pages. We will talk about what’s commonly called "JavaScript", but is more accurately the DOM HTML API. We’ll also talk about Scalable Vector Graphics, or SVG—but only to the extent that we discuss mixing SVG and HTML within the same document.

This book is intended as a short introduction to HTML5. For that reason, we won't cover advanced features in depth. This book will, however, give you an introduction to what's new and different about HTML5 versus previous versions.

A Note on the HTML5 Specification

Both the Web Hypertext Application Technology Working Group (**WHATWG**) and the World Wide Web Consortium (**W3C**) publish HTML5 specifications. The two groups worked together for years, creating a single specification managed by a single editor. However in 2011, they diverged. There are now two competing, though largely similar, versions of the specification. Each has its own editor.

The WHATWG version[8] of the specification is a "living document." New features are added, tweaked, and occasionally removed after some discussion within the community. This version is far more fluid and subject to change.

The W3C, however, has a different process. Specification editors still consult the community, but each document moves in phases from "Working Draft" to "Candidate Recommendation" to "W3C Recommendation." As a result, W3C specifications are versioned. The 2011 joint version of HTML5 specification became the W3C's HTML5 Specification[9]. Subsequent revisions are part of the HTML 5.1 specification[10].

There are differences between the two specifications, some subtle, some significant. These differences are not well documented, however. Since this book doesn't delve in to the minutiae of HTML5, these differences won't mean much for us.

[8] http://www.whatwg.org/specs/web-apps/current-work/multipage/
[9] http://www.w3.org/TR/html5/
[10] http://www.w3.org/html/wg/drafts/html/master/Overview.html

Chapter 2

The Anatomy of HTML5

Every HTML document is made from elements, and elements are represented by tags. Tags are a sequence of characters that mark where different parts of an element start and/or stops.

All tags begin with a left-facing angle bracket (<) and end with a right-facing angle bracket (>). Every element has a **start tag** or **opening tag**, which starts with <, and is followed by the element name (or an abbreviation of it). The element name may be followed by an **attribute** (or series of attributes) that describes how that instance of an element is supposed to behave. You can set an explicit value for an attribute with an = sign. Some attributes, however, are empty. If an empty attribute is present, the value is true. Let's look at an example using the `input` element.

```
<input type="text" name="first_name" disabled>
```

Here, `type`, `name` and `disabled` are all attributes. The first two have explicit values, but `disabled` is empty. Some elements allow empty attributes, and these are usually those that might otherwise accept true/false values. Here's the tricky part: The value of an empty attribute is either true or false based on the presence or absence of the attribute, regardless of its set value. In other words, both `disabled="true"` and `disabled="false"` would also disable input control.

Most elements also have a closing tag. Closing tags also start with `<`, but rather than being immediately followed by the element name, they are followed by a forward slash (`/`). Then comes the element name, and right-angle bracket or `>`. However, some elements are known as **void elements**. These elements cannot contain content, and so do not have a closing tag. The `input` element shown above is an example of a void element.

Now that we've covered the basics of tags, let's take a closer look at an HTML5 document.

Your First HTML5 Document

Open up your favorite text editor and type the following. Save it as **hi.html**.

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

Congratulations—you've written your first HTML5 document! It's not fancy, perhaps, but it does illustrate the basics of HTML5.

Our first line, `<!DOCTYPE html>` is required. This is how the browser knows that we're sending HTML5. Without it, there's a risk of browsers parsing our document incorrectly. Why? Because of **DOCTYPE switching.**

DOCTYPE switching means that browsers parse and render a document differently based on the value of the `<!DOCTYPE` declaration, if it's served with a `Content-type:text/html` response header. Most browsers implemented some version of DOCTYPE switching in order to correctly render documents that relied on non-standard browser behavior, or outdated specifications.

HTML 4.01 and XHTML 1.0, for example, had multiple modes—strict, transitional, and frameset—that could be triggered with a DOCTYPE declaration, whereas HTML 4.01 used the following DOCTYPE for its **strict mode**.

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

Transitional, or loose DOCTYPE declarations trigger **quirks mode.** In quirks mode, each browser parses the document a little bit differently based on its own bugs and deviations from web standards.

Strict DOCTYPE declarations trigger **standards mode** or **almost standards mode**. Each browser will parse the document according to rules agreed upon in the HTML and CSS specifications.

A missing DOCTYPE, however, also triggers quirks mode. So HTML5 defined the shortest DOCTYPE possible. The HTML5 specification explains:

> "DOCTYPEs are required for legacy reasons. When omitted, browsers tend to use a different rendering mode that is incompatible with some specifications. Including the DOCTYPE in a document ensures that the browser makes a best-effort attempt at following the relevant specifications."

And so, using the HTML5 DOCTYPE (`<!DOCTYPE html>`) triggers standards mode, even for older browsers that lack HTML5 parsers.

The Two Modes of HTML5 Syntax

HTML5 has two **parsing modes** or **syntaxes**: HTML and XML. The difference depends on whether the document is served with a `Content-type: text/html` header or a `Content-type: application/xml+xhtml` header.

If it's served as `text/html`, the following rules apply:

- Start tags are not required for every element.
- End tags are not required for every element.
- Only ***void elements*** such as `br`, `img`, and `link` may be "self-closed" with `/>`.

- Tags and attributes are case-insensitive.
- Attributes do not need to be quoted.
- Some attributes may be empty (such as `checked` and `disabled`).
- Special characters, or entities, do not have to be escaped.
- The document must include an HTML5 DOCTYPE.

HTML Syntax

Let's look at another HTML5 document.

```
<!DOCTYPE html>
  <html>
    <head>
      <meta charset=utf-8>
      <title>Hi</title>
      <!--
      This is an example of a comment.
      The lines below show how to include CSS
      -->
      <link rel=stylesheet href=style.css type=text/css>
      <style>
        body{
          background: aliceblue;
        }
      <style>
    </head>
    <body>
     <p>
        <img src=flower.jpg alt=Flower>
        Isn't this a lovely flower?

      <p>
        Yes, that is a lovely flower. What kind is it?

      <script src=foo.js></script>
    </body>
</html>
```

Again, our first line is a DOCTYPE declaration. As with all HTML5 tags, it's case-insensitive. If you don't like reaching for **Shift**, you could type `<!doctype html>`

instead. If you really enjoy using **Caps Lock**, you could also type `<!DOCTYPE HTML>` instead.

Next is the `head` element. The `head` element typically contains information about the document, such as its title or character set. In this example, our `head` element contains a `meta` element that defines the character set for this document. Including a character set is optional, but you should always set one and it's recommended that you use UTF-8[1].

Make Sure You're Using UTF-8

Ideally, verify your text editor saves your documents with UTF-8 encoding "without BOM" and uses Unix/Linux line-endings.

Our `head` element also contains our document title (`<title>Hi</title>`). In most browsers, the text between the `title` tags is displayed at the top of the browser window or tab.

Comments in HTML are bits of text that aren't rendered in the browser. They're only viewable in the source code, and are typically used to leave notes to yourself or a coworker about the document. Some software programs that generate HTML code may also include comments. Comments may appear just about anywhere in an HTML document. Each one must start with `<!--` and end with `-->`.

A document head may also contain `link` elements that point to external resources, as shown here. Resources may include style sheets, favicon images, or RSS feeds. We use the `rel` attribute to describe the *relationship* between our document and the one we're linking to. In this case, we're linking to a cascading style sheet, or CSS file. CSS is the stylesheet language that we use to describe the way a document *looks* rather than its structure.

We can also use a `style` element (delineated here by `<style>` and `</style>`) to include CSS in our file. Using a `link` element, however, lets us share the same style sheet file across multiple pages.

[1] http://www.w3.org/International/questions/qa-choosing-encodings

By the way, both `meta` and `link`, are examples of void HTML elements; we could also self-close them using `/>`. For example, `<meta charset=utf-8>` would become `<meta charset=utf-8 />`, but it isn't necessary to do this.

To Quote or Not Quote: Attributes in HTML5

In the previous example, our attribute values are unquoted. In our **hi.html** example, we used quotes. Either is valid in HTML5, and you may use double (") or single (') quotes.

Be careful with unquoted attribute values. It's fine to leave a single-word value unquoted. A space-separated list of values, however, must be enclosed in quotes. If not, the parser will interpret the first value as the value of the attribute, and subsequent values as empty attributes. Consider the following snippet:

```
<code class=php highlightsyntax><?php echo 'Hello!'; ?></code>
```

Because both values for the `class` attribute are not enclosed in quotes, the browser interprets it like so:

```
<code class="php" highlightsyntax><?php echo 'Hello!'; ?></code>
```

Only `php` is recognized as a class name, and we've unintentionally added an empty `highlightsyntax` attribute to our element. Changing `class=php highlightsyntax` to `class="php highlightsyntax"` (or the single-quoted `class='php highlightsyntax'`) ensures that both class attribute values are treated as such.

A Pared-down HTML5 Document

According to the rules of HTML—this is also true of HTML 4—some elements don't require start tags or end tags. Those elements are implicit. Even if you leave them out of your markup, the browser acts as if they've been included. The `body` element is one such element. We could, in theory, re-write our **hi.html** example to look like this.

```
<!DOCTYPE html>
<head>
  <meta charset=utf-8>
  <title>Hi</title>
  <p>Hi
```

When our browser creates the document **node tree**, it will add a `body` element for us.

Just because you *can* skip end tags doesn't mean you *should*. The browser will need to generate a DOM in either case. Closing elements reduces the chance that browsers will parse your intended DOM incorrectly. Balancing start and end tags makes errors easier to spot and fix, particularly if you use a text editor with syntax highlighting. If you're working within a large team or within a **CMS** (Content Management System), using start and end tags also increases the chance that your chunk of HTML will work with those of your colleagues. For the remainder of this book, we'll use start and end tags, even when optional.

Start and End Tags

To discover which elements require start and end tags, consult the World Wide Web Consortium's guide HTML: The Markup Language (an HTML language reference)[2]. The W3C also manages the Web Platform Docs[3] which includes this information.

"XHTML5": HTML5's XML Syntax

HTML5 can also be written using a stricter, XML-like syntax. You may remember from Chapter 1 that XHTML 1.0 was "a reformulation of HTML 4 as an XML 1.0 application." That isn't quite true of what is sometimes called "XHTML5". XHTML5 is best understood as HTML5 that's written and parsed using the syntax rules of XML and served with a `Content-type: application/xml+xhtml` response header.

The following rules apply to "XHTML5":

- All elements must have a start tag.

[2] http://www.w3.org/TR/html-markup/
[3] http://docs.webplatform.org/wiki/Main_Page

- Non-void elements with a start tag must have an end tag (`p` and `li`, for example).
- *Any* element may be "self-closed" using `/>`.
- Tags and attributes are case sensitive, typically lowercase.
- Attribute values *must be* enclosed in quotes.
- Empty attributes are forbidden (`checked` must instead be `checked="checked"` or `checked="true"`).
- Special characters must be escaped using character entities.

Our `html` start tag also needs an `xmlns` (XML name space) attribute. If we rewrite our document from above to use XML syntax, it would look like the example below.

```
<!DOCTYPE html>
  <html xmlns="http://www.w3.org/1999/xhtml">
    <head>
      <meta charset="utf-8" />
      <title>Hi</title>
    </head>
    <body>
      <p>
        <img src="flower.jpg" alt="Flower" />
        Isn't this a lovely flower?
      </p>
      <script src="foo.js" />
    </body>
</html>
```

Here we've added the XML name space with the `xmlns` attribute, to let the browser know that we're using the stricter syntax. We've also self-closed the tags for our **empty** or **void** elements, `meta` and `img`. According to the rules of XML and XHTML, all elements must be closed either with an end tag or by self-closing with a space, slash, and a right-pointing angle bracket (`/>`).

In this example, we have also self-closed our `script` tag. We could also have used a normal `</script>` tag, as we've done with our other elements. The `script` element is a little bit of an oddball. You can embed scripting within your documents by placing it between `script` start and end tags. When you do this, you *must* include an end tag.

However, you can also link to an external script file using a `script` tag and the `src` attribute. If you do so, and serve your pages as `text/html`, you *must* use a closing `</script>` tag. If you serve your pages as `application/xml+xhtml`, you may also use the self-closing syntax.

Don’t forget: in order for the browser to parse this document according to XML/XHTML rules, our document must be sent from the server with a `Content-type: application/xml+xhtml` response header. In fact, including this header will trigger XHTML5 parsing in conforming browsers even if the DOCTYPE is missing.

Configuring Your Server

In order for your web server or application to send the `Content-type: application/xml+xhtml` response header, it must be configured to do so. If you’re using a web host, there’s a good chance your web host has done this already for files with an **.xhtml** extension. Here you would just need to rename **hi.html** to **hi.xhtml**. If that doesn't work, consult your web server documentation.

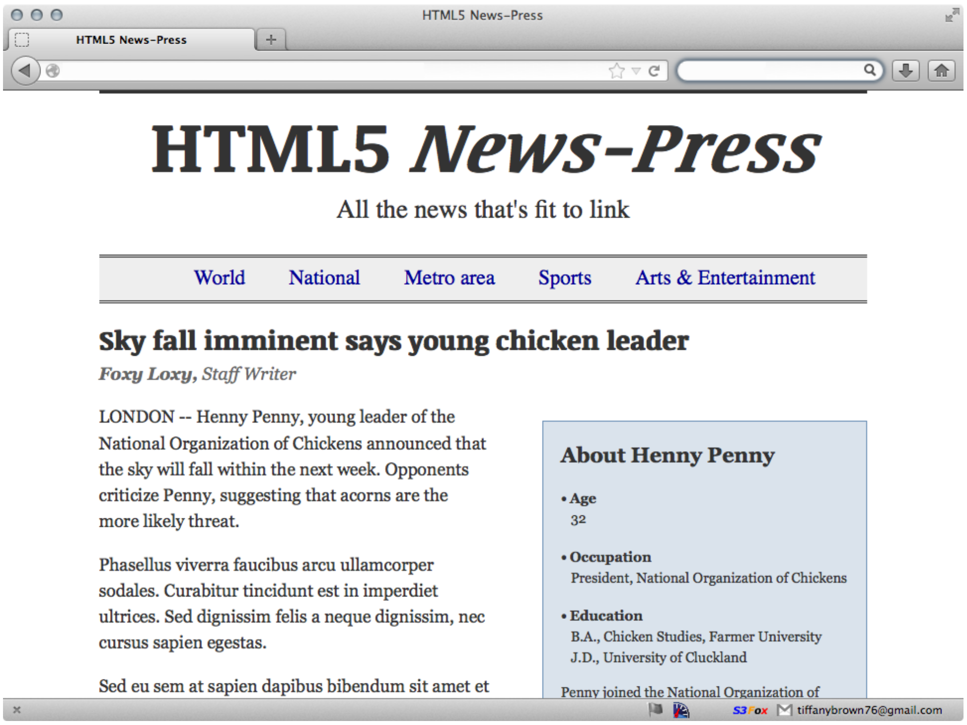
As you may have realized, XML parsing rules are more persnickety. It’s much easier to use the `text/html` MIME type and its looser HTML syntax.

Chapter 3

Structuring Documents

HTML5 adds several elements that provide a way to break a single document into multiple chunks of content—content that may be either related or independent. These elements add semantic richness to our markup, and make it easier to repurpose our documents across media and devices.

We'll take a look at these elements and how they interact using a fictitious top story from a fictitious news web site: The *HTML5 News-Press*, as shown in Figure 3.1.



HTML5 *News-Press*

All the news that's fit to link

World National Metro area Sports Arts & Entertainment

Sky fall imminent says young chicken leader

***Foxy Loxy**, Staff Writer*

LONDON -- Henny Penny, young leader of the National Organization of Chickens announced that the sky will fall within the next week. Opponents criticize Penny, suggesting that acorns are the more likely threat.

Phasellus viverra faucibus arcu ullamcorper sodales. Curabitur tincidunt est in imperdiet ultrices. Sed dignissim felis a neque dignissim, nec cursus sapien egestas.

Sed eu sem at sapien dapibus bibendum sit amet et

About Henny Penny

- **Age**
  32
- **Occupation**
  President, National Organization of Chickens
- **Education**
  B.A., Chicken Studies, Farmer University
  J.D., University of Cluckland

Penny joined the National Organization of

Figure 3.1. The HTML5 News-Press

Our news story page begins with a masthead and main navigation bar. In previous versions of HTML, we might have marked that up like so:

```
<div id="header">
  <h1>HTML5 <i>News-Press</i></h1>
  <h2>All the news that's fit to link</h2>
  <ul id="nav">
    <li><a href="#">World</a></li>
    <li><a href="#">National</a></li>
    <li><a href="#">Metro area</a></li>
    <li><a href="#">Sports</a></li>
    <li><a href="#">Arts &amp; Entertainment</a></li>
  </ul>
</div>
```

Our page ends with a footer element. Again, using HTML 4, our markup might look like this:

```
<div id="footer">
  <ul>
    <li><a href="#">Contact Us</a></li>
    <li><a href="#">Terms of Use</a></li>
    <li><a href="#">Privacy Policy</a></li>
  </ul>
  <p>No copyright 2013 HTML5 News-Press.</p>
</div>
```

HTML5, however, adds elements specifically for this purpose: `header`, `nav` and `footer`.

The `header` element functions as a header for the contents of a document segment. The `footer` functions as a footer for a document segment. Notice, I said *segment* and not *document* or *page*. Some elements are considered **sectioning elements**. They split a document into sections or chunks. One of these elements, of course, is the new `section` element. Other sectioning elements include `body`, `article`, `aside`, are `nav` as well. Here's the tricky part: each sectioning element may contain its own header and footer. It's a bit confusing, but the main point here is that a document may contain multiple header and footer elements.

```
<header>
  <h1>HTML5 <i>News-Press</i></h1>
  <h2>All the news that's fit to link</h2>
  <nav>
    <ul>
      <li><a href="#">World</a></li>
      <li><a href="#">National</a></li>
      <li><a href="#">Metro area</a></li>
      <li><a href="#">Sports</a></li>
      <li><a href="#">Arts &amp; Entertainment</a></li>
    </ul>
  </nav>
</header>
```

Here, we've wrapped our masthead and navigation in `header` tags. We've also swapped our `id="nav"` attribute and value for the `nav` element. Let's re-write our footer using HTML5's `footer` element.

```
<footer>
  <ul>
    <li><a href="#">Contact Us</a></li>
    <li><a href="#">Terms of Use</a></li>
    <li><a href="#">Privacy Policy</a></li>
  </ul>
  <p>No copyright 2013 HTML5 News-Press.</p>
</footer>
```

Here we've simply swapped `<div id="footer">` for `<footer>`. Our document bones are in place. Let's add some flesh.

The `article` Element

As defined by the HTML5 specification, the `article` element:

> "[R]epresents a complete, or self-contained, composition in a document, page, application, or site and that is, in principle, independently distributable or reusable, e.g. in syndication."

Magazine articles and blog posts are obvious examples of when an `article` element would be semantically appropriate. But you could also use it for blog comments. This element is appropriate for any almost content item that could be reused.

We can replace our `<div id="article">` start and end tags with `article` tags.

```
<article>
  <h1>Sky fall imminent says young chicken leader</h1>

  <p class="byline">
    <b class="reporter">Foxy Loxy</b>
    <i class="employment-status">Staff Writer</i>
  </p>

  <div class="aside">

    <h2>About Henny Penny</h2>

    <dl>
      <dt>Age</dt>
      <dd>32</dd>
      <dt>Occupation</dt>
      <dd>President, National Organization of Chickens</dd>

      <dt>Education</dt>
      <dd>B.A., Chicken Studies, Farmer University</dd>
      <dd>J.D., University of Cluckland</dd>
    </dl>

    <p>
      Penny joined the National Organization of Chickens in 2002
      ➥as a staff lobbyist after short, but effective career in
      ➥the Farmlandia senate. Penny rose through the
      ➥organization's ranks, serving as secretary, then vice-
      ➥president before being elected president by the group's
      ➥members in 2011.
    </p>

    <p>
      The National Organization of Chickens is an advocacy group
      ➥focused on environmental justice for chickens.
    </p>
  </div>

  <p>
    LONDON -- Henny Penny, young leader of the National
    ➥Organization of Chickens announced that the sky will fall
    ➥within the next week. Opponents criticize Penny,
    ➥suggesting that acorns are the more likely threat.
  </p>

  <p>
    Phasellus viverra faucibus arcu ullamcorper sodales. Curabitur
    ➥tincidunt est in imperdiet ultrices. Sed dignissim felis a
    ➥neque dignissim, nec cursus sapien egestas.
  </p>

  <div id="article-meta">
    <p class="reporter-contact">You may reach reporter Foxy Loxy
    ➥via email at foxy.loxy@html5newspress.com</p>
    <p class="contributor">Staff writer Turkey Lurkey contributed
    ➥to this report.</p>
    <p class="pubdate">Published:
```

```
    ➥<time>2013-07-11T09:00:00-07:00</time>.</p>
  </div>
</article>
```

The `article` element is an example of sectioning content, which means it may contain a header and a footer. If we think about it, our `<div id="article-meta">` could be considered a footer for our `article` element. How about we swap our `div` element tags for `footer` tags?

```
<footer id="article-meta">
  <p class="reporter-contact">You may reach reporter Foxy Loxy
  ➥via email at foxy.loxy@html5newspress.com</p>
  <p class="contributor">Staff writer Turkey Lurkey contributed
  ➥to this report.</p>
  <p class="pubdate">Published:
  ➥<time>2013-07-11T09:00:00-07:00</time>.</p>
</footer>
```

We are keeping our `id` attribute intact, however. This makes it easier to distinguish from other `footer` elements on the page if we add CSS or DOM scripting.

Think of the `aside` element as the HTML5 equivalent of newspaper or magazine sidebar. It denotes content that's *related to* the main article, but could stand alone. In our HTML 4 example, we used `<div class="sidebar">` to mark up our aside. However, the `aside` element offers more meaning and context. Let’s change our markup to use the `aside` element instead.

```
<aside>
  <h2>About Henny Penny</h2>
    <dl>
      <dt>Age</dt>
      <dd>32</dd>

      <dt>Occupation</dt>
      <dd>President, National Organization of Chickens</dd>

      <dt>Education</dt>
      <dd>B.A., Chicken Studies, Farmer University</dd>
      <dd>J.D., University of Cluckland</dd>
    </dl>
    <p>
      Penny joined the National Organization of Chickens in 2002
      ➥as a staff lobbyist after short, but effective career in
      ➥the Farmlandia senate. Penny rose through the
      ➥organization's ranks, serving as secretary, then vice-
      ➥president before being elected president by the group's
      ➥members in 2011.
    </p>

    <p>
      The National Organization of Chickens is an advocacy group
      ➥focused on environmental justice for chickens.
    </p>
</aside>
```

Putting It Together

Our finished HTML5 document looks like this.

```
<!DOCTYPE html>
<head>
  <meta charset="utf-8">
  <title>HTML5 News-Press</title>
</head>
<body>
  <header>
    <h1>HTML5 <i>News-Press</i></h1>
    <h2>All the news that's fit to link</h2>
    <nav>
      <ul>
        <li><a href="#">World</a></li>
        <li><a href="#">National</a></li>
        <li><a href="#">Metro area</a></li>
        <li><a href="#">Sports</a></li>
        <li><a href="#">Arts &amp; Entertainment</a></li>
      </ul>
    </nav>
</header>

<article>
  <h1>Sky fall imminent says young chicken leader</h1>

  <p class="byline">
    <b class="reporter">Foxy Loxy</b>
    <i class="employment-status">Staff Writer</i>
  </p>

  <aside>

    <h2>About Henny Penny</h2>

    <dl>
      <dt>Age</dt>
      <dd>32</dd>

      <dt>Occupation</dt>
      <dd>President, National Organization of Chickens</dd>

      <dt>Education</dt>
      <dd>B.A., Chicken Studies, Farmer University</dd>
      <dd>J.D., University of Cluckland</dd>
    </dl>

    <p>
      Penny joined the National Organization of Chickens in 2002
      ➥as a staff lobbyist after short, but effective career in
      ➥the Farmlandia senate. Penny rose through the
      ➥organization's ranks, serving as secretary, then vice-
      ➥president before being elected president by the group's
      ➥members in 2011.
    </p>

    <p>
      The National Organization of Chickens is an advocacy group
      ➥focused on environmental justice for chickens.
    </p>
  </aside>

  <p>
    LONDON -- Henny Penny, young leader of the National
    ➥Organization of Chickens announced that the sky will fall
    ➥within the next week. Opponents criticize Penny,
    ➥suggesting that acorns are the more likely threat.
  </p>

  <p>
    Phasellus viverra faucibus arcu ullamcorper sodales. Curabitur
    ➥tincidunt est in imperdiet ultrices. Sed dignissim felis a
    ➥neque dignissim, nec cursus sapien egestas.
  </p>

  <footer id="article-meta">
    <p class="reporter-contact">You may reach reporter Foxy Loxy
    ➥via email at foxy.loxy@html5newspress.com</p>
    <p class="contributor">Staff writer Turkey Lurkey contributed
    ➥to this report.</p>
    <p class="pubdate">Published:
    ➥<time>2013-07-11T09:00:00-07:00</time>.</p>
  </footer>

</article>

<footer>
  <ul>
    <li><a href="#">Contact Us</a></li>
    <li><a href=#">Terms of Use</a></li>
    <li><a href=#">Privacy Policy</a></li>
  </ul>
  <p>No copyright 2013 HTML5 News-Press.</p>
</footer>
```

The point of these new elements is to have a standardized way to describe document structures. HTML is, at heart, a language for exchanging and repurposing documents. Using these structural elements means that the same document can be published as a web page and also syndicated for e-book readers without having to sort through a jumble of arbitrary `div` elements and `id` attributes.

The `section` Element

HTML5 also introduces the `section` element, which is used to define segments of a document that are neither a header, footer, navigation, article, or aside. It's more specific than our old friend, the `div` element, but more generic than `article`.

Let's use the `section` element to mark up the *HTML5 News-Press* home page, shown in Figure 3.2.

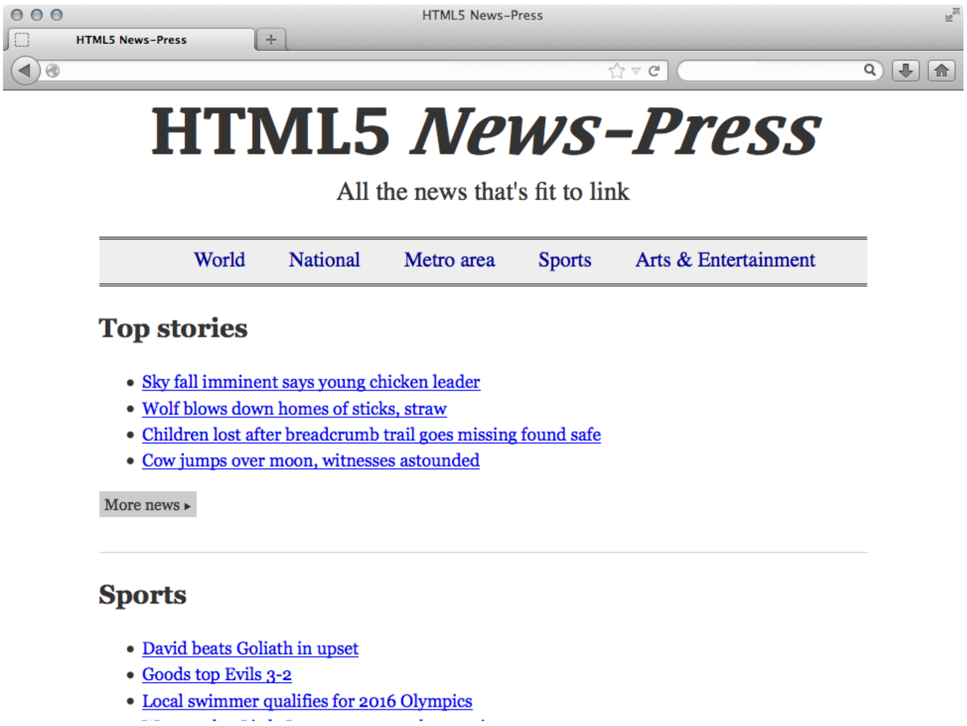

HTML5 News-Press
HTML5 News-Press
All the news that's fit to link
World
National
Metro area
Sports
Arts & Entertainment
Top stories
Sky fall imminent says young chicken leader
Wolf blows down homes of sticks, straw
Children lost after breadcrumb trail goes missing found safe
Cow jumps over moon, witnesses astounded
More news ▸
Sports
David beats Goliath in upset
Goods top Evils 3-2
Local swimmer qualifies for 2016 Olympics

Figure 3.2. HTML5 News-Press home page

Here we have three sections of news stories: "Top Stories," "Sports," and "Business." In HTML 4, the `div` elements is the clear choice to define these sections because it's our only choice. But in HTML5, we have the somewhat more descriptive `section` element.

```
<section id="topstories">
  <h1>Top stories</h1>

  <ul>
    <li><a href="#">Sky fall imminent says young chicken
    ➥leader</a></li>
    <li><a href="#">Wolf blows down homes of sticks, straw</a></li>
    <li><a href="#">Children lost after breadcrumb trail goes
    ➥missing found safe</a></li>
    <li><a href="#">Cow jumps over moon, witnesses
    ➥astounded</a></li>
  </ul>

  <footer>
```

```
    <p><a href="#" class="mlink">More news</a></p>
  </footer>
</section>

<section id="sports">
  <h1>Sports</h1>

  <ul>
    <li><a href="#">David beats Goliath in upset</a></li>
    <li><a href="#">Goods top Evils 3-2</a></li>
    <li><a href="#">Local swimmer qualifies for 2016
    ➥Olympics</a></li>
    <li><a href="#">Woonsocket Little League team reaches
    ➥semis</a></li>
  </ul>

  <footer>
    <p><a href="#" class="mlink">More sports</a></p>
  </footer>
</section>

<section id="business">
  <h1>Business</h1>
  <ul>
    <li><a href="#">BigWidgets, Ltd. expansion to create 3,000
    ➥jobs</a></li>
    <li><a href="#">U.S. dollar, Euro achieve parity</a></li>
    <li><a href="#">GiantAirline and Air Humongous to merge</a></li>
    <li><a href="#">WorldDomination Banc deemed 'too big to
    ➥fail'</a></li>
  </ul>

  <footer>
    <p><a href="#" class="mlink">More business</a></p>
  </footer>
</section>
```

`div` Versus `section`

While these new sectioning elements give us better semantics and meaning, they also bring a touch of confusion. One question you may be asking is: "Is it still okay to use the `div` element?

The short answer is "Yes." After all, `div` is still a valid HTML5 element. But `div` has little semantic value. It doesn't tell us anything about what it contains, so use it for those rare instances when another semantically-relevant tag doesn't exist. For example, you may find that you need to add an extra element as a "wrapper" to ease styling. Or perhaps you want to group several French-language elements within an English-language document. Enclosing those elements with `<div lang="fr">` and `</div>` is certainly appropriate. It most other cases, it's better to use `section`, `header`, `nav`, `footer`, `article` or `aside`.

Other Document Elements

New sectioning elements aren't the only new document-related elements introduced in HTML5. In this chapter, we'll look at the following elements:

- `figure` and `figcaption` for defining figures, charts, and diagrams
- `main` which explicitly defines the predominant portion of a document's content

`figure` and `figcaption`

If a document was accompanied by a chart or diagram, in HTML 4, we might have used a combination of `div` and `p` elements to mark it up as shown below.

```
<div class="graph" id="figure1">
  <img src="graph.jpg" alt="Price of chocolate since 2008">
  <p class="caption">Figure 1: Chocolate has increased in price
  ➥by 200% since 2008.</p>
</div>
```

That's an acceptable way to do it. But what happens when we want to read it on our fancy-pants ereader that displays HTML documents using a book-style layout? Using `<div class="chart">` doesn't tell us much about what this chunk of information is and how it should be displayed.

In this case, we should use the `figure` element. It gives us a way to mark up charts and captions and make them independent of document flow. How about we modify our markup a bit?

```
<figure class="graph" id="figure1">
  <img src="graph.jpg" alt="Price of chocolate since 2008">
  <p class="caption">Figure 1: Chocolate has increased in price
  ➥by 200% since 2008.</p>
</figure>
```

Now our e-reader knows this is a chart, and can display it in a suitable way. We've kept the class and ID attributes. The former comes in handy for CSS styling — perhaps we want to display graphs differently than diagrams. The latter makes it easy to link to our figure.

We still have `<p class="caption">` though, don't we? How about we use the `figcaption` element instead? `figcaption` serves as a caption or legend for its sibling content. It isn't required, but if included, `figcaption` needs to be either the first child or last child of a `figure` element. Let's change our markup once more:

```
<figure class="graph" id="figure1">
  <img src="graph.jpg" alt="Price of chocolate since 2008">
  <figcaption>Figure 1: Chocolate has increased in price
  ➥by 200% since 2008.</figcaption>
</figure>
```

`main` Element

The `main` element is one of the newest elements in HTML5. It actually isn't in the 2011 version of the specification published by the World Wide Web Consortium, but it is a part of the HTML 5.1 specification.

Unsurprisingly, `main` should be used to clarify the main part of a document or application. By "main part," of a document, we mean content that:

- is unique to the document or application
- doesn't include elements, such as a global header or footer, that are shared across a site

To date, only Chrome 26+, Firefox 21+, and Opera 15+ support the `main` element. Support should come to Internet Explorer and Safari in a future release.

What's the use case for `main`? Some browsers, such as Safari and Firefox for Android, offer a "reader mode" that strips away headers, footers, and ads. These browsers currently use heuristics—an educated guess—to determine the main content of a document. Using `main` makes it clear to the browser which segment of the page it should focus on. Browsers can also use the `main` element as a hook for accessibility features, such the ability to skip navigation.

Let's take one last look at our news article markup, this time including the `main` element.

```
<!DOCTYPE html>
<html lang="en-us">
<head>
  <meta charset="UTF-8">
  <title>HTML5 News-Press</title>
  <link rel="stylesheet" href="s.css" media="screen">
</head>
<body>
<div id="wrapper">
  <header>
    <h1>HTML5 <i>News-Press</i></h1>
    <h2>All the news that's fit to link</h2>
    <nav>
      <ul>
        <li><a href="#">World</a></li>
        <li><a href="#">National</a></li>
        <li><a href="#">Metro area</a></li>
        <li><a href="#">Sports</a></li>
        <li><a href="#">Arts &amp; Entertainment</a></li>
      </ul>
    </nav>
  </header>

  <main>
    <article>
      <header>
        <h1>Sky fall imminent says young chicken leader</h1>
        <p class="byline">
          <b class="reporter">Foxy Loxy</b>
          <i class="employment-status">Staff Writer</i>
        </p>
      </header>
      <aside>
        <h2>About Henny Penny</h2>
        <dl>
          <dt>Age</dt>
          <dd>32</dd>

          <dt>Occupation</dt>
          <dd>President, National Organization of Chickens</dd>

          <dt>Education</dt>
          <dd>B.A., Chicken Studies, Farmer University</dd>
          <dd>J.D., University of Cluckland</dd>
        </dl>
        <p>
          Penny joined the National Organization of Chickens in 2002
          ➥as a staff lobbyist after short, but effective career in
          ➥the Farmlandia senate. Penny rose through the
          ➥organization's ranks, serving as secretary, then vice-
          ➥president before being elected president by the group's
          ➥members in 2011.
        </p>

        <p>
          The National Organization of Chickens is an advocacy group
          ➥focused on environmental justice for chickens.
        </p>
      </aside>

      <p>
        LONDON -- Henny Penny, young leader of the National
        ➥Organization of Chickens announced that the sky will fall
        ➥within the next week. Opponents criticize Penny,
        ➥suggesting that acorns are the more likely threat.
      </p>

      <p>
        Phasellus viverra faucibus arcu ullamcorper sodales.
        ➥Curabitur tincidunt est in imperdiet ultrices. Sed
        ➥dignissim felis a neque dignissim, nec cursus sapien
        ➥egestas.
      </p>

    </article>
  </main>
```

```
  <footer>
    <ul>
      <li><a href="#">Contact Us</a></li>
      <li><a href="#">Terms of Use</a></li>
      <li><a href="#">Privacy Policy</a></li>
    </ul>
    <p>No copyright 2013 HTML5 News-Press.</p>
  </footer>

</div>
</body>
</html>
```

Now we have a document that's more accessible and easier for browsers of all kinds to consume and display.

Chapter 4

HTML5 Forms

HTML5 forms are a leap forward from those in previous versions of HTML. We now have a half dozen new input states or types, such as `range` and `url`; we have new attributes that let us require fields, or specify a particular format; we have an API that lets us constrain and validate input; and finally, we have new form elements, such as `datalist`, that let us create engaging user interfaces without heavy JavaScript libraries or plugins.

Unfortunately, not every browser supports all of these features just yet. For now we still need to use JavaScript libraries, and **polyfills**[1] (downloadable code which provides facilities that are not built into a web browser) as fall back strategies.

The best way to understand HTML5 forms is to build one. Let's try building a form that collects story suggestions for the HTML5 News-Press site from the previous chapter. We'll need to gather the following information with our form:

- Name
- City of residence

[1] http://remysharp.com/2010/10/08/what-is-a-polyfill/

- Email address
- A telephone number
- A URL where we can learn more (if there is one)
- The actual story idea

At the very least, we'll want to require the user to provide a first name, email address, and their story idea.

Starting an HTML5 Form

To build our HTML form, we'll need to start with an opening `form` element tag. Because we want to submit this form to a server-side processing script, we'll need to include two attributes.

- `action`: the URL of the script
- `method`: the HTTP request method to use, sometimes `GET`, but usually `POST`

Since this could be a lengthy message, we'll use `POST` rather than `GET`. `GET` is better suited to short key-value pairs, such as search boxes. Other HTTP methods, such as `PUT`, `HEAD`, or `DELETE` may also be used with forms, but most of the time you'll use `GET` or `POST`.

It's worth noting here that `application/x-www-form-urlencoded` is the default content type value for form data. We could also explicitly set it using the `enctype` attribute, but we don't have to.

We could also set our `enctype` attribute to `multipart/form-data` as shown below:

```
<form action="/script" method="post" enctype="multipart/form-data">
```

Either is fine for sending text. If we wanted to upload files, however, we would need to use `enctype="multipart/form-data"`.

The input Element

The input element is the most commonly used element for creating form controls. An input tag typically includes the following attributes.

- name: the name of the field
- type: indicates what kind of input control to display
- id: a unique identifier for the field
- value: sets a default value for the field

Of these, only name is required in order for our form to send data. Each name attribute becomes a key or field name for our server-side script. That said, in most cases, you'll also want to set the type attribute.

There are about a dozen possible values for the type attribute, most of which we'll cover in this chapter. Each type value corresponds to a different kind of user interface control and set of validation constraints. The most liberal value for the type attribute—and the default state of the input element—is text.

Collecting Names

People names and place names are usually a mix of alphanumeric characters, spaces, and punctuation marks. For this reason, we'll use the text input state for those fields. Let's add form fields for the letter writer's name and city of residence. Since we want to require the user to provide a name, we'll also add a required attribute.

```
<p>
  <label for="your_name">Your name:</label>
  <input type="text" name="your_name" id="your_name" required>
</p>

<p>
```

```
  <label for="city">City of residence:</label>
  <input type="text" name="city" id="city">
</p>
```

id and name attributes

The `id` attribute may, but *does not have to* be, the same as the `name` attribute.

Using Form Labels

We've added an unfamiliar element here: `label`. The `label` element in an HTML form works just like the label on a paper form. It tells the user what to enter in the field. In order to associate a label with a form control, the `label` must have a `for` attribute that matches the `id` attribute of its form field. Or you could place the form control inside of the `label` element.

```
<label>Your name:
  <input type="text" name="your_name" id="your_name" required>
</label>
```

Using the `for` and `id` attributes, however, offers a little more flexibility for page layouts.

Why not just use text without wrapping it in a `label` element? Using `label` increases the usability of the web for those with physical or cognitive challenges. Screen-reading software, for example, uses labels to help low-vision users in filling out forms. It's an accessibility feature that's baked into HTML.

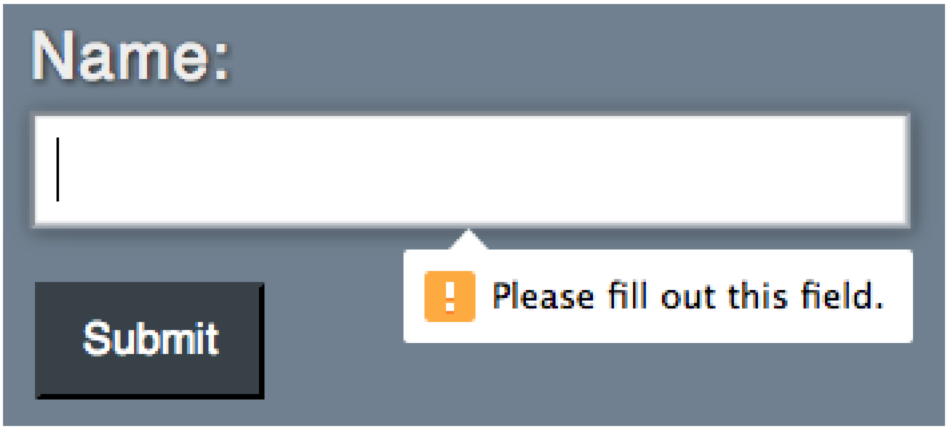
Requiring Form Fields

One of the great improvements of HTML5 over previous versions is native form validation. By adding the `required` attribute, we are asking the browser to make sure this field has been filled out before submitting the form.

Empty Attributes

The `required` attribute is an example of an empty attribute. Its presence or absence determines whether that value is set.

If the `your_name` field is empty when the user submits our form, Chrome, Opera, and Internet Explorer 10+ will prevent submission and alert the user, as shown in Figure 4.1. No DOM scripting is necessary.



Please fill out this field.

Figure 4.1. A form submission error message in Chrome

Notice that I didn't mention Safari. For better or worse, Safari versions 6.0.5 and older don't provide native user interface feedback. It does support HTML5 validation, but we'll still need to use DOM scripting and CSS to alert the user about form submission. We'll discuss one way to do this in the Validation API section.

Styling Required Forms

You may want to visually indicate which fields are required and which aren't using CSS. There are two CSS selectors we can use to target required fields.

1. Using the attribute selector `[required]`.

2. Using the `:required` pseudo-class.

The `:required` pseudo-class is a CSS Selectors, Level 4[2] selector, but support is available in the latest version of every major browser. CSS Level 4 selectors also adds an `:optional` pseudo-class that we could use instead to target the input fields that aren't required.

[2] http://www.w3.org/TR/selectors4/

To target older browsers that lack CSS4 selector support, use the attribute selector. For example, if we want to add a 1 pixel red border around our required fields, we could add the following to our CSS.

```
input[required], input:required
{
  border: 1px solid #c00;
}
```

Collecting Email Addresses, Phone Numbers, and URLs

We'll want to let our tipster know that we've received their input. That means our form needs fields for the email address and phone number. We also want to collect URLs where we can learn more information about this story idea, so our form will also need a field for the URL.

With previous versions of HTML, we'd use a text field for all of these and validate the data with JavaScript. HTML5, however, defines three new input types for this purpose: `email`, `tel` and `url`.

Let's add an email field to our form. We'll also make it required.

```
<p>
  <label for="email">E-mail address</label>
  <input type="email" name="email" id="email" required>
</p>
```

Using the `email` type tells the browser to check this field for a valid email address. It can't, of course, tell whether the address can receive mail. But it will check that the input for this field is syntactically valid. If a user enters an invalid address, most browsers will alert the user when he or she submits the form.

You may also want to let the user provide multiple email addresses. In that case, use the `multiple` attribute. The user can then enter one or more e-mail addresses, each separated by a comma.

Any time you permit multiple values for one input field, it's a good idea to indicate that it's allowed with a label or explanatory text.

On Validation

Although native data validation is a part of HTML5, you should still use server-side validation and escape any output that originates with the user. Not all browsers support HTML5, and not all input sent to your script will come from your form.

Phone numbers are another story. Email addresses adhere to a standard format, but telephone numbers do not. In the United Kingdom, phone numbers may be up to 11 digits long. In the United States, they are no more than 10 digits. Some countries have phone numbers that contain 13 digits. The formatting of phone numbers also varies by country. Inconsistent lengths and formats make native phone number validation difficult. As a result, the specification doesn't define an algorithm for doing so.

Let's add a telephone field to our form. To do that, we do need to add an input field and set the value of its type attribute to `tel`. We won't make it required.

```
<p>
  <label for="telephone">Telephone number:</label>
  <input type="tel" name="telephone" id="telephone">
</p>
```

The big advantage of using `tel` instead of `text` is to trigger a telephone input screen in browsers that support it.

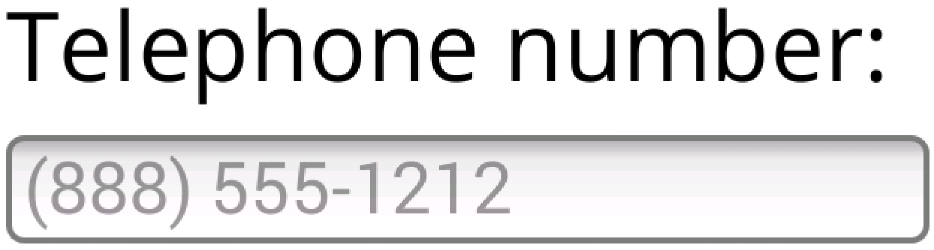


Figure 4.2. The tel input type in Firefox Mobile

Though `tel` doesn't give us automatic validation, we can shape user input using two attributes:

- `placeholder`, which offers a 'hint' to the user about what format this field expects.

- `pattern`, which sets a regular expression pattern that the browser can use to validate input.

Our imaginary newspaper is based in the United States, and has a US-based audience. We'll reflect that in our attribute values.

```
<p>
  <label for="telephone">Telephone number:</label>
  <input type="tel" name="telephone" id="telephone"
  ➥placeholder="(000) 000-0000"
  ➥pattern="\([2-9][0-9]{2}\) [0-9]{3}-[0-9]{4}">
</p>
```

For our `placeholder` attribute, we've just added text that reflects the expected format for this phone number.

Placeholder Text

Placeholder text is not a replacement for the `label` element. Provide a label for each input field, even if you use the `placeholder` attribute

For `pattern`, we've used a regular expression. This attribute provides a format or pattern that the input must match before the form can be submitted. Almost any valid JavaScript regular expressions can be used with the `pattern` attribute. Unlike with JavaScript, you can't set global or case-insensitive flags. To allow both upper and lower case letters, your pattern must use `[a-zA-Z]`.The `pattern` attribute itself may be used with `text`, `search`, `email`, `url` and `telephone` input types.

Regular Expressions

Regular expressions are a big, complex topic and, as such, they're beyond the scope of this book. For a more complete reference, consult WebPlatform.org's documentation[3].

The `url` input type works much the same way as `email` does. It validates user input against accepted URL patterns. Protocol prefixes such as `ftp://` and `gopher://` are

[3] http://docs.webplatform.org/wiki/concepts/programming/javascript/regex

permitted. In this case, we want to limit user input to domains using the `http://` and `https://` protocols. So we'll add a `pattern` attribute here as well.

```
<p>
  <label for="url">
    Please provide a web site where we can learn more (if
  ➥applicable):
  </label>
  <input type="url" name="current_site" id="current_site"
  ➥placeholder="http://www.example.com/"
  ➥pattern="http(|s)://[-0-9a-z]{1,253}\.[.a-z]{2,7}">
  </p>
```

We've also added `placeholder` text as a cue to the user about what we'd like them to tell us. Altogether, your form should resemble the one below:

```
<form action="./script" method="POST">
  <p>
    <label for="your_name">Your name:</label>
    <input type="text" name="your_name" id="your_name">
  </p>

  <p>
    <label for="city">City of residence:</label>
    <input type="text" name="city" id="city">
  </p>

  <p>
    <label for="email">
      E-mail address
      (separate multiple e-mail addresses with a comma):
    </label>
    <input type="email" name="email" id="email"
    ➥placeholder="jane.doe@example.com" multiple >
  </p>

  <p>
    <label for="tel">Telephone number:</label>
    <input type="tel" name="phone_number" if="phone_number"
    ➥placeholder="(000) 000-0000"
    ➥pattern="\([2-9][0-9]{2}\) [0-9]{3}-[0-9]{4}">
  </p>

  <p>
```

```
    <label for="url">
      Please provide a web site where we can learn more (if
    ➥applicable):
    </label>
    <input type="url" name="current_site" id="current_site"
    ➥placeholder="http://www.example.com/"
    ➥pattern="http(|s)://[-0-9a-z]{1,253}\.[.a-z]{2,7}">
  </p>

  <p>
    <label for="project">Tell us your story idea:</label>
    <textarea name="story_idea" id="story_idea"
    ➥placeholder="Briefly tell us what we should write about and
    ➥why."
    ➥maxlength="2000"></textarea>
  </p>

  <p>
    <button type="submit">Send it!</button>
  </p>
</form>
```

Uploading Files

The `file` input type is not new to HTML. We've been able to upload files since HTML 3.2. What *is* new, however, is the `multiple` attribute, which lets us upload multiple files using one form field. In this section, we'll build a form that lets users upload audio files.

First we'll need to create a start tag for the `form` element.

```
<form action="/script" method="post" enctype="multipart/form-data">
```

As with our previous form, our start tag has `action` and `method` attributes. But note that the value of its `enctype` attribute is `multipart/form-data`. Again, when uploading binary data, we must use the `multipart/form-data` encoding type.

Next, we need to add an `input` tag, and set the value of its `type` attribute to `file`. We'll name it `upload`, but you can choose almost any name you like. To permit multiple file uploads, we'll need to add the `multiple` attribute.

```
<input type="file" name="upload" id="upload" multiple>
```

PHP Form Keys

PHP requires form keys with multiple values to use square bracket array syntax. If you're using PHP to handle your forms, append square brackets to the name (for example: `name="upload"` would become `name="upload[]"`).

We can also restrict what files can be uploaded in the browser with the `accept` attribute. The value of `accept` may be any of the following:

- `audio/*`, `video/*`, `image/*`
- a valid MIME type such as `image/png` or `text/plain`
- a file extension that begins with '.'

You may include multiple `accept` values; separate them with a comma. Let's update our form field to accept only MP3 and Ogg Vorbis files.

```
<input type="file" multiple name="upload" id="upload"
➥accept=".mp3,.ogv">
```

We'll finish up our form with a submit button and closing form tag:

```
<form action="/script" method="post" enctype="multipart/form-data">
  <p>
    <label for="upload">Your file(s):</label>
    <input type="file" multiple name="upload" id="upload"
    ➥accept=".mp3,.ogv">
  </p>
  <p>
    <button type="submit">Upload!</button>
  </p>
</form>
```

When submitted, our server-side script will save those files, and return a "thank you" message.

Take Appropriate Precautions

You can't rely on browser-based validation or restrictions. Take appropriate precautions, and make sure that your file uploads are being placed in a directory that is not web-accessible.

The `datalist` Element

With the `datalist` element, we can add a predefined set of options to any form input control. Let's take a look at how we go about this. First, we'll create a `datalist` of destination options for a fictitious airline:

```
<datalist id="where_we_fly">
  <option>Accra, Ghana</option>
  <option>Paris, France</option>
  <option>Melbourne, Australia</option>
  <option>Lima, Peru</option>
  <option>Los Angeles, United Sates</option>
  <option>Kuala Lumpur, Malaysia</option>
</datalist>
```

Now we can associate it with an `input` field using the `list` attribute.

```
<p>
  <label for="destination">Where would you like to go?</label>
  <input type="text" name="destination" id="destination" value=""
  ➥list="where_we_fly">
</p>
```

In browsers that support the `datalist` element, the code above will associate a predefined list of options with the `input` element. When the user enters text, matching entries are displayed in the list below the field as shown in Figure 4.3.

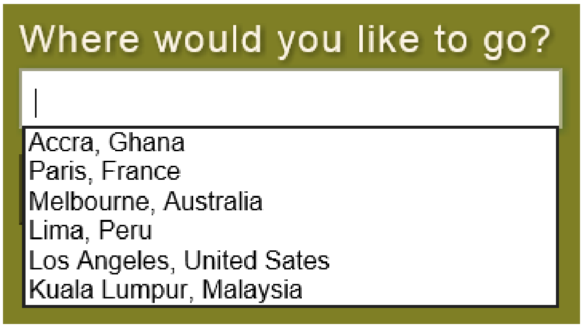


Figure 4.3. datalist in IE

In browsers without support for `datalist`, the text input field will behave normally. Although data lists may, in theory, be associated with other input types, not all browsers support this.

Other Input Types

We've already discussed several input types in this chapter, but there are a few more that we'll cover in this section.

- `search`
- `range`
- `number`
- `color`
- `datetime` and `datetime-local`
- `date`
- `month`
- `week`
- `time`

Aside from the `range` input type and `search`, support for these types varies wildly. Some browsers may have full support for one input type, complete with a user interface control, but lack another one entirely.

It's possible to determine whether a browser supports a particular input type by testing the value returned by its type attribute. If a browser doesn't support a particular type, the value of its type attribute will default to `text`. For example, consider the following range input:

```
<input type="range" value="" id="slider">
```

We could test for browser support using the following bit of JavaScript code.

```
var hasRange = function( elID ){
  return document.getElementById( elID ).type == 'range';
}
```

In browsers that do not support `range`, the function above will return `false`. Otherwise, it will return `true`. Libraries such as Modernizr[4] make it easier to check for support.

`input type="search"`

For the most part, `search` operates like the `text` input type. It merely provides a type that can be visually distinct from `text` boxes. For example, in Safari, Chrome, and Opera 15 on Mac OS X, search input fields have rounded corners.

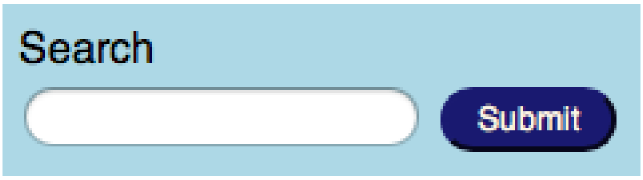


Figure 4.4. The Search input type in Safari

`input type="range"`

The `range` input type presents the user with a slider control that's well suited to approximate value inputs between an upper and lower boundary. By default, it's a horizontal control, as shown in Figure 4.5. However with some CSS (`transform: rotate(-90deg)`), you can also display range inputs vertically.

[4] http://modernizr.com/

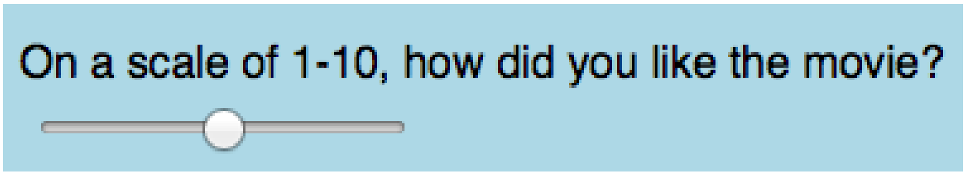


Figure 4.5. The range input type

By default, the upper and lower boundaries of the range type are 0 and 100. Change this by setting the `min` and `max` attributes. You can also control the 'smoothness' and precision of the thumb position using the `step` attribute as shown below.

```
<input type="range" value="" min="0" max="100" step="10">
```

Every time the user moves the thumb on this range input control, the value of the range will increase or decrease by 10 between 0 and 100. You can also control precision by associating a `datalist` element with the `range` input. Each option will be rendered as a 'notch' along the width of the range in browsers that support it—to date, that's Chrome and Opera 15.

Unfortunately, range isn't supported in Internet Explorer 9 and older, or Firefox 22 and older. In those browsers, the form control will be a text box instead of a range element.

`input type="number"`

The `number` type is another form control type for numeric input values. According to the specification, any floating point number is a valid value. In practice, though, things are little more complicated.

By default, the `number` input type only accepts integers. Entering 4.2776, for example, will cause a validation error in conforming browsers, such as shown in Figure 4.6.

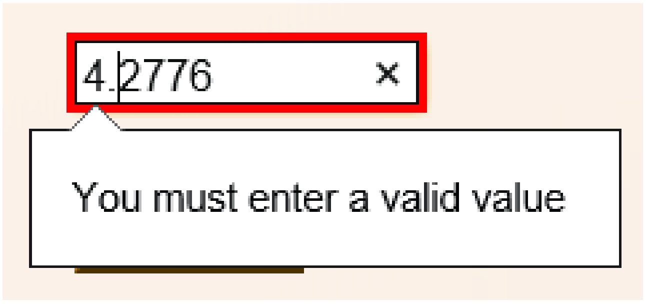


Figure 4.6. An error when entering a floating point number

In order to accept floating point values, we need to set the `step` attribute. In browsers with incremental arrow controls, such as shown in Figure 4.7, `step` controls how much the number is incremented or decremented with each press of the arrow button.

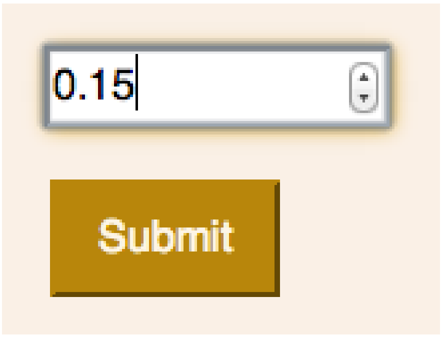


Figure 4.7. Decimal values with the step attribute

For example, when `step="0.5"`, both 1.5 and 98 are valid values, but 88.1 is not. When `step=".01"`, however, 88.1, 1.5, and 98 are all valid values, as is 3.14. In a way, you can use the `step` attribute to control the floating point precision of your numbers.

```
<!-- Increments number by 0.5 -->
<input type="number" name="num" id="num" value=""  step=".05">

<!-- Increments number by .01; precision to the hundredth -->
<input type="number" name="num" id="num" value="" step=".01">
```

```
<!-- Increments number by 0.001; precision to the thousandth -->
<input type="number" name="num" id="num" value="" step=".001">
```

In order to make our 4.2776 value an accepted one, we would need to use set our `step` attribute to .0001. Unfortunately, this workaround does not work in Opera 12 and older versions.

Date and Time Inputs

Finally, let's look at the date and time input types. There are six of them, listed below.

- `datetime`: Select a date and time as a global, forced-UTC string
- `datetime-local`: Select a date and time in the user's local time zone
- `date`: Select a single date with a time component of midnight UTC
- `month`: Select a month and year
- `week`: Select a week and year
- `time`: Select a time in hours and minutes

Browsers that support these types will display a time picker widget (for the `time` type), a date picker widget (for `date`, `month`, and `week`), or both (`datetime` and `datetime-local`), as shown in Figure 4.8.

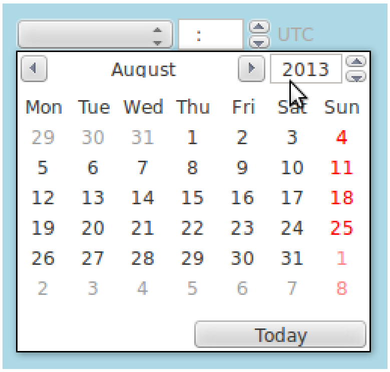


Figure 4.8. The time picker widget in Opera

The `datetime` and `datetime-local` input types are subtly different. The former treats all input as a UTC (coordinated, universal) date and time. Browsers may indicate to the user that this is a UTC time input—as Opera version 12 does—or it may display a localized user interface, and convert the time under the hood.

Chrome and Opera 15 support all but `datetime`. Safari as of version 6.0.5, Firefox as of version 25, and Internet Explorer as of version 10 do not support any of these types.

`input type="color"`

With the `color` input type, we can add a native color picker control to our applications. What the color picker looks like depends on the browser and operating system. However in all cases, only six-digit hexadecimal color values are valid. This means that you can't specify a color with transparency as you can with CSS. The default value for the `color` type is `#000000`.

Unfortunately, the `color` input type is only supported by Chrome and Opera 11 and 12 (but not version 15.0). It's not ready for prime time, but is mentioned here for completeness.

Chapter 5

HTML5 Multimedia: Audio and Video

Perhaps the biggest change in HTML5 is its multimedia capabilities. HTML5 brings with it native audio and video, and *almost* replaces the plugins of the old web. I say "almost" because browser vendors have yet to agree on a default format for web audio and video.

We'll talk about cross-browser support later in this chapter. First let's look at the bare minimum necessary to add video to your web page: a `video` tag and a `src` attribute.

```
<video src="path_to_file.video"></video>
```

In HTML5, that closing tag is required. If you're using XHTML syntax, you can self-close it instead like so: `<video src="path_to_file.video" />`. The `audio` element is almost the same. All that's required is the opening `audio` tag, a `src` attribute and a closing `</audio>` tag.

```
<audio src="path_to_file.audio"></audio>
```

Again, if we were using XHTML5 syntax, we could self-close our tag instead: `<audio src="path_to_file.audio" />`.

Adding Controls

Unfortunately, as shown in Figure 5.1, our snippet from above won't do anything besides add the media file. We won't be able to play our video, because it will be stuck at the first frame. We won't even know that there's audio on the page.

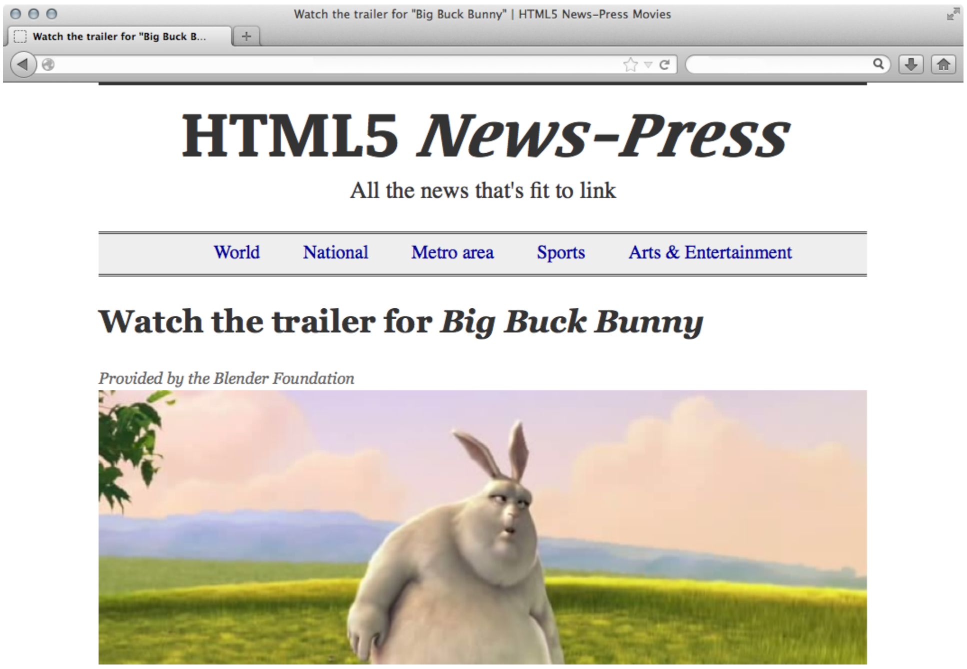

Watch the trailer for "Big Buck Bunny" | HTML5 News-Press Movies
Watch the trailer for "Big Buck B...
HTML5 *News-Press*
All the news that's fit to link
World
National
Metro area
Sports
Arts & Entertainment
Watch the trailer for *Big Buck Bunny*
Provided by the Blender Foundation

Figure 5.1. An example of a `video` element without controls in Firefox. Image from *Big Buck Bunny* by the Blender Foundation[1].

What we need are some controls: perhaps a play button and a scrubbable progress bar. We might also want a timer that reveals how much of the media has played, and a volume control. Luckily for us, browsers have these built-in to their audio and video support. We can activate them by adding the `controls` attribute.

```
<video src="path_to_file.video" controls>
```

[1] http://bigbuckbunny.org/

The `controls` attribute tells the browser that we want playback controls to be available for this media instance. In most browsers, this means the user will see a play and pause button, elapsed time indicator, and volume control, as shown in Figure 5.2. The player may also include a button that allows the user to toggle between full-screen and original size. What these default controls look like depends on the browser.

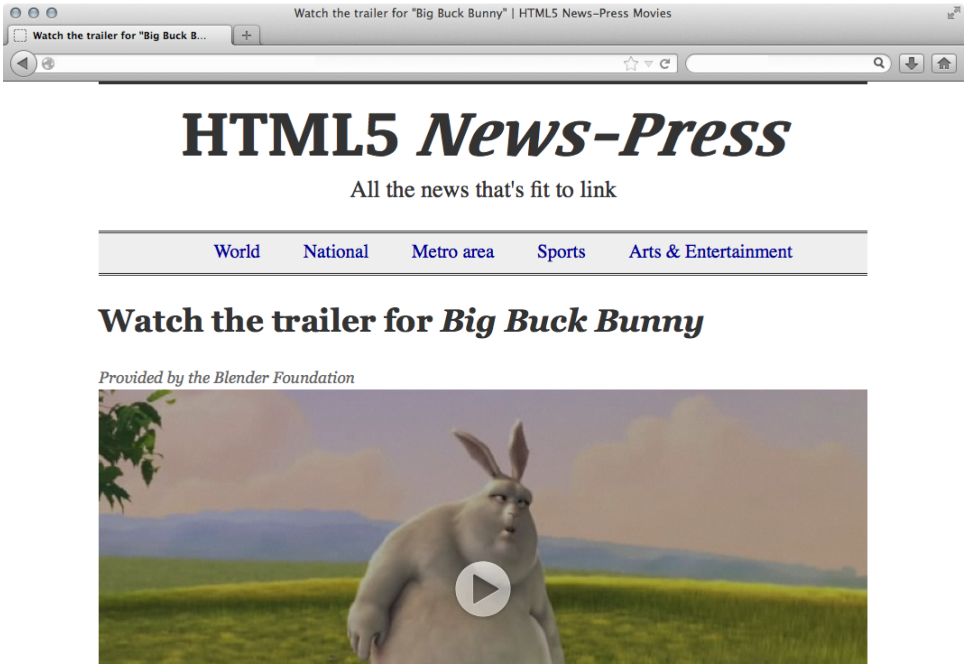

Watch the trailer for "Big Buck Bunny" | HTML5 News-Press Movies
Watch the trailer for "Big Buck B...
HTML5 News-Press
All the news that's fit to link
World
National
Metro area
Sports
Arts & Entertainment
Watch the trailer for Big Buck Bunny
Provided by the Blender Foundation

Figure 5.2. An example of a `video` element with controls in Firefox. Image from *Big Buck Bunny* by the Blender Foundation[2].

`controls` is another example of an empty attribute. We could also use `controls="true"` or `controls="controls"` if we were using XML syntax. Adding the `controls` attribute, regardless of its value, will make the controls visible. Using `controls=false` will not hide them.

Autoplaying and Looping Media

Perhaps we want to use a short video clip as a background element. We might want to create an "art installation" experience in which a video plays and re-plays auto-

[2] http://bigbuckbunny.org/

matically. Not a problem. We can do this by adding the `autoplay` and `loop` attributes to our video or audio tag.

```
<video src="path_to_file.video" autoplay loop></video>

<audio src="path_to_file.audio" autoplay loop></audio>
```

With `autoplay`, our media will begin as soon as the browser has received enough data to start playback. When the audio or video ends, `loop` tells the browser to restart the media file from the beginning.

Use `autoplay` with Caution

Some audio and video can be embarrassingly bad or just embarrassing if heard. Do your audience a favor: silence auto-playing media with the `muted` attribute.

```
<video src="annoying.video" autoplay loop muted></video>
```

Video-only Attributes

Although most attributes that apply to the `video` tag also apply to `audio`, a few—related to visual display—do not:

- `height`: Sets the height of the video player.
- `width`: Sets the width of the video player.
- `poster`: Specifies an image to display prior to video playback.

Place Holding with `poster`

A poster image acts as a placeholder for a video. It's typically a still image from the video, though it could be a company logo, title screen, icon, or some other image. Once the page loads, visitors will see the poster image until the video begins playback.

To add a poster image, add a `poster` attribute. Set its value to the path of an image. Most image formats work for poster images, although Internet Explorer sometimes struggles with SVG files.

```
<video src="path_to_file.video" poster="path_to_poster_image.jpg">
</video>
```

Be careful with the size of your poster image. Ideally, it should be the same dimensions as your video. Initially the video player dimensions will match the poster image dimensions. Then browser will resize the player once it has determined the intrinsic height and width of the video. Setting an explicit height and width for your video player prevents this resizing.

Controlling Video Dimensions

Whether you use a poster image or not, explicitly setting the height and width of your player prevents the browser from having to redraw the page once the video loads. One way to do this, of course, is to include the `height` and `width` attributes with the `video` tag. Both attributes accept percentages, which are useful when building a responsive or fluid layout.

```
<video src="path_to_file.video" poster="path_to_poster_image.jpg"
➥width="100%" height="100%"></video>
```

You may also set the height and width of the video player using CSS.

```
video {
  width: 960px;
  height: 540px;
}
```

In this example, we've used pixels. For responsive layouts, you may prefer to use viewport percentage units: `vh` and `vw`. Any valid CSS length unit outlined in the CSS Values and Units Module Level 3[3] is acceptable. Using CSS to set the video player's dimensions will override any `width` and `height` attributes applied to the element itself.

Bandwidth Use and Playback Responsiveness

Most browsers download a portion of an audio or video file as the page loads. Typically this snippet of media contains the file's metadata, such as duration and

[3] http://www.w3.org/TR/css3-values/

dimensions, and a few seconds of the playable data. When the user initiates playback, the browser makes a second request for the rest of the file.

Each of these requests places an additional demand on the server, whether or not your visitor interacts with the media file. You can change this behavior with the `preload` attribute. Set it to one of three possible values:

- `metadata` tells the browser that it's okay to download a portion of the file.
- `auto` tells the browser that it's okay to download as much of the video as it wants.
- `none` tells the browser not to download anything until the user requests it.

Using `preload="auto"` provides the fastest playback for the user. Browsers will download as much of the resource as it needs to provide consistent playback. For shorter clips, that could be the entire file.

With `preload="none"`, users could experience a significant lag between pressing the play control and media playback. However, this option will lead most browsers to download the least amount of data.

Set Explicit Video Height and Width When Using `preload="none"`

With `preload="none"`, you may want to set an explicit width and height either in the video element itself, or using CSS. Otherwise, you may trigger a page reflow when the video loads and begins to play.

Using `preload="metadata"` is a bit of a compromise between `auto` and `none`. In most browsers, there won't be a lag between the user requesting playback and the action, as you often get with `preload="none"`. But because the browser pre-loads a smaller portion of the media file, playback may not instantaneous as with `preload="auto"`.

Cross-browser Audio and Video

This almost sounds to good to be true, doesn't it? Native audio and video *without* a plug-in! Not so fast. There is one thing is holding us back: file format support. Browser vendors disagree about whether there should be a default multimedia codec, and if so, which one.

Apple and Microsoft have decided to support H.264/MPEG-4 video and MPEG-3 audio in their browsers (Safari and Internet Explorer, respectively). H.264 is a proprietary, high-definition format for displaying video, usually within an MPEG-4 container. MPEG-3 is an audio compression format. Because these formats are proprietary, browser developers must pay licensing fees if they'd like to add support for these formats to their software.

Mozilla and Opera are opposed H.264 and MPEG-3 largely because of those royalty fees. Instead their browsers (Firefox and Opera) support open source codecs such as Ogg Theora and WebM. Firefox *does* support H.264 and MPEG-3 for mobile devices, but not for desktop and laptops. Internet Explorer supports also other codecs if the user has installed them. Google Chrome, to its credit, supports all of the above.

The Great Codec Divide means that cross-browser video requires one of two approaches:

1. Encode only an H.264 version of the video and use a Flash video container as a fallback to serve the video to browsers that don't support H.264 natively.

2. Encode the video in multiple formats, and let the browser choose which to play.

The first option is best if you need to support older browsers. Internet Explorer 8, for example, lacks support for audio and video. JavaScript libraries such as Video.js[4] and audio.js[5] use this strategy.

The second option is better if you do not need to support older browsers. It will work for desktop and mobile device browsers. We'll use this approach here.

Transcoding Software

To transcode videos from one format to another, try FFMpeg[6], a command-line tool, or Miro Video Converter[7]. Both are free and open source, with Mac OS X, Windows, and Linux builds available.

[4] http://www.videojs.com/
[5] http://kolber.github.io/audiojs/
[6] http://www.ffmpeg.org/
[7] http://www.mirovideoconverter.com/

Using Multiple Video or Audio Files

To offer multiple file formats, we need to use the `source` element: one `<source>` tag for each file format. Attributes such as `autoplay`, `loop`, and `controls` should still be a part of the `<video>` or `<audio>` tag. But our `src` attribute must move to our `<source>` tags.

```
<audio controls>
  <source src="path_to_mpeg3_file.mp3">
  <source src="path_to_ogg_file.ogg">
</audio>
```

We can optionally add a `type` attribute to each source tag. At the very least, `type` should contain a valid MIME type. But it may also include a codec as shown below.

```
<audio controls>
  <source src="mpeg3_file.mp3" type="audio/mpeg">
  <source src="ogg_vorbis_file.ogg" type="audio/ogg; codecs=vorbis">
  <source src="ogg_flac_file.oga" type="audio/ogg; codecs=flac">
</audio>
```

`preload=none` on Safari

Using `preload=none` with multiple sources may prevent Safari from downloading the correct file. Safari 6.0.5 will ignore any file besides the first one when `preload=none`. Even if the user presses play, Safari will not load another video source. Avoid this by listing a Safari-compatible source first. Otherwise set the value of `preload` to `metadata` or `auto`.

Each browser will download the first available file that it's capable of playing.