HTML5
Up and Running
Mark Pilgrim
HTML5: Up and Running

If you don’t know about the new features available in HTML5, now’s the time to find out. This book provides practical information about how and why the latest version of this markup language will significantly change the way you develop for the Web.

HTML5 is still evolving, yet browsers such as Safari, Mozilla, Opera, and Chrome already support many of its features—and mobile browsers are even farther ahead. HTML5: Up and Running carefully guides you through the important changes in this version with lots of hands-on examples, including markup, graphics, and screenshots. You’ll learn how to use HTML5 markup to add video, offline capabilities, and more—and you’ll be able to put that functionality to work right away.

- Learn new semantic elements such as <header>, <footer>, and <section>
- Meet Canvas, a 2D drawing surface you can program with JavaScript
- Embed video in your web pages without third-party plugins
- Use Geolocation to let web application visitors share their physical location
- Take advantage of local storage capacity that goes way beyond cookies
- Build offline web applications that work after network access is disconnected
- Learn about several new input types for web forms
- Create your own custom vocabularies in HTML5 with microdata

“...this is the definitive book on HTML5 for beginners and advanced gurus alike.”

—Brad Neuberg
Software Engineer, Google

Mark Pilgrim works as a developer advocate for Google, specializing in open source and open standards. He’s the author of several technical books, including Greasemonkey Hacks (O’Reilly), Dive Into Python, and Dive Into Python 3 (both Apress).

Previous programming experience is recommended.

US $29.99  CAN $34.99
ISBN: 978-0-596-80602-6
HTML5: Up and Running
<table>
<thead>
<tr>
<th>Section</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>Preface</td>
<td>ix</td>
</tr>
<tr>
<td>1. How Did We Get Here?</td>
<td>1</td>
</tr>
<tr>
<td>Diving In</td>
<td>1</td>
</tr>
<tr>
<td>MIME Types</td>
<td>1</td>
</tr>
<tr>
<td>A Long Digression into How Standards Are Made</td>
<td>2</td>
</tr>
<tr>
<td>An Unbroken Line</td>
<td>7</td>
</tr>
<tr>
<td>A Timeline of HTML Development from 1997 to 2004</td>
<td>9</td>
</tr>
<tr>
<td>Everything You Know About XHTML Is Wrong</td>
<td>10</td>
</tr>
<tr>
<td>A Competing Vision</td>
<td>11</td>
</tr>
<tr>
<td>What Working Group?</td>
<td>12</td>
</tr>
<tr>
<td>Back to the W3C</td>
<td>13</td>
</tr>
<tr>
<td>Postscript</td>
<td>14</td>
</tr>
<tr>
<td>Further Reading</td>
<td>14</td>
</tr>
<tr>
<td>2. Detecting HTML5 Features</td>
<td>15</td>
</tr>
<tr>
<td>Diving In</td>
<td>15</td>
</tr>
<tr>
<td>Detection Techniques</td>
<td>15</td>
</tr>
<tr>
<td>Modernizr: An HTML5 Detection Library</td>
<td>16</td>
</tr>
<tr>
<td>Canvas</td>
<td>16</td>
</tr>
<tr>
<td>Canvas Text</td>
<td>17</td>
</tr>
<tr>
<td>Video</td>
<td>18</td>
</tr>
<tr>
<td>Video Formats</td>
<td>19</td>
</tr>
<tr>
<td>Local Storage</td>
<td>21</td>
</tr>
<tr>
<td>Web Workers</td>
<td>23</td>
</tr>
<tr>
<td>Offline Web Applications</td>
<td>23</td>
</tr>
<tr>
<td>Geolocation</td>
<td>24</td>
</tr>
<tr>
<td>Input Types</td>
<td>25</td>
</tr>
<tr>
<td>Placeholder Text</td>
<td>27</td>
</tr>
<tr>
<td>Form Autofocus</td>
<td>27</td>
</tr>
<tr>
<td>Microdata</td>
<td>28</td>
</tr>
</tbody>
</table>
### 3. What Does It All Mean? .......................... 31
- Diving In 31
- The Doctype 31
- The Root Element 33
- The `<head>` Element 34
  - Character Encoding 35
  - Friends and (Link) Relations 36
- New Semantic Elements in HTML5 41
- A Long Digression into How Browsers Handle Unknown Elements 42
- Headers 45
- Articles 47
- Dates and Times 49
- Navigation 51
- Footers 52
- Further Reading 54

### 4. Let’s Call It a Draw(ing Surface) .................. 57
- Diving In 57
- Simple Shapes 58
- Canvas Coordinates 60
- Paths 61
- Text 63
- Gradients 67
- Images 70
- What About IE? 73
- A Complete Example 75
- Further Reading 79

### 5. Video on the Web ........................................ 81
- Diving In 81
- Video Containers 81
- Video Codecs 83
  - H.264 84
  - Theora 84
  - VP8 85
- Audio Codecs 85
  - MPEG-1 Audio Layer 3 86
  - Advanced Audio Coding 87
  - Vorbis 87
- What Works on the Web 88
- Licensing Issues with H.264 Video 90
<table>
<thead>
<tr>
<th>6. You Are Here (And So Is Everybody Else)</th>
<th>117</th>
</tr>
</thead>
<tbody>
<tr>
<td>Diving In</td>
<td>117</td>
</tr>
<tr>
<td>The Geolocation API</td>
<td>117</td>
</tr>
<tr>
<td>Show Me the Code</td>
<td>118</td>
</tr>
<tr>
<td>Handling Errors</td>
<td>120</td>
</tr>
<tr>
<td>Choices! I Demand Choices!</td>
<td>121</td>
</tr>
<tr>
<td>What About IE?</td>
<td>123</td>
</tr>
<tr>
<td>geo.js to the Rescue</td>
<td>123</td>
</tr>
<tr>
<td>A Complete Example</td>
<td>125</td>
</tr>
<tr>
<td>Further Reading</td>
<td>126</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>7. The Past, Present, and Future of Local Storage for Web Applications</th>
<th>127</th>
</tr>
</thead>
<tbody>
<tr>
<td>Diving In</td>
<td>127</td>
</tr>
<tr>
<td>A Brief History of Local Storage Hacks Before HTML5</td>
<td>128</td>
</tr>
<tr>
<td>Introducing HTML5 Storage</td>
<td>129</td>
</tr>
<tr>
<td>Using HTML5 Storage</td>
<td>130</td>
</tr>
<tr>
<td>Tracking Changes to the HTML5 Storage Area</td>
<td>131</td>
</tr>
<tr>
<td>Limitations in Current Browsers</td>
<td>132</td>
</tr>
<tr>
<td>HTML5 Storage in Action</td>
<td>132</td>
</tr>
<tr>
<td>Beyond Named Key/Value Pairs: Competing Visions</td>
<td>134</td>
</tr>
<tr>
<td>Further Reading</td>
<td>135</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>8. Let’s Take This Offline</th>
<th>137</th>
</tr>
</thead>
<tbody>
<tr>
<td>Diving In</td>
<td>137</td>
</tr>
<tr>
<td>The Cache Manifest</td>
<td>138</td>
</tr>
<tr>
<td>Network Sections</td>
<td>139</td>
</tr>
<tr>
<td>Fallback Sections</td>
<td>140</td>
</tr>
<tr>
<td>The Flow of Events</td>
<td>141</td>
</tr>
<tr>
<td>The Fine Art of Debugging, a.k.a. “Kill Me! Kill Me Now!”</td>
<td>142</td>
</tr>
<tr>
<td>Let’s Build One!</td>
<td>145</td>
</tr>
<tr>
<td>Further Reading</td>
<td>146</td>
</tr>
</tbody>
</table>
9. A Form of Madness ........................................... 147
    Diving In 147
    Placeholder Text 147
    Autofocus Fields 148
    Email Addresses 150
    Web Addresses 151
    Numbers As Spinboxes 153
    Numbers As Sliders 155
    Date Pickers 156
    Search Boxes 158
    Color Pickers 160
    And One More Thing... 160
    Further Reading 161

10. “Distributed,” “Extensibility,” and Other Fancy Words ................. 163
    Diving In 163
    What Is Microdata? 164
    The Microdata Data Model 165
    Marking Up People 168
        Introducing Google Rich Snippets 174
    Marking Up Organizations 176
    Marking Up Events 180
        The Return of Google Rich Snippets 184
    Marking Up Reviews 185
    Further Reading 189

Appendix: The All-in-One Almost-Alphabetical Guide to Detecting Everything 191

Index ................................................................. 201
Diving In

What is HTML5? HTML5 is the next generation of HTML, superseding HTML 4.01, XHTML 1.0, and XHTML 1.1. HTML5 provides new features that are necessary for modern web applications. It also standardizes many features of the web platform that web developers have been using for years, but that have never been vetted or documented by a standards committee. (Would it surprise you to learn that the `window` object has never been formally documented? In addition to the new features, HTML5 is the first attempt to formally document many of the “de facto” standards that web browsers have supported for years.)

Like its predecessors, HTML5 is designed to be cross-platform. You don’t need to be running Windows or Mac OS X or Linux or Multics or any particular operating system in order to take advantage of HTML5. The only thing you do need is a modern web browser. There are modern web browsers available for free for all major operating systems. You may already have a web browser that supports certain HTML5 features. The latest versions of Apple Safari, Google Chrome, Mozilla Firefox, and Opera all support many HTML5 features. (You’ll find more detailed browser compatibility tables throughout this book.) The mobile web browsers that come preinstalled on iPhones, iPads, and Android phones all have excellent support for HTML5. Even Microsoft has announced that the upcoming Version 9 of Internet Explorer will support some HTML5 functionality.

This book will focus on eight topics:

- New semantic elements like `<header>`, `<footer>`, and `<section>` (Chapter 3)
- Canvas, a two-dimensional drawing surface that you can program with JavaScript (Chapter 4)
- Video that you can embed on your web pages without resorting to third-party plug-ins (Chapter 5)
- Geolocation, whereby visitors can choose to share their physical locations with your web application (Chapter 6)
- Persistent local storage without resorting to third-party plug-ins (Chapter 7)
• Offline web applications that work even after network access is interrupted (Chapter 8)
• Improvements to HTML web forms (Chapter 9)
• Microdata that lets you create your own vocabularies beyond HTML5 and extend your web pages with custom semantics (Chapter 10)

HTML5 is designed, as much as possible, to be backward compatible with existing web browsers. New features build on existing features and allow you to provide fallback content for older browsers. If you need even greater control, you can detect support for individual HTML5 features (Chapter 2) using a few lines of JavaScript. Don’t rely on fragile browser sniffing to decide which browsers support HTML5! Instead, test for the features you need using HTML5 itself.

**Conventions Used in This Book**

The following typographical conventions are used in this book:

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

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

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

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

This icon signifies a tip, suggestion, or general note.

This icon indicates a warning or caution.
Using Code Examples

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

We appreciate, but do not require, attribution. An attribution usually includes the title, author, publisher, and ISBN. For example: “HTML5: Up and Running by Mark Pilgrim. Copyright 2010 O’Reilly Media, Inc., 978-0-596-80602-6.”

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

A Note on the Editions of This Book

This book is derived from its HTML5 source, found at http://diveintohtml5.org/ and maintained by the author. The ebook and Safari Books Online editions include all the original hyperlinking, while the print edition includes only a subset of the hyperlinks, set as URLs in parentheses. If you are reading the print edition, please refer to one of the other editions—or the original source—for a richer linking experience. Because the author maintains http://diveintohtml5.org/ in HTML5, the site includes live examples of the code described in this book, many of which had to be modified for publication. Please visit http://diveintohtml5.org/ to see these examples, but be aware that their rendering may vary across browsers.

Safari® Books Online

Safari Books Online is an on-demand digital library that lets you easily search over 7,500 technology and creative reference books and videos to find the answers you need quickly.

With a subscription, you can read any page and watch any video from our library online. Read books on your cell phone and mobile devices. Access new titles before they are available for print, and get exclusive access to manuscripts in development and post feedback for the authors. Copy and paste code samples, organize your favorites, download chapters, bookmark key sections, create notes, print out pages, and benefit from tons of other time-saving features.
O’Reilly Media has uploaded this book to the Safari Books Online service. To have full digital access to this book and others on similar topics from O’Reilly and other publishers, sign up for free at http://my.safaribooksonline.com.

How to Contact Us

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

O’Reilly Media, Inc.
1005 Gravenstein Highway North
Sebastopol, CA 95472
800-998-9938 (in the United States or Canada)
707-829-0515 (international or local)
707-829-0104 (fax)

We have a web page for this book, where we list errata, examples, and any additional information. You can access this page at:

http://oreilly.com/catalog/9780596806026/

To comment or ask technical questions about this book, send email to:

bookquestions@oreilly.com

For more information about our books, conferences, Resource Centers, and the O’Reilly Network, see our website at:

http://www.oreilly.com
CHAPTER 1
How Did We Get Here?

Diving In
Recently, I stumbled across a quote from a Mozilla developer about the tension inherent in creating standards:

Implementations and specifications have to do a delicate dance together. You don’t want implementations to happen before the specification is finished, because people start depending on the details of implementations and that constrains the specification. However, you also don’t want the specification to be finished before there are implementations and author experience with those implementations, because you need the feedback. There is unavoidable tension here, but we just have to muddle on through.

Keep this quote in the back of your mind, and let me explain how HTML5 came to be.

MIME Types
This book is about HTML5, not previous versions of HTML, and not any version of XHTML. But to understand the history of HTML5 and the motivations behind it, you need to understand a few technical details first. Specifically, MIME types.

Every time your web browser requests a page, the web server sends a number of headers before it sends the actual page markup. These headers are normally invisible, although there are a number of web development tools that will make them visible if you’re interested. The headers are important, because they tell your browser how to interpret the page markup that follows. The most important header is called Content-Type, and it looks like this:

Content-Type: text/html

text/html is called the “content type” or “MIME type” of the page. This header is the only thing that determines what a particular resource truly is, and therefore how it should be rendered. Images have their own MIME types (image/jpeg for JPEG images, image/png for PNG images, and so on). JavaScript files have their own MIME type. CSS
stylesheets have their own MIME type. Everything has its own MIME type. The Web runs on MIME types.

Of course, reality is more complicated than that. Very early web servers (I’m talking web servers from 1993) didn’t send the Content-Type header, because it didn’t exist yet. (It wasn’t invented until 1994.) For compatibility reasons that date all the way back to 1993, some popular web browsers will ignore the Content-Type header under certain circumstances. (This is called “content sniffing.”) But as a general rule of thumb, everything you’ve ever looked at on the Web—HTML pages, images, scripts, videos, PDFs, anything with a URL—has been served to you with a specific MIME type in the Content-Type header.

Tuck that under your hat. We’ll come back to it.

A Long Digression into How Standards Are Made

Why do we have an <img> element? I don’t suppose that’s a question you ask yourself very often. Obviously someone must have created it. These things don’t just appear out of nowhere. Every element, every attribute, every feature of HTML that you’ve ever used—someone created them, decided how they should work, and wrote it all down. These people are not gods, nor are they flawless. They’re just people. Smart people, to be sure. But just people.

One of the great things about standards that are developed “out in the open” is that you can go back in time and answer these kinds of questions. Discussions occur on mailing lists, which are usually archived and publicly searchable. So, I decided to do a bit of “email archaeology” to try to answer the <img> element question. I had to go back to before there was an organization called the World Wide Web Consortium (W3C). I went back to the earliest days of the Web, when you could count the number of web servers on the fingers of both hands, and maybe a couple of toes.

On February 25, 1993, Marc Andreessen wrote:*  

I’d like to propose a new, optional HTML tag:  

IMG  

Required argument is SRC="url".  

This names a bitmap or pixmap file for the browser to attempt to pull over the network and interpret as an image, to be embedded in the text at the point of the tag’s occurrence.  

An example is:  

<IMG SRC="file://foobar.com/foo/bar/blargh.xbm">  

(There is no closing tag; this is just a standalone tag.)

This tag can be embedded in an anchor like anything else; when that happens, it becomes an icon that’s sensitive to activation just like a regular text anchor.

Browsers should be afforded flexibility as to which image formats they support. Xbm and Xpm are good ones to support, for example. If a browser cannot interpret a given format, it can do whatever it wants instead (X Mosaic will pop up a default bitmap as a placeholder).

This is required functionality for X Mosaic; we have this working, and we’ll at least be using it internally. I’m certainly open to suggestions as to how this should be handled within HTML; if you have a better idea than what I’m presenting now, please let me know. I know this is hazy with regard to image format, but I don’t see an alternative than to just say “let the browser do what it can” and wait for the perfect solution to come along (MIME, someday, maybe).

This quote requires some explanation. Xbm and Xpm were popular graphics formats on Unix systems.

“Mosaic” was one of the earliest web browsers. (“X Mosaic” was the version that ran on Unix systems.) When he wrote this message in early 1993, Marc had not yet founded the company that made him famous, Mosaic Communications Corporation, nor had he started work on that company’s flagship product, “Mosaic Netscape.” (You may know them better by their later names, “Netscape Corporation” and “Netscape Navigator.”)

“MIME, someday, maybe” is a reference to content negotiation, a feature of HTTP where a client (like a web browser) tells the server (like a web server) what types of resources it supports (like image/jpeg) so the server can return something in the client’s preferred format. “The Original HTTP as defined in 1991” (the only version that was implemented in February 1993) did not have a way for clients to tell servers what kinds of images they supported, thus the design dilemma that Marc faced.

A few hours later, Tony Johnson replied:

I have something very similar in Midas 2.0 (in use here at SLAC, and due for public release any week now), except that all the names are different, and it has an extra argument NAME=“name”. It has almost exactly the same functionality as your proposed IMG tag. e.g.,

<ICON name="NoEntry" href="http://note/foo/bar/NoEntry.xbm">

The idea of the name parameter was to allow the browser to have a set of “built in” images. If the name matches a “built in” image it would use that instead of having to go out and fetch the image. The name could also act as a hint for “line mode” browsers as to what kind of a symbol to put in place of the image.

I don’t much care about the parameter or tag names, but it would be sensible if we used the same things. I don’t much care for abbreviations, i.e., why not IMAGE= and SOURCE=. I somewhat prefer ICON since it implies that the IMAGE should be smallish, but maybe ICON is an overloaded word?

Midas was another early web browser, a contemporary of X Mosaic. It was cross-platform; it ran on both Unix and VMS. “SLAC” refers to the Stanford Linear Accelerator Center, now the SLAC National Accelerator Laboratory, which hosted the
first web server in the United States (in fact, the first web server outside Europe). When Tony wrote this message, SLAC was an old-timer on the WWW, having hosted five pages on its web server for a whopping 441 days.

Tony continued:

While we are on the subject of new tags, I have another, somewhat similar tag, which I would like to support in Midas 2.0. In principle it is:

\[
\text{<INCLUDE HREF="...">}
\]

The intention here would be that the second document is to be included into the first document at the place where the tag occurred. In principle the referenced document could be anything, but the main purpose was to allow images (in this case arbitrary sized) to be embedded into documents. Again the intention would be that when HTTP2 comes along the format of the included document would be up for separate negotiation.

“HTTP2” is a reference to Basic HTTP as defined in 1992. At this point, in early 1993, it was still largely unimplemented. The draft known as “HTTP2” evolved and was eventually standardized as “HTTP 1.0”. HTTP 1.0 did include request headers for content negotiation, a.k.a. “MIME, someday, maybe.”

Tony went on:

An alternative I was considering was:

\[
\text{<A HREF="..." INCLUDE>See photo</A>}
\]

I don’t much like adding more functionality to the \text{<A>} tag, but the idea here is to maintain compatibility with browsers that can not honour the INCLUDE parameter. The intention is that browsers which do understand INCLUDE, replace the anchor text (in this case “See photo”) with the included document (picture), while older or dumber browsers ignore the INCLUDE tag completely.

This proposal was never implemented, although the idea of providing text if an image is missing is an important accessibility technique that was missing from Marc’s initial \text{<IMG>} proposal. Many years later, this feature was bolted on as the \text{<img alt>} attribute, which Netscape promptly broke by erroneously treating it as a tooltip.

A few hours after Tony posted his message, Tim Berners-Lee responded:

I had imagined that figures would be represented as

\[
\text{<a name=fig1 href="fghjkdfghj" REL="EMBED, PRESENT">Figure</a>}
\]

where the relationship values mean

<table>
<thead>
<tr>
<th>REL</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>EMBED</td>
<td>Embed this here when presenting it</td>
</tr>
<tr>
<td>PRESENT</td>
<td>Present this whenever the source document is presented</td>
</tr>
</tbody>
</table>

Note that you can have various combinations of these, and if the browser doesn’t support either one, it doesn’t break.

[I] see that using this as a method for selectable icons means nesting anchors. Hmmm. But I hadn’t wanted a special tag.
This proposal was never implemented, but the `rel` attribute is still around (see “Friends and (Link) Relations” on page 36).

Jim Davis added:

It would be nice if there was a way to specify the content type, e.g.

```html
<img href="http://nsa.gov/pub/sounds/gorby.au" content-type=audio/basic>
```

But I am completely willing to live with the requirement that I specify the content type by file extension.

This proposal was never implemented, but Netscape did later add support for arbitrary embedding of media objects with the `<embed>` element.

Jay C. Weber asked:

While images are at the top of my list of desired medium types in a WWW browser, I don’t think we should add idiosyncratic hooks for media one at a time. Whatever happened to the enthusiasm for using the MIME typing mechanism?

Marc Andreessen replied:

This isn’t a substitute for the upcoming use of MIME as a standard document mechanism; this provides a necessary and simple implementation of functionality that’s needed independently from MIME.

Jay C. Weber responded:

Let’s temporarily forget about MIME, if it clouds the issue. My objection was to the discussion of “how are we going to support embedded images” rather than “how are we going to support embedded objections in various media.”

Otherwise, next week someone is going to suggest “let’s put in a new tag `<aud src="file://foobar.com/foo/bar/blargh.snd">`” for audio.

There shouldn’t be much cost in going with something that generalizes.

With the benefit of hindsight, it appears that Jay’s concerns were well founded. It took a little more than a week, but HTML5 did finally add new `<video>` and `<audio>` elements.

Responding to Jay’s original message, Dave Raggett said:

True indeed! I want to consider a whole range of possible image/line art types, along with the possibility of format negotiation. Tim’s note on supporting clickable areas within images is also important.

Later in 1993, Dave proposed HTML+ as an evolution of the HTML standard. The proposal was never implemented, and it was superseded by HTML 2.0. HTML 2.0 was a “retro-spec,” which means it formalized features already in common use: “This specification brings together, clarifies, and formalizes a set of features that roughly corresponds to the capabilities of HTML in common use prior to June 1994.”
Dave later wrote **HTML 3.0**, based on his earlier HTML+ draft. Outside of the W3C’s own reference implementation, **Arena**, HTML 3.0 was never implemented. It was superseded by **HTML 3.2**, which was also a “retro-spec”: “**HTML 3.2 adds widely deployed features** such as tables, applets and text flow around images, while providing full backward compatibility with the existing standard HTML 2.0.”

Dave later coauthored **HTML 4.0**, developed **HTML Tidy**, and went on to help with XHTML, XForms, MathML, and other modern W3C specifications.

Getting back to 1993, Marc replied to Dave:

> Actually, maybe we should think about a general-purpose procedural graphics language within which we can embed arbitrary hyperlinks attached to icons, images, or text, or anything. Has anyone else seen Intermedia’s capabilities with regard to this?

**Intermedia** was a hypertext project from Brown University. It was developed from 1985 to 1991 and ran on A/UX, a Unix-like operating system for early Macintosh computers.

The idea of a “general-purpose procedural graphics language” did eventually catch on. Modern browsers support both **SVG** (declarative markup with embedded scripting) and `<canvas>` (a procedural direct-mode graphics API), although the latter started as a proprietary extension before being “retro-specced” by the **WHAT Working Group**.

Bill Janssen replied:

> Other systems to look at which have this (fairly valuable) notion are Andrew and Slate. Andrew is built with _insets_, each of which has some interesting type, such as text, bitmap, drawing, animation, message, spreadsheet, etc. The notion of arbitrary recursive embedding is present, so that an inset of any kind can be embedded in any other kind which supports embedding. For example, an inset can be embedded at any point in the text of the text widget, or in any rectangular area in the drawing widget, or in any cell of the spreadsheet.

> “Andrew” is a reference to the **Andrew User Interface System**, although at that time it was simply known as the **Andrew Project**.

Meanwhile, Thomas Fine had a different idea:

> Here’s my opinion. The best way to do images in WWW is by using MIME. I’m sure postscript is already a supported subtype in MIME, and it deals very nicely with mixing text and graphics.

> But it isn’t clickable, you say? Yes, you’re right. I suspect there is already an answer to this in display postscript. Even if there isn’t the addition to standard postscript is trivial. Define an anchor command which specifies the URL and uses the current path as a closed region for the button. Since postscript deals so well with paths, this makes arbitrary button shapes trivial.

**Display PostScript** was an onscreen rendering technology codeveloped by Adobe and NeXT.

This proposal was never implemented, but the idea that the best way to fix HTML is to replace it with something else altogether still pops up from time to time.
On March 2, 1993, Tim Berners-Lee commented:

HTTP2 allows a document to contain any type which the user has said he can handle, not just registered MIME types. So one can experiment. Yes I think there is a case for postscript with hypertext. I don’t know whether display postscript has enough. I know Adobe are trying to establish their own postscript-based “PDF” which will have links, and be readable by their proprietary brand of viewers.

I thought that a generic overlaying language for anchors (HyTime based?) would allow the hypertext and the graphics/video standards to evolve separately, which would help both.

Let the IMG tag be INCLUDE and let it refer to an arbitrary document type. Or EMBED if INCLUDE sounds like a cpp include which people will expect to provide SGML source code to be parsed inline—not what was intended.

HyTime was an early, SGML-based hypertext document system. It loomed large in many early discussions of HTML, and later XML.

Tim’s proposal for an <INCLUDE> tag was never implemented, although you can see echoes of it in <object>, <embed>, and the <iframe> element.

Finally, on March 12, 1993, Marc Andreessen revisited the thread:

Back to the inlined image thread again—I’m getting close to releasing Mosaic v0.10, which will support inlineled GIF and XBM images/bitmaps, as mentioned previously. […] We’re not prepared to support INCLUDE/EMBED at this point. […] So we’re probably going to go with <IMG SRC="url"> (not ICON, since not all inlined images can be meaningfully called icons). For the time being, inlined images won’t be explicitly content-type’ed; down the road, we plan to support that (along with the general adaptation of MIME). Actually, the image reading routines we’re currently using figure out the image format on the fly, so the filename extension won’t even be significant.

An Unbroken Line

I am extraordinarily fascinated with all aspects of this almost 17-year-old conversation that led to the creation of an HTML element that has been used on virtually every web page ever published. Consider this:

- HTTP still exists. It successfully evolved from 0.9 into 1.0 and later 1.1, and still it evolves.

- HTML still exists. That rudimentary data format (it didn’t even support inline images!) successfully evolved into 2.0, 3.2, and 4.0. HTML is an unbroken line. A twisted, knotted, snarled line, to be sure—there were plenty of “dead branches” in the evolutionary tree, places where standards-minded people got ahead of themselves (and ahead of authors and implementors)—but still, here we are in 2010, and web pages from 1990 still render in modern browsers. I just loaded one up in the browser of my state-of-the-art Android mobile phone, and I didn’t even get prompted to “please wait while importing legacy format...”
• HTML has always been a conversation between browser makers, authors, standards wonks, and other people who just showed up and liked to talk about angle brackets. Most of the successful versions of HTML have been “retro-specs,” catching up to the world while simultaneously trying to nudge it in the right direction. Anyone who tells you that HTML should be kept “pure” (presumably by ignoring browser makers, or ignoring authors, or both) is simply misinformed. HTML has never been pure, and all attempts to purify it have been spectacular failures, matched only by the attempts to replace it.

• None of the browsers in use in 1993 still exist in any recognizable form. Netscape Navigator was abandoned in 1998 and rewritten from scratch to create the Mozilla Suite, which was then forked to create Firefox. Internet Explorer had its humble “beginnings” in “Microsoft Plus! for Windows 95,” where it was bundled with some desktop themes and a pinball game; but of course, that browser can be traced back further too.

• Some of the operating systems from 1993 still exist, but none of them are relevant to the modern Web. Most people today who “experience” the Web do so on a PC running Windows 2000 or later, a Mac running Mac OS X, a PC running some flavor of Linux, or a handheld device like an iPhone. In 1993, Windows was at Version 3.1 (and competing with OS/2), Macs were running System 7, and Linux was distributed via Usenet. (Want to have some fun? Find a graybeard and whisper “Trumpet Winsock” or “MacPPP.”)

• Some of the same people are still around and still involved in what we now simply call “web standards.” That’s after almost 20 years. And some were involved in predecessors of HTML, going back into the 1980s and before.

• Speaking of predecessors.... With the eventual popularity of HTML and the Web, it is easy to forget the contemporary formats and systems that informed their design. Before you read this chapter, had you ever heard of Andrew? Intermedia? HyTime? And HyTime was not some rinky-dink academic research project; it was an ISO standard approved for military use. It was Big Business. And you can read about it yourself at http://www.sgmlsource.com/history/hthist.htm.

But none of this answers the original question: why do we have an `<img>` element? Why not an `<icon>` element? Or an `<include>` element? Why not a hyperlink with an `include` attribute, or some combination of `rel` values? Why an `<img>` element? Quite simply, because Marc Andreessen shipped one, and shipping code wins.

That’s not to say that all shipping code wins; after all, Andrew and Intermedia and HyTime shipped code too. Code is necessary but not sufficient for success. And I certainly don’t mean to say that shipping code before a standard will produce the best solution. Marc’s `<img>` element didn’t mandate a common graphics format; it didn’t define how text flowed around it; it didn’t support text alternatives or fallback content for older browsers. And 17 years later, we’re still struggling with content sniffing, and it’s still a source of crazy security vulnerabilities. You can trace that through the Great
Browser Wars, all the way back to February 25, 1993, when Marc Andreessen off-handedly remarked, “MIME, someday, maybe,” and then shipped his code anyway.

A Timeline of HTML Development from 1997 to 2004

In December 1997, the World Wide Web Consortium (W3C) published HTML 4.0 and promptly shut down the HTML Working Group. Less than two months later, a separate W3C Working Group published XML 1.0. A mere three months after that, the W3C held a workshop called “Shaping the Future of HTML” to answer the question, “Has W3C given up on HTML?” This was the answer:

In discussions, it was agreed that further extending HTML 4.0 would be difficult, as would converting 4.0 to be an XML application. The proposed way to break free of these restrictions is to make a fresh start with the next generation of HTML based upon a suite of XML tag-sets.

The W3C rechartered the HTML Working Group to create this “suite of XML tag-sets.” The members’ first step, in December 1998, was to draft an interim specification that simply reformulated HTML in XML without adding any new elements or attributes. This specification later became known as “XHTML 1.0”. It defined a new MIME type for XHTML documents, application/xhtml+xml. However, to ease the migration of existing HTML 4 pages, it also included Appendix C, which “summarizes design guidelines for authors who wish their XHTML documents to render on existing HTML user agents.” Appendix C said you were allowed to author so-called “XHTML” pages but still serve them with the text/html MIME type.

The next target was web forms. In August 1999, the same HTML Working Group published a first draft of XHTML Extended Forms. Its members set the expectations in the very first sentences of this draft document:

After careful consideration, the HTML Working Group has decided that the goals for the next generation of forms are incompatible with preserving backward compatibility with browsers designed for earlier versions of HTML. It is our objective to provide a clean new forms model (“XHTML Extended Forms”) based on a set of well-defined requirements. The requirements described in this document are based on experience with a very broad spectrum of form applications.

A few months later, “XHTML Extended Forms” was renamed “XForms” and moved to its own Working Group. That group worked in parallel with the HTML Working Group and finally published the first edition of XForms 1.0 in October 2003.

Meanwhile, with the transition to XML complete, the members of the HTML Working Group set their sights on creating “the next generation of HTML.” In May 2001, they published the first edition of XHTML 1.1, which added only a few minor features on top of XHTML 1.0 but eliminated the “Appendix C” loophole. Starting with Version 1.1, all XHTML documents were to be served with a MIME type of application/xhtml+xml.
Everything You Know About XHTML Is Wrong

Why are MIME types important? Why do I keep coming back to them? Three words: draconian error handling. Browsers have always been “forgiving” with HTML. If you create an HTML page but forget to give it a `<title>`, browsers will display the page anyway, even though the `<title>` element has always been required in every version of HTML. Certain tags are not allowed within other tags, but if you create a page that puts them inside anyway, browsers will just deal with it (somehow) and move on without displaying an error message.

As you might expect, the fact that “broken” HTML markup still worked in web browsers led authors to create broken HTML pages. A lot of broken pages. By some estimates, over 99 percent of HTML pages on the Web today have at least one error in them. But because these errors don’t cause browsers to display visible error messages, nobody ever fixes them.

The W3C saw this as a fundamental problem with the Web, and set out to correct it. XML, published in 1997, broke from the tradition of forgiving clients and mandated that all programs that consumed XML must treat so-called “well-formedness” errors as fatal. This concept of failing on the first error became known as “draconian error handling,” after the Greek leader Draco, who instituted the death penalty for relatively minor infractions of his laws. When the W3C reformulated HTML as an XML vocabulary, the people in charge mandated that all documents served with the new `application/xhtml+xml` MIME type would be subject to draconian error handling. If there was even a single error in your XHTML page, web browsers would have no choice but to stop processing and display an error message to the end user.

This idea was not universally popular. With an estimated error rate of 99 percent on existing pages, the ever-present possibility of displaying errors to the end user, and the dearth of new features in XHTML 1.0 and 1.1 to justify the cost, web authors basically ignored `application/xhtml+xml`. But that doesn’t mean they ignored XHTML altogether. Oh, most definitely not. Appendix C of the XHTML 1.0 specification gave the web authors of the world a loophole: “Use something that looks kind of like XHTML syntax, but keep serving it with the `text/html` MIME type.” And that’s exactly what thousands of web developers did: they “upgraded” to XHTML syntax but kept serving it with a `text/html` MIME type.

Even today, while many web pages claim to be XHTML—they start with the XHTML doctype on the first line, use lowercase tag names, use quotes around attribute values, and add a trailing slash after empty elements like `<br />` and `<hr />`—only a tiny fraction of these pages are served with the `application/xhtml+xml` MIME type that would trigger XML’s draconian error handling. Any page served with a MIME type of `text/html`, regardless of its doctype, syntax, or coding style, will be parsed using a “forgiving” HTML parser, silently ignoring any markup errors and never alerting end users (or anyone else), even if the page is technically broken.
XHTML 1.0 included this loophole, but XHTML 1.1 closed it, and the never-finalized XHTML 2.0 continued the tradition of requiring draconian error handling. And that’s why there are billions of pages that claim to be XHTML 1.0, and only a handful that claim to be XHTML 1.1 (or XHTML 2.0). So, are you really using XHTML? Check your MIME type. (Actually, if you don’t know what MIME type you’re using, I can pretty much guarantee that you’re still using text/html.) Unless you’re serving your pages with a MIME type of application/xhtml+xml, your so-called “XHTML” is XML in name only.

**A Competing Vision**

In June 2004, the W3C held the Workshop on Web Applications and Compound Documents. Present at this workshop were representatives of several browser vendors, web development companies, and other W3C members. A group of interested parties, including the Mozilla Foundation and Opera Software, gave a presentation on their competing visions of the future of the Web: an evolution of the existing HTML 4 standard to include new features for modern web application developers:

The following seven principles represent what we believe to be the most critical requirements for this work:

*Backward compatibility, clear migration path*

Web application technologies should be based on technologies authors are familiar with, including HTML, CSS, DOM, and JavaScript.

Basic Web application features should be implementable using behaviors, scripting, and style sheets in IE6 today so that authors have a clear migration path. Any solution that cannot be used with the current high-market-share user agent without the need for binary plug-ins is highly unlikely to be successful.

*Well-defined error handling*

Error handling in Web applications must be defined to a level of detail where User Agents (UAs) do not have to invent their own error handling mechanisms or reverse engineer other User Agents’.

*Users should not be exposed to authoring errors*

Specifications must specify exact error recovery behaviour for each possible error scenario. Error handling should for the most part be defined in terms of graceful error recovery (as in CSS), rather than obvious and catastrophic failure (as in XML).

*Practical use*

Every feature that goes into the Web Applications specifications must be justified by a practical use case. The reverse is not necessarily true: every use case does not necessarily warrant a new feature.

Use cases should preferably be based on real sites where the authors previously used a poor solution to work around the limitation.

*Scripting is here to stay*

But should be avoided where more convenient declarative markup can be used. Scripting should be device and presentation neutral unless scoped in a device-specific way (e.g., unless included in XBL).
Device-specific profiling should be avoided
Authors should be able to depend on the same features being implemented in desktop and mobile versions of the same UA.

Open process
The Web has benefited from being developed in an open environment. Web Applications will be core to the Web, and its development should also take place in the open. Mailing lists, archives and draft specifications should continuously be visible to the public.

In a straw poll, the workshop participants were asked, “Should the W3C develop declarative extensions to HTML and CSS and imperative extensions to DOM, to address medium level Web Application requirements, as opposed to sophisticated, fully-fledged OS-level APIs?” The vote was 11 to 8 against. In their summary of the workshop, the W3C’s members wrote, “At present, W3C does not intend to put any resources into the third straw-poll topic: extensions to HTML and CSS for Web Applications, other than technologies being developed under the charter of current W3C Working Groups.”

Faced with this decision, the people who had proposed evolving HTML and HTML forms had only two choices: give up, or continue their work outside of the W3C. They chose the latter, registered the whatwg.org domain, and in June 2004, the WHAT Working Group was born.

What Working Group?
What the heck is the WHAT Working Group? I’ll let it explain for itself:

The Web Hypertext Applications Technology Working Group is a loose, unofficial, and open collaboration of Web browser manufacturers and interested parties. The group aims to develop specifications based on HTML and related technologies to ease the deployment of interoperable Web Applications, with the intention of submitting the results to a standards organisation. This submission would then form the basis of work on formally extending HTML in the standards track.

The creation of this forum follows from several months of work by private e-mail on specifications for such technologies. The main focus up to this point has been extending HTML4 Forms to support features requested by authors, without breaking backward compatibility with existing content. This group was created to ensure that future development of these specifications will be completely open, through a publicly-archived, open mailing list.

The key phrase here is “without breaking backward compatibility.” XHTML (minus the Appendix C loophole) is not backward compatible with HTML. It requires an entirely new MIME type, and it mandates draconian error handling for all content served with that MIME type. XForms is not backward compatible with HTML forms, because it can only be used in documents that are served with the new XHTML MIME type, which means that XForms also mandates draconian error handling. All roads lead to MIME.
Instead of scrapping over a decade’s worth of investment in HTML and making 99 percent of existing web pages unusable, the WHAT Working Group decided to take a different approach: documenting the “forgiving” error handling algorithms that browsers actually used. Web browsers have always been forgiving of HTML errors, but nobody had ever bothered to write down exactly how they did it. NCSA Mosaic had its own algorithms for dealing with broken pages, and Netscape tried to match them. Then Internet Explorer tried to match Netscape. Then Opera and Firefox tried to match Internet Explorer. Then Safari tried to match Firefox. And so on, right up to the present day. Along the way, developers burned thousands and thousands of hours trying to make their products compatible with those of their competitors.

If that sounds like an insane amount of work, that’s because it is. Or rather, it was. It took several years, but (modulo a few obscure edge cases) the WHAT Working Group successfully documented how to parse HTML in a way that is compatible with existing web content. Nowhere in the final algorithm is there a step that mandates that the HTML consumer should stop processing and display an error message to the end user.

While all that reverse-engineering was going on, the WHAT Working Group was quietly working on a few other things, too. One of them was a specification, initially dubbed Web Forms 2.0, that added new types of controls to HTML forms. (You’ll learn more about web forms in Chapter 9.) Another was a draft specification called “Web Applications 1.0” that included major new features, like a direct-mode drawing canvas (see Chapter 4) and native support for audio and video without plug-ins (see Chapter 5).

Back to the W3C

For several years, the W3C and the WHAT Working Group largely ignored each other. While the WHAT Working Group focused on web forms and new HTML features, the W3C HTML Working Group was busy with Version 2.0 of XHTML. But by October 2006, it was clear that the WHAT Working Group had picked up serious momentum, while XHTML 2 was still languishing in draft form, unimplemented by any major browser. In October 2006, Tim Berners-Lee, the founder of the W3C itself, announced that the W3C would work together with the WHAT Working Group to evolve HTML:

Some things are clearer with hindsight of several years. It is necessary to evolve HTML incrementally. The attempt to get the world to switch to XML, including quotes around attribute values and slashes in empty tags and namespaces all at once didn’t work. The large HTML-generating public did not move, largely because the browsers didn’t complain. Some large communities did shift and are enjoying the fruits of well-formed systems, but not all. It is important to maintain HTML incrementally, as well as continuing a transition to a well-formed world, and developing more power in that world.

The plan is to charter a completely new HTML group. Unlike the previous one, this one will be chartered to do incremental improvements to HTML, and also in parallel XHTML. It will have a different chair and staff contact. It will work on HTML and XHTML together. We have strong support for this group, from many people we have talked to, including browser makers.
There will also be work on forms. This is a complex area, as existing HTML forms and XForms are both form languages. HTML forms are ubiquitously deployed, and there are many implementations and users of XForms. Meanwhile, the Webforms submission has suggested sensible extensions to HTML forms. The plan is, informed by Webforms, to extend HTML forms.

One of the first things the newly rechartered W3C HTML Working Group decided was to rename “Web Applications 1.0” to “HTML5.” And here we are, diving into HTML5.

Postscript

In October 2009, the W3C shut down the XHTML 2 Working Group and issued this statement to explain the decision:

When W3C announced the HTML and XHTML 2 Working Groups in March 2007, we indicated that we would continue to monitor the market for XHTML 2. W3C recognizes the importance of a clear signal to the community about the future of HTML.

While we recognize the value of the XHTML 2 Working Group’s contributions over the years, after discussion with the participants, W3C management has decided to allow the Working Group’s charter to expire at the end of 2009 and not to renew it.

The ones that win are the ones that ship.

Further Reading

- “The History of the Web”, an old draft by Ian Hickson
- “HTML/History”, by Michael Smith, Henri Sivonen, and others
- “A Brief History of HTML”, by Scott Reynen
CHAPTER 2
Detecting HTML5 Features

Diving In
You may well ask, “How can I start using HTML5 if older browsers don’t support it?” But the question itself is misleading. HTML5 is not one big thing; it is a collection of individual features. So, you can’t detect “HTML5 support,” because that doesn’t make any sense. But you can detect support for individual features, like canvas, video, or geolocation.

Detection Techniques
When your browser renders a web page, it constructs a Document Object Model (DOM), a collection of objects that represent the HTML elements on the page. Every element—every <p>, every <div>, every <span>—is represented in the DOM by a different object. (There are also global objects, like window and document, that aren’t tied to specific elements.)

All DOM objects share a set of common properties, but some objects have more than others. In browsers that support HTML5 features, certain objects will have unique properties. A quick peek at the DOM will tell you which features are supported.

There are four basic techniques for detecting whether a browser supports a particular feature. From simplest to most complex:

1. Check if a certain property exists on a global object (such as window or navigator).
   For an example of testing for geolocation support, see “Geolocation” on page 24.

2. Create an element, then check if a certain property exists on that element.
   For an example of testing for canvas support, see “Canvas” on page 16.

3. Create an element, check if a certain method exists on that element, then call the method and check the value it returns.
For an example of testing which video formats are supported, see “Video Formats” on page 19.

4. Create an element, set a property to a certain value, then check if the property has retained its value.

For an example of testing which <input> types are supported, see “Input Types” on page 25.

**Modernizr: An HTML5 Detection Library**

Modernizr is an open source, MIT-licensed JavaScript library that detects support for many HTML5 and CSS3 features. At the time of writing, the latest version is 1.1. You should always use the latest version. To do so, include the following <script> element at the top of your page:

```html
<!DOCTYPE html>
<html>
<head>
  <meta charset="utf-8">
  <title>Dive into HTML5</title>
  <script src="modernizr.min.js"></script>
</head>
<body>
... 
</body>
</html>
```

Modernizr runs automatically. There is no `modernizr_init()` function to call. When it runs, it creates a global object called `Modernizr` that contains a set of Boolean properties for each feature it can detect. For example, if your browser supports the canvas API (see Chapter 4), the `Modernizr.canvas` property will be `true`. If your browser does not support the canvas API, the `Modernizr.canvas` property will be `false`:

```javascript
if (Modernizr.canvas) {
  // let's draw some shapes!
} else {
  // no native canvas support available :( 
}
```

**Canvas**

HTML5 defines the `<canvas>` element as “a resolution-dependent bitmap canvas which can be used for rendering graphs, game graphics, or other visual images on the fly.” A canvas is a rectangle in your page within which you can use JavaScript to draw anything you want. HTML5 defines a set of functions (“the canvas API”) for drawing shapes, defining paths, creating gradients, and applying transformations.

Checking for canvas API support uses detection technique #2 (see “Detection Techniques” on page 15). If your browser supports the canvas API, the DOM object it creates
to represent a `<canvas>` element will have a `getContext()` method (see “Simple Shapes” on page 58). If your browser doesn’t support the canvas API, the DOM object it creates for a `<canvas>` element will have only the set of common properties, not anything canvas-specific. You can check for canvas support using this function:

```javascript
function supports_canvas() {
    return !!document.createElement('canvas').getContext;
}
```

This function starts by creating a dummy `<canvas>` element:

```javascript
return !!document.createElement('canvas').getContext;
```

This element is never attached to your page, so no one will ever see it. It’s just floating in memory, going nowhere and doing nothing, like a canoe on a lazy river.

As soon as you create the dummy `<canvas>` element, you test for the presence of a `getContext()` method. This method will only exist if your browser supports the canvas API:

```javascript
return !!document.createElement('canvas').getContext;
```

Finally, you use the double-negative trick to force the result to a Boolean value (`true` or `false`):

```javascript
return !!document.createElement('canvas').getContext;
```

This function will detect support for most of the canvas API, including shapes (see “Simple Shapes” on page 58), paths (see “Paths” on page 61), gradients (see “Gradients” on page 67), and patterns. It will not detect the third-party `explorercanvas` library (see “What About IE?” on page 73) that implements the canvas API in Microsoft Internet Explorer.

Instead of writing this function yourself, you can use Modernizr (introduced in the preceding section) to detect support for the canvas API:

```javascript
if (Modernizr.canvas) {
    // let's draw some shapes!
} else {
    // no native canvas support available :(
}
```

There is a separate test for the canvas text API, which I will demonstrate next.

## Canvas Text

Even if your browser supports the canvas API, it might not support the canvas text API. The canvas API grew over time, and the text functions were added late in the game. Some browsers shipped with canvas support before the text API was complete.

Checking for canvas text API support again uses detection technique #2 (see “Detection Techniques” on page 15). If your browser supports the canvas API, the DOM object it creates to represent a `<canvas>` element will have the `getContext()` method (see “Simple Canvas Text | 17...
Shapes” on page 58). If your browser doesn’t support the canvas API, the DOM object it creates for a `<canvas>` element will have only the set of common properties, not anything canvas-specific. You can check for canvas text support using this function:

```javascript
function supports_canvas_text() {
    if (!supports_canvas()) { return false; }
    var dummy_canvas = document.createElement('canvas');
    var context = dummy_canvas.getContext('2d');
    return typeof context.fillText == 'function';
}
```

The function starts by checking for canvas support, using the `supports_canvas()` function introduced in the previous section:

```javascript
if (!supports_canvas()) { return false; }
```

If your browser doesn’t support the canvas API, it certainly won’t support the canvas text API!

Next, you create a dummy `<canvas>` element and get its drawing context. This is guaranteed to work, because the `supports_canvas()` function already checked that the `getContext()` method exists on all canvas objects:

```javascript
var dummy_canvas = document.createElement('canvas');
var context = dummy_canvas.getContext('2d');
```

Finally, you check whether the drawing context has a `fillText()` function. If it does, the canvas text API is available:

```javascript
return typeof context.fillText == 'function';
```

Instead of writing this function yourself, you can use Modernizr (see “Modernizr: An HTML5 Detection Library” on page 16) to detect support for the canvas text API:

```javascript
if (Modernizr.canvastext) {
    // let's draw some text!
} else {
    // no native canvas text support available :(
}
```

**Video**

HTML5 defines a new element called `<video>` for embedding video in your web pages. Embedding video used to be impossible without third-party plug-ins such as Apple QuickTime or Adobe Flash.
The `<video>` element is designed to be usable without any detection scripts. You can specify multiple video files, and browsers that support HTML5 video will choose one based on what video formats they support.*

Browsers that don’t support HTML5 video will ignore the `<video>` element completely, but you can use this to your advantage and tell them to play video through a third-party plug-in instead. Kroc Camen has designed a solution called Video for Everybody! that uses HTML5 video where available, but falls back to QuickTime or Flash in older browsers. This solution uses no JavaScript whatsoever, and it works in virtually every browser, including mobile browsers.

If you want to do more with video than plop it on your page and play it, you’ll need to use JavaScript. Checking for video support uses detection technique #2 (see “Detection Techniques” on page 15). If your browser supports HTML5 video, the DOM object it creates to represent a `<video>` element will have a `canPlayType()` method. If your browser doesn’t support HTML5 video, the DOM object it creates for a `<video>` element will have only the set of properties common to all elements. You can check for video support using this function:

```javascript
function supports_video() {
  return !!document.createElement('video').canPlayType;
}
```

Instead of writing this function yourself, you can use Modernizr (see “Modernizr: An HTML5 Detection Library” on page 16) to detect support for HTML5 video:

```javascript
if (Modernizr.video) {
  // let's play some video!
} else {
  // no native video support available :(
  // maybe check for QuickTime or Flash instead
}
```

There is a separate test for detecting which video formats your browser can play, which I will demonstrate next.

**Video Formats**

Video formats are like written languages. An English newspaper may convey the same information as a Spanish newspaper, but if you can only read English, only one of them will be useful to you! To play a video, your browser needs to understand the “language” in which the video was written.

* See “A gentle introduction to video encoding, part 1: container formats” and “part 2: lossy video codecs” to learn about different video formats.
The “language” of a video is called a “codec”—this is the algorithm used to encode the video into a stream of bits. There are dozens of codecs in use all over the world. Which one should you use? The unfortunate reality of HTML5 video is that browsers can’t agree on a single codec. However, they seem to have narrowed it down to two. One codec costs money (because of patent licensing), but it works in Safari and on the iPhone. (This one also works in Adobe Flash, if you use a solution like Video for Everybody!) The other codec is free and works in open source browsers like Chromium and Mozilla Firefox.

Checking for video format support uses detection technique #3 (see “Detection Techniques” on page 15). If your browser supports HTML5 video, the DOM object it creates to represent a <video> element will have a canPlayType() method. This method will tell you whether the browser supports a particular video format.

This function checks for the patent-encumbered format supported by Macs and iPhones:

```javascript
function supports_h264_baseline_video() {
  if (!supports_video()) { return false; }
  var v = document.createElement("video");
  return v.canPlayType('video/mp4; codecs="avc1.42E01E, mp4a.40.2"');
}
```

The function starts by checking for HTML5 video support, using the supports_video() function from the previous section:

```javascript
if (!supports_video()) { return false; }
```

If your browser doesn’t support HTML5 video, it certainly won’t support any video formats!

Next, the function creates a dummy <video> element (but doesn’t attach it to the page, so it won’t be visible) and calls the canPlayType() method. This method is guaranteed to be there, because the supports_video() function just checked for it:

```javascript
var v = document.createElement("video");
return v.canPlayType('video/mp4; codecs="avc1.42E01E, mp4a.40.2"');
```

A “video format” is really a combination of several different things. In technical terms, you’re asking the browser whether it can play H.264 Baseline video and AAC LC audio in an MPEG-4 container.†

The canPlayType() function doesn’t return true or false. In recognition of how complex video formats are, the function returns a string:

"probably"

If the browser is fairly confident it can play this format

† I’ll explain what all that means in Chapter 5. You might also be interested in reading “A gentle introduction to video encoding”.

---

20 | Chapter 2: Detecting HTML5 Features
"maybe"
  If the browser thinks it might be able to play this format
"" (an empty string)
  If the browser is certain it can’t play this format

This second function checks for the open video format supported by Mozilla Firefox and other open source browsers. The process is exactly the same; the only difference is the string you pass in to the `canPlayType()` function. In technical terms, you’re asking the browser whether it can play Theora video and Vorbis audio in an Ogg container:

```javascript
function supports_ogg_theora_video() {
  if (!supports_video()) { return false; }
  var v = document.createElement("video");
  return v.canPlayType('video/ogg; codecs="theora, vorbis"');
}
```

Finally, WebM is a newly open-sourced (and non-patent-encumbered) video codec that will be included in the next version of major browsers, including Chrome, Firefox, and Opera. You can use the same technique to detect support for open WebM video:

```javascript
function supports_webm_video() {
  if (!supports_video()) { return false; }
  var v = document.createElement("video");
  return v.canPlayType('video/webm; codecs="vp8, vorbis"');
}
```

Instead of writing this function yourself, you can use Modernizr to detect support for several different HTML5 video formats (note that Modernizr does not yet have support for detecting support for the open WebM video format):

```javascript
if (Modernizr.video) {
  // let’s play some video! but what kind?
  if (Modernizr.video.ogg) { // try Ogg Theora + Vorbis in an Ogg container
    }
  } else if (Modernizr.video.h264) { // try H.264 video + AAC audio in an MP4 container
    }
  }
```

## Local Storage

**HTML5 Storage** provides a way for websites to store information on your computer and retrieve it later. The concept is similar to cookies, but it’s designed for larger quantities of information. Cookies are limited in size, and your browser sends them back to the web server every time it requests a new page (which takes extra time and precious bandwidth). HTML5 Storage stays on your computer, and websites can access it with JavaScript after the page is loaded.
Ask Professor Markup

Q: Is local storage really part of HTML5? Why is it in a separate specification?
A: The short answer is yes, local storage is part of HTML5. The slightly longer answer is that local storage used to be part of the main HTML5 specification, but it was split out into a separate specification because some people in the HTML5 Working Group complained that HTML5 was too big. If that sounds like slicing a pie into more pieces to reduce the total number of calories...well, welcome to the wacky world of standards.

Checking for HTML5 Storage support uses detection technique #1 (see “Detection Techniques” on page 15). If your browser supports HTML5 Storage, there will be a localStorage property on the global window object. If your browser doesn’t support HTML5 Storage, the localStorage property will be undefined. You can check for local storage support using this function:

```javascript
function supports_local_storage() {
    return ('localStorage' in window) && window['localStorage'] !== null;
}
```

Instead of writing this function yourself, you can use Modernizr (see “Modernizr: An HTML5 Detection Library” on page 16) to detect support for HTML5 local storage:

```javascript
if (Modernizr.localstorage) {
    // window.localStorage is available!
} else {
    // no native support for local storage :(  
    // maybe try Gears or another third-party solution
}
```

Note that JavaScript is case-sensitive. The Modernizr attribute is called localStorage (all lowercase), but the DOM property is called window.localStorage (mixed case).

Ask Professor Markup

Q: How secure is my HTML5 Storage database? Can anyone read it?
A: Anyone who has physical access to your computer can probably look at (or even change) your HTML5 Storage database. Within your browser, any website can read and modify its own values, but sites can’t access values stored by other sites. This is called a same-origin restriction.
Web Workers

Web workers provide a standard way for browsers to run JavaScript in the background. With web workers, you can spawn multiple “threads” that all run at the same time, more or less. (Think of how your computer can run multiple applications at the same time, and you’re most of the way there.) These “background threads” can do complex mathematical calculations, make network requests, or access local storage while the main web page responds to the user scrolling, clicking, or typing.

Checking for web workers uses detection technique #1 (see “Detection Techniques” on page 15). If your browser supports the Web Worker API, there will be a `Worker` property on the global `window` object. If your browser doesn’t support the Web Worker API, the `Worker` property will be undefined. This function checks for web worker support:

```javascript
function supports_web_workers() {
    return !!window.Worker;
}
```

Instead of writing this function yourself, you can use Modernizr (see “Modernizr: An HTML5 Detection Library” on page 16) to detect support for web workers:

```javascript
if (Modernizr.webworkers) {
    // window.Worker is available!
} else {
    // no native support for web workers :(  
    // maybe try Gears or another third-party solution
}
```

Note that JavaScript is case-sensitive. The Modernizr attribute is called `webworkers` (all lowercase), but the DOM object is called `window.Worker` (with a capital “W” in “Worker”).

Offline Web Applications

Reading static web pages offline is easy: connect to the Internet, load a web page, disconnect from the Internet, drive to a secluded cabin, and read the web page at your leisure. (To save time, you may wish to skip the step about the cabin.) But what about using web applications like Gmail or Google Docs when you’re offline? Thanks to HTML5, anyone (not just Google!) can build a web application that works offline.

Offline web applications start out as online web applications. The first time you visit an offline-enabled website, the web server tells your browser which files it needs in order to work offline. These files can be anything—HTML, JavaScript, images, even videos (see “Video” on page 18). Once your browser downloads all the necessary files, you can revisit the website even if you’re not connected to the Internet. Your browser will notice that you’re offline and use the files it has already downloaded. When you get back online, any changes you’ve made can be uploaded to the remote web server.
Checking for offline support uses detection technique #1 (see “Detection Techniques” on page 15). If your browser supports offline web applications, there will be an applicationCache property on the global window object. If your browser doesn’t support offline web applications, the applicationCache property will be undefined. You can check for offline support with the following function:

```javascript
function supports_offline() {
    return !!window.applicationCache;
}
```

Instead of writing this function yourself, you can use Modernizr (see “Modernizr: An HTML5 Detection Library” on page 16) to detect support for offline web applications:

```javascript
if (Modernizr.applicationcache) {
    // window.applicationCache is available!
} else {
    // no native support for offline :( 
    // maybe try Gears or another third-party solution
}
```

Note that JavaScript is case-sensitive. The Modernizr attribute is called applicationcache (all lowercase), but the DOM object is called window.applicationCache (mixed case).

## Geolocation

Geolocation is the art of figuring out where you are in the world and (optionally) sharing that information with people you trust. There are many ways to figure out where you are—your IP address, your wireless network connection, which cell tower your phone is talking to, or dedicated GPS hardware that receives latitude and longitude information from satellites in the sky.

---

### Ask Professor Markup

**Q: Is geolocation part of HTML5? Why are you talking about it?**

**A: Geolocation support is being added to browsers right now, along with support for new HTML5 features. Strictly speaking, geolocation is being standardized by the Geolocation Working Group, which is separate from the HTML5 Working Group. But I’m going to talk about geolocation in this book anyway, because it’s part of the evolution of the Web that’s happening now.**

---

Checking for geolocation support uses detection technique #1 (see “Detection Techniques” on page 15). If your browser supports the geolocation API, there will be a geolocation property on the global navigator object. If your browser doesn’t support the geolocation API, the geolocation property will be undefined. Here’s how to check for geolocation support:
function supports_geolocation() {
  return !!navigator.geolocation;
}

Instead of writing this function yourself, you can use Modernizr (see “Modernizr: An HTML5 Detection Library” on page 16) to detect support for the geolocation API:

```javascript
if (Modernizr.geolocation) {
  // let's find out where you are!
} else {
  // no native geolocation support available :
  // maybe try Gears or another third-party solution
}
```

If your browser does not support the geolocation API natively, there is still hope. Gears is an open source browser plug-in from Google that works on Windows, Mac, Linux, Windows Mobile, and Android. It provides a number of features for older browsers that do not support all the fancy new stuff we’ve discussed in this chapter. One of the features that Gears provides is a geolocation API. It’s not the same as the `navigator.geolocation` API, but it serves the same purpose.

There are also device-specific geolocation APIs on several mobile phone platforms, including BlackBerry, Nokia, Palm, and OMTP BONDI.

Chapter 6 will go into excruciating detail about how to use all of these different APIs.

### Input Types

You know all about web forms, right? Make a `<form>`, add a few `<input type="text">` elements and maybe an `<input type="password">`, and finish it off with an `<input type="submit">` button.

You don’t know the half of it. HTML5 defines over a dozen new input types that you can use in your forms:

- `<input type="search">`
  - See [http://bit.ly/9mQt5C](http://bit.ly/9mQt5C) for search boxes
- `<input type="number">`
- `<input type="range">`
- `<input type="color">`
- `<input type="tel">`
- `<input type="url">`
Checking for HTML5 input types uses detection technique #4 (see “Detection Techniques” on page 15). First, you create a dummy `<input>` element in memory:

```javascript
var i = document.createElement("input");
```

The default input type for all `<input>` elements is "text". This will prove to be vitally important.

Next, set the `type` attribute on the dummy `<input>` element to the input type you want to detect:

```javascript
i.setAttribute("type", "color");
```

If your browser supports that particular input type, the `type` property will retain the value you set. If your browser doesn’t support that particular input type, it will ignore the value you set and the `type` property will still be "text":

```javascript
return i.type !== "text";
```

Instead of writing 13 separate functions yourself, you can use Modernizr (see “Modernizr: An HTML5 Detection Library” on page 16) to detect support for all the new input types defined in HTML5. Modernizr reuses a single `<input>` element to efficiently detect support for all 13 input types. Then it builds a hash called `Modernizr.inputtypes`, which contains 13 keys (the HTML5 `type` attributes) and 13 Boolean values (true if supported, false if not):

```javascript
if (!Modernizr.inputtypes.date) {
    // no native support for <input type="date"> :( 
    // maybe build one yourself with
    // Dojo
    // or jQueryUI
}
```
Placeholder Text

Besides new input types, HTML5 includes several small tweaks to existing forms. One improvement is the ability to set placeholder text in an input field. Placeholder text is displayed inside the input field as long as the field is empty and not focused. As soon as you click on (or tab to) the input field, the placeholder text disappears. “Placeholder Text” on page 147 has screenshots if you’re having trouble visualizing it.

Checking for placeholder support uses detection technique #2 (see “Detection Techniques” on page 15). If your browser supports placeholder text in input fields, the DOM object it creates to represent an <input> element will have a placeholder property (even if you don’t include a placeholder attribute in your HTML). If your browser doesn’t support placeholder text, the DOM object it creates for an <input> element will not have a placeholder property. Here’s how to check for placeholder support:

function supports_input_placeholder() {
    var i = document.createElement('input');
    return 'placeholder' in i;
}

Instead of writing this function yourself, you can use Modernizr (see “Modernizr: An HTML5 Detection Library” on page 16) to detect support for placeholder text:

    if (Modernizr.input.placeholder) {
        // your placeholder text should already be visible!
    } else {
        // no placeholder support :(
        // fall back to a scripted solution
    }

Form Autofocus

Many websites use JavaScript to focus the first input field of a web form automatically. For example, the home page of Google.com will autofocus the input box so you can type your search keywords without having to position the cursor in the search box. While this is convenient for most people, it can be annoying for power users or people with special needs. If you press the space bar expecting to scroll the page, the page will not scroll because the focus is already in a form input field. (Instead, you’ll type a space in the field.) If you focus a different input field while the page is still loading, the site’s autofocus script may “helpfully” move the focus back to the original input field upon completion, disrupting your flow and causing you to type in the wrong place.

Because the autofocus is done with JavaScript, it can be tricky to handle all of these edge cases, and there is little recourse for people who don’t want a web page to “steal” the focus.
To solve this problem, HTML5 introduces an autofocus attribute on all web form controls. The autofocus attribute does exactly what it says on the tin: it moves the focus to a particular input field. But because it’s just markup instead of a script, the behavior will be consistent across all websites. Also, browser vendors (or extension authors) can offer users a way to disable the autofocus behavior.

Checking for autofocus support uses detection technique #2 (see “Detection Techniques” on page 15). If your browser supports autofocusing web form controls, the DOM object it creates to represent an <input> element will have an autofocus property (even if you don’t include the autofocus attribute in your HTML). If your browser doesn’t support autofocusing web form controls, the DOM object it creates for an <input> element will not have an autofocus property. You can detect autofocus support with this function:

```javascript
function supports_input_autofocus() {
    var i = document.createElement('input');
    return 'autofocus' in i;
}
```

Instead of writing this function yourself, you can use Modernizr (see “Modernizr: An HTML5 Detection Library” on page 16) to detect support for autofocused form fields:

```javascript
if (Modernizr.input.autofocus) {
    // autofocus works!
} else {
    // no autofocus support :
    // fall back to a scripted solution
}
```

**Microdata**

Microdata is a standardized way to provide additional semantics in your web pages. For example, you can use microdata to declare that a photograph is available under a specific Creative Commons license. As you’ll see in Chapter 10, you can also use microdata to mark up an “About Me” page. Browsers, browser extensions, and search engines can convert your HTML5 microdata markup into a vCard, a standard format for sharing contact information. You can also define your own microdata vocabularies.

The HTML5 microdata standard includes both HTML markup (primarily for search engines) and a set of DOM functions (primarily for browsers). There’s no harm in including microdata markup in your web pages; it’s nothing more than a few well-placed attributes, and search engines that don’t understand the microdata attributes will just ignore them. But if you need to access or manipulate microdata through the DOM, you’ll need to check whether the browser supports the microdata DOM API.
Checking for HTML5 microdata API support uses detection technique #1 (see “Detection Techniques” on page 15). If your browser supports the HTML5 microdata API, there will be a `getItems()` function on the global `document` object. If your browser doesn’t support microdata, the `getItems()` function will be undefined. You can check for support as follows:

```javascript
function supports_microdata_api() {
    return !!document.getItems;
}
```

Modernizr does not yet support checking for the microdata API, so you’ll need to use a function like this one.

**Further Reading**

Specifications and standards:
- The `<canvas>` element
- The `<video>` element
- `<input>` types
- The `<input placeholder>` attribute
- The `<input autofocus>` attribute
- HTML5 Storage
- Web workers
- Offline web applications
- The geolocation API

JavaScript libraries:
- **Modernizr**, an HTML5 detection library
- **geo.js**, a geolocation API wrapper

Other articles and tutorials:
- Video for Everybody!
- “A gentle introduction to video encoding”
- “Video type parameters”
- The Appendix of this book