

ROCKABLE*



DECODING HTML5

HTML5 | CSS3 | JavaScript | DOM | Web API | Game Development

Jeffrey Way



DECODING HTML5

ROCKABLE *

Rockablepress.com
Envato.com

© Rockable Press 2012

All rights reserved. No part of this publication may be reproduced or redistributed in any form without the prior written permission of the publishers.

Contents

Introduction	8
<i>What to Expect</i>	8
<i>Let's Keep it Informal</i>	9
<i>What is HTML5?</i>	9
<i>Before We Begin</i>	11
The History of HTML5	17
<i>What's the Difference Between the W3C, WHATWG, and HTMLWG?</i>	17
<i>HTML vs. XHTML</i>	18
<i>Fight, Fight, Fight!</i>	20
<i>A Line in the Sand</i>	22
The State of HTML5	28
Semantic Markup	35
<i>What to Remove</i>	35
<i>HTML5ify</i>	41
<i>Inline Elements</i>	59
<i>What About New Browsers?</i>	60
Easy Queries with the Selectors API	64
<i>querySelector()</i>	65
<i>querySelectorAll()</i>	66
<i>A Couple Notes of Caution</i>	69
Custom Data Attributes	73
<i>Usage Options</i>	73
<i>Accessing Data Attributes with JavaScript</i>	76
<i>A Final Word of Caution</i>	78

Fun Fun Forms	80
<i>Elements</i>	80
<i>Attributes</i>	84
<i>New Input Types</i>	92
<i>Final Project</i>	103
<i>Summary</i>	116
The Essentials of Feature Detection	118
<i>Input Types</i>	118
<i>Input Attributes</i>	119
<i>Elements</i>	120
<i>Local Storage</i>	121
<i>Various APIs</i>	122
<i>Automated Detection with Modernizr</i>	124
Finally... Native Media	138
<i>Back in the Day</i>	138
<i>What HTML5 Video is not Appropriate For Usage</i>	139
<i>A Brief Overview of Codecs</i>	142
<i>Video Encoding Tools</i>	151
<i>What About Mom?</i>	152
<i>It Doesn't End There</i>	154
<i>Controlling Video with JavaScript</i>	156
<i>The Full Screen API</i>	156
<i>Summary</i>	181
Track that Sucka with Geolocation	185
<i>What is Geolocation?</i>	185
<i>Crash Course</i>	186
<i>Testing</i>	188
<i>The Two Core Methods</i>	188
<i>Google Maps</i>	192
<i>Summary</i>	209

The Basics of Painting with Canvas	211
<i>What is Canvas</i>	212
<i>Feature Detection</i>	212
<i>"Hello Canvas"</i>	213
<i>Paths</i>	215
<i>Animations</i>	225
<i>Generating Noise</i>	232
<i>101 Class Complete</i>	235
Don't Irritate Visitors — Use Web Storage	238
<i>What is Local Storage?</i>	239
<i>Is Local Storage HTML5?</i>	240
<i>How Do I Use It?</i>	240
<i>Test What You've Learned —</i>	
<i>Comment Form</i>	243
<i>How to Publish, or Announce Changes</i>	250
<i>Project 2 — Tasks</i>	255
<i>Form Data</i>	262
<i>Storing Objects</i>	267
<i>Summary</i>	269
The History API	271
<i>The history Interface</i>	271
<i>history.pushState</i>	273
<i>The popstate Event</i>	274
<i>Project</i>	274
<i>The Job Isn't Finished</i>	283
The File && Drag and Drop APIs	285
<i>Feature Detection</i>	286
<i>Capturing File Information</i>	288
<i>Multiple Files</i>	290
<i>Drag and Drop</i>	291
<i>Reading Files</i>	295
<i>Reporting Progress</i>	302

Web Workers are Ants	307
<i>Say Hello to Web Workers</i>	308
<i>A Crash Course</i>	309
<i>A Second Example</i>	317
<i>Importing Scripts</i>	320
<i>Inline Workers</i>	320
<i>Dealing with Errors</i>	325
Tools, Folks, and Blogs	329
<i>Tools</i>	329
<i>Folks</i>	335
<i>Sites</i>	338
<i>Mailing Lists</i>	339
Closing Notes	341
About The Author	342

INTRODUCTION

Introduction

What to Expect

Let's cut to the chase; if you're expecting a massive, all-encompassing analysis of the HTML5 spec, then you're reading the wrong book. For the complete resource, refer to the — wait for it — HTML5 spec! On the other hand, if you're in need of a book that will get you up and running with many of the new tags, form elements, and JavaScript APIs as quickly as possible, you've come to the right place! We won't cover every new API, but I'll detail the ones which I feel you'll get the most use out of right now.

My job is to decipher the massive and confusing spec, and transform it into something that the every day John Doe designer or developer can immediately pick up and understand. As such, this book will focus less on the politics of HTML5 (though we will touch on it), and more on ways to immediately integrate it — and its friends — into your web projects.

Wait—Friends?

It's true. Though this book has the term HTML5 in its title, we'll also be covering other new technologies, which aren't officially part of the HTML5 spec. This will include things like [geolocation](#) and [web storage](#). Some of these specifications were once part of the HTML5 core, but, due to various reasons — both practical and political — were exported to their own specifications.

ROCK*

TIP

These are technologies that you will want to learn! Whether or not they officially carry the HTML5 badge doesn't really matter. They represent the modern technologies that we all should be using. We're going to learn about them anyways!



Let's Keep it Informal

I'm not here to advertise how smart I am, or use jargon and terminology that you don't (yet) understand. This book is not a dictionary — it's a guide. This means that you should fully expect me to use personal and informal writing — almost as if I'm sitting next to you at your desk. Brace yourself for the occasional joke that only I find funny. If you're going to invest hours into this book and the projects within, then we should cut the formalities, and get to know each other!

So without further ado, and in the words of the great [Leroy Jenkins](#), let's do this!

What is HTML5?

Ready to get knocked off your feet? HTML5 is... HTML. It's simply the next version, which includes a new shortened **doctype**, a huge array of new semantic tags (such as **header** and **footer**), enhanced form support, native **audio** and **video**, and the ability to create complex games and graphics with **canvas**. It (and its friends) also provides us with access to a variety of fun new JavaScript APIs, such as geolocation, drag-and-drop, web workers, and local storage.

2022

You've likely heard at some point that HTML5 won't be "recommended" until 2022. Please don't let this dissuade you from embracing the techniques outlined in this book.

The idea behind this seemingly arbitrary date is that at least two browsers must completely pass the HTML5 test suites in order for HTML5 to be considered a "proposed recommendation."

“ We’re also fully intending to do something that none of the aforementioned specs really did, which is to have a comprehensive test suite that we will require at least two browsers to completely pass before we call it a day. **”** — *Ian Hickson*

In 2006, Ian Hickson (you’ll learn more about him in the next chapter) proposed a general timeline for the HTML5 lifecycle.

- First W3C Working Draft in October 2007.
- Last Call Working Draft in October 2009.
- Call for contributions for the test suite in 2011.
- Candidate Recommendation in 2012.
- First draft of test suite in 2012.
- Second draft of test suite in 2015.
- Final version of test suite in 2019.
- Reissued Last Call Working Draft in 2020.
- Proposed Recommendation in 2022.

The most important bullet point in this list is the fourth one: “Candidate Recommendation in 2012.” This is a far more realistic date, as far as developers are concerned. This signals the year when HTML5, in terms of its feature-set, is complete. And what do you know: that’s this year!

The truth is that HTML5 and its friends are ready to use right now. True, some of the specs and APIs are still in the early stages, but, again, you can still reliably use most of them... right now.

To offer a comparison, we gladly used CSS 2.1 for years and years before the specification was upgraded to “recommended” status in 2009. The same is true for HTML5.

Besides, as we all know, the world will end before 2013 hits. So, if you don’t use HTML5 this year, you never will!

Before We Begin

The surge in jQuery usage in the last several years has been so dramatic to the point that many newer designers and developers use it exclusively, without taking the time to truly learn and understand vanilla JavaScript. Though I’m not one to criticize these folks for making that decision, as an author, though, it does make my job a bit more difficult.

Do I appeal to the masses and exclusively use jQuery examples — potentially irritating some readers who prefer a different library (or none at all) — or do I stick with vanilla JavaScript and hope that the reader understands it — all the while knowing that many will not?

As much as possible, I will provide both (modern) vanilla JavaScript **and** jQuery solutions; however, in a few DOM-heavy chapters, we’ll stick with jQuery in order to focus more exclusively on the subject matter of the chapter.

A Word to the Vanilla JavaScript Folks

Those who prefer the vanilla JavaScript snippets will likely be aware of the necessary quantity of code that must be written to achieve cross-browser compliance. As such, for convenience, I’ll often refer to modern JavaScript techniques and methods which may not be widely supported, such as `forEach`.

I'd like there to be an understanding that, when I use `document.querySelector()`, you're expected to know that [older browsers](#) don't support it. Most of the time, this will be a non-issue, since IE7 obviously doesn't support the new APIs to begin with!

Additionally, I'll often use `addEventListener` ► `Listener` exclusively. As you may be aware, Internet Explorer 8 implements its own event API, called `attachEvent`.

This means, when we need to listen for specific events, such as clicking an element, or focusing an `input`, we have to determine whether or not we need to use the standardized `addEventListener`, or IE's `attachEvent`. If I refrain from using `attachEvent`, it should be understood that it has only been omitted for brevity's sake (or because it's not necessary — IE8 can't take advantage of the feature or API), and that you should provide your own `addEventListener` solution.

Here is one that you might use:

```
var addEvent = (function () {
    var filter = function(el, type, fn) {
        for (var i = 0, len = el.length; i < len; i++) {
            addEvent(el[i], type, fn);
        }
    };
    if (document.addEventListener) {
        return function (el, type, fn) {
            if (el && el.nodeName || el === window) {
                el.addEventListener(type, fn, false);
            } else if (el && el.length) {
                filter(el, type, fn);
            }
        }
    }
});
```

ROCK★ TIP

Internet Explorer 9 has since upgraded to the recommended model.



```
    };
}

return function (el, type, fn) {
    if ( el && el.nodeName || el === window ) {
        el.attachEvent('on' + type, function () { return      ▶
            fn.call(el, window.event); });
    } else if ( el && el.length ) {
        filter(el, type, fn);
    }
};

})();

// usage
addEvent( document.getElementsByTagName('a'), 'click', fn);
```

This function simply normalizes the process of attaching cross-browser event listeners to one or more elements. We can use this custom event listener, like so:

```
var lis = document.getElementsByTagName('li');

// Arg 1 - the elements to apply the event to,
// Arg 2 - the type of event
// Arg 3 - the function to execute

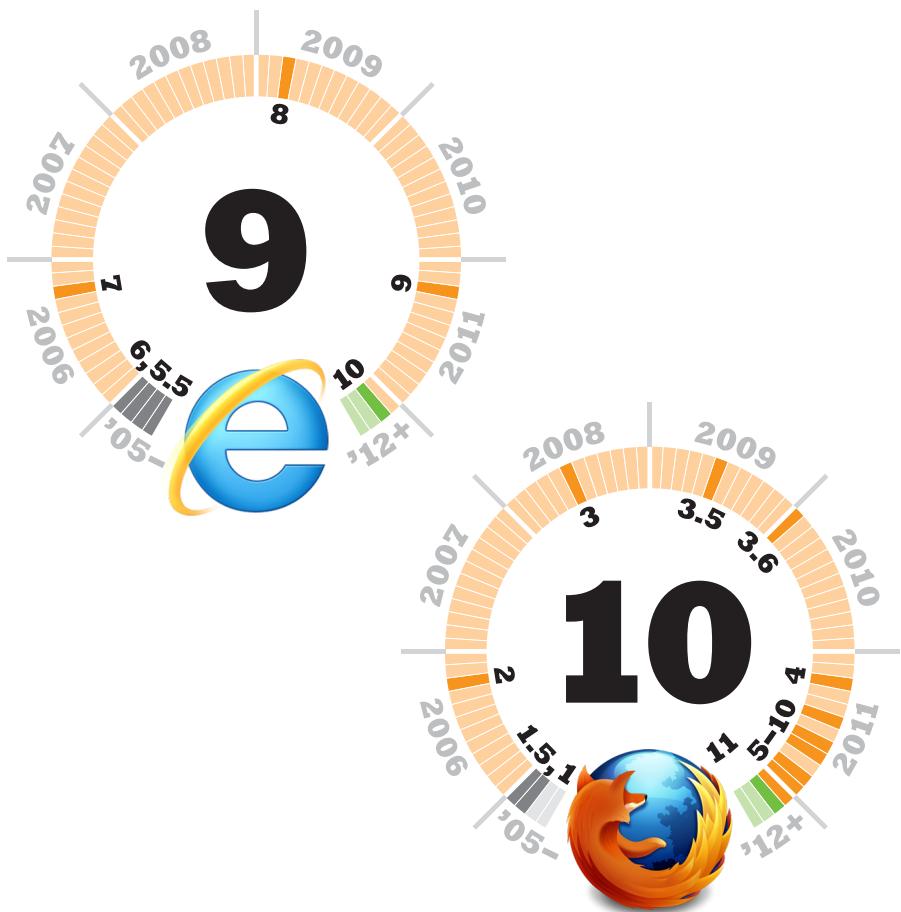
addEvent(lis, 'click', function() {
    alert('clickety clack!');
});
```

On the other hand, if you're new to the JavaScript world, and are mostly familiar with jQuery, that's okay; ignore this entire function. You won't be using it! Nothing to see here, folks; move on.

Next on this journey, we're on to a brief history of HTML5. I know what you're thinking: "Hmm... I'll skip to Chapter 3." Don't!

Browser Support

Throughout this book are graphs showing support for various features across browsers. The timespan shown is primarily from 2006 to 2011, with older versions and future releases tipping off the ends, as shown here in the Introduction. This allows you to compare the relative availability of features as some browsers have a very rapid versioning/release cycle (Chrome) while others are more relaxed (Safari). In the rest of the book, full support is indicated with a solid orange, partial support with a faded orange, and no support in light gray.







The History of HTML5

This is that chapter you typically skip over. It's the one where I don't detail an ounce of code, but instead describe the important events that lead to what you now recognize as HTML5. Some of us find this stuff interesting, but, certainly, a history lesson may not be what you had in mind when you picked this book up.

Wait — you're still here? Let's get on with it then. We won't travel as far back as the beginning. That's an entire book on its own. Instead, we'll rewind the clock to the release of HTML 4.01, at the tail-end of the nineties.

What's the Difference Between the W3C, WHATWG, and HTMLWG?

- **W3C** – A community with the sole purpose of working to develop web standards.
- **WHATWG** – Formed after various members of the W3C became agitated by the direction being taken with XHTML 2.0. They preferred a different, less drastic, approach, where the existing HTML was extended.
- **HTMLWG** – Once the W3C finally recognized that XHTML 2.0 was not the future, they indicated that they wished to work with the WHATWG on development of what would eventually become HTML5. They chartered the HTMLWG for this purpose.

If that still sounds confusing, don't worry. Continue reading for the full story.

HTML vs. XHTML

Right around the period that HTML advanced to version 4.01 (around 1998), the ground began to shift a bit. Developers started to talk about this next new thing the W3C was working on: XHTML, which stood for “Extensible Hypertext Markup Language.” This first 1.0 specification was more or less identical to HTML 4.01, other than the inclusion of a new MIME type, `application/xhtml+xml`.

Believe it or not, we’ve always been able to get away with omitting quotations around attribute values (for the most part), and not including closing tags. However, up until recently, it was widely considered to be a bad practice. For the youngsters among you readers, the reason why we viewed it as a bad practice is largely due to the popularity of XHTML.

Think of XHTML as your grandmother. When she comes to visit, she forces you to brush your teeth, stand up straight, and eat your peas. Now replace *teeth*, *posture*, and *peas*, with *quotation marks*, *self-closing tags*, and *lowercase tag names*.

Though I kid, mostly, we viewed XHTML 1.0 as a good thing — the **next step**. It required designers and developers to follow a set of standards when creating markup. How could that be bad? The irony is that, though we followed these new rules, the majority of us continued serving pages with the `text/html` MIME type, which meant that the browser didn’t really care **how** we created our markup. This way, XHTML could be **opt-in**.

So we were writing markup in a certain, strict fashion to pass XHTML validation that had zero effect or influence on the browser’s rendering. A bit odd, huh?

XHTML 1.1

This all changed with the introduction of [XHTML 1.1](#) — a significant shift toward pure XML. With this release, the `application/xhtml+xml` MIME type was required. Sure, this may sound like the natural next step, in theory, but there were a couple of glaring issues.

1. “Save to Disk”

First, Internet Explorer could not render documents with this MIME type. Instead, it would prompt a **save to disk** dialog. Yikes!

I've also been reading comments for some time in the IEBlog asking for support for the `application/xhtml+xml` MIME type in IE. I should say that IE7 will not add support for this MIME type — we will, of course, continue to read XHTML when served as `text/html`, presuming it follows the HTML compatibility recommendations.  — [Chris Wilson](#)

2. Take No Prisoners

Secondly, XHTML 1.1 was sort of like Professor Umbridge from Harry Potter: extremely harsh. Have you ever noticed how, if you leave off a closing `` tag, the browser doesn't flinch? Browsers are smart, and compensate for your **broken** markup (more on this shortly). While these days the community is beginning to embrace and take advantage of this truth, with XHTML the W3C wanted to enforce XHTML's stricter syntax. Though, up to this point, developers could get away with leaving off, say, the closing `<head>` or `<body>` tag, the W3C implemented a new *fail on the first error* system, known as *draconian error handling*. If an error was detected, the browser was expected to cease rendering the page and display an according error message. Like I said: incredibly harsh for markup.

As a result, few of us ever used XHTML 1.1; it was too risky. Instead, we adopted general XML best practices, and continued to serve our pages as `text/html`.

XHTML 2

In their minds, the W3C was finished with HTML 4. They even shut down and rechartered the HTML Working Group, and transferred their focus to XHTML — or, at this point, XHTML 2.0.

XHTML2 was an effort to draw a line, fix the web, and right the wrongs of HTML. Though, again, this sounds fabulous, in truth, it angered much of the community due to the fact that it was never intended to be backward compatible with HTML 4. In fact, it was entirely different from XHTML 1.1 as well!

Get where I'm going here? For all intents and purposes, they were ignoring the current web, and the demands of its developers, in favor of a pure XML-based approach. It simply wasn't practical to expect such a huge transition.

XHTML2 was never finalized.

Fight, Fight, Fight!

(Okay, not as Fight Club as that.) Right around this time, the idea that, “**Hey – maybe we should return to HTML and work off that**” began to come up again. Work had begun on [Web Forms 2.0](#), which managed to spark renewed interest in HTML, rather than scrapping it entirely for XHTML2. This notion was put to the test in 2004, during a [W3C workshop](#), where the advocates for HTML presented their case, and the work they had already done with Web Forms 2.0.

Unfortunately, the proposal was rejected on the grounds that it didn't fall in line with the original goal of working toward XHTML2.

Needless to say, this rejection angered some in the group, including representatives from Mozilla and Opera.

The group consequently branched out, and formed the [WHAT Working Group](#) (or *Web Hypertext Application Technology Working Group*), after, for lack of better words, becoming pissed off at the way XHTML 2 seemed to be heading. Their goal was to keep from throwing the baby out with the bath water. Continue and extend development of HTML via three specifications: Web Forms 2.0, Web Apps 1.0, and Web Controls 1.0.

The Two Golden Rules

This new group would embrace two core principles:

1. Backward compatibility is paramount. Don't ignore the existing web.
2. Specifications and implementations **must** match one another. This means that the spec should be incredibly detailed (hence, the 900 pages).

“ *The Web Hypertext Applications Technology working group therefore intends to address the need for one coherent development environment for Web Applications. To this end, the working group will create technical specifications that are intended for implementation in mass-market Web browsers, in particular Safari, Mozilla, and Opera. **”***
— [WHATWG.org](#)

Parser

While XHTML 2.0 intended to enforce perfect XML, the WHAT Working Group instead took it upon themselves to document exactly how HTML is and should be [parsed](#) — a five year task!

Remember when we discussed how browsers do a great job of compensating for your broken markup? The interesting thing is that, before the creation of the WHAT Working Group, there wasn't any specification that provided instructions to the browser vendors for how to deal with errors. This naturally leads up to the question: how did the browsers match one another's error handling? The answer is through tireless (though essential) reverse engineering. Firefox, Safari, and Opera copied Internet Explorer's implementation, and Internet Explorer reverse engineered Netscape's handling.

Over the course of five years, the WHATWG charted out what's now referred to as the [HTML5 parser](#). Don't underestimate how significant an achievement this was. Today, all modern browsers parse HTML according to the guidelines of this specification. Though perhaps not as sexy as, say, `canvas`, the HTML5 parser is a massive achievement.

A Line in the Sand

As you might expect, an imaginary line was drawn in the sand. You were either for XHTML2, or (what would eventually become) HTML5.

Rather than a consensus-based approach, where members debated and voted on what they felt was best, the WHATWG took a bit more of a dictator-like stance, with Ian Hickson at the helm.

Wait—Dictator?!

Don't we usually try to overthrow these power mongers? What's the deal? I must admit, on paper it sounds awful, doesn't it? One guy determines the future of the web. We prefer this system? Politically speaking, yes, a dictatorship is a bad idea. But, when you think about it in terms of the web, imagine how much more

quickly things can get done. When a community is as passionate as ours, things tend to move very slowly, as debates continue on and on and on.

“*The Web is, and should be, driven by technical merit, not consensus. The W3C pretends otherwise, and wastes a lot of time for it. The WHATWG does not.* **”**
— Ian Hickson

While discussion certainly (and rightfully) takes place at the WHATWG, ultimately, Ian Hickson has his finger on the button, unless the group and community strongly disagrees with a particular decision. At this point, he can either be *impeached* (not in the Bill Clinton sort of way), or, more often than not, he'll re-evaluate and reverse his decision.

That said, it's certainly not ideal. The W3C has its slow and steady consensus-based approach, which many still prefer. On the other hand, while the WHATWG moves at a quicker pace, there certainly are hiccups. Then, when you combine the two groups, things can sometimes get a bit muddy!

The `time` Debacle

In October, 2011, Ian Hickson took the initial steps to remove the new HTML5 element, `time`, from the specification. Instead, it was to be replaced by the more generic, `data`. In his own words:

“*There are several use cases for <time>:*
a. Easier styling of dates and times from CSS.
b. A way to mark up the publication date/time for an article (e.g. for conversion to Atom).
c. A way to mark up machine-readable times and dates for use in Microformats or microdata.

Use cases A and B do not seem to have much traction.

Use case C applies to more than just dates, and the

lack of solution for stuff outside dates and times is being problematic to many communities.

Proposal: we dump use cases A and B, and pivot <time> on use case C, changing it to and making it like the <abbr> for machine-readable data, primarily for use by Microformats and HTML's microdata feature. „ — [Ian Hickson](#)

What he possibly didn't realize was that much of the community did, in fact, use the `<time>` tag. Further, they (myself included) felt that, though more flexible, the proposed `<data>` tag was too ambiguous; `<data>` has as much meaning as a ``, when it comes to semantics. After a significant level of uproar from the community, the HTMLWG announced that the `<time>` change must be reverted. They gave Ian a short deadline to make the reversal. Though not without additional layers of drama, the following month `<time>` was reinstated.

This chain of events simply proves that, even though Ian has the right to **steamroll** these sorts of changes, the web community, as a whole — and, of course, the browser vendors — have quite a bit of control over the specification. There's a difference between the spec creators, and the authors who integrate these new elements and APIs into their projects. If the authors don't use them, they might as well be removed from the spec. Remember: you have much more control over the shape of the web than you likely give yourself credit for!

Sign up for the various [mailing lists](#) and be loud! Otherwise, folks like Ian won't know if or how you use these new features.

“ Is there any data showing how people actually use `<time>` in practice? i.e. is it actually giving anyone any of its hypothetical benefits? „ — [comment by Ian Hickson](#)

The Shape of a Specification

While some may think that a small group of people determine the future of the web, that's far from the case. Three factions receive equal weight, when it comes to specifications.

1. **Spec Creators** – Obviously...
2. **Authors** – People like us; if we reject (i.e. don't use) a particular element or API, it might as well be dead in the water.
3. **Vendors** – Browsers have a huge amount of input into these specifications, many times leading the way.

If you'd like to learn more about the `<time>` debacle, [review the bug thread](#), and Ian's [Google+ post](#) on the subject. They're both interesting reads, and aren't nearly as black or white as you might think.

Back at the W3C...

Back to the W3C vs. WHATWG feud. Well, it was less a feud, and more like two groups ignoring one another for a couple years.

While work at the WHATWG progressed relatively quickly, work on XHTML 2.0 at the W3C was — how should I put this — not going well (almost non-existent). As time progressed, it became clearer and clearer that XHTML 2.0 was not the solution (though it wouldn't be fully dropped until 2009). In 2006, the W3C relented, and signaled that they were interested in collaborating with the WHATWG on (what would be) HTML5. They chartered yet another group for this purpose: HTMLWG, or the *Hypertext Markup Language Working Group*.

They intended to use the work of the WHATWG as a basis for continued development of HTML. Weird, huh? Now we have two

different groups: the W3C HTMLWG and the WHATWG. Technically, the W3C hadn't yet given up on XHTML. Nonetheless, as part of the newly formed HTMLWG, they renamed Web Apps 1.0 to HTML5.

“Apple, Mozilla, and Opera allowed the W3C to publish the specification under the W3C copyright, while keeping a version with the less restrictive license on the WHATWG site. **”** — WHATWG.org

Today

These days, the WHATWG and W3C collaborate with one another on HTML5. It's a bit of an odd relationship, but somehow manages to function, thanks to an endless supply of incredibly passionate activists.

2

The State of HTML5

In the last several years, the popularity of HTML5 has skyrocketed. You've undoubtedly seen HTML5 used in the titles of countless, "traffic seeking" web development tutorials (even [mine](#)) around the web. It's quite possible that your clients and non-developer friends are also throwing around the terms HTML5 and CSS3, as if they have the slightest concept of what these technologies are, or how they can improve their websites.

A sibling of mine — also a developer — recently told me that his boss condescendingly instructed him to "abandon JavaScript and embrace HTML5"... Yes, these people do exist. For better or worse, HTML5 is officially a buzz word.

Remember, in 2009, when the rest of the world discovered Twitter — long after the tech community (thanks, Oprah) embraced it? Suddenly, every news channel and talk show host felt the need to throw around the word Twitter (or "Twitters"), as if it somehow possessed magical abilities to increase their "cool factor." Well the term HTML5 is sort of like that! As long as you don't walk around saying "HTML5s," you should be good!

What's the Scuttlebutt?

When we uses phrases, like "The State of HTML5," what we really mean is: which new features can we reliably use without destroying our web applications in older browsers that don't adequately support HTML5? Alternatively, which features, generally referred to as "edge features," aren't quite ready for everyday usage?

An excellent resource to bookmark and refer to is [CanIUse.com](#). For example, if I'm curious about which browsers I can use local storage in, looking up that term on the site reveals that we can use it in most browsers these days, excluding Internet Explorer 7 and below, and Safari 3.

The screenshot shows the CanIUse.com compatibility table for the 'local storage' feature. The table includes columns for various browsers (IE, Firefox, Chrome, Safari, Opera, iOS Safari, Opera Mini, Opera Mobile, Android Browser) and their versions. It also includes columns for 'Usage stats' (Global Support: 87.34%, Partial support: 0.59%, Total: 87.93%) and a 'Feedback' button.

	IE	Firefox	Chrome	Safari	Opera	iOS Safari	Opera Mini	Opera Mobile	Android Browser
3 versions back	6.0	5.0	12.0	3.2	10.6	3.2		10.0	
2 versions back	7.0	6.0	13.0	4.0	11.0	4.0-4.1		11.0	2.1
Previous version	8.0	7.0	14.0	5.0	11.1	4.2-4.3		11.1	2.2
Current		8.0	15.0		11.5	5.0	5.0-6.0	11.5	2.3
Near future	9.0	9.0	16.0	5.1	11.6				4.0
Farther future	10.0	10.0	17.0	6.0	12.0				

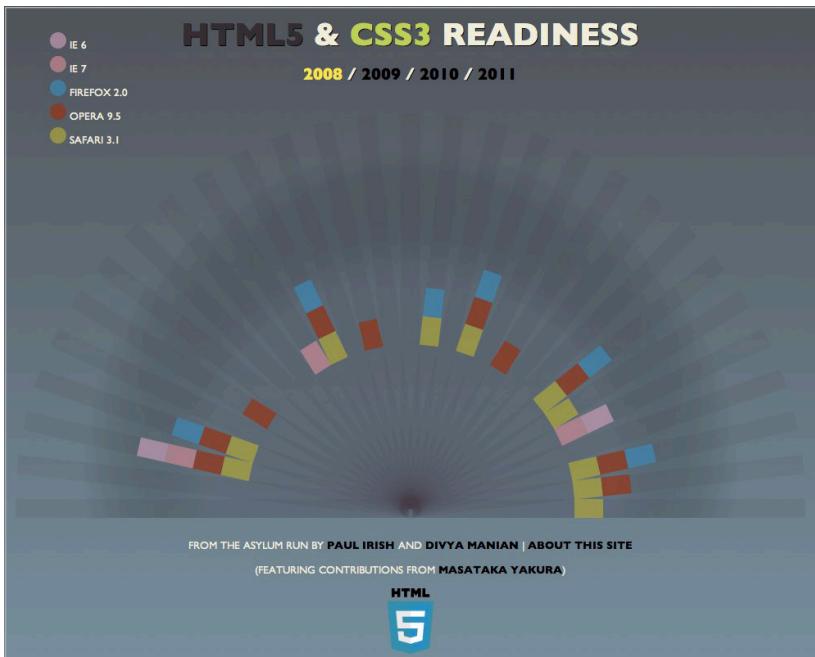
Browser support for local storage at CanIUse.com.

The Browser Wars

Thankfully, in the last two years, we've experienced a new age of browser wars as Firefox, Opera, and Chrome compete for best standards support (and highest version number). Also in that time span, as Internet Explorer's market share plummeted to all time lows, Microsoft finally realized that, hey, standards are important! Though they're still lagging behind in many areas, Internet Explorer 10 is a vast improvement over its predecessors.

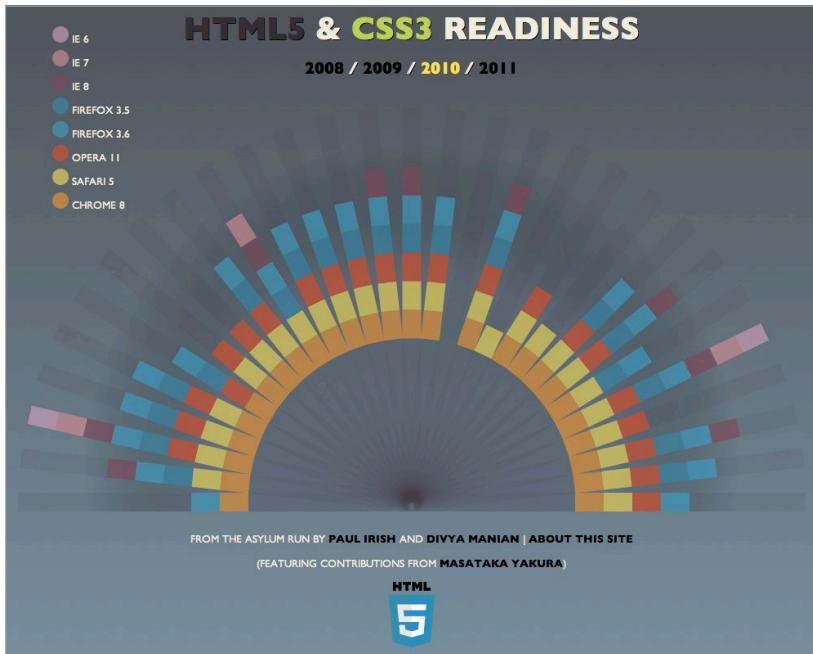
Browser Support from 2008–2011

[Paul Irish](#) and [Divya Manian](#) are incredibly active in the web development community, and have built various helpful tools, such as [HTML5 Boilerplate](#), [CSS3 Please](#), and [HTML5 and CSS3 Readiness](#). The latter of the bunch offers an excellent way to review just how far browsers have come in the last four years.

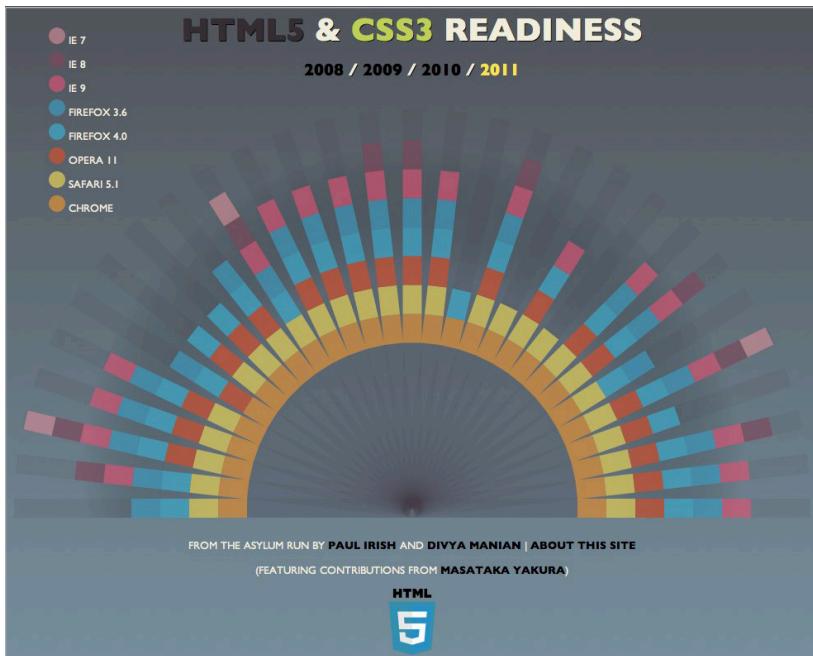


2008 Statistics (above); (below) 2009 Statistics (below).





2010 Statistics (above); (below) 2011 Statistics (below).



Needless to say, 2010 and 2011 offered significant advances in browser support for the various new HTML5 and CSS3 modules. It's a fantastic time to be a web designer or developer! I encourage you to toy around with these two websites a bit.

While I could ramble on about browser support for miscellaneous features like HTML5 parsing and local storage, the truth is: you don't care (yet). We need to learn how to use these technologies before we can worry about browser support and *polyfills*.

Wait – Polyfills?

I promised I wouldn't use confusing terminology, but already in Chapter 2 I'm throwing around terms like "polyfill!" Don't worry. It's just a word. Let's imagine that you want to take advantage of HTML5's ability to display a date picker. We can create a form `input` with a `type` of `date` to accomplish this task.

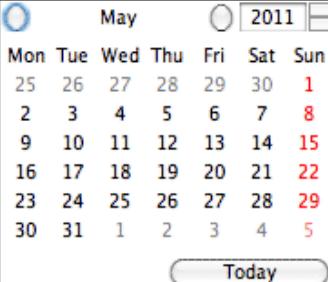
```
<input type="date">
```

How will that display? Well, to be honest, it depends on your browser. At the time of this writing, you'll see:

chrome

2011-05-25

Opera



Firefox



Yikes! No support in Firefox 9. Internet Explorer 9? Forget about it! Wouldn't it be great if we could take advantage of the excellent date picker support in Opera, while still providing JavaScript fallbacks for older browsers? Of course it would! This is where the term "polyfill" comes into play.

For now, file that one away. We'll refer to it later in this book.

ROCK*

TECH TERM

A **polyfill** is what we refer to as a shim that mimics APIs, providing fallback functionality for older, less capable browsers. Remy Sharp can tell you [more about polyfills](#).



3

Semantic Markup

Browser Support



While HTML5 certainly provides us with access to plenty of fun new APIs (which you'll learn about soon), we also now have access to a plethora of HTML elements that can more accurately describe our content.

Examples

- Do you generally wrap your blog articles within a `div`? Well now, you can use the more appropriate `article` element.
- What about the footer of your website, where you list your copyright information and additional links? With HTML5, you can use the appropriately named `footer` element.

What to Remove

Before we get into these various new HTML elements, let's first learn what can safely be removed! Refer to the following typical (but dated) HTML page.

```
<!DOCTYPE HTML PUBLIC "-//W3C//DTD HTML 4.01 Transitional//  
EN" "http://www.w3.org/TR/html4/loose.dtd">  
<html>  
<head>  
<meta http-equiv="Content-Type" content="text/html;  
charset=UTF-8" />
```

```
<title>Old</title>
<link type="text/css" rel="stylesheet" href="style.css" />
<script type="text/javascript" src="scripts.js"></script>
</head>
<body>
  <h1> My Old Site </h1>
</body>
</html>
```

There's nothing particularly **wrong** with the markup above. That said, there's quite a few needless bits and pieces, which can easily be removed.

It's time for a haircut. Let's start at the top, work our way down, and cut, cut, cut.

DOCTYPE

First, we have that confusing **DOCTYPE**.

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

Admit it: you never memorized this. I sure didn't! Now, we can use a much shorter and simpler version.

```
<!DOCTYPE html>
```

“But Jeffrey, what about older browsers? We can't forget them! **”**

There's nothing to worry about. All browsers will look at this **DOCTYPE** and switch the content into standards mode, regardless of whether the browser version supports HTML5 or not. This means that you can confidently use this shorter **DOCTYPE** in all of your projects without hesitation.

Have Some Fun

We mustn't always be so serious. In fact, you can personalize the DOCTYPE, if only to serve as a silly **reward** for those crazy **View Source** kids.

```
<!DOCTYPE html public ''>
```

You see those quotation marks after **public**? That's your space. Write what you wish—the browser doesn't care. In consideration of how most kids these days suffer from ADD, let's keep their focus by providing a unicode video game controller.

```
<!DOCTYPE html public '🎮'>
```

Or some words of wisdom on developer etiquette?

```
<!DOCTYPE html public 'ly calling out your fellow developers  
is in poor taste. Be constructive, but polite.'>
```

meta

Next, we come to another long string: the **meta** tag. This is where the character encoding for the document is declared.

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

Let's replace the whole thing with something considerably more memorable.

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

Wonderful! Again, we can reliably use this in all current browsers. Now, if you're thinking, “Jeffrey, you're making mistakes. Where is the closing forward slash?”, it's important to understand that doing

so has always been optional. With the rise of XHTML, we (along with the validator) deemed it a best practice to always close tags in this way. However, they've always been 100% optional. If it keeps you in the good graces of the church, though, feel free to continue closing your tags with `>`. I won't lose any sleep over it though.

script and link Types

Lastly for this example, we come to the `link` and `script` tags.

```
<link type="text/css" rel="stylesheet" href="style.css" />
<script type="text/javascript" src="scripts.js"></script>
```

When referencing stylesheets and scripts, we can remove `type="text/css"` and `type="script/javascript"`, as it's redundant to include them. Those are the default types.

```
<link rel="stylesheet" href="style.css">
<script src="scripts.js">
```

While we're here, it's likely that the `script` file doesn't need to be loaded within the `head` section of the page.

And with that last change, our example markup should now look like so:

```
<!DOCTYPE html>
<html lang="en">
<head>
  <meta charset="utf-8">
```

ROCK*

TIP

Always remember to include this tag, and place it before the title tag. A page that does not specify a character set can potentially leave Internet Explorer users open to an XSS attack. Technically, the rule is that it needs to be declared within the first 512 bytes of the page.



ROCK*

TIP

If a script is not vital to the immediate display or functionality of a page, it's more performant to place the script at the bottom of the page, just before the closing body tag.



```
<title>New</title>
<link rel="stylesheet" href="style.css">
</head>
<body>
  <h1> My New Site </h1>
  <script src="scripts.js"></script>
</body>
</html>
```

Optional Quotations

Unless quotations are required, they're optional. This means that, just as we can write `role="main"`, we can alternatively omit the quotations: `role=main`. Please note, however, that they are required in some instances, such as when applying multiple classes.

```
<!-- Bad -->
<div class=wrap main>

<!-- Good -->
<div class="wrap main">
```

In this case, due to the space, the HTML5 parser will incorrectly assume that `main` is an attribute name, rather than a value. All of the following are perfectly acceptable in HTML5.

```
<div class="box">
<div CLASS="box">
<div class=box>
```

As with everything we've discussed in the last few pages, whether or not you choose to wrap your attributes within quotations is up to you. Most coders who have the teachings of XHTML engrained within them will find it difficult to revert back. They may even see it as **sloppy** markup. The truth is, in situations like this, it comes

down to personal preference. I find that I generally prefer them, while other developers like to omit them.

It's important to remember (from Chapter 1) that HTML5 is not XHTML. The XHTML validator enforced various restrictions, such as closing tags and wrapping all attributes within quotes. But the browsers never truly cared either way. They've always been able to read your HTML pages, with or without quotation marks.

When I type HTML like this (omitting quotes), I love it.

It's like walking around naked in your apartment.

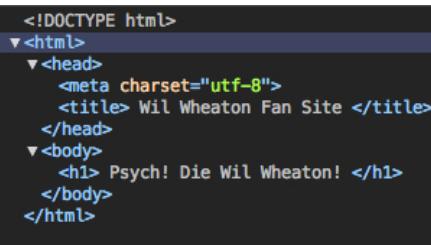
— Paul Irish

Unnecessary Closing Tags

I'll let you in on a little secret: many HTML tags are technically optional. This includes everything from the `html` tag, to the `tbody`. In fact, the following snippet will work just fine.

```
<!DOCTYPE html>
<meta charset=utf-8>
<title> Wil Wheaton Fan Site </title>
<h1> Psych! Die Wil Wheaton! </h1>
```

The browser will correctly insert the missing tags. Don't believe me? Paste the previous snippet into a file, and view it in the browser. Here's a snapshot from the element inspector in [Chrome Developer Tools](#).



The screenshot shows the DOM tree for the provided HTML snippet. The root node is `<!DOCTYPE html>`, which has a child node `<html>`. The `<html>` node has two children: `<head>` and `<body>`. The `<head>` node contains `<meta charset="utf-8">` and `<title> Wil Wheaton Fan Site </title>`. The `<body>` node contains `<h1> Psych! Die Wil Wheaton! </h1>`.

Neat, right? There's lots of tags which can be omitted. Leave those **suckas** off. Don't worry. The browser has your back. Further, many closing tags are optional as well. If not present, the browser adds them. In the example below, the `html`, `head`, `body`, and `li` closing tags have been omitted.

```
<!DOCTYPE html>
<html>
  <head>
    <meta charset="utf-8">
    <title> Wil Wheaton Fan Site </title>
  <body>
    <ul>
      <li> Die
      <li> Wil
      <li> Wheaton
    </ul>
```

Now there's certainly a lot to be said for structure. I'm not suggesting that you go crazy, but, still, it's always helpful to understand how the browser works. If closing your list items makes you feel secure (and clothed), then, by all means, close them. However, don't think for a second that the browser will break if these closing tags are left off. I promise — they won't.

ROCK*

TIP

While I prefer to close most tags, I frequently take advantage of this technique in blog posts. When creating bulleted lists or paragraphs, I rarely add the closing tag.



HTML5ify

In 2008, [Opera MAMA](#) analyzed millions of URLs to determine what the most popular `ids` and `class` names were. As you might expect, at the top of this list rested the names that we're all too familiar with: `#header`, `#footer`, and `.wrapper`.

In the ongoing effort to better describe our content, it certainly made sense to create new HTML elements to fill these rolls. Why use a meaningless `div` when a more appropriate and meaningful element can be implemented?

Sectioning Elements

header and footer

We now have access to `header` and `footer` elements, though their usage may certainly exceed the traditional page header and footer spots. Consider the following typical layout.

```
<div id="header">
  <h1>My Website</h1>
  <ul>
    <li><a href="#">Home</a></li>
    <li><a href="#">About</a></li>
    <li><a href="#">Contact</a></li>
  </ul>
</div>
<div id="main">
  <div class="article">
    <h1> My Blog Post </h1>
    <h4> Posted <a href="#">today</a> </h4>
    <p> My post content goes here. </p>
  </div>
</div>
<div id="footer">
  <h5>My Website - 2011</h5>
  <ul>
    <li><a href="#">Home</a></li>
    <li><a href="#">About</a></li>
    <li><a href="#">Contact</a></li>
  </ul>
</div>
```

The markup above surely looks familiar to you. It's the typical **bread-and-butter** layout: a header, the main content, and then the footer. Let's **HTML5ify** this markup, and remove those now superfluous ids.

We begin by replacing both `<div id="header">` and `<div id="footer">` with the more appropriate **header** and **footer** elements.

```
<header>
  <h1>My Website</h1>
  <ul>
    <li><a href="#">Home</a></li>
    <li><a href="#">About</a></li>
    <li><a href="#">Contact</a></li>
  </ul>
</header>
<div id="main">
  <div class="article">
    <h1> My Blog Post </h1>
    <h4> Posted <a href="#">today</a> </h4>
    <p> My post content goes here. </p>
  </div>
</div>
<footer>
  <h5>My Website - 2011</h5>
  <ul>
    <li><a href="#">Home</a></li>
    <li><a href="#">About</a></li>
    <li><a href="#">Contact</a></li>
  </ul>
</footer>
```

Please note that, while it's perfectly okay to use these two elements in the traditional *page-header* and *page-footer* format, you are not limited only to those uses. It would also be appropriate,

for example, to use a **header** tag to designate the header of a blog posting, or the section where you place the title, meta information for the post, which categories it was assigned to, etc. Further, if your article contains information at the bottom, that links to other relevant articles, or social network links, you may also freely wrap this markup within a **footer** tag.

nav

Next, let's tackle the navigation links. In its current form, we have a simple and usable unordered list.

```
<ul>
  <li><a href="#">Home</a></li>
  <li><a href="#">About</a></li>
  <li><a href="#">Contact</a></li>
</ul>
```

There's certainly nothing wrong with this markup, but we have to ask ourselves, "Does this markup **describe** the links?" Not really. At most, it describes "a list of links"; nothing more.

Let's wrap this markup within the new **nav** tag.

```
<nav>
  <ul>
    <li><a href="#">Home</a></li>
    <li><a href="#">About</a></li>
    <li><a href="#">Contact</a></li>
  </ul>
</nav>
```

Your first thought may be, "**Wait – I still have an unordered list, and, now, I've added yet another tag. What is all this for?**"

Our job, when it comes to creating markup, is to describe the content as eloquently and accessibly as possible. This is why it's

preferable to wrap a list of links within an unordered list, rather than stacking a number of anchor tags on top of one another. Sure, you can get away with doing so, but it doesn't **describe** the content as well as an unordered list does. The same is true for the new HTML5 elements.

Always consider accessibility, or, more directly, how your content is interpreted by a screen reader. When a screen reader scans your page, how will it know that your navigation links are, in fact, site navigation links, and not, perhaps, links to your favorite websites? It doesn't. However, HTML5 provides us with the means to fix this. Now, assuming the screen reader has been upgraded to interpret HTML5, it can properly pinpoint exactly where your navigation section is. Please note that, as with **headers** and **footers**, you are not limited to a single **nav** element. It's equally likely that, in the page **footer**, you'll also have an additional set of navigation links.

ROCK*

TIP

As a rule of thumb, don't go overboard with the nav element, or you defeat the purpose of using them entirely! If you wrap every list of links on your website within a nav element, the screen reader is back to having no clue which list specifically refers to the site navigation.



article

Before HTML5, we were forced to use meaningless class names to designate and target articles, news items, or blog postings. Now, we can refer to the **article** element in these situations. Again, remember that applying a **class** of **article** — `<div class="article">` — does nothing more than provide you with a hook to target or style that section. It does not describe the content within.

If we refer back to the markup that we're incrementally improving, we should currently have:

```
<header>
  <h1>My Website</h1>
  <nav>
    <ul>
      <li><a href="#">Home</a></li>
      <li><a href="#">About</a></li>
      <li><a href="#">Contact</a></li>
    </ul>
  </nav>
</header>
<div id="main">
  <div class="article">
    <h2> My Blog Post </h2>
    <h4> Posted <a href="#">today</a> </h4>
    <p> My post content goes here. </p>
  </div>
</div>
<footer>
  <h5>My Website - 2011</h5>
  <nav>
    <ul>
      <li><a href="#">Home</a></li>
      <li><a href="#">About</a></li>
      <li><a href="#">Contact</a></li>
    </ul>
  </nav>
</footer>
```

Once we replace `<div class=article>` with the more appropriate **article** tag, we now have:

```
<header>
  <h1>My Website</h1>
  <nav>
    <ul>
      <li><a href="#">Home</a></li>
```

```
<li><a href="#">About</a></li>
<li><a href="#">Contact</a></li>
</ul>
</nav>
</header>
<div id="main">
<article>
<h2> My Blog Post </h2>
<h4> Posted <a href="#">today</a> </h4>
<p> My post content goes here. </p>
</article>
</div>
<footer>
<h5>My Website - 2011</h5>
<nav>
<ul>
<li><a href="#">Home</a></li>
<li><a href="#">About</a></li>
<li><a href="#">Contact</a></li>
</ul>
</nav>
</footer>
```

“ The **article** element represents a self-contained composition in a document, page, application, or site and that is, in principle, independently distributable or reusable, e.g. in syndication. This could be a forum post, a magazine or newspaper article, a blog entry, a user-submitted comment, an interactive widget or gadget, or any other independent item of content. **”** — [HTML5 spec](#)

While the implementation in the previous example is the most obvious use for the **article** tag, it can also be used in a variety of situations. An easy way to determine whether or not to use the **article** tag is to think to yourself, “Would this content ever be

redistributed via an RSS feed, or something similar?" If the answer is yes, then go ahead and use it! It's most likely the correct choice.

Another Use-Case for the header

Remember when I noted that the **header** tag isn't only limited to page headers? It is also appropriate to use it as the **header** for the **article**.

```
<div id="main">
  <article>
    <header>
      <h2> My Blog Post </h2>
      <h4> Posted <a href="#">today</a> </h4>
    </header>
    <p> My post content goes here. </p>
  </article>
</div>
```

The main Element?

You may have noticed one glaring omission: the **main**, or **main-content** element. Unfortunately, one does not yet exist. To better describe what the boundaries are for the main content section of your website, we can take advantage of [WAI ARIA's roles](#).

```
<div id="main" role="main">
```

The **role** attribute gives us an additional means to describe the content — in this case, the **main** section.

In basic terms, `<div role=main>` signals to screen readers that this is the section where the main content for the webpage is located.

ROCK* **TIP**

WAI ARIA defines a way to make web applications more accessible to people with disabilities. It is available in all browsers, excluding Internet Explorer 7 and below.



Hooks

Even better, we can use the **role** attribute as a hook within our stylesheets. This means that, if you wish to do so, you can remove yet another **id**.

```
div[role=main] {  
    ...  
}
```

This style of CSS selector is referred to as an *attribute selector*. It allows us to target elements on the page, based upon whether they contain a certain attribute. Support for attribute selectors extends as far back as Internet Explorer 7.

section

The **section** element is easily one of the most confusing additions to HTML5. According to the [specification](#):

“ *The **section** element represents a generic section of a document or application. A section, in this context, is a thematic grouping of content, typically with a heading.* **”**

Considering the definition alone, you wouldn't be foolish to assume that it, too, could be used for a blog posting. However, for things like that, the **article** element still makes more sense, as a blog posting is more likely to be syndicated.

Instead, an easy way to wrap your head around the **section** element is to think of newspapers. They are divided into multiple sections: the sports section, the living section, the money section. This style of separation can be applied to your websites.

```
<section>  
    <h1> Sports </h1>
```

```
<p> Going... going... gone. </p>
</section>

<section>
  <h1> Living </h1>
  <p> Britney Spears shaves her child's head. </p>
</section>

<section>
  <h1> Money </h1>
  <p> OWS jibba-jabba. </p>
</section>
```

Notice how multiple `<h1>` tags are used. In HTML4, this was frowned upon. However, that is no longer the case. The `section` will create a new “stacking” context. What you must understand is that the `section` element is not meant to replace the `div`. It’s meant to group related content, which generally comes with a heading. A good rule of thumb is: if you only need a wrapping element for the purposes of positioning or styling, then a `div` is the correct choice.

Another example, which the specification references, is using `sections` to divide chapters of a book.

```
<section>
  <h1>Chapter 1</h1>
  <p>Bla Bla Bla.</p>
</section>

<section>
  <h1>Chapter 2</h1>
  <p>Bla Bla Bla.</p>
</section>

<section>
  <h1>Chapter 3</h1>
  <p>Bla Bla Bla.</p>
</section>
```

```
<section>
  <h1>Appendix</h1>
  <p>Bla Bla Bla.</p>
</section>
```

The truth is: if you still feel uneasy about the **section** element, you're not alone. There are plenty of folks who feel that **article** and **section** overlap too much. What do you think?

aside

The **aside** element represents a section of a page that is related to the content that surrounds it. The most obvious and frequent use for **aside** is when defining the page sidebar.

```
<div class=container>
  <aside>
    <h2>Sidebar</h2>
    <p>Info...</p>
  </aside>
  <div role=main>
    <p>Main content here.</p>
  </div>
</div>
```

In this position, **aside** refers to secondary content.

However, it can also be used in more specific situations, such as within an **article**. In this context, **aside** refers to content that is tangentially related to the **article**, such as a glossary.

```
<article>
  <h1>Twitter</h1>
  <aside>
    <h3>Helpful Definitions</h3>
    <dl>
```

```
<dt>Twitter(s):</dt>
<dd>Incorrect usage - refers to a grouping of status
    messages. </dd>
<dt>Tweet(s): </dt>
<dd>One or more status updates.</dd>
</dl>
</aside>
<p>The post goes here.</p>
</article>
```

details

One of my favorite new HTML5 tags, which has only recently been integrated into Chrome (as of Version 12), is the **details** element.

What Does It Do?

It essentially allows us to show and hide content with the click of a button. You're surely familiar with this type of effect, but up until now, it had always been achieved with JavaScript. Imagine a heading with an arrow next to it, and when you click on it, additional information below becomes visible. Clicking the arrow again hides the content. This sort of functionality is very common, for example, in FAQ pages.

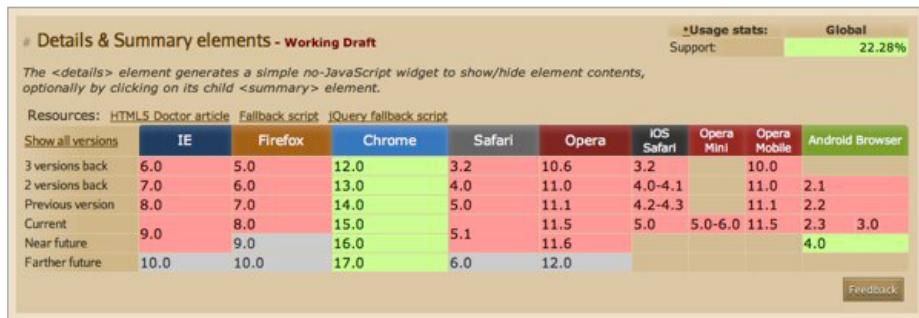
The **details** element allows us to omit the JavaScript entirely. Or, better put, it eventually will. Browser support is still a bit sparse.

ROCK*

TIP

Here's a [two minute example](#) of this sort of effect. (Type Control + Enter to process the JavaScript.)





Browser support for the details element, as of November 17th, 2011.

An Example

So let's dive in and learn how to use this new tag. We begin by creating a new **details** element.

```
<details>
</details>
```

Next, we need to give it a title, or **summary** of the content contained within.

```
<details>
<summary> Who Goes to College? </summary>
</details>
```

By default, in browsers that understand the **details** element, everything within it — other than the **summary** tag — will be hidden. Let's add a paragraph after the **summary**.

```
<details>
<summary> Who Goes to College? </summary>
<p> Your mom. </p>
</details>
```

► Who Goes to College?



Go ahead and [test the demo](#) in Chrome 12 or higher.

Okay, let's do something a bit more practical. I want to display various Nettuts+ articles using the `details` element. We first create the markup for a single article.

```
<details>
  <summary>Dig Into Dojo</summary>
  
  <div>
    <h3> Dig into Dojo: DOM Basics </h3>
    <p>
      Maybe you saw that tweet: "jQuery is a gateway drug.
      It leads to full-on JavaScript usage." Part of that
      addiction, I contend, is learning other JavaScript
      frameworks. And that's what this four-part series
      on the incredible Dojo Toolkit is all about: taking
      you to the next level of your JavaScript addiction.
    </p>
  </div>
</details>
```

▼ Dig Into Dojo



dojo
toolkit

Dig into Dojo: DOM Basics

Maybe you saw that tweet: "jQuery is a gateway drug. It leads to full-on JavaScript usage." learning other JavaScript frameworks. And that's what this four-part series on the incredible to the next level of your JavaScript addiction.

Next, we'll give it just a touch of styling.

```
body { font-family: sans-serif; }
details {
  overflow: hidden;
  background: #e3e3e3;
  margin-bottom: 10px;
  display: block;
}
details summary {
  cursor: pointer;
  padding: 10px;
}
details div {
  float: left;
  width: 65%;
}
details div h3 { margin-top: 0; }
details img {
  float: left;
  width: 200px;
  padding: 0 30px 10px 10px;
}
```

▼ Dig Into Dojo



Dig into Dojo: DOM Basics

Maybe you saw that tweet: "jQuery is a gateway drug. It leads to full-on JavaScript usage." Part of that addiction, I contend, is learning other JavaScript frameworks. And that's what this four-part series on the incredible Dojo Toolkit is all about: taking you to the next level of your JavaScript addiction.

Please note that I'm showing you the `open` state for convenience, but when the page loads, you'll only see the `summary` text. If you'd prefer to be in this state by default, add the `open` attribute to the `details` element: `<details open>`

Styling the Arrow

It's not quite as straight-forward to style the arrow itself as we might hope. Nonetheless, it is possible; the key is to use the `-webkit-details-marker` pseudo class.

```
details summary::-webkit-details-marker {  
    color: green;  
    font-size: 20px;  
}
```

► Dig Into Dojo

Should you need to use a custom icon, possibly the easiest solution is to hide the arrow (using the pseudo class above), and then either apply a background image to the `summary` element, or use the `:after` or `:before` pseudo elements.



[View the final project.](#)

It's certainly a simple effect, but it sure is nice to have such a common feature built-in. Until we can reliably use the `details` element across all browsers (it's currently [only available in Chrome](#)), you can use [this polyfill](#) to provide fallback support.

One final note: at the time of this writing, there doesn't seem to be a way to toggle the contents with a keyboard. This certainly presents some accessibility issues, though there are talks underway to get this fixed.

mark

The `mark` element offers a way to highlight text for reference purposes. As the [spec](#) puts it:

“ When used in the main prose of a document, it indicates a part of the document that has been highlighted due to its likely relevance to the user’s current activity. **”**

This means we should use the `mark` element when we need to draw the user’s attention to a specific piece of text. Most frequently, this is interpreted in two ways.

Search Results

Most CMSs offer some form of built-in search functionality. If the visitor searches your blog for “HTML5”, within the search results you could wrap each occurrence of the term with the `mark` element.

```
<article>
  <h2>Clueless Posting About <mark>HTML5</mark></h2>
  <p class="excerpt">
    In this article, I will talk about how we need to stop
    using JavaScript, and instead embrace <mark>HTML5
  </mark>
  </p>
</article>
```

Clueless Posting About HTML5

In this article, I will talk about how we need to stop using JavaScript, and instead embrace **HTML5**.

Note that the default styling for `mark` is to mimic a highlighter. As with all base styling, this can easily be overridden.

After the Fact

Imagine that you're referencing a person's quote:

```
<blockquote>
  <p> I have a strong feeling that the world will end...
    on New Year's Eve. </p>
</blockquote>
```

What if you need to call special attention to a specific section of this person's quote? Maybe New Year's has come and gone.

```
<blockquote>
  <p> I have a strong feeling that the world will end...
    <mark>on New Year's Eve.</mark> </p>
</blockquote>
```

Now we're drawing the reader's focus to that specific piece of the original quote, calling attention to the fact that the person was clearly wrong in his prediction.

figure

Consider the following markup for an image:

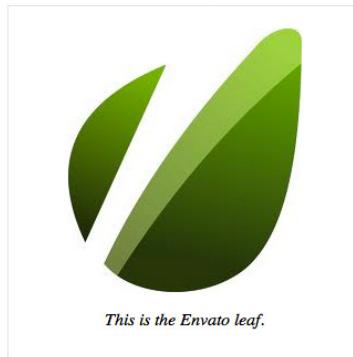
```

<p>Image of Mars. </p>
```

In the past, there was never an easy or semantic way to associate a caption with the image element itself. HTML5 rectifies this with the introduction of the **figure** element. When combined with the **figcaption** element, we can now semantically associate captions with their counterparts.

```
<figure>
  
```

```
<figcaption>
  <p>This is the Envato leaf. </p>
</figcaption>
</figure>
```



However, **figure** is not limited to only images. According to the spec:

“The **figure** element can be used to annotate illustrations, diagrams, photos, code listings, etc, that are referred to from the main content of the document, but that could, without affecting the flow of the document, be moved away from that primary content, e.g. to the side of the page, to dedicated pages, or to an appendix. **”**

Inline Elements

I bet you didn't know that all elements have a default display of **inline**. It's possible that you thought, yes, some elements are **inline**, but others have a default of **block**, such as **divs** and **h1s**. What's the deal?

Well, we say things like “default display” because it's how every browser styles these elements in their stylesheet. Without that

browser-supplied stylesheet, however, all elements would default to a display of `inline`. Herein lies our problem.

Do you get where I'm going? These new elements will naturally have a display of `inline`, which isn't ideal in most cases. You likely need your `header` and `footer` to be set to a `block` display. To work around this issue, it's important to explicitly declare a `block` display in your stylesheet for all applicable elements.

```
header, footer, article, section, nav {  
    display: block;  
}
```

ROCK*

TIP

In browsers which don't yet understand the new HTML5 elements (such as IE8 and Firefox/Safari 3), there is no browser styling for these new elements.



What About New Browsers?

The current iteration of modern browsers are BFFs with HTML5, so this isn't an issue for them. Their stylesheets will **correctly** set these elements to `block`. As these browsers see it, we're simply being redundant. No harm done. Place this at the top of your stylesheet for every new project.

That Damn IE

CSS was created to give us the ability to target and style XML elements. This means that, technically, I can create a bogus `<jeff> </jeff>` tag, and style it as I normally would within my stylesheet. CSS doesn't care what the function or purpose of a tag is. It's simply looking for an XML element with the designated name.

```
jeff {  
    font-size: 30px;  
    color: red;  
}
```



Unfortunately, Internet Explorer 8 and below — as I’m sure you’ve learned — don’t always play nicely. They completely ignore the new HTML elements! Actually, they don’t exactly ignore them; they can’t see them at all! This presents a major hurdle, as we can’t style an element if IE can’t see it! How can we take advantage of these new HTML5 tags if Internet Explorer 7 chokes on them?

Luckily, there is a solution. In 2008, [Sjoerd Visscher](#) discovered that, by merely creating the desired DOM elements — within the **head** section of the document — this somehow manages to trigger Internet Explorer into recognizing them.

```
<script>  
    document.createElement("article");  
    document.createElement("footer");  
    document.createElement("header");  
    document.createElement("section");  
    document.createElement("nav");  
    document.createElement("menu");  
</script>
```

They don't even need to be injected into the DOM. This simple act does the trick. Funny, huh? We can shorten this code a bit by using a `while` or `for` statement, like so:

```
<script>
  var els = ['article', 'footer', 'header', 'section',
    'nav', 'menu'];
  while(els.length) {
    document.createElement( els[0] );
    els.shift();
  }
</script>
```

HTML5 Shiv/Shim

Because this technique has become common-place — even required — for all HTML5 websites, [Remy Sharp](#) created a popular and simple script, [HTML5shim](#), which can easily be included in your projects. This takes care of not only the process of creating the DOM elements to make IE happy, but it also ensures that printing pages in IE works as expected.

“ Since HTML5 is getting more attention by way of marking up our new pages, and the only way to get IE to acknowledge the new elements, such as `<article>`, is to use the HTML5 shiv, I've quickly put together a mini script that enables all the new elements. **”** — [Remy Sharp](#)

As this script is only applicable to Internet Explorer, we should use conditional stylesheets to keep from adding an unnecessary HTTP request in browsers that won't benefit from the script.

```
<!--[if lt IE 9]>
<script src="http://html5shim.googlecode.com/svn/trunk/
  html5.js"></script>
<![endif]-->
```

4

Easy Queries with the Selectors API

Browser Support



Over the course of this book, you're quickly going to find that there is considerably more to HTML5 (and its friends) than a handful of new elements.

The Selectors API introduces two new ways to query the DOM. You're likely already familiar with `getElementById()`, `getElementsByTagName`, and `getElementsByClassName` (finally standardized in HTML5).

```
// Get the element with an id  
// of box  
document.getElementById('#box');  
  
// Collect all list items  
document.getElementsByTagName('li');  
  
// Search for all elements with  
// a class of 'selected'  
document.getElementsByClassName('selected');
```

Technically, the Selectors API is not part of HTML5. As with plenty of topics in this book, it's contained within [its own W3C specification](#). Nonetheless, we will be using the following methods often in future chapters. Pay attention.

Now we can use the CSS selectors that you’re already familiar with to dive into the DOM. JavaScript library (jQuery) users will feel right at home!

querySelector()

This function will only return the first element that matches a specified CSS selector.

```
// Find the first element with a class of box document.  
document.querySelector('.box');
```

```
// Find the first input that has a placeholder attribute  
document.querySelector('input[placeholder]');
```

```
// Find the first list item within #container  
document.querySelector('#container li');
```

```
// Get the first paragraph within the div with a role  
// of "main"  
document.querySelector('div[role=main] > p:first-child');
```

```
// Above, the child selector ('>') and ':first-child'  
// pseudo class are superfluous. querySelector will always  
// return the first match from the selector.  
// It can be rewritten as:  
document.querySelector('div[role=main] p');
```

Notice how we’re using the simple CSS selectors we’re all familiar with at this point. The huge bonus, though, is that we’re taking advantage of the browser’s native, super-speedy engine to perform queries, rather than using slow JavaScript loops.

The `document` needn’t always be the context for `querySelector`. An element may also be used as the base, or “starting point” for the query.

```
// Get first .box
var box = document.querySelector('.box');

// Get first span that is within an li
box.querySelector('li span');
```

querySelector()

When you only require a single element from the DOM, use `querySelector`; it's optimized for this very purpose. However, many times, we need to capture multiple elements within the DOM. In these situations, `querySelectorAll` is the correct choice.

```
// Select all elements which match this selector
document.querySelectorAll('li span');

// Capture all external links
document.querySelectorAll('a[href^=http]');
```

This function is particularly helpful in situations when you need to bind event handlers to a set of elements.

```
[].forEach.call( document.querySelectorAll('a[href^=http]'),>
  function(anchor) {
    anchor.addEventListener('click', function(e) {
      e.preventDefault();
      alert('You just clicked an external link');
    }, false);
});
```



Grouping

To select all `em` and `strong` tags from within the document, you can pass a comma-separated list to both `querySelector` and `querySelectorAll`.

```
// Return the first em or strong
// element that you find
document.querySelector('em,           ▷
strong');

// Return all em and strong
// elements
document.querySelectorAll('em, strong');
```

ROCK*

TIP

`querySelectorAll` will return a static `NodeList`, which is very much like an array, but not quite. As such, we can't use methods like `forEach` on it directly. Calling `forEach` on the array's prototype remedies this limitation.



Does it Match?

Excluding Internet Explorer 8, all modern browsers support `matchesSelector`, which allows us to determine whether a DOM element matches a specified selector.

```
<div>
  <a href="#">The Last Starfighter Fan Site</a>
</div>
<script>
  var a = document.querySelector('a');
  a.matchesSelector('div a'); // true
  a.matchesSelector('ol a'); // false
</script>
```

This can be particularly useful when dealing with event delegation. Event delegation is a technique in which a single event listener is applied to a common ancestor, rather than to potentially dozens of elements.

```
<ul class="links">
  <li> <a href="#">The Last Starfighter Fan Site</a> </li>
  <li> <a href="#">War Games Fan Site</a> </li>
  <li> <a href="#">Space Camp Fan Site</a> </li>
</ul>
<script>
  // Attach a single listener to the window
  window.addEventListener('click', function(e) {
    if ( e.target.webkitMatchesSelector('.links a') ) {
      alert('Event delegation & matchesSelector are BFFs');
      e.preventDefault();
    }
  }, false);
</script>
```

Vendor Prefixes

At the time of this writing, all supporting browsers implement their own prefixed version of `matchesSelector`.

```
el.webkitMatchesSelector();
el.mozMatchesSelector();
el.oMatchesSelector();
el.msMatchesSelector();
```

To accommodate each browser, our code should be updated, accordingly.

```
var matches;
(function(doc) {
  matches = doc.matchesSelector ||
            doc.webkitMatchesSelector ||
            doc.mozMatchesSelector ||
            doc.oMatchesSelector ||
            doc.msMatchesSelector;
})(document.documentElement);
```

```
window.addEventListener('click', function(e) {  
  if ( matches.call(e.target, 'ul a') ) {  
    alert('Event delegation and matchesSelector are BFFs');  
    e.preventDefault();  
  }  
}, false);
```

With this technique, in Webkit, `matches` will refer to `webkitMatchesSelector`, and, in Mozilla, `matches` will refer to `mozMatchesSelector`.



A Couple Notes of Caution

Selector Limitations

While nearly all relevant browsers support the Selectors API, the specific CSS selectors you pass are still limited to the capability of the browser. Translation: Internet Explorer 8 will only support CSS 2.1 selectors.

Performance

Both `querySelector` and `querySelectorAll` will return a `NodeList` (the same as what you'd expect from `getElementsByName`), however, there's one very important distinction: they return a `static NodeList`, rather than `live`.

Live NodeLists

Okay, let's tackle live `NodeLists` first. Consider the following code:

```
<a href="#">One</a>  
<a href="#">Two</a>  
<a href="#">Three</a>  
<a href="#">And to the Four</a>
```

```
<script>
  var anchors = document.links;
  anchors.length; // 4
</script>
```

There's nothing overly confusing here. The `anchors` variable now refers to a `NodeList` of all the links on the page. But what happens if we add another anchor tag to the page?

```
var anchors = document.links;
anchors.length; // 4

// clone the first anchor, and throw it into the DOM
document.body.appendChild ( anchors[0].cloneNode(true) );
anchors.length; // 5
```

Does that surprise you? Even though we previously created the `anchors` variable, and made it equal to the existing set of links on the page at the time, its `length` property somehow reflects the new anchor that was thrown into the DOM.

This isn't a bug; it's exactly how a live `NodeList` should respond. When you think about older forms of traversal, such as `document.forms` or `document.images`, it suddenly makes perfect sense why the `NodeList` should be live. Those objects need to be as up-to-date as possible.

Static `NodeLists`

A static `NodeList`, which `querySelector` and `querySelectorAll` returns, is different. As referenced from [the specification](#):

“ Subsequent changes to the structure of the underlying document must not be reflected in the `NodeList` object. **”**

When compared to the previous code example:

```
// this time, use QSA
var anchors = document.querySelectorAll('a');
anchors.length; // 4

// clone the first anchor, and throw it into the DOM
document.body.appendChild(anchors[0].cloneNode(true));
anchors.length; // 4
```

Because we're dealing with a static **NodeList**, future modifications to the DOM will not be reflected.

As a result, unfortunately, in some browsers you may detect a considerable performance difference between using `document.getElementsByName('li')`, and `document.querySelectorAll('li')`. While a live **NodeList** can be generated and returned much more quickly, a static **NodeList** needs to have all the necessary data up front.

If this all seems confusing to you, that's okay. The most important take-away is that a static **NodeList** can't be as fast as a live one. In effect, `querySelectorAll` is slower.

However, with all things in JavaScript, pre-optimization is rarely a smart choice. Don't worry about this too much, unless you can verbalize why you should.

5

Custom Data Attributes

Browser Support



In the past, we often resorted to random element attributes for the purposes of containing, or storing data.

```
<a href="#" class="modal"> Instructions </a>  

```

With HTML5, we no longer need to reach for these attributes when we need to store metadata. Instead, we can use custom attributes.

“Custom data attributes are intended to store custom data private to the page or application, for which there are no more appropriate attributes or elements.” — [HTML5 spec](#)

```
<a href="#" data-modal>  
<img src="" data-overlay="gallery">
```

A custom data attribute is formed in the same way as all attributes. The only requirement, however, is that **data-** must be prefixed to the attribute name, to designate that it is, in fact, a data attribute.

Usage Options

The sky is the limit when it comes to usage options for data attributes.

CSS Styling

What if you need to target and style links which meet a certain criteria? For demo purposes, maybe you have a long blogroll, and need a way to highlight some of your favorites links in the bunch.

```
<ul>
  <li><a href="#">Blog 1</a>
  <li><a href="#"> Blog 2</a>
  <li><a href="#">Amazing Blog 3</a>
  <li><a href="#">Blog 4</a>
  <li><a href="#">Amazing Blog 5</a>
</ul>
```

To provide special styling — perhaps an icon or callout — to the “amazing” blogs in the list, we can use a custom attribute.

```
<ul>
  <li><a href="#">Blog 1</a>
  <li><a href="#"> Blog 2</a>
  <li><a href="#" data-amazing-blog>Amazing Blog 3</a>
  <li><a href="#">Blog 4</a>
  <li><a href="#" data-amazing-blog>Amazing Blog 5</a>
</ul>
```

Now, we have a “hook” that can easily be referenced in a stylesheet, via an attribute selector.

```
a[data-amazing-blog] {
  color: blue;
  font-weight: bold;
}
```

Or, to place a smiley-face “icon” before the amazing links, we can use the `:before` pseudo element.

- Blog 1
- Blog 2
- **Amazing Blog 3**
- Blog 4
- **Amazing Blog 5**

- Blog 1
- Blog 2
- ☺ Amazing Blog 3
- Blog 4
- ☺ Amazing Blog 5

Blog bulleting before (left) and after (right).

```
a[data-amazing-blog]:before {  
    content: url(path/to/icon.png);  
    color: blue;  
    display: inline-block;  
    width: 20px;  
    margin-left: -20px;  
}  
  
/* Or with a unicode icon */  
a[data-amazing-blog]:before {  
    content: '☺';  
    ...  
}
```

ROCK★ TIP

When you need quick unicode icons for your projects, [copypastecharacter](#) is a fantastic reference.



JavaScript Targeting

Custom attributes can also be conveniently used as targets for a particular script or plugin.

Perhaps you need an easy way to designate sections of your site which should serve as modal overlays.

```
<a href="#" data-modal data-target="contact">Contact Us</a>  
<div id="contact" style="display: none;">  
    <h2>Contact Us</h2>  
    ...  
</div>
```

In this code snippet, the anchor tag uses two custom attributes:

- **data-modal** – specifies that the anchor should serve as a modal trigger
- **data-target** – tells the script what the **id** of the target modal box is

The plugin now queries the DOM, and captures all **data-modal** elements. It then attaches a **click** event listener, which triggers the modal functionality, accordingly.

JavaScript

```
[].forEach.call( document.querySelectorAll('a[data-modal]'), ▶
  function(el) {
    el.addEventListener('click', function(e) {
      e.preventDefault();
      var target = document.querySelector('#' + el.dataset. ▶
        target);
      triggerModal( target );
    }, false);
  });

```

jQuery

```
$( 'a[data-modal]' ).on('click', function(e) {
  e.preventDefault();
  var target = $( '#' + $(this).data('target') );
  triggerModal(target);
});
```

Accessing Data Attributes with JavaScript

Custom attributes can be accessed in JavaScript much like any other attribute.

```
<a href="#" data-length="3m30s" data-category="pop">Wake Me Up Before You Go-Go </a>
<script>
  // pop
  document.querySelector('a').getAttribute('data-category');
</script>
```

That said, there's a better API for interacting with data attributes.

Dataset

Browser Support



The **dataset** object, or **DOMStringMap**, returns a list of all data-attributes for the associated element.

```
<a href="#" data-length="3m30s" data-category="pop">Wake Me Up Before You Go-Go </a>
```

Both data attributes may be accessed, like so:

```
var link = document.querySelector('a');
console.log(link.dataset);
```

```
▼ DOMStringMap
  category: "pop"
  length: "3m30s"
  ► __proto__: DOMStringMap
```

```
link.dataset.length; // 3m30s
```

jQuery

With jQuery, data attributes can be accessed, via the `data` method.

```
// contact
$('a:first').data('target')
```

A Final Word of Caution

Clearly, custom data attributes will be used more and more frequently over the coming years. As a result, the likelihood of name clashing will be a real possibility. With this mind, for large projects, rather than a generic name, such as `data-height`, you might consider using a naming convention for all of your data attributes: `data-myapp-height`. Use your best judgment in these situations.

ROCK*

TIP

If your custom attribute contains a dash, such as `data-my-attribute`, when accessing its value, you'll need to use camelCasing: `el.dataset.myAttribute`.



6

Fun Fun Forms

I know, I know; forms are boring. Luckily, HTML5 makes them — not fun — but less boring! If you've ever found yourself setting default placeholder text, performing client-side validation, or using date picker plugins, you've no doubt had first-hand experience with the limitations of the browser. Though it's taking a bit longer than we might hope, browsers are beginning to come around, with Opera and Webkit leading the pack. Let's take a stroll through some of the new options that are available to us.

Before we move forward, please remember that, just because native form validation may work in the latest version of Opera, does not mean that it will in Firefox or Internet Explorer. We can't yet depend upon these technologies for serious validation. That said, you can freely use them to serve as an additional level of protection.

Elements

output

You should reach for the **output** element when you need to display the result of a particular calculation. There's no inherent styling, so it can freely be used in all browsers.

```
<output value=100> 100 </output>
```

As an example, we could use the **output** element to represent the results of a simple form-based Math calculation.

```
<form>
  <input name=num1 id=num1 type=number> + <input name=num2 >
    id=num2 type=number> = <output></output>
</form>
```

A simple form for adding two numbers. It consists of two input fields for numbers, a plus sign, an equals sign, and a result field.

Notice that we're using a new input type of **number**. More on that later!

Association

Just as a **label** can be associated with a specific form element, the same is true for the **output**. Its **for** attribute should reference the **ids** of the elements it corresponds to (space separated).

```
<form>
  <input name=num1 id=num1 type=number> + <input name=num2 >
    id=num2 type=number> = <output for="num1 num2"></output>
</form>
```

With the markup complete, we can use JavaScript to listen for when an input is **blurred**, and update the value of the **output**, accordingly.

JavaScript

```
var form = document.querySelector('form'),
  num1 = form.num1,
  num2 = form.num2,
  output = document.querySelector('output');
[].forEach.call(document.querySelectorAll('input'), ▶
  function(el) {
  el.addEventListener('blur', function() {
    output.value = ~~num1.value + ~~num2.value;
  }, false);
});
```

jQuery

```
var form = $('form'),
  num1 = form[0].num1,
  num2 = form[0].num2,
  output = $('output');
```

```
form.on('blur', 'input', function() {  
    output.val( ~~num1.value + ~~num2.value );  
});
```

$$\boxed{3} \quad \boxed{+} \quad \boxed{5} \quad \boxed{=} \quad \boxed{8}$$

This code listens for when a user tabs away from the input. When they do, the value of the **output** element is updated to equal the sum of the two numbers.

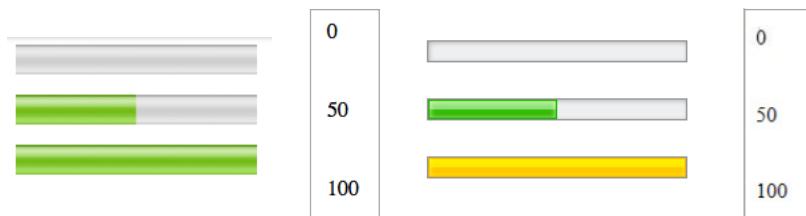


[View the output demo.](#)

meter

The new **meter** tag can be used to provide a graphical display for a particular value. This makes it the perfect choice for things like quiz scores, and disk usage indicators.

```
<meter min=0 max=100 value=50>
```



The meter element in (left to right) Chrome 16, Firefox 9, Opera 11.6, Internet Explorer 9.

At the time of this writing, only Chrome and Opera provide full support. In older browsers, the text content of the **meter** tag will display instead, as shown below.

Additionally, we can specify a range for the **low**, **medium** or **optimum**, and **high** sections of the **meter**.

```
<meter low=25 optimum=50 high=75 min=0 max=100 value=20> 20
```

```
</meter>  
<meter low=25 optimum=50 high=75 min=0 max=100 value=50> 50  
</meter>  
<meter low=25 optimum=50 high=75 min=0 max=100 value=80> 80  
</meter>
```



The effect of setting an optimum value using meter in Opera 11.6.

I'm not as young as I used to be, but I would have expected the browsers to provide three color ranges, rather than repeating the yellow for both **low** and **high**.



[View the meter demo.](#)

progress

While the **output** element is used to display the result of a calculation, the **progress** element should represent the completion progress of a task. If the browser does not support the element, its text content will be displayed instead.

```
<progress max=10 value=6> 6 </progress>
```

Two attributes determine the position of the display: **value** and **max**. Should **value** be omitted, the element will be considered indeterminate, and the display will reflect that.



The progress element in (left to right) Chrome 16 and Firefox 9, Opera 11.6, Safari 6, Internet Explorer 9.



[View the progress demo.](#)

Attributes

placeholder

Sometimes, it may be helpful to provide users with a hint as to what they're expected to type. Traditionally, JavaScript has been used to achieve this effect. Below is one possible way to accomplish the placeholder effect with JavaScript.

HTML

```
<input type="text" id="date-input" value="12/13/2011">
```

JavaScript

```
var dateInput = document.querySelector('#date-input'),  
    dateDefault = dateInput.value;  
dateInput.addEventListener('focus', function() {  
    // If this value of the input equals our sample,  
    // hide it when the user clicks on it.  
    this.value === dateDefault && this.value = '';  
}, false);  
dateInput.addEventListener('blur', function() {  
    // When they click off of the input, if  
    // the value is blank, bring back the sample.  
    this.value === '' && this.value = dateDefault  
}, false);
```

jQuery

```
var dateInput = $('#date-input'),  
    dateDefault = dateInput.val();  
dateInput.on({  
    focus: function() {
```

```
var $this = $(this);
// If this value of the input equals our sample,
// hide it when the user clicks on it.
$this.val() === dateDefault && $this.val('');
},
blur: function() {
    var $this = $(this);
    // When they click off of the input, if
    // the value is blank, bring back the sample.
    $this.val() === '' && $this.val(dateDefault);
}
});
```

This code will ensure that, when the user clicks on the input, the placeholder text immediately disappears, and, if they click off — and the input's value is blank — the placeholder text returns.

Needless to say, this is a tedious task, and one that the browsers should really offer **out of the box**. With HTML5, we can use the **placeholder** attribute to accomplish this exact effect.

```
<input type="text" id="date-input" placeholder="12/13/2011">
```



12/13/2011

How easy is that?

Keep in mind, though, that in older browsers, the input will still default to blank. It's up to you to determine whether or not the placeholder text is vital for the user to see.

Your First Polyfill

With that in mind, let's build our first polyfill to provide support for placeholder text, regardless of the browser. When creating a polyfill, you must first determine three things:

1. What is the proper, HTML5 way to accomplish this task?
2. What is the fallback (usually JavaScript-based) solution?
3. How can I test whether or not the browser supports this new feature?

Let's work through this list one at a time.

1. What is the proper, HTML5 way to accomplish this task?

We already know that. The proper way is to use the placeholder attribute.

```
<input type="text" id="date-input" placeholder="12/13/2011">
```

2. What is the fallback (JavaScript-based) solution?

Again, we know this as well. We already created a JavaScript solution.

```
var dateInput = $('#date-input')
dateInput
  .focus(function() {
    var $this = $(this);
    $this.val() === dateDefault && $this.val('');
  })
  .blur(function() {
    var $this = $(this);
    $this.val() === '' && $this.val(dateDefault);
  });
});
```

However, our method was a bit rigid; it was intended for one element. Let's revise the code to find all `input` elements, which contain the `placeholder` attribute.

```
var inputs = $('input[placeholder]'),
    dateDefault;
inputs.each(function () {
    $(this).val(this.getAttribute('placeholder'));
});
inputs.on({
    focus: function () {
        var $this = $(this);
        dateDefault = this.getAttribute('placeholder');
        $this.val() === dateDefault && $this.val('');
    },
    blur: function () {
        var $this = $(this);
        $this.val() === '' && $this.val(dateDefault);
    }
});
```

3. How can I test whether or not the browser supports this new feature?

In JavaScript, we can determine whether or not an **input** supports a particular attribute by doing:

```
if ("attribute-name" in document.createElement('input')) {
    // supported
}
if (!("attribute-name" in document.createElement('input'))){
    // NOT supported
}
```

For our situation, the test will be:

```
if (!("attribute-name" in document.createElement('input'))){
    // NOT supported
}
```

The Completed Polyfill

Now that we've identified all three components, let's group them and complete our polyfill.

HTML

```
<input type="text" id="date-input" placeholder="12/13/2011">
```

JavaScript

```
if (!('placeholder' in document.createElement('input'))) {  
    // 1  
    var inputs = document.querySelectorAll('input['  
        'placeholder']'), // 2  
        input, dateDefault;  
    // 3  
    for (var i = 0; i < inputs.length; i++) {  
        input = inputs[i];  
        input.value = input.getAttribute('placeholder');  
        // 4  
        input.addEventListener('focus', function () {  
            dateDefault = this.getAttribute('placeholder');  
            (this.value === dateDefault) && (this.value = '');  
        }, false);  
        input.addEventListener('blur', function () {  
            (this.value === '') && (this.value = dateDefault);  
        }, false);  
    }  
}
```

jQuery

```
if (!('placeholder' in document.createElement('input'))) {  
    // 1  
    var inputs = $('input[placeholder]'), // 2  
        dateDefault;
```

```
// 3
inputs.each(function () {
  $(this).val(this.getAttribute('placeholder'));
});
// 4
inputs.on({
  focus: function () {
    var $this = $(this);
    dateDefault = this.getAttribute('placeholder');
    $this.val() === dateDefault && $this.val('');
  },
  blur: function () {
    var $this = $(this);
    $this.val() === '' && $this.val(dateDefault);
  }
});
```

Though slightly more complex, the goal of this code is still the same:

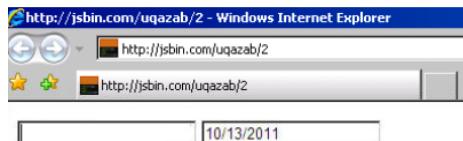
If the browser supports the **placeholder**:

1. Awesome — don't do anything. The browser has it covered.

If it does not support it...

2. Search for all inputs, which contain the **placeholder** attribute. Filter through the wrapped collection, and set the value of the **input** equal to its value.
3. When the input is focused and blurred, remove or append its value, accordingly.

In this example, we've used a simple textbox. However, as you'll learn shortly, when accepting a date value from the user, it's more appropriate to use an input **type of date**.



The `placeholder` polyfill in Internet Explorer 7.



[View the `placeholder` demo.](#)

Styling

You'll likely find that browsers are considerably inconsistent when it comes to HTML5 forms. For example, I can adjust the color of `placeholder` text in Firefox in the same way that I would with normal input text. However, Webkit is different. I'll need to use the `::-webkit-input-placeholder` property.

```
::-webkit-input-placeholder {  
  color: green;  
}
```

required

We can specify that an input requires a value without using an ounce of JavaScript — at least in the browsers which support the `required` attribute.

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

Name: Submit

This is a required field

Name: Submit Query

Please fill out this field.

Name: Submit

! Please fill out this field.

Opera 11.5

Firefox 8

Chrome 15

If the form's `submit` button is clicked, and a value has not been supplied, it will not submit, and a popup bubble will display.



[View the required demo.](#)

autofocus

Another tedious JavaScript task that we've relied on in the past is focusing a specific input. With jQuery:

```
($('textarea').focus());
```

HTML5 now offers us the **autofocus** attribute, which removes any need for JavaScript.

```
<textarea autofocus></textarea>
```



[View the autofocus demo.](#)

pattern

If the browser is going to perform validation, we need more control than simply whether or not a value is provided. This is where the **pattern** attribute comes into play. We can use regular expressions to specify precisely what format we require.

Let's imagine that I need the user to create a username, but it has to be between four and twelve characters.

```
<label for="username" Desired Username: </label>
<input type="text" id="username" required pattern=▶
  "\w{4,12}">
<input type="submit">
```

Note that there is no need to specify the beginning and end of the string with ^ and \$, respectively. The browser assumes as much.

The **required** attribute from the previous snippet may seem superfluous. Why is it necessary if we're already writing a pattern that ensures that there's at least four characters? A bit strangely, if

we omit the **required** attribute, the form will submit successfully. It appears to be a bug. Here's [an example of the issue](#).



[View the pattern demo.](#)

maxlength

HTML5 introduces the **maxlength** attribute, which allows us to limit the number of characters that may be typed into an **input** or **textarea**. Oddly, though you might expect it, we do not also have **minlength**. Continuing with the username example, if we need to limit the number of characters to twelve (without using the **pattern** attribute), we could do:

```
<input type="text" maxlength="12">
```

In browsers which support the **maxlength** attribute, the user will not be able to type more than twelve characters. The browser will not register or display anything beyond that.



[View the maxlength demo.](#)

New Input Types

In addition to the various new attributes in HTML5, we also have a wide array of new input **types**.

- **email**
- **range**
- **color**

- `number`
- `datetime`
- `date`
- `month`
- `week`
- `time`
- `url`
- `tel`
- `search`

Let's take a look!

email

If we apply a type of `email` to form inputs, we can instruct the browser to only allow strings that conform to a valid email address structure. That's right; built-in form validation is here! We can't 100% rely on this just yet, for obvious reasons.

```
<input type="email">
```


Please enter a valid email address

Please enter an email address.

Please enter an email address.

Opera 11.5

Firefox 8

Chrome 15

range

The `range` type is an exciting one, and allows us to create native sliders.

```
<input type="range">
```

Most notably, it can receive `min`, `max`, `step`, and `value` attributes, among others.

For a quick demonstration, let's build a gauge that will allow users to specify how much they enjoy the movie "Clueless," on a scale of 1–10. We won't build a real-world polling solution, but we'll review how it could be done quite easily.

Markup

First, we create our markup.

```
<form>
  <h1> Clueless Greatness Gauge </h1>
  <input type="range" id="range" name="range" min="0"
         max="10" step="1" value="5">
  <output name="result" for="range"> </output>
</form>
```

Clueless Greatness Gauge



Notice that, in addition to setting `min` and `max` values for our slider, we can also specify what the `step` for each transition will be. In our case, we'll stick with one, however, we can also make the `step` interval equal to two, so that the available choices are two, four, six, eight, and ten.

CSS

One glaring issue is that there's no built-in option for the user to see what the value is that they're selecting! Perhaps it was decided that the exact value is less important to the user. Their decision is relative to the available notches on the slider. The exact value is

only relevant to the website backend. Whatever the reason for the decision was, let's create our own simple display.

We'll utilize the `:before` and `:after` pseudo-elements in order to inform the user what our designated `min` and `max` values are.

```
input[type=range]:before { content: attr(min); }
input[type=range]:after { content: attr(max); }
```



We could then continue applying some styling to render the numbers exactly how we wish. However, there's still no easy way to visualize what the currently selected value is. Let's take advantage of the new `output` element to fix this.

output

We'll update the HTML to include the `output` element.

```
<form>
  <h1> Clueless Greatness Gauge >
  </h1>
  <input type="range" name="clueless-gauge" id="clueless-gauge" min="0" max="10" step="1" value="5">
  <output for="clueless-gauge"></output>
</form>
```

ROCK★ TIP

The `attr` function is fun; it can be used to capture values from the associated element's attributes. To determine what the max value for the input is equal to, we can set the content property to `attr(max)`.



JavaScript

Now, we'll use JavaScript to set the value of the **output**.

```
(function() {  
  var range = document.querySelector('#clueless-gauge'),  
      output = range.nextElementSibling;  
  // Set the output's initial value to the specified value  
  // for the range.  
  output.value = range.value;  
  // When the slider is changed, update the value of  
  // the output.  
  range.addEventListener('change', function() {  
    output.value = range.value;  
  }, false);  
}());
```

jQuery

```
(function() {  
  var range = $('#clueless-gauge'),  
      output = range.siblings('output');  
  // Set the output's initial value to the specified value  
  // for the range.  
  output.val( range.val() );  
  // When the slider is changed, update the value of  
  // the output.  
  range.on('change', function() {  
    output.val( range.val() );  
  });  
}());
```

This snippet first makes the value of the **output** equal to what we've specified in the **value** attribute of the **range** **input**. Then, we **listen** for when the slider is adjusted, and update the value of the **output** accordingly.

Clueless Greatness Gauge



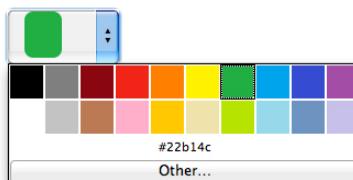
9

[View the range demo.](#)

color

Though not supported beyond Opera at the time of this writing, we'll eventually be able to provide color pickers by using the `color` input type.

```
<input type="color">
```



Opera 11.5

What happens in browsers that don't support the `color` type? Luckily, they degrade gracefully, and become simple text boxes.

[View the color demo.](#)

number

The `number` input can be used to limit a user's input to only digits — perhaps when you require their age or year of birth. These sorts of inputs can absolutely be used right now, as, again, they

fall back gracefully to simple text boxes. The only caution is that you should still perform your own validation before processing the data.

```
<input type="number">
```

Opera 11.5

Firefox 8

Chrome 15



[View the number demo.](#)

date

The **date** input provides us with the ability to force the user to select a valid and formatted date. In its current state, implementation is scattered, at best.

```
<input type="date">
```

A screenshot of the Opera 11.5 browser showing a date input field. A calendar overlay is displayed, showing the month of November 2011. The days of the week are labeled from Monday to Sunday. Specific dates are highlighted in red, including the 6th, 13th, 20th, 27th, and the 4th of December. The input field itself is empty.

Firefox 8

Chrome 15

◀ Opera 11.5

- Opera 11 will display a nice, though barely skinnable, calendar view. This is helpful, because the user has no way to type a non-valid date.
- Chrome and Safari have yet to implement a UI for their respective implementations, though it's in the pipeline.

- As of version 8, Firefox has yet to provide support for date inputs.

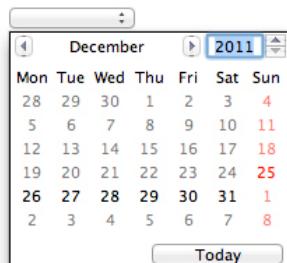
Options

min/max

Some websites force you to select a date that occurs within a specific timespan. The `min` attribute may be used for this purpose:

```
<input type="date" min=
"2011-12-25" max="2011-12-31">
```

In browsers which display a calendar view, only those specified dates will be selectable.



step

Alternatively, if we want to limit the user's date selection to be, say, within seven day increments, we can use the `step` attribute accordingly — which defaults to 1.

```
<input type="date" step="7">
```

Assuming the current date is 2011-11-10, the available choices will then be `2011-11-10, 2011-11-17, 2011-11-24`, etc.

ROCK*

TIP

What if the user specifies an invalid date? According to the spec, “If the value of the element is not a valid date string, then set it to the empty string instead.”

Alternate Datetime Options

In addition to the simple `date` type, HTML5 also offers a handful of alternate choices:

- `datetime` – `yyyy-mm-dd HH:MM`

- **month** – yyyy-mm
- **week** – yyyy-mmW
- **time** – HH:MM

Datetime:	<input type="datetime"/> 2011-11-19T19:36	<input type="datetime">
Date:	<input type="date"/> 2011-11-18	<input type="date">
Month:	<input type="month"/> 2011-10	<input type="month">
Week:	<input type="week"/> 2011-W45	<input type="week">
Time:	<input type="time"/> 14:36	<input type="time">

Cross-Browser Support

In situations where a cross-browser datepicker is paramount for a web application, [jQuery UI's datepicker](#) is a popular fallback.



[View the date demo.](#)

url

The **url** type helps to ensure that the user's input is a “valid” url.

```
<input type="url">
```

The image shows two browser screenshots side-by-side. On the left, in Firefox 8, a user has typed 'yo' into a text input field. A black callout bubble below the field contains the text 'Please enter a URL.' On the right, in Chrome 15, the same user has typed 'yo' into a text input field. A white callout bubble with a yellow exclamation mark icon above it contains the text 'Please enter a URL.'

Firefox 8

Chrome 15

I've placed “valid” in quotes because, certainly, the browser can't verify that the url is 100% active. It merely determines whether or not the passed string seems to take the form of a url.

Unfortunately, consistency from browser to browser is splotchy at best.

- In Firefox 8 and Chrome 15, as long as the text contains a colon, it will pass validation. Surely, the implementation could have been better!
- In Opera 11.5, the browser will submit any text that is entered, but it will prepend `http://` if it was not added by the user.



[View the url demo.](#)

tel

The `tel` (telephone) input does not impose any restrictions on the entered text. The current advantage comes in the form of mobile phones. In mobile browsers which understand this new type, they will correctly display a number pad when bringing up the keyboard.

```
<input name="phone" type="tel">
```

If you find yourself in the position of needing to validate a phone number, you'll want to take advantage of the `pattern` attribute. For instance, let's imagine that I need the user to enter a USA-formatted phone number:

```
<input name="phone" type="tel" pattern="\d{3}-?\d{3}-?\d{4}">
```

This will ensure that the form will only submit if the provided value takes the form of 555-555-5555 or 5555555555. If the area code is optional (not advisable), we can revise our regular expression: `(?:\d{3}-?)?\d{3}-?\d{4}`.



[View the tel demo.](#)

search

The **search** input, at first glance, doesn't do much more than create search-bar-like box with rounded corners.

```
<input type="search">  
<input type="submit">
```



We can extend this functionality somewhat. In Webkit, the perhaps oddly named **results** attribute will display a search icon on the left-hand side of the input, and allow the browser to **remember** your previous search queries.

```
<input type="search" results="5">  
<input type="submit">
```



For these previous search queries to persist across page loads, I'm told that applying the **autosave** attribute will do the trick, though I've honestly struggled to get it to work properly in any capacity.

```
<input type="search" results="5" autosave="search-queries">
```

Styling

Be cautious when styling **search** inputs. Perhaps in an effort to achieve consistency across the web, Webkit browsers seem to have made it particularly difficult to adjust the display of the **input**. CSS properties, such as **background** and **line-height** will have zero effect on the visuals. To compensate, we'd need to reset

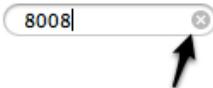
ROCK*

TIP

Refer to Chris Coyier's [article on HTML5 search inputs](#) for additional details.



`webkit-appearance` for the applicable pseudo-element. For instance, to adjust the visuals of the `cancel` icon...



that displays after entering a search query in Webkit, we'd do:

```
::-webkit-search-cancel-button {  
    -webkit-appearance: none;  
    /* Now apply your own styling */  
}
```



[View the search demo.](#)

Final Project

Let's now take everything we've learned, and create a modern HTML5 form. True server-side validation will be beyond the scope of this project, as preferred languages will vary from reader to reader. Additionally, we'll skip any necessary polyfills, as we've yet to truly dig into feature detection and Modernizr. Instead, let's focus on using the new input types as they were intended.

Guidelines

Let's imagine that our client needs us to create a sign-up form in order to capture the following information from the new fictional user:

- **Name** – required
- **Email Address** – required, must be an email address
- **Phone Number** – optional, if provided, should be in the format of a USA number, provide sample number

- **Age** – optional, if provided, must be a number
- **Date of Birth** – required, must be a year, provide sample
- **Website** – optional, if provided, should be in the format of a url
- **Bio** – optional, should allow the user to write a few paragraphs about himself

Markup

We first create our standard wrapping `form` element. Because this isn't a true form with a backend, we'll set the `action` attribute to post to the current page.

```
<form method="post" action="">  
</form>
```

Fieldsets

Next, we need to determine how we wish to organize and separate our inputs. It has always been a best practice to use `fieldsets` to organize your `form` sections; it's just that many of you haven't!

```
<fieldset>  
  <legend> Personal Information: </legend>  
</fieldset>
```

In addition to better describing our content, the `legend` attribute allows us to assign a simple, browser-styled heading for our `form` section.

Personal Information: _____

Lists

It's fairly common to find `inputs` being separated with wrapping paragraph tags. The reason why is because a paragraph immediately provides a helpful bit of spacing and separation. Honestly, though not a huge issue, this is lazy. How does a `p` tag describe an `input`? It doesn't. It's only being used for the purposes of styling, which can easily be achieved with CSS.

Instead, I tend to feel that an unordered list is the most appropriate choice. When the user looks at a `form`, he sees a **list** of `inputs` that he needs to fill out.

```
<form method="post" action="">
  <fieldset>
    <legend> Personal Information: </legend>
    <ul>
      <li> </li>
    </ul>
  </fieldset>
</form>
```

The easiest way to tackle the client's demands is to take it **one bullet point** at a time.

Name – required

This lets us know that we require a simple textbox, and that the user **must** insert a value. Well that's easy!

ROCK*

TIP

Never allow default styling to deter you from using the most appropriate elements. Ignore the bullets. They're easily removable.

```
<li>
  <label for="name">Name: </label>
  <input id="name" name="name" type="text" required>
</li>
```

Personal Information:

- Name:

Remember to always apply an `id` to your `inputs` and `textareas`. The `label's for` attribute refers specifically to an `id`. If we ignore that attribute, clicking on the label won't do a thing.

Email Address — required; must be an email address

Hmm... this one requires a bit more effort. While we can certainly get away with using a simple text box here, let's use the new `email` type that we learned about in this chapter.

```
<li>
  <label for="email">Email Address: </label>
  <input id="email" name="email" type="email" required>
</li>
```

We don't need to worry about validation in this case, as the browser will automatically determine whether or not a properly formatted email address has been inserted.

“ *The truth is: browsers currently do a subpar job of checking for email addresses. As long as an @ symbol exists between a set of letters, it will pass. As such, “a@a” is considered valid by the browser.* **”**

Phone Number — optional; should be in the format of a USA number; provide sample number

This one will require the new `tel` input type, a bit of custom validation, as our client needs us to ensure that a USA-formatted phone number (555-555-5555) is provided. We even need to provide a sample, or `placeholder`, for the user. Let's see...

```
<li>
  <label for="phone">Phone: </label>
  <input id="phone" name="phone" type="tel" pattern=▶
    "\d{3}-?\d{3}-?\d{4}" placeholder="555-555-5555">
</li>
```

This regular expression, `\d{3}-?\d{3}-?\d{4}`, will search for three **d**igits, followed by an optional dash, then three more, then, again, an optional dash, and then four final digits. This way, 555-555-5555 passes, and 5555555555 does as well.

We've also provided a sample phone number — just in case the user needs a nudge in the right direction.

Personal Information:

- Name:
- Email address:
- Phone:
-

Age — optional; must be a number

At first glance, this seems like an easy one. Smack a **number** type on that **input**, and be done with it.

```
<li>
  <label for="age">Age: </label>
  <input id="age" name="age" type="number">
</li>
```

Unfortunately, this is where we come to a snag. Webkit doesn't seem to listen to the **pattern** attribute when a **number** type is applied to an **input**. It sort of performs its own validation. But what if we want to further specify that the entered age must be

between one and three digits? `<input type="number" pattern="\d{1,3}">` will not work (at the time of this writing). In these cases, a **pattern** isn't too important; we can instead use the **min** and **max** attributes.

```
<li>
  <label for="age">Age: </label>
  <input id="age" name="age" type="number" min="1" max="120">
</li>
```

Date of Birth — required; must be a year; include sample

We know that we can use a **date** type (or one of its siblings), however, in this case, we require a date in the form of Year-Month-Date. With inputs like **age**, it's not a huge issue if it falls back to a textbox in older browsers. But the date is an important one! Ideally, we'd use both the HTML5 version, as well as a polyfill, using jQuery UI and Modernizr. For now, we'll stick with a simple **date** input, with the understanding that a JavaScript fallback is required for real-world usage.

```
<li>
  <label for="dob">Date of Birth: </label>
  <input id="dob" name="dob" type="date" required>
</li>
```

Even though the client has requested a placeholder for the **input**, it's probably best if we leave that one off. Technically, the **placeholder** attribute on **date** inputs will be ignored entirely, though, even if it wasn't, the example date may not be in the correct format that the user agent (UA) accepts.

Website — optional; should be in the format of a url

```
<li>
  <label for="website">Personal Website: </label>
  <input id="website" name="website" type="url">
</li>
```

As we reviewed previously, the browser's implementation of url validation is poor. As long as it finds a : followed by one or more letters, the value passes validation. Hopefully, this will be improved in the coming months and years.

Bio – optional; should allow the user to write a few paragraphs about himself

This time, we'll want to use the more appropriate **textarea** tag.

```
<li>
  <label for="bio"> Bio: </label>
  <textarea id="bio" name="bio" placeholder="I am..."> ▶
    </textarea>
</li>
```

And with that last snippet, we've completed the markup for our form. It's not pretty (yet), but what we have here is solid and meaningful HTML5 markup.

```
<form method="post" action="">
  <fieldset>
    <legend> Personal Information: </legend>
    <ul>
      <li>
        <label for="name"> Name: </label>
        <input id="name" name="name" type="text" required>
      </li>
      <li>
        <label for="email">Email address: </label>
        <input id="email" name="email" type="email" required> ▶
      </li>
      <li>
        <label for="phone">Phone: </label>
        <input id="phone" name="phone" type="tel" pattern=▶
          "\d{3}-?\d{3}-?\d{4}" placeholder="555-555-5555">
```

```
</li>
<li>
    <label for="age">Age: </label>
    <input id="age" name="age" type="number" min="1" max="120">
</li>
<li>
    <label for="dob">Date of Birth: </label>
    <input id="dob" name="dob" type="date" required>
</li>
<li>
    <label for="website">Personal Website: </label>
    <input id="website" name="website" type="url">
</li>
<li>
    <label for="bio"> Bio: </label>
    <textarea id="bio" name="bio" placeholder="I am..."></textarea>
</li>
<li>
    <input type="submit">
</li>
</ul>
</fieldset>
</form>
```

Personal Information:

- Personal Website:
- Name:
- Email address: ! Please fill out this field.
- Phone:
- Age:
- Date of Birth:
- Bio:
-

Bonus—CSS

If you'd like to take a break from HTML5, let's style our form and make it pretty. We'll begin by zeroing out the margins, and setting some default widths.

```
body {  
    font-family: sans-serif;  
    width: 80%;  
    margin: 40px auto 0;  
}  
/* Quick reset */  
ul, li, input {  
    margin: 0;  
    padding: 0;  
}  
li {  
    list-style: none;  
    margin-bottom: 30px;  
}
```

The screenshot shows a web page with a form enclosed in a light gray border. The form contains the following elements:

- A label "Personal Information:" followed by a horizontal line.
- A label "Personal Website:" followed by an empty input field.
- A label "Name:" followed by an empty input field.
- A label "Email address:" followed by an empty input field.
- A label "Phone:" followed by an input field containing "555-555-5555".
- A label "Age:" followed by an input field containing "18" with a dropdown arrow.
- A label "Date of Birth:" followed by an input field containing "1990-01-01" with a dropdown arrow.
- A label "Bio:" followed by a text area containing "I am...".
- A "Submit" button at the bottom left.

Next, we'll take care of the base styling for the `fieldset` and `form`.

```
form { color: #666; }  
fieldset {  
    padding: 40px 80px;  
    border: 1px solid #e3e3e3;  
}  
legend { font-weight: bold; }
```

Personal Information:

Personal Website:

Name:

Email address:

Phone: 555-555-5555

Age: 8

Date of Birth: 8/8/88

Bio: I am...

Personal Information:

Personal Website:

Name:

Email address:

Phone: 555-555-5555

Age: 8

Date of Birth: 8/8/88

Bio: I am...

(left) Setting fieldset and form styling, (right) setting label styling.

Labels

Next, we should ensure that the **label** appears to be clickable. We'll also set a **display** and **color**.

```
label {  
    width: 18%;  
    display: inline-block;  
    font-weight: bold;  
    color: #000;  
    cursor: pointer;  
}
```

Inputs

Okay, those inputs are still looking pretty sloppy. Let's get that fixed.

```
input, textarea {  
    width: 81%;  
    max-width: 500px;  
    background: white;  
    color: #666;  
    line-height: 35px;  
    height: 35px;  
    padding: 0 20px;  
    border: 1px solid #e3e3e3;  
    -webkit-border-radius: 25px;  
    -moz-border-radius: 25px;  
    border-radius: 25px;  
    cursor: pointer;  
    position: relative;  
    -webkit-box-sizing: border-box;  
    -moz-box-sizing: border-box;  
    box-sizing: border-box;  
}  
  
input:focus, input:hover {  
    border-color: black;  
}  
  
textarea {  
    height: 300px;  
    vertical-align: top;  
}
```

Personal Information:

Personal Website:

Name:

Email address:

Personal Information:

Personal Website:

Name:

Email address:

Phone:

Age:

Date of Birth:

Bio:



Improving the input styling.

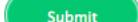
The reason why we're using percentages here is because, this way, the **form** is 100% flexible. We're also using CSS3 rounded corners to make the **inputs** pop out just a bit more.

Buttons

We certainly don't want that **submit** button to be so large.

```
input[type=submit] {  
    width: 100px;  
    padding: 0;  
    background: #12C977;  
    color: white;  
    -webkit-box-shadow: inset 0 0 0 1px #0EE886, 0 0 0 1px ▷  
        #11BF71;  
    -moz-box-shadow: inset 0 0 0 1px #0EE886, 0 0 0 1px ▷  
        #11BF71;  
    box-shadow: inset 0 0 0 1px #0EE886, 0 0 0 1px #11BF71;  
}
```

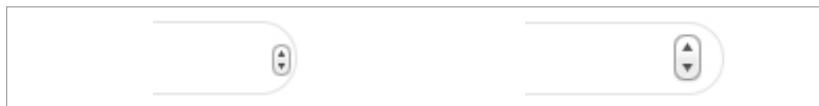
Above, we're using nifty CSS3 **box-shadows** to add more depth to the button.



Ticker

Did you notice how the number ticker on the *Age* and *Date of Birth* **input** is far too close to the edge? We can target the ticker by using a CSS extension:

```
input::-webkit-inner-spin-button {  
    padding-right: 10px;  
}
```



The ticker (left) before and (right) after.

It's a bit awkward isn't it? Nonetheless, at least we do have a way to style these seemingly invisible widgets.

Media Queries

There's one small issue. If the browser window is too narrow, the form begins to look a bit cramped. CSS media queries can be used to **listen** for when the window becomes too narrow, and restyle accordingly.

```
@media screen and (max-width: 750px) {  
    label {  
        margin-bottom: 10px;  
        width: auto;  
        display: block;  
    }  
    input, textarea {  
        width: 100%;  
    }  
}
```

The figure consists of two side-by-side screenshots of a 'Personal Information' form. Both screenshots show the same fields: Personal Website, Name, Email address, Phone, Age, and Date of Birth. In the left screenshot (cramped layout), the labels are very narrow and positioned directly above their respective inputs. In the right screenshot (restyled layout), the labels are wider and centered above the inputs, providing more space between them. The inputs themselves are also wider in the right version.

Using media queries to adjust label and input widths from cramped (left) to fill the space (right).

Much better; aren't media queries great? In the snippet above, we're specifying that, if the `width` of the window is `750px` or less, we should then use the styling within.

Summary

You've just taken a whirlwind tour of the new form features in HTML5. In the next chapter, we'll dig into wonderful native media!

7

The Essentials of Feature Detection

While, ultimately, I will recommend that you use a tool, called [Modernizr](#) to perform feature tests, it's certainly important to understand how it's done manually. Abstractions are fantastic – just as long as you have a modest idea of what's going on behind the scenes.

So, let's not waste any more time than is necessary.

Input Types

So you want to determine whether the browser recognizes the new HTML5 `input` types?

When we attempt to apply an input type that the browser does not recognize, it will always fall back to a type of 'text'. This can be used to our advantage when testing.

```
var input = document.createElement('input');
input.setAttribute('type', 'color');
if ( input.type === 'color' ) {
  // browser supports this input type
}
```

If you'd prefer to abstract this code away to a function...

```
function supportsInputType(type, cb) {
  var input = document.createElement('input');
  input.setAttribute('type', type);
  return input.type === type;
}
```

```
if ( supportsInputType('url') ) {  
    // browser recognizes this type  
}
```



[View the `input` type demo.](#)

Input Attributes

We use a slightly different technique, when determining whether a browser recognizes an input *attribute*.

```
if ( 'autocomplete' in document.createElement('input') ) {  
    // browser recognizes the autocomplete attribute  
}
```

Like most things in JavaScript, `input` elements are *objects*. These objects will contain a list of all the properties that the browser supports. So this code simply determines if a given *property* exists in the object. Simple, really. It's the exact same thing as:

```
var name = {  
    first: 'Pam',  
    last: 'Halbert'  
};  
  
if ( 'first' in name ) { ... }
```

Again, we can place this code within a *function*.

```
function supportsInputAttribute(attr) {  
    return attr in document.createElement('input');  
}  
  
if ( supportsInputAttribute('placeholder') ) {  
    // browser recognizes the placeholder attribute  
}
```

Or, if you plan to use this function multiple times, you can use a closure to keep from creating the test `input` repeatedly.

```
var supportsInputAttribute = (function() {  
    var input = document.createElement('input');  
    return function(attr) {  
        return attr in input;  
    };  
})();
```



[View the `attributes` demo.](#)

Elements

Tests for element support need to be handled on a case-by-case basis. But still, we're simply creating the desired element, and determining whether it recognizes a particular *property* or *method*. If it does, then we can deduce that the browser **must** support the element. Yep — I just used the word, “deduce.”

canvas

If we need to test for `canvas` support:

```
if ( !document.createElement('canvas').getContext ) {  
    // Get your canvas on!  
}
```

Or, as a *function*:

ROCK*

TIP

The double negative (!!) allows us to cast a value to a boolean. “Hello” will be turned into true, while an empty string (' ') will be cast to false. It’s a helpful trick!



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

audio and video

Nearly all modern browsers support native **audio** and **video** (excluding IE8). Nonetheless, we should still provide the necessary fallback for older browsers. We can test for media support by determining whether the **canPlayType** method exists on the media object.

```
if ( !document.createElement('audio').canPlayType ) {  
    // Turn my headphones up!  
}
```

Or:

```
var supportsMedia = (function() {  
    return !( 'canPlayType' in document.createElement('video') );  
}());  
  
if ( supportsMedia ) { ... }
```

Note that this code will work for both **audio** and **video**. Simply substitute accordingly. Because the APIs are nearly identical, we can rest assured that the browsers which recognize the **audio** tag will also recognize **video**.

Local Storage

Testing for local storage support is a simple one — though it's admittedly a bit more complicated than we would like, due to the fact that some versions of IE will throw an error if we don't use a **try** block to catch any potential errors.

```
function supportsLocalStorage() {  
    try {  
        return 'localStorage' in window && window.localStorage >  
            !== null;  
    } catch(e) {  
        return false;  
    }  
}
```

This code determines whether `localStorage` exists on the `window` object. If any errors occur in the process, it instantly returns `false`.

Various APIs

Testing for the other new APIs is more or less the same process. Let's knock these out.

Geolocation

```
if ( !navigator.geolocation ) alert('Track that sucka');
```

Offline

```
if ( !!window.applicationCache ) alert('Get on that plane. >  
It don't matta!');
```

Web Workers

```
if ( !!window.Worker ) alert ('ahh yeah');
```

History

```
if ( window.history && history.pushState ) alert('AJAX      >  
proof. Go ahead; click Back.'');
```

File

```
if ( window.File && window.FileReader >
    && window.Blob ) {
  // browser provides support for all File APIs
}
```

Drag and Drop

```
var supportsDragAndDrop = (function() {
  var el = document.createElement('div');
  return ('ondragstart' in el && 'ondrop' in el);
})();

// usage
if ( supportsDragAndDrop ) {
  // ahh yeah
}
```

Still Use Modernizr

If all feature tests were this easy, there wouldn't be much need for Modernizr. Unfortunately, things quickly become a bit tricky, once we factor in all of the various browsers, both desktop and mobile-based.

Now that you have a basic understanding of how to detect these new elements, inputs, and APIs, file them away in your brain. In the next section, I'll show you how to use the excellent Modernizr to automate this process.

Automated Detection with Modernizr

Up to this point, we've taken a **best guess** approach to feature detection. How do we detect whether a browser supports the `placeholder` attribute on `inputs`? Create the element, and verify that the `placeholder` property exists in this object.

```
if ( 'placeholder' in document.createElement('input') ) {  
    // browser supports this attribute  
}
```

But this is an easy one. What about local storage support? Using the same methodology, we might do:

```
if ( localStorage ) {  
    // browser supports storage  
}
```

And this will work in some browsers! That's the dilemma. Have you tested your code in all applicable browsers? Will this code work in, say, Internet Explorer 7? Will an error be thrown? The answer, unfortunately, is yes. Enter [Modernizr](#).

The Basic Idea

Modernizr's entire purpose for existing is to handle the process of HTML5 and CSS3 feature detection for you. It performs countless (and deep) cavity searches in the browser to ensure that its detection results are as accurate as possible. If asked to describe Modernizr's objective in a few bullet points, one might come up with:

1. Perform dozens of tests to determine browser support for the new CSS3 and HTML5 features and APIs.

2. Apply classes to the `html` element, based upon the results of these tests.
3. Provide a simple API to programmatically access the test results: **Modernizr**
4. Integrate [yepnope](#) to provide an easy-to-use asynchronous script loader, which is handy for referencing polyfills without creating extra bloat and HTTP requests for modern browsers.

A Common Misconception

The Modernizr team will surely admit that the name of the library is somewhat misleading. You wouldn't be remiss to assume that Modernizr somehow manages to **upgrade** older browsers to HTML5 and CSS3. Take an old browser, like Internet Explorer 7, and make it **modern!** Alas, it's not quite as magical as that. Instead, Modernizr takes the tedious task of performing feature detection out of your hands. This means that you can focus less on browser quirks, and more on getting the job done. As the team puts it:

“ *The name Modernizr actually stems from the goal of modernizing our development practices...* **”**

Consider the **placeholder** example from the beginning of the chapter.

```
if ( 'placeholder' in document.createElement('input') ) {  
    // browser supports this attribute  
}
```

Once we [download Modernizr](#), we can test for **placeholder** support, like so:

```
if ( Modernizr.input.placeholder ) {  
    // browser supports this attribute  
}
```

If we take a look at [Modernizr's source code](#) (something you should always do and learn from), we'll find:

```
Modernizr['input'] = (function( props ) {  
    for ( var i = 0, len = props.length; i < len; i++ ) {  
        attrs[ props[i] ] = !(props[i] in inputElem);  
    }  
    return attrs;  
})('autocomplete autofocus list placeholder max min'      ▷  
    'multiple pattern required step'.split(' '));
```

This bit of code filters through all of the various `input` attributes, and determines whether the browser recognizes each. This is the key line:

```
attrs[ props[i] ] = !(props[i] in inputElem);
```

If we substitute the `placeholder` term where appropriate, we get something a bit more readable:

```
attrs[ 'placeholder' ] = !!( 'placeholder' in inputElem );
```

See? This is essentially the same as our own test. We can reference this test with `Modernizr.input.ATTRIBUTE_NAME`. Pretty nifty, ay? And while testing for `placeholder` support is relatively painless, the same isn't necessarily true for other features. False positives, mobile issues, and older browsers can often throw wrenches in what would otherwise be a simple test. That's why Modernizr is so helpful. It does the work for us.

What if want to determine whether the browser recognizes `email` input types?

```
if ( Modernizr.inputtypes && Modernizr.inputtypes.email ) {  
    // browser recognizes email inputs  
}
```

View Source

```
// input features and input types go directly onto the ret object, bypassing the tests loop.  
// Hold this guy to execute in a moment.  
function webforms() {  
    // Run through HTML5's new input attributes to see if the UA understands any.  
    // We're using f which is the <input> element created early on  
    // Mike Taylr has created a comprehensive resource for testing these attributes  
    // when applied to all input types:  
    // http://miketaylr.com/code/input-type-attr.html  
    // spec: http://www.whatwg.org/specs/web-apps/current-work/multipage/the-input-element.html#input  
    // nary  
  
    // Only input placeholder is tested while textarea's placeholder is not.  
    // Currently Safari 4 and Opera 11 have support only for the input placeholder  
    // Both tests are available in feature-detects/forms-placeholder.js  
    Modernizr['input'] = (function( props ) {  
        for ( var i = 0, len = props.length; i < len; i++ ) {  
            attrs[ props[i] ] = !! (props[i] in inputElem);  
        }  
        return attrs;  
    })('autocomplete autofocus list placeholder max min multiple pattern required step'.split(' '));  
  
    // Run through HTML5's new input types to see if the UA understands any.  
    // This is put behind the tests runloop because it doesn't return a  
    // true/false like all the other tests; instead, it returns an object  
    // containing each input type with its corresponding true/false value  
  
    // Big thanks to @miketaylr for the html5 forms expertise. http://miketaylr.com/
```

Viewing the source code for popular JavaScript libraries is a fantastic way to learn how to properly comment your code. It makes your code significantly more readable, and gives you a place to provide *hat-tips*.

Getting Started with Modernizr

Okay, you're sold on the Modernizr library; the first step is to download it. You have two choices:

1. Reference the development version: <http://www.modernizr.com/downloads/modernizr-latest.js>.
2. Use the [build tool](#) to create a customized version of Modernizr. The advantage to this method (which you should use for production) is that you can pick and choose exactly which tests you require. There's also a variety

of community add-ons and extras available exclusively through the build tool.

The screenshot shows the Modernizr build tool's interface. It features three main sections: 'CSS3' (with a 'TOGGLE' button), 'HTML5' (with a 'TOGGLE' button), and 'Misc.' (with a 'TOGGLE' button). Each section contains a list of browser features. Below these is an 'Extra' section with several checkboxes. At the bottom left is a 'Community add-ons' section with a 'GENERATE!' button. On the right side, there is a large, empty text area labeled '// Minified source'.

- CSS3:**
 - @font-face
 - background-size
 - border-image
 - border-radius
 - box-shadow
 - Flexible Box Model (flexbox)
 - hsla()
 - multiple backgrounds
 - opacity
 - rgba()
 - text-shadow
 - CSS Animations
 - CSS Columns
 - CSS Generated Content
 - CSS Gradients
 - CSS Reflections
 - CSS 2D Transforms
 - CSS 3D Transforms
 - CSS Transitions
- HTML5:**
 - applicationCache
 - Canvas
 - Canvas Text
 - Drag 'n Drop
 - hashchange
 - History (pushState)
 - HTML5 Audio
 - HTML5 Video
 - IndexedDB
 - Input Attributes

Note: does not add classes
 - Input Types

Note: does not add classes
 - localStorage
 - postMessage
 - sessionStorage
 - Web Sockets
 - Web SQL Database
 - Web Workers
- Misc.:**
 - Geolocation API
 - Inline SVG
 - SML
 - SVG
 - SVG clip paths
 - Touch Events
 - WebGL
- Extra:**
 - HTML5 Shim/IEP
 - Modernizr.load ([vepropo.js](#))
 - MQ Polyfill ([respond.js](#))
 - Media Queries
 - Add CSS Classes

Community add-ons

GENERATE!

// Minified source

At this stage, while we're still in the process of toying around with Modernizr, let's use the [development version](#). Reference the script within the **head** of your document.

```
<!doctype html public 'the Modernizr chapter, yo'>
<html>
<head>
  <meta charset="utf-8">
  <title>Modernizr Testing</title>
  <script src="http://www.modernizr.com/downloads/
    modernizr-latest.js"></script>
</head>
```

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

Why Not the Bottom?

You've likely read that it's a best practice, when possible, to place your scripts at the bottom of the page, before the closing `</body>` tag. This is absolutely true. When you do so, your pages will appear to load more quickly. However, Modernizr **must** be placed in the **head**, after your stylesheet references. There are a couple reasons why this is necessary:

1. A simple script that makes Internet Explorer 8 and below recognize and style the new HTML5 elements must be placed in the **head** of your document to work properly. Note that this script is identical to Remy Sharp's [html5shim](#).
2. Visitors to your website may see a flash of unstyled content if the script is not added to the **head**.

HTML Classes

By referencing this single script, Modernizr will instantly run dozens of tests, and apply classes to the `html` tag.

```
<!DOCTYPE html PUBLIC "the modernizr chapter, yo!">  
▼<html class=" js flexbox canvas canvastext webgl no-touch geolocation postmessage websqldatabase indexeddb hashchange history draganddrop websockets rgba hsla multiplebgs backgroundsize borderimage borderradius boxshadow textshadow opacity cssanimations csscolumns cssgradients csstransitions csstransforms csstransforms3d fontface generatedcontent video audio localstorage sessionstorage webworkers applicationcache svg inlinesvg smil svgclippaths">  
▶<head>...</head>  
  <body>...</body>  
</html>
```

These classes are unique to the browser and what it supports — in my case, Google Chrome 16. What's helpful is that this now gives us the means to style our content, based upon whether it supports a particular feature. I can now use the CSS3 flexible box model in browsers which support it.

The flexible box model determines the manner in which boxes are distributed inside other boxes. It also automatically determines spacing.

```
.flexbox .container {  
    /* This browser supports the flexible box model */  
}
```

Alternatively, if the browser does not support **flexbox**, then Modernizr will instead apply a class of **no-flexbox**.

```
.no-flexbox .container {  
    /* ehh - I better use floats. */  
}
```

Remember: the class names that Modernizr applies to your **html** tag are entirely dependent upon which CSS3 and HTML5 features the browser supports. That's why it's so powerful!

no-js

If you wish, you can also apply a class name of **no-js** to the **html** tag.

```
<html class="no-js">
```

By using simple CSS classes like this, we can easily determine whether or not JavaScript is enabled in the user's browser. If it is, Modernizr will replace **no-js** with **js**.

Perhaps you have a widget on your website that depends on JavaScript to function correctly. It might be smart to **hide** this content if the user has disabled JavaScript. This way, it won't be confusing when, say, the *prev* and *next* buttons don't seem to work as expected.

```
.no-js .widget {  
    display: none;  
}
```

This styling will only take effect on the condition that JavaScript is not enabled. If it is, the `no-js` class won't be available.

Modernizr.load

[Yepnope](#), the technology behind Modernizr's `load` tool, is an asynchronous and conditional resource loader. Okay, that's confusing. What the heck does that mean? Let's break some of these terms down, piece by piece.

Asynchronous

Yepnope will load your assets in such a way so that the rest of your document does not have to linger while the script loads.

Think of a factory consisting of a single employee. This employee's only job is to load your stylesheets and scripts (I know this is reaching, but stick with me). The problem is that this factory worker can only do one thing at a time. While he's loading script number one, the rest of your document sits and spins.

Next, he loads resource number two, and the cycle repeats. Once he's finished loading these resources, only then does the visitor see the page content. This is why it's often recommended that you place as many scripts as possible at the bottom of the page. Though the page still takes the same length of time to load, it **appears** to the visitor as if it's loading more quickly.

ROCK*

TIP

Keep in mind that we're dealing with a matter of milliseconds, however, it's a proven fact that even a 200 millisecond lag can have significant effects on traffic and sales.



While modern browsers have begun to download scripts in parallel, they're not yet perfect, and, besides, we still have older browsers to worry about.

Wouldn't it be great if we could sew extra hands onto this factory worker, so that each can load a resource individually? That way, with all of these hands working in **parallel**, the worker can be far more efficient at his job.

Star Wars geeks: just think of General Grievous's four light sabers. That's the basic idea. Each hand can work independently (or asynchronously), and, as a result, get more work done (or slice more heads). </end-terrible-analogy>

Conditional

This is where resource loaders shine; we can specify that a *script* or *stylesheet* should only be loaded on a specific condition.

Resource loaders are tailor-made for polyfills. For instance, you've already learned about using the `date` input type. As discussed in that chapter, older browsers must be considered. If you expect a date in a certain format, providing IE6 users with the fallback textbox is a terrible choice. Instead, we can use a polyfill, such as [jQuery UI's date picker plugin](#). If the browser does not support the `date` input type, only on that condition should it load this polyfill or fallback script.

Resource

Don't let this confuse you. It's merely a way to reference both *stylesheets* and *scripts*. Many loaders exclusively focus on *scripts*, hence the name, "script loader." Yepnope handles both.

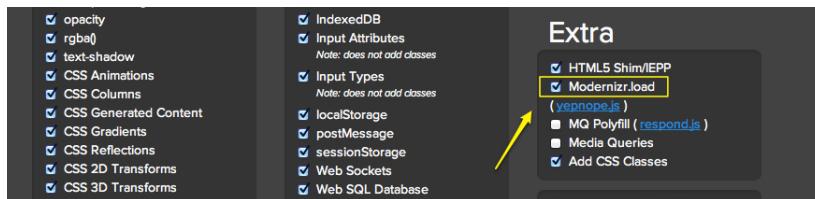
What does yepnope have to do with Modernizr?

“Currently yepnope is being included in special builds of Modernizr, because it's such a good companion to a feature testing library. The output of Modernizr

is a fantastic input to `yepnope`. `Yepnope` is currently included unmodified from its original state, but is also aliased as `Modernizr.load` — you can use the two interchangeably, though we suggest that you stick to one or the other.  — yepnopejs.com

Usage

Before proceeding, you must create a custom Modernizr script by using the [Modernizr build tool](#), and specify that it should include the `Modernizr.load` functionality within the *Extra* section.



The simplest way to use `yepnope`, or `Modernizr.load`, is to pass it a *string*, like this:

```
<!doctype html>
<html class="no-js">
<head>
  <meta charset="utf-8">
  <title>Modernizr Testing</title>
  <!-- Place after stylesheets -->
  <script src="modernizr.js"></script>
  <script>
    Modernizr.load('jquery-local.js');
  </script>
```

This will, as soon into the future as possible, load the local jQuery file. If you're new to the idea of a resource loader, you might be surprised to find that the following won't work the way you expect.

```
<script>
  Modernizr.load('jquery-local.js');
  $.get(...);
</script>
```

Even though we've seemingly pulled in the jQuery library, we don't yet have access to the `jQuery` function. In fact:

```
<script>
  Modernizr.load('jquery-local.js');
  console.log(typeof jQuery); // undefined
</script>
```

What's the deal? We don't have access to jQuery because the script has not yet finished downloading! Though it may initially be confusing, this is exactly why it's so great! `Modernizr.load` does not wait for the resource to finish loading before it moves on. This is how our pages can load faster.

This then raises the question, "When do I reference jQuery?" We can use the `callback` method, which will be called as soon as the script has fully loaded.

```
Modernizr.load({
  load: 'jquery-local.js',
  callback: function(url, result, key) {
    console.log(typeof jQuery); // function
  }
});
```

This time, rather than passing a *string* to `yepnope`, we instead pass an *object*.

```
Modernizr.load('jquery-local.js');
```

... is identical to:

```
Modernizr.load({  
    load: 'jquery-local.js'  
});
```

Tests

We haven't yet taken advantage of the **conditional** aspect of **Modernizr.load**. We can do so, via the **test** property. Let's imagine that we want to use the HTML5 date picker, and a [jQuery UI fallback](#) for older browsers. Here's how it's done, son.

```
Modernizr.load({  
    test: Modernizr.inputtypes && Modernizr.inputtypes.date,  
    nope: [  
        'ui/js/jquery-ui-custom.min.js',  
        'ui/css/ui-lightness/jquery-ui-custom.css',  
        'datepicker.js' // your script that executes the  
        // necessary function  
    ]  
});
```

First, we use **test**, passing in a *boolean*, to determine what we should and should not do. In this case, we're determining whether the browser recognizes **date** inputs. Dependent upon whether that test returns **true** or **false**, we can use the **yep** and **nope** methods, accordingly. We don't really need to do anything specific if the browser has support; but, if it doesn't, we need to load our polyfill.

Go ahead and try it out in your browser. Inspect the generated HTML in both Opera and Firefox (8 or lower). In Opera, the test returns **true**, in which case nothing happens. However, in Firefox 8, the test returns **false**. At this point, the **nope** callback is called, and the array of asset paths are loaded.

Unfortunately, at the time of this writing, Webkit will incorrectly return **false** for the **date** test, even though it does recognize it

(while providing a bland date clicker). This is less a bug, and more an issue with the UI in Webkit not being in place yet. Once it is, the Modernizr test should pass.

Why the Extra Work?

You might be thinking to yourself, why do all this extra work, when we can instead load the jQuery UI files regardless of whether the browser supports the feature? That way, we don't need to bother with these sorts of tests.

This is terrible thinking! Don't be lazy. What you're suggesting is:

1. Who cares if we use up more HTTP requests and bandwidth in browsers that don't require these polyfills?
2. I don't care how quickly the assets take to load, as long as they... do.

We're web developers, fool; our job is to provide the best possible experience for all devices. If we choose to load multiple unnecessary assets, we are no longer providing the best possible experience. Don't be lazy; use `Modernizr.load`.

8

Finally... Native Media

Browser Support



Ahh... we come to HTML5 audio and video — one of the most flashy, press-worthy new additions to HTML. We now have the ability to provide native audio and video playback in the browser. No longer must we rely on buggy, browser-crashing third party tools, like Flash. Right? Well, sort of, but I'm getting ahead of myself.

Back in the Day

Not too long ago, if we wanted to embed a video into a website or blog posting, we had to resort to using a nasty block of HTML, much like the following:

```
<object width="600" height="400">
  <param name="movie" value="URL">
  <param name="allowFullScreen" value="true">
  <param name="allowScriptAccess" value="always">
  <embed src="URL" type="application/x-shockwave-flash" >
    allowscriptaccess="always" allowfullscreen="true" >
    width="600" height="400">
  </embed>
</object>
```

Likely, you never wrote this HTML manually. Most video hosts, such as YouTube or Blip, provide an *Embed* option that you can

copy and paste. Nonetheless, it's a sure-fire way to turn your beautiful and succinct HTML into a big mess.

The worst part, though, is that video support was not provided natively in the browser back then. Instead, the browser would pass the video on to a third-party plugin — usually Flash.

```
type="application/x-shockwave-flash"
```

Do you really think that your mom is updating her Flash plugin when prompted? Mine sure isn't! When combined with potential **browser-crashing** bugginess and accessibility issues, it's fairly clear that third party tools are not the most optimal way to handle this (except for a handful of situations, which I'll detail shortly).

With Flash, you're heavily restricted when it comes to layout and functionality (and cleanliness). On the flipside, by providing video and audio natively in the browser, we're afforded all the power of HTML5, CSS3 effects, JavaScript, and whatever else you can think of!

Note that we will be focusing specifically on the **video** element, due to the fact that the APIs for **video** and **audio** are mostly the same. Besides, video is more fun!

What HTML5 Video is not Appropriate For

Before we move on to usage information, it's important to keep in mind that there are still times when we should remain with third-party tools.

DRM

DRM, or Digital Rights Management, isn't the most popular topic in the HTML5 world. For those unfamiliar with the term, the basic idea is that some developers and companies need a way to limit a video's availability to only the users who meet some certain level of criteria, for example "This video is only available to subscribers." In its present state, this is not possible (without some trickery). Nonetheless, until it's provided in some form, much of the video playback on the web will continue to be presented via Flash or Silverlight.

In September, 2010, John Foliot opened discussions at the W3C on the subject, noting:

“If <video> is to see widespread adoption by commercial content providers, then there must be a means to support some form of native DRM if it is to compete with Flash.”

In response, Ian Hickson closed the bug report, with a simple response: "DRM is evil." Needless to say, this angered much of the community, at which point the thread was re-opened (along with [his more in depth response](#)).

While it's easy to quickly jump to one side of the debate, this is not a black and white issue. Further, it's not a simple one either; how would it be possible for HTML5 to provide DRM? We can't do so with images; why would there be a difference with native video? As this author sees it, providing documentation for DRM is realistically outside the scope of HTML, in the same way that codecs are. HTML5 neither endorses nor restricts its usage. Instead, the role of DRM should be assigned to the codec, itself.

Within John Foliot's bug thread, [Lachlan Hunt](#) made a handful of excellent notes:

“[...] Requiring any form of DRM scheme in HTML5 would more than likely not succeed because:

- *The decryption algorithm would have to be public, and any implementation would need access to the keys, defeating the delusion about stopping piracy.*
- *It would make implementers subject to the licensing fees of the DRM provider.*
- *It would expose browser vendors to legal nightmare that is DMCA compliance, and similar draconian anti-circumvention laws around the world dealing with TPMs*
- *It would make it impossible for free and open source implementations to implement the DRM scheme.* **”**

However, on the other hand, there are those who strongly feel that the issue absolutely must be addressed. Whether we agree with it or not, DRM does exist on the web. The only question is: how can we deal with it, and does it make sense for an open standard to be the responsible party? I guess we'll find out soon enough.

Ads

Some companies and websites depend on advertising revenue. For example, Nettuts+ hosts some videos on Blip.tv, which attaches advertising before and after the video. This extra income is an excellent way for us to warrant the cost of producing the video tutorial itself. Unfortunately, HTML5 does not provide any method to mimic this sort of functionality. At least for the time being, should you require advertising, you'll need to stick with Flash or Silverlight.

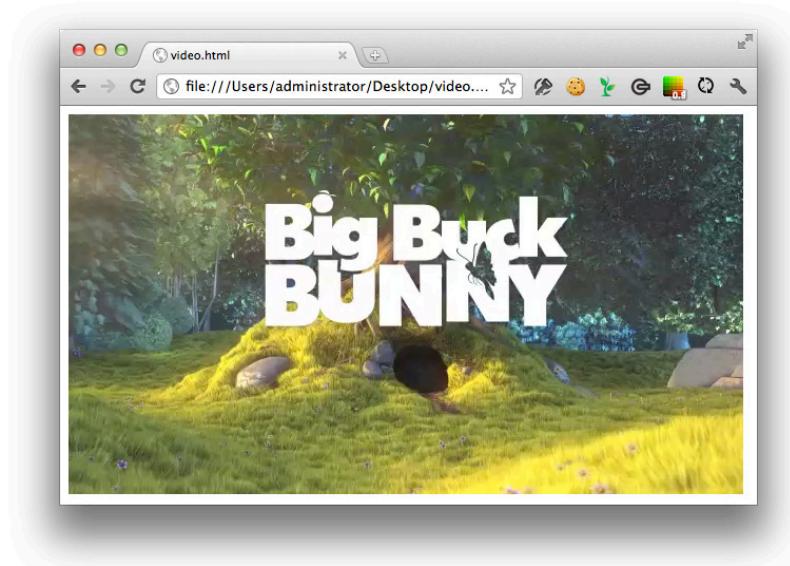
Usage

Remember that huge `object` tag I detailed earlier? It can now be replaced with:

```
<video src="bbb.mp4"></video>
```

Or:

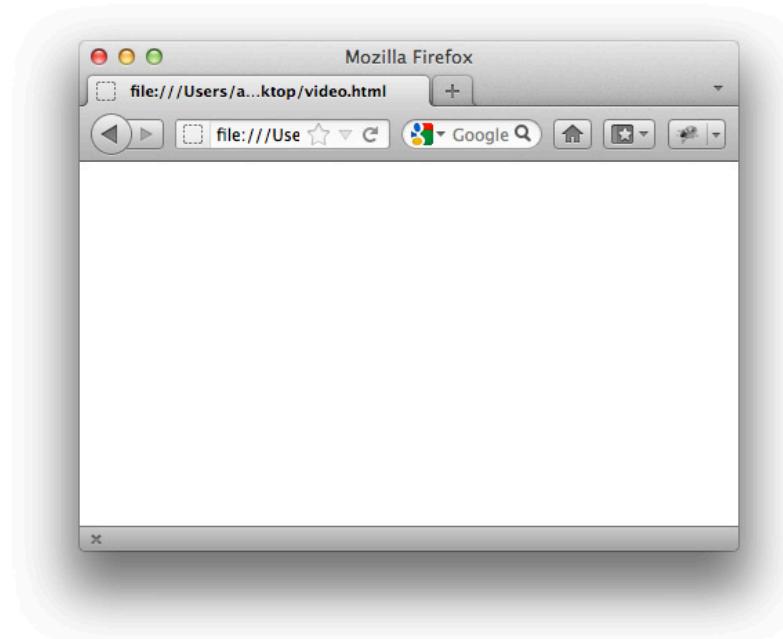
```
<video>
  <source src="bbb.mp4"></source>
</video>
```



The video clip used in this chapter, Big Buck Bunny, is a Creative Commons-licensed “open movie,” that is often used for HTML5 video demonstrations. The entire movie is available in multiple formats at BigBuckBunny.org.

Well... it's not quite that simple. You're about to learn a whole lot about the codec battles. But, for now, switch over to Firefox, and take a look at your beautiful... blank screen. Great.

Firefox does not support the **locked down** mp4 format (more on that later). This means that we're left in a position of having to provide multiple versions of a video to accommodate all the various browsers.

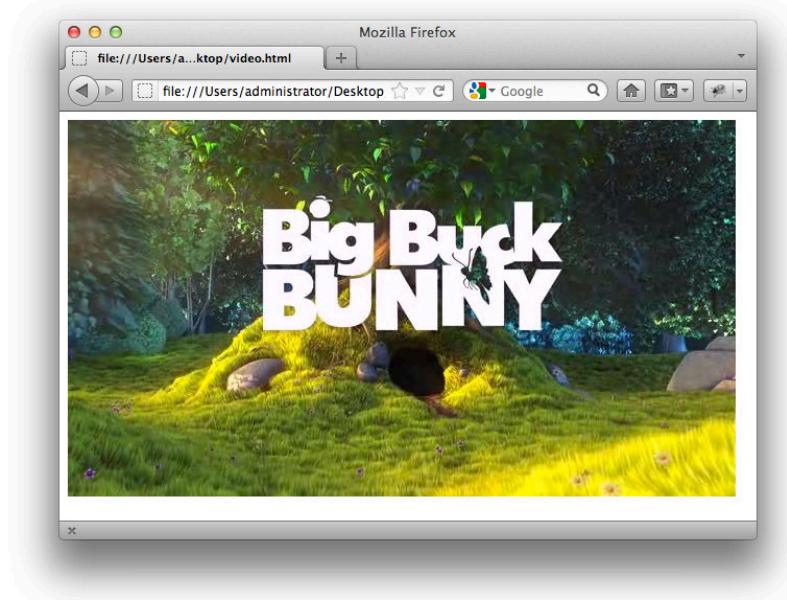


Luckily, by providing multiple **source** tags, we can make each browser happy. In Firefox's case, it will want to see a **.webm** formatted video (among others).

The browser will read each **source** tag in order. If it does not recognize a particular codec, it will move on to the following **source** tag and try again.

```
<video>
  <source src="bbb.mp4">
  <source src="bbb.webm">
</video>
```

There we go; that's better. Note that older versions of the iPad would only read the first **source** element. As such, it's best practice to place the **source** element with the **mp4** version of the video first.



The type Attribute

To prevent the browser from “sniffing” each **source** tag, we can specify a MIME type.

```
<video>
  <source src="bbb.mp4" type="video/mp4">
  <source src="bbb.webm" type="video/webm">
</video>
```

This way, the browser doesn't need to download the meta information for the video to determine if it can be played. However, this only takes us part of the way. The MIME type refers to the container format of the video, which you can almost think of in the same way that you would a zip file. To truly investigate, we need to learn what's inside that container format. The `codecs` parameter can be used to perform this sort of deep cavity inspection.

```
<video controls>
  <source src="video.mp4" type='video/mp4; codecs=
    "avc1.42E01E, mp4a.40.2"'>
  <source src="video.webm" type='video/webm; codecs="vp8,
    vorbis"'>
</video>
```

Because multiple versions of a video must be provided to accommodate all browsers, for now, the shorthand, `<video src="my-video">`, should only be used when assigning the video's source dynamically with JavaScript.

A Text Fallback

What happens in browsers which do not recognize the new `video` tag? As you learned with the previous Firefox example, they'll do... absolutely nothing. We can provide a text fallback for these situations.

```
<h1> Watch This Video </h1>
<video>
  <source src="bbb.mp4">
  <source src="bbb.ogv">
  <a href="bbb.mp4">Download the video.</a>
</video>
```

Watch This Video

[Download the video.](#)

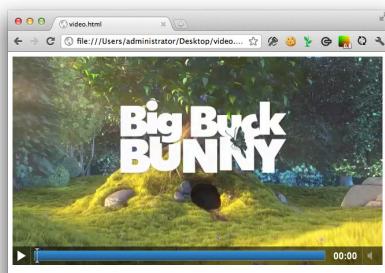
This download link will only display in browsers which do not recognize the `video` tag. This is an important distinction to make. It's perfectly understandable for one to assume that the text will display on the condition that the video, itself, is not understood (e.g. when only an MP4 version is provided, and the user is in Firefox). However, this is not the case; if the `video` tag is understood by the browser, the fallback text will never render.

Before the end of the chapter, we'll also review Flash video fallbacks.

controls

If working along, you'll find that in its current state, our video might as well be an image. Let's add the standard controls by applying the `controls` attribute to the `video` tag.

```
<video controls>
  <source src="bbb.mp4">
  <source src="bbb.ogv">
  <a href="bbb.mp4">Download the video.</a>
</video>
```



The `controls` attribute as rendered in Chrome (left) and Safari (right).



The `controls` attribute as rendered in Firefox (left) and Opera (right).

Note that, in Firefox, the controls will only display when hovering over the video.

On that note, as a user, I'd love to learn why the default setup for the `video` element is without controls. Doesn't that seem counter-intuitive? There are certainly cases when it makes sense to play the video dynamically and omit the controls, however, in this author's opinion, the controls should display by default.

autoplay

Now what if we want the video to play immediately when the page loads (95% of the time not advised)? The `autoplay` attribute will do the trick.

```
<video controls autoplay>
  <source src="bbb.mp4">
  <source src="bbb.ogv">
  <a href="bbb.mp4">Download the video.</a>
</video>
```

A Note on Accessibility

At the time of this writing, accessibility concerns are considerable. For example, without the `controls` attribute present, no browser

will respond to keyboard commands, such as using the spacebar to play the video, when focused.

With this attribute present:

- **Chrome 15** – No response to keyboard navigation.
- **Safari 5** – No response to keyboard navigation.
- **Firefox 9** – When focused, responds to the space bar, and arrow keys for navigation and audio levels
- **Opera 11** – Offers optimal accessibility; each control on the video player can be accessed via the *tab* key.

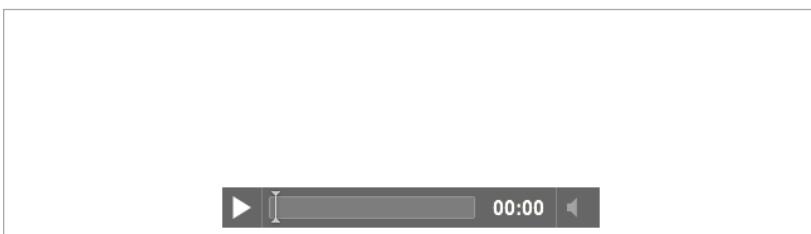
poster

By default, the first frame of the video will be used as the initial display for the video. The **poster** attribute can be used to specify a custom image.

```
<video controls poster="poster.jpg">
  <source src="bbb.mp4">
  <source src="bbb.ogv">
  <a href="bbb.mp4">Download the video.</a>
</video>
```

Additionally, it can be used in situations when the video data (**preload=none**, as you'll learn shortly) is not yet available. Without the **poster** attribute, an empty, un-sized video container will be presented.

```
<video preload="none" controls>
```



The `poster` attribute: (above) a JPEG is used for the initial image, (below) when no image is specified, an empty space is reserved above the controls.

However, when provided, the intrinsic width and height of the poster image will be used as the initial dimensions for the video, that is, until the meta information for the video is available.

```
<video preload=none controls poster="poster.jpg">
```



loop

I'm not sure when this particular attribute would be of use, but, nonetheless, a video will automatically loop if the `loop` attribute is specified.

```
<video controls poster="poster.jpg" loop>
  <source src="bbb.mp4">
  <source src="bbb.ogv">
    <a href="bbb.mp4">Download the video.</a>
</video>
```

preload

There will likely be situations when you can bet on the fact that the user will, at some point, view the video on the page. In these cases, it makes sense to begin preloading the video immediately when the page loads. This way, buffering will become less of an issue.

A handful of values are available to the `preload` attribute:

```
<video controls preload>
```

ROCK*

TIP

It only makes sense to enable this option on the condition that the user will most likely view the video at some point. Otherwise, you're only wasting bandwidth!



Setting the attribute alone implies a value of `auto`, and instructs the browser to determine for itself how best to proceed. In most cases, the browser will begin downloading the file, however, the presence of this attribute alone does not guarantee it. For instance, some devices, based upon the internet connection, may choose not to preload the asset.

```
<video controls preload="none">
```

A value of **none** instructs the browser to begin loading the video only when the user activates the controls.

```
<video controls preload="meta">
```

Rather than preloading the video, a value of **meta** will only capture the meta information that is related to the video, such as its length, dimensions, etc.

A Brief Overview of Codecs

It's not glamorous, but you should, nonetheless, have a basic understanding of video codecs.

A codec will take a large number of images or audio samples, and turn them into a compressed video or audio stream, respectively. The quality of each codec can vary considerably. Unfortunately, it does not end there. When audio and video streams are merged, we end up with a specialized container format. For instance, H.264 video and AAC audio merge into the popular MP4 format, or container.

When the HTML5 specification was first drafted, it required every browser to provide support for both the Ogg Vorbis and Ogg Theora codecs, for audio and video playback, respectively.

The web would have been a better (or at least easier) place, had this taken effect, however, a handful of companies objected to the mandate, including Apple. As a result, this section of the spec was removed, leaving the decision of codecs up to each browser maker. Consequently, we're left with somewhat muddy waters:

- **Firefox** – supports Vorbis and Theora, VP8
- **Opera** – supports Vorbis and Theora, VP8

- **Internet Explorer 9** – supports H.264 and MP3
- **Safari** – supports H.264 and MP3
- **Chrome** – supports Theora, H.264, Vorbis, MP3, and also champions the open-source video codec, VP8

So What Do We Do?

As a rule of thumb, don't leave any browser in the dust. If your video is encoded as H.264, you'll make IE, Safari, and Chrome happy; however, Firefox users won't be able to view your video. Don't be a jerk — make sure everybody gets a piece of the cake. Provide both formats.

ROCK*

TIP

VP8 is a top-notch, royalty free, video codec that is rapidly gaining support. At the time of this writing, Firefox, Chrome, Opera, and Android support the format. You'll likely interact with it, via the WebM container (VP8 for video, and Vorbis for audio).



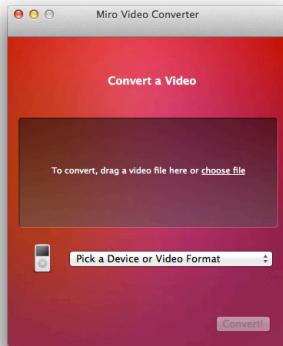
Video Encoding Tools

HandBrake



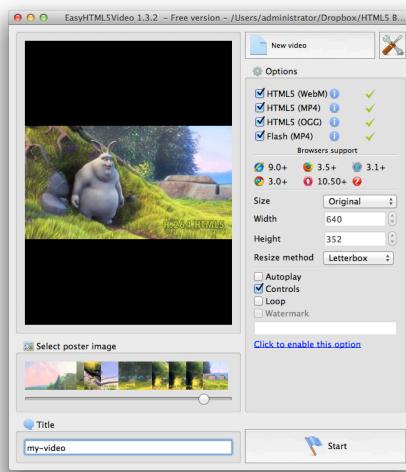
[Handbrake](#) is a popular open-source transcoder, and is available for Mac, Linux, and Windows.

Miro (Windows and Mac)



Miro Video Converter allows you to convert virtually any video to MP4, WebM, and Ogg Theora.

Easy HTML5 Video (Windows and Mac)



My [tool of choice](#) will automatically convert your video to the necessary formats, and will dynamically create the appropriate markup for both the native video versions as well as the Flash fallback.

If It Still Won't Play

If you convert a video to the necessary format, and it still doesn't seem to be recognized, it's possible that your server isn't configured properly. As a result, the video will be served as `text/plain`, rather than something along the lines of `video/ogg`. Apache users will likely need to update their `.htaccess` file, accordingly.

```
AddType video/ogg .ogv  
AddType audio/ogg .oga  
AddType video/mp4 .mp4  
AddType video/webm .webm
```

What About Mom?

So far, we've taken the necessary steps to make all modern browsers happy, while providing a download link for legacy browsers (and your mom). Are you okay with this? In some cases — sure, I am! That said, this method isn't always appropriate.

Should you require video playback for all users, you'll need to provide a Flash fallback. We've already discovered that any content, other than the `source` tags, will be displayed in the event that the browser does not support HTML5 video. With that in mind, we can fallback to our old `embed` code.

```
<video controls poster="poster.jpg" loop>  
  <source src="bbb.mp4">  
  <source src="bbb.ogv">  
  <embed src="URL" type="application/x-shockwave-flash" >  
    allowscriptaccess="always" allowfullscreen="true" >  
    width="600" height="400">  
</video>
```

A common sequence of events, once you have a video in hand, is to first convert it to the necessary video formats (H.264 and

WebM), then upload the video to one of the various video hosts, such as YouTube. This way, no matter what, all visitors will see the video — some of them, via Flash, and the others via native video playback!

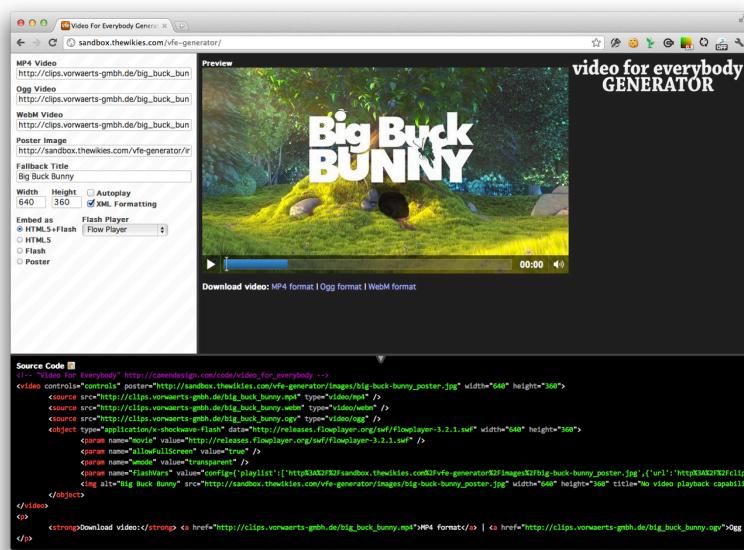
It's not pretty, but, for the time being, "that'll do, pig."

Video for Everybody

This basic technique (HTML5 video with a Flash fallback) was first popularized by "Video for Everybody."

“ Video for Everybody is simply a chunk of HTML code that embeds a video into a website using the HTML5 <video> element, falling back to Flash automatically without the use of JavaScript or browser-sniffing. **”** — [CamenDesign.com](http://camendesign.com)

To automate the process of creating the necessary markup, [Jonathan Neal](#) created a generator that works quite well.



It Doesn't End There

Sure, at this point, we've made all users happy. Well — not quite. What about mobile visitors? Does it make sense for them to download your large, HD video on a tiny device? It sure would be nice if we could target these devices, and provide highly compressed versions of the video. Well we can!

Adopted from CSS3 media queries, we can specify that one version of a video should be loaded on the condition that the device's width meets a certain criteria.

```
<source src="bbb.mp4" media="(min-device-width: 750px)">
<source src="bbb-small.mp4">
<source src="bbb.webm" media="(min-device-width: 750px)">
<source src="bbb-small.webm">
```

So does this mean that, for every video, you must provide a dozen different formats and sizes? Well it depends. Certainly, it couldn't hurt. For instance, if the video is for a preview of, say, your new web app, I'd absolutely recommend that you go to the effort. On the other hand, if you're simply providing a quick video example for your blog, you may elect to forego providing optimized video for mobile users.

Controlling Video with JavaScript

In addition to the built-in controls, we can also control a video's playback with HTML5's new media API. Yes, yes, another new API. Don't worry, it's simple!

Detection

Detecting browser support for both the **video** and **audio** elements is simple. We only need to determine if the **canPlayType** method is available on the media object.

```
if ( 'canPlayType' in document.createElement('video') ) {  
    // Give these people air...  
}  
  
if ( 'canPlayType' in document.createElement('audio') ) {  
    // I'm the one that jaded you...  
}
```

Note that, if the browser recognizes the **video** tag, then it also support audio. The APIs are virtually identical, and all browsers which recognize one, will also recognize the other. This means that our detection script can be simplified to:

```
var supportsMedia = (function() {  
    return !!('canPlayType' in document.createElement('video'));  
})();  
  
if ( supportsMedia ) { ... }
```

Alternatively, if using Modernizr, we can stick with:

```
if ( Modernizr.video ) { ... }  
if ( Modernizr.audio ) { ... }
```

Detecting Codec Support

Our detection duties don't end here. In some instances, it's also helpful to determine whether the browsers recognize a particular codec. This way, we can dynamically load the most appropriate video format. Unfortunately, this isn't quite as straight-forward as we might hope. Luckily, Modernizr provides a nice solution. It uses

the new `canPlayType` method of the media object to determine if a particular codec is recognized. For example, in Chrome 16:

```
video.canPlayType('video/webm; codecs="vp8, vorbis"');
// probably
```

These tests are available, via Modernizr's `.video` test. Alone, `Modernizr.video` will return a boolean indicating the browser's support; however, Modernizr extends the boolean constructor so that we may also verify support for the `ogg`, `h264`, and `webm` formats.

```
Modernizr.video // browser supports video
Modernizr.video.webm // browser supports the WebM format
Modernizr.video.h264 // browser supports the H.264 format
```

Earlier, I noted that the APIs for HTML5 video and audio are identical. As such, it might surprise you to find that Modernizr provides two separate tests: `Modernizr.video` and `Modernizr.audio`. The feature detection, itself, is identical. The only difference stems from the extended boolean that detects support for the various codecs.

```
Modernizr.audio.mp3
Modernizr.audio.ogg
Modernizr.audio.wav
Modernizr.audio.m4a
```

Methods

There's only a handful of new methods available to the media element, whether it be `video` or `audio`.

- `play()` – Begins or continues the referenced video.
- `pause()` – Pauses the video.

- `load()` – Triggers a handful of actions, but, in practice, can be used to fetch a new media resource.
- `canPlayType()` – Determines the likelihood of the browser's capability of interpreting the passed codec.

These methods can be used, like so:

```
// Find the first video tag on the page
var video = document.querySelector('video');

// play the video
video.play();

// pause the video
video.pause();

// Does the browser recognize the ogg format?
video.canPlayType('audio/ogg; codecs="vorbis"');
```

Events

The new methods alone aren't enough; we also need a way to **listen** for when specific events occur, such as when the video begins playing, or has ended.

- **play** – Fires once when the video begins or continues playing.
- **pause** – Fires once when the video is paused. Note that there is no **pause** event.
- **ended** – Fires once when the video has ended.
- **progress** – Fires once when the media has been retrieved, but potentially before it has been decoded.
- **canplay** – Fires when the video has been fully loaded, and is ready to be played.

Attributes

- **duration** – The length of the associated video.
- **currentTime** – The current position of the video. This can be used, for instance, when the video has been paused.
- **autoplay** – A boolean, indicating if the video was set to autoplay.
- **poster** – The path to the provided poster image, or an empty string.
- **paused** – A boolean, indicating if the video is currently paused.
- **ended** – A boolean, indicating if the video has completed.

To retrieve the length, or duration of a video, we use `video.duration`; however, `Nan` will be returned if the meta information for the video has not yet been retrieved. As such, to ensure that the correct value is returned, first listen for when either the meta information or the video has loaded entirely.

```
var video = document.querySelector('video');
video.addEventListener('progress', function() {
  console.log(video.duration); // 60.13968276977539
}, false);
```

Now, we have the tools to hook into the lifecycle of a video. To create a toggling play/pause button, we only need to add a button to the page.

```
<video poster="poster.jpg">
  <source src="bbb.mp4">
  <source src="bbb.ogv">
</video>
<button>Play</button>
```

Wait – not so fast. What if the visitor has JavaScript disabled? Doesn't it make better sense to append the **button** with JavaScript? Of course it does. This time, we'll apply **controls** to the video by default, and then remove them dynamically.

```
<video poster="poster.jpg" controls>
  <source src="bbb.mp4">
  <source src="bbb.ogv">
</video>
```

And then, within our JavaScript:

JavaScript

```
var video = document.querySelector('video'),
    button = document.createElement('button');

// set the button's text
button.innerHTML = 'Play';

// Remove the controls
video.removeAttribute('controls');

// Append the button after the video
document.body.insertBefore(button, video.nextSibling);

// Listen for when the button is clicked.
button.addEventListener('click', function() {
  if (video.ended) video.currentTime = 0;

  // play or pause the video
  video[video.paused ? 'play' : 'pause']();
}, false);

video.addEventListener('play', function() {
  button.innerHTML = 'Pause';
}, false);
```

```
video.addEventListener('pause', function() {
    button.innerHTML = 'Play';
}, false);

video.addEventListener('ended', function() {
    button.innerHTML = 'Play';
}, false);
```

jQuery

```
var $video = $('video'),
    video = $video[0], // convenience
    button = $('Play</button>');

$videoremoveAttr('controls').after(button);

$video
    .on('play', function() {
        button.text('Pause');
    })

    .on('pause ended', function() {
        button.text('Play');
    });

button.on('click', function() {
    if ( video.ended ) video.currentTime = 0;

    // play or pause the video
    video[video.paused ? 'play' : 'pause']();
});
```

Notice how we use the `play`, `pause`, and `ended` event listeners to update the value of the button, accordingly.

Building a Playlist

Let's imagine that you want to provide a simple playlist for users. Perhaps you've separated a video course into multiple chapters,

and now need an easy way to allow the user to load each chapter without performing a full page refresh. We'll review two ways to accomplish this task.

Automatically Changing the Video

First, we'll listen for when each video has completed, and then load and play the next chapter, or video, automatically for the user.

JavaScript

```
var video = document.querySelector('video'),
    playlist = ['chapter-1', 'chapter-2', 'chapter-3'],
    extension = Modernizr.video.webm ? '.webm' : '.mp4',
    i = 1; // chapter-1 is the default video, so we begin      ▶
           with the second item in the array

video.addEventListener('ended', function() {
    !playlist[i] && (i = 0); // if at the end of the array,      ▶
                           reset

    this.src = playlist[i++] + extension; // chapter-1.webm, ▶
                                         or chapter-1.mp4
    this.poster = '';
    this.load(); // load the video

    this.play();
}, false);
```

jQuery

```
var video = $('video').first(),
    playlist = ['chapter-1', 'chapter-2', 'chapter-3'],
    extension = Modernizr.video.webm ? '.webm' : '.mp4',
    i = 1;
```

```
video.on('ended', function() {
    !playlist[i] && (i = 0); // if at the end of the array, >
    reset

    this.src = playlist[i++] + extension; // chapter-1.webm, >
    or chapter-1.mp4
    this.poster = '';
    this.load(); // load the video

    this.play();
});
```

Note that we're taking advantage of Modernizr to detect support for both the `video` element, and the `webm` format.

This snippet uses the `ended` event to detect when each video has completed. Once it has, we assign the next chapter's path to the `video`'s `src` attribute, load it, and begin playing. Note that the script assumes that you have both `chapter-x.webm` and `chapter-x.mp4` versions of the video available.

This process will repeat infinitely, due to:

```
!playlist[i] && (i = 0);
```

This line specifies that, if there is no additional items in the `array` to fetch, we reset `i` back to zero, and wrap around to the first video.

Notice that we're setting the `src` attribute on the `video` tag itself. You'll find that, if you instead do the same to the `source` element, it will have zero effect. To instruct the browser to fetch this newly applied resource, trigger the `load` method: `video.load()`.

Providing a Chapter List

While it's helpful to automatically switch to the next video once the previous one completes, we can do better. Let's create an actual playlist, so that the user can easily switch between lessons.

The Markup

We begin with the markup. This time, we'll need a list of links to serve as the playlist:

```
<video controls preload>
  <source src="chapter-1.m4v">
  <source src="chapter-1.webm">
  <a href="chapter-1.m4v">Download the video.</a>
</video>
<h2>Playlist</h2>
<ul class="playlist">
  <li><a href="chapter-1.m4v" class="playing">First Video ▶
    </a></li>
  <li><a href="chapter-2.m4v">Second Video</a></li>
  <li><a href="chapter-3.m4v">Third Video</a></li>
</ul>
```



Playlist

- [First Video](#)
- [Second Video](#)
- [Third Video](#)

This method is ideal. In the event that the browser does not natively support video, we still have a list of links that the user may download, and view locally.

The Styling

Next, if we add just a touch of styling, we can create a more playlist-like display.

```
body {  
    text-align: center;  
    padding: 2em 0;  
    width: 640px;  
    margin: auto;  
    font-family: sans-serif;  
    -moz-box-sizing: border-box;  
    box-sizing: border-box;  
}  
  
ul { padding: 0; }  
  
li {  
    background: #292929;  
    color: white;  
}  
  
li:hover, .playing { background: black; }  
  
li a {  
    color: white;  
    text-decoration: none;  
    font-size: 1.5em;  
    border-bottom: 1px solid black;  
    border-top: 1px solid #666;  
    display: block;  
    padding: 1em;  
}  
  
li:first-child a { border-top: 0; }  
li:last-child a { border-bottom: 0; }  
  
video { width: 640px; }
```



Playlist

First Video

Second Video

Third Video

Okay, it won't win any awards. I already told you; get off my case with this design stuff. It's hard!

Notice that a *class* of **playing** has been applied to the first video in the playlist. This is due to the fact that this item in the playlist is the default video when the page loads.

The JavaScript

We'll begin by performing some feature detection, and setting some base variables that will be needed.

JavaScript

```
if ( Modernizr.video ) {  
    (function() {  
        var video = document.querySelector('video'),  
            playlist = document.querySelector('ul.playlist'),  
            setExtension = (function() {  
                var extension = Modernizr.video.webm ? '.webm' : ▷  
                    '.m4v';  
                return function(src) {  
                    return src.replace(/.\.\w{3,4}$/, extension);  
                };  
            })();  
        })();  
    }  
}
```

jQuery

```
if ( Modernizr.video ) {  
    (function() {  
        var video = $('video').first(),  
            playlist = $('ul.playlist'),  
            setExtension = (function() {  
                var extension = Modernizr.video.webm ? '.webm' : ▷  
                    '.m4v';  
                return function(src) {  
                    return src.replace(/.\.\w{3,4}$/, extension);  
                };  
            })();  
        })();  
    }  
}
```

The **setExtension** method uses Modernizr to determine which codec is most appropriate for the browser. If we refer to the excellent [CanIUse.com](#) website, we'll find that, between the WebM and H.264 codecs, we'll make nearly all modern browsers happy.



Dependent upon the result of the `Modernizr.video.web` test, the variable `extension` will either be equal to `webm` or `m4v`. Ultimately, this value will be appended to the name of the video that needs to be loaded. This way, `some-video` can be `some-video.m4v` in, say, Safari, and `some-video.web` in Chrome. Next, we'll attach a `click` handler to the playlist. When an item is selected, or clicked, we need to update the `video` tag to include the new `src`.

JavaScript

```
if ( Modernizr.video ) {
  (function() {
    var video = document.querySelector('video'),
        playlist = document.querySelector('ul.playlist'),
        setExtension = (function() {
          var extension = Modernizr.video.webm ? '.webm' :
            '.m4v';
          return function(src) {
            return src.replace(/.\w{3,4}$/, extension);
          }
        })()
  })
}
```

```
        };
    })());
    playlist.addEventListener('click', function(e) {
        // video-2.m4v becomes either video-2.webm or
        // remains video-2.m4v
        var href = setExtension( this.href );
        e.preventDefault();
    }, false);
})();
}
```

jQuery

```
if ( Modernizr.video ) {
    (function() {
        var video = $('video').first(),
            playlist = $('ul.playlist'),
            setExtension = (function() {
                var extension = Modernizr.video.webm ? '.webm' : '.m4v';
                return function(src) {
                    return src.replace(/.\w{3,4}$/, extension);
                };
            })();
        playlist.on('click', 'a', function(e) {
            var link = $(this),
                href = setExtension( link.attr('href') );
            e.preventDefault();
        });
    })();
}
```

The most notable bit of logic here is:

```
// video-2.m4v becomes either video-2.webm or      ▷  
  remains video-2.m4v  
  href = setExtension( link.attr('href') );
```

Because we'll be dynamically assigning a new source to the `video` tag, we need to provide the correct video format. For instance...

```
document.querySelector('video').src = 'new-video.m4v';
```

... will, indeed, update the video; however, what if the user is in Firefox? That browser doesn't currently support the H.264 codec. That's why this line references the `setExtension` function, which slices off the extension of the selected video, and replaces it with a new one, determined by Modernizr's feature detection.

Note that this code assumes that, for every `video-x.m4v`, there is also a respective `video-x.webm` within the same folder.

At this point, `href` will be equal to the path to the next video that should be played. Let's update the `video` tag:

JavaScript

```
if ( Modernizr.video ) {  
  (function() {  
    var video = document.querySelector('video'),  
        playlist = document.querySelector('ul.playlist'),  
        setExtension = (function() {  
          var extension = Modernizr.video.webm ? '.webm' :      ▷  
            '.m4v';  
          return function(src) {  
            return src.replace(/.\w{3,4}$/, extension);  
          };  
        })(),
```

```
changeVideo = function(newVideoPath) {
    video.poster = '';
    video.src = newVideoPath;
    video.load();
    video.play();
};

playlist.addEventListener('click', function() {
    var href = setExtension( this.href );
    changeVideo(href);
    e.preventDefault();
}, false);
})();
}
```

jQuery

```
if ( Modernizr.video ) {
(function() {
    var video = $('video').first(),
        playlist = $('ul.playlist'),
        setExtension = (function() {
            var extension = Modernizr.video.webm ? '.webm' : >
                '.m4v';
            return function(src) {
                return src.replace(/.\w{3,4}$/, extension);
            };
        })(),
        changeVideo = function(newVideoPath) {
            if ( video instanceof jQuery ) video = video[0];
            video.poster = '';
            video.src = newVideoPath;
            video.load();
            video.play();
        };
    playlist.on('click', 'a', function(e) {
        var link = $(this),

```

```
    href = setExtension( link.attr('href') );
    changeVideo(href);
    e.preventDefault();
});
})();
}
```

Within the `changeVideo` function, we must either reassign the video's `poster` attribute to a new image, or remove it entirely when a new video is dynamically added. Otherwise, it will appear as if we've loaded the same video as the previous one.

```
video.poster = '';
```

Next, the `video` tag is updated with the new `src` that was passed to the function.

```
video.src = newVideoPath;
```

Once the source has been updated, we only need to load and play it!

```
video.load();
video.play();
```

To ensure that we never use a jQuery-wrapped object when referencing the media element's attributes and methods, we can provide a simple check at the top of the function: `if (video instanceof jQuery) video = video[0];`. This translates to, “If a jQuery-wrapped object was passed to this function, rather than the node itself, update the `video` variable to equal the node only.”

Toggling the playing Class

To provide additional feedback to the user, we should update the playlist styling, so that the newly selected item receives the

playing class, and its associated styling. However, in addition to adding the **playing** class, we must also be sure to remove this class name from its siblings.

JavaScript

```
if ( Modernizr.video ) {  
    (function() {  
        var video = document.querySelector('video'),  
            playlist = document.querySelector('ul.playlist'),  
            setExtension = (function() {  
                var extension = Modernizr.video.webm ? '.webm' : '.m4v';  
  
                return function(src) {  
                    return src.replace(/.\.\w{3,4}$/, extension);  
                };  
            })(),  
            changeVideo = function(newVideoPath) {  
                video.poster = '';  
                video.src = newVideoPath;  
                video.load();  
                video.play();  
            },  
            toggleclassNames = function(context, name) {  
                name = name || 'playing';  
                [].forEach.call(context.querySelectorAll('.' + name), function(el) {  
                    el.classList.remove(name);  
                });  
                this.classList.add(name);  
            };  
        playlist.addEventListener('click', function() {  
            var href = setExtension( this.href );  
            changeVideo(href);  
            toggleclassNames.call( this, playlist );  
        });  
    })();  
}
```

```
        e.preventDefault();
    }, false);
})());
}
```

jQuery

```
if ( Modernizr.video ) {
(function() {
var video = $('video'),
playlist = $('ul.playlist'),
setExtension = (function() {
var extension = Modernizr.video.webm ? '.webm' : '.m4v';
return function(src) {
return src.replace(/.\w{3,4}$/, extension);
};
})(),
changeVideo = function(newVideoPath) {
if ( video instanceof jQuery ) video = video[0];
video.poster = '';
video.src = newVideoPath;
video.load();
video.play();
},
toggleclassNames = function(context, name) {
name = name || 'playing';
context.find('.'+name).removeClass(name);
this.addClass(name);
};
playlist.on('click', 'a', function(e) {
var link = $(this),
href = setExtension( link.attr('href') );
changeVideo(href);
toggleClassNames.call( link, playlist );
e.preventDefault();
});
```

```
});  
})();  
}
```

The `toggleclassNames` function accepts a context, and an optional class name to assign (which defaults to `playing`).

```
toggleclassNames = function(context, name) {  
    name = name || 'playing';  
    [].forEach.call(context.querySelectorAll('.' + name), ▶  
        function(el) {  
            el.classList.remove(name);  
        });  
    this.classList.add(name);  
};
```

When called, it queries the DOM, using the provided context as a base, for any potential elements with a class of `playing`. Should it find any, it will remove the class from the node. That way, we don't end up with multiple items in the playlist containing this class.

Lastly, it assigns the `playing` class to `this`, which will be equal to the item in the playlist that was clicked.

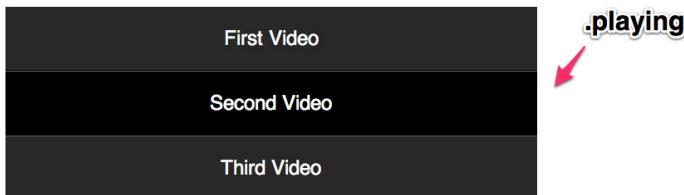
To ensure that `this` correctly refers to the anchor tag that was clicked in the playlist (rather than the `window` object), we use:

```
toggleclassNames.call( link, playlist );
```

The first parameter declares what should be considered as `this`. The second parameter is the context that will be used as the base for searching for any lingering class names that need to be removed.



Playlist



When Each Video Ends

The final step is to automatically transition to the next video when the current one completes. To accomplish this, we can hook into the **ended** event, as reviewed earlier.

JavaScript

```
if ( Modernizr.video ) {  
    (function() {  
        var video = document.querySelector('video'),  
            playlist = document.querySelector('ul.playlist'),  
            setExtension = (function() {  
                var extension = Modernizr.video.webm ? '.webm' : '.m4v';  
                return function(src) {  
                    return src.replace(/.\w{3,4}$/, extension);  
                };  
            }());  
        video.addEventListener('ended', function() {  
            var index = playlist.querySelector('li.current').index;  
            if (index < playlist.children.length - 1) {  
                index++;  
            } else {  
                index = 0;  
            }  
            var item = playlist.children[index];  
            item.classList.add('current');  
            item.querySelector('img').src = item.getAttribute('data-video');  
            video.setAttribute('src', item.getAttribute('data-video'));  
        }, false);  
    }());  
}
```

```
})(),
changeVideo = function(newVideoPath) {
    video.poster = '';
    video.src = newVideoPath;
    video.load();
    video.play();
},
toggleclassNames = function(context, name) {
    name = name || 'playing';
    [].forEach.call(context.querySelectorAll('.' + name), function(el) {
        el.classList.remove(name);
    });
    this.classList.add(name);
};
playlist.addEventListener('click', function() {
    var href = setExtension( this.href );
    changeVideo(href);
    toggleclassNames.call( this, playlist );
    e.preventDefault();
}, false);
video.addEventListener('ended', function() {
    var nextVideo = playlist.querySelector('a.playing').►
        parentNode.nextElementSibling.querySelector('a'),
        nextVideoPath;
    // If nothing found, we're at the end. Reset.
    if ( !nextVideo[0] ) nextVideo = playlist.►
        querySelector('li a');
    // Get the path to the video we should play.
    nextVideoPath = setExtension( nextVideo.href );
    // Change to the next video.
    changeVideo(nextVideoPath);
    toggleclassNames.call( nextVideo, playlist );
}, false);
})();
}
```

jQuery

```
if ( Modernizr.video ) {  
    (function() {  
        var video = $('video'),  
            playlist = $('ul.playlist'),  
            setExtension = (function() {  
                var extension = Modernizr.video.webm ? '.webm' : ►  
                    '.m4v';  
                return function(src) {  
                    return src.replace(/\.\\w{3,4}$/, extension);  
                };  
            })(),  
            changeVideo = function(newVideoPath) {  
                if ( video instanceof jQuery ) video = video[0];  
                video.poster = '';  
                video.src = newVideoPath;  
                video.load();  
                video.play();  
            },  
            toggleclassNames = function(context, name) {  
                name = name || 'playing';  
                context.find('.'+name).removeClass(name);  
                this.addClass(name);  
            };  
            playlist.on('click', 'a', function(e) {  
                var link = $(this),  
                    href = setExtension( link.attr('href') );  
                changeVideo(href);  
                toggleclassNames.call( link, playlist );  
                e.preventDefault();  
            });  
            video.on('ended', function() {  
                var nextVideo = playlist.find('a.playing').parent().►  
                    next().children('a'),  
                    nextVideoPath;
```

```
// If nothing found, we're at the end. Reset.  
if ( !nextVideo[0] ) nextVideo = playlist.find('li a').first();  
// Get the path to the video we should play.  
nextVideoPath = setExtension( nextVideo.attr('href') );  
// Change to the next video.  
changeVideo(nextVideoPath);  
toggleClassNames.call( nextVideo, playlist );  
});  
})();  
}
```

When the `ended` event fires, we only need to determine the path to the next video, apply the necessary extension, and call the `changeVideo` function. Because we've abstracted most of the logic away to its own function, at this point our job is a simple one! It's important to remember that, eventually, we'll get to the last video in the playlist. Once we do, the playlist should reset back to the first video, and begin again. This line accomplishes that:

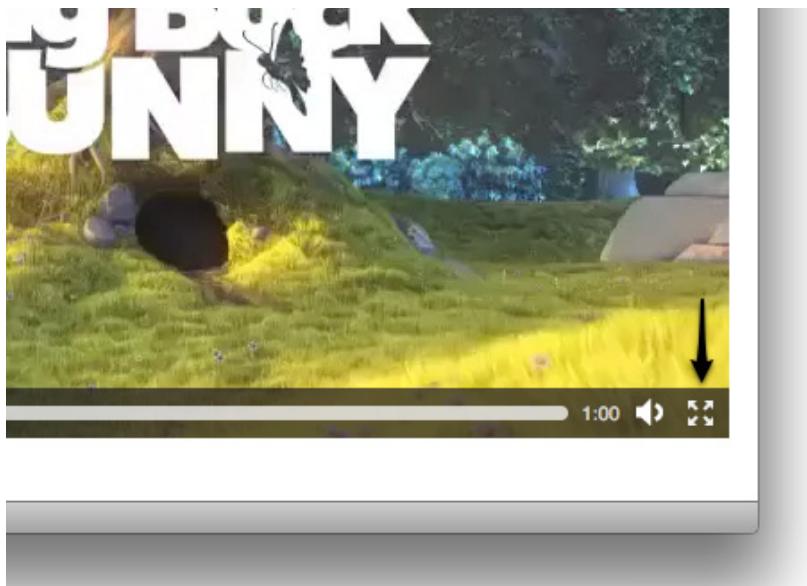
```
// If there isn't a next video,  
// make it equal to the first one.  
if ( !nextVideo[0] ) nextVideo = playlist.find('li a').first();
```

Complete

That wasn't too bad! It's certainly not rocket science! We're merely hooking into a handful of events, and updating the DOM accordingly.

The Full Screen API

While, initially, the idea of programmatically making an element display in “full screen” was argued against (due to security concerns), we’ve slowly come to the realization that it’s a necessity, as more and more apps move to the web. For example, consider the benefits of an HTML5 game displayed in full screen. Or, more naturally, full screen video! Thanks to Webkit, and [Robert O’Callahan’s proposal](#), the [Full Screen API](#) — which is not limited to video — now affords us with this ability.



The native full screen widget shown in Firefox 12 Beta.

At the time of this writing, the spec is partially implemented in stable versions of Chrome and Safari, and the nightly builds of Firefox 12. The latter even provides a full-screen button on its video player!

ROCK* TIP

YouTube now makes use of the Full Screen API.



We can switch a video to full screen programmatically, via the `requestFullScreen` method. As with most new experimental features, we need to use a vendor-specific version of the method, while they iron out the kinks, and more fully implement the spec (which, itself, is still in the early stages).

The following code snippet listens for when a user presses the `enter` key, and then calls the `requestFullScreen` method on the media object, accordingly. This will transition the video to fullscreen.

```
var video = document.querySelector("video");
document.addEventListener("keydown", function(e) {
  if (e.keyCode === 13) {
    // enter key was pressed
    var fs = video.requestFullScreen || >
              video.mozRequestFullScreen || >
              video.webkitRequestFullScreen;
    fs && fs.call(video);
  }
}, false);
```

Note that we're assigning the method to a variable to keep from adding multiple `if` statements to detect support, such as:

```
if ( video.mozRequestFullScreen ) {
  video.mozRequestFullScreen();
} else if ( video.webkitRequestFullScreen ) {
  video.webkitRequestFullScreen();
}
```

Once in full screen mode, we can exit via the control panel, provided in Chrome, or by pressing the `escape` key. Alternatively, we can programmatically exit, via the `cancelFullScreen` method.



Firefox 12

Refer to the [specification](#) for more details on working with the full screen API.

Summary

The truth is, we've barely scraped the surface of what's possible with native HTML5 video and audio. I urge you to begin toying around with the idea of using canvas to manipulate video in real-time. Yes — fun times are ahead!

9

Track that Sucka with Geolocation

Browser Support



Sometimes, when we read the acronym, “API,” we immediately assume a certain level of complexity. But, luckily, HTML5 geolocation is incredibly simple. You’ll likely find that you spend more time working with a third party tool, like the Google Maps API, rather than the geolocation, itself!

Is Geolocation Part of HTML5?

Technically, no. It’s part of its own [W3C specification](#). That said, it’s commonly grouped into the term, HTML5, because it, too, represents the new technologies that we should be working with.

What is Geolocation?

It’s very much what you probably expect. It allows a browser or device to pin-point your current location. The API allows us to either perform this tracking on a one-off basis, or continuously, which could be helpful in situations when the visitor to your site is on the move.

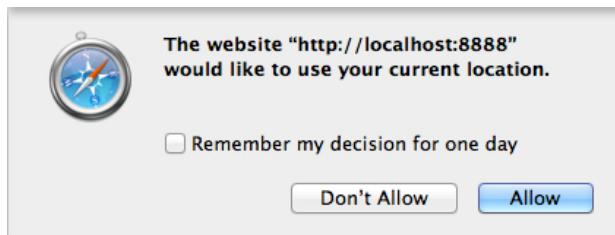
Crash Course

Okay, let's cut the jibber-jabba. Personally, I like to view the most minimal example of an API before I dig into the available options. Here's the simplest demo I can manage:

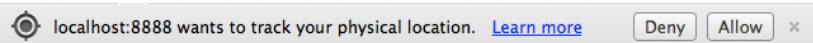
```
<!doctype html public 'track that sucka'>
<meta charset=utf-8>
<title> Geolocation </title>
<script>
(function(geo) {
  if ( geo ) {
    geo.watchPosition(function(location) {
      console.log(location);
    });
  }
})(navigator.geolocation);
</script>
```

Paste this snippet into a blank HTML page, and view it in the browser. However, you may find that nothing is logged to the console. This is due to the fact that, for geolocation to work as expected, you either need to store the file on your web host, or use your localhost to access the page — such as MAMP (if you're a PHP user). Alternatively, use [JSBIN.com](http://jsbin.com).

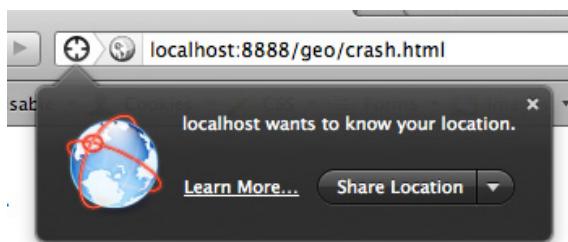
Once this issue is remedied, reload the page, and the browser will request access to track your physical location.



Safari



Chrome



Firefox

Note that the browsers aren't doing this of their own accord. The specification requires it.

“A conforming implementation of this specification must provide a mechanism that protects the user's privacy and this mechanism should ensure that no location information is made available through this API without the user's express permission. **”** — [Geo Spec](#)

When you think about it, of course it's a necessity to first request permission before tracking the user. Otherwise, that could be a huge security concern!

Next, view the console, and you'll find an **object**, which contains, among other things, the latitude and longitude coordinates of your current location.



[View the simplest geolocation demo possible.](#)

Now that's you've had a taste, let's dig in a bit more.

Testing

As with all of the new APIs, we first need to determine whether the browser **recognizes** the API. As you learned in the feature detection chapter, testing for geolocation support is a cinch:

```
if ( navigator.geolocation ) alert('ahh hell yeah');
```

Or, with Modernizr:

```
if ( Modernizr.geolocation ) alert('ahh hell yeah');
```

If we review the Modernizr source, we can see that, yep, it's doing the same thing:

```
tests['geolocation'] = function() {
    return !!navigator.geolocation;
};
```

The Two Core Methods

The Geolocation API is composed of two primary methods:

1. **getCurrentPosition**
2. **watchPosition**

These two methods are quite similar, except for the fact that **watchPosition** will continue to update your code. Naturally, this should be used in situations where the user's position will continuously be changing.

That said, at the time of this writing, it's better to use **watchPosition** in nearly all situations, due to the fact that you'll get a more accurate tracking. When **watchPosition** is used, many browsers will use the first ten seconds to fine-tune, or pinpoint the visitor's location.

getCurrentPosition()

Both of these methods accept one, two, or three arguments.

1. **successCallback** – Will execute if, and when the browser successfully tracks the visitor.
2. **errorCallback** – Will execute if the attempt to track the visitor fails.
3. **positionOptions** – Optional object for setting **enableHighAccuracy**, **timeout**, and **maximumAge**.

These arguments may be applied, like so:

```
navigator.geolocation.getCurrentPosition(successCallback, ▶
    errorCallback, {
        enableHighAccuracy: true, // slower, but more accurate
        timeout: 15000, // expressed in milliseconds (15 seconds)
        maximumAge: 0 // means, acquire a new position
    });

```

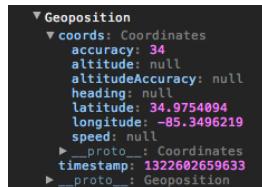
If I test the following code myself — as I sit in a Starbucks in Chattanooga, TN — **location** returns:

```
if ( navigator.geolocation ) {
    navigator.geolocation.getCurrentPosition(function(▶
        location) {
        // location contains timestamp, and coordinates
        console.log(location);
    });
}

```

Generally, you'll find that the two properties you require are **latitude** and **longitude**. We can log those values to the console, like so:

```
console.log( location.coords.latitude, location.coords.▶  
    longitude); // 34.9754094 -85.3496219
```



watchPosition()

This should be your preferred method in most situations. Beyond the accuracy aspect that I mentioned earlier, the core difference between `getCurrentPosition` and `watchPosition` is that the latter will continue keeping an eye on the user's position. When it changes, the applicable success or error function will be triggered, accordingly.

“The watch operation then must continue to monitor the position of the device and invoke the appropriate callback every time this position changes. The watch operation must continue until the `clearWatch` method is called with the corresponding identifier.**”**

— [Geolocation Spec](#)

`watchPosition` will return a value that identifies the current `watch` operation. This can be used to clear it, if necessary — similar to what you might be familiar with from `setInterval`. To clear it, we use the `clearWatch` method.

```
var watch = navigator.geolocation.watchPosition(function(▶  
    location) {  
});  
  
// later on  
navigator.geolocation.clearWatch(watch);
```

```
console.log(watch); // unique identifier that increases      ▶  
each time a new position is acquired
```

Options

enableHighAccuracy (defaults to false)

This is our way to say to the browser, “Hey — I’m cool with potentially waiting a bit longer. Give me the most accurate results possible.” According to the spec:

“The intended purpose of this attribute is to allow applications to inform the implementation that they do not require high accuracy geolocation fixes and, therefore, the implementation can avoid using geolocation providers that consume a significant amount of power (e.g. GPS). **”**

```
navigator.geolocation.watchPosition(successCallback,      ▶  
errorCallback, {  
  enableHighAccuracy: true  
});
```

timeout (defaults to infinity)

This option allows us to specify exactly how long we’re willing to wait for a response before the necessary callback function is called. If the `timeout` elapses, and a position has not yet been acquired, the `errorCallback` method will be called. Otherwise, `successCallback` executes.

```
navigator.geolocation.watchPosition(successCallback,      ▶  
errorCallback, {  
  timeout: 15000 // fifteen seconds  
});
```

maximumAge (defaults to 0)

This option determines how long the browser is allowed to cache the user's position. If set to 0, on each page load, a new position will be acquired. Otherwise, after the first request, the browser will wait until the `maximumAge` length (in milliseconds) expires before acquiring a new position.

```
navigator.geolocation.watchPosition(successCallback,           ▶
  errorCallback, {
    maximumAge: 1000 * 60 * 60 // one hour
  });
}
```

Google Maps

As mentioned earlier in this chapter, there honestly isn't too much to the Geolocation API, thankfully. There are additional properties that might be available to mobile users, such as determining the altitude of the user. But, other than that, the API is a cinch to learn! The bulk of your work will come from the process of integrating the results with a third party tool, like the Google Maps API.

It doesn't feel fair to instruct how to acquire a user's latitude/longitude coordinates without at least demonstrating how to get that information on a map.

Demo #1—Hello World of Google Maps

If you have absolutely zero experience with the Google Maps API, give [this tutorial a read-through](#). Pay special attention to the instructions for obtaining an API key. I'll expect you to handle that process on your own. Don't worry; it's easy!

“ All Maps API applications should load the Maps API using an API key. Using an API key enables you to monitor your application's Maps API usage, and

ensures that Google can contact you about your application if necessary. If your application's Maps API usage exceeds the Usage Limits, you must load the Maps API using an API key in order to purchase additional quota. ,,

Next, create a new HTML file, and paste the following snippet in:

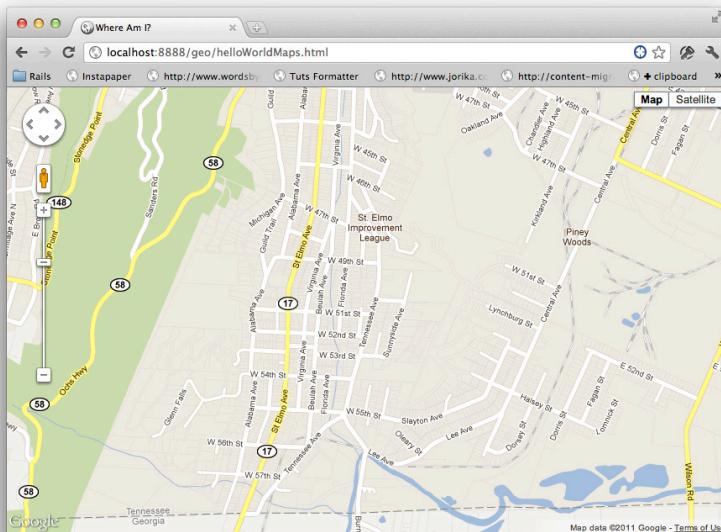
```
<!doctype html>
<html>
<head>
<meta charset="utf-8">
<title>Where Am I?</title>
<style>
    html, body, #map { width: 100%; height: 100%; margin: 0; }
</style>
</head>
<body>
<div id="map"></div>
<script src="http://maps.googleapis.com/maps/api/js?key= YOUR_API_KEY&sensor=true_or_false"></script>
<script>
if ( navigator.geolocation ) {
    navigator.geolocation.watchPosition(function(pos) {
        var coords = new google.maps.LatLng(pos.coords.
            latitude, pos.coords.longitude),
            opts = {
                zoom: 15,
                center: coords,
                mapTypeId: google.maps.MapTypeId.ROADMAP
            },
        map = new google.maps.Map(document.querySelector( '#map'), opts);
    });
}
</script>
```

194

Track that Sucka with Geolocation

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

Be sure to substitute your API key within the `script` tag, accordingly. Before we decode this, view it in your browser to see the result.



Nifty, ay? This script uses HTML5 geolocation to pin-point your location (don't stalk me). We then pass the coordinates of the visitor on to the Google Maps API, which displays the appropriate map. Let's take a closer look, from top to bottom.

I've decided that the map should take up the entire width and height of the browser window. This bit of styling simply ensures that it does.

```
<style>  
    html,body,#map { width: 100%; height: 100%; margin: 0; }  
</style>
```

This **div** gives us a place to display our map. Plain and simple.

```
<div id="map"></div>
```

To gain access to the Google Maps API, we must pull in the necessary **script**, while passing in our own unique API key.

```
<script src="http://maps.googleapis.com/maps/api/js?key= ▶
YOUR_API_KEY&sensor=true_or_false"></script>
```

Remember: don't execute a bunch of code if the browser doesn't provide any support for geolocation! That's wasteful.

```
if ( navigator.geolocation ) {}
```

We use the **watchPosition** method to track the visitor's position. The coordinates (among other things) can be accessed, via the **pos** argument.

```
navigator.geolocation.watchPosition(function(pos) {} );
```

Here's our first bit of Google Maps code. The **LatLng** method accepts two parameters: the latitude and longitude coordinates, respectively. We pass in the values that we captured, thanks to the Geolocation API. This code essentially tells Google, "Hey dude — go fetch me the piece of the globe that matches these coordinates."

```
var coords = new google.maps.LatLng(pos.coords.latitude, ▶
pos.coords.longitude);
```

Now, we haven't yet placed a map on the page. We do that with this block of code. We call **google.maps.Map** and provide it with a path to the location on the page where the map should be displayed. Secondly, we provide a handful of options, which specify how the map should appear. In this case, the map should

be zoomed to level fifteen, the visitor's coordinates should be centered on the map, and Google should use the standard road-type map (rather than something like "Satellite").

```
opts = {
  zoom: 15,
  center: coords,
  mapTypeId: google.maps.MapTypeId.ROADMAP
},
map = new google.maps.Map(document.querySelector('#map'), ▶
  opts);
```

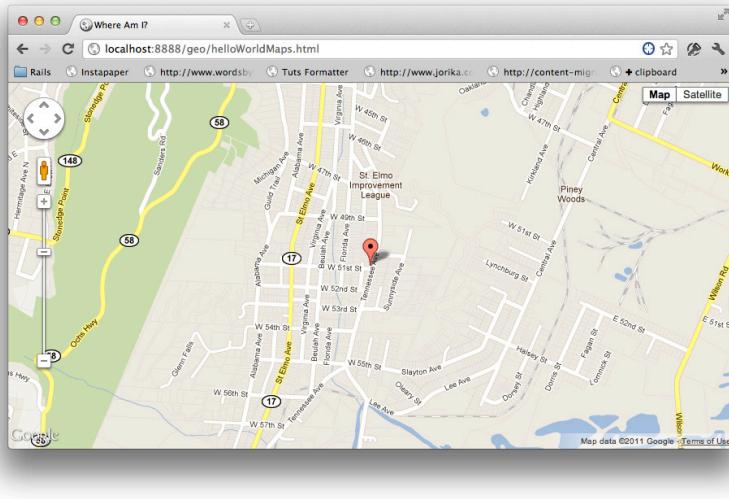
Hey, that's not too difficult!

marker

There is one problem, though: we can't tell what the exact position is, because there aren't any markers on the map. Let's add one now.

```
if ( navigator.geolocation ) {
  navigator.geolocation.watchPosition(function(pos) {
    var coords = new google.maps.LatLng(pos.coords.▶
      latitude, pos.coords.longitude),
    opts = {
      zoom: 15,
      center: coords,
      mapTypeId: google.maps.MapTypeId.ROADMAP
    },
    map = new google.maps.Map(document.querySelector(▶
      '#map'), opts),
    marker = new google.maps.Marker({
      position: coords,
      map: map
    });
  });
}
```

Notice that we've added a new variable that places a *marker* exactly at the user's coordinates. We also must specify which map the marker corresponds to, since there could be more than one on the page.



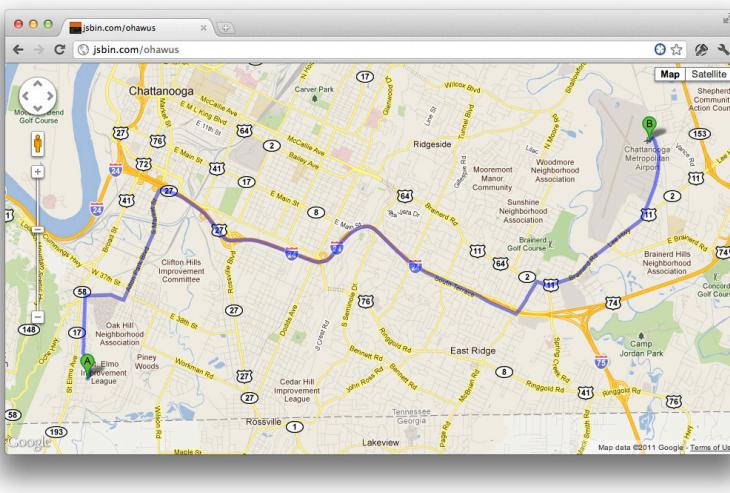
What we have here is a map that shows the visitor their current position on the globe. Neat, but not very useful.



[View the geolocation demo.](#)

Demo #2—Directions

I always find myself looking up how to get to the airport. I travel by plane rarely enough to the point that, each time, I have to visit Google Maps and get a refresher. Let's create a web page that will provide me with directions from my current location (wherever that might be) to my local Chattanooga airport.



Necessities

As I see it, there are a few things that **must** be integrated:

1. The map should keep track of my location, and update the map and directions at a regular interval.
2. I should provide a polyfill for users who can't make use of the HTML5 geolocation API.
3. It's imperative that my acquired location is as exact as possible.

Let's get started. We begin with the skin for our page.

```
<!DOCTYPE html>
<html>
<head>
  <meta charset="utf-8">
  <title> Directions to the Chattanooga Airport </title>
<style>
  html, body, #map { width: 100%; height: 100%; margin: 0; }
</style>
```

```
</head>
<body>
  <div id="map"></div>
  <script src="http://maps.googleapis.com/maps/api/js?key= ▶
    YOUR_API_KEY&sensor=false"></script>
  <script>
    (function() {
      if ( navigator.geolocation ) {
        var coords = navigator.geolocation.watchPosition( ▶
          callback);
      }());
    }
  </script>
</body>
</html>
```

The map should keep track of my location, and update the map and directions at a regular interval.

With this guideline in mind, rather than using `getCurrentPosition`, it's more appropriate to use `watchPosition`. That way, when the user's position changes (maybe they're in a car), our callback function will be triggered again.

```
(function() {
  if ( navigator.geolocation ) {
    var coords = navigator.geolocation.watchPosition( ▶
      hereWeGooo);
  }();
})
```

I should provide some level of feedback for users who can't make use of the HTML5 geolocation API.

Next, there needs to be accessibility for users in older browsers, like Internet Explorer 7. We'll create a simple polyfill at the end of this chapter.

It's imperative that my acquired location is as exact as possible.

Of course, we don't want the map to display directions from a location that's twenty miles from us! Even if it takes a bit longer, we'll pass the `enableHighAccuracy` option.

```
(function() {  
  if ( navigator.geolocation ) {  
    var coords = navigator.geolocation.watchPosition( ►  
      hereWeGooo, null, { enableHighAccuracy: true } );  
  }();  
})()
```

hereWeGooo

It only seems appropriate for our `success` callback's name to pay tribute to the world's favorite plumber.

This is where we'll create the map, and apply the route to the Chattanooga Airport. We'll begin by storing a reference to the `coords` object in a variable, appropriately called, `coords`. This is simply a convenience.

```
(function() {  
  if ( navigator.geolocation ) {  
    var coords = navigator.geolocation.watchPosition( ►  
      hereWeGooo, null, { enableHighAccuracy: true } );  
  }  
  function hereWeGooo(position) {  
    var coords = position.coords  
  }  
})()
```

Next, we'll take the location of the user, and set up the map.

```
function hereWeGooo(coords) {  
  var coords = coords.coords,  
    latlng = new google.maps.LatLng(coords.latitude, ▶  
      coords.longitude),  
    map = new google.maps.Map(document.querySelector(▶  
      "#map"),  
      { mapTypeId: google.maps.MapTypeId.ROADMAP }  
    );  
}
```

Most of this is similar to what we did in the previous demo, but, now, we'll apply directions.

```
function hereWeGooo(coords) {  
  var coords = coords.coords,  
    latlng = new google.maps.LatLng(coords.latitude, ▶  
      coords.longitude),  
    map = new google.maps.Map(document.querySelector(▶  
      "#map"),  
      { mapTypeId: google.maps.MapTypeId.ROADMAP }  
    ),  
  
    // Takes care of the traced route, and the markers  
    directionsDisplay = new google.maps.DirectionsRenderer(),  
  
    // Specifies that we want to use the directions API  
    directionsService = new google.maps.DirectionsService(),  
  
    // This is where we specify where we're coming from,  
    // and going to.  
    // origin = start point  
    request = {  
      origin: latlng,  
      destination: 'Chattanooga Airport', // replace with ▶  
        your own airport  
      travelMode: google.maps.DirectionsTravelMode.DRIVING  
    };  
}
```

```
// Apply the directions to the map
directionsDisplay.setMap(map);

// Attempt to set the route. If successful, apply the      ▶
  directions to the map
directionsService.route(request, function(response,      ▶
    status) {
  if (status === google.maps.DirectionsStatus.OK) {
    directionsDisplay.setDirections(response);
  }
});
}
```

Polyfill

The last step is to provide a fallback for users in older browsers, which don't support HTML5 geolocation. Let's build a simple polyfill, inspired by a [Gist](#) created by Paul Irish.

We begin by creating a self-invoking function, which accepts the `geolocation` object.

```
(function(geolocation) {
})(navigator.geolocation);
```

Now, in modern browsers, `geolocation` will be equal to the `navigator.geolocation` object. However, in older browsers, where the API is not supported, `geolocation` will equal `undefined`, which is good.

Feature Test

Next, before we do anything else, we perform a quick feature test to determine if we need to even bother with providing a polyfill.

```
(function(geolocation) {  
    // If the browser recognizes geolocation, sweet!  
    // Get outta here.  
    if ( geolocation ) return;  
})(navigator.geolocation);
```

cache

Let's implement a makeshift caching system. If the `cache` object is empty, we'll use Google's JavaScript API to fetch the user's current location. If it's not empty, then we can save a bit of bandwidth, and exit early. We'll do the brunt of the work shortly; for now, we'll simply create the variable.

```
(function(geolocation) {  
    // If the browser recognizes geolocation, sweet!  
    // Get outta here.  
    if ( geolocation ) return;  
    var cache;  
})(navigator.geolocation);
```

Redeclare `navigator.geolocation`

A polyfill shouldn't require any additional coding on the developer's behalf. That means we need to overwrite `navigator.geolocation` in older browsers which don't support it. That way, in modern browsers, it correctly refers to the geolocation API; but, in older browsers, it refers to our custom polyfill.

```
(function(geolocation) {  
    if ( geolocation ) return;  
    var cache;  
    // Overwrite navigator.geolocation in old browsers  
    // For now, we'll set it to an empty object  
    geolocation = navigator.geolocation = {};  
})(navigator.geolocation);
```

watchPosition

Now, we'll create our own custom `watchPosition` method. For brevity's sake, it won't be as full featured as it should be. This is mostly a proof of concept.

```
(function(geolocation) {  
    if ( geolocation ) return;  
    var cache;  
    // Overwrite navigator.geolocation in old browsers  
    // For now, we'll set it to an empty object  
    geolocation = navigator.geolocation = {};  
    geolocation.watchPosition = function(successCallback,      ▷  
        failureCallback) {}  
})(navigator.geolocation);
```

Google's JavaScript API

We can use Google's own API to determine the visitor's position.

```
(function(geolocation) {  
    if ( geolocation ) return;  
    var cache;  
    // Overwrite navigator.geolocation in old browsers  
    // For now, we'll set it to an empty object  
    geolocation = navigator.geolocation = {};  
    geolocation.watchPosition = function(successCallback,      ▷  
        failureCallback) {  
        // If we already have results, send those through to      ▷  
            the cb func.  
        if (cache) successCallback(cache);  
        // We'll use the Google JS API to track the visitor  
        // Note that $.getScript requires jQuery to work  
        $.getScript('http://www.google.com/jsapi', function() {  
            var location = google.loader.ClientLocation;  
            // If the location couldn't be captured, trigger  
            failureCallback.
```

```
if ( !location ) {  
    if ( failureCallback && typeof(failureCallback)  
        === 'function' ) {  
        failureCallback();  
    }  
    return;  
}  
// Grab the coordinates & store them in the cache.  
cache = {  
    coords : {  
        'latitude': google.loader.ClientLocation.▶  
                    latitude,  
        'longitude': google.loader.ClientLocation.▶  
                     longitude  
    }  
};  
// Execute the success callback function,  
// and send the cache obj through.  
if ( cb && typeof(cb) === 'function' ) ▶  
    successCallback(cache);  
});  
});  
}(navigator.geolocation);
```

Usage

It's a very simple and unfinished polyfill, but this will do the trick for our little demo. What's excellent about polyfills is that you don't need to **learn** how to use them. Use the geolocation API as you normally would, and this code will only kick in when necessary. It's like the autopilot inflatable guy in the movie *Airplane*.

Final Script

```
<!DOCTYPE html>
<html>
<head>
  <meta charset="utf-8">
  <title> Directions to the Chattanooga Airport </title>
  <style>
    html, body, #map { width: 100%; height: 100%; margin: 0; }
  </style>
</head>

<body>
  <div id="map"></div>
  <script src="http://maps.googleapis.com/maps/api/js?key= ▶
    YOUR_API_KEY&sensor=false">
  </script>
  <script src="https://ajax.googleapis.com/ajax/libs/jquery/ ▶
    1.7.0/jquery.min.js"></script>

<script>
// Polyfill
(function(geolocation) {
  // If the browser recognizes geolocation, no need for ▶
  // polyfill
  if ( geolocation ) return;

  // We'll cache the results
  var cache;

  // Overwrite navigator.geolocation in old browsers.
  // We'll create our own watchPosition
  geolocation = navigator.geolocation = {};

  // write our polyfill
  geolocation.watchPosition = function(successCallback, ▶
    failureCallback) {
```

```
// If we already have results, send those through to ▶
// the cb func.
if (cache) successCallback(cache);

// We'll use the Google JS API to track the visitor
$.getScript('http://www.google.com/jsapi', function() {
    var location = google.loader.ClientLocation;

    // If the location couldn't be captured, trigger ▶
    // the failure cb.
    if ( !location ) {
        if ( failureCallback && typeof(failureCallback)▶
            === 'function' ) {
            failureCallback();
        }
        return;
    }

    // Grab the coordinates & store them in the cache.
    cache = {
        coords : {
            'latitude': google.loader.ClientLocation. ▶
                latitude,
            'longitude': google.loader.ClientLocation. ▶
                longitude
        }
    };

    // Execute the success callback function,
    // and send the cache obj through.
    if ( cb && typeof(cb) === 'function' ) ▶
        successCallback(cache);

    });
});

})();

(navigator.geolocation); // execute function, and pass
// either geo or undefined
```

```
// Proceed as usual.

(function() {
    if ( navigator.geolocation ) {
        var coords = navigator.geolocation.watchPosition( ▶
            hereWeGooo,
            function() {
                // failure
                document.body.innerHTML = 'Sorry. We can\'t get ▶
                    your current location.'
            },
            { enableHighAccuracy: true }
        );
    }

    function hereWeGooo(coords) {
        coords = coords.coords;
        var latlng = new google.maps.LatLng(coords.latitude, ▶
            coords.longitude),
            myOptions = {
                //zoom: 15,
                //center: latlng,
                mapTypeId: google.maps.MapTypeId.ROADMAP
            },
        map = new google.maps.Map(document.querySelector( ▶
            "#map"),
            myOptions
        ),

        directionsDisplay = new google.maps.DirectionsRenderer(),
        directionsService = new google.maps.DirectionsService(),

        request = {
            origin: latlng,
            destination: 'Chattanooga Airport', // replace with ▶
                your own airport
            travelMode: google.maps.DirectionsTravelMode.DRIVING
        };
    }
}
```

```
directionsDisplay.setMap(map);
directionsService.route(request, function(response,      ▶
    status) {
    if (status === google.maps.DirectionsStatus.OK) {
        directionsDisplay.setDirections(response);
    }
});
})();
})();
</script>
</body>
</html>
```

ROCK*
TIP

For a fully featured Geo-location polyfill, refer [here](#).



[View the directions and polyfill demo.](#)

Summary

The fact that the huge majority of this chapter was dedicated to the Google Maps API is a testament to just how easy it is to work with the new Geolocation API. There's absolutely no reason not to take advantage of it.

LO

The Basics of Painting with Canvas

Browser Support



Canvas is an incredibly powerful new 2D drawing tool in HTML5 that will very likely revolutionize the landscape of web applications in the future. With that in mind, it may surprise you to find that, for a brief period, I toyed with the idea of not including a canvas chapter at all in the book. Let me explain.

When setting out to write this book, I had one goal: interpret and describe the complicated specifications in a way that's easiest for the everyday web designer and developer to understand. Which APIs are most applicable to John Q. Front-ender? I worried that if I dedicated fifty pages to a drawing API you might feel a massive pressure to research and understand the considerable complexity of the API.

The truth is, most of the “website builders” among you likely won’t work with canvas much (yet) in your everyday projects — and that’s perfectly okay.

Secondly, canvas is a sufficiently complex API that requires its own book (and there are several). As a result, I’m going to cover the basics: what is canvas, and how can you begin playing around with it. Beyond that, should your needs require extensive use of a native drawing API, at the end of this chapter, I’ll recommend a handful of books and resources, where you can dig deeper into the process of generating graphs, and even creating games.

What is Canvas

“ The canvas element provides scripts with a resolution-dependent bitmap canvas, which can be used for rendering graphs, game graphics, or other visual images on the fly. **”** — [HTML5 Spec](#)

The Painter

Think of HTML5 **canvas** in the same way you would think of a painter’s canvas: a rectangular area that we can draw upon. Similar to the painter, we have the ability to draw lines, circles, squares — any shape can be constructed. Further, we can style these shapes, animate them, specify how they should interact with one another, and much more.

Feature Detection

Detecting support for **canvas** is simple: ensure that the **getContext** method is available.

```
if ( document.createElement('canvas').getContext ) {  
    // browser supports canvas - have fun.  
}
```

If we refer to [Modernizr’s source code](#), we’ll find a helpful comment informing us that the BlackBerry Storm’s browser **incorrectly reports** support for **getContext**, however, it will always return **undefined**. The solution is to ensure that **getContext('2d')** does not return **undefined**.

```
var canvas = document.createElement('canvas');  
if ( canvas.getContext && canvas.getContext('2d') ) {  
    // have fun on BlackBerry too.  
}
```

Or:

```
var supportsCanvas = (function() {  
    var canvas = document.createElement('canvas');  
    return function() {  
        return canvas.getContext && canvas.getContext('2d');  
    };  
})();  
if ( supportsCanvas ) {  
    // ahh yeah  
}
```

Or, as always, use Modernizr:

```
if ( Modernizr.canvas ) { ... }
```

“Hello Canvas”

It's time for the obligatory “create a square” introduction to canvas.

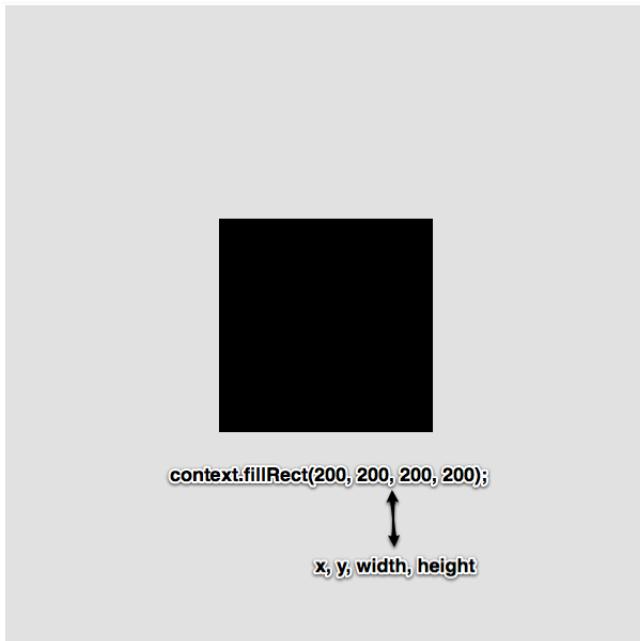
```
<!doctype html>  
<html>  
<head>  
    <meta charset=utf-8>  
    <title></title>  
    <style> canvas { background: #e3e3e3; } </style>  
</head>  
<body>  
  
    <canvas width="600" height="600" id="canvas"></canvas>  
  
    <script>  
        (function() {  
            var canvas = document.getElementById('canvas'),  
                context = canvas.getContext('2d');  
            context.fillStyle = "#000";  
        })();  
    </script>
```

```
    context.fillRect(200, 200, 200, 200);  
})();  
</script>  
  
</body>  
</html>
```



[View the square demo.](#)

This snippet begins with the creation of our **canvas**.



```
<canvas width="600" height="600" id="canvas"></canvas>
```

The **canvas** element can accept a **width** and **height**, which obviously designates the area.

The script then queries the DOM for the desired canvas, and then obtains a reference to its 2D context.

```
var canvas = document.getElementById('canvas'),  
    context = canvas.getContext('2d');
```

While we can't refer to CSS classes to provide styling, the API does provide a variety of options.

- **fillStyle** – What background, or fill, color will be applied to the object?
- **fillRect** – Creates a rectangle, using the provided dimensions (**x**, **y**, **width**, and **height**), and fills it.

```
context.fillStyle = "#000";  
context.fillRect(200, 200, 200, 200);
```

This code translates to: "Create a rectangle that is both 200px wide and tall, and is offset 200px from the x (left) and y (top) origins. Also, fill that rectangle with the color specified, via `context.fillStyle`.

ROCK*

TIP

When specifying offsets, always remember that your starting point is the top-leftmost position of the canvas. Do note, however, that this origin can be changed if necessary.



Paths

Canvas doesn't provide shapes out of the box, other than a primitive rectangle (`rect`). This means that nearly all shapes will need to be created manually (or dynamically), using paths. To build a triangle:

```
(function() {  
    var canvas = document.getElementById('canvas'),  
        context = canvas.getContext('2d');  
  
    context.fillStyle = "#000";  
    context.strokeStyle = "#666";
```

```
context.lineWidth = 5.0;  
  
context.beginPath();  
context.moveTo(100,300);  
context.lineTo(300,300);  
context.lineTo(200,150);  
context.closePath();  
  
context.fill();  
context.stroke();  
})();
```



[View the triangle demo.](#)

This time, rather than calling `fillRect`, we're creating the shape of a triangle, stroke-by-stroke. The script begins by specifying the fill (background) color, and the width and color of the border (`lineWidth`).

```
context.fillStyle = "#000";  
context.strokeStyle = "#666";  
context.lineWidth = 5;
```

Next, the path is drawn.

```
context.beginPath();  
context.moveTo(100,300);  
context.lineTo(300,300);  
context.lineTo(200,150);  
context.closePath();
```

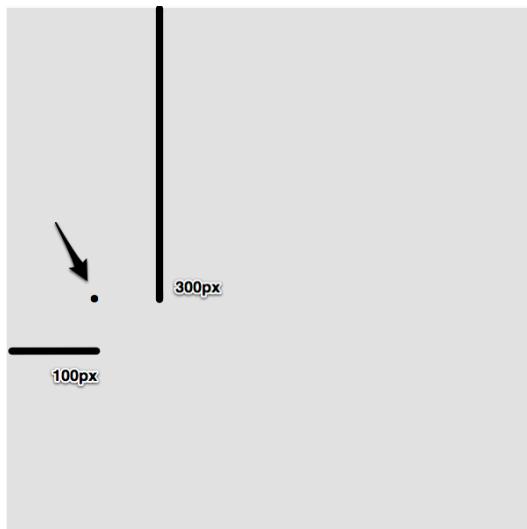
The difference between `moveTo` and `lineTo` is an important distinction to make. The easiest way to comprehend it is to think in terms of a paint brush on a canvas.

- `moveTo` – Don't draw (lift the paint brush); simply move to these coordinates (x and y).
- `lineTo` – Put the paint brush on the canvas, and draw a line from the current coordinates to the proposed one.

In the case of our code snippet:

```
context.moveTo(100,300);
```

Continuing with the painter analogy, before pushing a single bit of paint onto the canvas, the “hand” moves 100px from the x offset (left edge), and 300px from the y offset (top edge).



If it helps, think in terms of CSS and absolute positioning.

```
.hand {  
    position: absolute;
```

218

The Basics of Painting with Canvas

```
left: 100px;  
top: 300px;  
}
```

Now that the “hand” is at our desired coordinates, the brush should be placed on the canvas, and should draw a line from its current position to the new coordinates.

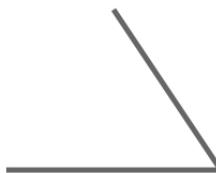
```
context.lineTo(300,300);
```

This creates a single line to 300px from both the left and top edges of the container.



We have the base of the triangle; now we need to create the final stroke, and “close” our shape with `context.closePath()`.

```
context.lineTo(200,150);
```



```
context.closePath();
```



If you’re working along, you may find that your canvas is still blank — even after drawing. Any fills or strokes will not take effect until the `fill()` and `stroke()` methods have been called, respectively.

```
context.fill();
context.stroke();
```



There we go!

Fills

We're not limited to single color fills. We can use a whole bunch of things.

Gradients

Rather than using, `context.fillStyle` to specify a solid color, we'll use `createLinearGradient` and build a gradient that transitions from yellow to red.

```
var bg = context.createLinearGradient(200, 150, 200, 300);
bg.addColorStop(0, "yellow");
bg.addColorStop(1, "red");
context.fillStyle = bg;
```



It's extremely important that the values you pass to `createLinearGradient` match up with the shape. Admittedly, it's not as intuitive as we might hope.

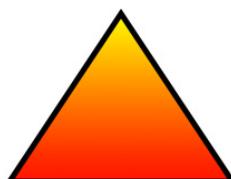
- **createLinearGradient** – Set a gradient’s path using the `x1` and `y1` coordinates as the starting point, and `x2` and `y2` coordinates as the ending point.
- **addColorStop** – Create a “color change” point.

```
var bg = context.createLinearGradient(200, 150, 200, 300);
```

These coordinates specify that the gradient should begin at 200px and 150px from the left and top edges, and end at 200px and 300px, respectively. Further, notice how the provided coordinates match up perfectly with the dimensions and positioning of the triangle.

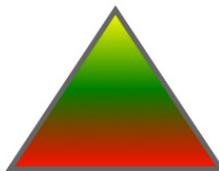
`createLinearGradient` won’t appear to do anything until we specify the colors for the gradient. So far, we’ve only specified the location. `addColorStop` is the method we require. In this case, a simple transition from yellow to red should suffice.

```
var bg = context.createLinearGradient(200, 150, 200, 300);
bg.addColorStop(0, "yellow");
bg.addColorStop(1, "red");
context.fillStyle = bg;
```



Should you require another color stop, update the values accordingly.

```
bg.addColorStop(0, "yellow");
bg.addColorStop(.5, "green");
bg.addColorStop(1, "red");
```



[View the color-stops demo.](#)

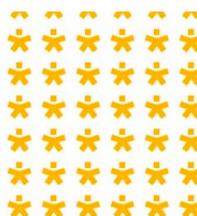
Images

Alternatively, images can be used as the fill. We only need to convert them to patterns.

```
<canvas width="230" height="250" id="canvas"></canvas>

<script> (function() {
    var canvas = document.getElementById('canvas'),
        context = canvas.getContext('2d'),
        bg = new Image();

    bg.src = "path/to/image.jpg";
    bg.onload = function() {
        var pattern = context.createPattern(bg, 'repeat');
        context.fillStyle = pattern;
        context.fillRect(30, 50, 200, 200);
    };
})();
</script>
```



[View the pattern demo.](#)

In this example, a new image is created, its source is applied, and when it has fully loaded into memory, it's converted to a pattern.

```
var pattern = context.createPattern(bg, 'repeat');
```

Once a pattern has been defined, it can be applied to `context.fillStyle` in the same way that a single color or gradient is.

```
context.fillStyle = pattern;
```

Curves

`bezierCurveTo` can be used to create... wait for it... Bézier curves. Photoshop masters will feel right at home here. Generally, they are used to create complex shapes, but as a first taste:

```
var canvas = document.getElementById('canvas'),
    context = canvas.getContext('2d');

context.fillStyle = "#000";
context.lineWidth = 4;
context.strokeStyle = "#666";

context.beginPath();

// cp = control point
// cp1x, cp1y, cp2x, cp2y, x, y
context.bezierCurveTo(100, 100, 300, 400, 500, 100);
context.closePath();

context.fill();
context.stroke();
```





[View the bezierCurveTo demo.](#)

When you think of Bézier curves, think of control points, or handles. What makes them considerably difficult to manage in the browser is the fact that we don't have visual access to these handles the way we normally would in a program like Photoshop.

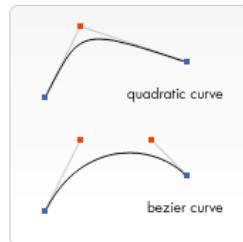


Image courtesy of the Mozilla Developer Network.

Arcs

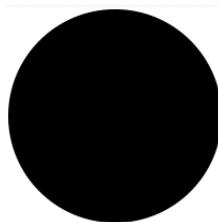
Now, in situations when we only need an arc (such as in the previous example) or circle, an easier solution is to use the `arc` method.

```
arc(x, y, radius, startAngle, endAngle, anticlockwise)
```

The following snippet will generate a perfect circle.

```
// get the canvas element using the DOM
var canvas = document.getElementById('canvas'),
    ctx = canvas.getContext('2d');

ctx.arc(100, // x coord
        100, // y coord
        100, // radius (or 1/2 the width and height)
        0, // begin point on circle
        Math.PI * 2, // end point
        true // clockwise
);
ctx.fill();
```



[View the arc demo.](#)

This next snippet will dynamically create 100 circles, and randomly place them on the canvas.

```
var canvas = document.      ▶  
    getElementById('canvas'),  
    ctx = canvas.getContext('2d'),  
    i = 0;  
  
ctx.globalCompositeOperation = ▶  
    "darker"; // the blend mode  
ctx.fillStyle = "green";  
  
for ( ; i < 100; i++ ) {  
    ctx.beginPath();  
    ctx.arc(Math.random() * 500, // x coord  
           Math.random() * 500,  
           30, // radius (or 1/2 the width and height)  
           0, // begin point on circle  
           Math.PI * 2, // end point  
           true // clockwise  
    );  
    ctx.closePath();  
    ctx.fill();  
}
```

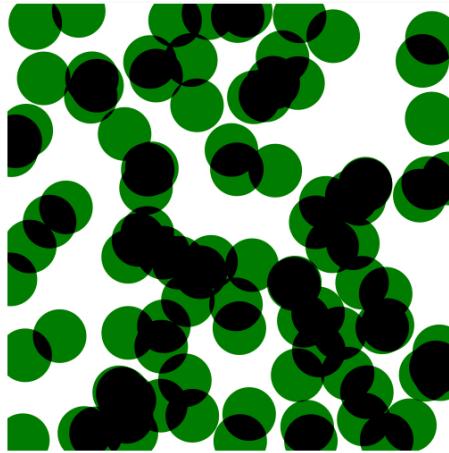
ROCK*

TIP

*Note: angles are measured in radians, not degrees, as you might be used to. To convert degrees to radians, use the following formula: (Math.PI/180)*degrees.*



[View the circles demo.](#)



Pay special attention to `globalCompositeOperation`; this essentially establishes a blend mode, which you might be familiar with in Photoshop.

Animations

This is the point when the painter analogy ends. If canvas was only capable of creating static image-like effects, it wouldn't be overly useful.

Hello Animation

The process of animating an object simply involves re-drawing the desired shape at regular intervals.

```
setInterval(function() {  
    // re-draw shape at new location  
}, 50);
```

It's important to remember, though, that, when you draw a new shape, the old one will still exist on the canvas. To accurately

animate an object, it first needs to be removed from the canvas, and then redrawn at the new coordinates.

```
setInterval(function() {  
    // remove shape  
    // re-draw shape at new location  
}, 50);
```

The following snippet will repeatedly transition a square from the top of the canvas to the bottom.

```
<canvas width="500" height="500" id="canvas"></canvas>  
  
<script>  
var canvas = document.getElementById('canvas'),  
    ctx = canvas.getContext('2d'),  
    i = 1;  
  
ctx.fillStyle = 'red';  
  
setInterval(function() {  
    // Clear the canvas  
    ctx.clearRect(0,0,canvas.width, canvas.height);  
  
    // Redraw square at the new location  
    ctx.fillRect(50, 50 + i++, 30, 30); // x, y, width, height  
  
    // if the square is off the canvas, reset i back to 1.  
    if ( i >= canvas.height ) i = 1;  
}, 10);  
</script>
```

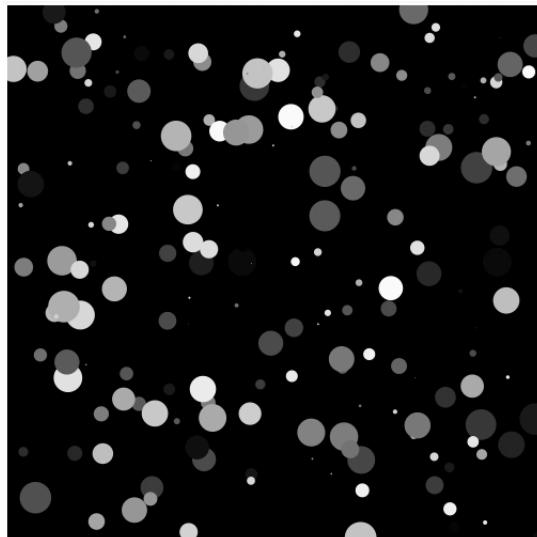


[View the hello animation demo.](#)

This effect is achieved due to `setInterval`, which executes every ten milliseconds. Each time the function is run, the y coordinates of the square will be updated (`i++`) by one pixel.

Particles

The particle demonstration is one of the first techniques you'll learn in any canvas book.



It's an excellent way to learn how shapes, widths, and speeds can be determined and manipulated dynamically.

The particle effect depends on a few key components: the positioning, radius, speed, and color must all be determined randomly. I'll let you fool around with the final code shortly, but, before I do, let's review a couple of the key functions:

```
var particles = [];  
  
function generateParticle() {  
    // We'll stick with a max of 200 particles  
    if ( particles.length >= 200 ) return;  
  
    // Each particle will be a shade of gray  
    var gray = Math.round( Math.random() * 255 );
```

```
// These values will be applied to the arc method
particles.push({
  x: Math.random() * canvas.width,
  y: 0, radius: Math.random() * 15,
  speed: Math.random() * 4,
  color: 'rgb(' + [gray, gray, gray].join(',') + ')'
});
}
```

This function will be called at regular intervals (with `setInterval`). Its only job is to push a new object to the `particles` array. This object will contain randomly generated values for the components listed in the previous paragraph.

Once the function has been called two hundred times (the max that this function allows), `particles` will contain two hundred unique objects called `particles`.

```
function draw() {
  // black background
  ctx.fillStyle = '#000';
  ctx.fillRect(0, 0, canvas.width, canvas.height);

  // filter through particles list and draw them
  for ( var i = 0; i < particles.length; i++ ) {
    var p = particles[i];

    ctx.beginPath();

    // If the position is off the canvas, reset.
    if ( p.y >= canvas.height ) p.y = 0;
    if ( p.x >= canvas.width ) p.x = 0;

    // Draw the circle using the saved dimensions from the >
    // particles array
    // set x, y, radius, begin, end:
    ctx.arc(p.x, p.y, p.radius, 0, Math.PI * 2);
  }
}
```

```
// Set the color of the particle  
ctx.fillStyle = p.color;  
  
ctx.closePath();  
ctx.fill();  
}  
}
```

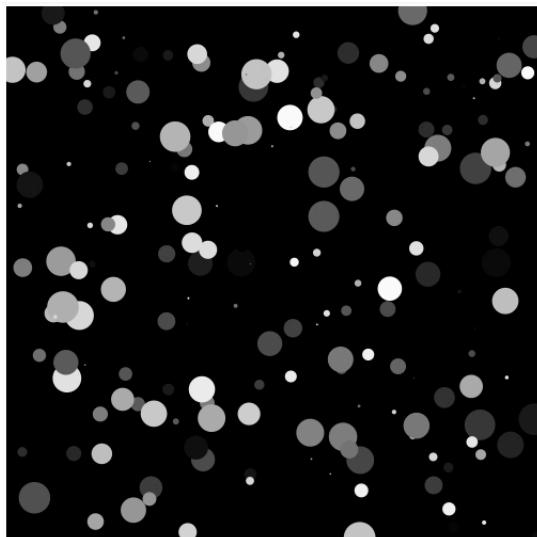
`draw()` will handle the process of “drawing” the particles on the canvas. Notice that we filter through all the particles, and create a new `arc`. Because the positioning and dimensions that are passed to the `arc` method were determined randomly, we can rest assured that the particle (or circle, essentially) will show up in a different location each time.

```
function updatePosition() {  
    // Update the y and x value positioning of the particle  
    // We update x to mimic a slight wind  
    for ( var i = 0; i < particles.length; i++ ) {  
        particles[i].y += particles[i].speed;  
        particles[i].x += .2;  
    }  
}
```

After the particles are drawn onto the canvas, `updatePosition` will be called. Its sole responsibility is to filter through the `particles` array, and update both its `y` and `x` coordinates.

This way, on the next iteration, when the particles are drawn again, the positioning for each will be in a slightly different location — thus, completing the illusion. (Yep — I just used, “thus”. Deal with it.)

Though not necessary, the `x`, or horizontal location of the particle is updated as well for the purposes of simulating a slight wind.



Final Particle Script

```
<canvas width="500" height="500" id="canvas"></canvas>

<script>
(function() {
    var canvas = document.getElementById('canvas'),
        ctx = canvas.getContext('2d'),
        particles = [];

    function generateParticle() {
        // We'll stick with a max of 200 particles
        if ( particles.length >= 200 ) return;

        // Each particle will be a shade of gray
        var gray = Math.round( Math.random() * 255 );

        // These values will be applied to the arc method
        particles.push({
            x: Math.random() * canvas.width,
            y: 0,
            radius: Math.random() * 15,
            speed: Math.random() * 4,
        });
    }

    generateParticle();
})()
</script>
```

```
        color: 'rgb(' + [gray, gray, gray].join(',') + ')'
    });
}

function draw() {
    // black background
    ctx.fillStyle = '#000';
    ctx.fillRect(0, 0, canvas.width, canvas.height);

    // filter through particles list and draw them
    for ( var i = 0; i < particles.length; i++) {
        var p = particles[i];

        ctx.beginPath();

        // If the position is off the canvas, reset.
        if ( p.y >= canvas.height ) p.y = 0;
        if ( p.x >= canvas.width ) p.x = 0;

        // Draw the circle using the saved dimensions from ▶
        // the particles array
        // set x, y, radius, begin, end:
        ctx.arc(p.x, p.y, p.radius, 0, Math.PI * 2);

        // Set the color of the particle
        ctx.fillStyle = p.color;

        ctx.closePath();
        ctx.fill();
    }
}

function updatePosition() {
    // Update the y and x value positioning of the particle
    // We update x to mimic a slight wind
    for ( var i = 0; i < particles.length; i++ ) {
        particles[i].y += particles[i].speed;
        particles[i].x += .2;
    }
}
```

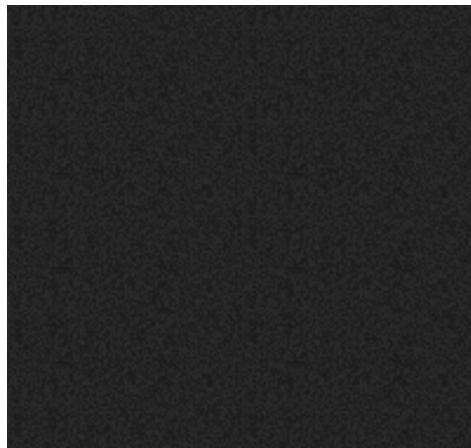
```
// Begin
setInterval(function() {
  generateParticle();
  draw();
  updatePosition();
}, 40);
})();
</script>
```



[View the particles demo.](#)

Generating Noise

This next one is mostly for fun. The goal is to dynamically generate background noise for a website, like this:



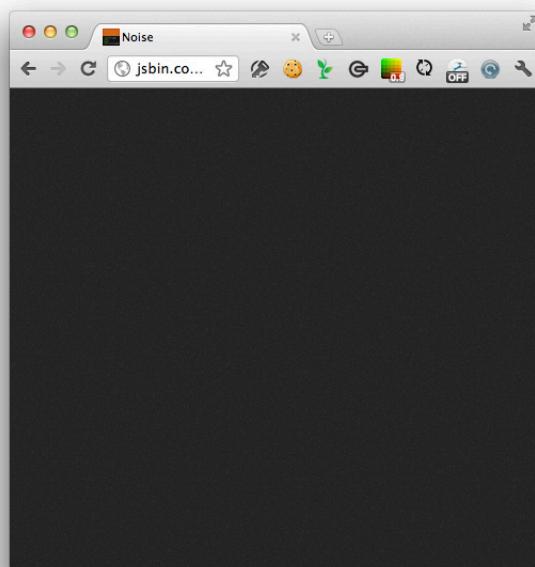
The key to this effect is in the realization that noise is random. With that in mind, we only need to generate a random gray-scale color pixel-by-pixel.

```
// Determine a random color...
var number = Math.floor( Math.random() * 60 );
```

```
// And set the fillStyle for that single pixel to the      ▶  
    generated color  
    // Will generate - rgba(43, 43, 43, .5);  
    ctx.fillStyle = 'rgba(' + [number, number, number,  
        opacity].join(',') + ')';  
  
    // Create the 1px rectangle  
    ctx.fillRect(x, y, 1, 1);
```

Once a decent-sized square, full of randomly colored pixels, has been generated, it can then easily be converted to a DataURL, and applied to the background of the **body** element, as desired!

```
document.body.style.backgroundImage = "url(" + canvas.      ▶  
    toDataURL("image/png") + ")";
```



Final Noise Script

```
<!DOCTYPE html>
<html>
<head>
<meta charset=utf-8>
<title>Noise</title>
<style> body { background: #292929; } </style>
</head>
<body>

<script>
function generateNoise(opacity) {
  if ( !!!document.createElement('canvas').getContext ) {
    return false;
  }

  // For demo purposes, this time, we're creating the
  // canvas element dynamically
  var canvas = document.createElement("canvas"),
    ctx = canvas.getContext('2d'),
    x,
    y,
    number,
    opacity = opacity || .2;

  canvas.width = 100;
  canvas.height = 50;

  // Filter through every pixel in the canvas...
  for ( x = 0; x < canvas.width; x++ ) {
    for ( y = 0; y < canvas.height; y++ ) {
      // Determine a random color...
      number = Math.floor( Math.random() * 60 );

      // And set the fillStyle for that single pixel to      ▶
      // the generated color
      ctx.fillStyle = 'rgba(' + [number, number, number,      ▶
        opacity].join(',') + ')';
    }
  }
}

generateNoise();
</script>
```

```
// Create the 1px rectangle
ctx.fillRect(x, y, 1, 1);
}

// Convert the contents of the canvas to a dataURL,
// and set it as the background of the body
document.body.style.backgroundImage = "url(" + canvas.►
    toDataURL("image/png") + ")";
}

generateNoise();

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



[View the noise demo.](#)

101 Class Complete

That's the most I'm going to cover in this book. Honestly, we've barely scratched the surface of what's possible; this chapter merely outlined the basic concepts behind working with the API.

Further Reading

Should you desire to take your canvas skills to the next level, the following resources are vital:

- [HTML5 Specification](#) – The spec should always be your first step when learning a new API.
- [Canvas from Scratch](#) – A four-part session that introduces the languages, and provides an overview on gaming implementations for canvas.

- [Improving HTML5 Canvas Performance](#) – HTML5 Rocks provides some of the best tutorials on the web, when it comes to working with HTML5.
- [*HTML5 Canvas*](#) (book)
- [*Foundation HTML5 Canvas*](#) (book)
- [*The HTML5 Canvas Cookbook*](#) (book)



Don't Irritate Visitors — Use Web Storage

Browser Support



Think about it: how many times have you found yourself writing a long comment on some blog posting, when, suddenly, your internet connection fumbles. Once you manage to get back online, you reload the page, see the empty form, and find yourself thinking (or saying), “Son of a b\$%!(h!” Yes, we’ve all been there. Usually the rewritten comment isn’t quite so long!

Or maybe you’re filling out a form on a site like Craigslist. While waiting for your photos to upload, their server times out, and, again, you’re back at square one. The truth is that this should never happen. With HTML5, it’s incredibly easy to save state so that events like this are impossible.

In the past few years, multiple specs have been competing for the “web database” spotlight: Web SQL, IndexDB, Local/Session Storage, etc. Web SQL is no longer being developed or maintained, and IndexDB, though promising and powerful, doesn’t yet have enough browser support (Internet Explorer coming on board with version 10) to warrant its usage.

We’ll instead focus on local and session storage, which are luckily very similar!

What is Local Storage?

You'll be happy to hear that local storage is incredibly easy to understand. It allows us to store key-value pairs for future use. Unlike sessions or cookies, local storage is considered by the browser to be long term. It's as plain and simple as that. By implementing this natively in the browser, we now have access to an easy-to-use, standardized API.

Is This The Same As Cookies?

No, it not the same. The basic idea is similar, but it ends at that.

- Cookies are pushed to the server for every request. Especially for larger data sets, this could present some potential performance issues. Local storage does not.
- Cookies may not exceed 4KB worth of data. Local storage is not burdened by this limitation. You can generally count on 5MB worth of storage, which is significant.
- Cookies have an expiration date; local storage keys will exist indefinitely, or until the data is deleted.

ROCK* TECH TERM

What's a cookie? It's a small piece of data that can be sent through an HTTP request to the server. This allows us to maintain state across page requests. There is a JavaScript API for accessing and editing them, but it's not nearly as intuitive and easy to use as local storage.



I Still Don't Get It...

Still struggling to figure out what the advantage to local storage is? Even if you close and reopen the browser, the data that you've saved will still be available! We're not dealing with temporary sessions here. This data can exist for an indefinite period of time!

It's vital that you understand that local storage data is **user-specific**. This data is stored on the user's computer, and, as a result, cannot be accessed elsewhere. Always think of local storage as a convenience for the user; not a convenience for yourself.

Is Local Storage HTML5?

Technically, no. It fits in the **friends** category — similar to your relationship with that girl in high school who began referring to you as *buddy*, ultimately ruining any chance of a romantic relationship. But I digress. It was part of the core HTML5 spec at one time, but due to various political reasons, it was eventually exported to its own specification, called [Web Storage](#). Nonetheless, it's commonly grouped under the term HTML5.

What's the Difference Between Web Storage and Local Storage?

Nothing at all. They're simply different ways to refer to the same specification. That said, do note that there are two forms of web storage: local and session. The APIs are nearly identical, with the only difference being that session storage is limited to the lifetime of the window.

How Do I Use It?

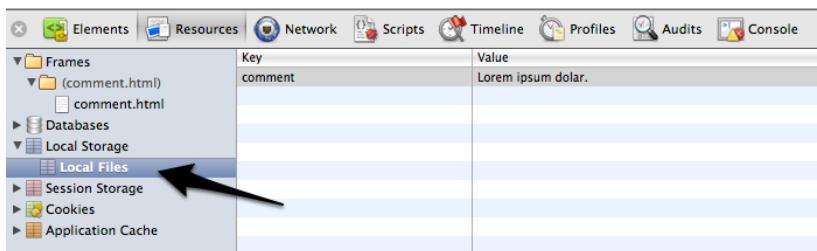
Browser support for local storage is quite good, you'll find. Even Internet Explorer 8 supports it!

Setting

We can set a key for future use, like so:

```
localStorage.setItem('comment', 'Lorem ipsum dolor.');
```

Try it out for yourself. Create a new HTML file, and paste the line above within `script` tags. Next, view the page in Chrome, open Chrome Developer Tools, and choose the *Resources* tab. You should now see your key within the *Local Storage* panel. That was easy!



The screenshot shows the Chrome Developer Tools interface with the 'Resources' tab selected. On the left, the 'Local Storage' section is expanded, revealing a table with one entry: 'comment' (Key) with value 'Lorem ipsum dolor.' (Value). An arrow points to the 'Local Storage' heading.

	Key	Value
	comment	Lorem ipsum dolor.

Getting

Next, we need a way to capture this value upon future page requests. As you might have guessed, we can use the `getItem` method for this purpose.

```
localStorage.getItem('comment'); // Lorem ipsum dolor.
```

Deleting

What if we instead need to delete the key entirely?

```
localStorage.removeItem('comment');
```

We could also use the following syntax to achieve the same result.

```
delete localStorage.comment;
```

Notice something a bit odd about that last snippet? We can alternatively reference these keys in the same way that we would call methods on an object. Rather than `localStorage.getItem('comment')`, we can also do: `localStorage.comment`.

The reason why this is possible is because, behind the scenes, `getItem`, `setItem`, and `removeItem` are *getters*, *setters*, and *deleters*, respectively.

This means that the following lines all achieve the same outcome:

```
// Set
localStorage.setItem('test', 'some val');
localStorage.test = 'some val';
localStorage['test'] = 'some val';

// Get
var test = localStorage.getItem('test'); // some val
var test = localStorage.test; // some val
var test = localStorage['test']; // some val

// Delete
localStorage.removeItem('test');
localStorage.test = null;
localStorage['test'] = null;
delete localStorage.test;
delete localStorage['test'];
```

Now if we'd instead prefer to clear all local storage key value pairs, we can use the `clear()` method.

```
localStorage.clear();
```

Testing for Support

Remember: we can't blindly assume that the user's browser supports local storage. If they're browsing in, for example, Internet Explorer 7, the use of the `localStorage` object will throw an error. Instead, we should always test for support before moving forward.

We can test for local storage support by using the following snippet:

```
var supportsStorage = (function() { try {  
    return 'localStorage' in window && window['localStorage']>  
        !== null;  
} catch(e) {  
    return false;  
}  
})();
```

It's unfortunately a tad more complicated than we would like. Ideally, we could do `if (localStorage) ...`, but, due to various browser quirks and potential errors, we need to run the code through a `try/catch` block. That way, if the browser does not support local storage, or an odd error is thrown in the process, our `supportsStorage` variable will be equal to `false`.

```
if ( supportsStorage ) { ... }
```

Alternatively, I will once again recommend that you use [Modernizr](#) to perform feature testing. Its entire reason for existing is to handle these sorts of tasks! With Modernizr, we'd instead write:

```
if ( Modernizr.localstorage ) { ... }
```

Test What You've Learned— Comment Form

To test our newly learned skill, let's write a script that will save a user's comment in local storage. This way, if the internet cuts out, or the browser crashes, they don't lose their comment.

Markup

We begin, as always, with a bit of markup. We won't simulate a complete comment form. For simplicity's sake, we'll stick with the

textarea and **submit** button. If you're working along (I hope you are), paste the following into your editor:

```
<!doctype html>
<html>
<head>
<title>Comments and Local Storage</title>
<style>
label { display: block;}
textarea { width: 300px; height: 150px; }
</style>
</head>
<body>

<form>
<label for="comment">Your Comment?</label>
<textarea name="comment" id="comment"></textarea><br>

<input type="submit">
</form>

<script src="https://ajax.googleapis.com/ajax/libs/jquery/ ►
1.7.0/jquery.min.js"></script>

</body>
</html>
```

Your Comment?

Submit

Okay, okay; it won't win any awards. That's not our goal here. Sheesh, get off my case with this design stuff!

Minus the server-side processing, this represents a typical comment **form**. If we were to write a long comment, and accidentally close the window, we'd lose it all. This must be remedied!

The Game Plan

I generally find that I'm most efficient if I first chart out exactly what I need to accomplish.

1. I need to detect whether the browser supports local storage.
2. While the user is writing in the **textarea**, every five seconds, I want to save the value of the **textarea** to local storage.
3. When the page subsequently loads, if there is local storage available, I should set the default **value** of the **textarea** equal to its value.

That sounds easy enough! Let's knock these out one at a time.

1. I need to detect whether the browser supports local storage.

As we learned earlier in this chapter, we can test for local storage support by using the following snippet:

```
var supportsStorage = function() {
  try {
    return 'localStorage' in window && window.localStorage >
      !== null;
  } catch(e) {
    return false;
  }
};
```

Alternatively, we can use Modernizr:

```
Modernizr.localstorage
```

2. While the user is writing into the textarea, every five seconds, I want to save the value of the textarea to local storage.

Hmm... sounds like we'll need to first grab the **textarea** from the DOM, listen for when it's focused, and then set an interval.

```
if ( supportsStorage ) {  
    var comment = document.getElementById('#comment'),  
        updateInterval;  
  
    // Listen for when the textarea is focused  
    comment.addEventListener('focus', function() {  
        // Every 5 seconds, update the localStorage value.  
        updateInterval = setInterval(function() {  
            localStorage.comment = comment.value  
        }, 5000);  
    }, false);  
}
```

That should do the trick.

- We first query the DOM for the **textarea** with an **id** of **comment**.
- Next, we add a **focus** event listener to the **textarea**.
- Finally, we set a five second interval that updates local storage with the **value** of the comment.

There's one problem here, though. While the user is focused on the **textarea**, yes, the interval is running. However, even when they click off of the **textarea**, the interval continues on. It seems wasteful to repeatedly update local storage with the exact same

value, indefinitely. Instead, we'll also apply a `blur` event listener, and then use `clearInterval` to cancel it.

```
if ( supportsStorage ) {  
    var comment = document.getElementById('#comment'),  
        updateInterval;  
  
    // Listen for when the textarea is focused  
    comment.addEventListener('focus', function() {  
        // Every 5 seconds, update the localStorage value.  
        updateInterval = setInterval(function() {  
            localStorage.comment = comment.value  
        }, 5000);  
    }, false);  
  
    // Also listen for when the user leaves the textarea,  
    // and cancel the interval  
    comment.addEventListener('blur', function() {  
        clearInterval(updateInterval);  
    }, false);  
}
```

3. When the page subsequently loads, if there is local storage available, I should set the default value of the textarea equal to its value.

We can determine whether there is a `comment` key in the user's local storage, like so:

```
if ( localStorage.comment )
```

If the `comment` key does not exist, `null` will be returned, in which case the condition will equal `false`. If the condition is `true`, we should set the default `value` of the `textarea` to whatever is stored in local storage.

```
var comment = document.getElementById('#comment');
```

```
if ( localStorage.comment ) {  
    comment.value = localStorage.comment  
}
```

If you prefer, this can be shortened to a *one-liner*.

Vanilla JavaScript

```
var comment = document.getElementById('#comment');  
!!localStorage.comment && (comment.value = localStorage.►  
comment);
```

Final Script

As you can see, it's really not too difficult! Here's our final script:

JavaScript

```
(function() {  
    var supportsStorage =  
        (function() {  
            try {  
                return 'localStorage' in window && window.►  
                    localStorage !== null;  
            } catch(e) {  
                return false;  
            }  
        })(),  
        comment,  
        updateInterval;  
  
    if ( supportsStorage ) {  
        comment = document.getElementById('#comment');  
  
        !!localStorage.comment && ( comment.value = ►  
            localStorage.comment );
```

```
comment.addEventListener('focus', function() {
    updateInterval = setInterval(function() {
        localStorage.comment = comment.value
    }, 5000);
}, false);

comment.addEventListener('blur', function() {
    clearInterval(updateInterval);
}, false);
})();
```

jQuery

```
(function() {
    var supportsStorage =
        (function() {
            try {
                return 'localStorage' in window && window.
                    localStorage !== null;
            } catch(e) {
                return false;
            }
        })(),
        comment,
        updateInterval;

    if ( supportsStorage ) {
        comment = $('#comment');

        !!localStorage.comment && ( comment.val(localStorage.
            comment) );

        comment.on({
            'focus': function() {
                updateInterval = setInterval(function() {
                    localStorage.comment = comment.val();
                }, 5000);
            }
        });
    }
});
```

```
    },
    'blur': function() {
      clearInterval(updateInterval);
    }
  );
}
})();
```

Try it out. Begin writing a comment, and after fifteen seconds or so, close the browser, reopen, and return to the page. You should now find that your comment is still there! How helpful!



[View the local storage comments demo.](#)

How to Publish, or Announce Changes

One of the neatest things about local storage is that its API provides us with the means to announce changes. For instance, imagine that you're creating a to-do list (which we'll build together shortly) with local storage. It makes sense that you could potentially have two tabs open at the same time, both on your to-do list page. The Web Storage API gives us the ability to detect changes — such as adding, updating, or deleting a task — and then **announce** those changes to anything that's interested. With this functionality available, we could quite easily update the second tab's to-do list without even visiting or refreshing the page. Neat, huh?

The storage Event

The **storage** event will fire every time the **setItem**, **removeItem**, and **clear** methods are triggered. However, it's smart enough to

know that it should only fire if the data was actually modified. Let's set up an event listener.

JavaScript

```
window.addEventListener('storage', function(e) {  
    console.log(e);  
}, false);
```

jQuery

```
$(window).on('storage', function(e) {  
    console.log(e);  
});
```

The `e`, or *event object* that is passed to the callback function will contain all pertinent information related to the change.

- **key** – The name of the key that was modified
- **oldValue** – The original value of the key
- **newValue** – The new value
- **url** – The URL of the page that triggered the `storage` event

Try it Out

Create a new HTML file.

```
<!doctype html>  
<html>  
<head>  
    <meta charset="utf-8">  
    <title>Storage</title>  
</head>  
<body>
```

```
<script src="https://ajax.googleapis.com/ajax/libs/jquery/ ▶
  1.7.0/jquery.min.js"></script>
</body>
</html>
```

Next, we need a way to specify a value that will be saved to local storage. We'll use a simple form input.

```
<!doctype html>
<html>
<head>
  <meta charset="utf-8">
  <title>Storage</title>
</head>
<body>
<form>
  <label for="name">Name: </label>
  <input type="input" id="name" name="name">
</form>
<script src="https://ajax.googleapis.com/ajax/libs/jquery/ ▶
  1.7.0/jquery.min.js"></script>
</body>
</html>
```

Default Value

When the page initially loads, we should first determine whether or not a local storage **key** is available. We'll call ours **name**.

```
(function() {
  var supportsStorage =
    (function() {
      try {
        return 'localStorage' in window && window[ ▶
          'localStorage'] !== null;
      } catch(e) {
```

```
        return false;
    }
})(),
name;

if ( supportsStorage ) {
    name = document.getElementById('name');

    name.value = localStorage.name || ''; // set the value ▶
        of the input equal to the key, or nothing.
}
})();
```

Next, we can listen for when the user updates this `input` by using the `blur` event type.

```
(function() {
    var supportsStorage =
        (function() {
            try {
                return 'localStorage' in window && window[▶
                    'localStorage'] !== null;
            } catch(e) {
                return false;
            }
        })(),
    name;

    if ( supportsStorage ) {
        name = document.getElementById('name');

        name.value = localStorage.name || ''; // set the value ▶
            of the input equal to the key, or nothing.

        name.addEventListener('blur', function() {
            if ( this.value ) {
                localStorage.name = this.value;
            }
        })
    }
})();
```

```
    }, false);
}
})();
```

This new section of our code will listen for when the user tabs off of the `input`, and then — as long as the `input`'s value is not blank — save it to a local storage key called `name`.

Lastly, we'll listen for when the local storage key is changed, and make an **announcement** accordingly. This way, other tabs can auto-update. If the `key` was in fact updated, we'll update the value of the `input`, accordingly.

```
(function() {
  var supportsStorage =
    (function() {
      try {
        return 'localStorage' in window && window[
          'localStorage'] !== null;
      } catch(e) {
        return false;
      }
    })(),
    name;

  if ( supportsStorage ) {
    name = document.getElementById('name');

    name.value = localStorage.name || '';►
    // set the value of the input equal to the key, or nothing.

    name.addEventListener('blur', function() {
      if ( this.value ) {
        localStorage.name = this.value;
      }
    }, false);
```

```
window.addEventListener('storage', function() {  
    name.value = localStorage.name;  
}, false);  
})();
```

That should do it! Open this page in two browser tabs, and play around with changing the values!



[View the `localStorage` announcement demo.](#)

Project 2—Tasks

Let's work through one more demo together. We'll build the seemingly obligatory to-do list with local storage. Don't worry; we won't dive in too deep. I simply want to make sure that you grok the concepts behind local storage.

Markup

Of course, we begin with our markup. This time, because it's only for ourselves, we'll let it all hang out. Who needs `head` and `body` tags? (*</snark>*)

```
<!doctype html public 'urination is creepy'>  
<meta charset=utf-8>  
<title> My Local Storage To-Do List </title>  
<h1> My Tasks </h1>  
<ul class=tasks>  
  <li> Add a task  
</ul>
```

```
<!DOCTYPE html PUBLIC "urination is creepy">  
▼<html>  
  ▼<head>  
    <meta charset="utf-8">  
    <title> My Local Storage To-Do List </title>  
  </head>  
  ▼<body>  
    <h1> My Tasks </h1>  
    ▶<ul class="tasks" contenteditable>..</ul>  
  </body>  
</html>
```

Remember: for simple sites like this, we don't really **need** to insert the `html`, `head`, and `body` tags. I'm not suggesting that you omit them in your everyday projects, but, for simple demos and prototypes like this, save yourself the writing! The browser will tack it on for you.

Content Editable

Next, the to-do list should be editable from directly within the browser. We'll use a new helpful attribute, `contenteditable`, to handle this task.

```
<ul class=task contenteditable>
```

The `contenteditable` attribute allows the text content of the element it is assigned to to be modified in the browser.



[View an example of contenteditable in action.](#)

JavaScript

Next, we'll get started on the local storage. As always, we should chart out our game plan.

1. Fetch the task list from the DOM, and save it for future use.
2. When the user creates a new task and tabs off, we should set or update a local storage key.
3. When a change occurs, make a behind-the-scenes **announcement!**
4. When the page loads, if local storage exists, use it to populate the task list.

Let's begin.

1. Fetch our task list from the DOM, and save it for future use.

Let's create a `Task` object that will contain all of our required properties and methods. This is a clean and simple way to organize related code.

```
(function() {  
    var Task = {  
        };  
    })();
```

Next, we'll create an `init` method that gets the ball rolling, so to speak.

```
(function() {  
    var Task = {  
        init : function() { }  
    };  
  
    // Here we gooooo! (Mario voice)  
    Task.init();  
})();
```

We know that we'll need to work with the task list; let's query the DOM, and store it within a `property` called `container`.

```
(function() {  
    var Task = {  
        // supported in IE8+:  
        container : document.querySelector('.tasks'),  
        init : function() { }  
    };  
  
    // Here we gooooo! (Mario voice)  
    Task.init();  
})();
```

2. When the user creates a new task and tabs off, we should set or update a local storage key.

Sounds like we should use the `blur` event.

JavaScript

```
(function() {
  var Task = {
    container : document.querySelector('.tasks'),
    init : function() {
      this.container.addEventListener('blur', this.▶
        saveToStorage, false);
    },
    saveToStorage : function() {
      // update local storage
    }
  };
  // Here we gooooo! (Mario voice)
  Task.init();
})();
```

With the newly added lines above, whenever the user clicks on the unordered list, and tabs or clicks off (or `blur`), the `saveToStorage` method will be called. Let's now update local storage.

```
(function() {
  var Task = {
    container : document.querySelector('.tasks'),
    init : function() {
      this.container.addEventListener('blur', this.▶
        saveToStorage, false);
    },
    saveToStorage : function() {
      localStorage.tasks = this.innerHTML;
    }
})();
```

```
        }
    };

    // Here we gooooo! (Mario voice)
    Task.init();
})();
```

In the `saveToStorage` method, `this` refers to the element that was *blurred*. In our case, that will be the unordered list. As such, we can capture all of the tasks, or list items, and reference the HTML with `this.innerHTML()`.

3. When a change occurs, make a behind-the-scenes announcement!

We'll use the `storage` event to handle the task of announcing any relevant changes.

```
(function() {
    var Task = {
        container : document.querySelector('.tasks'),
        init : function() {
            this.container.addEventListener('blur', this.
                saveToStorage, false);
        },
        saveToStorage : function() {
            localStorage.tasks = this.innerHTML;
        },
        announce : function() {
            window.addEventListener('storage', function() {
                Task.applyStorage(); // to be created
            }, false);
        }
    };
});
```

```
// Here we gooooo! (Mario voice)
Task.init();
})();
```

The `announce` method specifies that, when a change occurs, we should call a yet-to-be-created method called `applyStorage`.

4. When the page loads, if local storage exists, use it to populate the task list.

When the page subsequently loads, we need to determine whether local storage for the task list exists. If it does, we should update the list accordingly. We'll build that `applyStorage` method that we referenced in step three.

```
(function() {
  var Task = {
    container : document.querySelector('.tasks'),

    init : function() {
      this.container.addEventListener('blur', this.▶
        saveToStorage, false);
    },

    saveToStorage : function() {
      localStorage.tasks = this.innerHTML;
    },

    applyStorage : function() {
      if (localStorage.tasks) {
        Task.container.innerHTML = localStorage.tasks;
      }
    }

  announce : function() {
    window.addEventListener('storage', function() {
      Task.applyStorage();
    }, false);
```

```
        }
    };

// Here we gooooo! (Mario voice)
Task.init();
})();
```

The `applyStorage` method is fairly self-explanatory. When the page loads, we determine whether `localStorage.tasks` exists. If data is available, we need to populate the unordered list, with `this.container.html(localStorage.tasks)`.

Local Storage Support

The final step is to check for local storage support before running a single line of code. There's no need to waste time if the user is in Internet Explorer 7! Here's our final script! As with the first project, we'll use the `supportsStorage` variable that we created at the beginning of the chapter.

```
(function() {
    var supportsStorage =
        (function() {
            try {
                return 'localStorage' in window && window[
                    'localStorage'] !== null;
            } catch(e) {
                return false;
            }
        })();

    if ( supportsStorage ) {
        var Task = {
            container : document.querySelector('.tasks'),
            init : function() {
                this.applyStorage();
            }
        };
        Task.init();
    }
});
```

```
        this.container.addEventListener('blur', this.  
            saveToStorage, false);  
    }  
    this.announce();  
},  
  
saveToStorage : function() {  
    console.log('saving');  
    localStorage.tasks = this.innerHTML;  
},  
  
applyStorage : function() {  
    console.log('apply');  
    if (localStorage.tasks) {  
        Task.container.innerHTML = localStorage.tasks;  
    }  
},  
  
announce : function() {  
    window.addEventListener('storage', function() {  
        console.log('announce');  
        Task.applyStorage();  
    }, false);  
}  
};  
  
// Here we gooooo! (Mario voice)  
Task.init();  
}  
})();
```



[View the to-do list demo.](#)

Form Data

As developers become acquainted with `localStorage`, one of the first use-cases that may come to their mind is applying it to form

data. Given a few form `inputs`, it wouldn't be uncommon for a developer to assign each `input` to its own `localStorage` key:

```
// Don't do
var first = document.querySelector('#nameInput'),
    last = document.querySelector('#lastInput');
if ( Modernizr.localstorage ) {
    localStorage.first = first.value;
    localStorage.last = last.value
}
```

While the world won't end if you choose this route, there are more optimized ways to handle form data binding. Do we really need to create potentially dozens of `localStorage` keys? Instead, let's serialize the form data, and save that big chunk to a single key.

jQuery's `Serialize()` Method

jQuery offers a helpful method, called (wait for it...) `serialize()`. The `.serialize()` method creates a text string in standard URL-encoded notation. It operates on a jQuery object representing a set of form elements. If we use this method on a given `form`, it will translate those values into key-value pairs.

```
($('form').serialize());
```

... will yield something along the lines of:

```
first=jeffrey&last=way&home=chattanooga+tennessee
```

Helpful, isn't it? We can now save this serialized string to local storage.

```
localStorage.formData = $('form').serialize();
```

Deserialize

jQuery currently doesn't offer any sort of `deserialize` method, but we can handle the task quite easily. It only takes a few steps:

1. Determine if a local storage key is available for the form.
2. Serialize the data, and bind it to the form inputs.
3. Save the form data to storage while being filled out.

Assuming the following chunk of HTML...

```
<!doctype html>
<html>
<head>
  <meta charset=utf-8>
  <title>Form Data</title>
</head>
<body>

<form>
  <input name="title">
  <input name="name">
  <textarea name="comment"></textarea>
  <input type="checkbox" name="someCheck">
  <input type="radio" name="someRadio">

  <input type="submit">
</form>

</body>
</html>
```

... when the page loads, we'll knock out task number one.

1. Determine if a local storage key is available for the form.

```
(function() {  
  if ( Modernizr.localstorage && localStorage.formData ) {  
  }  
})();
```

Remember: don't rely on `localStorage` being available. Always perform feature detection before working with it.

2. Split the string into chunks.

Now, we'll capture the serialized string, and translate it into something we can work with.

```
(function () {  
  var form, serialized, sp;  
  
  if (localStorage.formData) {  
    form = document.querySelector('form');  
  
    // title=my+title&name=jeff+way&comment=what+about+  
    // quest&someCheck=on&someRadio=on  
  
    serialized = localStorage.formData;  
    // First, "deserialize"  
    // ["title=my+title", "name=jeff+way", "comment=what+  
    // about+quest", "someCheck=on", "someRadio=on"]  
    sp = serialized.split('&');  
  
    // Filter over this new array  
    $.each(sp, function (i, pair) {  
      // i = index, pair = title=my+title  
      // ["title", "my+title"]  
      pair = pair.split('=');  
  
      // Are we dealing with a radio or checkbox? If so,  
      // we need to apply the checked attribute
```

```
if (pair[1] === 'on') {
    form[pair[0]].checked = true; // look for the input>
    with a name equal to pair[0]
} else {
    // Otherwise, it's an input or text area.
    form[pair[0]].value = unescape(pair[1]).replace(/\+/g, ' ');
}
});
```

By using the `split()` function a couple times, we can turn that long, serialized string into key-value pairs that we can work with. Then, we filter over them, and query the DOM for the applicable form elements, and reattach the applicable data.

3. Save the form data to storage while being filled out.

Lastly, we only need to create the necessary serialized data. We'll simply listen for when the `form` is submitted, and then serialize and save the data to `localStorage`. In your project though, you'd likely opt to use the techniques we learned about in the previous section of this chapter. Set an interval, and, every five seconds or so, save the form data.

Let's keep it simple for this demo, as we're focusing more on the process of binding existing `localStorage` data to the form.

```
// Serialize the form data
$(form).submit(function (e) {
    localStorage.formData = $(this).serialize();
    alert('Okay, refresh the page now.');
    // just for demo
    e.preventDefault(); // disable the default action of the >
    form's submission
});
```

And that does it. Pretty simple, huh?



[View the form data demo.](#)

Storing Objects

Local storage is limited to key-value pairs; but, what if we need to save an object of data? Is that possible? Yes — but it requires a bit of trickery.

The key is to convert the object to a string before saving it to local storage. Then, when you wish to re-capture that data, it should be transformed back into an object. In browsers which support JSON natively, we can use the `JSON.stringify` and `JSON.parse` methods.

Convert an Object to a String

We'll use the following object for testing purposes.

```
var person = {  
    firstName : 'Douglas',  
    realFirstName : 'Hauser'  
};
```

To convert this object to a string, we use `JSON.stringify`.

```
JSON.stringify(person);  
// '{"firstName":"Douglas","realFirstName":"Hauser"}'
```

This is handy; now that we have a string, we can save it to local storage, per usual.

```
var person = {  
    firstName : 'Douglas',  
    realFirstName : 'Hauser'  
}, str;
```

```
if ( supportsStorage ) {  
    str = JSON.stringify(person);  
    // '{"firstName":"Douglas","realFirstName":"Hauser"}'  
    localStorage.person = str;  
}
```

Retrieval

Upon subsequent page loads, this data can be transformed back into an object by conversely using `JSON.parse`.

```
var person;  
if ( supportsStorage && localStorage.person ) {  
    person = JSON.parse( localStorage.person );  
    console.log(person.firstName); // Douglas  
}
```

localStorage.setObj ?

If you anticipate saving JavaScript objects to local storage on a regular basis, you might consider abstracting this code away to its own function. You could even modify the native `Storage` object itself — that is, if you're the type who dances with the devil in the pale moon light.

```
Storage.prototype.setObj = function(key, val) {  
    this.setItem( key, JSON.stringify(val) )  
};  
  
Storage.prototype.getObj = function(key) {  
    return JSON.parse( this.getItem( key ) );  
};
```

Be careful extending native objects, but, if you do, you can now use `setObj` in the same way that you would use `setItem`.

```
var person = {  
    firstName : 'Douglas',
```

```
    realFirstName : 'Hauser'  
};  
  
localStorage.setObj('somePerson', person);  
// '{"firstName":"Douglas","realFirstName":"Hauser"}'  
  
localStorage.getObj('somePerson').firstName; // Douglas
```

To verify this, run `console.dir(Storage);`.

```
> dir(Storage);  
  ▼ function Storage() { [native code] }  
    arguments: null  
    caller: null  
    length: 0  
    name: "Storage"  
    ▼ prototype: Storage  
      ► clear: function clear() { [native code] }  
      ► constructor: function Storage() { [native code] }  
      ► getItem: function getItem() { [native code] }  
      ► getObj: function (key) {  
        ► key: function key() { [native code] }  
        ► removeItem: function removeItem() { [native code] }  
        ► setItem: function setItem() { [native code] }  
        ► setObj: function (key, val) {  
          ► __proto__: Object  
        ► toString: function toString() { [native code] }  
        ► __proto__: Object
```

Notice that, because we've extended this object, `setObj` and `getObj` now show up in the `Storage.prototype` object. As I mentioned earlier, be very careful when doing such things. It's largely considered to be a risky, or even bad practice. As with everything, know the rules, so that you can break them.

Why `Storage` and not `localStorage`? `localStorage` is meant for storing information related to the actual content that's been saved. It extends the `Storage` object, which is what houses the core methods, like `getItem`, `removeItem`, etc.

Summary

With a solid understanding of the Web Storage API under your belt, when you're ready, it's time to move on the History API.

2

The History API

Browser Support



One of the earliest criticisms of AJAX was that it made it difficult to save state. For example, if we asynchronously pull in some set of data when a link is clicked, the user will not have the ability to link specifically to that state of the page. This has long since been fixed by clever developers, however, now, with the updated [HTML5 History API](#), we can rely on a native solution to remedy this issue — at least within the browsers which support it. This API allows us dynamically set and edit records in the browser's history.

Still confused? Here's the easy definition: the History API gives you the ability to press the “Back” or “Forward” button, and revert to previous states on the same page.

The history Interface

The following properties and methods are available, via the `history` object.

```
interface History {  
    readonly attribute long length;  
    readonly attribute any state;  
    void go(optional long delta);  
    void back();  
    void forward();  
    void pushState(any data, DOMString title, optional  
        DOMString url);
```

```
void replaceState(any data, DOMString title, optional    >
    DOMString url);
};
```

If that looks like Greek to you (unless you **are** Greek), you should still be able to decipher which methods are available:

- `go()`
- `back()`
- `forward()`
- `pushState()`
- `replaceState()`

The history API isn't brand new; we've long since had access to `go()`, `forward()` and `back()`. For instance, to send the user back a page:

```
history.back();
```

Or, to travel back two pages:

```
history.go(-2);
```

Or, to reload the current page:

```
history.go(0);
```

That's simple stuff. What you're most interesting in is applying new state, right? Of course you are.

Or, you're still struggling to understand what I mean by *state*, and are, at this point, half-blindly reading word after word with the sole intention of getting to the end of the chapter so that you can pat yourself on the back and go to bed. That's okay, too! We've all been there.

history.pushState

To push a new *record*, or *state* object to the browser's **history**, we can use the **pushState** method: **history.pushState**. Don't let this confuse you; just say it out loud to yourself. This method allows us to push a particular state of a page to the browser's history records.

This method accepts three parameters:

1. **Data** – The information, or *state* to be saved.
2. **Title** – Serves as the heading for that particular *state*.
3. **URL** – Optional value to be applied to the URL.

Try it out. Create a new HTML file, and paste the following between the **script** tags.

```
history.pushState(  
  'some data to store',  
  'My Page Title',  
  'page'  
) ;
```



[View the **pushState** demo.](#)

When testing this code at JSBin.com, you'll see that the URL immediately updates to `/page`. Also, there will now be a new history record.

 jsbin.com/page

Note that, at the time of this writing, no browser currently supports the **title** parameter. That's okay; there's no harm in adding it. They will provide support at some point in the future.

The popstate Event

Pushing a new record to the browser's history stack is great; but, how do we use that data? We can use the **popstate** event to **listen** for when session history is accessed (without leaving the current page). Try adding the following **listener** to your document.

```
window.addEventListener('popstate', function(event) {  
    console.log(event);  
  
    // do something to update the page  
}, false);
```

You'll notice an interesting thing if you open Chrome Dev Tools, switch to the *Console* tab, and load your file. The **popstate** event fires immediately — without us ever pressing the *Forward* or *Back* button. How come? In Chrome's eyes, the initial page load is considered to be a change in state (which it sort of is). This event does not fire twice in other browsers.

You might be expecting the page to automatically revert to your previously saved state when you now hit the *Back* button. But, unfortunately, it doesn't quite work that way. We still need to manually set up the display. The API simply gives us a hook to update sections of a page without performing a full reload.

Project

In this **hands-on** project, we'll build a simple image viewer. We'll set up three links to images. When one is clicked, rather than redirecting to the image, we'll instead load the link's **href** into an **img** tag, and display it on the page.

Remember: just because you can manipulate the address bar's URL does not mean that your website or application should suddenly depend on JavaScript to function correctly. This is

a terrible practice. Keep that in mind as you work through this project.

1. Links

We begin with a handful of links.

```
<ul>
  <li><a href="html5.jpg" title="HTML5 Rocks" data-url="html5">HTML5</a></li>
  <li><a href="wpplugincourse.png" title="WordPress is Great" data-url="wp">WordPress</a></li>
  <li><a href="bestoftuts.jpg" title="Best of Tuts+" data-url="tuts">Best of Tuts+</a></li>
</ul>
```

Notice that we've also added a **title** attribute and **data-url**. This last custom attribute (also part of HTML5) will be used as the value for the address bar: `mysite.com/html5`.

2. Template

Next, we should create a simple template. While it may be easier to embed a bunch of HTML into your JavaScript, try not to do so. It's a bad practice.

```
<div class="result">
  <script type="template" id="template">
    <h2> {{title}} </h2>
    
  </script>
</div>
```

As this will be a makeshift tempting solution, I've decided to use the `{{NAME}}` syntax. Feel free to use what you prefer.

3. Anchor Click

We should *listen* for when one of those anchor tags is clicked. When it is, we'll disable its default action (to redirect to a different page), apply its attribute values to the template we created in step two, and then add a new record to the browser's history. That way, when the visitor clicks and views a few images, when they press the *Back* button, the page will incrementally switch to the previous states, or images, in this case.

We'll stick with jQuery in this chapter, in order to focus more exclusively on the History API.

```
; (function() {  
    var Viewer = {  
        // store location of template  
        template: $('#template').html(),  
  
        init: function() {  
            $('ul').on('click', 'a', function(e) {  
                Viewer.applyTemplate(this); // this equals element >  
                // that was clicked  
                e.preventDefault(); // disable default action of >  
                // anchor tag  
            });  
        },  
  
        applyTemplate: function(data) {  
            //  
        }  
    }  
  
    Viewer.init();  
})();
```

Above, we've set up the basic structure for our script. The `init` method will be the first to execute. It simply applies an event listener to the anchor tags (you'll want to be more specific in your projects). When clicked, we call the `applyTemplate` method.

The on Method

New to jQuery 1.7, the `on()` method attaches event handlers to a specified set of elements. In the case of the previous code snippet, rather than attaching an event handler to every anchor tag on the page, we instead attach it to the parent `ul`, and then pass a selector as the second parameter of the `on()` method. For those familiar with jQuery's `delegate()` method, this works in the same fashion. This way, we're attaching a single event listener, rather than multiple ones.

Admittedly, in our case, the performance difference is negligible, however, it's still a good habit to get into.

4. Apply the Template

The `applyTemplate` method should take the attributes from the anchor tag that was clicked, and attach them to the template that we created.

```
applyTemplate: function(data) {  
    var template =  
        Viewer  
            .template  
            .replace(/{{title}}/gi, data.title)  
            .replace(/{{imgSrc}}/gi, data.href);  
  
    // Append the filled-in template to the result div.  
    $('.result')  
        .children(':not(#template)')  
        .remove()
```

ROCK*

TIP

Curious about the semi-colon before the self-executing anonymous function? It's not required, but is a best practice – particularly when scripts will be concatenated. It ensures that any potential missing semicolons that come before it won't break your scripts when they're joined.



```
.end()  
.append(template);  
  
return this;  
}
```

In the template, we can replace all occurrences of `{{title}}` with the anchor tag's `title` attribute by using a simple regular expression.

```
template.replace(/{{title}}/ig, data.title)
```

This snippet essentially translates to, “Replace every sequence of `{{title}}` with the `title` attribute of the anchor tag that was clicked.”

Flags

- `i` – Capitalization is irrelevant. (Look for `{{title}}` or `{{TiTLE}}`)
- `g` – Perform a global search. Without this flag, only the first occurrence of `{{title}}` would be updated.

Now that the snippet has been prepared, we only need to insert it into the DOM. Normally, we could simply do:

```
$('.result').append(template);
```

However, every time the user clicks a new link, an additional fragment will be added. What we really need to do is remove any existing fragments, and **then** append the new one. That way, at any given time, only one image will display.

```
// Append the filled-in template to the result div.  
$('.result')  
.children(':not(#template)') // don't remove the template
```

```
.remove()  
.end()  
.append(template);
```

- [HTML5](#)
- [WordPress](#)
- [Best of Tuts+](#)

Best of Tuts+



View the [Step 4 demo](#).

5. Update History

At this point, the user has clicked on an anchor tag, and we've asynchronously displayed the data on the page. We're now in a new state. We should update the history accordingly. If we return to the `init` method, we'll make a call to a new *method* that handles this task.

```
init: function() {  
    $('ul').on('click', 'a', function(e) {  
        Viewer  
            .applyTemplate(this)  
            .updateHistory(this);  
        e.preventDefault();  
    });  
},
```

updateHistory()

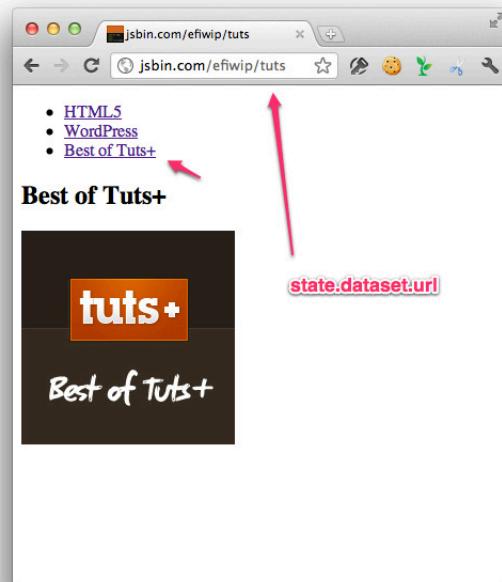
We can use the new `history.pushState()` method that you learned earlier in this chapter. We pass three parameters:

1. The data to be saved (may be anything that can be serialized).
2. The title for the document, or state (not currently implemented by browsers).
3. Optional value to be applied to the address bar.

Note that the function will accept an argument, `state`, which is still equal to the anchor tag that was clicked.

```
updateHistory: function(state) {  
  
    // This can be anything you want.  
    var dataToSave = {  
        title: state.title,  
        href: state.href,  
        url: state.dataset.url  
    };  
  
    // Save a new record to the browser's history history.  
    pushState(  
        dataToSave,  
        state.title,  
        state.dataset.url  
    );  
};
```

When this method is called, a new history record will be added. Go ahead and try it out.



[View the Step 5 demo.](#)

6. Pressing the Back Button

Now that we've successfully added a new record to the browser's history, we should listen for when the *Back* or *Forward* button is pressed. However, because we've manipulated the history stack, we need to use the **popstate** event to mimic this functionality. This way, only a portion of the page will be refreshed, rather than the entire page itself! Neat, ay?

Once again, to keep our code clean, we'll abstract this away to its own function.

init()

```
init : function() {
  $('ul').on('click', 'a', function(e) {
```

```
Viewer  
  .applyTemplate(this)  
  .updateHistory(this);  
  
  e.preventDefault();  
});  
  
this.handleState(); // listen for back or forward button  
,
```

handleState()

```
handleState: function() {  
  $(window).on('popstate', function(e) {  
    // forward or back was pressed  
    Viewer.applyTemplate(e.originalEvent.state)  
  });  
}
```

Here, we've attached the `popstate` event to the `window` object. Once it fires, the data we saved when we initially called `history.pushState` will be accessible, via the event object: `e.state`.

We've already written the necessary functionality that updates the page with a new image. Now that we've hooked into when the *Back* button is pressed, we can call the `handleState` method once again, while sending through the object (`dataToSave`, or `e.originalEvent.state`) that was saved. This way, the necessary image is loaded both when the user clicks on a link, and also when the *Forward* or *Back* button is pressed.

Confused by `e.originalEvent.state`? To achieve cross-browser compatibility, jQuery will overwrite the existing event object that is passed to the `popstate` event's handler. With vanilla JavaScript, we could access the data that was saved with `e.state`. However, with jQuery, we must first access the original event: `e.originalEvent.state`.



[View the final product demo.](#)

The Job Isn't Finished

To fully take advantage of the History API, we should use a server-side language to compensate for the various new URLs that we're applying to the address bar. For instance, if we load `example.com/index.html`, then call `history.pushState`, and set the third `url` parameter to, say, "`my-custom-page`", the url will now be equal to `example.com/my-custom-page`. However, don't for a second think that you've suddenly created a brand new link that can be indexed by search engines, and shared with your friends. Remember — we're faking these URLs, and what happens when the user clicks the *Back* or *Forward* buttons.

A server-side language, like PHP, should still be used to handle these requests. But other than that, isn't the History API a cinch to work with?

13

The File && Drag and Drop APIs

Browser Support



Browser Support for the File API.



Browser Support for the Drag and Drop API.

Before HTML5, we had no standardized way to work with a user's local file system. Thankfully, that's no longer the case. We now have the ability to read, validate, and modify a file's contents — just as long as the user takes an active role in the process. It would be a significant security concern if we performed these actions without the user's consent!

Even better, when we combine this with the new Drag and Drop API, we can now provide an unprecedented level of convenience for the user. Need to upload a handful of photos to your server? Simply drag a folder from your desktop onto a drop zone region in the browser. It couldn't be simpler!

Though the Drag and Drop API may appear to be new to you, Internet Explorer provided support for it as far back as... wait for it... Internet Explorer 5, in 1999! In fact, the spec is based upon their implementation. So the next time you reach in your pocket

to grab another “kill IE” sticker, keep in mind that — though plenty of criticism is valid — they’re still responsible for some fantastic advancements in our industry.

Feature Detection

As with all the new APIs, we shouldn’t write a line of code before determining whether the browser provides support.

Testing the File APIs

Depending on which APIs you need access to...

```
if ( window.File && window.FileReader && window.Blob ) {  
    // browser provides support  
}
```

Feel free to remove any references that you don’t require. For instance, if you don’t plan on using the `Blob` interface — which allows us to slice a given file into a range of bytes — you can remove that piece of detection.

Modernizr

If using Modernizr, you’ll want to download a custom build, and select “File API” from the “Community Add-ons” section.

Taking a look under the hood reveals the following bit of feature detection:

```
Modernizr.addTest("file", function() {  
    return !(window.File && window.FileReader);  
});
```

Excellent! That's essentially the same as our version.

Testing the Drag and Drop API

As you'll learn in this chapter, we can make an element *draggable* by applying an attribute, appropriately called, **draggable**.

```
<div draggable>Hi There</div>
```

At first glance, testing for drag-and-drop support might appear to be as simple as doing:

```
if ( 'draggable' in document.createElement('div') ) {  
    // your browser supports the drag and drop API  
}
```

However, there are a couple of issues with this method:

1. Older versions of iOS incorrectly reported support for drag and drop.
2. Internet Explorer was the first champion of drag and drop. They supported it long before the other browsers. Ironically, though, IE9 claims that it doesn't recognize the **draggable** attribute, when, in fact, it does.

The screenshot shows the 'Community add-ons' section of the CanIUse.com website. It lists various experimental features as checkboxes. A red arrow points to the 'File API' checkbox, which is checked. Other listed features include 'Cookies', 'background-repeat', 'background-size-cover', 'box-sizing', 'cubic-bezier-range', 'display table', 'overflow-scrolling', 'pointer events', 'userselect', 'Custom Protocol Handler', 'createElement attr', 'details', 'Progress Meter', 'Emoji', 'deviceorientation motion', 'placeholder', 'hyphens', 'webp', 'data-uri', 'webgl extensions', 'framed', and 'sharedworkers'. At the bottom, there are 'GENERATE!' and 'DOWNLOAD Custom Build' buttons.

Instead, a better route is to determine whether the browser supports two key events in the Drag and Drop API: `dragstart` and `drop`.

```
var supportsDragAndDrop = (function() {  
    var el = document.createElement('div');  
    return ('ondragstart' in el && 'ondrop' in el);  
})();  
  
// usage if ( supportsDragAndDrop ) {  
//   ah yeah  
}  
}
```

This function will run immediately when the page loads, determine whether both the `ondragstart` and `ondrop` events are recognized, and, if so, return `true`.

Modernizr

Though the code is different, you'll find that this is essentially the same as Modernizr's approach.

```
tests['draganddrop'] = function() {  
    return isEventSupported('dragstart') &&  
        isEventSupported('drop');  
};  
  
// usage  
if ( Modernizr.draganddrop ) ...
```

Capturing File Information

The best way to learn a new API is to dig in; let's write a small demo that displays information about an uploaded file on the page. We want to fetch both the name of the file, and its size.

The Markup

We begin with a simple form:

```
<form>
  <input type="file" id="file">
</form>
```

The jQuery

Next, we'll *listen* for when a file is selected. jQuery's `change` event will work nicely here.

```
$('#file').on('change', function(e) {
  console.log(e);
});
```

We can access information that relates to the uploaded file, via `e.target.files`.

```
$('#file').on('change', function(e) {
  console.log(e.target.files);
});
```

Though it essentially is an *array*, technically, `e.target.files` is part of the `FileList` interface.

“A `FileList` interface represents an array of individually selected files from the underlying system. **”**

Using this bit of code, after choosing a photo, here is what was logged to my console:

```
▼FileList
  ▼0: File
    fileName: "Bering_Land_Bridge_NPr_Serpentine_Hot_Springs.JPG"
    fileSize: 4076138
    ► lastModifiedDate: Date
      name: "Bering_Land_Bridge_NPr_Serpentine_Hot_Springs.JPG"
      size: 4076138
      type: "image/jpeg"
      webkitRelativePath: ""
    ► __proto__: File
    length: 1
    ► __proto__: FileList
```

As we're only working with a single file in this case, we can retrieve the name of the file and its size, like so:

```
$('#file').on('change', function(e) {  
  var file = e.target.files[0];  
  console.log(file.name + ': ' + file.size);  
});
```



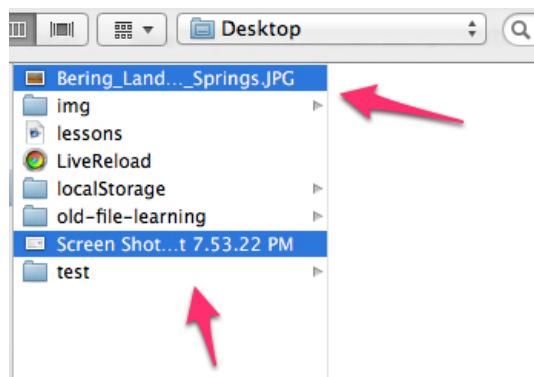
[View the file information demo.](#)

Multiple Files

By applying the `multiple` attribute to a `file` input, we can, as a result, select multiple files (by holding *Command* or *Alt*). It's about time!

The Markup

```
<form>  
  <input type="file" id="file" multiple>  
</form>
```



The jQuery

To filter through the uploaded files, we only need to modify our JavaScript a bit:

```
$('#file').on('change', function(e) {  
    var files = e.target.files;  
  
    $.each(files, function(i, file) {  
        console.log(file.name + ': ' + file.size);  
    });  
});
```



[View the multiple files demo.](#)

Drag and Drop

If we're going to toy around with this new File API, let's also include some drag-and-drop action. We'll modify our original demo, and replace the `file` input with a "drop zone."

The Markup

```
<div class="drop">Drop Files Here</div>
```

To make it more "drop" friendly, we can also add a touch of styling:

```
.drop {  
    border: 3px dashed #666;  
    padding: 20px;  
    text-align: center;  
    width: 50%;  
    margin: auto;  
}
```



The jQuery

Next, we'll listen for when a file is dragged over this region.

```
$(‘div.drop’).on(‘dragover’, function() {  
    console.log(‘Dragging!’);  
});
```

If you're working along (you should be), open your browser's console, and drag a file from your desktop over this region. You'll find that "Dragging!" is logged to the console continuously.



[View the dragging demo.](#)

If you'd prefer to listen for when the file first enters the region, you'd instead use the **dragenter** event.

Drag Events

In fact, there's a plethora of new events available to the object being dragged:

- **dragstart** – User begins dragging the object.
- **dragend** – The object has been released by the user (**mouseup**).
- **drag** – Fires continuously, while the object is being dragged.

Don't allow this to be overwhelming, but there are also events which fire on the drop zone, or a specified region of the page.

- **dragenter** – Fires when an object enters a designated drop region
- **dragleave** – Fires when the object exits the drop region

- **dragover** – Similar to drag; fires repeatedly while the object is within the drop region
- **drop** – Fires on the condition that the object is dropped in the drop zone, and the default action of the **dragover** event is prevented with `e.preventDefault()`.

That last **drop** event might have confused you. Due to Internet Explorer's original implementation of drag-and-drop (which the W3C spec is based upon), the **drop** event will only fire if the default **dragover** event is prevented. That's a bit odd, isn't it? If we refer to [the specification](#), we'll find that the default action for the **dragover** event is to "reset the current drag operation to none." Huh? Why?

Unfortunately, I'm not smart enough to explain to you why this decision was made. The most I can offer is that, if you want the **drop** event to fire, you must first prevent the **dragover**'s default action.

```
$('.drop').on({
  'dragover': function() {
    e.preventDefault(); // required for drop event to fire
  },
  'drop': function() {
    console.log('dropped');
  }
});
```

Disabling the Defaults

Back to our demo, if you drop an image onto the drop region we specified, you'll still find that the image opens in a new tab. To fix this, we need to disable the default action.

```
$(div.drop).on('dragover', function(e) {
  console.log('Dragging!');
  e.preventDefault();
});
```

With this addition, though, dropping the image onto the drop zone does nothing at all. Let's capture this data by using the `drop` event listener.

```
$('.div.drop').on({
  'dragover': function(e) { e.preventDefault(); },
  'drop': function(e) {
    console.log(e.originalEvent.dataTransfer.files);
  }
});
```

Paste that snippet into your local project, and take a look at the console once you've dropped a file onto the region.

We can use `e.dataTransfer.files` to access information related to the dropped files. However, jQuery rewrites the `event` object to make it more cross-browser friendly. As such, to access the `dataTransfer` object when working with jQuery, we must first dig into the original event first.

Here is what was logged to my console after dropping two random images onto the drop zone:

```
▼ fileList
  ▼ 0: File
    fileName: "Screen Shot 2011-12-27 at 7.53.22 PM.png"
    fileSize: 654956
    ► lastModifiedDate: Date
      name: "Screen Shot 2011-12-27 at 7.53.22 PM.png"
      size: 654956
      type: "image/png"
      webkitRelativePath: ""
    ► __proto__: File
  ▼ 1: File
    fileName: "Bering_Land_Bridge_NPr_Serpentine_Hot_Springs.JPG"
    fileSize: 4076138
    ► lastModifiedDate: Date
      name: "Bering_Land_Bridge_NPr_Serpentine_Hot_Springs.JPG"
      size: 4076138
      type: "image/jpeg"
      webkitRelativePath: ""
    ► __proto__: File
  length: 2
  ► __proto__: fileList
```

Isn't that neat? Providing *drag-and-drop* support only requires a few additional lines (and a polyfill for older browsers, of course).



View the [drag-and-drop demo](#).

Reading Files

In this section, we'll move on to the fun stuff: reading files.

Using a Server

Before we continue, you need to be working on a server in order to read and manipulate files. If you're a PHP developer, drag your project directory into your *htdocs* folder. Or, if you have no idea what I'm talking about right now, work through this section using [JSBIN.com](#) as your coding environment.

FileReader()

To read a file asynchronously, we can use the `FileReader()` interface. It's honestly a fairly simple process:

1. Acquire a `FileList` of the selected files (you've already learned how to do this).
2. Create a new instance of `FileReader()`.
3. Begin reading the file.
4. Listen for when the reading has completed successfully.
5. Access the results of the read, via `e.target.results`.

ROCK* TIP

If Python is installed on your system (Mac and Linux users: it is), add the following alias to your `~/.bash_profile` file for an instant server: `alias server='open http://localhost:8000 && python -m SimpleHTTPServer'`. Next, open the Terminal, browse to your project directory, and type `server`. Voila – instant server!



Reading the File

To read a file, we first must determine how to read it. For instance, are we dealing with text? What if the file is an image? We have four reading options, each of which accepts a the selected file as an argument.

1. `readAsText()` – Fairly self explanatory. This method will read the text contents of the selected file.
2. `readAsDataURL()` – This will interpret the file as an encoded Data URL. This is most commonly used with images.
3. `readAsArrayBuffer()` – This will read the contents of the file as an array buffer object.
4. `readAsBinaryString()` – For a more general option, this will read the file as binary data.

Read Events

Once the *reading* process has begun, we can hook into various events during this cycle, as part of the `ProgressEvent` interface.

1. `load` – The read has completed successfully.
2. `loadstart` – The read has begun.
3. `loaded` – The read has completed (either success or failure).
4. `progress` – Fires every time a blob has been read into memory. This can be used to report to the user how far along the reading has progressed.
5. `abort` – The read has been aborted (likely, via the `abort()` method).
6. `error` – The read has failed.

An example usage of one of these events might be:

```
var reader = new FileReader();

reader.onprogress = function(e) {
    console.log('So far, this file is ' + e.loaded / e.total * 100 + '% loaded');
};
```

Hands-On

Let's create a demo that will allow the user to drag a bunch of images onto the page, and view all of them, without any page refreshes, or being redirected.

The Markup

As with the previous demo, we'll create a simple drop zone for the images.

```
<div class="drop">Drop Images Here</div>
```

The CSS

We'll also apply a touch of styling, and set a maximum width for the images.

```
.drop {
    text-align: center;
    width: 50%;
    margin: 100px auto 0;
    padding: 30px;
    border: 3px dashed #666;
}

img {
    max-width: 200px;
    margin: 0 20px 20px 0;
}
```

The jQuery

Referring back to the `FileReader()` order of operations, we'll work through them one at a time.

Acquire a `FileList` of the selected files.

```
$('div.drop').on({
  'dragover': function(e) {
    e.preventDefault();
  },
  'drop': function(e) {
    // FileList is available, via dataTransfer.files
    console.log(e.originalEvent.dataTransfer.files);
  }
});
```

This should be fairly familiar to you at this point.

Create a new instance of `FileReader()`.

Because it's possible that the user will drag multiple images onto the drop zone, we need to filter through the `FileList` and read each image. We'll begin by creating the `read` function, itself. This function's sole responsibility is to read the file that is passed to it, and then append an image to the DOM.

```
var addImage = function(i, file) {
  var reader = new FileReader();

  // We're only working with images right now.
  if ( !/image/.test(file.type) ) return;
};
```

Note that we can't blindly create the image without first detecting what type of file was supplied. What if the user drags a PDF onto the drop zone? To compensate, we should first determine what

the file's type is, via `file.type`. If I've uploaded an image, the type will be set to `image/png`, `image/jpg`. As such, we can run a quick regular expression test to determine if the string, "image" is contained within the file type. If it's **not**, we can return from the function: the file type is not correct.

Begin reading the file.

```
var addImage = function(i, file) {  
    var reader = new FileReader();  
  
    // We're only working with images right now.  
    if ( !/image/.test(file.type) ) return;  
  
    reader.readAsDataURL(file);  
};
```

We've created a new instance of `FileReader()`, but haven't physically begun reading the data. Because we're working with an image, it makes sense to read the file as a DataURL.

Listen for when the reading has completed successfully.

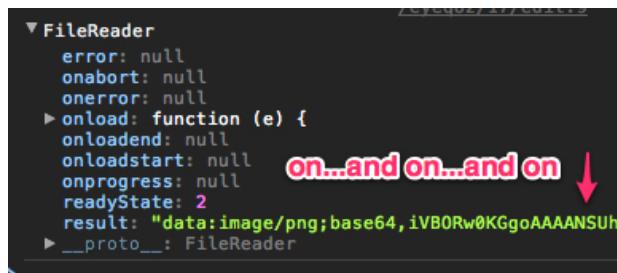
```
var addImage = function(i, file) {  
    var reader = new FileReader();  
  
    // We're only working with images right now.  
    if ( !/image/.test(file.type) ) return;  
  
    reader.readAsDataURL(file);  
    reader.onload = function(e) {  
        /// file has finished reading successfully  
    };  
};
```

`reader.onload` will fire only on the condition that the file has finished reading successfully.

Access the results, via e.target.results.

```
var addImage = function(i, file) {  
    var reader = new FileReader();  
  
    // We're only working with images right now.  
    if ( !/image/.test(file.type) ) return;  
  
    reader.readAsDataURL(file);  
    reader.onload = function(e) {  
        $('<img>', {  
            'src': e.target.result  
        }).appendTo(document.body);  
    };  
};
```

Once the file has been read successfully, the results will be available, via `e.target.result`. In this case, that will be equal to a huge DataURL, which we can then assign to a new image tag, and throw it into the DOM.



Filter through the dropped files.

So far, we've created a function that will read an image and throw it into the DOM. The last step is to filter through the files/images that were dropped into the drop zone, and call the `addImage` function, accordingly.

```
$( 'div.drop' ).on({  
    'dragover': function(e) { e.preventDefault(); },
```

```
'drop': function(e) {
    $('img').remove(); // Generic, for demo. Careful.

    // Filter through the files,
    // and call the addImage function.
    $.each(e.originalEvent.dataTransfer.files, addImage);
}

});
```

Final jQuery

```
var addImage = function(i, file) {
    var reader = new FileReader();

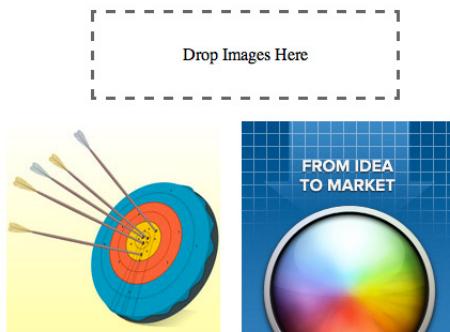
    // We're only working with images right now.
    if ( !/image/.test(file.type) ) return;

    reader.onload = function(e) {
        $('', {
            'src': e.target.result
        }).appendTo(document.body);
    };

    reader.readAsDataURL(file);
};

$('div.drop').on({
    'dragover': function(e) { e.preventDefault(); },
    'drop': function(e) {
        $('img').remove(); // Generic, for demo. Careful.

        // Filter through the files,
        // and call the addImage function.
        $.each(e.originalEvent.dataTransfer.files, addImage);
    }
});
```



[View the images demo.](#)

Reporting Progress

The **progress** event is a neat one; it provides us with the ability to monitor a file's reading. When combined with the new HTML5 **progress** tag, we get an instant, dynamic progress bar!

Markup

To test this idea, we only need to add the new **progress** tag to our HTML:

```
<div class="drop">Drop Images Here</div>
<progress min=0 max=100 value=0>
```



onprogress

Next, we hook into the **progress** event. When it fires, among other things, the event object will contain both a **loaded** and **total**

property. These two can be combined to determine how much of the file has been loaded by using the following equation:

```
e.loaded / e.total * 100
```

However, this will likely result in a long decimal, such as:

```
99.47035434562974
```

We should round it to the nearest whole number:

```
Math.round( e.loaded / e.total * 100 ); // 99
```

This number then only needs to be assigned to the **progress** tag's **value** attribute.

```
var progress = $('progress')[0];
...
progress.value = Math.round( e.loaded / e.total * 100 );
```

At this point, you should be moderately familiar with the process of capturing a value, and creating a new instance of **FileReader()**. The only new piece of functionality is attaching the **onprogress** event.

```
(function() {
    var progress = $('progress')[0];

    // feature detection
    if ( !Modernizr.file ) {
        alert('Your browser unfortunately does not support the >
            File API');
    }

    $('div.drop').on({
        'dragover': function(e) { e.preventDefault(); },
        'drop': function(e) {
```

```
var reader = new FileReader();

reader.readAsBinaryString(e.originalEvent.▶
  dataTransfer.files[0]);
// Every time a blob is read into memory, update the progress bar.
reader.onprogress = function(e) {
```



Drop A Large File Here



[View the progress demo.](#)

```
progress.value = Math.round(e.loaded / e.total * 100);
};

reader.onload = function(e) {
  // do something with e.target.result
  // without assigning a final 100% value,
  // the progress bar will linger around 97%, because
  // it doesn't fire upon completion
  progress.value = 100;
};
}
})()
});
```

Note that the demo works best when uploading significantly large files. Otherwise, it's possible that the `progress` event won't fire at all.

Naturally, this is only test code. For a real-world project, you'll likely want to

ROCK*
TIP

A massive list of polyfills is available in the [Modernizr Wiki](#).



only execute this code on the condition that the browser supports the File API (refer to the beginning of the chapter), and implement a polyfill for older browsers.

See what I mean? This stuff isn't so tough once you learn the basic methods and events! Go grab some raisins, and then it's on to the next chapter!



Web Workers are Ants

Browser Support



JavaScript is a single-threaded environment. What this essentially translates to is only one script can execute at a single time. Needless to say, as our dependency on JavaScript has become greater and greater over the last five years, the fact that we're limited to a single thread can be a significant hindrance to the overall performance of our web applications. For instance, consider the fairly common case of a JavaScript application that needs to query a client-side database, perform complex DOM manipulations, respond to UI events, and request data from third-party APIs.

In the past, developers have resorted to a variety of tricks and hacks to achieve the illusion of concurrency; you've likely used some of them yourself (`setInterval`, `setTimeout`, etc.), however, now there's a better way.

Before We Begin

When experimenting with web workers in Google Chrome, due to security concerns, you must either be working on your server, or open Chrome with the `--allow-file-access-from-files` flag.

```
open my-page.html --args --allow-file-access-from-files
```

Alternatively, Mac users might prefer the simple-server approach. Within `~/.bash_profile`, add a new alias:

```
alias server='open http://localhost:8000 && python -m
SimpleHTTPServer'
```

Now, within Terminal, browser to the folder you're working in, and type **server**. Instant server for the current directory!

Note: this is not necessary in Firefox, Opera, or Safari.

Say Hello to Web Workers

The new [Web Worker API](#), based upon the [Gears WorkerPool API](#) (deprecated), allows us to execute any number of scripts “in the background.” The huge advantage to this is in the fact that they won’t block the user interface. Have you ever come across that nasty browser popup, notifying you that a script on the page is taking too long to respond?



Well, if that apparently intensive script was executed in the background, things like this would never occur.

For a simple analogy, think of workers as ants. In a single threaded environment, we only have one ant, who carries the brunt of the load. However, with web workers, we can assign duties, or threads to multiple ants. This ant will focus on a complex database query; this other ant will handle the process of repeatedly

ROCK★ TECH TERM

A **worker** is a script that is loaded and executed in the background.



querying this web service. Sounds good, right?

A Crash Course

Detection

The `Worker` function is available, via the `window` object. This makes it particularly easy to test.

```
if ( window.Worker ) {  
    // let's get to work  
}
```

Or, with Modernizr:

```
if ( Modernizr.webworkers ) {  
    // let's get to work  
}
```

As you'll learn shortly, there are also other interfaces that you'll likely find yourself working with, including `window.BlobBuilder` and `window.URL`. These two are slightly more difficult to detect, due to the fact that, at the time of this writing, both Webkit and Mozilla implement them via their own vendor prefixes.

```
if ( window.URL || window.webkitURL || window.mozURL ) {  
    // browser supports the URL interface  
}  
  
if ( window.BlobBuilder || window.WebKitBlobBuilder || ▶  
    window.mozBlobBuilder ) {  
    // browser supports the BlobBuilder interface  
}
```

ROCK*

TIP

There are two types of web workers: dedicated and shared. In this chapter, we'll be focusing exclusively on the former, due to the fact that browser support for the latter is still somewhat sparse (no Firefox or IE at the time of this writing).



Usage

To create a new thread, or worker, we need store the applicable code within its own file. Technically, there are ways around this (which we'll get to shortly), however, for the time being, let's keep it simple.

While it's not exactly appropriate, for the initial purposes of interacting with the API, we'll stick with a very simple example. I don't want to flood you with code just yet!

We begin with a fresh page:

```
<!doctype html>
<html>
<head>
  <title>Learning Web Workers</title>
</head>
<body>

  <script></script>

</body>
</html>
```

Next, we'll create a new instance of the `Worker` object, while passing in a path to the name of the script that we wish to execute; we'll call ours, `lookup.js`.

```
var worker = new Worker('lookup.js');
```

When this line is executed, the browser will attempt to download the file, and create a new thread. However, the worker won't yet be activated, or triggered. We can do so, via the `postMessage()` method.

ROCK*

TIP

All worker threads must be stored in their own file.



```
worker.postMessage(''); // lookup.js worker starts
```

We can also pass a *string* or *object*, which will be sent to the worker.

```
worker.postMessage('Hannah Montana');

worker.postMessage({
  first: 'Hannah',
  last: 'Montana'
});
```

Firefox will squawk if you attempt to call `postMessage()` without passing a value. If you don't need to send through any data, pass an empty string, as demonstrated in the previous example.

This data can then be accessed, via the event object: `e.data`. More on that shortly.

When browsers first began integrating the Web Worker API, there were some implementation inconsistencies from browser to browser. For example, while Gecko (Firefox) has always allowed users to pass any data type through `postMessage()` from the start (via serialization), initially, Chrome and Safari only allowed strings to be sent. This is no longer the case. Nonetheless, it's worth mentioning.

Alerts and Logs

If you're like me, from within `lookup.js`, your first instinct is to show an `alert`, or log a simple message to the console, to test whether it's working correctly. Unfortunately, that's not the way it works. A web worker does not have access to the DOM, the `document`, or the `window` object. Should you try to use `alert`, you'll be met with an error: "Uncaught ReferenceError: alert is not defined."

However, a web worker does have access to:

- The **location** object
- **XMLHttpRequest**
- Timeouts and intervals
- The **navigator** object
- Subworkers
- The application cache

Lookup.js

Within **lookup.js**, we'll simply store a function that returns the name of the editor, when a Tuts+ site is referenced. For example, if the user passes "Jeffrey," the function will return "Nettuts+."

```
// lookup.js
function getSite(name) {
  var lookup = {
    'Jeffrey': 'Nettuts+',
    'Grant': 'Psdtuts+',
    'Ian': 'Webdesigntuts+',
    'Japh': 'Wptuts+'
  };

  return lookup[name] || 'Hmm... not sure about that one';
}
```

The Event Model

Notice that we haven't called the function from anywhere within this script. Within our HTML file, we posted a message, with **postMessage()**. Web workers implement a simple event model when communicating. On each side of the phone line, they post and listen for messages.

To *listen* for when a message has been received, we use the **message** event listener.

```
// lookup.js
this.addEventListener('message', function() {
  // do something
}, false);
```

Or, with the attribute event handler method:

```
// lookup.js
this.onmessage = function() {
  // do something
};
```

Keep in mind that the former approach is recommended.

Within the context of a web worker, **this** and **self** will both refer to the global scope, **[WorkerGlobalScope]**.

```
this = self = [WorkerGlobalScope]
```

In translation, this means that it's not necessary to use **this** when referencing **addEventListener**. As a result, our code can be shortened to:

```
// lookup.js addEventListener('message', function() {
  // do something
}, false);
```

Once the **message** event fires from within **lookup.js**, we can call the **getSite** function. For now, we'll hardcode a site name.

```
// lookup.js
function getSite(name) {
  var lookup = {
```

```
'Jeffrey': 'Nettuts+',  
'Grant': 'Psdtuts+',  
'Ian': 'Webdesigntuts+',  
'Japh': 'Wptuts+'  
};  
  
return lookup[name] || 'Hmm... not sure about that one';  
}  
  
addEventListener('message', function() {  
  getSite('Jeffrey');  
}, false);
```

To communicate back to the main execution thread (the HTML file, in our case), we, again, use `postMessage()`. We'll take the result of the `getSite` function, and pipe it back to the HTML file.

```
// lookup.js  
function getSite(name) {  
  var lookup = {  
    'Jeffrey': 'Nettuts+',  
    'Grant': 'Psdtuts+',  
    'Ian': 'Webdesigntuts+',  
    'Japh': 'Wptuts+'  
  };  
  
  return lookup[name] || 'Hmm... not sure about that one';  
}  
  
// when called, respond with the name of the site Jeffrey ▷  
// edits.  
addEventListener('message', function() {  
  postMessage( getSite('Jeffrey') );  
}, false);
```

In the same way that the `lookup.js` worker listened for a message from the HTML page, the same is true the other way around. To listen for a response from the worker:

```
// index.html
var worker = new Worker('lookup.js');
worker.postMessage(''); // activate the thread

// listen for a response
worker.addEventListener('message', function(e) {
  console.log(e.data); // Nettuts+
}, false);
```

Notice that we're accessing the information that the worker sent through `postMessage()` with `e.data`.

Completing the Example

To finish our example, the string that is passed to the `getSite` function shouldn't be hardcoded. Instead, we'll add a dropdown `select` tag to the page, so that the user can make a selection, and see the results.

We first modify the HTML:

```
<form>
  <select>
    <option value="">--</option>
    <option value="nettuts">Nettuts+</option>
    <option value="psdtuts">Psdtuts+</option>
    <option value="webdesigntuts">Webdesigntuts+</option>
    <option value="wptuts">Wptuts+</option>
  </select>
</form>

<output></output>
```

As a bonus, we get to take advantage of the new HTML5 `output` tag, to display the results.

Next, we listen for when an **option** is selected. At that point, we take the associated value, and send its value to the worker, with **postMessage()**.

```
var worker = new Worker('lookup.js'),
sites = $('select');

sites.on('change', function() {
  var selectedSite = $(this).find('option:selected').val();
  selectedSite && worker.postMessage( selectedSite );
});
```

Once **postMessage()** is called, **lookup.js** will become “activated.” This thread will *listen* for any posted messages, capture any relevant data (the name of the Tuts+ site), and send it to the **getSite** function. The results from this function will then be posted back.

```
// lookup.js
function getSite(name) {
  var lookup = {
    'nettuts': 'Jeffrey',
    'psdtuts': 'Grant',
    'webdesigntuts': 'Ian',
    'wptuts': 'Japh'
  };

  return lookup[name];
}

addEventListener('message', function(e) {
  postMessage( getSite(e.data) ); // e.data = name of the >
  Tuts+ site
}, false);
```

Lastly, we listen for the result from the HTML file, and append it to the **output** tag.

```
worker.onmessage = function(e) {  
    document.querySelector('output').innerHTML = e.data;  
};
```

Nettuts+ ▾

Jeffrey

That'll do it!

A Second Example

Web Workers are frequently used to handle complex mathematical operations. For instance, consider a script that will calculate all prime numbers — up to 100,000. We might use the following function to determine if a number is prime:

```
function isPrime(num) {  
    for ( var i = 2; i < num; i++ ) {  
        if ( num % i === 0 ) return false;  
    }  
    return true;  
}
```

Then, to generate and log the list of prime numbers to the console:

```
var prime_nums = [];  
for ( var i = 2; i < 100000; i++ ) {  
    if ( isPrime(i) ) {  
        prime_nums.push(i);  
    }  
}  
console.log(prime_nums);
```

This will likely take five seconds or so.

```
[2, 3, 5, 7, 11, 13, 17, 19, 23, 29, 31, 37, 41, 43, 47, 53, 59, 61, 67, 71, 73, 79, 83, 89, 97, 101, 103, 107, index.html:5]
109, 113, 127, 131, 137, 139, 149, 151, 157, 163, 167, 173, 179, 181, 191, 193, 197, 199, 211, 223, 227, 229, 233, 239, 241,
251, 257, 263, 269, 271, 277, 281, 283, 293, 307, 311, 313, 317, 331, 337, 347, 349, 353, 359, 367, 373, 379, 383, 389, 397,
401, 419, 423, 431, 437, 443, 451, 457, 461, 467, 473, 481, 487, 493, 497, 503, 509, 511, 517, 523, 529, 531, 537, 541, 547, 553,
569, 571, 577, 581, 587, 591, 599, 607, 613, 617, 619, 631, 641, 643, 647, 653, 659, 661, 673, 677, 683, 691, 693, 701, 709, 719,
727, 733, 739, 743, 751, 757, 761, 769, 773, 777, 797, 809, 811, 821, 823, 827, 829, 839, 853, 857, 859, 863, 877, 881, 883,
887, 907, 911, 919, 929, 937, 941, 947, 953, 967, 971, 977, 983, 991, 997, 1009, 1013, 1019, 1021, 1021, 1031, 1033, 1039, 1049, 1051,
1061, 1063, 1069, 1087, 1091, 1093, 1097, 1109, 1117, 1123, 1129, 1151, 1153, 1163, 1171, 1181, 1187, 1193, 1201, 1213,
1217, 1223, 1239, 1231, 1237, 1249, 1259, 1277, 1279, 1283, 1289, 1291, 1297, 1301, 1303, 1307, 1319, 1321, 1327, 1361, 1367,
1373, 1389, 1399, 1409, 1427, 1429, 1437, 1439, 1447, 1451, 1453, 1459, 1471, 1473, 1481, 1487, 1489, 1499, 1511,
1521, 1531, 1537, 1541, 1553, 1561, 1567, 1571, 1577, 1581, 1587, 1591, 1597, 1601, 1607, 1613, 1619, 1621, 1627, 1631, 1637, 1643,
1667, 1669, 1693, 1697, 1699, 1709, 1721, 1723, 1733, 1741, 1747, 1753, 1759, 1777, 1783, 1787, 1789, 1801, 1811, 1823, 1831,
1847, 1861, 1867, 1871, 1873, 1877, 1879, 1889, 1901, 1907, 1913, 1931, 1933, 1949, 1951, 1973, 1979, 1987, 1993, 1997, 1999,
2003, 2011, 2017, 2027, 2029, 2039, 2053, 2063, 2069, 2081, 2083, 2087, 2089, 2099, 2111, 2113, 2129, 2131, 2137, 2141, 2143,
2153, 2161, 2179, 2203, 2207, 2213, 2221, 2237, 2239, 2243, 2251, 2267, 2269, 2273, 2281, 2287, 2293, 2297, 2309, 2311, 2333,
2339, 2341, 2347, 2351, 2357, 2371, 2377, 2381, 2383, 2389, 2393, 2399, 2411, 2417, 2423, 2437, 2441, 2447, 2459, 2467, 2473,
2477, 2561, 2523, 2571, 2549, 2543, 2549, 2557, 2579, 2591, 2593, 2609, 2617, 2623, 2631, 2638, 2647, 2651, 2659, 2663, 2671,
2677, 2683, 2689, 2695, 2701, 2707, 2713, 2719, 2725, 2731, 2737, 2743, 2749, 2755, 2761, 2767, 2773, 2779, 2785, 2791, 2797,
2803, 2809, 2833, 2837, 2843, 2851, 2857, 2861, 2879, 2887, 2897, 2903, 2909, 2917, 2927, 2939, 2953, 2957, 2963, 2969, 2971,
2999, 3001, 3011, 3019, 3023, 3037, 3041, 3049, 3061, 3067, 3079, 3083, 3089, 3109, 3119, 3121, 3137, 3163, 3167, 3169, 3181,
3187, 3191, 3203, 3209, 3217, 3221, 3229, 3251, 3253, 3259, 3271, 3289, 3301, 3307, 3313, 3319, 3323, 3329, 3331]
```

As we've discovered, the problem is that, because JavaScript is single-threaded, any operations which follow this code will not execute until it has completed.

```
function isPrime(num) {
  for ( var i = 2; i < num; i++ ) {
    if ( num % i === 0 ) return false;
  }
  return true;
}

var prime_nums = [];
for ( var i = 2; i < 100000; i++ ) {
  if ( isPrime(i) ) {
    prime_nums.push(i);
  }
}
console.log(prime_nums);

// Won't execute until the above completes
// ... or in roughly five seconds. Yikes!
document.body.appendChild( document.createTextNode(
  'Finally... '));
▶
```

A Case For Workers

A smarter choice is to export this code to a new thread. That way, it will execute in the background, and won't block the rest of the user interface.

index.html

```
var w = new Worker('prime.js');
w.postMessage(''); // activate the worker

// listen for a response (postMessage) from prime.js
w.addEventListener('message', function(e) {
    // log the list of prime numbers between 1 and 100,000
    console.log(e.data);
}, false);
```

prime.js

```
function isPrime(num) {
    for ( var i = 2; i < num; i++ ) {
        if ( num % i === 0 ) return false;
    }
    return true;
}

addEventListener('message', function() {
    var res = [];
    for ( var i = 2; i < 100000; i++ ) {
        if ( isPrime(i) ) {
            res.push(i);
        }
    }

    // send list of prime numbers to main execution thread
    postMessage(res);
}, false);
```

With this arrangement...

```
document.body.appendChild( document.createTextNode(
    'Finally... '));
```

... will execute immediately, rather than after the math calculations have been completed!

Importing Scripts

Within a worker, `this.importScripts(urls)` can be used to import any dependent scripts or libraries.

```
this.importScripts('script1.js', 'script2.js');
```

It's important to note that, just because a library like jQuery is referenced in the main execution thread, does not mean that it will be available to the worker. “No worries,” you might think. “I'll simply import jQuery.”

```
this.importScripts('jquery.js');
```

Nope — that won't work either. Think about it — jQuery is primarily a library for manipulating and working with the DOM. However, the `window` object isn't available to worker threads (due to security concerns). As a result, any attempts to import one of the popular libraries will result in an error.

The best alternative is to roll your own custom library, which, admittedly, isn't ideal. jQuery users might be interested in [Rick Waldron's jQuery.hive solution](#).

Inline Workers

There will likely be situations when you need to generate a worker dynamically. In these cases, abstracting a script to its own file may not be possible. The `BlobBuilder` interface provides us with the means to keep workers inline.

```
var blob = new BlobBuilder();
```

Next, we append the applicable worker code to the blob. Whatever would normally be placed in the external file should be placed

here. In this case, we'll simply post a message back to the main execution thread.

```
var blob = new BlobBuilder();

blob.append( "addEventListener('message', function() {
    postMessage('Hello from the worker.');
}, false)"
);
```

Now, we dynamically create a custom URL reference.

```
var blob = new BlobBuilder(),
    url;

blob.append( "addEventListener('message', function() {
    postMessage('Hello from the worker.');
}, false)"
);

url = window.URL.createObjectURL( blob.getBlob() );
```

`URL.createObjectURL()` will generate a URL that can then be passed to `new Worker()` in the same way that we would reference a real file's path. This URL will reference our blob's data, and look something like:

```
blob:http://localhost:8000/24316d18-2f2e-454d-85b3-
f45af1c846c1
```

At this point, it's business as usual:

```
var worker = new Worker( url );

worker.addEventListener('message', function(e) {
    console.log(e.data); // Hello from the worker
}, false);

worker.postMessage(''); // begin
```

Should you need to remove a blob URL, the `removeObjectURL()` method should do the trick.

```
window.URL.revocateObjectURL( url );
```

Vendor Prefixes

If working along, you'll find that this code doesn't work as expected. At the time of this writing, `window.BlobBuilder` and `window.URL` require vendor prefixes for Webkit and Mozilla.

```
var blob = new BlobBuilder();  
  
// becomes  
var blob = new WebKitBlobBuilder();
```

Let's make things easier on ourselves:

```
if ( !window.BlobBuilder ) {  
    window.BlobBuilder = window.WebKitBlobBuilder ||  
        window.MozBlobBuilder;  
}  
  
if ( !window.URL ) {  
    window.URL = window.webkitURL || window.mozURL;  
}
```

This snippet will determine if `window.BlobBuilder` and `window.URL` exist. If not, we create the official method on the `window` object, and make it equal to either Webkit or Mozilla's implementation.

Final Code

```
if ( !window.BlobBuilder ) {  
    window.BlobBuilder = window.WebKitBlobBuilder ||  
        window.MozBlobBuilder;  
}
```

```
if ( !window.URL ) {  
    window.URL = window.webkitURL || window.mozURL;  
}  
  
if ( !window.BlobBuilder || !window.URL || !window.Worker ) >  
    return;  
  
var blob = new BlobBuilder(),  
    url,  
    worker;  
  
blob.append( "addEventListener('message', function() {  
    postMessage('Hello from the worker.');//  
}, false)" );  
  
url = window.URL.createObjectURL( blob.getBlob() );  
  
worker = new Worker( url );  
  
worker.addEventListener('message', function(e) {  
    console.log(e.data); // Hello from the worker.  
}, false);  
  
worker.postMessage('');
```

Inline Workers

[HTML5Rocks](#) has an excellent example of using a template to store a worker inline. This is infinitely more convenient than embedding the worker's content within a string.

To take advantage of this technique, you only need to create a template, which contains your worker.

```
<!doctype html>  
<html>  
<head>  
    <title>Web Workers</title>
```

```
</head>
<body>

<script id="worker" type="app/worker">
    addEventListener('message', function() {
        postMessage('What up, sucka.');
    }, false);
</script>

</body>
</html>
```

Next, rather than creating a new instance of `BlobBuilder()` and appending a string, we fetch the contents of the template as a string, and append it.

```
<!doctype html>
<html>
<head>
    <title>Web Workers</title>
</head>
<body>

<script id="worker" type="app/worker">
    addEventListener('message', function() {
        postMessage('What up, sucka.');
    }, false);
</script>

<script> (function() {
    if ( !window.BlobBuilder ) {
        window.BlobBuilder = window.WebKitBlobBuilder ||           ▶
        window.MozBlobBuilder;
    }

    if ( !window.URL ) {
        window.URL = window.webkitURL || window.mozURL;
```

ROCK*
TIP

As long as the type attribute of the script tag is one that the browser does not recognize, it won't be parsed.



```
}

var blob = new BlobBuilder(),
    url,
    worker;

// Append our template to the blob
blob.append(
  document.querySelector('#worker').textContent
);

url = window.URL.createObjectURL( blob.getBlob() );

worker = new Worker( url );

worker.addEventListener('message', function(e) {
  console.log(e.data); // What up, sucka.
}, false);

worker.postMessage('');

})();

</script>

</body>
</html>
```

That makes for a much cleaner solution, doesn't it?

Dealing with Errors

Certainly, you'll need to keep an eye on any potential errors. The **error** event will fire when an error within the associated worker occurs.

```
worker.addEventListener('error', function(e) {
  console.log(e);
}, false);
```

The **event** object will contain a handful of useful properties that we can use to pinpoint when and how the error occurred, most notably:

- **lineno** – The line number where the error occurred. For inline workers, this number will be relative to the starting line of the worker.
- **filename** – The name of the file where the error occurred.
- **message** – A description of the error.

```
▼ ErrorEvent
  bubbles: false
  cancelBubble: false
  cancelable: true
  clipboardData: undefined
  ▶ currentTarget: Worker
  defaultPrevented: false
  eventPhase: 0
  filename: "blob:http%3A%2F%2Flocalhost%3A8000/1755e616-b6ad-4f7d-9ef3-4d547d45080c"
  lineno: 3
  message: "Uncaught ReferenceError: y is not defined"
  returnValue: true
  ▶ srcElement: Worker
  ▶ target: Worker
  timeStamp: 1326494507924
  type: "error"
  ▶ __proto__: ErrorEvent
```

```
worker.addEventListener('error', function(e) {
  document.body.appendChild(
    document.createTextNode(
      'An error occurred in ' + unescape(e.filename) +           ▶
      ' on line ' + e.lineno + ': ' + e.message
    )
  ), false);

// An error occurred in blob:http://localhost:8000/               ▶
  461a6e60-4f97-459f-a8c8-63ba1a31174d on line 3:                ▶
    Uncaught ReferenceError: y is not defined
```

Or, if your desired error message is more complex, you might use a template approach:

```
worker.addEventListener('error', function(e) {
  var error =
```

```
'An error occurred in {{filename}} on line {{lineno}}: ▶
{{message}}'
.replace(
/{{(.+?)}}/g,
function(reg, match) {
    return unescape(e[match]);
}
);

document.body.textContent = error;
}, false);

// An error occurred in blob:http://localhost:8000/ ▶
461a6e60-4f97-459f-a8c8-63ba1a31174d on line 3: ▶
Uncaught ReferenceError: y is not defined
```

Summary

Ultimately, you shouldn't reach for a web worker unless you can describe exactly why one is needed to the person sitting next to you. That's a good rule of thumb. Maybe you're updating a local database extensively, or need to use canvas to manipulate video on the page. The possibilities are endless.

15

Tools, Folks, and Blogs

We now come to the obligatory roundup chapter. In many ways, this chapter may prove to be the most beneficial to you. Learning which developers and blogs to follow can have a massive effect on your learning. Always surround yourself with the people you wish to emulate.

Tools

HTML5 Boilerplate



“HTML5 Boilerplate is the professional badass’s base HTML/CSS/Javascript template for a fast, robust and future-proof site. **”**

Think of [HTML5 Boilerplate](#) as an amalgamation of countless recommended front-end best practices. It's not a framework, but a template. That means, even if you choose not to use everything within it, you can still pick and choose any fragments that you'd like to use in your projects.

Modernizr

Modernizr is an open-source JavaScript library that helps you build the next generation of HTML5 and CSS3-powered websites.

[Modernizr](#) performs massive amounts of feature detection, so that you can focus less on browser quirks, and more on providing the best possible experience to all visitors of your application. It's a necessity for modern front-end development. Use it.

haz.io

The screenshot shows the haz.io homepage with a green header and a white content area. At the top right, there's a link to 'Find me on GitHub'. The main content area contains four columns of browser support data:

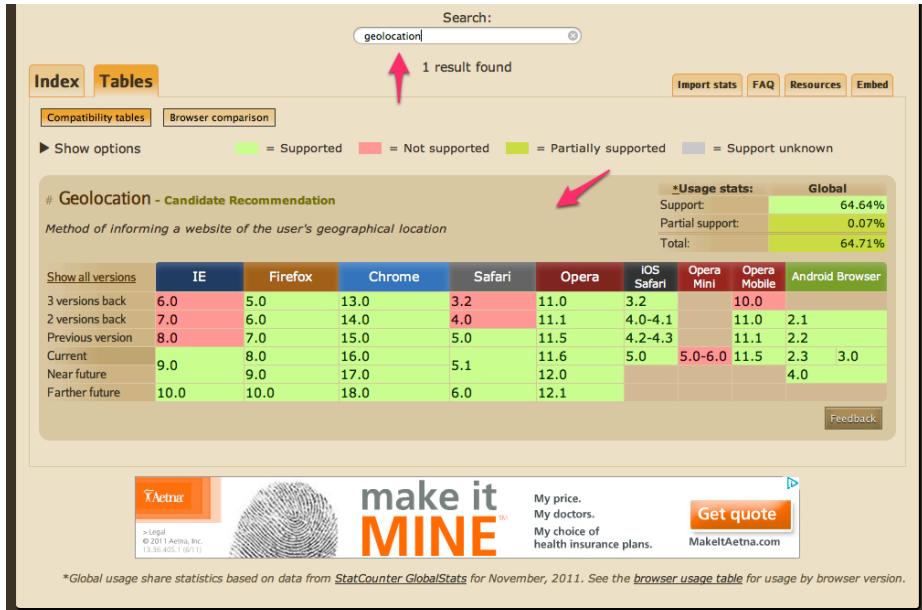
- CSS3** (Leftmost column): A list of CSS3 properties with 'YES' or 'NO' status indicators.
- HTML5**: A list of HTML5 features with 'YES' or 'NO' status indicators.
- Input Types**: A list of HTML5 input types with 'YES' or 'NO' status indicators.
- Input Attributes**: A list of HTML5 input attributes with 'YES' or 'NO' status indicators.
- Miscellaneous**: A list of various browser features with 'YES' or 'NO' status indicators.

Each row in the tables includes a small circular icon indicating the status: green for 'YES', yellow for 'MAYBE', and red for 'NO'.

[haz.io](#) uses Modernizr to filter through all of the new HTML5 and CSS3 features, and provide an overview of a browser's support

level. It's a nice bookmark to have, when testing new browsers and devices.

Can I Use?

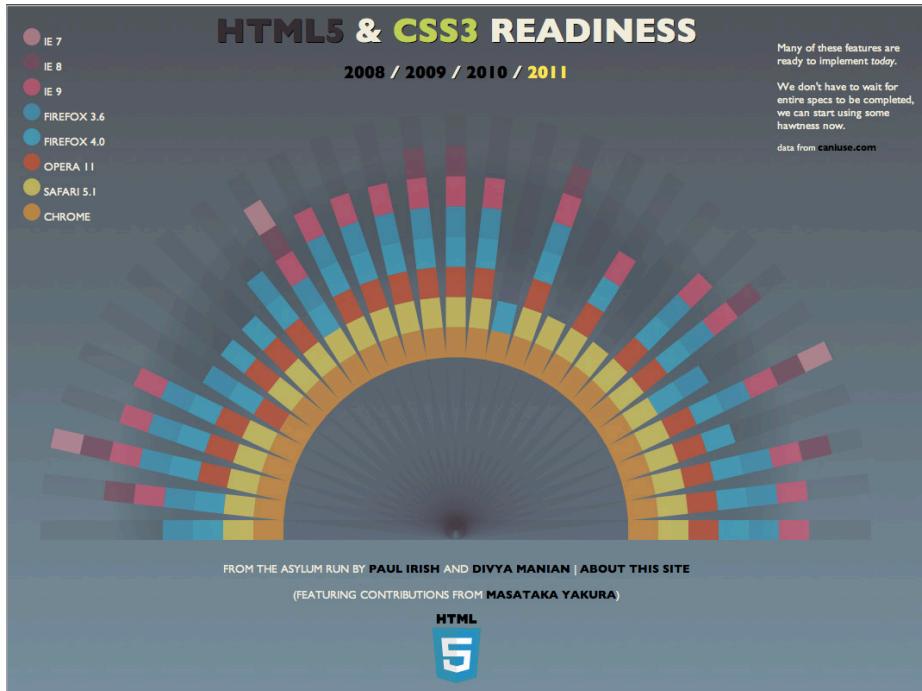


The screenshot shows the CanIUse.com interface for the 'geolocation' API. At the top, there's a search bar with 'geolocation' typed in. Below it, a message says '1 result found'. To the right of the search bar are buttons for 'Import stats', 'FAQ', 'Resources', and 'Embed'. On the left, there are tabs for 'Index' and 'Tables'. Underneath these are 'Compatibility tables' and 'Browser comparison' buttons. A legend indicates: green = Supported, red = Not supported, yellow = Partially supported, and grey = Support unknown. The main content area is titled '# Geolocation - Candidate Recommendation' and describes it as 'Method of informing a website of the user's geographical location'. It features a large table comparing browser support across various versions. To the right of the table is a 'Usage stats' section with a global summary: Support (64.64%), Partial support (0.07%), and Total (64.71%). At the bottom of the page, there's a footer with the Aetna logo, a fingerprint graphic, and text about health insurance plans. It also includes a 'Get quote' button and the URL 'MakeItAetna.com'.

The state of the web changes at an alarming pace. Sometimes, it's difficult to determine which browsers support which APIs or CSS3 properties. [CanIUse.com](#) serves as the perfect reference in this situations. I referred to it continuously while writing this book.

HTML5 and CSS3 Readiness

CanIUse.com is excellent, but sometimes, you need a more visual overview of what is and isn't possible. While still using data acquired from CanIUse, you might prefer the simplicity (and beauty) of [this site's](#) approach.

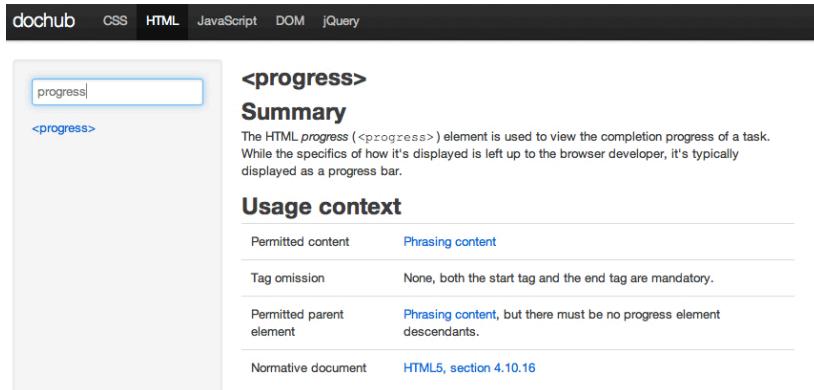


HTML5 Pattern

Description	Pattern	Testfield	Source
Credit Card Number	[0-9]{13,16}	<input type="text"/>	by dipser
Amex Credit Card	[0-9]{4} *[0-9]{6} *[0-9]{5}	<input type="text"/>	by Bad
Diners Club Card	^(30 36 38){2})([0-9]{12})\$	<input type="text"/>	by regexlib.com
ICQ UIN	((1-9))+(?:-?d){4,}	<input type="text"/>	by dipser & Flobose
Alpha-Numeric	[a-zA-Z0-9]+	<input type="text"/>	by dipser
Domain like "abc.de"	^((a-zA-Z0-9) (a-zA-Z0-9-){0,61}(a-zA-Z0-9) ?.)+([a-zA-Z]{2,6})\$	<input type="text"/>	by Unknown

Not everyone is comfortable with regular expressions (though I highly recommend [learning them](#)). For these folks, [HTML5 Pattern](#) contains a list of common patterns which can be copy-and-pasted into your HTML5 forms.

Dochub



The screenshot shows a search bar at the top with the word "progress". Below it is a code snippet: "<progress>". To the right, there's a section titled "Summary" which contains a brief description of the <progress> element. Below the summary is a section titled "Usage context" with two rows of information. The first row compares "Permitted content" (which is "Phrasing content") and "Tag omission" (where both start and end tags are mandatory). The second row compares "Permitted parent element" (which is "Phrasing content" with no progress element descendants) and "Normative document" (which is "HTML5, section 4.10.16").

Think of [Dochub](#) as a beautiful and crazy-fast resource for documentation. It's absolutely one that should be bookmarked. Even better, it provides documentation for CSS, JavaScript, HTML, jQuery, and the DOM.

HTML5 Demos

Demo	Support	Technology
Simple class manipulation		classlist
Storage events		storage
dataset (data-* attributes)		dataset
History API using pushState		history
Browser based file reading Not part of HTML5		file-api
Drag files directly into your browser Not directly part of HTML5		file-api dnd
Simple chat client		websocket
Two videos playing in sync		video
Interactive canvas gradients		canvas
Canvas & Video		video canvas

Remy Sharp maintains a large number of demos related to the new HTML5 APIs. It's a helpful site when you require a quick overview and demo of a particular API.

Polyfills

Web Storage (LocalStorage and SessionStorage)

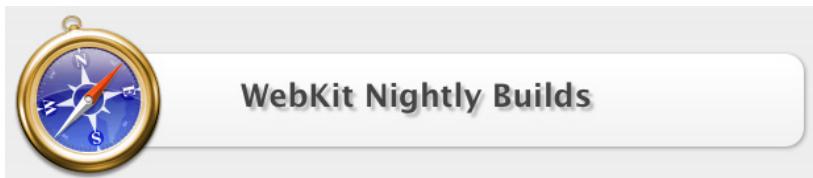
- [storage polyfill](#) by remy sharp
- [sessionstorage](#) by andrea giammarchi
- [Amplify.js](#) by appendTo *HTML5 API with fallbacks for HTML4 browsers (including IE6)*
- [realStorage](#) *HTML5 API is a subset of overall API*
- [YUI3 CacheOffline](#) by YUI team

Non HTML5 API Solutions

- [ssw](#) by matthias schäfer
- [\\$.store](#) by rodney rehm
- [lawnchair](#) by brian leroux
- [jStorage](#)
- [webstorage](#) by ryan westphal
- [store.js](#) by marcus westin
- [PersistJS](#) by paul duncan

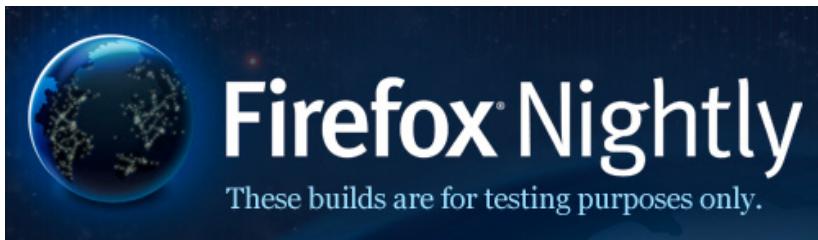
Within the Modernizr wiki, Paul Irish maintains a list of the most popular HTML5 polyfills.

Webkit Nightlies



Be sure to use the nightly version of Webkit to test cutting edge features that haven't yet been officially released to the public.

Mozilla Nightlies



The same goes for [Firefox nightlies!](#)

Folks

The following folks — in no specific order — are incredibly passionate about HTML5. Remember: surround yourself with the people you aspire to be like.

Paul Irish

- **Job Title:** Developer Evangelist for Google, jQuery Team Member
- **Website:** paulirish.com
- **Twitter:** [@paul_irish](https://twitter.com/paul_irish)
- **Known For:** Creator of HTML5 Boilerplate and Modernizr

Addy Osmani

- **Job Title:** AOL
- **Website:** addyosmani.com
- **Twitter:** [@addy_osmani](https://twitter.com/addy_osmani)

Chris Coyier

- **Job Title:** Wufoo Designer
- **Website:** css-tricks.com
- **Twitter:** [@chriscoyier](https://twitter.com/chriscoyier)
- **Known For:** CSS-Tricks

Remy Sharp

- **Job Title:** Developer
- **Website:** remysharp.com
- **Twitter:** [@rem](https://twitter.com/rem)
- **Known For:** Introducing HTML5, jQuery for Designers

Bruce Lawson

- **Job Title:** Opera Web Evangelist
- **Website:** brucelawson.co.uk
- **Twitter:** [@brucel](https://twitter.com/brucel)
- **Known For:** Introducing HTML5

Nicole Sullivan

- **Job Title:** Web Developer
- **Website:** stubbornella.org/content
- **Twitter:** [@stubbornella](https://twitter.com/stubbornella)
- **Known For:** OOCSS

Nicolas Gallagher

- **Job Title:** Web Developer
- **Website:** nicolasgallagher.com
- **Twitter:** [@necolas](https://twitter.com/necolas)
- **Known For:** Normalize.css, HTML5 Boilerplate

Jeremy Keith

- **Job Title:** Web Developer
- **Website:** adactio.com
- **Twitter:** [@adactio](https://twitter.com/adactio)
- **Known For:** HTML5 for Designers, Bulletproof AJAX

Peter Lubbers

- **Job Title:** Web Developer
- **Website:** runlaketahoe.blogspot.com
- **Twitter:** [@peterlubbers](https://twitter.com/peterlubbers)
- **Known For:** Pro HTML5 Programming, Founder of San Francisco HTML5 User Group

Lea Verou

- **Job Title:** Web Developer
- **Website:** lea.verou.me
- **Twitter:** [@leaverou](https://twitter.com/leaverou)
- **Known For:** Dabblet.com, Prefix-free

Divya Manian

- **Job Title:** Web Developer
- **Website:** nimbu.in
- **Twitter:** [@divya](https://twitter.com/divya)
- **Known For:** HTML5 Readiness, HTML5 Boilerplate, HTML5 Please

Not to appear too conceited, but if you'd like to follow this author, say hello on Twitter; I'm [@jeffrey_way](https://twitter.com/jeffrey_way).

Sites

- Nettuts+ — net.tutsplus.com
- HTML5 Rocks — www.html5rocks.com
- CSS-Tricks — css-tricks.com
- Smashing Magazine — smashingmagazine.com
- W3C — w3c.org
- WHATWG — whatwg.org
- Developers WHATWG — developers.whatwg.org
- Mozilla — developer.mozilla.org/en/HTML/HTML5
- Move The Web Forward — movethewebforward.org
- Dive into HTML5 — diveintohtml5.info
- HTML5 Doctor — html5doctor.com

Subscription-Based

- Tuts+ Premium – tutsplus.com
- CodeSchool – codeschool.com
- TreeHouse – teamtreehouse.com
- Lynda – lynda.com

Mailing Lists

- HTML5 Weekly – html5weekly.com
- W3C HTML5 – lists.w3.org/Archives/Public/public-html-comments
- Web Design Weekly – web-design-weekly.com
- JavaScript Weekly – javascriptweekly.com

CLOSING NOTES

Closing Notes

The truth is, there is so much more to HTML5 than I've covered in this book. We didn't touch on WebSockets, Server-Sent Events, or taking your applications offline, to name a few. To fit everything in properly would have required a thousand pages, and at that point you might as well get your information from the [HTML5 spec](#) itself. Admittedly, that can be a bit (or very) overwhelming.

Instead, as succinctly as possible, my goal was to describe what I consider to be the most exciting parts of HTML5 in a manner that is easiest to understand for everyone. I hope I didn't fail; if I did, please keep it to yourself!

Hopefully, the mix of JavaScript and jQuery snippets was helpful to you. Particularly as more and more users are being introduced to JavaScript through jQuery first (nothing wrong with that), it's helpful (and fun) to learn how to write code in both styles.

But your training is not yet finished, young padawan (second required tech book Star Wars reference fulfilled). There is still much to learn.

And the best part? You (yes, you) have a say in the future of HTML5. Don't for a second think that you don't. Join the various [W3C mailing lists](#), and become active in the community... right now. Ask questions — lots of questions (even dumb ones).

“ *The only stupid questions are the ones left unasked.* **”** — Albert Einstein

What's most amazing about our industry is the fact that, even if we didn't earn a living as web designers and developers, we'd still do it for free (though we'd certainly never admit this to our employers). This is specifically why countless programmers work all day, only to come home and contribute to open source projects... for free.

About The Author

[Jeffrey Way](#) is a developer evangelist and instructor who works for Envato. When not managing one of the most popular web development blogs in the world, [Nettuts+](#), or creating courses for [Tuts+ Premium](#), he enjoys playing with cutting edge web technologies, and finding new ways to teach complicated subjects and concepts, so that anyone can understand.



He is also the author of two other Rockable books: [*From Photoshop to HTML*](#) and [*Theme Tumblr Like a Pro*](#), and can typically be found scanning Twitter when he should really be getting back to work.

DECODING HTML5

Hi, I'm *Jeffrey Way*, a web developer and the author of *Decoding HTML5*. There's been a lot of talk about HTML5 in the industry—the good and the bad, what it has and does not have.

I wanted to dive straight into HTML5 and see what it has to offer us. The result is *Decoding HTML5*. This book deciphers the vast and confusing HTML5 spec and transforms it into something that the everyday designer or developer can immediately pick up and understand.

In this book I'll focus less on the politics of HTML5 (though we will touch on it), and more on the ways to immediately integrate it—and its friends—into your web projects.

If you're in need of a book that will get you up and running with many of the new tags, form elements, and JavaScript APIs as quickly as possible, then you've come to the right place!

Join me as I jump in and start *Decoding HTML5*.

ROCKABLE*

