SMASHING MAGAZINE

Mastering HTML5

Imprint

Version 1: July 2012

ISBN: 978-3-943075-35-9

Cover Design: Ricardo Gimenes

PR & Press: Stephan Poppe

eBook Strategy: Thomas Burkert

Technical Editing: Thomas Burkert

Idea & Concept: Smashing Media GmbH

ABOUT SMASHING MAGAZINE

Smashing Magazine is an online magazine dedicated to Web designers and developers worldwide. Its rigorous quality control and thorough editorial work has gathered a devoted community exceeding half a million subscribers, followers and fans. Each and every published article is carefully prepared, edited, reviewed and curated according to the high quality standards set in Smashing Magazine's own publishing policy. Smashing Magazine publishes articles on a daily basis with topics ranging from business, visual design, typography, front-end as well as back-end development, all the way to usability and user experience design. The magazine is — and always has been — a professional and independent online publication neither controlled nor influenced by any third parties, delivering content in the best interest of its readers. These guidelines are continually revised and updated to assure that the quality of the published content is never compromised.

ABOUT SMASHING MEDIA GMBH

Smashing Media GmbH is one of the world's leading online publishing companies in the field of Web design. Founded in 2009 by Sven Lennartz and Vitaly Friedman, the company's headquarters is situated in southern Germany, in the sunny city of Freiburg im Breisgau. Smashing Media's lead publication, Smashing Magazine, has gained worldwide attention since its emergence back in 2006, and is supported by the vast, global Smashing community and readership. Smashing Magazine had proven to be a trustworthy online source containing high quality articles on progressive design and coding techniques as well as recent developments in the Web design industry.

About this eBook

The Web changes everyday and, as a Web-developer, you are probably eager to keep up with the various techniques that help optimizing your workflow. This eBook "Mastering HTML5" explains to you the facts and myths of HTML5, shows how to use local storage on websites, teaches how to optimize images with HTML5 canvas and how to sync content with HTML5 video.

Table of Contents

Learning to Love HTML5

By Louis Lazaris

It seems that new resources and articles for teaching and promoting HTML5 are popping up almost daily. We've been given HTML5 templates in the form of the HTML5 boilerplate and HTML5 Reset (although they both go beyond just HTML5 stuff). We've got a plethora of books to choose from that cover HTML5 and its related technologies. We've got shivs, galleries, and a physician to help heal your HTML5 maladies. And don't forget the official spec.

From my own vantage point — aside from a few disputes about what the term "HTML5" should and shouldn't mean — the web design and development community has for the most part embraced all the new technologies and semantics with a positive attitude.

While it's certainly true that HTML5 has the potential to change the web for the better, the reality is that these kinds of major changes can be difficult to grasp and embrace. I'm personally in the process of gaining a better understanding of the subtleties of HTML5's various new features, so I thought I would discuss some things associated with HTML5 that appear to be somewhat confusing, and maybe this will help us all understand certain aspects of the language a little better, enabling us to use the new features in the most practical and appropriate manner possible.

The Good (and Easy) Parts

The good stuff in HTML5 has been discussed pretty solidly in a number of sources including books by Bruce Lawson, Jeremy Keith, and Mark Pilgrim,

to name a few. The benefits gained from using HTML5 include improved semantics, reduced redundancies, and inclusion of new features that minimize the need for complex scripting to achieve standard tasks (like input validation in forms, for example).

I think those are all commendable improvements in the evolution of the web's markup language. Some of the improvements, however, are a little confusing, and do seem to be a bit revolutionary, as opposed to evolutionary, the latter of which is one of the design principles on which HTML5 is based. Let's look at a few examples, so we can see how flexible and valuable some of the new elements really are — once we get past some of the confusion.

An <article> Isn't Just an Article

Among the additions to the semantic elements are the new **<section>** and **<article>** tags, which will replace certain instances of semantically meaningless ** ** tags that we're all accustomed to in XHTML. The problem arises when we try to decipher how these tags should be used.

Someone new to the language would probably assume that an **<article>** element would represent a single article like a blog post. But this is not always the case.

Let's consider a blog post as an example, which is the same example used in the spec. Naturally, we would think a blog post marked up in HTML5 would look something like this:

```
<article>
<h1>Title of Post</h1>
<p>Content of post...</p>

<p>Content of post...</p>

</article>
<section>
  <section>
  <p>Comment by: Comment Author</p>
  <p>Comment #1 goes here...</p>
  </section> <section> <p>Comment by: Comment Author</p>
  <p>Comment #2 goes here...</p>
  </section> <section> <p>Comment by: Comment Author</p>
  <p>Comment #3 goes here...</p>
  </section>
</section>
```

For brevity, I've left out some of the other HTML5 tags that might go into such an example. In this example, the **<article>** tags wrap the entire article, then the "section" below it wraps all the comments, each of which is in its own "section" element.

It would not be invalid or wrong to structure a blog post like this. But according to the way **<article>** is described in the spec, the **<article>** element should wrap the entire article *and the comments*. Additionally, each comment itself could be wrapped in **<article>** tags that are nested within the main **<article>** tag.

Below is a screen grab from the spec, with **<article>** tags indicated:

Here is that same blog post, but showing some of the comments:

```
<article>
 <header>
  <h1>The Very First Rule of Life</h1>
  <p><time pubdate datetime="2009-10-09T14:28-08:00"></time></p>
 </header>
 <p>If there's a microphone anywhere near you, assume it's hot and
 sending whatever you're saying to the world. Seriously.</p>
 <p>...</p>
 <section>
  <h1>Comments</h1>
  <article>
   <footer>
    <p>Posted by: George Washington</p>
    <p><time pubdate datetime="2009-10-10T19:10-08:00"></time></p>
   </footer>
   <p>Yeah! Especially when talking about your lobbyist friends!</p>
  </article>
  <article>
   <footer>
    <p>Posted by: George Hammond</p>
    <p><time pubdate datetime="2009-10-10T19:15-08:00"></time></p>
   </footer>
   <p>Hey, you have the same first name as me.</p>
  </article>
 </section>
</article>
```

Article tags can be nested inside article tags — a concept that seems confusing at first glance.

So, an **`<article>`** element can have other **`<article>`** elements nested inside it, thus complicating how we naturally view the word “article”. Bruce Lawson, co-author of Introducing HTML5, attempts to clear up the confusion in this interview:

> *"Think of **<article>** not in terms of print, like "newspaper article" but as a discrete entity like "article of clothing" that is complete in itself, but can also mix with other articles to make a wider ensemble."*
> *— Bruce Lawson*

So keep in mind that you can nest **<article>** elements and an **<article>** element can contain more than just article content. Bruce's explanation above is very good and is the kind of HTML5 education that's needed to help us understand how these new elements can be used.

Section or Article?

Probably one of the most confusing things to figure out when creating an HTML5 layout is whether or not to use **<article>** or **<section>**. As I write this sentence, I can honestly say I don't know the difference without actually looking up what the spec says or referencing one of my HTML5 books. But slowly it's becoming more clear. I think Jeremy Keith defines **<article>** best on page 67 of *HTML5 for Web Designers*:

> *"The **article** element is [a] specialized kind of **section**. Use it for self-contained related content... Ask yourself if you would syndicate the content in an RSS or Atom feed. If the content still makes sense in that context, then **article** is probably the right element to use."*
> *— Jeremy Keith, HTML5 for Web Designers*

Keith's explanation helps a lot, but then he goes on to explain that the difference between **<article>** and **<section>** is quite small, and it's up to each developer to decide how these elements should be used. And

adding to the confusion is the fact that you can have multiple articles within sections and multiple sections within articles.

As a result, you might wonder why we have both. The main difference is that the **<article>** element is designed for syndication, whereas the **<section>** element is designed for document structure and portability. This simple way to view the differences certainly helps make the two new elements a little more distinct. The important thing to keep in mind here is that, despite our initial confusion, these changes, when more widely adopted, are going to help developers and content creators to improve the way they work and the way content is shared.

Headers and Footers (Plural!)

Two other elements introduced in HTML5 are the **<header>** and **<footer>** elements. On the surface, these seem pretty straightforward. For years we've marking up our website headers and footers with ** **, ** ** or similar. This is great for DOM manipulation and styling, because we can target these elements directly. But they mean nothing semantically.

> *"The **div** element has no defined semantics, and the **id** attribute has no defined semantics. (User agents are not allowed to infer any meaning from the value of the **id** attribute.)"*
>
> — *Mark Pilgrim,* Dive Into HTML5

HTML5's introduction of **<header>** and **<footer>** elements is the perfect way to remedy this problem of semantics, especially for such often-used elements. But these elements are not as straightforward as they seem.

Technically speaking, if every website in the world added one **<header>** and one **<footer>** to each of their pages, this would be perfectly valid HTML5. But these new elements are not just limited to use as a "website header" and "website footer".

A header is designed to mark up introductory or navigational aids, and a footer is designed to contain information about the containing element. For example, if you used the footer element as the footer for a full web page, then in that case copyright, policy links, and related content might be appropriate for it to hold. A header on the same page might contain a logo and navigation bar.

But the same page might also include multiple **<section>** elements. Each of those sections is permitted to contain its own header and/or footer element. Keith sums up the purpose of these elements well:

> *"A **header** will usually appear at the top of a document or section, but it doesn't have to. It is defined by its content… rather than its position."*
> *"Like the **header** element, **footer** sounds like it's a description of position, but as with **header**, this isn't the case."*
> *— Jeremy Keith, HTML5 for Web Designers*

And the spec adds to Keith's clarification by noting:

> *"The header element is not sectioning content; it doesn't introduce a new section."*
> — *The header element in the HTML5 specification*

These explanations help dispel any false assumptions we might have about these new elements, so we can understand how these elements can be used. Really, this method of dividing pages into portable and syndicatible content is just adding semantics to what content creators and developers have been doing for years.

Headings Down A Different Path

Prior to HTML5, heading tags (**<h1>** through **<h6>**) were pretty easy to understand. Over the years, some best practices have been adopted in order to improve semantics, SEO, and accessibility. Generally, we've become accustomed to including a single **<h1>** element on each page, with the other heading elements following sequentially without gaps (although sometimes it would be necessary to reverse the order).

With the introduction of HTML5, to use the new structural elements we need to rethink the way we view the structure of our pages.

Here are some things to note about the changes in heading/document structure in HTML5

- Instead of a single **<h1>** element per page, HTML5 best practice encourages up to one **<h1>** for each **<section>** element (or other section defined by some other means)
- Although we're permitted to start a section with an **<h2>** (or lower-ranked) element, it's strongly encouraged to start each **<section>** with an **<h1>** element to help sections become portable
- Document nodes are created by sections, not headings (unlike previous versions of HTML)

- An **<hgroup>** element is used to group related heading elements that you want to act as a single heading for a defined or implied section; **<hgroup>** is not used on every set of headings, only those that act as a single unit outside of adjacent content
- To see if you're structuring your document correctly, you can use the HTML5 Outliner
- Despite the above points, whatever heading/document structure you used in HTML4 or XHTML will still be valid HTML5

So, although the old way we structure pages does not amount to invalid HTML5, our view of what constitutes "best practice" document structure is changing for the better.

Block or Inline? Neither! (Sort of...)

For layout and styling purposes, CSS developers are accustomed to HTML elements (for styling and layout purposes) being defined under one of two categories: Block elements and inline elements (although you could divide those two into further categories). This understanding simplified our expectations of an element's display on any given page, making it easier (once we grasp the difference between the two) to style and maneuver the elements.

HTML5 evolves this concept to include multiple categories, none of which is block or inline. Well, theoretically, block and inline elements still exist, but they do so under different labels. Now the different categories of elements include:

- Grouping Content
- Text-Level Semantics

- Sectioning Content
- Form Elements
- Embedded Content

I certainly welcome this kind of improvement to more appropriately categorize elements, and I think developers will adapt well to these changes, but it is important that we promote proper nomenclature to ensure minimal confusion over how these elements will display by default. Of all the areas discussed in this article, however, I think this one is the easiest to grasp and accept.

Conclusion

While this summarizes some of what I've learned in my study of HTML5, a far better way for anyone to learn about these new features to the markup is to pick up a book on the topic. I highly recommend one of those mentioned in the article, or you can read Mark Pilgrim's book online.

These new elements and concepts don't have to be confusing. We can take the time to study them carefully, avoiding confusion and dispelling myths. This will help us enjoy the benefits of these new elements as soon as possible, and will help developers and content creators pave the way towards a more meaningful web — a web that, to paraphrase Jeremy Keith, 'wouldn't exist without markup'.

HTML5: The Facts And The Myths

By Bruce Lawson

You can't escape it. Everyone's talking about HTML5. it's perhaps the most hyped technology since people started putting rounded corners on everything and using unnecessary gradients. In fact, a lot of what people call HTML5 is actually just old-fashioned DHTML or AJAX. Mixed in with all the information is a lot of misinformation, so here, JavaScript expert Remy Sharp and Opera's Bruce Lawson look at some of the myths and sort the truth from the common misconceptions.

First, Some Facts

Once upon a time, there was a lovely language called HTML, which was so simple that writing websites with it was very easy. So, everyone did, and the Web transformed from a linked collection of physics papers to what we know and love today.

Most pages didn't conform to the simple rules of the language (because their authors were rightly concerned more with the message than the medium), so every browser had to be forgiving with bad code and do its best to work out what its author wanted to display.

In 1999, the W3C decided to discontinue work on HTML and move the world toward XHTML. This was all good, until a few people noticed that the work to upgrade the language to XHTML2 had very little to do with the real Web. Being XML, the spec required a browser to stop rendering if it encountered an error. And because the W3C was writing a new language that was better than simple old HTML, it deprecated elements such as **`<img>`** and **`<a>`**.

A group of developers at Opera and Mozilla disagreed with this approach and presented a paper to the W3C in 2004 arguing that, “We consider Web Applications to be an important area that has not been adequately served by existing technologies… There is a rising threat of single-vendor solutions addressing this problem before jointly-developed specifications.”

The paper suggested seven design principles:

1. Backwards compatibility, and a clear migration path.
2. Well-defined error handling, like CSS (i.e. ignore unknown stuff and move on), compared to XML’s “draconian” error handling.
3. Users should not be exposed to authoring errors.
4. Practical use: every feature that goes into the Web-applications specifications must be justified by a practical use case. The reverse is not necessarily true: every use case does not necessarily warrant a new feature.
5. Scripting is here to stay (but should be avoided where more convenient declarative mark-up can be used).
6. Avoid device-specific profiling.
7. Make the process open. (The Web has benefited from being developed in the open. Mailing lists, archives and draft specifications should continuously be visible to the public.)

The paper was rejected by the W3C, and so Opera and Mozilla, later joined by Apple, continued a mailing list called Web Hypertext Application Technology Working Group (WHATWG), working on their proof-of-concept specification. The spec extended HTML4 forms, until it grew into a spec called Web Applications 1.0, under the continued editorship of Ian Hickson, who left Opera for Google.

In 2006, the W3C realized its mistake and decided to resurrect HTML, asking WHATWG for its spec to use as the basis of what is now called HTML5.

Those are the historical facts. Now, let's look at some hysterical myths.

The Myths

"I CAN'T USE HTML5 UNTIL 2012 (OR 2022)"

This is a misconception based on the projected date that HTML5 will reach the stage in the W3C process known as Candidate Recommendation (REC). The WHATWG wiki says this:

> *For a spec to become a REC today, it requires two 100% complete and fully interoperable implementations, which is proven by each successfully passing literally thousands of test cases (20,000 tests for the whole spec would probably be a conservative estimate). When you consider how long it takes to write that many test cases and how long it takes to implement each feature, you'll begin to understand why the time frame seems so long.*

So, by definition, the spec won't be finished until you can use *all of it*, and in two browsers.

Of course, what really matters is the bits of HTML5 that are already supported in the browsers. Any list will be out of date within about a week because the browser makers are innovating so quickly. Also, much of the new functionality can be replicated with JavaScript in browsers that don't yet have support. The **`<canvas>`** property is in all modern browsers and will be

in Internet Explorer 9, but it can be faked in old versions of IE with the excanvas library. The **<video>** and **<audio>** properties can be faked with Flash in old browsers.

HTML5 is designed to degrade gracefully, so with clever JavaScript and some thought, all content should be available on older browsers.

"MY BROWSER SUPPORTS HTML5, BUT YOURS DOESN'T"

There's a myth that HTML5 is some monolithic, indivisible thing. It's not. It's a collection of features, as we've seen above. So, in the short term, you cannot say that a browser supports everything in the spec. And when some browser or other does, it won't matter because we'll all be much too excited about the next iteration of HTML by then.

What a terrible mess, you're thinking? But consider that CSS 2.1 is not yet a finished spec, and yet we all use it each and every day. We use CSS3, happily adding **`border-radius`**, which will soon be supported everywhere, while other aspects of CSS3 aren't supported anywhere at all.

Be wary of browser "scoring" websites. They often test for things that have nothing to do with HTML5, such as CSS, SVG and even Web fonts. What matters is what you need to do, what's supported by the browsers your client's audience will be using and how much you can fake with JavaScript.

HTML5 LEGALIZES TAG SOUP

HTML5 is a lot more forgiving in its syntax than XHTML: you can write tags in uppercase, lowercase or a mixture of the two. You don't need to self-close tags such as `img`, so the following are both legal:

```
<img src="nice.jpg" />
<img src="nice.jpg">
```

You don't need to wrap attributes in quotation marks, so the following are both legal:

```
<img src="nice.jpg">
<img src=nice.jpg>
```

You can use uppercase or lowercase (or mix them), so all of these are legal:

```
<IMG SRC=nice.jpg>
<img src=nice.jpg>
<iMg SrC=nice.jpg>
```

This isn't any different from HTML4, but it probably comes as quite a shock if you're used to XHTML. In reality, if you were serving your pages as a combination of text and HTML, rather than XML (and you probably were, because Internet Explorer 8 and below couldn't render true XHTML), then it never mattered anyway: the browser never cared about trailing slashes, quoted attributes or case—only the validator did.

So, while the syntax appears to be looser, the actual parsing rules are much tighter. The difference is that there is no more tag soup; the specification describes exactly what to do with invalid mark-up so that all conforming browsers produce the same DOM. If you've ever written JavaScript that has to walk the DOM, then you're aware of the horrors that inconsistent DOMs can bring.

This error correction is no reason to churn out invalid code, though. The DOM that HTML5 creates for you might not be the DOM you want, so ensuring that your HTML5 validates is still essential. With all this new stuff, overlooking a small syntax error that stops your script from working or that makes your CSS unstylish is easy, which is why we have HTML5 validators.

Far from legitimizing tag soup, HTML5 consigns it to history. Souper.

“I NEED TO CONVERT MY XHTML WEBSITE TO HTML5”

Is HTML5’s tolerance of looser syntax the death knell for XHTML? After all, the working group to develop XHTML 2 was disbanded, right?

True, the XHTML 2 group was disbanded at the end of 2009; it was working on an unimplemented spec that competed with HTML5, so having two groups was a waste of W3C resources. But XHTML 1 was a finished spec that is widely supported in all browsers and that will continue to work in browsers for as long as needed. Your XHTML websites are therefore safe.

HTML5 KILLS XML

Not at all. If you need to use XML rather than HTML, you can use XHTML5, which includes all the wonders of HTML5 but which must be in well-formed XHTML syntax (i.e. quoted attributes, trailing slashes to close some elements, lowercase elements and the like.)

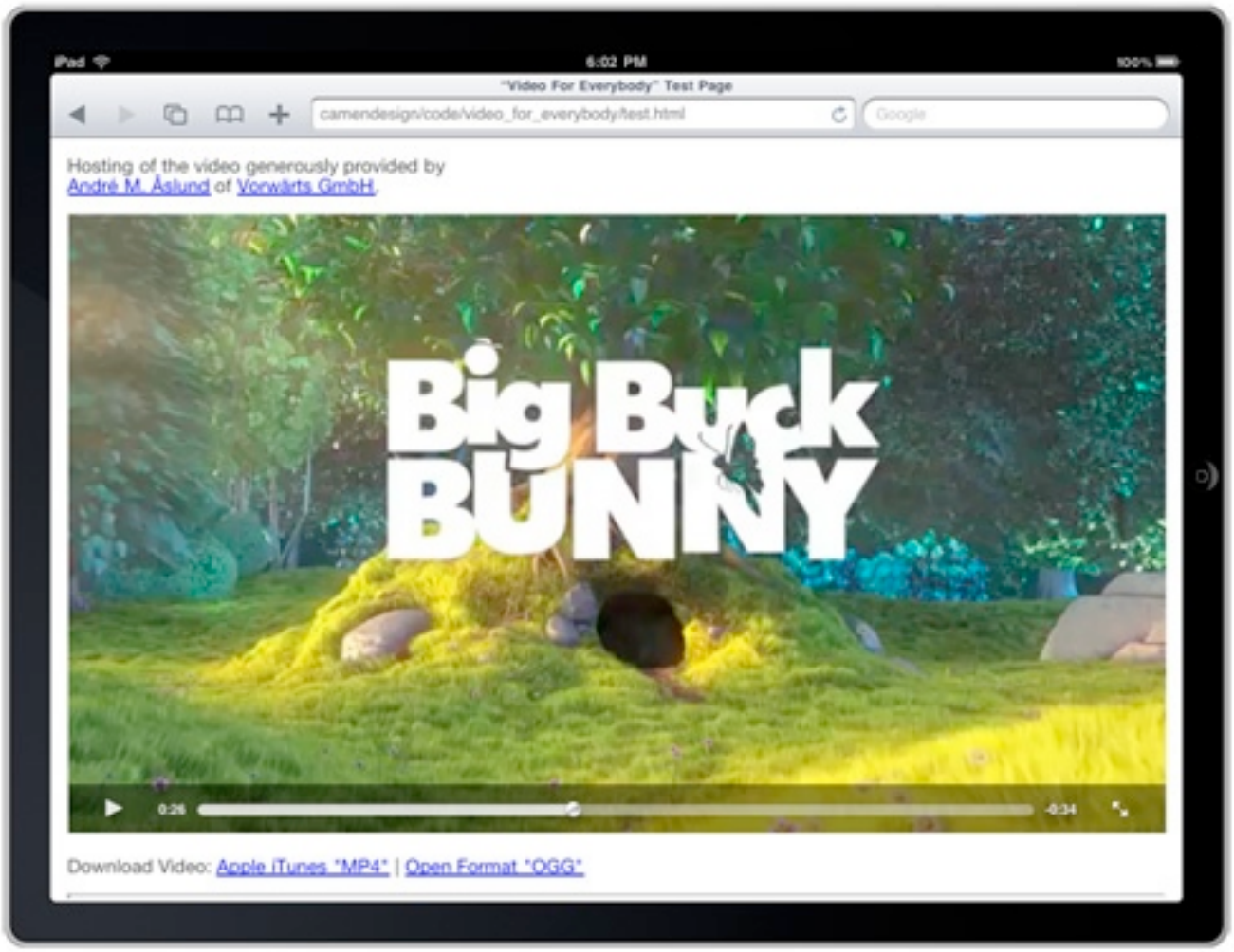
Actually, you can’t use *all* the wonders of HTML5 in XHTML5: **`<noscript>`** won’t work. But you’re not still using that, are you?

HTML5 WILL KILL FLASH AND PLUG-INS

The **`<canvas>`** tag allows scripted images and animations that react to the keyboard and that therefore can compete with some simpler uses of Adobe Flash. HTML5 has native capability for playing video and audio.

Just as when CSS Web fonts weren’t widely supported and Flash was used in sIFR to fill the gaps, Flash also saves the day by making HTML5 video backwards-compatible. Because HTML5 is designed to be “fake-able” in older browsers, the mark-up between the video tags is ignored by browsers that understand HTML5 and is rendered by older browsers. Therefore,

embedding fall-back video with Flash is possible using the old-school **<object>** or **<embed>** tags, as pioneered by Kroc Camen is his article "Video for Everybody!" (see the screenshot below).



But not all of Flash's use cases are usurped by HTML5. There is no way to do digital rights management in HTML5; browsers such as Opera, Firefox and Chrome allow visitors to save video to their machines with a click of the context menu. If you need to prevent video from being saved, you'll need to use plug-ins. Capturing input from a user's microphone or camera is currently only possible with Flash (although a `<device>` element is being

specified for “post-5″ HTML), so if you’re keen to write a Chatroulette killer, HTML5 isn’t for you.

HTML5 IS BAD FOR ACCESSIBILITY

A lot of discussion is going on about the accessibility of HTML5. This is good and to be welcomed: with so many changes to the basic language of the Web, ensuring that the Web is accessible to people who cannot see or use a mouse is vital. Also vital is building in the solution, rather than bolting it on as an afterthought: after all, many (most?) authors don’t even add alternate text to images, so out-of-the-box accessibility is much more likely to succeed than relying on people to add it.

This is why it’s great that HTML5 adds native controls for things like sliders (**`<input type=range>`**, currently supported in Opera and Webkit browsers) and date pickers (**`<input type=date>`**, Opera only)—see Bruce’s HTML5 forms demo)—because previously we had to fake these with JavaScript and images and then add keyboard support and WAI-ARIA roles and attributes.

The **`<canvas>`** tag is a different story. It is an Apple invention that was reverse-engineered by other browser makers and then retrospectively specified as part of HTML5, so there is no built-in accessibility. If you’re just using it for eye-candy, that’s fine; think of it as an image, but without any possibility of alternate text (some additions to the spec have been suggested, but nothing is implemented yet). So, ensure that any information you deliver via **`<canvas>`** supplements more accessible information elsewhere.

Text in a **`<canvas>`** becomes simply pixels, just like text in images, and so is invisible to assistive technology and screen readers. Consider using the W3C graphics technology Scalable Vector Graphics (SVG) instead,

especially for things such as dynamic graphs and animating text. SVG is supported in all the major browsers, including IE9 (but not IE8 or below, although the SVGweb library can fake SVG with Flash in older browsers).

The situation with **<video>** and **<audio>** is promising. Although not fully specified (and so not yet implemented in any browsers), a new <track> element has been included in the HTML5 spec that allows timed transcripts (or karaoke lyrics or captions for the deaf or subtitles for foreign-language media) to be associated with multimedia. It can be faked in JavaScript. Alternatively (and better for search engines), you could include transcripts directly on the page below the video and use JavaScript to overlay captions, synchronized with the video.

"AN HTML5 GURU WILL HOLD MY HAND AS I DO IT THE FIRST TIME"

If only this were true. However, the charming Paul Irish and lovely Divya Manian will be as good as there for you, with their HTML5 Boilerplate, which is a set of files you can use as templates for your projects. Boilerplate brings in the JavaScript you need to style the new elements in IE; pulls in jQuery from the Google Content Distribution Network (CDN), but with fall-back links to your server in case the CDN server is down.

It adds mark-up that is adaptable to iOS, Android and Opera Mobile; and adds a CSS skeleton with a comprehensive reset style sheet. There's even an *.htaccess* file that serves your HTML5 video with the right MIME types. You won't need all of it, and you're encouraged to delete the stuff that's unnecessary to your project to avoid bloat.

Local Storage And How To Use It On Websites

By Christian Heilmann

Storing information locally on a user's computer is a powerful strategy for a developer who is creating something for the Web. In this article, we'll look at how easy it is to store information on a computer to read later and explain what you can use that for.

Adding State To The Web: The "Why" Of Local Storage

The main problem with HTTP as the main transport layer of the Web is that it is stateless. This means that when you use an application and then close it, its state will be reset the next time you open it. If you close an application on your desktop and re-open it, its most recent state is restored.

This is why, as a developer, you need to store the state of your interface somewhere. Normally, this is done server-side, and you would check the user name to know which state to revert to. But what if you don't want to force people to sign up?

This is where local storage comes in. You would keep a key on the user's computer and read it out when the user returns.

C Is For Cookie. Is That Good Enough For Me?

The classic way to do this is by using a cookie. A cookie is a text file hosted on the user's computer and connected to the domain that your website runs

on. You can store information in them, read them out and delete them. Cookies have a few limitations though:

- They add to the load of every document accessed on the domain.
- They allow up to only 4 KB of data storage.
- Because cookies have been used to spy on people's surfing behavior, security-conscious people and companies turn them off or request to be asked every time whether a cookie should be set.

To work around the issue of local storage — with cookies being a rather dated solution to the problem — the WHATWG and W3C came up with a few local storage specs, which were originally a part of HTML5 but then put aside because HTML5 was already big enough.

Using Local Storage In HTML5-Capable Browsers

Using local storage in modern browsers is ridiculously easy. All you have to do is modify the **`localStorage`** object in JavaScript. You can do that directly or (and this is probably cleaner) use the **`setItem()`** and **`getItem()`** method:

```
localStorage.setItem('favoriteflavor','vanilla');
```

If you read out the **`favoriteflavor`** key, you will get back "vanilla":

```
var taste = localStorage.getItem('favoriteflavor');
// -> "vanilla"
```

To remove the item, you can use — can you guess? — the **`removeItem()`** method:

```
localStorage.removeItem('favoriteflavor');
var taste = localStorage.getItem('favoriteflavor');
// -> null
```

That's it! You can also use **`sessionStorage`** instead of **`localStorage`** if you want the data to be maintained only until the browser window closes.

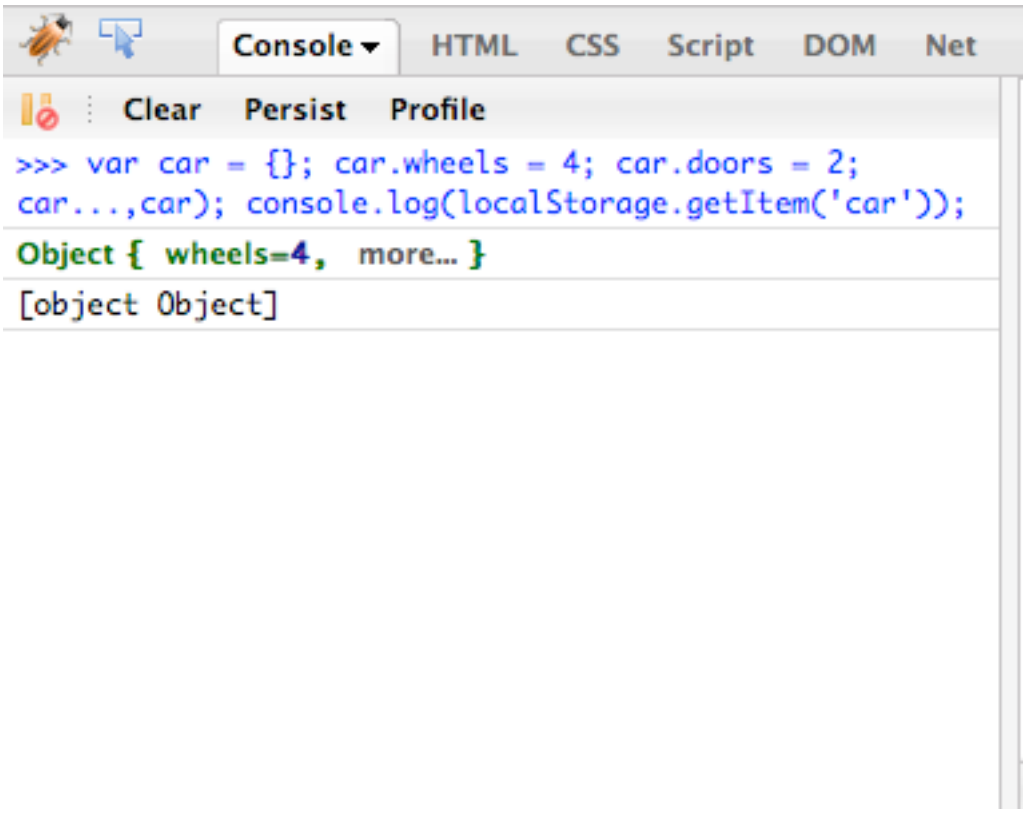
Working Around The "Strings Only" Issue

One annoying shortcoming of local storage is that you can only store strings in the different keys. This means that when you have an object, it will not be stored the right way.

You can see this when you try the following code:

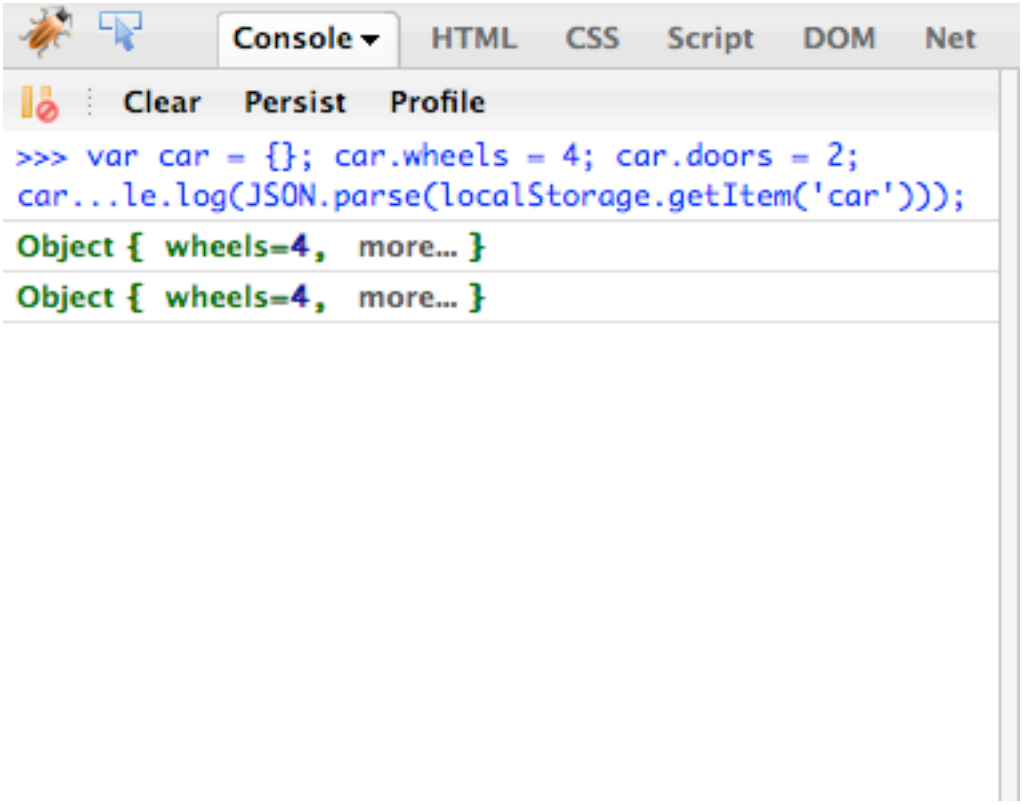
```
var car = {};
car.wheels = 4;
car.doors = 2;
car.sound = 'vroom';
car.name = 'Lightning McQueen';
console.log( car );
localStorage.setItem( 'car', car );
console.log( localStorage.getItem( 'car' ) );
```

Trying this out in the console shows that the data is stored as **`[object Object]`** and not the real object information:



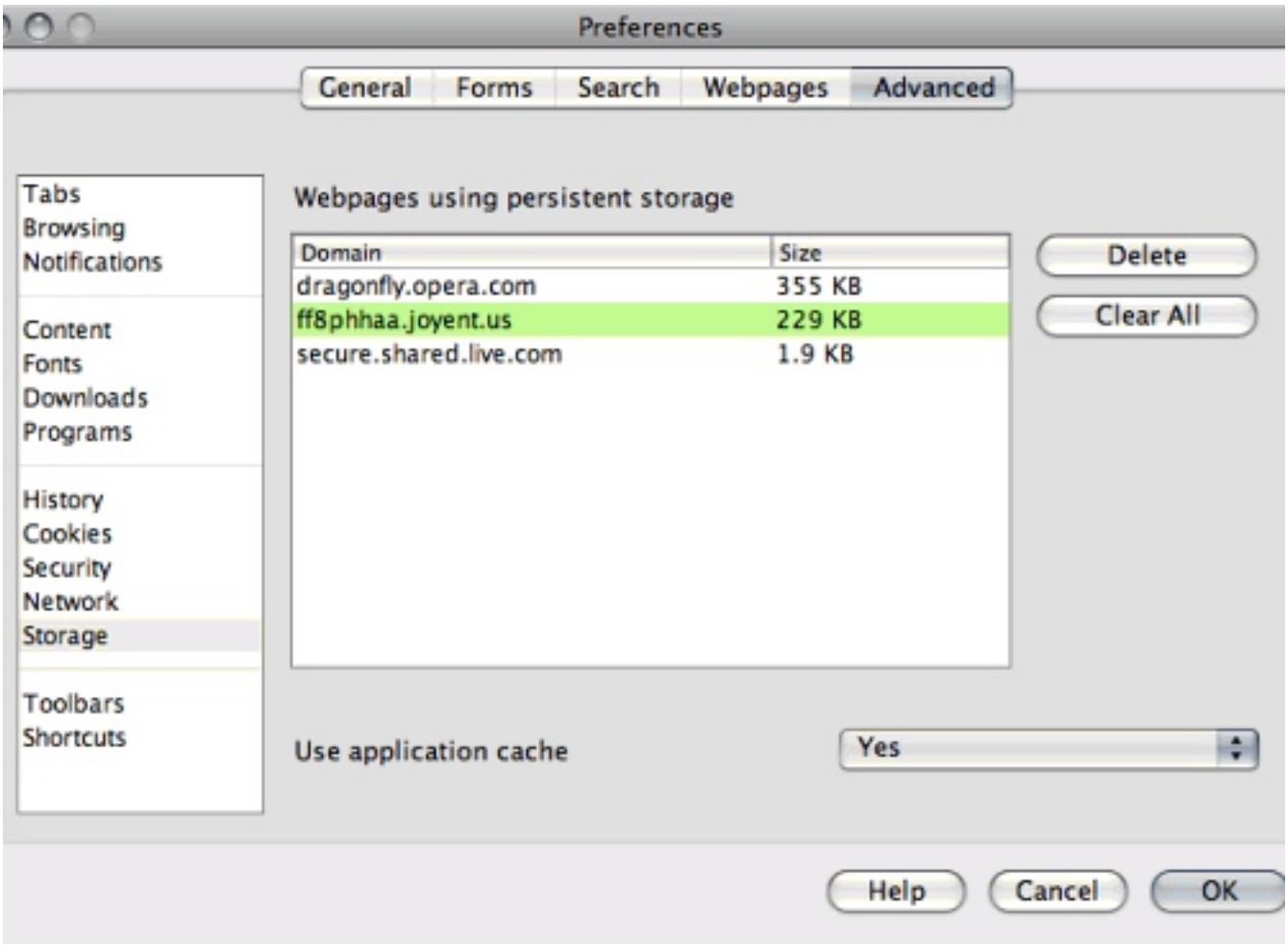
You can work around this by using the native **`JSON.stringify()`** and **`JSON.parse()`** methods:

```
var car = {};
car.wheels = 4;
car.doors = 2;
car.sound = 'vroom';
car.name = 'Lightning McQueen';
console.log( car );
localStorage.setItem( 'car', JSON.stringify(car) );
console.log( JSON.parse( localStorage.getItem( 'car' ) ) );
```



Where To Find Local Storage Data And How To Remove It

During development, you might sometimes get stuck and wonder what is going on. Of course, you can always access the data using the right methods, but sometimes you just want to clear the plate. In Opera, you can do this by going to *Preferences* → *Advanced* → *Storage*, where you will see which domains have local data and how much:



Doing this in Chrome is a bit more problematic, which is why we made this screencast.

Mozilla has no menu access so far, but will in future. For now, you can go to the Firebug console and delete storage manually easily enough.

So, that's how you use local storage. But what can you use it for?

Use Case #1: Local Storage Of Web Service Data

One of the first uses for local storage that I discovered was caching data from the Web when it takes a long time to get it. My World Info entry for the Event Apart 10K challenge shows what I mean by that.

When you call the demo the first time, you have to wait up to 20 seconds to load the names and geographical locations of all the countries in the world from the Yahoo GeoPlanet Web service. If you call the demo a second time, there is no waiting whatsoever because — you guessed it — I've cached it on your computer using local storage.

The following code (which uses jQuery) provides the main functionality for this. If local storage is supported and there is a key called **`thewholefrigginworld`**, then call the **`render()`** method, which displays the information. Otherwise, show a loading message and make the call to the Geo API using **`getJSON()`**. Once the data has loaded, store it in **`thewholefrigginworld`** and call **`render()`** with the same data:

```
if(localStorage &&
localStorage.getItem('thewholefrigginworld')){

render(JSON.parse(localStorage.getItem('thewholefrigginworld')
));
} else {
  $('#list').html('<p class="load">'+loading+'</p>');
  var query = 'select centroid,woeid,name,boundingBox'+
              ' from geo.places.children(0)'+
              ' where parent_woeid=1 and placetype="country"'+
              ' | sort(field="name")';
  var YQL = 'http://query.yahooapis.com/v1/public/yql?q='+
             encodeURIComponent(query)
+'&diagnostics=false&format=json';
  $.getJSON(YQL,function(data){
```

```
    if(localStorage){

localStorage.setItem('thewholefrigginworld',JSON.stringify(dat
a));
    }
    render(data);
  });
}
```

You can see the difference in loading times in this screencast.

The code for the world info is available on GitHub.

This can be extremely powerful. If a Web service allows you only a certain number of calls per hour but the data doesn't change all that often, you could store the information in local storage and thus keep users from using up your quota. A photo badge, for example, could pull new images every six hours, rather than every minute.

This is very common when using Web services server-side. Local caching keeps you from being banned from services, and it also means that when a call to the API fails for some reason, you will still have information to display.

`getJSON()` in jQuery is especially egregious in accessing services and breaking their cache, as explained in this blog post from the YQL team. Because the request to the service using **`getJSON()`** creates a unique URL every time, the service does not deliver its cached version but rather fully accesses the system and databases every time you read data from it. This is not efficient, which is why you should cache locally and use **`ajax()`** instead.

Use Case #2: Maintaining The State Of An Interface The Simple Way

Another use case is to store the state of interfaces. This could be as crude as storing the entire HTML or as clever as maintaining an object with the state of all of your widgets. One instance where I am using local storage to cache the HTML of an interface is the Yahoo Firehose research interface (screencast, source on GitHub):

The code is very simple — using YUI3 and a test for local storage around the local storage call:

```
YUI().use('node', function(Y) {
  if(('localStorage' in window) && window['localStorage'] !==
null){
    var key = 'lastyahoofirehose';
  <!--?php if(isset($_POST['sent']) || isset($_POST['moar']))
{?-->
    localStorage.setItem(key,Y.one('form').get('innerHTML'));
  <!--?php }else{ ?-->
  if(key in localStorage){

Y.one('#mainform').set('innerHTML',localStorage.getItem(key));
      Y.one('#hd').append('<p
class="message"><strong>Notice:</strong> We restored your last
search for you - not live data');
  }
  <!--?php } ?-->
  }
});</p>
```

You don't need YUI at all; it only makes it easier. The logic to generically cache interfaces in local storage is always the same: check if a "Submit" button has been activated (in PHP, Python, Ruby or whatever) and, if so, store the **`innerHTML`** of the entire form; otherwise, just read from local storage and override the **`innerHTML`** of the form.

The Dark Side Of Local Storage

Of course, any powerful technology comes with the danger of people abusing it for darker purposes. Samy, the man behind the "Samy is my hero" MySpace worm, recently released a rather scary demo called Evercookie, which shows how to exploit all kind of techniques, including local storage, to store information of a user on their computer even when cookies are turned off. This code could be used in all kinds of ways, and to date there is no way around it.

Research like this shows that we need to look at HTML5's features and add-ons from a security perspective very soon to make sure that people can't record user actions and information without the user's knowledge. An opt-in for local storage, much like you have to opt in to share your geographic location, might be in order; but from a UX perspective this is considered clunky and intrusive. Got any good ideas?

Optimize Images With HTML5 Canvas

By Sergey Chikuyonok

Images have always been the heaviest component of websites. Even if high-speed Internet access gets cheaper and more widely available, websites will get heavier more quickly. If you really care about your visitors, then spend some time deciding between good-quality images that are bigger in size and poorer-quality images that download more quickly. And keep in mind that modern Web browsers have enough power to enhance images right on the user's computer. In this article, I'll demonstrate one possible solution.

Let's refer to an image that I came across recently in my job. As you can see, this image is of stage curtains and has some (intentional) light noise:

Optimizing an image like this would be a real pain because it contains a lot of red (which causes more artifacts in JPEG) and noise (which creates awful artifacts in JPEG and is bad for PNG packing). The best optimization I could get for this image was 330 KB JPEG, which is quite much for a single image. So, I decided to do some experiments with image enhancement right in the user's browser.

If you look closely at this image, you'll see that it consists of two layers: the noise and the stage curtains. If we remove the noise, then the image shrinks to 70 KB in JPEG, which is really nice. Thus, our goal becomes to pass a noiseless image to the user and then add noise to the image right in the Web browser. This will greatly reduce the downloading time and make the Web page perform better.

In Photoshop, generating monochromatic noise is very easy: just go to **`Filter → Noise → Add Noise`**. But in the original image, the noise actually darkens some pixels (i.e. there are no white pixels). This brings a new challenge: to apply a noise layer with the "Multiply" blending mode on the stage image.

HTML5 Canvas

All modern Web browsers support the canvas element. While early canvas implementations offered only a drawing API, modern implementations allow authors to analyze and manipulate every image pixel. This can be done with the ImageData interface, which represents an image data's width, height and array of pixels.

The canvas pixel array is a plain array containing each pixels's RGBa data. Here is what that data array looks like:

```
pixelData = [pixel1_red, pixel1_green,
pixel1_blue, pixel1_alpha, pixel2_red,
pixel2_green, pixel2_blue, pixel2_alpha, …];
```

Thus, an image data array contains **`total_pixels×4`** elements. For example, a 200×100 image would contain 200×100×4 = 80,000 elements in this array.

To analyze and manipulate individual canvas pixels, we have to get image data from it, then modify the pixel array and then put data back into the canvas:

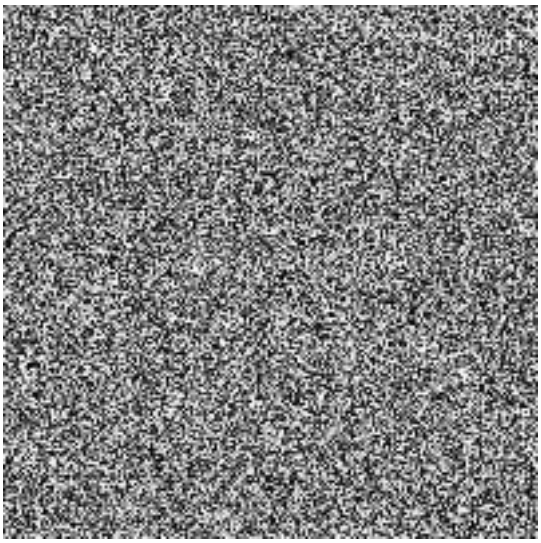
```
// Coordinates of image pixel that we will modify
var x = 10, y = 20;

// Create a new canvas
var canvas = document.createElement('canvas');
canvas.width = canvas.height = 100;
document.body.appendChild(canvas);

// Get drawing context
var ctx = canvas.getContext('2d');
// Get image data
var imageData = ctx.getImageData(0, 0, canvas.width,
canvas.height);
// Calculate offset for pixel
var offset = (x - 1 + (y - 1) * canvas.width) * 4;

// Set pixel color to opaque orange
imageData.data[offset]     = 255; // red channel
imageData.data[offset + 1] = 127; // green channel
imageData.data[offset + 2] = 0;   // blue channel
imageData.data[offset + 3] = 255; // alpha channel

// Put image data back into canvas
ctx.putImageData(imageData, 0, 0);
```

Generating Noise

Once we know how to manipulate individual canvas pixels, we can easily create noise layer. A simple function that generates monochromatic noise might look like this:

```
function addNoise(canvas) {
   var ctx = canvas.getContext('2d');

   // Get canvas pixels
   var imageData = ctx.getImageData(0, 0, canvas.width,
canvas.height);
   var pixels = imageData.data;

   for (var i = 0, il = pixels.length; i < il; i += 4) {
var color = Math.round(Math.random() * 255);

      // Because the noise is monochromatic, we should put the
same value in the R, G and B channels
      pixels[i] = pixels[i + 1] = pixels[i + 2] = color;

      // Make sure pixels are opaque
      pixels[i + 3] = 255;
   }

   // Put pixels back into canvas
   ctx.putImageData(imageData, 0, 0);
}

// Set up canvas
var canvas = document.createElement('canvas');
canvas.width = canvas.height = 200;
document.body.appendChild(canvas);

addNoise(canvas);
```

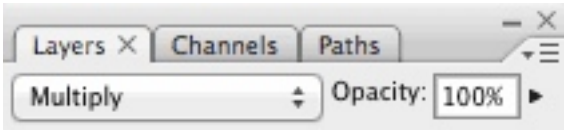
The result could look like this:



Pretty good for starters. But we can't just create a noise layer and place it above the scene image. Rather, we should blend it in "Multiply" mode.

Blend Modes

Anyone who has worked with Adobe Photoshop or any other advanced image editor knows what layer blend modes are:



Some folks regard image blend modes as some sort of rocket science, but in most cases the algorithms behind them are pretty simple. For example, here's what the Multiply blending algorithm looks like:

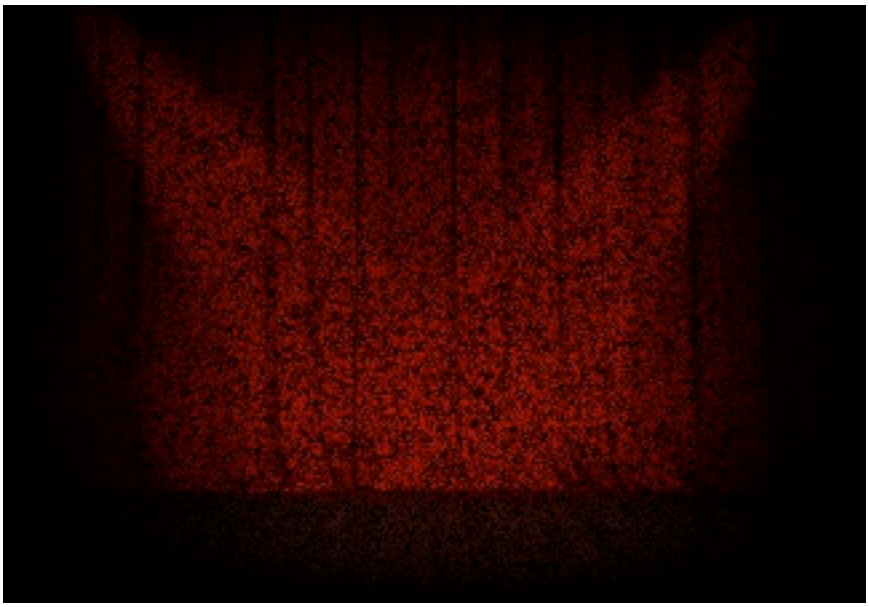
```
(colorA * colorB) / 255
```

That is, we have to multiply two colors (each channel's value) and divide it by 255.

Let's modify out code snippet: load the image, generate the noise and apply it using the "Multiply" blend mode:

```
// Load image. Waiting for onload event is important
var img = new Image;
img.onload = function() {
addNoise(img);
};
img.src = "stage-bg.jpg";
function addNoise(img) {
   var canvas = document.createElement('canvas');
   canvas.width = img.width;
   canvas.height = img.height;

   var ctx = canvas.getContext('2d');
   // Draw image on canvas to get its pixel data
   ctx.drawImage(img, 0, 0);
   // Get image pixels
   var imageData = ctx.getImageData(0, 0, canvas.width,
canvas.height);
   var pixels = imageData.data;
   for (var i = 0, il = pixels.length; i < il; i += 4) {
      // generate "noise" pixel
      var color = Math.random() * 255;

      // Apply noise pixel with Multiply blending mode for
each color channel
      pixels[i] =     pixels[i] * color / 255;
      pixels[i + 1] = pixels[i + 1] * color / 255;
      pixels[i + 2] = pixels[i + 2] * color / 255;
   }
   ctx.putImageData(imageData, 0, 0);
   document.body.appendChild(canvas);
}
```

The result will look like this:



Looks nice, but the noise is very rough. We have to apply transparency to it.

Alpha Compositing

The process of combining two colors with transparency is called “Alpha compositing.” In the simplest case of compositing, the algorithm would look like this:

```
colorA * alpha + colorB * (1 - alpha)
```

Here, **`alpha`** is the composition coefficient (transparency) from 0 to 1. Choosing which color will be the background (**`colorB`**) and which will be the overlay (**`colorA`**) is important. In this case, the background will be the curtains image, and the noise will be the overlay.

Let’s add one more argument for the **`addNoise()`** function, which will control the alpha blending and modify the main function to respect transparency while blending images:

```
var img = new Image;
   img.onload = function() {
   addNoise(img, 0.2); // pass 'alpha' argument
};
img.src = "stage-bg.jpg";

function addNoise(img, alpha) { // new 'alpha' argument
   var canvas = document.createElement('canvas');
   canvas.width = img.width;
   canvas.height = img.height;

   var ctx = canvas.getContext('2d');
   ctx.drawImage(img, 0, 0);

   var imageData = ctx.getImageData(0, 0, canvas.width,
canvas.height);
   var pixels = imageData.data, r, g, b;

   for (var i = 0, il = pixels.length; i < il; i += 4) {
      // generate "noise" pixel
      var color = Math.random() * 255;

      // Calculate the target color in Multiply blending mode
without alpha composition
      r = pixels[i] * color / 255;
      g = pixels[i + 1] * color / 255;
      b = pixels[i + 2] * color / 255;

      // alpha compositing
      pixels[i] =     r * alpha + pixels[i] * (1 - alpha);
      pixels[i + 1] = g * alpha + pixels[i + 1] * (1 - alpha);
      pixels[i + 2] = b * alpha + pixels[i + 2] * (1 - alpha);
   }

   ctx.putImageData(imageData, 0, 0);
   document.body.appendChild(canvas);
}
```

The result is exactly what we want: a noise layer applied to the background image in Multiply blending mode, with a transparency of 20%:

Optimization

While the resulting image looks perfect, the performance of this script is pretty poor. On my computer, it takes about 300 milliseconds (ms). On the average user's computer, it could take even longer. So, we have to optimize this script. Half of the code uses browser API calls such as for creating the canvas, getting the image data and sending it back, so we can't do much with that. The other half is the main loop for applying noise to the stage image, and it can be perfectly optimized.

Our test image size is 1293×897, which leads to 1,159,821 loop iterations. A pretty big number, so even small modifications could lead to significant performance boosts.

For example, in this cycle we calculated `1 - alpha` three times, but this is a static value. We should define a new variable outside of the **`for`** loop:

```
var alpha1 = 1 - alpha;
```

And then we would replace all occurrences of `1 - alpha` with **`alpha1`**.

Next, for the noise pixel generation, we use the **`Math.random() * 255`** formula. But a few lines later, we divide this value by 255, so **`r = pixels[i] * color / 255`**. Thus, we have no need to multiply and divide; we just use a random value. Here's what the main loop looks like after these tweaks:

```
var alpha1 = 1 - alpha;
for (var i = 0, il = pixels.length; i < il; i += 4) {
   // generate "noise" pixel
   var color = Math.random();

   // Calculate the target color in Multiply blending mode
without alpha composition
   r = pixels[i] * color;
   g = pixels[i + 1] * color;
   b = pixels[i + 2] * color;

   // Alpha compositing
   pixels[i] =     r * alpha + pixels[i] * alpha1;
   pixels[i + 1] = g * alpha + pixels[i + 1] * alpha1;
   pixels[i + 2] = b * alpha + pixels[i + 2] * alpha1;
}
```

After these little optimizations, the **`addNoise()`** function runs at about 240 ms (a 20% boost).

Remember that we have more than a million iterations, so every little bit counts. In the main loop, we're accessing the **`pixels`** array twice: once for blending and once for alpha compositing. But array access is too resource-

intensive, so we need to use an intermediate variable to store the original pixel value (i.e. we access the array once per iteration), like so:

```
var alpha1 = 1 - alpha;
var origR, origG, origB;

for (var i = 0, il = pixels.length; i < il; i += 4) {
   // generate "noise" pixel
   var color = Math.random();

   origR = pixels[i]
   origG = pixels[i + 1];
   origB = pixels[i + 2];

   // Calculate the target color in Multiply blending mode
without alpha composition
   r = origR * color;
   g = origG * color;
   b = origG * color;

   // Alpha compositing
   pixels[i] =     r * alpha + origR * alpha1;
   pixels[i + 1] = g * alpha + origG * alpha1;
   pixels[i + 2] = b * alpha + origB * alpha1;
}
```

This reduces the execution of this function down to 200 ms.

Extreme Optimization

An attentive user would notice that the stage curtains are red. In other words, the image data is defined in the red channel only. The green and blue ones are empty, so there's no need for them in the calculations:

```
for (var i = 0, il = pixels.length; i < il; i += 4) {
   // generate "noise" pixel
   var color = Math.random();

   origR = pixels[i]
   // Calculate the target color in Multiply blending mode
without alpha composition
   r = origR * color;

   // Alpha compositing
   pixels[i] = r * alpha + origR * alpha1;
}
```

With some high-school algebra, I came up with this formula:

```
for (var i = 0, il = pixels.length; i < il; i += 4) {
   pixels[i] = pixels[i] * (Math.random() * alpha + alpha1);
}
```

And the overall execution of the function is reduced to 100 ms, one-third of the original 300 ms, which is pretty awesome.

The **`for`** loop contains just one simple calculation, and you might think that we can do nothing more. Actually, we can.

During the execution of the loop, we calculate the random pixel value and apply it to original one. But we don't need to compute this random pixel on each iteration (remember, we have more than a million of them!). Rather, we can pre-calculate a limited set of random values and then apply them to original pixels. This will work because the generated value is… well, random. There's no repeating patterns of special cases—just random data.

The trick is to pick the right value's array size. It should be large enough to not produce visible repeating patterns on the image and small enough to be generated at a reasonable speed. During my experiments, the best random value's array length was 3.73 of the image width.

Now, let's generate an array with random pixels and then apply them to original image:

```
// Pick the best array length
var rl = Math.round(ctx.canvas.width * 3.73);
var randoms = new Array(rl);

// Pre-calculate random pixels
for (var i = 0; i < rl; i++) {
   randoms[i] = Math.random() * alpha + alpha1;
}

// Apply random pixels
for (var i = 0, il = pixels.length; i < il; i += 4) {
   pixels[i] = pixels[i] * randoms[i % rl];
}
```

This will cut down the execution time to 80 ms in Webkit browsers and have a significant boost in Opera. Also, Opera slows down in performance when the image data array contains float values, so we have to round them with fast bit-wise **OR** operator.

The final code snippet looks like this:

```
var img = new Image;
   img.onload = function() {
   addNoise(img, 0.2); // pass 'alpha' argument
};
img.src = "stage-bg.jpg";

function addNoise(img, alpha) {
   var canvas = document.createElement('canvas');
   canvas.width = img.width;
   canvas.height = img.height;

   var ctx = canvas.getContext('2d');
   ctx.drawImage(img, 0, 0);
```

```
    var imageData = ctx.getImageData(0, 0, canvas.width,
canvas.height);
    var pixels = imageData.data;
    var alpha1 = 1 - alpha;

    // Pick the best array length
    var rl = Math.round(ctx.canvas.width * 3.73);
    var randoms = new Array(rl);

    // Pre-calculate random pixels
    for (var i = 0; i < rl; i++) {
        randoms[i] = Math.random() * alpha + alpha1;
    }

    // Apply random pixels
    for (var i = 0, il = pixels.length; i < il; i += 4) {
        pixels[i] =  (pixels[i] * randoms[i % rl]) | 0;
    }

    ctx.putImageData(imageData, 0, 0);
    document.body.appendChild(canvas);
}
```

This code executes in about 80 ms on my laptop in Safari 5.1 for an image that is 1293×897, which is *really* fast. You can see the result online.

The Result

In my opinion, the result is pretty good:

- The image size was reduced from 330 KB to 70 KB, plus 1 KB of minified JavaScript. Actually, the image could be saved at a lower quality because the noise might hide some JPEG artifacts.

- This is a perfectly valid **progressive enhancement** optimization. Users with modern browsers will see a highly detailed image, while users with older browsers (such as IE 6) or with JavaScript disabled will still be able to see the image but with less detail.

The only downside of this approach is that the final image should be calculated each time the user visits the Web page. In some cases, you can store the final image as **`data:url`** in **`localStorage`** and restore it the next time the page loads. In my case, the size of the final image is 1.24 MB, so I decided not to store it and instead spend the default 5 MB of local storage on more useful data.

BLEND MODES PLAYGROUND

I've created a small playground that you can use as a starting point for your experiments and optimizations. It contains most Photoshop blend modes and opacity control. Feel free to copy any code found on this playground into your pages.

Conclusion

The technique you've just learned can be used in many different ways on modern Web pages. You don't need to create many heavy, detailed images and force users to download them. Highlights, shades, textures and more can all be generated very quickly right in the user's browser.

But use this technique wisely. Putting many unoptimized image generators on a single page could cause the browser to hang for a few seconds. You shouldn't use this technique if you don't really understand how it works or know what you're doing. Always prioritize performance ahead of cool technology.

Also, note that the curtains image in this article is for demonstration purposes only. You can achieve almost the same result — a noisy image — just by placing a semi-transparent noise texture above the scene image, as shown in this example. Anyway, the point of this article was to show you how to enhance images right in the Web browser while keeping their size as small as possible.

Syncing Content With HTML5 Video

One of the main changes from HTML4 to HTML5 is that the new specification breaks a few of the boundaries that browsers have been confined to. Instead of restricting user interaction to text, links, images and forms, HTML5 promotes multimedia, from a generic **`<object>`** element to a highly specified **`<video>`** and **`<audio>`** element, and with a rich API to access in pure JavaScript.

Native multimedia capability has a few benefits:

- **End users have full control over the multimedia.**
  The native controls of browsers allow users to save videos locally or email them to friends. Also, HTML5 video and audio are keyboard-enabled by default, which is a great accessibility benefit.
- **End users do not need to install a plug-in to play them.**
  The browser already has everything it needs to play movies and sound.
- **Video and audio content on the page can be manipulated.**
  They are simply two new elements like any other that can be styled, moved, manipulated, stacked and rotated.
- You can build your own controls using HTML, CSS and JavaScript.
  No new skills or development environment needed.
- **Interaction with the rest of the page is simple.**
  The multimedia API gives you full control over the video, and you can make the video react both to changes in the video itself and to the page around it.

Let's quickly recap how you can use native video in the browser, starting with the embedding task.

Embedding Video

This is old news. Embedding video in a document is as easy as adding a **<video>** element and pointing it to the source video. Adding a **controls** attribute gives you native controls:

```
<video src="chris.ogv" controls></video>
```

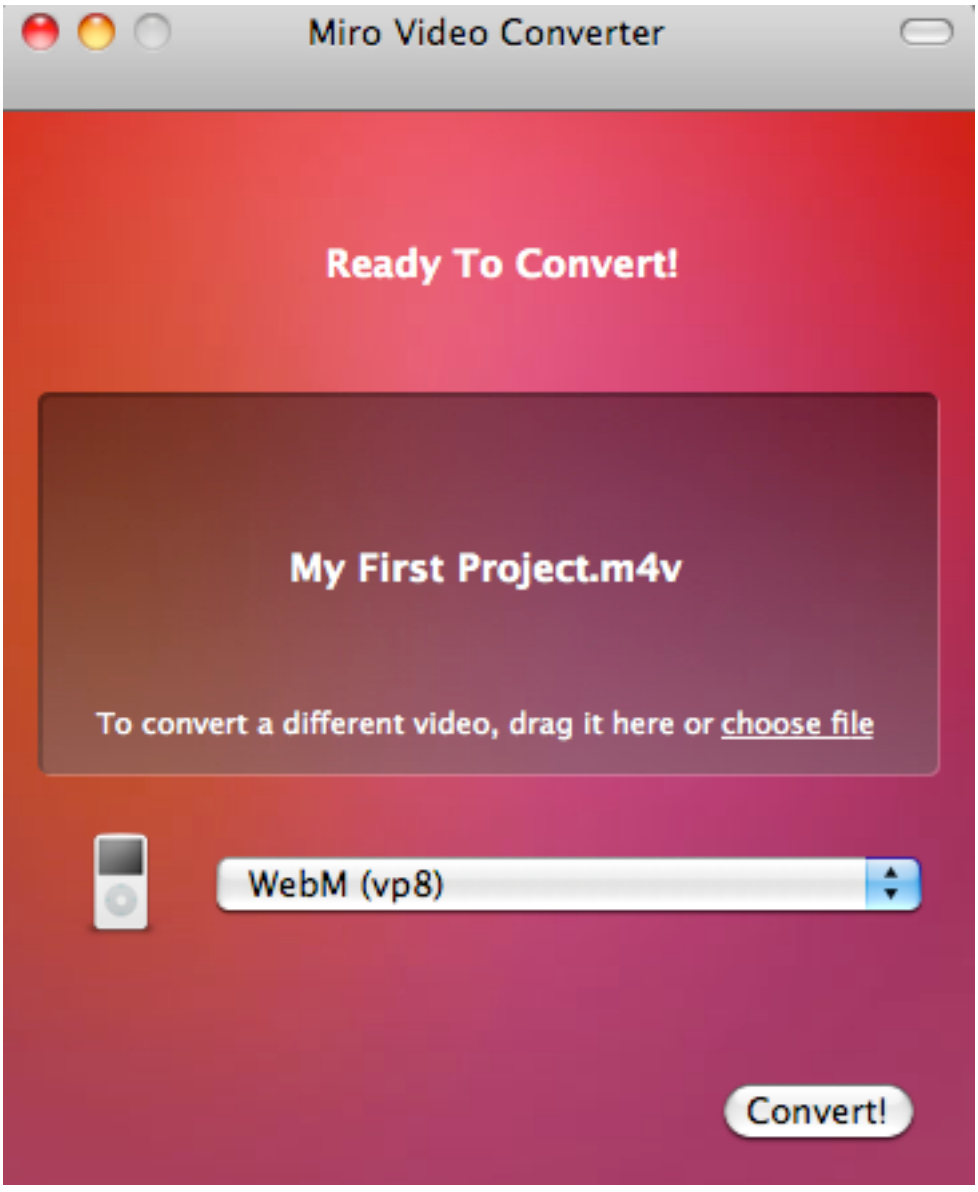
This is the theory, though. In the real world of intellectual property, corporate competition and device-specific solutions, we as developers have to jump through a few hoops:

```
<video controls="true" height="295" width="480">
  <!-- hello iOS, Safari and IE9 -->
  <source src="chris.mp4" type="video/mp4">
  <!-- Hello Chrome and Firefox (and Opera?) -->
  <source src="chris.webm" type="video/webm">
  <!-- Hello Firefox and Opera -->
  <source src="chris.ogv" type="video/ogg">
  <!-- Hello legacy -->
  Your browser does not support the video tag,
  <a href="http://www.youtube.com/watch?v=IhkUe_KryGY">
    check the video on YouTube
  </a>.
</video>
```

This shows how we need to deliver video in three formats in order to satisfy all of the different browsers out there. There are a few ways to accomplish this. Here's what I do…

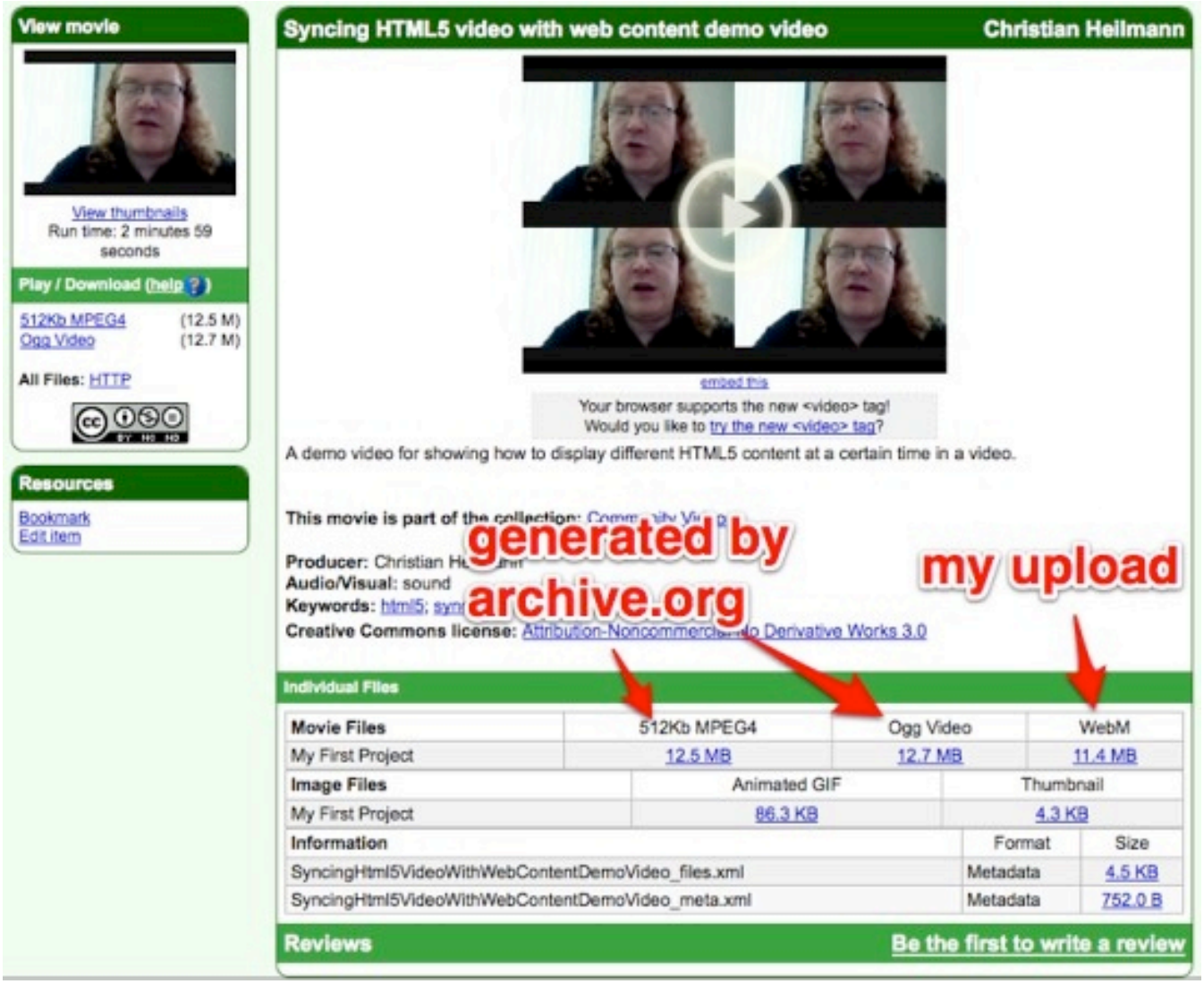
Convert Video With Miro Video Converter

Miro Video Converter is an open-source tool for Mac that makes converting videos dead easy. Simply drag the video to the tool, select WebM as the output format, and watch the progress. A few other converters for Windows and Linux are available, too.



Hosting And Automated Conversion On Archive.org

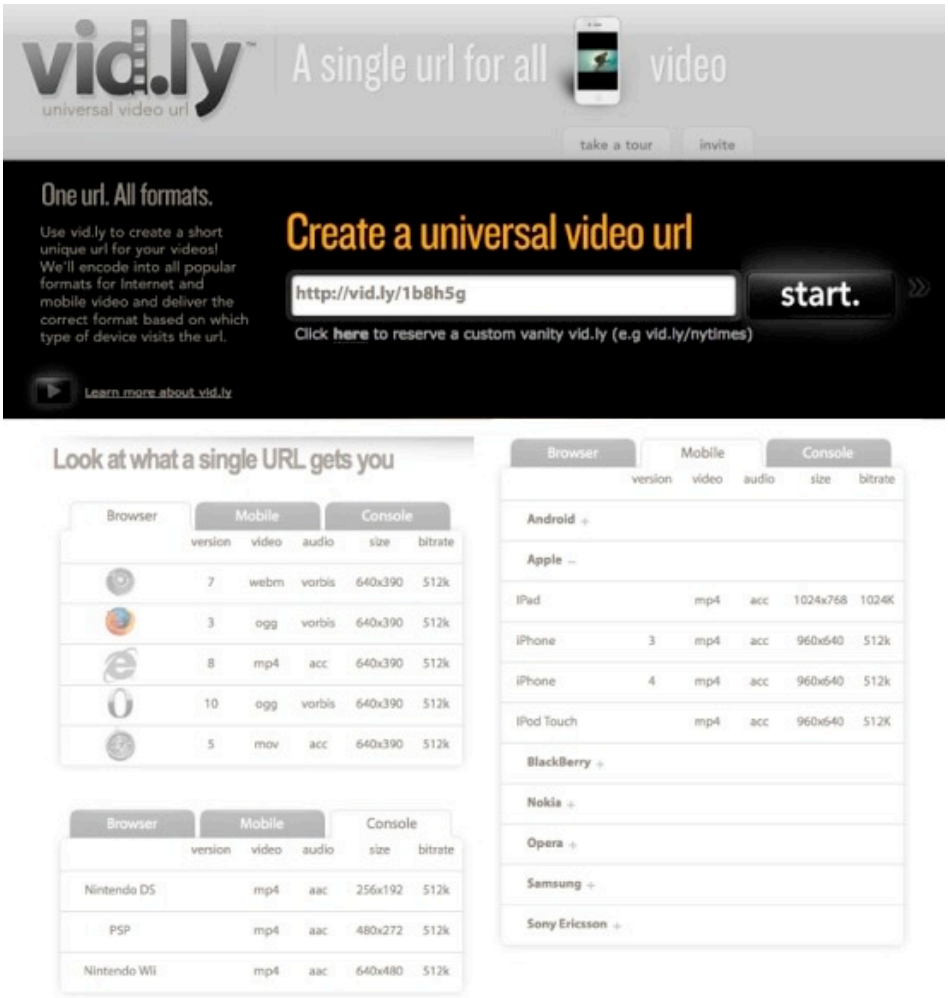
Because I license my videos with Creative Commons, I can use Archive.org to both host the videos and convert the WebM versions to MP4 and OGV. Simply upload your video and wait about an hour. Reload the page, and the server pixies at Archive.org will have created the other two formats (and also a cool animated GIF of your video).



You can use Archive.org to both host the videos and convert the WebM versions to MP4 and OGV.

Industrial-Strength Conversion With Vid.ly

WebM, OGV and MP4 take care of only the major browsers, though. If you want to support all mobile devices, tablets and consoles and you want the video quality to adapt to the user's connection speed, then you'll have to create a few dozen versions of the same video. Encoding.com feels our pain and has released a free service called Vid.ly, which converts any video you upload into many different formats more or less in real time. Unfortunately, the service is in private beta at the moment, but you can use the invite code **`HNY2011`**.



Vid.ly converts any video you upload into many different formats more or less in real time.

Furthermore, Vid.ly creates a URL for your video that automatically redirects the browser or device calling it to the right format. This keeps your embed code as simple as possible:

```
<video src="http://vid.ly/4f3q1f?content=video" controls>
</video>
```

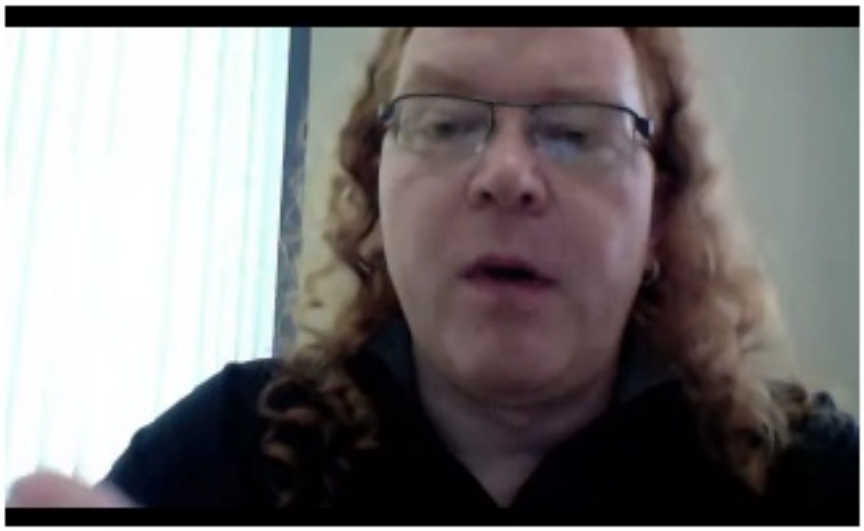
Cool, isn't it?

The Power Of The HTML5 Video API: Syncing Content

Now that our video is on the page, let's check out the power of the API. Say, for example, you want to know what part of the movie is playing right now. This is as simple as subscribing to an event of the **<video>** element:

```
<div id="stage">
  <video src="http://vid.ly/4f3q1f?content=video" controls></
video>
  <div id="time"></div>
</div>
<script>
  (function(){
    var v = document.getElementsByTagName('video')[0]
    var t = document.getElementById('time');
    v.addEventListener('timeupdate',function(event){
      t.innerHTML = v.currentTime;
    },false);
  })();
</script>
```

If you try this out in your browser, you will see the current time below the video when you play it.



19.632

You will also see that the **`timeupdate`** event gets fired a lot and at somewhat random times. If you want to use this to sync the showing and hiding of parts of the document, then you'll need to throttle it somehow. The easiest way to do this is to limit the number to full seconds using **`parseInt()`:**

```
<div id="stage">
  <video src="http://vid.ly/4f3q1f?content=video" controls></
video>
  <div id="time"></div>
</div>
<script>
  (function(){
    var v = document.getElementsByTagName('video')[0]
    var t = document.getElementById('time');
    v.addEventListener('timeupdate',function(event){
      t.innerHTML = parseInt(v.currentTime) + ' - ' +
v.currentTime;
```

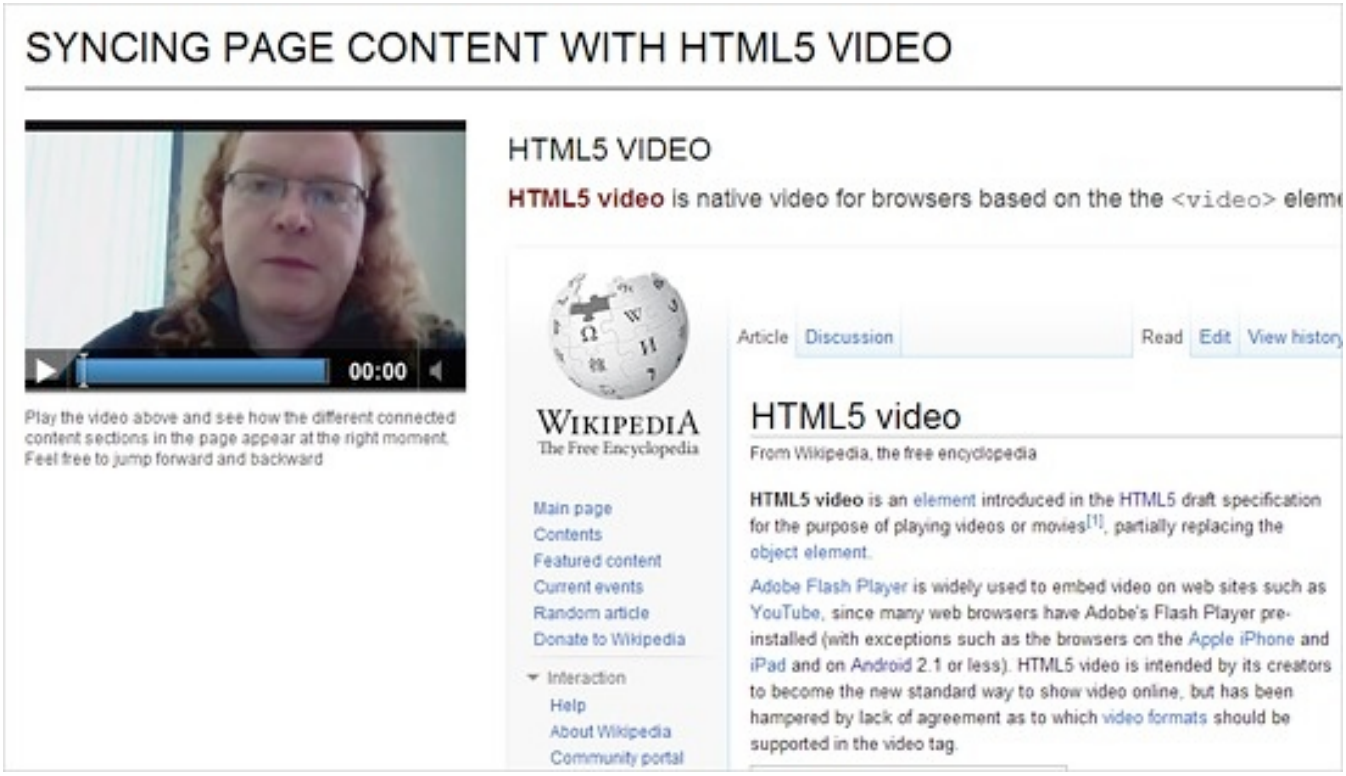
```
    },false);
  })();
</script>
```

3 - 3.97

You can use this to trigger functionality at certain times. For example, you can sync an Indiana Jones-style animation of a map to a video.

For a full explanation of this demo, check out the blog post on Mozilla Hacks.

Let's have a go at something similar: a video that shows the content from web pages being referred to by a presenter. Check out this video demo of me explaining what we're doing here, with the content appearing and disappearing at certain times in the video. Make sure to jump around the video with the controls.



We've already covered how to get the current time of a video in seconds. What I want now is to display and hide a few parts of the website at certain times in the video. If video is not supported in the browser, then I would just show all of the content without any syncing.

The first issue I have to solve is to allow the maintainer to control what is shown when. Normally, I'd use a JSON object in the JavaScript, but I figure that keeping the maintenance in the markup itself makes much more sense.

HTML5 allows you to store information in **`data-` attributes**. So, to make it easy to tell the script when to show what, I just use **`data-start`** and **`data-end`** attributes, which define the time frames for the articles that I want to sync with the video:

```
<article data-start="64" data-end="108">
  <header><h1>Archive.org for conversion</h1></header>
  <p><a href="http://archive.org">Archive.org</a> is a website
dedicated to
archiving the Internet. For content released as under a
Creative Commons
license, it offers hosting for video and audio and
automatically converts the
content to MP4 and Ogg video for you.</p>
  <iframe src="http://archive.org"></iframe>
</article>
```

You can try it out by downloading the code and changing the values yourself (or use Firebug or the Web Inspector to change it on the fly).

Here's the script (using jQuery) that makes this happen:

```
/* if the document is ready… */
$(document).ready(function(){

/* if HTML5 video is supported */
  if($('video').attr('canPlayType')){

    $('aside::first').append('<p>Play the video above and see
how ' +
                             'the different connected content
sections ' +
                             'in the page appear at the right
moment. '+
                             'Feel free to jump forward and
backward</p>');

    var timestamps = [],
        last = 0,
        all = 0,
        now = 0,
        old = 0,
```

```
        i=0;

/* hide all articles via CSS */
    $('html').addClass('js');

/*
   Loop over the articles, read the timestamp start and end
and store
   them in an array
*/
    $('article').each(function(o){
      if($(this).attr('data-start')){
        timestamps.push({
          start : +$(this).attr('data-start'),
          end : +$(this).attr('data-end'),
          elm : $(this)
        });
      }
    });

    all = timestamps.length;

/*
  when the video is playing, round up the time to seconds and
call the
  showsection function continuously
*/
    $('video').bind('timeupdate',function(event){
      now = parseInt(this.currentTime);

      /* throttle function calls to full seconds */
      if(now > old){
        showsection(now);
      }
      old = now;

    });
```

```
/*
  Test whether the current time is within the range of any of
the
  defined timestamps and show the appropriate section.
  Hide all others.
*/
    function showsection(t){
      for(i=0;i<all;i++){
        if(t >= timestamps[i].start && t <= timestamps[i].end)
{
          timestamps[i].elm.addClass('current');
        } else {
          timestamps[i].elm.removeClass('current');
        }
      }
    };

  };
});
```

So, what's going on here? First, we're checking whether the browser is capable of playing HTML5 video by testing for the **`canPlayType`** attribute. If all is fine, then we add some explanatory text to the document (which wouldn't make sense if the browser couldn't show a video). Then, we define some variables to use and add a class of `js` to the root element of the document. This, together with the **`.js article`** selector in the CSS, hides all of the articles in the document.

We then loop through the articles, read out the timestamps for the start and end of each of the sections and store them in an array called **`timestamps`**.

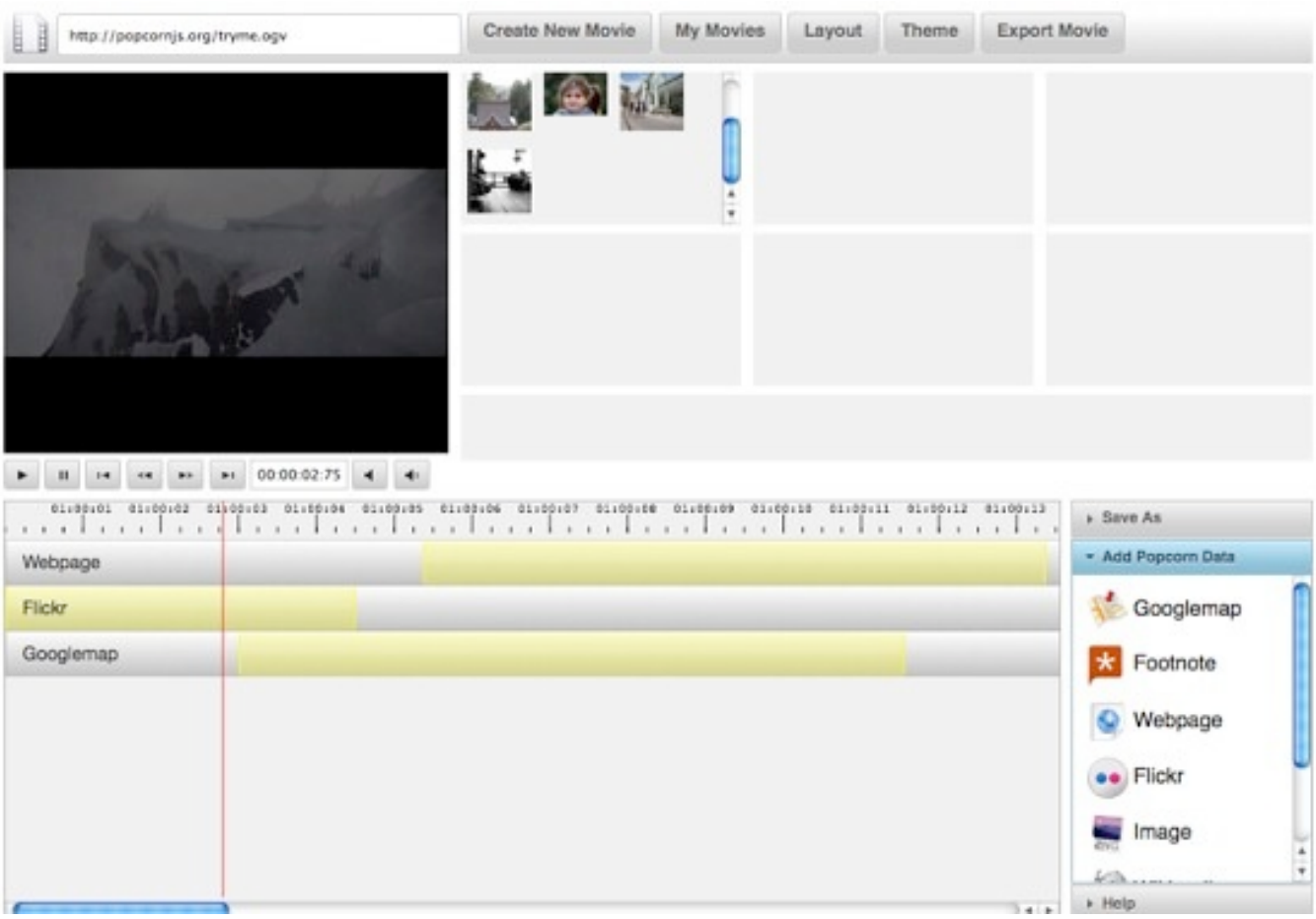
We then subscribe to the **`timeupdate`** event, rounded up to full seconds, and call the **`showsection()`** function every new second.

The **`showsection()`** function loops through all of the timestamps and tests whether the current time of the video is in the range of one of the articles. If it is, then that article is displayed (by adding a **`current`** class) and all the others are hidden. This could be optimized by storing the current section and removing the class from only that element.

Can We Do The Same With Less Or No Code?

If you like the idea of syncing content and video, check out the Popcorn framework, which is based on the same techniques but gives you much more control over the video itself.

Butter is a point-and-click interface to go on top of Popcorn. It has a nice timeline editor that allows you to play a video and show all kinds of Web content at certain times. You can export and send your creations to friends, too.



With systems like Popcorn and Butter, we are one step closer to having authoring tools for the rich interactions that HTML5 offers us. What can you think of building?

Summary

Today we looked at how to embed video onto a Web document; and with the native video API that gives us event handlers for changes in a video, we saw how easy it is to make the video interact with the rest of the document. Instead of trying to control the video, we use native controls to make the page react to what is happening in the video itself. We used semantic HTML and data attributes to allow maintainers to use the syncing script without having to touch any JavaScript, and we looked at some services that make hosting and converting video easy.

All of these cool technologies give us a lot of power, but we can't just, say, write some simple CSS, JavaScript and HTML to use them. If we want open technologies to succeed, then we have to make them easy for people to use. The next step now is to move from the "one-off implementation" phase and think about creating tools and step-by-step code-creation systems for users who want to use these cool new technologies but don't want to spend much time and effort doing it.

With native audio and video in browsers, we've taken a massive step toward make the open Web more engaging and beautiful. The next step will be to use multimedia not only for output but for input. A lot of hardware these days comes with cameras and microphones; we need to start using and supporting open technology that allows our users to take advantage of this hardware to interact with our Web products.

Behind The Scenes Of Nike Better World

By Richard Shepherd

Perhaps one of the most talked about websites in the last 12 months has been Nike Better World. It's been featured in countless Web design galleries, and it still stands as an example of what a great idea and some clever design and development techniques can produce.

In this article, we'll talk to the team behind Nike Better World to find out how the website was made. We'll look at exactly how it was put together, and then use similar techniques to create our own parallax scrolling website. Finally, we'll look at other websites that employ this technique to hopefully inspire you to build on these ideas and create your own variation.

Nike Better World

Nike Better World is a glimpse into how Nike's brand and products are helping to promote sports and its benefits around the world. It is a website that has to be viewed in a browser (preferably a latest-generation browser, although it degrades well) rather than as a static image, because it uses JavaScript extensively to create a parallax scrolling effect.

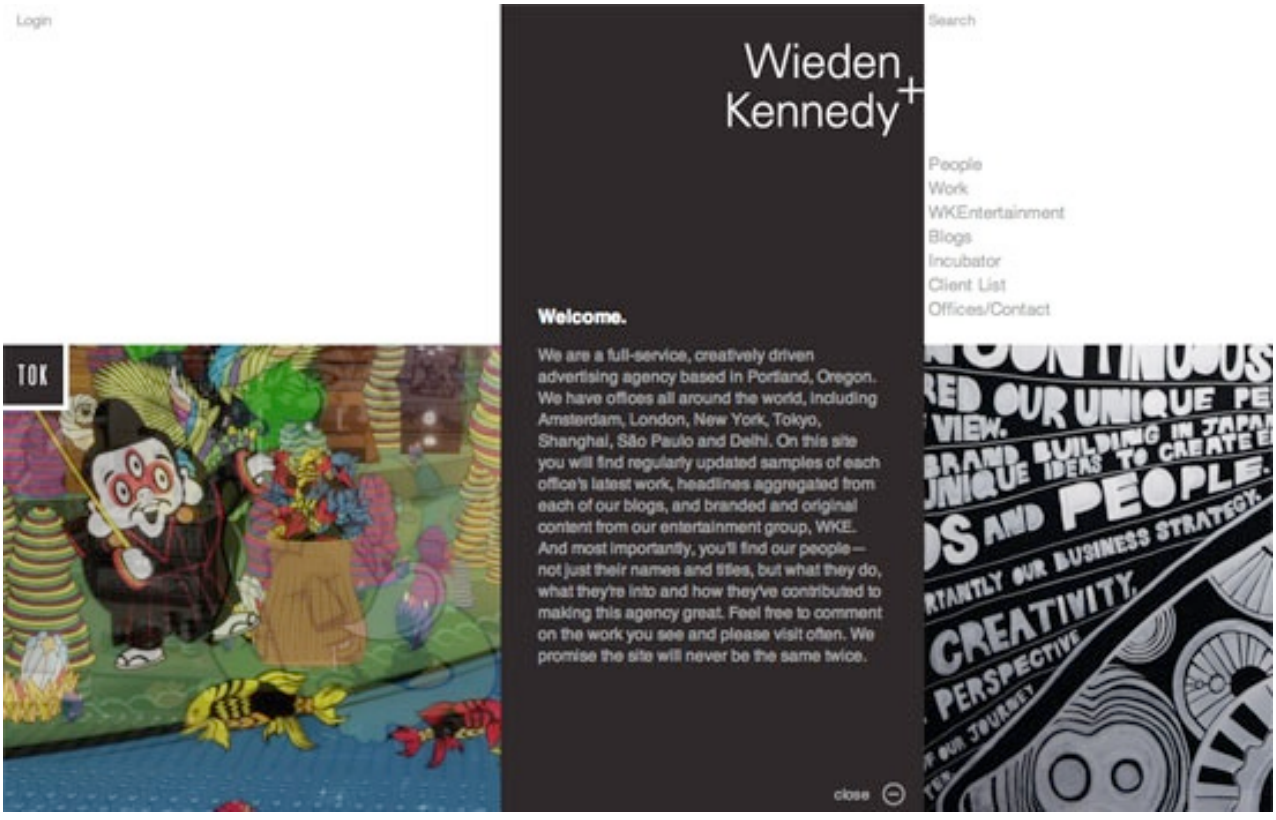
A good deal of HTML5 is used to power this immersive brand experience and, whatever your views on Nike and its products, this website has clearly

been a labor of love for the agency behind it. Although parallax scrolling effects are nothing new, few websites have been able to sew together so many different design elements so seamlessly. There is much to learn here.

AN "INTERACTIVE STORYTELLING EXPERIENCE"

> *In our opinion, technologies are independent of concept. Our primary focus was on creating a great interactive storytelling experience.*
> *– Wieden+Kennedy*

Nike turned to long-time collaborator Wieden+Kennedy (W+K), one of the largest independent advertising agencies in the world, to put together a team of four people who would create Nike Better World: Seth Weisfeld was the interactive creative director, Ryan Bolls produced, while Ian Coyle and Duane King worked on the design and interaction.



I started by asking the team whether the initial concept for the website pointed to the technologies they would use. As the quote above reveals, they in fact always start by focusing on the concept. This is a great point. Too many of us read about a wonderful new technique and then craft an idea around it. W+K walk in the opposite direction: they create the idea first, and sculpt the available technologies around it.

So, with the concept decided on, did they consciously do the first build as an "HTML5 website," or did this decision come later?

> *There were some considerations that led us to HTML5. We knew we wanted to have a mobile- and tablet-friendly version. And we liked the idea of being able to design and build the creative only once to reach all the screens we needed to be on. HTML5 offered a great balance of creativity and technology for us to communicate the Nike Better World brand message in a fresh and compelling way.*
> *— W+K*

HTML5 is still not fully supported in all browsers (read "in IE") without JavaScript polyfills, so just how cross-browser compatible did the website have to be?

> *The primary technical objectives were for the site to be lightweight, optimized for both Web and devices, as well as to be scalable for future ideas and platforms.*
> *— W+K*

To achieve these goals, the website leans on JavaScript for much of the interactivity and scrolling effects. Later, we'll look at how to create our own

parallax scrolling effect with CSS and jQuery. But first, we should start with the template and HTML.

THE STARTING BOILERPLATE

It's worth pointing out the obvious first: Nike Better World is original work and should not be copied. However, we can look at how the website was put together and learn from those techniques. We can also look at other websites that employ parallax scrolling and then create our own page, with our own code and method, and build on these effects.

I asked W+K if it starts with a template.

> *We started without a framework, with only reset styles. In certain cases, particularly with experimental interfaces, it ensures that complete control of implementation lies in your hands.*
> *— W+K*

If you look through some of the code on Nike Better World, you'll see some fairly advanced JavaScript in a class-like structure. However, for our purposes, let's keep things fairly simple and rely on HTML5 Boilerplate as our starting point.

Download HTML5 Boilerplate. The "stripped" version will do. You may want to delete some files if you know you won't be using them (*crossdomain.xml*, the test folder, etc.).

As you'll see from the source code (see the final code below), our page is made up of four sections, all of which follow a similar pattern. Let's look at one individual section:

```
<section class="story" id="first" data-speed="8" data-
type="background">
 <div data-type="sprite" data-offsetY="950" data-
Xposition="25%" data-speed="2"></div>
 <article>
  <h2>Background Only</h2>
  <div>
   <p>Pellentesque habitant morbi tristique senectus et netus
et malesuada fames ac turpis egestas. Vestibulum tortor quam,
feugiat vitae, ultricies eget, tempor sit amet, ante. Donec eu
libero sit amet quam egestas semper. Aenean ultricies mi vitae
est. Mauris placerat eleifend leo.</p>
  </div>
 </article>
</section>
```

I'm not sure this is the best, most semantic use of those HTML5 tags, but it's what we need to make this effect work. Normally, a section has a heading, so, arguably, the section should be a div and the article should be a section. However, as W+K points out, the HTML5 spec is still young and open to interpretation:

> *HTML5 is still an emerging standard, particularly in implementation. A lot of thought was given to semantics. Some decisions follow the HTML5 spec literally, while others deviate. As with any new technology, the first to use it in real-world projects are the ones who really shape it for the future.*
> *— W+K*

This is a refreshing interpretation. Projects like Nike Better World are an opportunity to "reality check" an emerging standard and, for the conscientious among us, to provide feedback on the spec.

> *In short, is the theory of the spec practical? W-K elaborates:*
> *We use the article tag for pieces of content that can (and should) be individually (or as a group) syndicated. Each "story" is an article. We chose divs to wrap main content. We took the most liberty with the section tag, as we feel its best definition in the spec is as chapters of content, be it globally.*
> *— W+K*

As an aside (no pun intended!), HTML5 Doctor has begun a series of mark-up debates called Simplequizes, which are always interesting and illustrate that there is rarely one mark-up solution for any problem. Make sure to check them out.

In *style.css*, we can add a background to our section with the following code:

```
section { background: url(../images/slide1a.jpg) 50% 0 no-repeat fixed; }
```

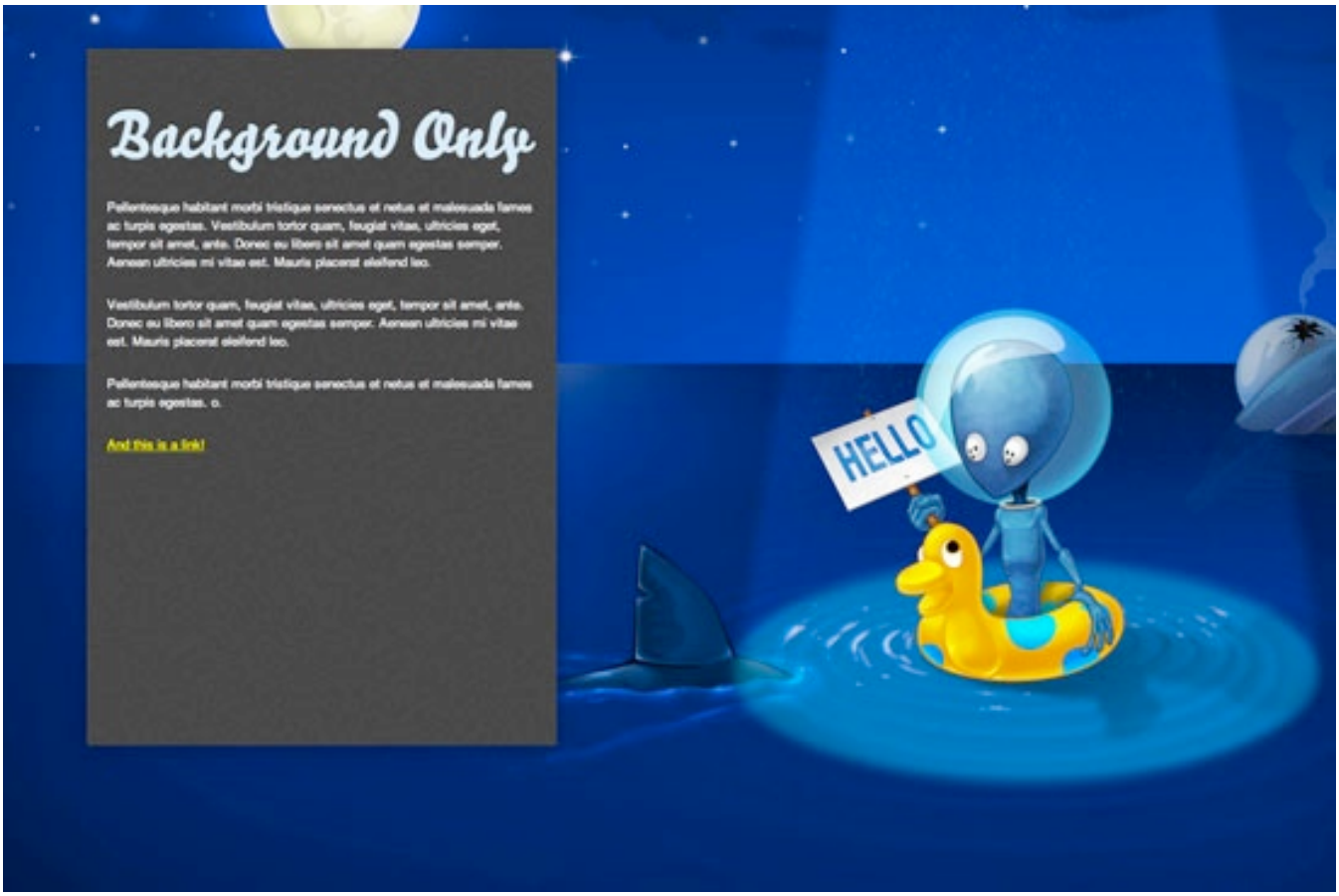
We should also give our sections a height and width, so that the background images are visible:

```
.story { height: 1000px; padding: 0; margin: 0; width: 100%; max-width: 1920px; position: relative; margin: 0 auto; }
```

I've set the height of all our sections to 1000 pixels, but you can change this to suit your content. You can also change it on a per-section basis.

We have also made sure that the maximum width of the section is the maximum width of the background (1920 pixels), and we have specified a relative position so that we can absolutely position its children.

Here's the page before adding JavaScript. It's worth digging into the source code to see how we've duplicated the sections in the HTML and CSS.



Even with this code alone, we already have a pleasing effect as we scroll down the page. We're on our way.

THE HTML5 DATA ATTRIBUTE

Before looking at parallax scrolling, we need to understand the new **`data`** attribute, which is used extensively throughout the HTML above.

Back in the good old days, we would shove any data that we wanted to be associated with an element into the **`rel`** attribute. If, for example, we needed to make the language of a story's content accessible to JavaScript, you might have seen mark-up like this:

```
<article class='story' id="introduction" rel="en-us">
</article>
```

Sometimes complex DOM manipulation requires more information than a **`rel`** can contain, and in the past I've stuffed information into the **`class`** attribute so that I could access it. Not any more!

The team at W+K had the same issue, and it used the **`data`** attribute throughout Nike Better World:

> *The **`data`** attribute is one of the most important attributes of HTML5. It allowed us to separate mark-up, CSS and JavaScript into a much cleaner workflow. Websites such as this, with high levels of interaction, are almost application-like behind the scenes, and the **`data`** attribute allows for a cleaner application framework.*
> *— W+K*

So, what is the data attribute? You can read about it in the official spec, which defines it as follows:

> *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.*
> *— W+K*

In other words, any attribute prefixed with data- will be treated as storage for private data; it does not affect the mark-up, and the user cannot see it. Our previous example can now be rewritten as:

```
<article class='story' id="introduction" data-language="en-
us"></article>
```

The other good news is that you can use more than one **`data`** attribute per element, which is exactly what we're doing in our parallax example. You may have spotted the following:

```
<div data-type="sprite" data-offsetY="100" data-
Xposition="50%" data-speed="2"></div>
```

Here, we are storing four pieces of information: the x and y data offsets and the data speed, and we are also marking this element as a data type. By testing for the existence of **`data-type`** in the JavaScript, we can now manipulate these elements as we wish.

Parallax Scrolling

On our page, three things create the parallax scrolling illusion:

- The background scrolls at the slowest rate,
- Any sprites scroll slightly faster than the background,
- Any section content scrolls at the same speed as the window.

With three objects all scrolling at different speeds, we have created a beautiful parallax effect.

It goes without saying that we don't need to worry about the third because the browser will take care of that for us! So, let's start with the background scroll and some initial jQuery.

```
$(document).ready(function(){
 // Cache the Window object
 $window = $(window);
// Cache the Y offset and the speed
$('[data-type]').each(function() {
  $(this).data('offsetY', parseInt($(this).attr('data-
offsetY')));
  $(this).data('speed', $(this).attr('data-speed'));
});
// For each element that has a data-type attribute
 $('section[data-type="background"]').each(function(){
  // Store some variables based on where we are
  $(this).data('speed', parseInt($(this).attr('data-speed')));
   var $self = $(this),
   offsetCoords = $self.offset(),
   topOffset = offsetCoords.top;
   $(window).scroll(function(){
    // The magic will happen in here!
   }); // window scroll
 });  // each data-type
}); // document ready
```

First, we have our trusty jQuery **`document ready`** function, to ensure that the DOM is ready for processing. Next, we cache the browser window object, which we will refer to quite often, and call it **`$window`**. (I like to prefix jQuery objects with **`$`** so that I can easily see what is an object and what is a variable.)

We also use the jQuery **`.data`** method to attach the Y offset (explained later) and the scrolling speed of the background to each element. Again, this is a form of caching that will speed things up and make the code more readable.

We then iterate through each section that has a `data` attribute of **`data-type="background"`** with the following:

```
$('section[data-type="background"]').each(function(){}
```

Already we can see how useful data attributes are for storing multiple pieces of data about an object that we wish to use in JavaScript.

Inside the **`.each`** function, we can start to build up a picture of what needs to be done. For each element, we need to grab some variables:

```
// Store some variables based on where we are
var $self = $(this),
    offsetCoords = $self.offset(),
    topOffset = offsetCoords.top;
```

We cache the element as **`$self`** (again, using the $ notation because it's a jQuery object). Next, we store the **`offset()`** of the element in **`offsetCoords`** and then grab the top offset using the **`.top`** property of **`offset()`**.

Finally, we set up the window **`scroll`** event, which fires whenever the user moves the scroll bar or hits an arrow key (or moves the trackpad or swipes their finger, etc.).

We need to do two more things: check that the element is in view and, if it is, scroll it. We test whether it's in view using the following code:

```
// If this section is in view
if ( ($.Window.scrollTop() + $.Window.height()) >
($offsetCoords.top) &&
( ($offsetCoords.top + $self.height()) >
$.Window.scrollTop() ) ) {
}
```

The first condition checks whether the very top of the element has scrolled into view at the very bottom of the browser window.

The second condition checks whether the very bottom of the element has scrolled past the very top of the browser window.

You could use this method to check whether *any* element is in view. It's sometimes useful (and quicker) to process elements only when the user can see them, rather than when they're off screen.

So, we now know that some part of the section element with a **`data-type`** attribute is in view. We can now scroll the background. The trick here is to scroll the background slower or faster than the browser window is scrolling. This is what creates the parallax effect.

Here's the code:

```
// Scroll the background at var speed
// the yPos is a negative value because we're scrolling it UP!
var yPos = -($window.scrollTop() / $self.data('speed'));

// If this element has a Y offset then add it on
if ($self.data('offsetY')) {
  yPos += $self.data('offsetY');
}

// Put together our final background position
var coords = '50% '+ yPos + 'px';

// Move the background
$self.css({ backgroundPosition: coords });
```

The **`y`** position is calculated by dividing the distance that the user has scrolled from the top of the window by the speed. The higher the speed, the slower the scroll.

Next, we check whether there is a **`y`** offset to apply to the background. Because the amount that the background scrolls is a function of how far the window has scrolled, the further down the page we are, the more the background has moved. This can lead to a situation in which the background starts to disappear up the page, leaving a white (or whatever color your background is) gap at the bottom of each section.

The way to combat this is to give those backgrounds an offset that pushes them down the page an extra few hundred pixels. The only way to find out this magic offset number is by experimenting in the browser. I wish it was more scientific than this, but this offset really does depend on the height of the browser window, the distance scrolled, the height of your sections and the height of your background images. You could perhaps write some JavaScript to calculate this, but to me this seems like overkill. Two minutes experimenting in Firebug yields the same result.

The next line defines a variable **`coords`** to store the coordinates of the background. The **`x`** position is always the same: 50%. This was the value we set in the CSS, and we won't change it because we don't want the element to scroll sideways. Of course, you're welcome to change it if you want the background to scroll sideways as the user scrolls up, perhaps to reveal something.

(Making the speed a negative number for slower scrolling might make more sense, but then you'd have to divide by **`-$speed`**. Two negatives seems a little too abstract for this simple demonstration.)

Finally, we use the **`.css`** method to apply this new background position. Et voila: parallax scrolling!

Here's the code in full:

```
// Cache the Window object
$window = $(window);

// Cache the Y offset and the speed of each sprite
$('[data-type]').each(function() {
  $(this).data('offsetY', parseInt($(this).attr('data-
offsetY')));
  $(this).data('speed', $(this).attr('data-speed'));
});

// For each element that has a data-type attribute
$('section[data-type="background"]').each(function(){

// Store some variables based on where we are
var $self = $(this),
    offsetCoords = $self.offset(),
    topOffset = offsetCoords.top;

$(window).scroll(function(){

// If this section is in view
if ( ($window.scrollTop() + $window.height()) > (topOffset) &&
( (topOffset + $self.height()) > $window.scrollTop() ) ) {

  // Scroll the background at var speed
  // the yPos is a negative value because we're scrolling it
UP!
  var yPos = -($window.scrollTop() / $self.data('speed'));

  // If this element has a Y offset then add it on
  if ($self.data('offsetY')) {
    yPos += $self.data('offsetY');
  }

  // Put together our final background position
```

```
    var coords = '50% '+ yPos + 'px';

    // Move the background
    $self.css({ backgroundPosition: coords });

    }; // in view

  }); // window scroll

  });  // each data-type
```

Of course, what we've done so far is quite a bit simpler than what's on Nike Better World. W+K admits that the parallax scrolling threw it some challenges:

> *The parallax scrolling presented a few small challenges in cross-browser compatibility. It took a little experimenting to ensure the best scrolling experience. In the end, it was less about the actual parallax effect and more about synchronized masking of layers during transitions.*
> *— W+K*

W+K also reveals how it maintained a fast loading and paint speed by choosing its tools wisely:

> *The key to maintaining faster speeds is to use native CSS where possible, because DOM manipulation slows down rendering, particularly on older browsers and processors.*
> *— W+K*

For example, the “More” button below spins on hover, an effect achieved with CSS3. In browsers that don’t support CSS3 transforms, the purpose of the graphic is still obvious.

ADDING MORE ELEMENTS

Of course, one of the other common features of parallax scrolling is that *multiple* items on the page scroll. So far, we have two elements that move independently of each other: the first is the page itself, which scrolls when the user moves the scroll bar, and the second is the background, which now scrolls at at slower rate thanks to the jQuery above and the **`background-position`** CSS attribute.

For many pages, this would be enough. It would be a lovely effect for the background of, say, a blog. However, Nike and others push it further by

adding elements that move at a different speed than that of the page and background. To make things easy — well, easier — I'm going to call these new elements sprites.

Here's the HTML:

```
<div id="smashinglogo" data-type="sprite" data-offsetY="1200"
data-Xposition="25%" data-speed="2"></div>
```

Put this just before the closing **</article>** tag, so that it appears *behind* the contents of **<article>**. First, we give the div an id so that we can refer to it specifically in the CSS. Then we use our HTML5 **data** attribute to store a few values:

- The status of a **sprite**,
- A **y** (vertical) offset of 1200 pixels,
- An **x** (horizontal) position as a percentage,
- A scrolling speed.

We give the **x** position of the sprite a percentage value because it is relative to the size of the viewport. If we gave it an absolute value, which you're welcome to try, there's a chance it could slide out of view on either the left or right side.

Now about that **y** offset…

INCEPTION

This is the bit that's going to mess with your noodle and is perhaps the hardest part of the process to grasp.

Thanks to the logic in the JavaScript, the sprite won't move until the parent section is in view. When it does move, it will move at (in this case) half the speed. You need the vertical position, then, to account for this slower movement; elements need to be placed higher up if they will be scrolling more slowly and, therefore, moving less on the `y` axis.

We don't know how far the user has to scroll before the section appears at the bottom of the page. We could use JavaScript to read the viewport size and then do some calculations based on how far down the page the section is positioned. But that is already sounding too complicated. There is an easier way.

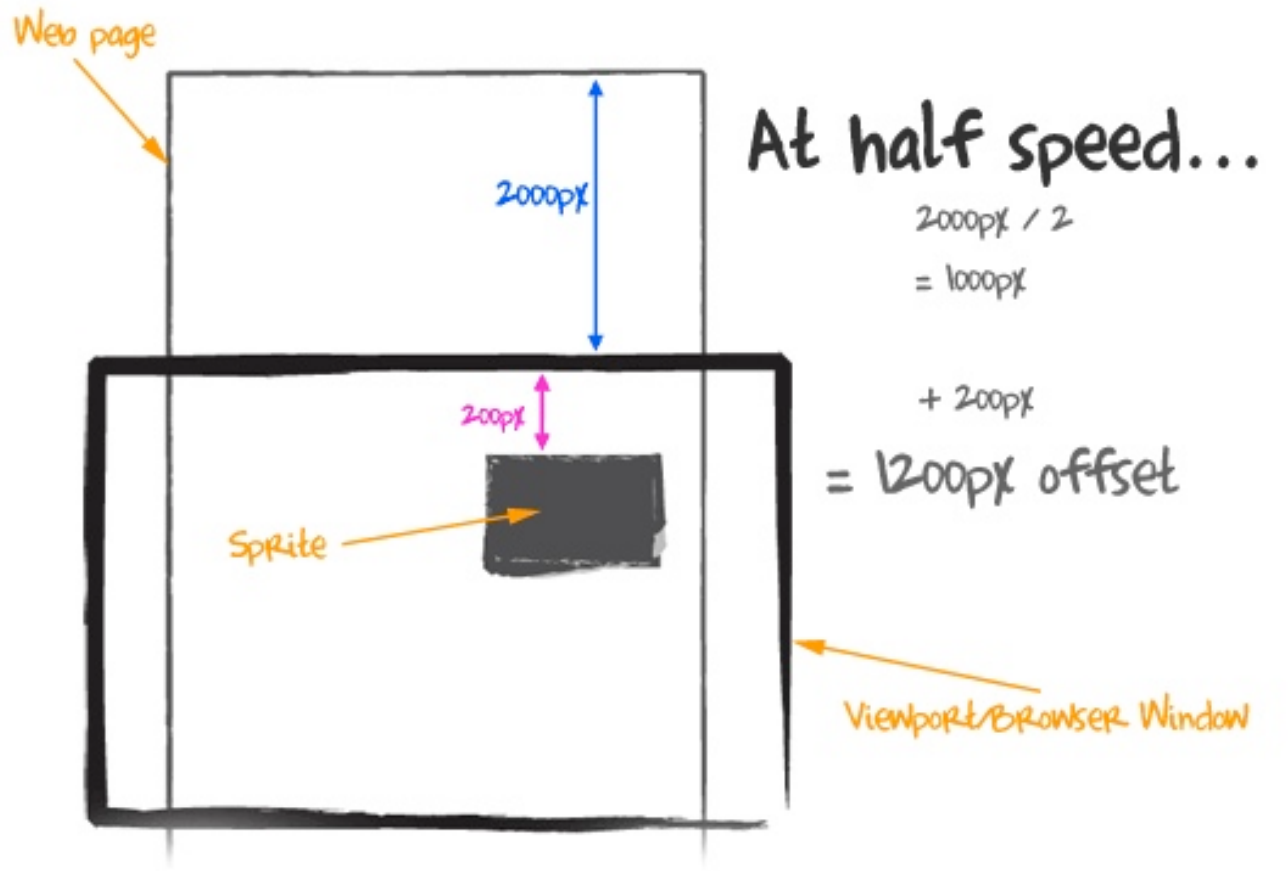
What we *do* know is how far the user has scrolled before the current section is flush with the *top* of the viewport: they have scrolled the `y` offset of that particular section. (Put another way, they have scrolled the height of all of the elements *above* the current one.)

So, if there are four sections, each 1000 pixels high, and the third section is at the top of the viewport, then the user *must* have scrolled 2000 pixels, because this is the total height of the preceding sections.

If we want our sprite to appear 200 pixels from the top of its parent section, you'd figure that the total vertical offset we give it should be 2200 pixels. But it's not, because this sprite has speed, and this speed (in our example) is a function of how far the page has been scrolled.

Let's assume that the speed is set as 2, which is half the speed at which the page is scrolling. When the section is fully in view, then the window has scrolled 2000 pixels. But we divide this by the speed (2) to get 1000 pixels. If we want the sprite to appear 200 pixels from the top, then we need to *add* 200 to 1000, giving us 200 pixels. Therefore, the offset is 1200. In the JavaScript, this number is inverted to -1200 because we are pushing the **`background-position`** down off the bottom of the page.

Here's a sketch to show this in action.



This is one of those concepts that is easier to understand when you view the page and source code and scroll around with the console open in Firebug or Developer Tools.

The JavaScript looks like this:

```
// Check for other sprites in this section
$('[data-type="sprite"]', $self).each(function() {

  // Cache the sprite
  $sprite = $(this);

  // Use the same calculation to work out how far to scroll
the sprite
```

```
    var yPos = -($.Window.scrollTop() / $sprite.data('speed'));
    var coords = $sprite.data('Xposition') + ' ' + (yPos +
$sprite.data('offsetY')) + 'px';
    $sprite.css({ backgroundPosition: coords });
}); // sprites
```

HTML5 VIDEO

One criticism levelled at Nike Better World is that it didn't use HTML5 video. HTML5 is still not fully supported across browsers (I'm looking at you, Internet Explorer), but for the purposes of this article, we'll embrace HTML5 video, thanks to the lovely folks at Vimeo and Yum Yum London.

But we can't set a video as a background element, so we have a new challenge: how to position and scroll this new sprite?

Well, there are three ways:

1. We could change its **`margin-top`** property within its parent section;
2. We could make it a **`position: absolute`** element and change its top property when its parent section comes into view;
3. We could define it as **`position: fixed`** and set its **`top`** property relative to the viewport.

Because we already have code for the third, let's grab the low-hanging fruit and adapt it to our purposes.

Here's the HTML we'll use, which I've placed *after* the closing **`</article>`** tag:

```
<video controls width="480" width="320" data-type="video"
data-offsetY="2500" data-speed="1.5">
  <source src="video/parallelparking.theora.ogv" type="video/
ogg" />
  <source src="video/parallelparking.mp4" type="video/mp4" />
  <source src="video/parallelparking.webm" type="video/webm" /
>
</video>
```

First, we've opened our HTML5 video element and defined its width and height. We then set a new **`data-type`** state, **`video`**, and defined our **`y`** offset and the speed at which the element scrolls. It's worth nothing that some experimentation is needed here to make sure the video is positioned correctly. Because it's a **`position: fixed`** element, it will scroll on *top* of all other elements on the page. You can't cater to every viewport at every screen resolution, but you can play around to get the best compromise for all browser sizes (See "Bespoke to Broke" below).

The CSS for the video element looks like this:

```
video { position: fixed; left: 50%; z-index: 1;}
```

I've positioned the video `50%` from the left so that it moves when the browser's size is changed. I've also given it a **`z-index: 1`**. This z-index prevents the video element from causing rendering problems with neighboring sections.

And now for the JavaScript! This code should be familiar to you:

```
// Check for any Videos that need scrolling
$('[data-type="video"]', $self).each(function() {

  // Cache the sprite
  $video = $(this);

  // Use the same calculation to work out how far to scroll
the sprite
  var yPos = -($window.scrollTop() / $video.data('speed'));
  var coords = (yPos + $video.data('offsetY')) + 'px';
  $video.css({ top: coords });
}); // video
```

And there you have it! A parallax scrolling HTML5 video.

BESPOKE OR BROKE

Of course, every design is different, which means that *your* code for *your* design will be unique. The JavaScript above will plug and play, but you will need to experiment with values for the `y` offset to get the effect you want. Different viewport sizes means that users will scroll different amounts to get to each section, and this in turn affects how far your elements scroll. I can't control it any more than you can, so you have to pick a happy medium. Even Nike Better World suffers when the viewport's vertical axis stretches beyond the height of the background images.

I asked W+K how it decides which effects to power with JavaScript and which are better suited to modern CSS techniques:

> *Key points that required complex interaction relied on JavaScript, while visual-only interactivity relied on CSS3. Additionally, fewer browsers support native CSS3 techniques, so items that were more important to cross-browser compatibility were controlled via JavaScript as well.*
> *— W+K*

This is a wonderful example of "real-world design." So often we are bamboozled with amazing new CSS effects, and we make websites that sneer at older browsers. The truth is, for most commercial websites and indeed for websites like Nike Better World that target the biggest audience possible, stepping back and considering how best to serve your visitors is important.

W+K explains further:

> *We started by creating the best possible version, but always kept the needs of all browsers in mind. Interactive storytelling must balance design and technology to be successful. A great website usable in one or two browsers ultimately fails if you want to engage a wide audience.*
> *— W+K*

And Internet Explorer?!

> *IE was launched in tandem with the primary site. Only IE6 experienced challenges, and as a deprecated browser, it gracefully degrades.*
> *— W+K*

The Final Code

The code snippets in this piece hopefully go some way to explaining the techniques required for a parallax scrolling effect. You can extend them further to scroll multiple elements in a section at different speeds, or even scroll elements sideways!

Feel free to grab the full source code from GitHub, and adapt it as you see fit. Don't forget to let us know what you've done, so that others can learn from your work.

Of course, remember that manipulating huge images and multiple sprites with JavaScript can have huge performance drawbacks. As Keith Clark recently tweeted:

Test, test and test again. Optimize your images, and be aware that you may have to compromise to support all browsers and operating systems.

Tell A Story

Above and beyond the technical wizardry of parallax websites — some of the best of which are listed below — the common thread that each seems to embody is *story*. That's what makes them great.

I asked W+K what it learned from the project:

> *That a strong voice, simplicity and beauty are integral to a great interactive storytelling experience. We often hear things like "content is king, and technology is just a tool to deliver it," but when you're able to successfully combine a powerful message with a compelling execution, it creates an experience that people really respond to and want to spend time with.*
> *— W+K*

We really have just scratched the surface of the work that goes into a website like Nike Better World. The devil is in the details, and it doesn't take long to see how much detail goes into both the design and development.

However, if you have a compelling story to tell and you're not afraid of a little JavaScript and some mind-bending offset calculations, then a parallax website might just be the way to communicate your message to the world.

THANKS

Putting together this article took the cooperation of a number of people. I'd like to thank Seth, Ryan, Ian and Duane for answering my questions; Katie Abrahamson at W+K for her patience and for helping coordinate the interview; and Nike for allowing us to dissect its website so that others could learn.

About The Authors

Louis Lazaris

Louis Lazaris is a freelance web developer based in Toronto, Canada. He blogs about front-end code on Impressive Webs and is a coauthor of HTML5 and CSS3 for the Real World, published by SitePoint. You can follow Louis on Twitter or contact him through his website.

Bruce Lawson

Bruce Lawson evangelizes open web technologies for Opera. He co-authored Introducing HTML5, the best-selling book on HTML5 that has just been published in its second edition. He blogs at brucelawson.co.uk.

Christian Heilmann

An international Developer Evangelist working for Mozilla in the lovely town of London, England.

Sergey Chikuyonok

Sergey Chikuyonok is a Russian front-end web-developer and writer with a big passion on optimization: from images and JavaScript effects to working process and time-savings on coding.

Richard Shepherd

Richard (@richardshepherd) is a UK based web designer and front-end developer. He loves to play with HTML5, CSS3, jQuery and WordPress, and currently works full-time bringing VoucherCodes.co.uk to life. He has an awesomeness factor of 8, and you can also find him at richardshepherd.com.