Jumping Into HTML5

Written by Ben Schwarz

Reviewed by Russ Weakley

If you asked me to explain HTML5 to you, I would probably start by explaining that your role as a Web developer has changed from the days of old. I would expect you to be an expert in HTML (the markup language), CSS (and all of its permutations across browsers), JavaScript (and the subtle differences between its APIs in browsers). And then I would roll on to design theory, animation, 3-D, server-side technologies and sound engineering.

After a moment of silence, you would probably wonder why so many technologies are brought under the umbrella of HTML and perhaps wonder why you decided to build for the Web in the first place.

HTML5 (as a specification) is broken into many pieces, covering distinct areas of specialization, so try not to fret. Getting a solid, basic understanding of HTML, CSS and JavaScript will enable you to continue on your own and develop specialized skills that others do not possess. In essence, you will become invaluable to your team or company by focusing on "non-core" technology.

Best practices have not been established for many of these fancy new features, so if you want to learn something cool (and maybe become famous in the process), then it is time to download a beta browser and start experimenting.

Most browser vendors release beta versions of their browsers to allow developers to experiment with cutting-edge features. The "big five" all have betas that you can download:

- Google Chrome has three not-ready-for-prime-time versions: "Beta" (for developers), "Dev channel" (for developers who want to use features that have been released within a week) and "Canary" (a nightly release, totally untested). You can get all of them from smashed.by/chromedev.
- Apple's Safari browser has one version: WebKit (webkit.org).
- Opera has a "Next" version: smashed.by/operadev.
- Firefox has a nightly edition (smashed.by/ffndev) and a pre-beta build called Aurora (smashed.by/ffadev).
- Last but certainly not least, Microsoft releases new builds of IE manually (i.e. not nightly): smashed.by/iedev.

Browser support for new features is rolled out in a modular fashion. And with browser vendors (notably, Google and Mozilla) now releasing on a six- to eight-week cycle, version numbers are clearly less important than they used to be. One could liken this to the way developers make changes to websites. A website has a version, but it is unimportant to the end user. So, as a Web developer, concern yourself with which features to use to best tell your story and to translate your designs into living, breathing products.

As Web technologies evolve, we have to be constantly aware of the past. Thankfully, this goal is shared by both of HTML's standards bodies, so you will be relieved to know that HTML5 will not alienate your user base or make your job harder. Whichever DOCTYPE you use, the user's browser will render the website as best it can. If you use a new HTML5 feature with an old DOCTYPE, it will still render correctly.

In this chapter, we will not talk about WebGL, audio and video, device APIs, Web sockets or SVG. I will leave them for you to discover because each warrants its own chapter. Instead, I will give you a tour of the ground floor. We will cover everything that is important to get right before moving on to advanced topics.

Where We've Come From, Where We're Going

HTML5 is a lot of things. And since the last major "version" of HTML, we have come a long way. The Web Hypertext Application Technology Working Group (WHATWG) refers to it as "HTML: The Living Standard" (it drops the 5). That is, HTML is defined as version-less technology. And as mentioned, browser vendors cherry-pick which features to implement, which is why browsers vary in their support.

WHATWG, W3C AND "THE COMPANIES"

You have probably heard of the World Wide Web Consortium (W3C). More recently (perhaps as recently as the last few paragraphs), you have seen reference to WHATWG. WHATWG was formed by representatives of Apple, Mozilla and Opera, who were concerned about the lack of development of HTML by the W3C and thus decided to form their own group.

Much of the work of WHATWG is shared by the W3C, and the license for the specification states that, "You are granted a license to use, reproduce and create derivative works of this document."

The W3C indeed shares the work. It does not create standards, but rather makes recommendations. And while the W3C is funded by all of the big computing and browser companies, it is dedicated to open standards that do not put any one company at an advantage.

So, as a Web developer, you can be sure that all new developments in HTML (particularly with regard to Web applications) are developed with considerable financial backing from browser implementors (Webkit, Gecko and Opera) and are approved by the W3C over time.

This odd relationship has led to a situation in which technology that comes with licensing fees or that is strictly proprietary is not looked upon favorably by many. The browser race is as competitive as when it began.

KNOWING WHAT FEATURES TO USE

A modern Web developer has to understand the audience they are serving, be able to choose the right technology for the job, and know what the impact will be if a feature is not supported by the browsers used by their audience.

Figure 3.1. Caniuse (smashed.by/ciu) illustrates when you can, or should use a HTML5 feature.

Only wizards magically know whether a given feature is widely supported. If you're not one of them, you can be thankful for When Can I Use.[1] The site lists what features are supported in current versions of all the major desktop and mobile browsers and what features will be in future versions. It is searchable, and it even hooks up to Google Analytics to show you what browsers your audience is using. Now, let's dive in and look at HTML, from the ground floor.

THE DOCTYPE

Think back as far as you can remember. Did you ever remember the full DOCTYPE for HTML 4.01 (or for XHTML, for that matter)? I didn't think so. Let me show you the HTML5 DOCTYPE:

```
<!doctype html>
```

That's it! It can be in uppercase or lowercase letters, and it is all you need to put the browser in standards-compliance mode. It makes you wonder why we had to copy and paste the top of our HTML documents all the time.

Of course, we have been littering our HTML with a bunch of other important tags for years now. Let's look at what else has been simplified.

META CHARACTER SET

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

Argh, what a mess! This meta tag is rather important and should be added before the title tag to ensure that the browser sets the character encoding correctly. Thankfully, it has been simplified to something memorable:

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

Some XML parsers have trouble with tags that are not self-closing, which is why some Web developers choose to use self-closing tags (i.e. XHTML style). It is entirely up to you, but we suggest leaving the tags open.

[1] smashed.by/wcai

STYLE SHEET LINK AND SCRIPT TAGS

The type attribute can be omitted from both the <link rel="stylesheet" href="layout.css"> and script tags.

In the old days, the type attribute could be used in the script tag if you wanted to use VBScript instead of JavaScript, but these days it is not at all required.

Being able to omit these details that made our documents longer and harder to write feels great. But we have just scratched the surface. Let's add a little something to the script tag.

ASYNCHRONOUS SCRIPT DOWNLOADS

First, a word on how the browser downloads files. After the browser downloads and parses HTML, it collects a list of assets (i.e. images, CSS, JavaScripts, etc.) and prioritizes them for downloading in order of appearance.

In the past, we connected to the Internet through dial-up, which did not handle multiple concurrent connections very well. Now, because bandwidths vary drastically (especially with mobile devices in the picture), browsers are limited to downloading only a few assets at a time per top-level domain.

This is why some developers use content delivery networks or assign assets to a subdomain (such as assets.example.com); using different top-level domains gives the developer more download "slots" for scripts, style sheets, images and iframes. Be aware that this comes with a performance hit!

When browsers download JavaScript, they do so one script at a time, allowing the browser to parse and pre-run magical optimizations. Now, rather than leaving the whole experience to chance, we are able to use script loaders (such as LABjs, Yepnope, RequireJS and many others) to load multiple scripts at once, set up dependencies and determine whether a particular script file is required at all.

Improving the performance of pages where possible makes sense. Amazon claims that an increase of 100 milliseconds in page-loading speed leads to a decrease in sales of 1%.[2] With that in mind, let's look at my favorite of script loaders, Yepnope.[3]

Yepnope can be used to conditionally load scripts based on tests. Simply put, you can request a JavaScript only if the browser needs it.

[2] smashed.by/amzspeed
[3] smashed.by/yepnope

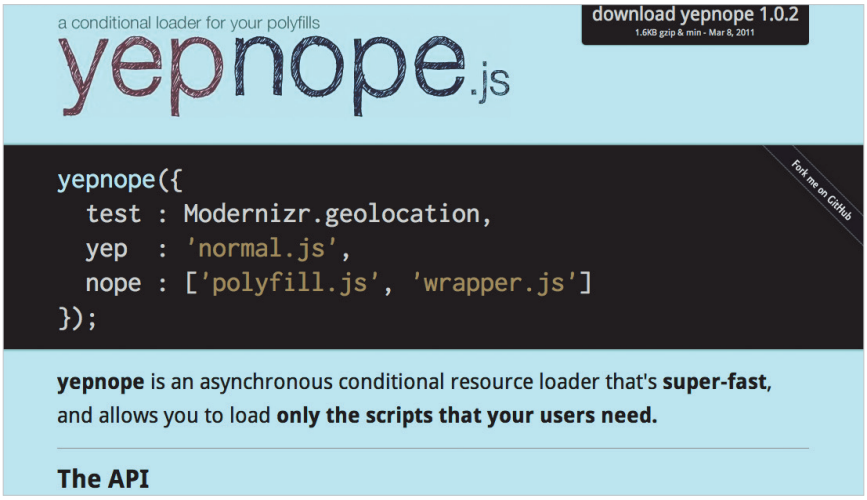

Figure 3.2. Yepnope: a conditional script loader.

For example:

CODE

```
yepnope([
  {
    test: window.JSON,
    nope: '/javascripts/json2.js'
  }
])
```

This clever bit of JavaScript checks whether the browser has a native JSON parser, and for those that do not (which is IE 6 and 7), it loads /javascripts/json2.js, which is a JSON polyfill.

Now that we have (briefly) covered the basics of script loaders and talked about loading scripts in parallel, it is time to look at two new attributes in the script tag. First up, async:

```
<script src="/javascripts/application.js" async></script>
```

The async tag is a boolean attribute, which means that its mere presence in the browser indicates true, or "Yes, please use this feature." It tells the browser to execute application.js as soon as it is available. Scripts that are loaded using async are executed as soon as they are downloaded—that is, not in the order of their appearance in the HTML.

GETTING THE FILES TO THE CLIENT FASTER

It is worth mentioning that the biggest performance gain we get is in reducing the size of the scripts as a whole. The first way to reduce the size of scripts (as well as of style sheets and HTML files) is to serve them using gzip. To add gzip support to your website, check out the HTML5 Boilerplate Webserver configuration repository on GitHub.[4]

If you are not sure how your website is being served, it is time to familiarize yourself with the Web developer toolbar. In Webkit-based browsers (i.e. Safari and Chrome), you can open the Web developer toolbar by pressing Command + Option + I on a Mac and Control + Shift + I in Windows.

Under the "Network" tab, you will see a list of files that were loaded for the current page. You can inspect the request and the response headers for each file listed.

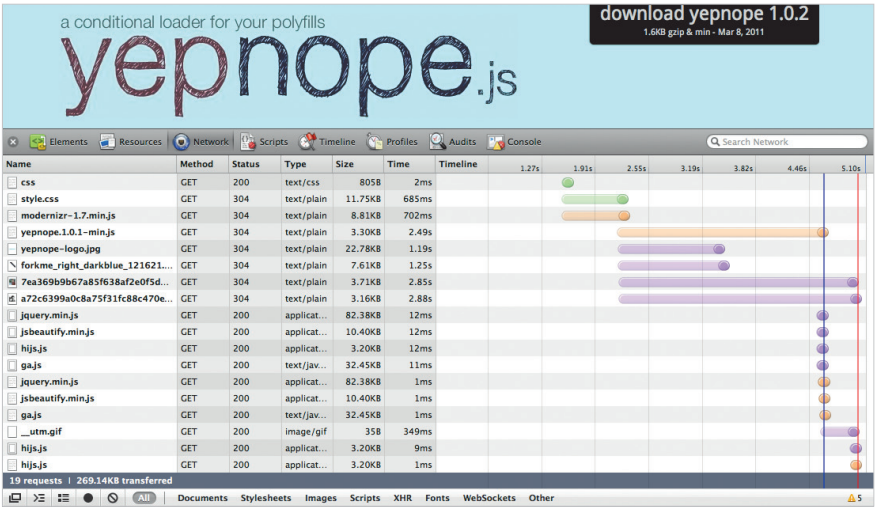

Figure 3.3. Safari's developer toolbar that shows the network activity on Yepnope.

The second tip for improving your website's performance (actually, the best way to improve performance when it comes to scripts) is to concatenate and compress your files. I recommend the UglifyJS compiler.[5]

Clever people such as Steve Souders have devoted themselves to understanding how all browsers download, parse and display websites. If you are interested in producing better-performing websites and applications, follow Steve's work.

[4] smashed.by/configs
[5] smashed.by/uglify

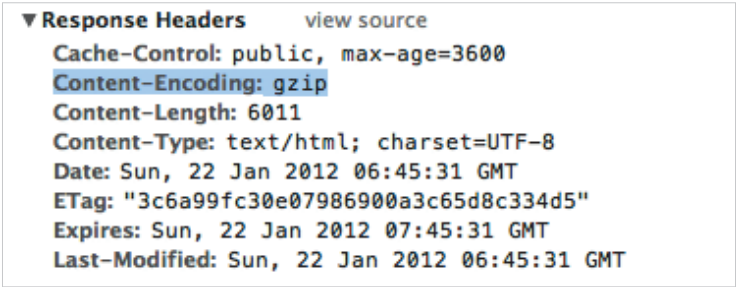

Figure 3.4. This file was returned to the browser using gzip encoding.

NEW SEMANTIC TAGS AND WHEN TO USE THEM

Because we are talking about HTML5, all of this talk of performance and scripting probably feels a bit foreign. Let's look at the new semantic tags to work out when to use them.

Before using any new tags, be sure to use what is known as the HTML5 Shiv. The script is essential because without it, Internet Explorer 6, 7 and 8 will ignore any unknown styles (i.e. the new HTML5 tags, which were unknown when those versions were built). You can get a copy of HTML5 Shiv from Google Code, where the project is hosted.[6]

You can also get HTML5 Shiv by using Modernizr.[7] We won't cover Modernizr in this chapter, but do check it out. I have been using it on every website that I have built in the least two years.

RESETTING DEFAULT HTML STYLES

Browsers render unstyled elements slightly differently; so, to normalize the code base for a better cross-browser development and maintenance, the second thing you will want to do is use a CSS reset.

Use one of the newer CSS resets, because the older ones do not style HTML5 elements. I strongly recommend Normalize.css[8] by Nicolas Gallagher[9] and Jonathan Neal.[10] Many of the older reset scripts (Eric Meyer's classic Reset CSS, for instance) are somewhat heavy-handed: they reset every element, and some of their changes are debatable,

[6] smashed.by/steve
[7] smashed.by/modernizr
[8] smashed.by/normalize
[9] smashed.by/nicolas
[10] smashed.by/jon

such as setting strong to be unbolded by default. Normalize.css resets elements more gracefully, and it handles a few browser quirks. It gives you as close to an even playing field as you have ever had in a browser. Jon Neal and Nicolas Gallagher have carefully explained everything the script does. Read the heavily commented code—it's fantastic!

Rebuilding Your Website

Having reached this point, you're probably thinking, "OK, it's time to start writing the website using the latest and greatest tags." When the new semantic tags were established, developers had to do some work in analyzing what classes and IDs they were applying to their websites. What they discovered really wasn't astounding at all: they were all using the same naming conventions (or slight permutations thereof). So, the names for the new tags will probably fit what we are already doing.

SECTION

A section tag could be used to break up distinct parts of the home page. Perhaps your blog consists of personal information about you, presentations that you've given and regular posts. You could probably break these up into section elements:

CODE

```
<section class="articles"></section>
<section class="about-me"></section>
<section class="presentations"></section>
```

Each of these parts could probably take its own header, so we could regard them all as reasonably important, thus justifying our decision to make them sections. If you were writing a book like the one you're holding in your hands right now, you might make each chapter its own section, and then make each section within a chapter a nested section. As with everything related to semantics on the Web, don't get hung up on the details. Choose an element based on the best information you have at the time and move on. Semantics are subjective—don't sweat the small stuff!

ARTICLE

When diving into HTML5, you might have wondered about the difference between section and article. Perhaps you guessed that an article tag is used primarily for blogs and online news. You would not have been far from the truth.

If you remember one thing about the article tag, remember this rule of thumb: if the piece of content would still make sense outside of its current context (that is, if the user could not see any of the page surrounding that piece of content), then it is probably an article. Hence, blogs and online news.

I use article for collections of content—say, a list of presentations that I have given in the past, perhaps with synopses:

CODE

```
<section class="presentations">
  <header>
    <h1>My Presentations</h1>
  </header>
  <article>
    <h2>An Introduction to HTML5</h2>
    <p>A four-hour workshop that I ran around Australia.</p>
  </article>
  <article>
    <h2>Compass and SASS</h2>
    <p>Use a well-written library of CSS so that you can focus
on the important things.</p>
  </article>
</section>
```

Eagle-eyed readers might ask, "That looks like a list! Why not use ul?" You would not be wrong; it absolutely could be a list element. But article indicates that, while these elements are similar, they are not related to one another. We could argue about this for hours; in the end, you will have to make up your own mind.

HEADER

Have you ever used a class or ID like masthead, banner or even top for the header section of a website? The header tag can be used for much more than just the head of the website. It can be used within an article or section, but it is entirely optional. Just use it when you need a block-level element to mark off space on the page for clarity. For example, I often keep titles and meta information in the head of a blog post.

FOOTER

The footer tag is just like header. You can use it within article or section or globally within body.

ASIDE

The aside tag can be used at the top level or within article. Its contents can be regarded as useful but not essential information.

For the mobile version of your website, for example, you could choose to hide aside elements. However you treat it, the tag forces you to make some decisions about your content. A blog post could be set up as follows:

CODE

```
<article>
  <header>
    <h1>All About Tractors</h1>
    <time datetime="2012-01-01">1 January 2012</time>
  </header>
  <aside>
    <p>Written entirely by Bruce Lawson</p>
  </aside>
  <!-- The body of the post goes here. -->
</article>
```

TIME

Did you spot the new tag in the last code snippet? The time tag is simple: use it to display the time. You can provide a machine-readable version, too.

CODE

```
<article>
  <p>Published on <time datetime="1984-04-03"
        pubdate>3 April 1984</time></p>
</article>
```

The pubdate attribute can be used to indicate the initial publication date of an article. The specification states that the pubdate attribute should be used only once for an article.

NAVIGATION

The nav element is, obviously, for the navigation of a website. You can nest nav tags to create a drop-down menu. The tag would not be suitable for that list of presentations I referred to earlier when talking about the article tag. Reserve nav for the structural navigation of the website itself. For example:

CODE

```
<nav>
  <ul role="navigation">
    <a href="/products">Products</a>
    <a href="/contact">Contact and Locations</a>
    <a href="/about">About Our Company</a>
  </ul>
</nav>
```

(Wondering what the role attribute is for? You will have to keep reading!)

FIGURE AND FIGURE CAPTION

You probably add a lot images to your pages. Have you ever considered the best way to apply captions to those images? Wouldn't it be nice to be able to wrap a caption tidily with its image? Well, that is what the figure tag is for.

CODE

```
<figure>
  <img src="whisky.png" alt="A glass of whisky,
        with a side of water in a small jug.">
  <figcaption>A glass of whisky, with a side of
        water in a small jug.</figcaption>
</figure>
```

It doesn't end there. You can also use the figure tag for video, svg and pretty much anything that is visual and could take a caption.

DIV

With all of these new tags, you would think that div was a thing of the past. It is not. Developers have been using the humble div tag for everything under the sun for years now, to the point of some contracting the debilitating disease of "divitus."

A div is a "division" and sometimes there is no better tag with which to describe a piece of content. Perhaps all you need is a box in which to add some styles. It happens. I don't blame you. Semantics are tricky. If you really cannot describe a piece of content using any of the HTML tags mentioned above, then use a div and don't feel guilty about it.

A FEW WORDS ON SEMANTIC OUTLINING

Now that we have some new sectioning elements (i.e. section and article), the plain old document outline that we have been used to has changed a bit. Sectioning elements can be thought of almost as documents of their own. In other words, the heading levels h1 through h6 can be used within them.

But hold on! This means you could encounter something like the following:

CODE

```
<body>
  <header role="banner">
    <h1>Full Frame: A Blog About Cycling</h1>
  </header>
  <article>
    <h1>Early Morning Over Black Spur</h1>
    ...
  </article>
</body>
```

Multiple h1s in the same document? That's crazy! Instead, I use headings to show the structure within a given section:

CODE

```
<body>
  <header role="banner">
    <h1>Full Frame: A Blog About Photography</h1>
  </header>
  <article>
    <h2>Early Morning Over Black Spur</h2>
    ...
  </article>
  <section>
    <h2>Buy Our Book!</h2>
```

```
    <section>
      <h3>Print</h3>
      ...
      <button>Purchase: $90</button>
    </section>
    <section>
      <h3>Electronic: PDF or eBook</h3>
      ...
      <button>Purchase: $15</button>
    </section>
  </section>
</body>
```

Not only does this make styling the headings easier, but it just feels much better: less confusing, and no swimming against the current.

Before having tags such as section and article at our disposal, we really only had h1 to h6 to describe the depth or hierarchy of a website. Now, we can describe infinite levels of depth and can represent each level of content accurately.

If after all this, you are still not sure which element to use, check out the fabulous flowchart on HTML5 sectioning elements[11] that was developed by Oli Studholme and Piotr Petrus. Print it out, stick it on your wall, and you will always know which elements to use. Maybe—no promises. As always, you will probably want to validate your HTML to keep it in check. I prefer Validator.nu.[12]

WORKING WITH WAI-ARIA ROLES FROM THE GROUND FLOOR

Roles for WAI-ARIA (short for Web Accessibility Initiative: Accessible Rich Internet Applications) have always been a part of modern HTML technology—so much that perhaps most developers glaze over as soon as they're mentioned.

The roles are designed to make websites and applications more accessible to users with screen readers. Any professional accessibility expert (there really are not enough of them!) would attest to the importance of WAI-ARIA; and for the most part, they are largely ignored, too.

[11] smashed.by/h5doc

[12] smashed.by/vldnu

Companies may talk a lot about how important accessibility is; whether they are conducting studies on it or designing for it is a different story. But your job as a website builder is to enable everyone to consume your content.

For blind and low-vision users, WAI-ARIA roles describe the context and purpose of the information laid out for them. A section of a page is not just visually different—it is contextually different, and screen-reading software can explain that difference to the user, enabling the user to interact with that section without missing a beat.

I am by no means an accessibility expert, but I will try to give you the best no-nonsense rationale for why the importance of WAI-ARIA goes far beyond accessibility. If you have ever used any of the sectioning elements described in this chapter, then you have probably encountered something like the following:

CODE

```
<body>
  <header>
    <h1>Tractors: An Interactive Guide</h1>
  </header>
  <article>
    <header>
      <h2>Tractor Maintenance</h2>
    </header>
  </article>
</body>
```

Did you spot it? The document has two `header` tags, both legitimately used. The problem lies in the CSS:

```
header {
  margin: 0 2em;
}
```

This element selector targets both headers. We could use a descendent selector (i.e. `body > header`), but that feels a little heavy-handed, not to mention that the top-level header might be the masthead for the entire website. We can use a WAI-ARIA role to our advantage here simply by adding `role="banner"` to the HTML:

CODE

```
<body>
  <header role="banner">
    <h1>Tractors: An Interactive Guide</h1>
  </header>
  <article>
    <header>
      <h2>Tractor Maintenance</h2>
    </header>
  </article>
</body>
```

This role attribute states that <header role="banner"> is a "global" element that contains content that applies to the entire website, rather than just the current page. This is a good fit and, thanks to a simple attribute selector, not too difficult to style:

```
header[role="banner"] {
  margin: 0 2em;
}
```

Because the header and footer tags can be used in multiple places, we seem to be left without a tag especially for the main content. Thankfully again, an ARIA role is ripe for the picking.

By adding role="main" to article (thus, <article role="main">), we can easily specify that the main content for the current document is contained within article. (You might have noticed in the snippets above that h1 is used in the top (root-level) header, and h2 is used in the nested header. Combined, this best describes the hierarchy of the document.)

You are probably starting to appreciate the gracefulness of this approach. The descriptions of content are becoming more detailed, and we are able to apply styles to our new tags with minimal effort. A third ARIA role is contentinfo, which is often used for copyright notices, privacy statements and general information about the current page or website. (Some people would call this "meta information.")

Finally, a fourth highly useful ARIA role to be aware of is navigation, which easily distinguishes a navigational section from a regular old list of links. Adding ARIA roles is a good way to make the content and context of your existing website more descriptive. Then, when you decide to overhaul the website, you can use the newer tags.

Hopefully, this short introduction has helped you see the benefits that semantic content brings to everyone. ARIA roles are an excellent example of this.

CLIENT-SIDE STORAGE

Now for an entirely new subject: HTML5 client-side storage. To date, we have had few options for storing data on the client's side. The most common has been the humble cookie-based session; but cookies are beset by a host of small problems, the more bothersome of which are these:

- The data you store in the session is transported back and forth between client and server with every request,
- The data you store has a limit of 4 KB;
- All cookies are timed to expire.

Cookies are not all gloom and doom, though. A cookie is what stores a user's data for logging into a website, and it helps the server to identify who the user is. It is clear, then, that we need some other options just for storing data. Thankfully, we have a fantastic solution in local and session storage. What are they? I'm glad you asked.

With localStorage and sessionStorage, we have two JavaScript APIs for storing strings of text to the browser. The sessionStorage API is purged when the user's session has ended (i.e. the tab or browser is closed), while localStorage sticks around until the developer (through JavaScript) or the user (through their browser settings) decides to remove it. The APIs are virtually identical—the only difference being the length of time of the storage.

Open your developer toolbar in a modern browser (i.e. one released within the last three years). Type in localStorage.setItem("name", "Ben"). In Webkit-based browsers, you will see my name stored under the "Resources" tab (you will have to expand "Local Storage" to see it). You have just stored your first item in localStorage.

Now, let's retrieve what we've stored by using localStorage.getItem("name"). You should see "Ben" printed neatly in the console. Finally, to clean up after yourself, use either localStorage.deleteItem("name") to delete my name or localStorage.clear() to remove everything in localStorage. When the user calls localStorage.clear(), they are only clearing it for the current domain. So, if the user stores some data on the website hosted at example.com and then switches tabs to google.com, they would see that they cannot access the data that they stored in the example.com tab.

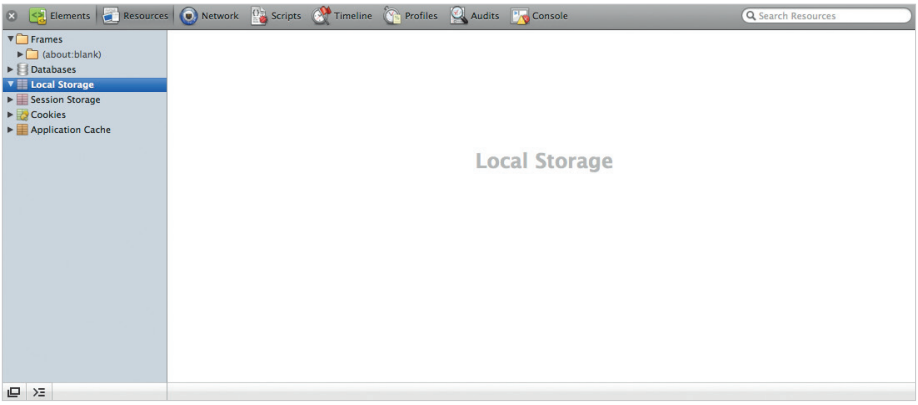

Figure 3.6. The local storage object explorer in Safari.

The localStorage API is highly useful. Say you are building a Twitter client which you want to be able to do the following things:

- Use in your desktop browser and on your mobile phone;
- When online, view tweets from your last online session;
- When offline, queue tweets to be posted later.

With localStorage, this is all possible. The snippet below illustrates this. (It is purely hypothetical, so don't sweat the small stuff.)

CODE

```
postTweet = function(tweetText) {
  // Check if we're online
  if(navigator.onLine) {
    // Hey, we're online! Send that tweet, baby!
  } else {
    // Hm, we aren't online right now. Better store it for an-
other day.
    localStorage.setItem("queue-" + +new Date(), tweetText)
  }
}
```

That wasn't hard, was it? To see all of the items in localStorage, we can iterate over them, just like an array:

```
for (item in localStorage) { console.debug(item) }
```

This will print out a list of all of the keys of items you have stored. Say you want to display your queued tweet? Here is how you would do that:

```
for (item in localStorage) { console.debug(localStorage[item]) }
```

The localStorage and sessionStorage API can be found in all modern browsers (including IE 8+), so there is no reason why you couldn't start building your own applications or just start experimenting with it in client-side applications.

In Summary

Before you replace all of the markup in your current website, take some time to study ARIA roles and browser performance and to generally learn how to structure code. Using a new tag will make only you feel good, whereas using ARIA roles will make a lot of people feel good. It sounds almost poetic, but it really is far simpler than that: it's your job.

To start using the new technology, don't feel like you have to use the new minimal DOCTYPE. Browsers will use whatever features they can when displaying your website. There is no "HTML5 mode," so you might as well dive in!

This is just a taste of the platform we call HTML5. We could go on for days about it, but instead, we will leave you with a few references to bookmark:

- HTML5 Please, html5please.us
  Want to know when you need to patch older browsers? Or when a super-new tag isn't quite ready for prime time? This website will give you the grounding to really turn it up a notch.
- HTML5: A Technical Specification for Web Developers, smashed.by/whatwg
  This guide is an abbreviated version of the full HTML5 specification. It removes all of those obtuse details that browser vendors need to build browsers. It is searchable, works on mobile devices (even offline), and was created by yours truly.
- HTML5 Rocks, html5rocks.com
  This website is maintained by staff at Google, and nearly every article posted is not only enlightening, but mind-blowing.

- HTML5 Doctor, html5doctor.com
  Aside from being written by a bunch of stand-up chaps, HTML5 Doctor has penetrated to depths that no others would dare. An excellent resource.

I'm an excellent name-dropper; throughout this chapter I have been dropping names like nobody's business. But it has not been gratuitous. The people and websites mentioned are leaders in the industry. I highly suggest you follow them on Twitter or Google+, subscribe to their blogs or buy them a beer. Nothing will teach you more about the Web than helping to build a strong online community. I'll leave you to start rebuilding your website. Good luck!

About the Author

Ben Schwarz funds his love of good food (at home) and sake (in bars) by designing sophisticated Web applications using standards-based technology. More than anything else, he is driven by a maniacal desire to produce not only elegant code, but also beautiful software in the hands of its users. He's also a committee member of Ruby Australia and joined the W3C CSS Working Group as an "invited expert" in December 2011.

About the Reviewer

Russ Weakley (1965) was born in Sydney, Australia, and lives in Chatswood West, a leafy suburb of northern Sydney. He has a diploma in visual arts and graphic design and works on user-focused Web design, markup and code, project management, user experience, accessibility and training. He has been working on the Web since 1995.

Russ has two young kids and, therefore, no time for hobbies. He used to have three dogs, but since all died, he doesn't have any. Russ' favorite color is black. An important lesson he's learned during his career is that "This too shall pass"—it applies to everything in life, including business. His personal message to readers is "Get busy!"