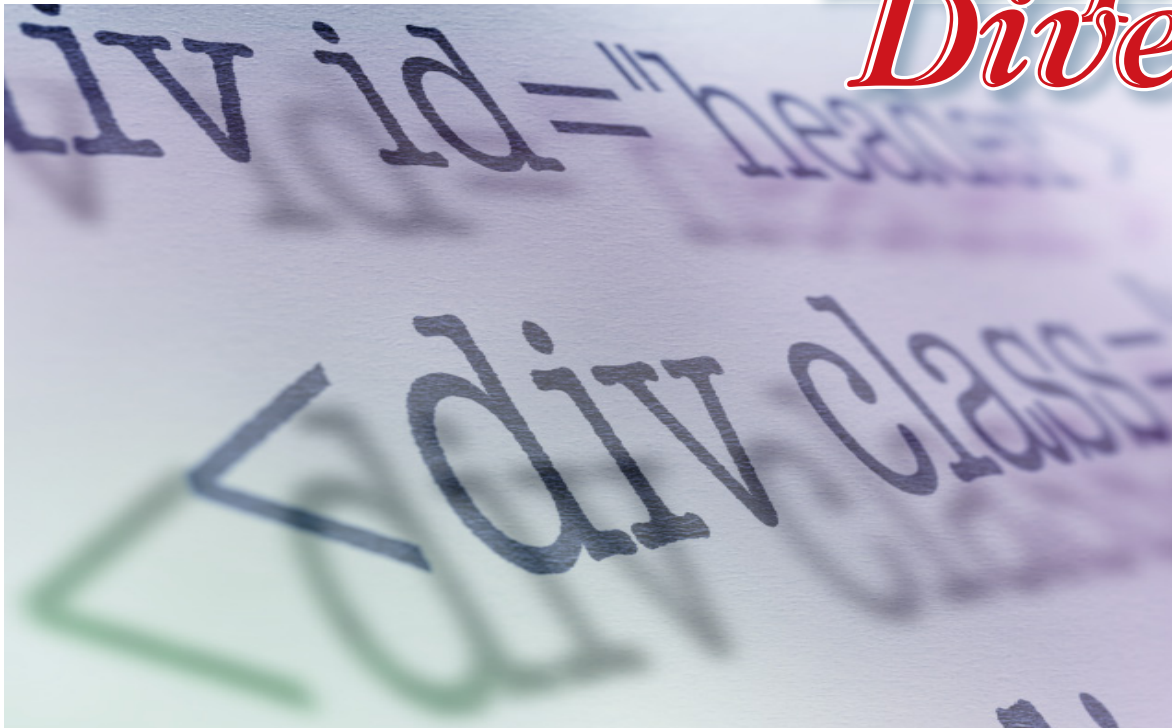


HTML5 *Deep Dive*



**The key new
HTML specs and how
they'll change the Web**

HTML5 presentation tools

HTML5 canvas, video, and other new tags make it easier to jazz up Web pages

 By Peter Wayner

THE FIVE CHARACTERS [HTML5](#) are now an established buzzword, found everywhere on the Web and often given top billing in slides, feature lists, and other places where terms du jour congregate. Nonprogrammers who must either manage or work with programmers are even beginning to pick up the term. Just two days ago, someone who can't manage a TV remote explained that he was sure his company's Web presence would be much better because they were using HTML5.

The five characters are in reality just the name of a document that isn't even finished. The W3C, whose job it is to build standard-setting descriptions of Web technology, has been contemplating the fifth version of the HTML standard for almost seven years. The [latest HTML5 draft](#) may finally become official in 2022, at least [according to Ian Hickson](#), one of the authors who works for Google.

That is clearly too far in the future for many bosses and potential clients, who've put the HTML5 buzzword on their checklist. The good news is that most modern browsers have implemented a solid collection of the features, and it's [quite possible to put HTML5 to work on your site](#) for users who are equipped with the latest browsers.

The biggest questions center on the newest technologies entering the standard. Many of these have been around for several years in various forms in various browsers, but now that the standard is coalescing, the major browsers are all lining up in support. Not least among the new specifications are flashy new presentation tools. The pun is intended, as the new layers will make it easier for designers to create slicker graphical extravaganzas with a collection of tags. The `<video>` and `<canvas>` tags go a long way to replacing Adobe's Flash.

HTML5 CANVAS

There may be no greater apostasy in HTML5 than the creation of the `<canvas>` tag, a marker used for reserving a rectangular chunk of real estate where JavaScript

code can manipulate individual pixels. In the original vision, HTML was a markup layer for the data only. The browser was responsible for deciding how to render the data in the best possible way for the current reader. And the scalable vector graphics set of tags -- now often considered part of HTML5 -- was the right way to draw lines by putting the information in an endless stream of tags.

Alas, artists don't think like computer programmers, and they don't grok the idea of separating the data layer from the presentation layer. When they want to put down a letter or a dab of color, they want it in one place and they want it to stay there -- or the harmony of the design will be forever destroyed.

The solution is the `<canvas>` element, an idea that originated with Apple, which incorporated `<canvas>` into the WebKit project. From there, it was copied by everyone but Microsoft. Eventually, Microsoft capitulated and added the option in IE9, but only after realizing everyone was using a free translation tool that mapped all of the methods from the `<canvas>` element into their own version.

Though Microsoft has joined the standard version, there are still differences between the results. Philip Taylor built a very complete set of [tests for the Canvas object](#) that runs through all of the various ways JavaScript can apply dabs or washes of color. Many of the most common routines -- like drawing a line -- are now implemented in more or less the same way in all browsers. However, there continues to be a surprisingly large number of areas where the browsers don't produce the same results from the same code. Many of the routines for rendering text and gradients act differently or just crash.

To test the Canvas tags, I [built a set of flying links that are animated](#) using a jQuery plug-in written by Graham Breach. The code looks for a set of links in your HTML, then starts drawing them in a cloud that rotates in response to the mouse. It illustrates a few of the actions that can be achieved with the Canvas element.

The code won't work in IE8 and earlier versions of Internet Explorer, which don't support Canvas directly,

but it will work if you use a thin translation layer called [ExplorerCanvas](#). This code will turn Canvas calls into code that is understood by the earlier browsers, a technology Microsoft called VML (Vector Markup Language). This translation layer will be necessary for anyone who needs to support the older but still very common versions of IE.

| NATIVE <CANVAS> SUPPORT IN CURRENT WEB BROWSERS | |
|---|-----|
| Apple Safari 5.0 | Yes |
| Google Chrome 14 | Yes |
| Microsoft IE 9 | Yes |
| Mozilla Firefox 7 | Yes |
| Opera 11 | Yes |

HTML5 VIDEO AND AUDIO

The <video> tag looks perfectly innocuous, like an tag where the pictures move. Anyone can add a video to a page by inserting the <video> tag, uploading the file to the Web server, and putting the right path in the source attribute for the tag. Voilà.

If only progress were that easy. While the format for the tag is always the same, the format for the file varies dramatically. This might not matter if the browsers all supported the same collection of formats as they do with the GIF, PNG, and JPG files used to hold still images. But they don't. Everyone has their own idea of which video formats belong, and support can change from revision to revision of the browsers.

The most common formats are the Ogg Theora, developed by an open source group, Xiph.org; H.264, built by the MPEG group; and VP8, created by On2, a company recently purchased by Google. There are others, but they don't have as much support. Apple's Safari browser, for instance, displays anything that can be decoded by QuickTime.

All three of the main formats offer pretty comparable results, although video specialists can often find artifacts to argue about. The biggest point of contention is the cost of satisfying anyone who might hold a patent. The H.264 format was one of the most commonly used formats in the past, but it requires payment for patent licenses from

the MPEG LA group. As an advantage, there are a number of chips that can speed up reconstructing the pixels from the compressed MPEG file. These can lower the battery consumption of handheld devices like the iPhone.

The On2 patents are now available for use royalty-free, which makes them more desirable for everyone. At this time, no one is publicly discussing patents that might govern the Ogg Theora format, but there are worries that the patents filed long ago may emerge some time in the future, giving the patent holder the right to sue everyone who's been using the format since then.

The patent questions may not be too important for small website designers and filmmakers because many tools come from companies that pay the licensing fees for encoding and decoding the H.264 stream. That license may not be enough for bigger fish, however, because the MPEG LA group also wants fees for broadcasting the video to big audiences. If you reach certain thresholds, you may need to buy this license or you may not. One [MPEG LA press release \[PDF\]](#) promises not to charge patent royalties on rebroadcast before 2015. After that? It's all a bit complicated. The lawyers won't be cut out of the loop for some time.

Is there a solution? Not an easy one. While everyone continues to experiment with all of the features, the best solution is probably to encode the video in multiple formats, then use a script to detect the browser.

A more entertaining way of testing HTML5 video in your browser is [found in this table filled with flying InfoWorld logos](#). If you can see the logo, your browser supports the format.

The <audio> tag is similar to the <video> tag, largely because it's just a subset. The <video> tag produces both sound and images, and the audio tag produces only the sound.

| NATIVE <VIDEO> SUPPORT IN CURRENT WEB BROWSERS | |
|--|-----|
| Apple Safari 5.0 | Yes |
| Google Chrome 14 | Yes |
| Microsoft IE 9 | Yes |
| Mozilla Firefox 7 | Yes |
| Opera 11 | Yes |

In most cases, the audio tracks are a subset of the video format. The popular MP3 format for music and sound is a subset of the MPEG standard for encoding video.

For now, it seems as if the audio support will largely mirror the video support. If Firefox is pushing Ogg Theora video, then it also supports the Ogg Vorbis audio format because Ogg Vorbis is used for the audio portion of Ogg Theora. If some browser supports WebM for video, it will undoubtedly support WebM for audio. The one inconsistency may be the MP3 format, which is technically part of the general MPEG-4 video format but is now so common that browsers will support it independently.

SCALABLE VECTOR GRAPHICS

As I mentioned in the Canvas section, the Web was built on the idea that data would be delivered in one format, then rendered or interpreted by the local computer. The Scalable Vector Graphics (SVG) format was invented to translate ordinary ASCII text into graphical shapes. The first committees began in the relatively ancient year of 1999, but the hard work is finally reaching fruition. Now that Microsoft has added SVG support to IE9, all of the major browsers support the format, more or less.

The goal of the SVG format is to bring the same kind of infinite precision to specifying a drawing that PostScript brought to print documents. Instead of rendering objects by specifying pixels, the drawing is made of lines and shapes, which are spelled out with tags like this one for creating a circle with a radius of 40 pixels:

```
<circle cx="100" cy="50" r="40" stroke="black" stroke-width="2" fill="red">
```

The results are great for line drawings because the browser can create an image tuned to the resolution of the screen. The user can zoom in or out and the video card renders the result. Animated videos and games made of animated sprites are easy to specify and deliver because they can be so small. Even though all of the tags make the format insanely prolix, basic gzip compression can remove 80 percent of this syntactic lard. The SVGZ files are precompressed.

Does Microsoft's arrival at the party mean that everyone speaks the same language now? Of course not. Cell phone companies worried about wasting battery power on ornate SVG panoramas came up with two smaller subsets: SVGB (SVG Basic) and SVGT (SVG Tiny). These leave out a number of features that probably don't add much to tiny mobile screens, such as the ability to specify dimensions in values other than pixels. There are no picas or millimeters, and SVGT allows solid color fills only.

There are also numerous differences on the desktop implementations of the browsers. While they're all said to implement the core version of SVG, some developers are experimenting with doing even more with the flexible standard. One group, for example, started adding new <animate> tags that describe a path for objects to move along. After drawing the objects, the SVG engine can calculate a new location for them and draw them again. All the same math and tag infrastructure for drawing the lines can be reused to plot the path the objects should follow. At this point, the Gecko and WebKit browsers support the feature, but Microsoft hasn't built it yet.

Another plan is to let the browser apply all of the SVG effects such as filters and clipping to any arbitrary piece of HTML. The Mozilla team first implemented this and created a [draft of SVG effects for CSS](#) for the HTML5

VIDEO AND AUDIO FORMATS SUPPORTED IN CURRENT BROWSERS

| | H.264/MPEG-4 | Ogg Theora/Ogg Vorbis | VP8 |
|-------------------|--------------|-----------------------|-----|
| Apple Safari 5.0 | Yes | No | No |
| Google Chrome 14 | No | Yes | Yes |
| Microsoft IE 9 | Yes | No | No |
| Mozilla Firefox 7 | No | Yes | Yes |
| Opera 11 | No | Yes | Yes |

team to consider. The other browsers offer some support, but they're widely considered to be more buggy – and inconsistent with the way the ideas are implemented in the Mozilla browser. This shouldn't be surprising, given the same people at Mozilla wrote both the code and the spec.

A third area of inconsistency is in the implementation of SVG fonts. Unlike PostScript, the basic version of SVG doesn't make it easy to turn any arbitrary pattern of lines into a glyph. The WebKit browsers offer basic support for SVG fonts, so it's easier to add words to SVG drawings. IE and Gecko are still working on this.

Another potential extension has also been adopted by some browsers. After creating the rendering engines that produce infinitely crisp images with endless precision, some SVG users took a step back and decided the images might be nicer if they could be a bit blurrier. (Some people are never happy.) In any case, Chrome, Opera, and Firefox offer SVG filters that average all nearby pixels to produce smoother, prettier, often bizarre effects that aren't as cold or as crisp as line drawings.

My [quick SVG test](#) offers an easy way to check your browser's ability to host both embedded and framed SVG tags. A good collection of [SVG examples and tests collected by Andre M. Winter](#) shows many of the ways that people are experimenting with the basic SVG standard, as well as several of the proposed additions and extensions.

WEBGL: 3D CANVAS

The WebGL format helps bring all of the power of OpenGL to the Canvas. Many of the development versions of the major browsers already support it, but they don't turn this on at the start. You need to activate it yourself by

twiddling some of the configuration files. WebGL support comes with the current versions of Chrome and Firefox, but is still in beta in Safari and Opera. Microsoft has no plans to support WebGL in Internet Explorer.

The technology is not officially part of the HTML5 standard, but many people mention it in the same context because it is both experimental and tied to the Canvas object. The performance, of course, depends heavily on your video card and the memory available. Users with powerful graphics hardware report running an open source port of Quake 2 at 20 to 30 frames per second.

Andor Salga offers [a collection of WebGL routines](#) that test most corners of the spec.

MORE HTML5 GOODNESS

The next several articles dig deeper into the various enhancements to the HTML5 standard, many of which are hidden from view. While the <video> tags are hard to miss, the HTML5 standard also includes a number of ways that a website can begin to act like locally installed software. The second article focuses on local data storage, a set of features that lets a website stash information on the local disk and even access it when there's no Internet connection. The third article digs into the newer ways that the new Web apps can communicate when there is a connection. The fourth piece examines enhancements to HTML forms, and the fifth summarizes a number of new features, such as geolocation, that don't fall into an easy pile.

All of these sets of new HTML5 features will radically expand the abilities of Web applications to be more than just a pile of links.

VIDEO AND AUDIO FORMATS SUPPORTED IN CURRENT BROWSERS

| | Displays SVG | SVG effects for HTML | SVG filters | SVG fonts | SVG in CSS backgrounds | SVG animation |
|-------------------|--------------|----------------------|-------------|-----------|------------------------|---------------|
| Safari 5.0 | Yes | Partial | Partial | Yes | Yes | Yes |
| Chrome 14 | Yes | Partial | Yes | Yes | Yes | Yes |
| IE 9.0 | Yes | Partial | No | No | Yes | No |
| Firefox 7 | Yes | Yes | Yes | No | Yes | Yes |
| Opera 11 | Yes | Partial | Yes | Yes | Yes | Yes |

Source: When can I use...

HTML5 local data storage

Web Storage, Web Database, and other specs for transforming Web pages into local applications

 By Peter Wayner

OF ALL THE CHANGES bundled in the HTML5 drafts, few are as radical or subversive as the options for storing data locally. From the very beginning, the Web browser was intended to be a client in the purest sense of the word. It would display information it downloaded from a distant server, and it would do everything the distant server would tell it to do.

Programmers discovered the limitations to this fairly soon, and before long browsers started offering website developers the chance to leave a little piece of data behind. The creators tried giving this 4,096-byte text string a cute name, “cookie,” but that didn’t stop the controversy. Cookies became the focus as the greater public started to wonder just how the inscrutable gnomes at the central office were tracking their every move. People demanded and got the ability to delete cookies, which limited their possibilities for the developers.

There were deeper problems with the spec. The cookies weren’t just stored in the computer – they were sent back to the server with requests. Savvy Web developers know it’s not worth using many of those 4,096 bytes because the cost of accepting too much data on each and every call will drive up bandwidth bills and slow responsiveness.

The HTML5 standards crew chose to fix all of these problems and lay the foundation for the final victory of browser-based software by giving the JavaScript programmer the ability to store practical amounts of data on the local computer. At the simplest, this might be a cache for all of the calls to the central computer, but it can be much more. The more sophisticated programmers might allow the users to store their Web pages locally, imitating the last major feature of desktop software by gaining access to the disk. There’s no need to install software any longer.

WEB STORAGE: SESSION STORAGE

The simplest level of [Web Storage](#) will store data for the current session – in other words, as long as the browser tab or window remains open. This may not be a hard limit,

however, because the spec leaves open the opportunity for the browser to keep this data around “during restarts.”

There’s not much to the mechanism. Each document gets a *sessionStorage* object with a few major functions: *setItem*, *getItem*, and *clear*. The items are just pairs of keys and data just like an associative array. The data is a clone of the current values.

That’s about it. New documents get new objects. There’s not much difference between storing information in this *sessionStorage* and declaring a global variable.

WEB STORAGE: LOCAL STORAGE

The real advantages come with access to the *localStorage* object, which looks quite similar to the *sessionStorage* object but behaves very differently. Where the *sessionStorage* forgets, the *localStorage* remembers. Data is supposed to stick around even after the window closes and the computer shuts down.

The persistence goes deeper. Two windows visiting the same website should share the data. A change by the code running in one window should change the data accessed by the other. As the spec says, a *storageChange* event in one window should propagate to all windows. (This isn’t always the case. In some browsers *sessionStorage* isn’t shared between tabs, while in others it is. The edge conditions are not set. The sharing of *localStorage* and *sessionStorage* is not perfectly implemented yet.)

There’s been some debate over how tightly to limit connection to this object. Right now only scripts from the same scheme, domain, and port are allowed access. This is pretty strict and prevents any confusion that might come about when people load common scripts or switch between HTTP and HTTPS.

While this sounds like a dream for many Web programmers, there’s the very distinct possibility that it could cause nightmares because it’s easy for two windows to access the same data and create a race condition that corrupts the data. There’s a great deal of debate over whether the storage object should defend against this by implementing a mutex (a mutual exclusion algorithm)

that can limit data corruption.

[One recent draft notes](#), “The use of the storage mutex to avoid race conditions is currently considered by certain implementors to be too high a performance burden, to the point where allowing data corruption is considered preferable. Alternatives that do not require a user-agent-wide per-origin script lock are eagerly sought after.”

This probably won’t affect many of the simplest uses of the object, but it could easily produce bizarre errors when people leave several windows open. I often leave Web mail windows alive on my desktop, then open another instance because I’m too lazy to dig through the pile of windows and tabs. Programmers must be aware that different instances of their code will run in the same browser, and this code will have access to the same data.

Note this quote from the spec: “Different authors sharing one host name, for example users hosting content on geocities.com, all share one local storage object. There is no feature to restrict the access by pathname.”

How much room do you get? Can you count on having enough room? Is there a way to defend against DNS spoofing? For all of the questions that localStorage answers, it creates many more.

WEB SQL DATABASE AND INDEXED DB

The key-value pairs in the *localStorage* object are usually powerful enough for many basic projects, but they’re not comparable to relational databases that store the information in indexed tables. For that, there are not one but two options.

The first, the [Web SQL Database](#) standard, was drafted and implemented before being abandoned for a more abstract version. People using WebKit browsers and Opera will find that a small database engine, SQLite, was grafted onto the JavaScript engine to let people create tables and store rows using all that knowledge about SQL.

The work, though, was for naught because the committee decided it wanted something else. While the features are still available in the supporting browsers, the Web database standard is now filled with language designed to scare people away. “Beware,” it warns. “This specification is no longer in active maintenance and the Web Applications Working Group does not intend to maintain it further.”

The new king is the more abstract idea of the [Indexed Database](#), an SQL-free pile of keys and values just like the *localStorage* object. The difference is that an index

can speed finding the necessary object. In practice, this seems to mean that the browsers will store each table in a B-tree to speed lookup and allow the programmer to page through the data in some repeatable order.

The indexed storage also includes the ability to execute the changes as transactions, eliminating the questions about race conditions that may bedevil the *localStorage* object. This may warm the hearts of database programmers, but it may be too early to know exactly what to expect. The version of the draft I read while writing this includes the line, “TODO: decide what happens when dynamic transactions need to lock a database object that is already exclusively locked by another transaction.” There are many details to work out.

FILE API: FILEREADER

The final apostasies in HTML5 are the *FileReader* and *FileWriter* objects, two devices that reach outside of the browser’s sandbox and actually touch the file system. It’s one thing to offer the JavaScript programmer the ability to store objects from trip to trip, and it’s another to let them have access to functions like *readAsBinaryString*.

The [File API](#) is not widely implemented yet, but it promises to dissolve the wall separating the “personal” part of the PC with the “inter” part of the Internet. There’s even an updated scheme to make all of the *file:///C:* URIs behave more like distant websites. JavaScript will see fewer differences between loading a local file and downloading data from a distant website with an *XMLHttpRequest* call.

The details of this API are still missing. The spec is filled with useful suggestions like, “System-sensitive files (e.g. files in /usr/bin, password files, other native operating system executables) typically should not be exposed.” Well, duh -- but notice the use of the word “should.” The spec suggests that the browser “may” raise a *SECURITY_ERR*. The details are still in flux, and I don’t think anyone knows what may come of opening up this Pandora’s box. Perhaps the Web applications will routinely need access to the /usr/bin directory and all of the *SECURITY_ERR* events will drive the user mad. We can’t be certain.

FILE API: FILEWRITER

If the *FileReader* API sounds like a recipe for massive privacy invasions, imagine what potential evil lurks in an API with the name *FileWriter*. Presumably there will be

much good as well, including the ability to simplify the installation of new software. We can only hope.

The design of the [FileWriter API](#) is similar to all the other [File APIs](#). You create a block of bytes called a *Blob*, pass it to a *FileWriter* object, and invoke the `append` or `write` methods. The next thing you know, your disk is filled with viruses. There are also mechanisms so that the viruses can choose between installing themselves synchronously or asynchronously. The data can be found inside the file with offset methods like `seek`.

The jokes about the viruses should remain jokes if the security model works as planned. Although the draft spec doesn't say much about the security model, it looks as if the goal is to give the average user all the rope they need to tie themselves up and hang themselves inadvertently. The browser will pop up a search box and ask where the data should be stored. Sensitive areas of the OS will probably be off limits, but I still wonder about the damage that could be done with supposedly safe sections. Imagine, for instance, an application that can write a block of bits to the Desktop directory and give it the name "Click Me." What percentage of the world can resist a message like that?

We can hope that the browser builders will move slowly with this tool. Perhaps they'll leave it disabled until a user decides to opt in through the Preferences interface. Ideally, they'll bury the access even deeper as Mozilla does with the preferences hidden beneath Firefox's `about:config` menu.

OFFLINE WEB APPLICATIONS: APPCACHING

Letting Web pages store data locally can certainly help reduce network traffic by caching the AJAX calls and other important information displayed to the user. It doesn't do anything, though, about the Web page caching itself. (This isn't exactly true because websites can download much of their own logic and evaluate it, but a bit of bootstrapping code is still necessary to start up the page.)

The [AppCaching API](#) spells out just how long the browser can keep pieces of the Web pages stored locally. This not only reduces the need for reloading the pages, but it also makes it possible for the Web pages to work without an Internet connection. In other words, it makes them more like installed software.

There's already a fair amount of support for caching in the header tags of an HTML page, but it doesn't extend to the JavaScript files or the CSS pages. The [AppCaching](#)

API solves this by creating a manifest file that lists all of the important parts of the Web page so that the browser won't get confused. The manifest is listed in the big HTML tag like this: `<html manifest="page.manifest">`. The browser looks at the list of files -- delivered with its own MIME type `text/cache-manifest` -- and treats them as a unit.

The application cache treats a few items differently. Data calls to server-side CGI functions, for instance, can be labeled in a separate part of the list so that they're not cached. There's also a general fallback section to handle problems. I'm guessing that these might be most useful when loading parts of a page from problematic websites. It could provide icons for unavailable images, for instance, that can't be found at a photo sharing site.

When the manifest changes, the browser reloads everything -- a process that might be transparent but doesn't have to be. There's an *ApplicationCache* object that fires events whenever significant actions happen. If the code is updated, these events could be used to tell the user.

EMBEDDING CUSTOM NONVISIBLE DATA

Another interesting option is burying the data inside of the DOM tree. The JQuery framework, for instance, comes with a data method that will [attach arbitrary objects to pieces of the DOM tree](#), allowing you to store data with some part of the screen that the user can't see. This makes it simpler to code operations like drag and drop because there's no need to keep separate track of the data and the representation of it.

Some HTML5 developers want to bring this feature to HTML5 in a standard way. The [proposal](#) lets arbitrary name-value pairs be attached to parts of the DOM tree. It isn't well supported yet.

HTML5 MICRODATA

The idea behind the [HTML Microdata spec](#) is to create a class of machine-readable metadata tags that websites might add to the visible information. Instead of just inserting the characters "January 1, 2011" or "New Year's Day," the website builder can add the `<time>` tag like this:

```
<time itemprop="birthday"
datetime="2011-01-01">New Year's Day</time>
```

Search engines and other Web crawlers will be able to understand the underlying data without doing too much work trying to parse or guess the writer's meaning. It will

HTML5 data communications

Cross-document messaging, WebSockets, server-sent events, and other interaction boosters

 By Peter Wayner

FROM THE BEGINNING, Web users have had mixed feelings about the way their browser communicates. On one hand, the idea of a tightly controlled sandbox is appealing because it limits the damage a website may do to our personal data and to the Web as a whole. Without these controls, just clicking on a link could unleash viruses, worms, and worse.

On the other hand, programmers have always complained about the browser's restrictions, pointing out the ways they limit the services that might be made available. Every AJAX developer can easily identify one way they could make their code that much cooler and more awesome if only the browser would loosen the rules governing the sandbox, but just this once and only for their code.

HTML5 is here to change this view toward communication – radically in some ways and slightly in others. The rules for communication are changing, and in most cases, the developers are getting their wish. The limits are loosening but with enough strictures intact to provide greater flexibility without really endangering anyone.

The models should be familiar to most programmers because they're largely extensions of ideas that are common and generally successful in other parts of the stack. Most developers of user interfaces, for instance, arrange for the buttons and sliders to send events back and forth to other parts of the code called listeners. The HTML5 team extended this idea by arranging for code from different websites to tunnel through the wall between the different sandboxes that would normally prevent them from communicating. The sandboxes aren't being merged; the browser is simply offering a tunnel that's used only if both sandboxes agree to communicate.

All the specs have a similar flavor. The old idea of forcing the code to live in a sandbox isn't going away. The specs are just grafting on ways to use traditional approaches to break the difficult rules in a few simple

and well-confined approaches. The sandboxes are growing well-guarded tentacles that link them to each other.

CUTTING THROUGH COMPLEXITY

The benefits to this should be obvious. Programmers have crafted a number of hacks to work around cross-site scripting and cross-site information fetching, and these add to the programming complexity and the network traffic. Many websites host proxies simply to get around these issues. The new HTML specs will let savvy programmers slice away at the layers with a machete, adding speed while cleaning up the code base.

The first place we'll start to see these new technologies appear will probably be in advertising. The most creative attempts to get us to part with our money already link together disparate parts of the pages. When the different blocks can communicate, these attempts will be even more clever. The aggregation sites that glue together widgets, RSS feeds, and other chunks of data will probably be next, although it's not clear that these need to offer the different building blocks a chance to communicate.

A good place to look for a preview might be the [Yahoo Pipes](#) site, which shows how people are creating interesting mashups from different feeds. The Yahoo site, of course, does most of the work on the server, whereas the HTML5 specs allow the client to take over some of these chores. Yahoo Pipes is filled with mashups that link RSS feeds to maps and other services. One, for instance, looks up the latest reviews from a movie site, then searches out the trailers from another.

Here is a tour of the major new HTML5 specs for opening up the communication between the layers.

FROM WEB DOCS TO WEB APPS

The original idea of the Web document was just that: a document with words and pictures in a rectangle. Sometimes the rectangles were subdivided into smaller rectangles with slightly different coloring, but still filled with words and pictures. It was an easy model when the job was merely distributing words and pictures.

After AJAX became popular and the document turned into software, some people wanted the rectangles to talk with each other. That was fine when all of the rectangles came from one source, but this dialog failed when the rectangles originated from different sources, as they do on most sites with advertising sold by companies like Google. The rectangles filled with content come from one site, and the rectangles filled with ads come from another.

The problem became even more confusing as Web developers started swapping widgets that made it easy for one website to include a small rectangle with content from another website. A blog, for instance, might want to include a widget with a weather forecast, movie times, sports scores, or all three.

The original idea of the Web was that the words, images, and JavaScript code from one source shouldn't fraternize with those from another source. Naturally, the website developers started to say, "This would be so much cooler if the widgets could talk to each other." Then the weather widget could display the forecast for the games in the sports widget or something similar.

The only solution was to run through some central server, but that was always a bit of a kludge. In many cases, it was purely impossible because the cross-domain scripting limitations imposed by the browser blocked the contact. AJAX programmers figured out more and better solutions, like loading JavaScript from all of the widget sites, but they are still kludges and not very secure at all.

CROSS-DOCUMENT MESSAGING

The HTML5 team is pushing the idea of [cross-document messaging or communications](#). This would let the different rectangles set up a communications path by creating listeners that wait dutifully for message events from other rectangles. There's no need to run through a central server. The code just packs up the messages and sends them to some listener. It's a little like one neighbor talking to another by tying the message to a rock and throwing it over a fence. Good fences make good neighbors.

The API does arrange for communication to be limited to specific domains. It's not possible to broadcast messages to all listeners who might want to receive them. They need to be targeted at windows of particular documents. For sustained communications, a specific

channel can be created to act like a two-way pipe.

The details aren't final by any means, and programmers should be wary. Although Chrome, Firefox, IE, Opera, and Safari all implement the feature and let you create the listener objects, the draft of the API spec contains a warning: "Implementors should be aware that this specification is not stable."

At this point, the paradigm is well understood because many UI programmers use a similar model to structure the way that an application communicates with itself and its many parts. The changes probably won't affect the basic model of listener and event, but the details are still being worked out. To see how your browser handles cross-document messaging, point it to my [cross-document messaging test page](#).

CROSS-ORIGIN RESOURCE SHARING

Sending messages is not the only solution for sharing information between different websites. The [cross-origin resource sharing](#) API loosens the controls over AJAX calls to anywhere but the home domain. A website can specify a list of allowable targets, and the *XMLHttpRequest* calls will just work -- at least, they should.

The information is bundled in the header of the document, which places it a bit out of reach of the average HTML coder. The server itself must be reconfigured to include parameters like these:

```
Access-Control-Allow-Origin: http://infoworld.com
Access-Control-Max-Age: 10000 Access-Control-
Allow-Methods: PUT, DELETE
```

Any website that receives this will be able to put and delete data from InfoWorld.com for all of 10,000 seconds. The original website, in essence, is giving the software the permission to call up someone else for extra data. The deadline may be useful for closing out sessions and blocking access when people inadvertently leave windows open.

WEBSOCKETS

When AJAX calls take a long time to complete, they traditionally fail with a time-out. This may be acceptable for basic tasks like collecting the latest headlines, but the eventual time-out makes it a bit trickier to implement interactive websites. Developers have traditionally worked

HTML5 forms

The new specs provide more control over data input and validation while off-loading much of the work to the browsers

 By Peter Wayner

THE CHANGES AND ENHANCEMENTS to the form tags are some of the most extensive amendments to the HTML5 standard, offering a wide variety of options that once required add-on libraries and a fair amount of tweaking. All of the hard work that went into building self-checking widgets and the libraries that ensure the data is of the correct format is now being poured into the browser itself. The libraries won't be necessary -- in theory -- because the work will be done seamlessly by all browsers that follow the standard. In practice, we'll probably continue to use small libraries that smooth over slight inconsistencies.

The new HTML specifications include input types that offer a number of new options for requesting just the right amount of data -- say, a form element that requests the time in different levels of granularity, such as month, week, or minute. Other new input types insist that the user type in only valid URLs or email addresses. All of these input fields will be tested to ensure that the text in them is valid and that the user's progress toward satisfying the data integrity police will be tracked by a series of events. There are even hooks for a value sanitization algorithm that checks the information and perhaps cleans it up with some AJAX.

Compliance with these options is gradually appearing in the browsers. At the time of this writing, for instance, Chrome lets you pin the *min* and *max* for some dates, but you can't install a value sanitization function. The minimum and maximum values are, of course, the simplest controls to create. It's much harder to offer the deeper hooks.

Holes like this are sprinkled throughout the new options. Firefox, Safari, Opera, and Internet Explorer are all slowly rolling out the new form features, and they're pretty much done with the most important ones. Alas, not all of them support the new features in exactly the same way, so it's still a bit complicated to create content that uses them. But as these gaps close, the new form

elements will make it much easier for Web developers to gather information and enforce a few rules that keep the users in line.

To find out if your browser supports the new input data types and controls, try my [experimental HTML5 table at wayner.org](http://experimental.html5table.at.wayner.org).

INPUT ELEMENT TYPE

In the old days, there were only a few types of input widgets in the forms: radio buttons, check boxes, and catchall boxes that accepted text. Even the color wheel choosers in JavaScript libraries would simply place the RGB values for selected hues in a text input box. If you wanted to do any value checking, it was up to you to implement it with JavaScript. In time, data validation became relatively easy to do with the various libraries, but it was still up to the programmer to handle.

The new options take on some of these chores. The compliant browser will now make a distinction between a wide range of data types, including dates, email addresses, numbers, and URLs. Each of these types has several more specific options. The date field may ask for a full date, a year and week alone, a year and month alone, or just the time of day. If you want to be very specific, you can mix together a date and time with the option of including or leaving off a time zone.

Some of these types seem like invitations to trouble. I'm happy I'm not responsible for implementing the code that will validate all of the different kinds of telephone numbers around the world. In America, it's a hassle because some folks will punctuate the number in odd ways, like wrapping the area code in parentheses. Freezing these rules in the browser standard will be problematic if the phone companies dream up new ways of using the numbers. Of course, if that day arrives we can always override the validation because there are attributes that allow specifying *novalidate=true* or *formnovalidate=true*. Or we can just forget about the extra features, flip the input type back to pure text, and use JavaScript the way we've always done it.

INPUT ELEMENT TYPE ATTRIBUTES

Choosing the type is just the beginning of the fun when creating these new form elements. Each type may or may not have additional features that can be specified with additional attributes. Many of these attributes are straightforward. For example, *min* and *max* can only be used with times and numbers, and not with unlikely items like email addresses, even though they're technically sortable.

By my quick count, there are 37 attributes and 14 different types. The current version of the [HTML5 input element specs](#) includes a table that shows which attributes are allowed (limiting the *max* value of a number, for example) and which are ignored (limiting the *max* value of an email address) for which types. I'm still a bit confused by why you can only specify a *placeholder* for some types. This short suggestion (for example, "your email address") isn't available for times or colors. Most of the other pairs that are allowed or forbidden are easy to understand, but I think most will find one or two combinations that they wish were there.

The new mechanisms are meant to extend the status quo, and that means not changing some of the old patterns. To me, it might make sense to allow each type of input to be hidden with an attribute, but the new standard continues the old approach of making "hidden" a type that accepts generic text. That's the price of backward compatibility.

CLIENT-SIDE FORM VALIDATION

Specifying the type and attribute are just the beginning because the validation process is fairly transparent. While the form will handle most of the work for you, it will also allow a number of hooks for interrupting the process or replacing it.

When something seems incorrect, the validation will set up a data structure that can be queried. The method *validity.patternMismatch*, for instance, will return true if a pattern is specified but the data doesn't fit it.

If you want to specify your own validation, you can add a custom message indicating why the data might not be acceptable. You can fire off this routine with an *oninput* or *onchange* event.

Problematic input data can also trigger events of their own that you can trap. Data checks can be set off by hitting the *checkValidity* method.

It's all pretty flexible and built in a way that will be

familiar to everyone used to the traditional mechanism of attaching functions that listen for particular events. There are probably three or four different ways to check each form field.

The standard also includes a good reminder that the clients can't be trusted to enforce these rules. Although testing the data locally will save time and energy, it won't be a perfect solution because older browsers may not implement the validity checks. It's also possible that clever users may override some of the methods and block checking. For this reason, any serious data validation rules must be re-evaluated at the server. The browser can't be trusted.

CUSTOMIZABLE OPTIONS

Simply validating the data as acceptable or not acceptable is not the only option anymore. HTML5 includes several attributes that let you offer help and suggestions to the visitor.

The simplest option lets you [turn on spell-check](#) for any input element that's marked as editable. This will normally apply to form elements like *textarea* but may also include any part of the document that's marked *contenteditable*. (Editable content is discussed below.) The attribute *spellcheck='true'* determines when it applies.

I'm guessing that the *spellcheck* attribute also toggles the grammar checker, but it's not immediately apparent to me. The title of the section of the spec is "[Spelling and grammar checking](#)," but the text only mentions one attribute called *spellcheck*. If I were designing the spec, I would make them independent, if only because I've found that one feature is much more accurate than the other.

The *datalist* element lets you add a list of strings that can automatically complete a form element. The structure is like the option tags used in select elements. At this point, only Opera seems to support the feature, and some feel it makes the HTML that much grungier by larding it up with suggested answers. I'm also a bit annoyed by the idea that each potential option comes with a label that is displayed and a value that actually fills up the form element. It seems like a dangerous way to hide functionality from the user and perhaps trick them into thinking that one thing is going in the form (the label), while filling it with another (the value).

I was also confused by the possibility of having an external list of data options stored in an XML file

independent of the current HTML form. This would not only simplify the HTML but also make the data reusable in different pages. It seems like a good idea, but the spec doesn't mention it yet. I've found only secondary references to this option.

AUTHENTICATION

One of the most tempting options brings authentication or certification to the form information, but it is still rather unformed and not very well implemented. The so-called *keygen* element adds some form of cryptography using public-key encryption, but it is only partially implemented on Chrome, Firefox, and Opera, despite [dating from the time of Netscape](#). The potential power is huge, but I think it will take several more iterations to find a good set of features that work the way that people expect.

The idea is to get the browser to offer a way to generate pairs of public and private keys automatically. Many programmers who've tried to use *keygen* say it's confusing for the average person because it requires too much understanding of such details as the length of keys. There are also deeper issues about how users might move the certificates from computer to computer or how malware might target them.

In the future, the option might include a better way to automatically use a key pair to sign all data in the form, not just the challenge attribute attached to the *keygen* item. This, of course, requires a more standard mechanism for creating the signature over all possible forms of data. The standard hash functions and message digests are probably a good place to begin. This will have to wait until the feature is more fully formed.

DRAG AND DROP

The ability to drag HTML elements around and drop them somewhere else is an old option for Web designers who are willing to use their own libraries, but it's always been mired in some confusion. After Microsoft included drag-and-drop support in what was called DHTML in 1999, developers had to struggle with cross-browser problems. A number of good cross-browser scripts appeared over the years, and many sites use them, even though they seem to confuse the public, who tend to expect the items on Web pages to be somewhat fixed in place. I've often expected companies like Netflix to implement drag and drop to maintain lists, but they never seem to choose that path.

In any case, the [HTML5 drag-and-drop spec](#) smoothes away many of the browser differences. In theory, the cross-browser scripts won't be necessary as long as all browsers follow the standard in exactly the same way. All that you need to do is add the attribute `draggable='true'` and the element can be picked up and moved.

Well, that's not quite all. If you want to do something with the dragged element, you must be able to handle at least seven different events that fire as it moves around the page. Struggling to deal with all possible options has driven some people to write long complaints about the complexity. (A "[disaster](#)" and "[far from complete](#)" are two early gripes.)

There are also some compatibility issues. Safari, for instance, requires a separate CSS entry to turn on dragging even after you add the `draggable='true'` attribute. All of these issues point to the fact that someone is going to write a simpler drag-and-drop library that abstracts away much of this complexity and makes it as easy as adding the `draggable='true'` attribute.

SELF-CALCULATING FORM FIELDS

One of the traditional jobs of JavaScript has been to perform calculations for the user who is adding data to the form. The traditional way was to set up some text input elements and let the JavaScript change the other elements whenever an *onchange* event fires.

The new idea is to create a new output element that will work in concert with the *input* element. An attribute specifies the formula for the *output* field. The browser is responsible for updating the output field whenever the form changes by calculating the formula. I have tried to use this on several browsers without success. It just seems easier to use good old input fields instead.


The output can also be represented graphically using the progress and meter tags. Both essentially represent some fraction between zero and one as a thermometer-like rectangle that fills up with color -- but there are differences. The progress element has an "indeterminate" setting that indicates the software has no clue what the value really is. This is usually displayed as wavy lines.

The *meter* tag is a bit more interesting because it includes the opportunity to specify a *low* and *high* attribute, as well as a *min* and *max*. Presumably a value between *min* and *low* or *high* and *max* is undesirable and the user should try to push whatever buttons are necessary to shift this value between *low* and *high*.



HTML5 FORMS AND EDITABLE CONTENT

All of the work that's been done on the forms is quite nice, but the irony is that the form tag itself is sort of passe. The greatest change in the form tag may be the fact that it's no longer necessary. Now most HTML elements can be edited by simply adding the attribute *contenteditable="true"* to any old div or span. It's pretty freaky. If the user doesn't like what you write on your blog, you can give them the opportunity to rewrite it to fit their preconceived notions. In essence, any table or pile of data could be turned into a form just waiting for the user to click and change. Everything can be wikified.

This section of the API is changing a bit. Just recently, [the *getSelection* method was moved](#), changing the best way to capture any of the editing process. Is *getSelection* ideal? Nope. Is it dangerous? Perhaps in the wrong hands. Will it be confusing to old users who still think that they can only monkey with data in the form boxes? Certainly. Will it encourage more traffic when people save entire pages just to store one tweak? No doubt. But editable content opens up more possibilities than ever. I'm sure that creative Web designers will find clever ways to make everything a form that comes alive. 

Geolocation and other tidbits

New tools for handling location, background tasks, browsing histories, and document layout

 By Peter Wayner

ONE OF THE SLY GAMES that smart managers play is attaching their current project to a big, high-profile tar ball rolling down the hill, full of momentum. Now that HTML5 has become white hot after languishing for 10 years of relative disinterest, many ideas that began as cool enhancements for the Web are latching on to the bandwagon. They may be relatively independent projects, but because they involve JavaScript and HTML, they're now part of the HTML5 juggernaut.

Consider the new JavaScript functions that let your Web page determine latitude and longitude if the code is running on a device that knows where it is. These are small enhancements that would have happened whether or not HTML5 became a buzzword that managers everywhere now feel compelled to add to their deck of slides. Yet now they're often considered in the same checklist full of HTML5 features.

As we put together this series of articles about HTML5, a number of little ideas and features ended up with no place to go. They didn't naturally fit with the other articles that focused on areas such as HTML5 forms, data communications, local data storage, or the games people play with the HTML5 Canvas, video, and graphics specs. Some of these tidbits are officially part of the HTML5 standard and some are just fellow travelers, but they're all appearing in your browser soon. We had no better place to put them, so they're here.

GEOLOCATION

The HTML5 spec doesn't officially offer the JavaScript layer the ability to find the device's location on earth, but HTML5 and geolocation have been gathering momentum at the same time, so people tend to lump them together. The geolocation tags are technically in [a separate API](#), if you're curious.

The features are pretty basic and originally intended for smartphones and PDAs, but they're also implemented in many desktop browsers, including Chrome 5.0 and

later, Firefox 3.5 and later, and [Internet Explorer 9](#). But just because the JavaScript object *navigator.geolocation* is offered doesn't mean the answer returned will be available or correct. The API thankfully allows for users to cloak their location, and many implementations ask the user for permission before revealing the geolocation object to the JavaScript code. The API returns an error of the type *PERMISSION_DENIED* to make it clear that the user didn't want to cooperate.

A nice feature includes a rough estimate of the accuracy along with the coordinates. There's a separate error range for the altitude because many GPS tools are less accurate in their estimates of the altitude than they are of the position on earth. The error estimates for the latitude and longitude, however, are the same, perhaps a mistake near the poles.

There are two major functions: *getCurrentPosition* and *watchPosition*. The first finds the position and the second wraps a loop around the process, generating an event only if there's a change. While *getCurrentPosition* is pretty straightforward, I'm still wondering why *watchPosition* doesn't have some parameter that defines just how much motion should trigger an event. There's just one parameter, the boolean *enableHighAccuracy*, that "may or may not make a difference, depending on your hardware." What is high accuracy? That's a good question. So you get to implement your own loop wrapped around the loop in *watchPosition*.

Much of this detail is pretty hypothetical for desktop and laptop users, even when they're using a browser that has the capability. Although desktop browsers offer the geolocation object, the computers often don't provide any mechanism for generating even a guess at the location. There are tools that can list the Wi-Fi routers in range and use the information to identify the latitude and longitude with surprising accuracy, but these haven't caught on beyond tablets and smartphones. When the device has no way of locating itself, the API generates a *POSITION_UNAVAILABLE*. To see if your browser supports the geolocation capabilities, try my test page at <http://www.wayner.org/node/78>.



WEB WORKERS

Who doesn't want an army of little servants working behind the scenes to make everything wonderful? The HTML5 team borrowed the imagery of this idle fantasy when they rediscovered background threads and called them "Web Workers." Now all of the hacks we've written using the JavaScript wait, delay, and pause commands to poll distant websites or animate sprites on the page can be retired and sent off to the ranch with those crazy symbols from APL.

John Resig, the genius behind the jQuery library, calls Web Workers "the coolest new feature." Although the UI specialists who work with native apps (and have known about threads for a long time) will roll their eyes at this attempt to bring JavaScript up to date with application programming circa 1990, the feature is quite novel and certainly more ornate than the average background thread. The [Web Workers spec](#) is part of the HTML5 document.

The biggest difference from threads is the way that the Web Workers objects operate in their own sandbox; they can't do anything directly to the Web page or the DOM describing it. The Web Workers must pass everything through messages, and both the Worker object and the DOM-level JavaScript must set up code to send and receive messages across the gap. This model will be familiar to everyone who's pulled out hair while trying to work around the client/server model of the Web -- think of Web Workers as little servers that sit alongside your client.

All of the major browsers except Internet Explorer (including IE9) support Web Workers today.

IFRAMES

In almost all cases, progress on the Web is marked by bold strides forward with new features that enable wonderful code to be written in fewer and cleaner lines. One exception is the [<iframe> tag in HTML5](#), a rare case of a tag that's losing some features. Don't worry, though, or start to feel claustrophobic. The functionality is merely being moved to make the integration of the iframe more seamless.

In the past, the Web designer could add scroll bars, borders, and margins to the content embedded in the iframe. Now all of that work has to be done by the HTML of the iframe itself. The Web designer coding the iframe won't have these options any longer.

That designer does get a few new options. The "seamless" attribute removes any of the borders and scroll bars,

rendering the iframe like a `<div>` tag that acquires its information from another source.

The other option will comfort those who worry about the security of their Web pages. The "sandbox" attribute turns off many of the more dangerous features sometimes given to content inside the iframe. The main page's author needs to explicitly enable them by adding attributes such as "allow-scripts" or "allow-forms."

These new iframe attributes are useful features that will make it easier for Web page designers to collaborate with other sites, yet not worry as much about dangerous behavior. Advertisers who create more interactive campaigns will love these options because they let websites adopt the ads while providing enough security to block wayward behavior. Websites won't need to trust the ad companies as they do now.

PARSING

No specification is ever complete because no one can even begin to imagine all of the ways that someone will use it. Over the years, the browser programmers have been surprised again and again by the ways that HTML writers found new and unexpected ways to use the tags. In the most glaring cases, the HTML creators unearthed spots where the browser developers made different assumptions. The HTML5 specification tries to spell out these places and smooth over the differences.

For example, WebKit browsers used to allow `<script>` tags that closed themselves with a final slash, `</>`. Anyone who included an outside file with such a tag would find that the code worked in the WebKit world but not in the other browsers. There are a bazillion examples like this that have appeared and disappeared in different versions of the browsers.

The [HTML5 Parsing spec](#) includes dozens of steps that the browsers should use to determine the encoding delivered by the distant Web server. There are also a surprisingly large number of suggestions for how to do the right thing when working through the tags in a `<table>`. I'm thankful for this because I've pulled out my hair in the past when one browser (that will remain nameless) wouldn't work unless I inserted a proper `<tbody>` layer. Yech.

There are hundreds of different ways that the new rules will unify the browsers, almost all of them small but occasionally maddening. It would be difficult to list or even test them all. One of the more notable changes is

in how the MathXML and SVG files can now be embedded inline like this:

```
<math>
  <mi>x</mi>
  <mo>=</mo>
  <mfrac>
    <mrow>
      <mo>&minus;</mo>
      <mi>b</mi>
      <mo>&PlusMinus;</mo>
      <msqrt>
        <msup>
          <mi>b</mi><mn>2</mn>
        </msup>
      <mo>&minus;</mo>
      <mn>4</mn>
      <mo>&InvisibleTimes;</mo>
      <mi>a</mi>
      <mo>&InvisibleTimes;</mo>
      <mi>c</mi>
    </mrow>
  </mfrac>
  <mi>a</mi>
  <mo>&InvisibleTimes;</mo>
  <mi>c</mi>
</math>
```

In other words, MathML and SVG are now pretty much part of regular HTML, except on older browsers, all of which will have to be explicitly supported for some time.

A number of these enhancements rise above the truly minor. Some of the so-called [text-level semantic enhancements](#) are like the [microformats](#) designed for the standard data elements floating around in text. For example, the `<abbr>` tag will mark all TLAs (three-letter acronyms) and allow you to embed the full definitions in case anyone is curious.

HTML5 HISTORY API

Who wouldn't want to rewrite history? The new [HTML5 History object](#) provides a limited number of

ways to meddle with the browser's history. You can't take a broadsword and change the entries for different sites, but you can add new pages and rewrite the entries from the current site.

For instance, if you happen to click through n pages of a Top 1,000 list on a site and generate 1,000 page views for the ad sales department, you'll end up with 1,000 different entries in your history. If you want to go back to the page before the slideshow, you'll need to slog through 1,000 entries. JavaScript manipulation of the history can save you from this trouble.

```
<script type="text/javascript">
function changeURL(url, title)
{
  if (typeof history.pushState == "undefined")
    { alert("Browser doesn't allow changing
      history with pushState"); }
  else { var state = {address : url}; window.history.
    pushState(state.address, title, url);
  }
}</script>
```

The main effect is to make navigation a bit more fluid and open to experimentation. The draft of the spec encourages developers to think about "nonlinear" solutions, while advising them to use their newfound power with an eye toward avoiding the confusion it might create. Rewriting history could be vexing to users and even dangerous if exercised in the wrong way, but it could also help clean up much of the ugliness where the History object is too literal.

UNDO HISTORY

Many of these new features change the browser from an app that displays a distant file in a rectangle to one that allows the user to interact and change objects inside a rectangle. When humans change data, they often make mistakes and want to undo their changes. This is where the [undo transaction history](#) and the *UndoManager* object come in.

A major source of the events that end up in the undo transaction history list will be changes to the input boxes of forms. Giving the JavaScript programmer access to the *UndoManager* may make it possible to customize this interaction and produce more sophisticated forms.

LAYOUT ENHANCEMENTS

One of the biggest new developments in HTML5, at least in terms of raw features, is the large collection of tags added to mark different sections of the document. Although the original HTML offered tags to mark the beginning and end of significant parts of a document's structure, there weren't many of them beyond the header (<h1> through <h6>) and the paragraph (<p>) tags. The rest were largely devoted to typographic signals such as bold () and italics (<i>).

Many of the new layout tags recognize what Web developers have been building on their own. There are now tags like <header> and <footer> that tell the browser to put which information at the top and bottom of the pages. Some of these will add confusion. The <section> tag, for instance, operates similarly to the <div> tag, so there will be some who use <div> where others use <section> and vice versa.

Many of the basic elements are supported across all major browsers, but the same can't be said for a number of the elements that seem less obviously useful. At this moment, the <figure> tag for attaching a movable figure to a section of text will work in Firefox 4.0 but not in Safari. The <ruby> tag used to annotate Asian symbols sort of works with Safari but not with Firefox.

Saying that a tag is "supported" here is not as straightforward as with other HTML5 features -- the semantics require more specification. It's one thing to write down requirements to store data, but it's another to specify just what a browser should do when laying out blocks of text. Even after the browsers start recognizing these tags, the browsers will probably choose to display the information

inside the tags in slightly different ways.

Many of the ideas that ended up in this collection seem tiny or inconsequential, but underestimating them would be a mistake. Although they are often just patches or fixes to ideas that date from the beginning of the Web, they open the door for many of the newest, savviest Web crawlers to extract more information from the pages.

While the first effect of these rules will be to improve the display and layout, they could also unlock deeper features by making it easier for computers to understand exactly what is going on. The newer tags do a better job of indicating the role of the text in the document, and this may aid artificial intelligence in making better sense of the text between the tags.

Smoothing the developer's job and doing a better job of segregating the DOM-level manipulation from the background tasks will open up another possibility: Automatic crawlers will be able to make some sense of the JavaScript running. While there are deep theoretical barriers, better parsing and a strict separation between the display-level work and the real processing may help crawlers suss out what the JavaScript is trying to do. This is a long way off, but it's one tiny method where the HTML5 may bring us closer to smart search engines and some of the predictions for artificial intelligence from the far-flung realms of science fiction.

Peter Wayner (<http://www.wayner.org>) is contributing editor to InfoWorld and author of 15 books on programming, databases, open source software, and other computing topics.