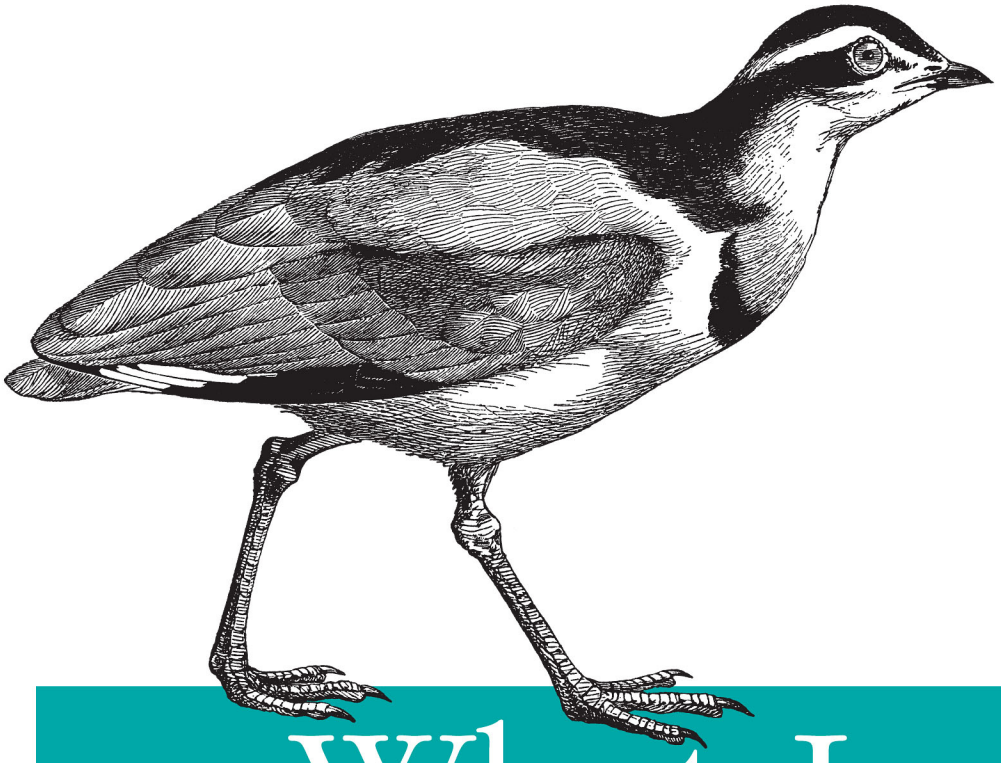
A New Way to Look at the Web



What Is HTML5?

O'REILLY®

Brett McLaughlin

What Is HTML5?

Brett McLaughlin

O'REILLY®
Beijing • Cambridge • Farnham • Köln • Sebastopol • Tokyo

What Is HTML5?
by Brett McLaughlin

Printing History:

ISBN: 978-1-449-31035-6
1310507581

Table of Contents

What Is HTML5?

Once you really understand HTML5, you'll change the way you think about the web.

HTML5: Everyone's using it, nobody knows what it is. I realize that sounds more like a line out of an existential movie — maybe *Waiting for Godot* or a screenplay by Sartre — than a statement about HTML5. But it's really the truth: most of the people using HTML5 are treating it as HTML4+, or even worse, HTML4 (and some stuff they don't use). The result? A real delay in the paradigm shift that HTML5 is almost certain to bring. It's certainly not time to look away, because by the time you look back, you may have missed something really important: a subtle but important transition centered around HTML5.

In this post, I want to take a deeper look at HTML5. I have a simple proposition with a lot of complex consequences: HTML5 is both something entirely new, and yet noth-

ing more than HTML was ever intended to be; and that once you really understand HTML5, you'll change the way you code and even think about the web and your own web applications.

A return to first principles

HTML has always been about interconnection. Back in the ancient days, when electronica was cool and *not* called "house music" and before the Rolling Stones qualified for Medicare, the web was littered with big huge documents. In fact, it was exactly the opposite of today, where most people think enhanced digital books are just electronic wrappers around full-text copies of what's in print.

In the '90s, the web was full of 15-page specifications, all in a single file. You scrolled through those massive documents just like you paged through an encyclopedia. Much of the early versions of HTML were intended to deal with this, widely considered a detriment to the readability and usability of the web. That's largely because Tim Berners-Lee, the recognized father of HTML, was a researcher enabling other researchers (mostly at CERN at the time). If you've ever known anyone mired in research, brevity is rarely their prevailing trait, so reading huge documents online was a necessity, but scrolling through 15- (or 1,500-) page documents just wasn't a long-term option.

So early on, HTML was not primarily about displaying those documents with lots of formatting. Most fundamental to HTML was the simple `a` tag. It gave a document the means to link to another document. Suddenly 15-page documents were reduced (mercifully) to 15 one-page documents, all linked together. Bye bye scrolling; hello useful linking. This is all pretty standard fare, and if much of this is new to you, then the waters are going to get deep quickly.

HTML5: Still connecting things

Fast forward to the present. Ultimately, this ability to connect things on the Internet is still the primary feature of HTML5. It's just that now, we're starting to realize the original vision of HTML, and connect a lot more than hypertext with static images. So the introduction of the `audio` and `video` elements in HTML5 are nothing more than logical extensions of the old `a` element.

(Note: in a more correct sense, `audio` and `video` replace `object` and all the embed code that people have been throwing into web pages for years, largely pulled from sites like YouTube or Vimeo. Still, those elements are semantically functioning more like an a element that drops the link into a page, rather than taking you off to another destination. In that sense, even the `img` element is in some ways nothing more than an inline `a` element: it grabs content from another location and pulls that content into a page. It's all just linking, and that's what HTML is really about: linking and connecting things.)

So now you can pull in images, audio, and video directly into a document. More importantly, because those items now have first-order elements, you can easily manipulate the audio and video from JavaScript. That's a big deal, and something I'll come back to later. In a nutshell, a first-order element is always going to encourage more direct programmatic access than one in which you have to be sneaky or clever to really get at it.

So while the `audio` and `video` elements are new, their purpose isn't. HTML5 allows you to integrate more assets into a single document, all while keeping the integrity and separation of those assets into place. This is nothing more than making sure that the bibliography of a document isn't physically stuck at the end of a long web page, but is in fact separate and easily maintained, but still able to be integrated into the rest of the page.

Now, before moving on, you need to immediately see that it's not (relatively) important that you can grab audio and video and drop it into an HTML page. What is important is that you can more easily grab "other stuff" and drop it into a page. There may eventually be 20 or 25 elements for things well beyond "audio" and "video," and the fundamental premise is still the same: the important thing here is that multiple pieces in multiple places can all be wired together in a meaningful way, with semantic elements describing that "stuff" easily accessible by JavaScript. The fact that, in HTML5, that "other stuff" happens to be audio and video is cool, but rather incidental.

HTML5 connections are the new rich media

So what, then? Why is this such a big deal? Well, largely for three reasons:

1. **Web pages no longer need to look (and act) like web pages.** The rise of Flash over the last years has largely been an attempt to overcome "limitations" of what HTML allows. Flash was initially often focused on animations and cool visual effects. But then entire sites got rolled into Flash, allowing for different types of navigation and page organization, richer programmatic access to the individual pieces of a web page, and the ability to avoid the quirkiness of JavaScript. (I'll leave out the obligatory comment here on the quirkiness of the Flash stack.)
2. **Web pages no longer need to represent one person/organization's content.** Even though web programmers and designers have been pulling in images from other sites for years, web pages are still largely homogenous in terms of the asset ownership. A web page today really has one person's content, images, pages, media, and the like. Even sites like Vimeo and YouTube are more often used as extensions of a private repository than an actual free medium for world access.
3. **Web pages can function intelligently and easily across display devices.** It's no secret that MOBILE (as one co-worker recently email-shouted at me) is the banner under which HTML5 most often flies. But the story really isn't that HTML5 has great mobile support; rather, it's that mobile is no longer a problem child. In

other words, the story is that what works on a desktop browser pretty much works on a phone. (The list of things not covered by "pretty much" is shrinking every few weeks, so better to not list them here and appear outdated next month.) Put another way, phones and tablets are first-class citizens, because they are privy to the same interconnections listed above. In fact, it's probably not too meta-physical to say that in addition to inter-connecting content, HTML5 has a really good shot at interconnecting all the devices floating around ... and that's arguably at least as big a deal as what it does for content and web applications.

HTML5 introduces — although it's not at all complete in its support for — a paradigm that moves away from both of these limitations. First, HTML5 and CSS3 provide for JavaScript a pretty solid working set of tools and effects, comparable to most Flash websites. I can generally take a medium-sized Flash website and recreate it in WordPress, HTML5, JavaScript (via jQuery), and CSS3 in a week, if not less. And the upside is enormous: text is again selectable, bookmarking works without lots of weird tricks, and of course website owners can actually update their own sites, rather than relying on some overly busy Flash programmer to help.

The result is that HTML5 is again the most usable and indexable tool available for web content; but that content is richer than ever before. Further, that content no longer has to be owned to the degree that older web pages had to be owned. For a lot of emotional and psychological reasons, mainly, the `audio` and `video` elements suggest pulling assets from other sources more than the more generic `object` element does. There's something about seeing `video` that screams, "Grab some video and stick it in your page!" But since there will always be at least an order of magnitude greater number of web page creators than video makers, how do you get that video? You grab it from someone else, hopefully someone who's used a nice Creative Commons license. The same is true for `audio`, of course: you can connect to someone else's audio incredibly easily, and so you should. And as already intimated, the audio and video that plays on your desktop browser also plays pretty well on an HTML5-enabled phone or tablet.

Underneath these limitation-stripping observations is something much bigger: content creators are moving from creating completely original content to amalgamating and assembling content. Sure, this isn't anything technically new, but it is something that's happening more due to the introduction of HTML5 elements like `audio` and `video`. And when the web starts to grow in its interconnectedness, it takes a giant step toward the world of Neal Stephenson and "Snow Crash" and "Neuromancer," doesn't it? One person creates a video and doesn't have to place it or build out a web page. They just throw it on a content-sharing medium like YouTube or Vimeo. Another person goes beyond a simple off-page link but actually integrates that video (fully accessible and mutable via JavaScript, something we'll get to shortly) into a web page. Neither person has to be an expert outside of their own domain (video and web page, respectively), but the result is a fusion of assets.

Now expand the working set of that fusion by a factor of 1,000. Or 1,000,000. Throw in data science and the ability to find, organize, and even localize data like never before.

Access that data not just from one browser, but all the major browsers, on all the major devices — laptops, netbooks, tablets, and phones. Toss in unbelievably cheap data storage, and then realize that we've yet to mention the increased power of JavaScript, the `canvas` tag and its ability to further manipulate the referenced audio and video, and web messaging, and you'll quickly forget that *we're talking about HTML here, not a traditional high-end programming language like Java or PHP!*

JavaScript isn't the focus of HTML5 ... right?

That greater ability of HTML5 pages to do a lot more brings JavaScript into focus. Honestly, there's not much in HTML5 that specifically speaks to JavaScript. Yes, the HTML5 draft, according to the W3C, is intended to replace the JavaScript APIs detailed in older HTML4, XML1, and DOM Level 2 documents. But no, don't expect the JavaScript language to be radically, or even that noticeably, changed in HTML5. Yet hang around the web programming water cooler, and you'll hear more about JavaScript than ever before. So what gives?

Well, first of all, HTML5 adds several important new first-order elements, as already mentioned. `audio` and `video` are key, as is another element we'll get to, `canvas`. That means that it no longer takes a lot of `document.getElementsByTagName("object")` or ID-tagging of `object` elements to work with media in a web page. Extending that, these elements (unlike the `object` element) have media-specific attributes like `autoplay` and `preload`. This means that with JavaScript, you can grab a video, display its controls, change the URL and effectively load a new video, and even control when the video loads (or preloads) and mute and unmute the audio.

All of this is done via JavaScript, but there's nothing new *in* JavaScript enabling this access. It's the addition of semantically meaningful elements in HTML5 that makes JavaScript more powerful, or at the least an easier medium in which to accomplish these tasks. Sure, a clever JavaScript programmer (who was comfortable tokenizing and parsing) could get most of this from grabbing particular attributes from an `object` element, adjust them, reset them, and so on. But my goodness, it was a mess, it was player-specific, and mostly made for annoying late nights trying to get the semicolon in just the right place.

In fact, much of what's great, albeit seemingly boring, about HTML5 is the move of important elements and attributes from customized or one-off items to semantically important ones. Even the `draggable` attribute — allowing native drag-and-drop in HTML5 — is cooler because it's now accessible and easily mutated via JavaScript. A page requires less textual data to mean something, and more of that meaning is pushed into the structure of the page itself.

I think it's fair to go as far as being axiomatic here: the more a document has semantic meaning, and the less attributes or even elements are nested as plain text within hidden `input` elements or `object` embeds, the better that document will be. The page becomes

easier to access, update, and make responsive from a JavaScript point of view; CSS (whether it's CSS2 or CSS3) can be applied directly without so many messy pseudo- and nested selectors; and the document describes itself. And when you have documents that have semantic meaning, you're going to find that your JavaScript is clearer. In fact, many mid-level programmers will realize that they can do things that the "advanced" programmers can: throw a video into the page based on a click, autoplay that video, mute the audio and load a different piece of audio, play that different audio over the video, and boom: you've got an all-HTML5 discussion by Hitler about JJ Abrams and Star Trek ... without having to modify the original audio and video assets.

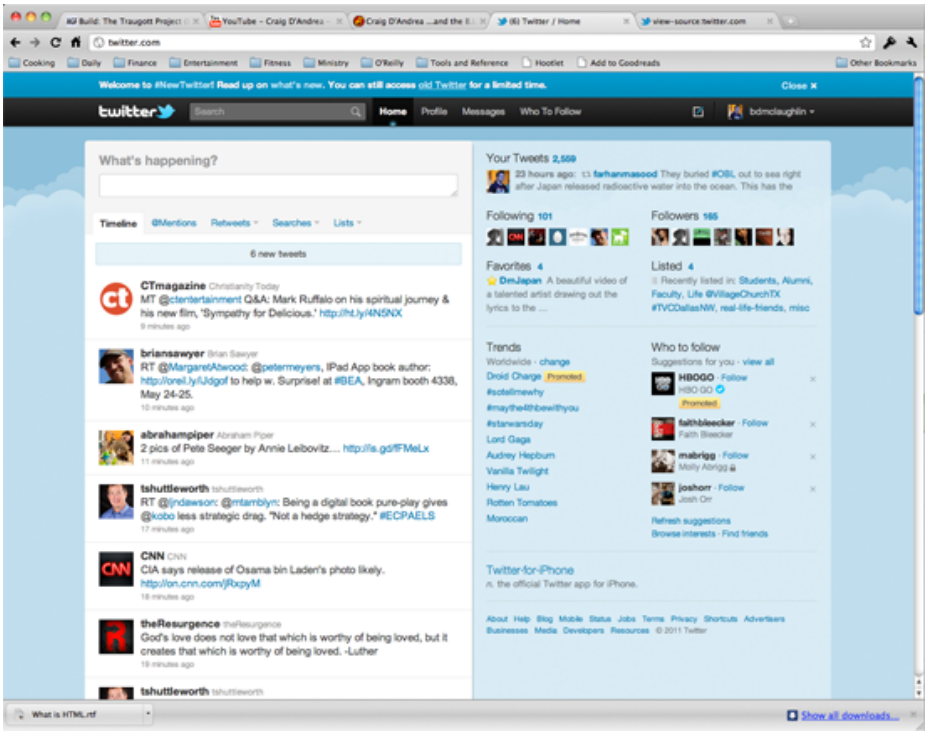
Put another way, the HTML page becomes more of a container (which contains other containers, which contain other containers ... and so on) than a page. JavaScript can easily access those nested containers, and modify them, update them, move them around, and do anything else you can dream up and code. CSS can also visually style and work with those containers, further making the HTML a purely organizational tool, and even that becomes loose. With absolute positioning, the order and position of elements within the HTML organization becomes at best a guide, if not totally irrelevant.

(It's worth saying a display approach that completely ignores the organization and sequence of elements within an HTML document is a pretty bad idea. What you gain with new semantically valuable elements is quickly lost when you start absolutely positioning everything, and removing any sense of hierarchy. Of course, that's true in HTML4 and XHTML1 as well as HTML5, so it's not something I'll focus on beyond this rather quick note.)

Container-based web pages: A step (sort of) in the right direction

The idea of an HTML page being little more than an organization of malleable elements is nothing new, but HTML5 seems to be a clear jumping-in point for JavaScript programmers and web developers that haven't yet made that conceptual leap. It's notable that with the rise of CMSes like WordPress for smaller businesses and personal websites, almost everyone is now familiar with the separation of concerns involved when even the text of an HTML page isn't necessarily encoded and stored within a static HTML file. This is all a good thing, too. As a JavaScript programmer myself, I'd rather have more programmatic control and create a page that is built based on user decisions, database-driven logic, and programming over one person with an HTML or text editor.

In fact, the most obvious example of this approach is the Twitter website. For example, when I visit Twitter, I see a ton of content:



But if you take a closer look, by viewing the source, you'll find hundreds of lines of JavaScript at the beginning of the file, a ton more JavaScript at the end of the file, and this rather pedestrian bit of HTML squeezed into the middle. Here's about 1/3 of that HTML, which is pretty brief:

```
<![CDATA[
  <body class="user-style-twttr  loading-body ">
    <div id="doc">
      <div id="top-stuff">
        <div id="banners" style="clear:both;"></div>
        <div id="top-bar-outer">
          <div id="top-bar-bg"></div>
          <div id="top-bar">
            <div class="top-bar-inside">
              <div class="static-links">
  <div id="logo">
    <a href="/">Twitter</a>
  </div>
  <form id="search-form" action="/search" method="GET">
    <span class="glass"><em></em></span>
    <input value="" placeholder="Search" name="q" id="search-query" type="text" />
  </form>
  <div id="global-nav">
    <ul>
      <li id="global-nav-home"><a href="/">Home</a></li>
```

```
            <li id="global-nav-profile"><a href="/bdmclaughlin">Profile</a></li>
            <li id="global-nav-messages"><a href="/messages">Messages</a></li>
            <li id="global-nav-whotofollow"><a href="/#!/who_to_follow">
                Who To Follow</a></li>
        </ul>
    </div>
    <div id="sections"></div>]]>
```

Not a lot to it, is there? Of course, even parts of this have been inserted programmatically, like links to my personal profile and messages. And even though this is an HTML4 site, it points to some important things with which HTML5 is attempting to deal.

Now, what I want to argue is that while this is a generally good thing — using JavaScript to connect the organization of HTML to content not encoded directly into that HTML — the step forward of new HTML5 elements is matched, if not overtaken, by some steps back in terms of this sort of programming, at least as it's currently implemented. It's not exaggeration to say that many pages, like Twitter's page, are basically a bunch of empty `div`s with `id` attributes and not much else. It's unclear whether Twitter's page will evolve into a more HTML5-centric approach, but unless it also evolves its structure, there are some problems.

Here's the thing: this approach is programmer-rich, yet semantically poor. First, elements aren't truly indicating content; they're just indicating buckets that can be filled. And don't mistake a textual value attached to a `div`'s `id` attribute as semantic meaning. That's no different than the `object` element when used to display audio or video; you've got to parse and tokenize to get any real sense of meaning. That's meaning, but it's not meaning encoded into the HTML in a particularly useful way.

This is another area that HTML5 seeks to improve. For many, the introduction of another few elements — `nav` and `header` and `footer`, for example — have been largely ignored. Why not just keep using `<div id="nav">`? Well, now the answer should be obvious: these new elements provide additional semantic meaning. If I'm linking to your page, or even pulling in part of your page, in my own page, I can grab (or not grab) your footer, your navigation, your figures and figure captions.

Don't let the SGML-ness of semantic meaning escape you. (Yes, I realize that SGML-ness isn't a word, but it should be.) When you really begin to connect heterogeneous parts of the web together, semantic meaning becomes huge. So does a page that is more than a bunch of `div`s with `id` attributes. You'll want to know what you're getting, and an HTML standard is a heck of a lot better way to determine meaning than arbitrary text values tucked away into `id` attributes.

By now, you're either seeing a theme in what I'm calling out about HTML5, or you're seeing little more than a grab bag of loosely connected features. If you're in the latter camp, let me make it explicit: HTML5, when used both as the 21st century web suggests and as the original HTML specification allowed, is best at interconnecting things. If you view your pages as a collection of content, and let go of the rather egotistical idea that all that content has to *be* your own, then all of the new features of HTML5 discussed

so far are hugely important. You can pull in audio and video and manipulate that audio and video as if it were your own. You can organize your page semantically and link to and even pull in content from other sites, using the semantics of that page. You can separate content from organization using more than just `div`s, but actual first-order elements for navigation and headers and footers.

And then, when things were going so well, there was `canvas`.

The canvas element is a programmable div

It's odd that the element that is probably at the top of the HTML5 spec in terms of "wow factor" is in fact the element that brings in the ability to undo most of the good I think HTML5 does. That element, of course, is `canvas`. `canvas` defines, well, a canvas on the HTML page. Using JavaScript, you can draw on that canvas using methods that look a lot like a graphics scripting language. JavaScript basically grabs the `canvas` by its `id` attribute, using `document.getElementById()`, and then gets a *context*. Most of the examples online that are really jaw-dropping use the 3D canvas, but there's a 2D canvas that's a lot better entry-point if you're new to this sort of thing.

Once you've got a context, you can operate upon that object, using properties like `fillStyle`, and methods like `fillRect(top, left, width, height);`, and `canvas`. Yes, it's really that simple, and even pretty intuitive. In fact, more than anything else, drawing on a canvas for the first time with JavaScript felt a lot like my old Logo days in grade school, or even Java AWT days in early Java. (Oddly, I'm prouder of what I did in Logo than AWT; don't judge.)

I don't think it's worth a lot of ink to walk you through using canvases. There are tons of web tutorials out there in both 2D and 3D, and a slew of good video and publications. Suffice it to say that the days of needing Flash to really do anything visually interesting, or at least needing to take that visually-interesting thing you created and turn it into an image or a video, are gone. What's even cooler is that you again get JavaScript integration into your page. So now a keypress or a form submission, or dragging an element on the page, or the playing of a video (because we have `draggable` and `video` now, remember?) can interact with the canvases on your page.

Now, there are two caveats: one small one and one giant one. The small one is that browsers are still getting their HTML5 act together, and the 3D context in particular is pushing browsers beyond what they were often intended to do. So realize that while you're not going to get the dreaded "Flash content won't play" sort of message on your iPhone, there is going to be a lot of rigorous testing required to get your 3D drawing just right on the most computers in the most browsers. But, honestly, that's a testing issue, and one that both time (as people upgrade) and testing can overcome.

The more serious issue is that the `canvas` becomes much like an anonymous, programmatically-filled `div`. The things within it have no semantic meaning and existing apart from the HTML's encoding on disk; it's a dynamic set of programmatically-created

lines and squares and spheres. No matter how cool of a texture you wrap around a 3D model, you can't go in with a separate JavaScript function (or better, a separate HTML page) and grab that textured sphere, link to it, spin it around its axis, or adjust the light source. You can do *all* of those things from the page itself, mind you; you just can't do it in the loosely-interconnected manner that is heralding the new and cool web that HTML5 helps drive.

So is this a stump speech against `canvas`? Of course not. I'd no more avoid `canvas` than I'd tell someone in HTML4 to avoid `div`. Yes, there is a limit to semantic meaning. Yes, the `canvas` becomes a big blob that contains something, but that something has minimal outside visibility. But it's still a step forward, and heralds what is hopefully a wealth of GPU studs and studettes making HTML5 do far more than anyone thought possible. (Studettes: another made up word, but these are *good* words. Webster, anyone? Urban Dictionary?)

Hopefully, there will soon be standardized ways to call into objects that are created within a canvas, and granularity becomes part of the advantages of a `canvas` rather than a drawback. Until then, you've got to decide if using a `canvas` makes sense. In general, if you're doing something page-specific, or if your `canvas` is itself a self-contained "mini-application," then go forth and prosper. If your `canvas` is tightly integrated into the rest of your page, you're probably in great shape, too. But if you want a lot of other pages and sites to pick up your `canvas` as any sort of component, you're probably out of luck, and might want to look at a different approach.

The good/bad news here is of course, that HTML allows for this sort of reuse, and people are starting to figure that out, but people are *just* starting to figure this out. Interconnections again; are you building your own little kingdom in the web, complete with moat and drawbridge to keep everyone else out? Or are you integrating everything you do into a larger-than-you Internet that shares (and sometimes steals) as much as it creates, and in that sharing and stealing recreates?

Mobile: Killer application, ho-hum client

For those of you already read up on HTML5, you may be surprised how little the word "mobile" has shown up. Sure, I mentioned earlier that mobile devices can view HTML5 web pages, and consume audio and video from `audio` and `video` without many problems. But isn't this the big story with HTML5? In fact, isn't it mobile devices, more than anything else, that are driving HTML5 adoption?

Well, yes and no.

As far as mobile devices driving adoption, absolutely: HTML5 owes a lot of its buzz to mobile devices, and in particular Steve Jobs and his near-militant campaign against Flash. The iPhone came with Safari, Safari had HTML5 support, so suddenly iPhone users were giddy and talkative. Now almost all modern smartphones have an HTML5-capable browser. So while the ability to view video without Flash is big, the yet-bigger

story is simply that anything that works on an HTML5 website as viewed within a desktop browser pretty much works on a phone. So yes, HTML5 wouldn't be the talk of the developer town if not for mobile.

But no, mobile can't be the big story with HTML5. (People will claim it is, and do so all the time, but they're misguided in that they're calling an effect a cause.) HTML5 doesn't offer all sorts of bells and whistles for working with mobile devices. Yes, there are ways to detect the orientation and screen size of a mobile device; but those also allow for detection of the size of a browser window, or a tablet display. And yes, certainly, the `canvas` element and new audio and video features make it possible to deliver amazing content to a mobile device. And yes, yet again, more and more folks are building mobile-optimized websites for phones. So how is this not a big deal?

Well, what's important is not that HTML5 works on phones. What's important is that HTML5 "just works." There aren't tons of issues when getting your site to work on a phone ... and *that's* the story. The big deal isn't that you can do all kinds of cool things with your website for mobile phones. The big deal is that you can do all kinds of cool things, period; and that those cool things just happen to work on mobile phones ... and tablets ... and netbooks ... and desktop clients. HTML5 really removes the need to think about mobile devices separately from other devices.

(Note: I'm not saying you should ignore mobile devices, or not think about them separately. The best sites I use on my phone are almost all at least mobile-optimized, if not complete with a pure-mobile theme or CSS styling. That said, you don't have to do this to get up and running. Also, almost all good mobile-themed sites have a "View normal site" that works perfectly well. That says a lot. It used to be you'd want to hide the normal site at almost all costs. Not with HTML5...)

... and a partridge and a pear tree

As this article turns the bend from readable to monolithic, at least in article terms, it's easy to look at what else HTML5 offers, and consider writing an entire book. There's web messaging, and offline applications, and local storage ... really, everything you need to write an entire application. And ultimately, that's the brief but accurate assessment I'd make of the "everything else" of HTML5. HTML5 is no longer about a single page, or even a collection of inter-linked pages. It's about interconnection — and I've tried to beat that drum consistently above all else — but that interconnectedness is about a lot more than pages. It's certainly about a lot more than just *your* pages.

Web messaging allows interconnecting HTML5 applications (I'll defend the use of "application" here in a moment; for now, just go with me). Those applications don't have to be in the same domain; for the first time, there's an intelligent JavaScript-based messaging framework that allows for communication across domains. You can write listeners to receive messages and senders to send them. In its simplest form, all the limitations of Ajax sandboxing that everyone griped about have been removed, albeit

with some security considerations and a lot more complex API than the pretty simple XmlHttpRequest object. But all the same, if you buy into the premise that HTML5 encourages sharing of resources across the web, then messaging makes that sharing more of a true two-way communication. Now you can build resources that can be consumed, even if that consumption involves receiving data as well as sending it.

And you've got local storage, too: a legitimate database-like storage mechanism for holding onto data and delivering it to a server on the Internet when that server is available — even if that's not the "now" of the page's existence and use by a real person. These tie in closely with forms, the most obvious means of getting this sort of data that local storage is optimized to store. In fact, while it's nice that HTML5 forms offer some new controls and validation and the like, it's their ability to interact intelligently with offline storage (via, again, JavaScript) that's the really big story. Think about that: for the first time in, well, ever, there's valuable offline access that can take place. Again, there are kinks (and some of them are big) and some new wrinkles to take into your JavaScript, but the web has always had kinks and wrinkles. It's just that now you're getting more than ever for dealing with those issues.

Add to that even heavier-duty types of functionality, like threads (which are called web workers, but really are just threads) and sockets. On the one hand, these aren't part of the core HTML5 spec. Instead, they've been moved into WHATWG, so that says something about their ultimate inclusion and importance as part of HTML5 to the W3C community. On the other hand, by untying these from the core spec, they may now be free to evolve further and more quickly than ever before. In any case, it's no small thing to be talking about threads and sockets not as part of code that generates or produces HTML, but as part of an HTML application itself. These are serious tools, and further indicate that HTML5 is not to be taken as a language for building interesting and largely static web pages anymore.

So about this word *application*: isn't that what's ultimately being created with HTML5? This isn't just about neat visuals and clever drag-and-drop controls and even replacing Flash-based sites. HTML5 pushes a good programmer to create semantically-meaningful and accessible resources. Those resources don't have to be entire pages. They can just be "things:" video and audio and navigation and footers. Those "things" can not only be used across domains by other "things," but there's actually communication that can happen. You can pull in a very well-designed footer by sending it a few choice bits of information, and have that footer pulled from one domain interact with a video pulled from another domain. When the user is offline, big deal. Local storage not only allows for database-like storage of an HTML page *without an installed database*, but is even kind enough to make sure that data gets to the right place where connectivity is restored. JavaScript is not a toy language anymore, and jQuery makes what was annoying in JavaScript lexically a piece of cake.

These are applications.

Stop building web pages. Please. Stop trying to only use "your" content. Slap a Creative Commons license not just on your video, or your images, but on your HTML, and then build interesting components and assume they'll be sucked into other things and used in ways you'd never considered. It is called the Internet, remember?

The winners in an HTML5 world are those who stop fearing being stolen from, and actually start handing out their candy to every kid on the block. It's about a lot more than YouTube-like sharing of videos. Applications are increasingly becoming a collection of mini-applications, and if you're thinking about your HTML5 work as collecting resources — some original, some shared — then you're way ahead of the curve.

Related:

- Can Flash and HTML5 get along?
- Why HTML5 is worth your time
- New directions in web architecture. Again.
- Where the semantic web stumbled, linked data will succeed