SMASHING MAGAZINE

HTML Semantics

Imprint

Version 1: July 2012

ISBN: 978-3-943075-36-6

Cover Design: Ricardo Gimenes

PR & Press: Stephan Poppe

eBook Strategy: Thomas Burkert

Technical Editing: Thomas Burkert

Idea & Concept: Smashing Media GmbH

ABOUT SMASHING MAGAZINE

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

ABOUT SMASHING MEDIA GMBH

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

About this eBook

Probably you are aware that one way to reinforce the meaning of website information is the use of HTML semantics. This eBook "HTML Semantics" addresses various topics such as outlining algorithms, the pursuit of semantic value and the semantic grid system.

Table of Contents

HTML5 Semantics

When One Word Is More Meaningful Than A Thousand

HTML5 And The Document Outlining Algorithm

Our Pointless Pursuit Of Semantic Value

Pursuing Semantic Value

The Semantic Grid System: Page Layout For Tomorrow

About The Authors

HTML5 Semantics

By Bruce Lawson

Much of the excitement we've seen so far about HTML5 has been for the new APIs: local storage, application cache, Web workers, 2-D drawing and the like. But let's not overlook that HTML5 brings us 30 new elements to mark up documents and applications, boosting the total number of elements available to us to over 100.

Sexy yet hollow demos aside, even the most JavaScript-astic Web 2.0-alicious application will likely have textual content that needs to be marked up sensibly, so let's look at some of the new elements to make sure that your next project is as semantic as it is interactive.

To keep this article from turning into a book, we won't look at each in depth. Instead, this is a taster menu: you can see what's available, and there are links that I've vetted for when you want to learn more.

Along the way, we'll see that HTML5 semantics are carefully designed to extend the current capabilities of HTML, while always enabling users of older browsers to access the content. We'll also see that semantic markup is not "nice to have," but is rather a cornerstone of Web development, because it is what enhances accessibility, search-ability, internationalization and interoperability.

A human language like English, with its vocabulary of a million words, can't express every nuance of thought unambiguously, so with only 100 or so words that we can use in HTML, there will be situations when it's not clear-cut which element to use for which piece of content. In that case, choose one; be consistent across the site.

Some Presentational Elements Are Gone

Purely presentational elements such as **`center`**, **`font`** and **`big`** are now obsolete. Their role has been perfectly usurped by Cascading Style Sheets. Now, this doesn't mean you have to rush out and recode all of those ancient pages; HTML5 makes them obsolete for authors, but because HTML5 strives not to break the Web, browsers will still render those cobwebbed legacy pages.

For the same reason, presentational attributes have been removed from current elements; for example, **`align`** on **`img`**, **`table`**, **`background`** on **`body`**, and **`bgcolor`** on **`table`**.

The evil **`frame`** element is absent in HTML5. Frames caused usability and accessibility nasties. If you get the urge to use them, use an older **`DOCTYPE`** so that your pages still validate.

Beyond this short overview, see the W3C's exhaustive list of removed elements and attributes.

Some Presentational Elements Have Been Redefined To Be Semantic

Not all presentational elements have been taken out and shot. Some have undergone an extensive re-education program and emerged with shiny new semantics. For example, the **`small`** element no longer means "use a small font," although it will display that way in browser style sheets. Now it represents side comments, such as small print:

> *Small print typically features disclaimers, caveats, legal restrictions, or copyrights. Small print is also sometimes used for attribution, or for satisfying licensing requirements.*

Some of the redefinitions feel to me to be a mop-up. While I can get behind **`<b>`** for drawing attention to product names, keywords and so forth, without any special emphasis implied, specifying the semantics for marking up ship names (**`<i>`**, if you're so inclined) feels weirdly precise. But I get seasick, and your nautical mileage may vary. With similar niche precision:

> *The `u` element [now] represents a span of text with an unarticulated, though explicitly rendered, non-textual annotation, such as labeling the text as being a proper name in Chinese text (a Chinese proper name mark), or labeling the text as being misspelt.*

You can read more about changed elements and attributes on the W3C website.

Sexy New Semantics

We all know about **`video`** and **`audio`**. And **`canvas`** is particularly popular at the moment because it allows for 3-D graphics using webGL, so game designers can port their products to the Web. Like good ol' **`img`**, these semantics are embedded content, because they drag in content from another source — either a file, a data URI or JavaScript.

Unlike **`img`**, however, they have opening and closing tags, allowing for fallbacks. Therefore, browsers that don't support the new semantics can be fed some content: an image could be the fallback for a canvas, for example, or a Flash movie could be the fallback for **`video`**, a technique called "video for everybody."

The **`source`** and **`track`** elements are empty elements (with no closing tags) that are children of **`video`** or **`audio`**. The **`source`** element gets past the codec Tower of Babel that we have. Each element points to a different source file (WebM, MP4, Ogg Theora), and the browser will play the first one it knows how to deal with:

```
<audio controls>
  <source src=bieber.ogg type=audio/ogg>
  <source src=bieber.mp3 type=audio/mp3>
    <!-- fallback content: -->
    Download <a href=bieber.ogg>Ogg</a> or <a
href=bieber.mp3>MP3</a> formats.
</audio>
```

In this example, Opera, Firefox and Chrome will download the Ogg version of Master Bieber's latest toe-tappin' masterpiece, while Safari and IE will grab the MP3 version. Chrome can play both Ogg and MP3, but browsers will download the first source file that they understand. The fallback content between the opening and closing tags is a link to download the content to the desktop and play it via a separate media player, and it is only shown in browsers that can't play native multimedia. For video, you could use an embedded Flash movie hosted on YouTube:

```
<video controls>
  <source src=best-video-ever.webm type=video/webm>
  <source src=best-video-ever.mp4 type=video/mp4>
    <!-- fallback content: -->
    <iframe width="480" height="360"
      src="http://www.youtube.com/embed/xzMUyqmaqcw?rel=0"
      frameborder="0" allowfullscreen>
    </iframe>
</video>
```

This way, users of older browsers, such as IE 6-8, will see a YouTube movie (as long as they have the Flash Player), so they will at least be able to see the video, while users with modern browsers will get the full native-video experience. Everyone gets the content, then, which is what your website is there for, after all.

The **`track`** element is a newer addition to the HTML5 family and is being implemented by Opera, Chrome and IE at the moment. It points to a subtitle file that contains text and timing information. When implemented, it synchronizes captions with the media file to enable on-demand subtitling and captioning; useful not only for viewers who are hard of hearing, but also for those who do not speak the language used in the audio or video file.

Semantics For Internationalization

Less woo! than the semantics for multimedia and games are the semantics for internationalization. It may surprise the cool kids in Silicon Valley to learn that a worldwide Web of people use languages other than English and even use different writing systems.

Languages such as Arabic and Hebrew are written right to left, unlike European languages, which are written left to right. On pages that use only one writing system, this doesn't present a problem, but on pages with bi-directional ("bidi") writing, browsers have to decide where to put punctuation, bullets, numbers and the like. Browsers usually do a pretty good job using the Unicode bidirectional algorithm, but it gets it wrong in some cases, which can seriously dent the comprehensibility of content.

HTML5 gives us a **`bdi`** element, which enables authors to override the Unicode bidirectional algorithm and make their text more comprehensible. For a further description of the problem and to see how **`bdi`** solves it, see

"HTML5's New `bdi` Element" by Richard Ishida, the W3C's internationalization activity lead.

Some languages have scripts that are not alphabetic at all, but that express an idea rather than a sound. Occasionally, an author will have to assist readers with pronunciation for especially rare or awkward characters, usually by providing an alternate script in a small font above the relevant character. In print, this was traditionally done with a very small 5-point font called "ruby," and HTML5 gives us three new elements for marking up ruby text: **`ruby`**, **`rt`** and **`rp`**.

For more information, see "The HTML5 `ruby` Element in Words of One Syllable or Less" by Daniel Davis.

Structural Semantics

Most people are aware that HTML5 gives us many new elements to describe parts of a Web page, such as **`header`**, **`footer`**, **`nav`**, **`section`**, **`article`**, **`aside`** and so on. These exist because we Web developers actually wanted such semantics. How did the authors of the HTML5 specification know this? Because in 2005 Google analyzed 1 billion pages to see what authors were using as class names on **`div`**s and other elements. More recently, in 2008, Opera MAMA analyzed 3 million URLs to see the top class names and top IDs used in the wild. These analyses revealed that authors wanted to mark up these areas of the page but had no elements to do so, other than the humble and generic **`div`**, to which they then added descriptive classes and IDs.

The new semantics were built to degrade gracefully. For example, consider what the specification has to say about the new **`figure`** element:

> *The `figure` element represents some flow content, optionally with a caption, that is self-contained and is typically referenced as a single unit from the main flow of the document.*
>
> *The element can thus be used to annotate illustrations, diagrams, photos, code listings, etc…*

This isn't a new idea. HTML3 proposed a `fig` element (which never made it into the final HTML 3.2 specification). It looked like this:

```
<FIG SRC="nicodamus.jpeg">
   <CAPTION>Ground dweller: <I>Nicodamus bicolor</I> builds
silk snares</CAPTION>
   <P>A small hairy spider.
   <CREDIT>J. A. L. Cooke/OSF</CREDIT></P>
</FIG>
```

There's a big problem with this. In browsers that do not support **`fig`** (and none do), the image wouldn't be displayed because the **`fig`** element would be completely ignored. The contents of the **`credit`** element would be displayed, because it's just text. So you'd get a credit with no image on older browsers.

In HTML5, you would code the same example like so:

```
<figure>
<img src="nicodamus.jpeg">
   <figcaption>
      <p>Ground dweller: <i>Nicodamus bicolor</i> builds silk
snares.</p>
      <p>A small hairy spider.
      <small>J. A. L. Cooke/OSF</small&gt</p>
   </figcaption>
</figure>
```

Unlike the aborted HTML3 syntax, the HTML5 version is backwards-compatible: a browser that doesn't "know" about the **`figure`** element will still show the `img` and the text inside **`figcaption`** (as the HTML3 **`credit`** element would similarly display its content). Note that we're using the redefined **`small`** element, instead of minting a new **`credit`** element. Remember that "Small print is also sometimes used for attribution."

HTML5 also gives us a new **`figcaption`** element. Originally, the specification's authors tried to reuse **`caption`**, as suggested in HTML3, but there were legacy problems, because **`caption`** had previously only been a child of **`table`**.

One of the design principles on which HTML5 is based is that new features should degrade gracefully. When they can't, the language allows for fallback content. It tries to reuse elements rather than mint new ones — but it's a pragmatic language: when minting something new is necessary, it does so.

Interactive Semantics

The structural elements of HTML5 currently don't do much in visual browsers, although software that sits on top of browsers (such as screen readers) are starting to use them (see "HTML5, ARIA Roles, and Screen Readers in March 2011" and "JAWS, IE and Headings in HTML5.")

Other elements do have a visual effect. The `details` element, for example, is a groovy interactive element that functions as "a disclosure widget from which the user can obtain additional information or controls."

Most browsers will implement it as an "expando box": when the user clicks on some browser-generated icon (such as a triangle or downwards-pointing arrow) or the word "Details" (which can be replaced by the author's own rubric in a child **`summary`**), the element will slide open, revealing its details within. The details could be a full description of an image or graph, a description of a complex table, advanced options for a search form, or just about anything else. This is a common need on the Web today, now made native and obviating the need for custom JavaScript.

Most of us have seen HTML5's new form semantics. Most of these are attributes of the **`input`** element, thereby ensuring graceful degradation to **`<input type=text>`** in older browsers. New elements include **`datalist`**, **`output`**, **`progress`** and **`meter`**.

Do We Have The Right Semantics?

So, we have many new semantics, but are they the right ones? After all, the Google research on which they were based was conducted in 2005 — quite some time ago! Perhaps the semantics are already somewhat behind the times? Many have noted that they're document-centric rather than application-centric. Do we need more application-centered semantics, such as a **`login`** or **`share`** element, or some kind of **`modal`** element for modal dialogue boxes?

I don't know; I'm not an app developer. But at least HTML is a "living standard," and so these can be added if strong enough use cases are presented to the Working Group.

I think most coders would welcome a new way to embed images that respond to the device's context. Borrowing from the **`video`** element, which displays source video according to what media queries instruct, I can imagine a new element such as **`picture`**:

```
<picture alt="angry pirate">
   <source src=hires.png media="min-width:800px">
   <source src=midres.png media="min-width:480px">
   <source src=lores.png>
      <!-- fallback for browsers without support -->
      <img src=midres.png alt="angry pirate">
</picture>
```

This would pull in **`hires.png`** for widescreen devices, **`midres.png`** for devices between 480 and 800 pixels wide, and **`lores.png`** for everything else, thereby rendering moot the question that designers currently ask themselves, “Do I make every browser download a high-resolution image and then squash it down for small screens, thus wasting bandwidth, or do I send a low-resolution image to every browser and scale it up for big screens, potentially sacrificing quality?”

Taking a leaf from the other popular semantics we’ve seen, there would be a fallback in the middle — in this case, a conventional **`img`** element — so everyone would get the right content.

Sending the right-sized image to devices without wasting bandwidth is one of the knottiest problems in cross-device and responsive design at the moment. Perhaps we’ll see a solution to this in HTML6. At the moment, the best solutions, which include Matt Wilcox’s Adaptive Images and Filament Group’s Responsive Images, require JavaScript and tweaks to the server’s `htaccess` file. The worst solutions require old-fashioned techniques, such as browser-sniffing, now rebranded as “device detection” but still the same old user-agent string-pattern matching, which is hilariously fragile, not future-proof or scalable, and straight out of the days of “Best viewed in Netscape Navigator at 800 × 600” badges on websites.

WHEN, WHERE, WHO?

A lot of data depends on three pieces of information: *when*, *where* and *who*?

HTML5 has a **`time`** element (which has been a bit of a battleground lately). This enables you to annotate a human-readable date with an unambiguous machine-readable one. It doesn't matter what goes between the tags, because that's the content for people to read. So, you could use either of the following:

```
<time datetime="1982-07-18">The day the woman I love was
born</time>
<time datetime="1982-07-18">Priyanka Chopra's birthday</time>
```

Whichever you choose, the machine would still know the date you mean because of the **`datetime`** attribute, formatted as **`YYYY-MM-DD`**. If you wanted to add a time, you could: separate the time from the date with a **`T`**, and then put the time in 24-hour format, terminated by a **`Z`**, along with any time-zone offset. So, **`2011-11-13T20:00Z`** would be **8:00** pm on 13 November 2011 UTC, while **`2011-11-13T23:26.083Z-05.00`** would be 23:26 pm and 83 milliseconds in the time zone lying 5 hours before UTC. A Sri Lankan-localised browser could use this information to automatically convert dates into Buddhist calendar. Search engines could use timestamps to help evaluate "freshness".

It's perhaps surprising that, even though geolocation is so prevalent now, we don't have a location element that simply takes three attributes: latitude, longitude and (optionally) altitude. It would be great to be able to write the following:

```
<location lat=51.502064 long=-0.131981>London SW1A 4WW</
location>
```

The browser would then offer to show you a map or give you directions from the current GPS location or any other location-based service.

(Since I gave the talk that this article is based on, Ian Hickson, the HTML5 editor, said that he expects to add a new <geo> element. If I could choose, I'd prefer `place`, so I could wear a T-shirt with the slogan "I've got the `time` if you've got the `place`".)

HTML3 had a `person` element, "used for names of people to allow these to be extracted automatically by indexing programs," but it was never implemented. In HTML4, the `cite` element could be used to wrap names of people, but this has been removed in HTML5 — controversially (see "Incite a Riot" by Jeremy Keith). In HTML5, then, we're left with no way to unambiguously denote a person. People's names are, however, a hard problem to solve. Whereas times and dates have well-known standardized ISO formats (`YYYY-MM-DD` and `HH:MM:SS.mmm`, respectively), and location is always latitude, longitude and altitude, personal names are harder to break down into useful parts: there are Russian patronymics, Indonesian single-word names, multiple family names, and Thai nicknames to consider. (See Richard Ishida's excellent article "Personal Names Around the World" for more information and discussion.)

The new `data` element, which replaces `time`, has a value attribute that passes machine-readable information, but it has no required or implied format, so there is no way for a browser or search engine to know, for example, whether `1936-10-19` is a date, a part number or a postal code.

Microdata

HTML5, like HTML4, is extensible (but not in the oh-so-dirty eXtensibility way of XML formats, so loathed by the Working Group). You can use the tried and tested microformats, which use HTML classes, or the full RDFa specification, which doesn't validate in HTML4 or HTML5. Because RDFa was considered to be too hard for authors to write (Google has conducted research that finds that authors make 30% more mistakes with RDFa than with other formats), HTML5 specifies microdata, a mechanism for adding common semantics via agreed-upon markup patterns. HTML5 Doctor has more information on HTML5 microdata, and Opera 11.60 supports the Microdata DOM API.

Like microformats and RDFa, the extra semantics added to the markup make sense only if you have a cheat sheet that tells you what each piece means. This means that the data has to point to a vocabulary that tells any crawler how to interpret the lump of data it finds. For microdata, there is the newly established Schema.org, which is "a collection of schemas, i.e. HTML tags, that webmasters can use to mark up their pages in ways recognized by major search providers."

Do Semantics Matter Anyway?

Now that more and more markup is generated by JavaScript, some people are tempted to think that semantics don't matter. We see various products marketed as HTML5 which simply make **`divs`** fly around the screen with JavaScript — simple DHTML techniques unchanged from 10 years ago.

I've even seen some Web pages with no markup at all. Some frameworks emit skeletal HTML with empty **`body`** tags and inject all the HTML with script. If you're squirting some minified JavaScript down the wire, with no markup at all, you're closer to Flash than you are to the Web.

In the same way that 47 minutes is (apparently) too long to to struggle making a CSS layout, at which point you should just give up and use tables, some people suggest that thinking about which element to use is a waste of time. “There are two types of developers: those who argue about div’s not being semantic and those who create epic shit” writes Thomas Fuchs, as if the two activities were mutually exclusive.

A better argument is that no software cares about or consumes semantics anyway, so why bother? This isn’t true (work is underway already to map assistive technologies to new semantics), but even if it were true, it ignores that this is a chicken-and-egg argument. It assumes that no new search engine will ever come to the market and be able to use new elements, or that browsers will never release new versions that can make use of these semantics, and that developers will write no new extensions — in short, it assumes that the evolution of the Web is complete.

Semantics do matter. Semantics communicate meaning, and once that is established, machines can do something meaningful with that data, without having to develop and use algorithms to guess. A browser extension might allow a user to jump straight to the **`nav`** with a single keystroke. It can do this because it looks for **`nav`** rather than having to employ heuristics to find a **`div`** with an **`id`** or **`class`** that would suggest it’s being used as navigation (assuming the author decided to use something sensible like **`nav`**, navigation, sidebar, or menu — and a restaurant site with a **`div`** called “menu” might be a list of foods rather than other pages...ah, the ambiguity of natural language). A crawler might dynamically assemble articles on a timeline. There are many more possibilities than my meagre imagination can dream up.

The Web is based on simple technologies, mashed up together to bring surprising results — results which have certainly surpassed the inventors’ original intents or expectations. The Web will continue to do so. What makes the Web so great, so flexible and so powerful is the fact that content is in open formats that can be parsed and mashed up in new and surprising ways.

These can happen if the content is marked up for meaning by the author — and if the language has the right markup elements for authors to use as a vocabulary. HTML5 extends our vocabulary. We'll need more words — and those will come about with HTML6 etc.

If, like me, you believe the Web to be a system that works across browsers, across operating systems, across devices, across languages, that is View-sourcable, hackable, mash-uppable, accessible, indexable, reusable, then we need to ensure that we use the small number of semantic tools at our disposal properly, and we'll all benefit.

When One Word Is More Meaningful Than A Thousand

By Niels Matthijs

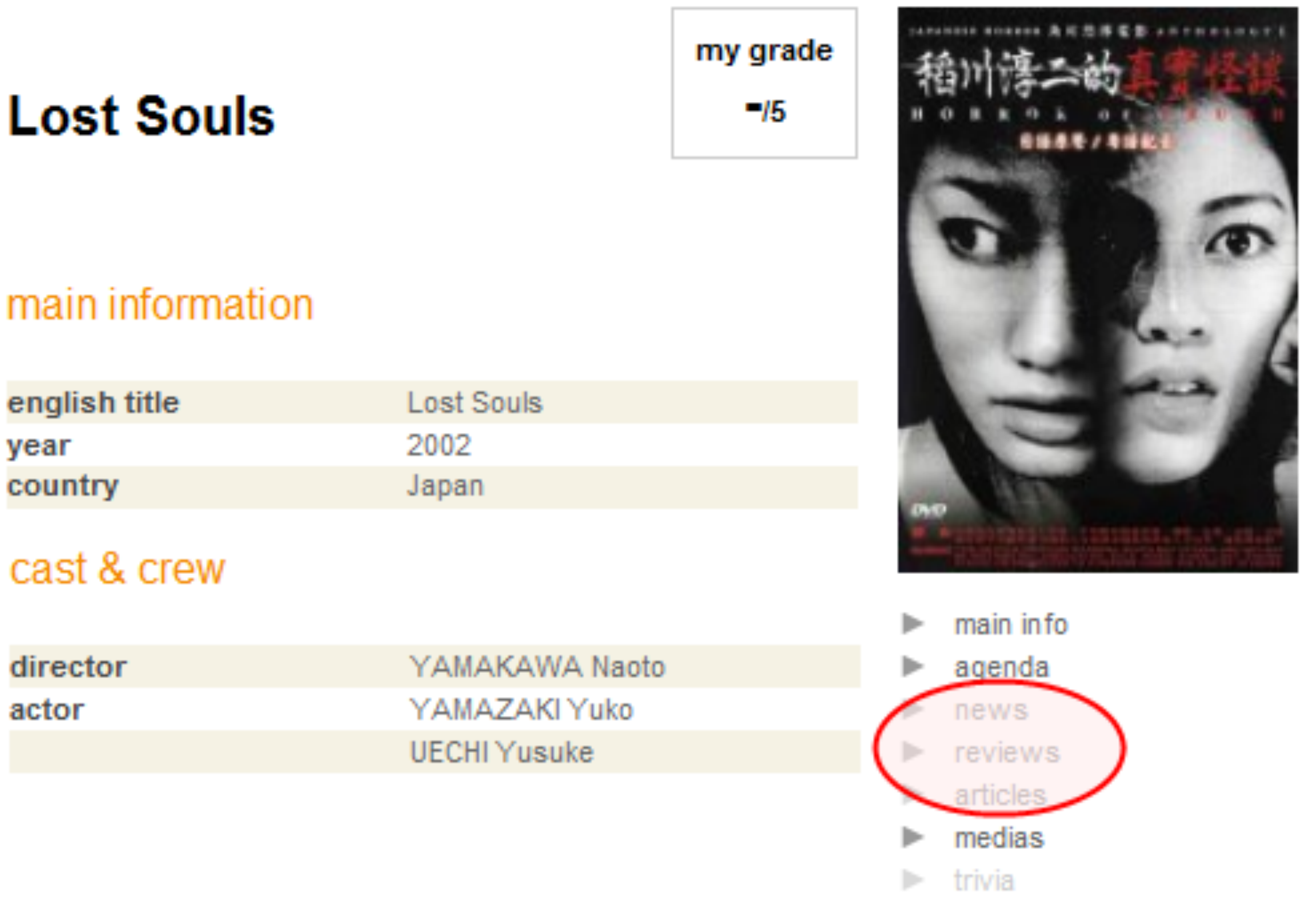
Why doesn't this article deal with HTML5 or another fancy new language: anything but plain, clear, tired old semantics. You may even find the subject boring, being a devoted front-end developer. You don't need a lecture on semantics. You've done a good job keeping up with the Web these last 10 years, and you know pretty much all there is to know.

I'm writing about HTML semantics because I've noticed that semantic values are often handled sloppily and are sometimes neglected, even today. A huge void remains in semantic consistency and clarity, begging to be filled. We need better and more consistent naming conventions and smarter ways to construct HTML templates, to give us more consistent, clearer and readable HTML code. If that doesn't sound like paradise, I don't know what does.

The Bare Necessities Of Semantics

With all the functional mumbo jumbo hidden away in HTML5, some of us seem to have forgotten what HTML is really all about. Native video support is considered way cooler than the new header tags, somewhat understandably, but from a semantic and structural point of view, these latter elements present the most valuable improvement.

Semantic importance got a serious boost when accessibility became a big deal to us Web developers. But its powers go way beyond making our content available to those lacking the skills to surf the Web in regular ways. For one, making content recognizable to all kinds of crawlers (but most importantly search engines) could greatly improve the results of search queries on the Web. Rather than wading through trailers, film websites and product pages, wouldn't it be much nicer to filter reviews directly and find out how a certain film has been received? Currently, no trustworthy mechanism exists to recognize or filter a broad range of content types, which is a serious loss for the Web as a whole.


Lost Souls

my grade
-/5

main information

english title	Lost Souls
year	2002
country	Japan

cast & crew

director	YAMAKAWA Naoto
actor	YAMAZAKI Yuko
	UECHI Yusuke

When looking for reviews, you don't want to end up on a page with grayed-out links.

If all of that sounds like a far-off dream, then note that once you've distinguished between all the elements on your website, you will have little to no trouble styling or adding functional behavior to the page. The combination of context and proper semantics ensures a solid structure for all further front-end work, which is only made stronger by making sure every element is defined correctly.

The (Very Simple) Basics

Absolutely nothing is complex about semantics, and the basics have been preached for a long time now. A recap of the bare minimum won't hurt anybody, though, so here it goes.

The HTML language has a range of tags with semantic meaning. If none of the available tags suits your needs, then two generic tags (**`span`** and **`div`**) are the HTML equivalents of the word "thing," which can be used in combination with one or more classes to add (not necessarily standardized) semantic value to your elements. It's the microformats idea without the actual microformats. Some basic examples:

- Main navigation: **`nav.main`** (HTML5) or **`div.navMain`**;
- An article: **`article`** (HTML5) or **`div.article`**;
- Article header: **`article>header`** (HTML5) or **`div.article>div.header`**

That's all there is to it, really. Adding semantic value is about choosing the correct tag(s) and/or applying the correct label(s) to an element. It really makes you wonder why applying this simple concept consistently to professionally developed websites has proven to be so difficult, even today.

For those of you who don't like the microformats ideology, you could also go all HTML5 and look at the HTML5 Microdata proposition. What follows in this article reflects both methodologies equally, so the choice is entirely up to you.

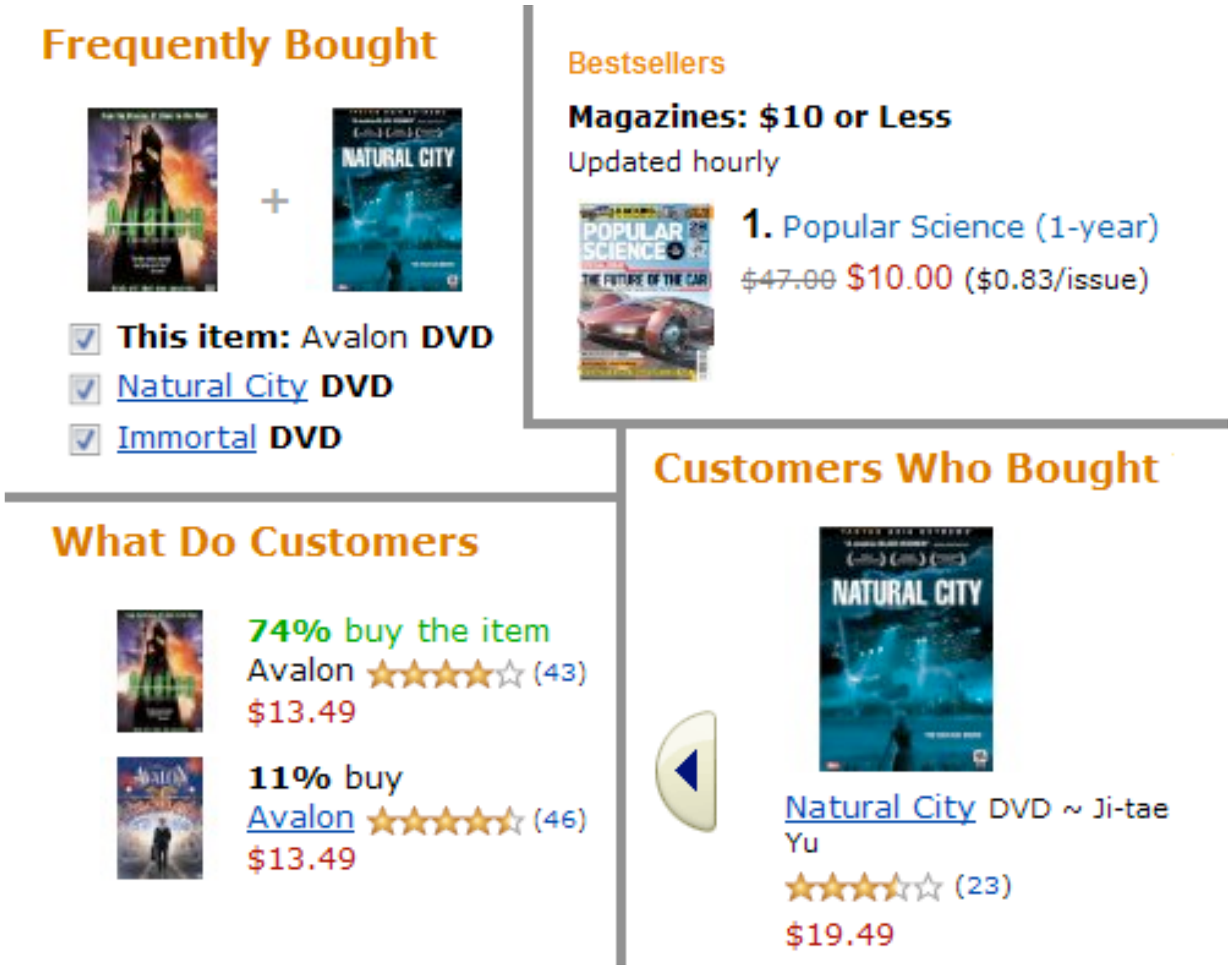
Sampling The Web

To illustrate my point, I took some quick samples from some of today's leading websites. By no means do these samples hold any scientific validity, nor is this a purposeful bash of the websites I've singled out. They are simply chosen because I believe they best represent their kind. I hope the data speaks for itself either way.

To grasp the semantic consistency within a website, I tried finding some common content types. Content types are easy to recognize and even easier to label. Before I get to the data, though, let's look at one way we could label products in a Web store:

- Product detail: **`div.product`**;
- Products added to your basket: **`.basket li.product`**;
- Promo product in a list: **`.categoryList .product.promo`**;
- Etc.

Products are everywhere in a Web store, so it seems logical that the product class would reappear across the pages for every instance of a product on the website. After all, whether a product is located in a "Related items" list, added to a basket or shown in full doesn't really change its semantic nature, so why change its structure or class name?



Frequently Bought

+

This item: Avalon DVD

Natural City DVD

Immortal DVD

Bestsellers

Magazines: $10 or Less

Updated hourly

1. Popular Science (1-year)

~~$47.00~~ $10.00 ($0.83/issue)

What Do Customers

74% buy the item
Avalon (43)
$13.49

11% buy
Avalon (46)
$13.49

Customers Who Bought

Natural City DVD ~ Ji-tae Yu
(23)
$19.49

These are all products, appearing as variants or in different contexts.

For my sample, I picked five content types (story, product, video, person, blog post) and picked four websites to represent each content type. To check for semantic consistency, I looked at the labels on a shortlist (a list of content type instances) and the content type's detail. The following table summarizes my findings:

Type	Website	Shortlist	Detail
Story	BBC	`div.hpData`	`table.storycontent`
Story	New York Times	`div.story`	`div#article`
Story	CNN	`ul.cnn_bulletbin li`	`div.cnn_storyarea`
Story	MSN	`li.ter`	`div.w649` (?)
Product	Amazon	`div.asinItem`	-
Product	Apple Store	`li.product`	`div.product-selection`
Product	Play.com	`div.info`	`div.dvd`
Product	YesAsia	`div.item`	`div#productpage`
Video	YouTube	`div.video-cell`	`div.video-info`
Video	Vimeo	`div.item`	`div.video_container_hd`
Video	Dailymotion	`div.video`	`div.dmco_box`
Video	eBaum's World	`div.mediaitem`	`div#videoContentContainer`
Person	Facebook	`div.UIFullListing`	`div.profile_top_wash` **and** `div.profile_bottom_wash`
Person	Last.fm	`div.user`	`div.user`
Person	Virb	`table.people td`	`div#profile_wrapper.artist`
Person	Twitter	`div#following_list span.vcard`	`div#profile`

Blog post	Zeldman	-	-
Blog post	A List Apart	`div.item`	- or `body.articles`
Blog post	Jens Meiert	`div.item`	`.content .col-1`
Blog post	Webaim	`div#features`	`div.section`

Apart from last.fm, none of the websites I checked got it right, even though the content types I chose were very easy to label. Apple and the New York Times came quite close, but some of the others are miles away from what you'd expect to find. And that's just looking at the root tag of the content type. The structure and classes within are often even worse, bordering on complete randomness. Another thing to note is that blogs about Web design seem to score the worst.

Think Components, Not Pages

There is, of course, not one single cause of this problem, nor is the solution simple. But you can make one important mental shift as a front-end developer to give your work more semantic consistency. The key is to stop thinking of a website as a collection of pages and to instead look for common components.

Front-end developers tend to work the same as designers: start with the home page, finish that, and then move on to the second wireframe — copy the reusable components, adapt if needed, and then repeat until all pages are done. This process requires a lot of copying, adapting and checking older pages to find reusable elements. It is a true killer of consistency — invoking spur-of-the-moment labels and destroying semantic consistency.

Because we want consistency, both in structure and semantics, focusing on a single component at a time is better. When you need to write the HTML code for a product, check each wireframe for variations within and across products. Write code that can handle all existing variants. Once that is done, you will have a consistent and solid model to describe your component that you can used wherever you want.

Making It All Happen

I know from experience that this mental shift takes some time to get used to, and the only way to get it working is to throw yourself in and practice. I'll share some quick pointers to make the whole process a little less daunting.

THINK BEYOND STYLING NEEDS AND PERFORMANCE

```
.productList li or .products li
ul li.product
```

Consider the example above. As Web developers, we've been taught that the first option should be preferred. From a performance and styling perspective, this is indeed the case. But putting on your semantic hat, you'll notice that to recognize the list items in the first example as products, you need to make a deduction. Singling out all products on a page isn't as easy as looking for the product class. Automated systems should also account for the possibility that a product is defined as a list item inside a parent that refers to a collection of products. Not such a big deal for the human brain, but writing a foolproof, fully automated implementation isn't as easy.

On top of that, the second option allows for more flexibility because it makes it possible to drop instances of other content types into the same list without running into styling hell, while at the same time ensuring semantic integrity. It wouldn't be the first time I was asked to merge a news and event shortlist into one big list just because there wasn't sufficient content to warrant separate lists. The second option would give you a smaller headache, especially if you're nearing an important deadline.

Bottom line: try to minimize semantic deductions, and keep the code clear and simple. Pick unique class names for components, and stick with them throughout the entire project.

DON'T MIX RESPONSIBILITIES

I know that many people like to mix wireframing, HTML and even design into one organic and homogeneous process. The downside to this is that you will have a hard time not compromising your work. When you're designing, writing HTML and CSS is not priority number one; and once the design is done, you'll find it tough to go back and rework your code to match HTML and CSS standards.

It's also refreshing to try to build a website based purely on a set of wireframes, without the slightest notion of design. It helps you focus on meaning and makes it easier to spot components that are actually the same but could differ wildly design-wise. And if you've done it right, you'll find that during CSS development, you don't have to adapt the HTML at all, unless the design calls for major structural changes.

Try to build your HTML templates based on wireframes, and save the design and CSS for when your static HTML templates are completed.

AUTOMATE YOUR JOB

Automation is a major key to success. Whether you use existing tools (such as a CMS) or build your own (as we do), automating the job of building static templates could help you to define a component once and reuse the code everywhere that the component is featured in your templates. The process itself (when done right) ensures semantic consistency and is sure to bring you new insight when constructing HTML templates.

At my current job, we build such a tool based on components (recurring HTML code blocks) and schemes (outlines of each template that refer to these components). Thrown in some simple program logic (**`if`** and **`loop`** statements, parameters) and allow for proper nesting methods, and you're good to go.

SEMANTIC CONSISTENCY ACROSS PROJECTS

Finally, keep a list of components you've made over multiple projects. Many components will be relevant for each new project and will be semantically identical, meaning that the HTML structure should be identical just as well (save some wrappers for visual CSS trickery, if you're into that).

Once you have such a list of components, starting up a new project will be a lot faster, and you'll have the added benefit of semantic consistency across all of your projects.

Banana ≠ Curvy Yellow Fruit

Semantics is all about identifying objects, but it goes beyond simply slapping a label on every object that comes your way. If you have a blog, and you randomly throw around classes like **`article`**, **`story`**, **`blogpost`** and **`news`**, then your website will lack semantic consistency, making all your hard work amount to very little. Semantics have no point when they are not applied consistently, even though today's technology does very little with them — which, by the way, is no surprise given that locating a simple "product" on most Web stores is nearly impossible these days.

People looking for bananas might think twice before buying these.

The next time you begin a project, try to view a Web page as a collection of building blocks. Start by constructing these building blocks first, and worry about building the pages later. Come up with a single label for an HTML component, and use it consistently across your website. It won't make styling harder, and it won't affect the way you write JavaScript. Over time, you can take it further by being semantically consistent over multiple projects.

If your main job is to develop static HTML templates, try to automate your work. You'll find that you spend more time writing flexible and solid HTML structures and less time copying and adapting code from point A to point B. It makes your job more interesting and makes the Web a better and more meaningful place.

HTML5 And The Document Outlining Algorithm

By Derek Johnson

By now, we all know that we should be using HTML5 to build websites. The discussion now is moving on to how to use HTML5 correctly. One important part of HTML5 that is still not widely understood is sectioning content: **`section`**, **`article`**, **`aside`** and **`nav`**. To understand sectioning content, we need to grasp the document outlining algorithm.

Understanding the document outlining algorithm can be a challenge, but the rewards are well worth it. No longer will you agonize over whether to use a **`section`** or **`div`** element—you will know straight away. Moreover, you will know *why* these elements are used, and this knowledge of semantics is the biggest benefit of learning how the algorithm works.

What Is The Document Outlining Algorithm?

The document outlining algorithm is a mechanism for producing outline summaries of Web pages based on how they are marked up. Every Web page has an outline, and checking it is easy using a really simple free online tool, which we'll cover shortly.

So, let's start with a sample outline. Imagine you have built a website for a horse breeder, and he wants a page to advertise horses that he is selling. The structure of the page might look something like this:

1. Horses for sale
    1. Mares
        1. Pink Diva
        2. Ring a Rosies
        3. Chelsea’s Fancy
    2. Stallions
        1. Korah’s Fury
        2. Sea Pioneer
        3. Brown Biscuit

Figure 1: How a page about horses for sale might be structured.

That’s all it is: a nice, clean, easy-to-follow list of headings, displayed in a hierarchy — much like a table of contents.

To make things even simpler, only two things in your mark-up affect the outline of a Web page:

- heading content (**`h1`** to **`h6`** and **`hgroup`**),
- sectioning content (**`section`**, **`article`**, **`aside`** and **`nav`**).

Obviously, the sectioning of content is the new HTML5 way to create outlines. But before we get into that, let’s go back to HTML 101 and review how we should all be using headings.

Creating Outlines With Heading Content

To create a structure for the horses page outlined in figure 1, we could use mark-up like the following:

```
<div>
   <h1>Horses for sale</h1>
   <h2>Mares</h2>

   <h3>Pink Diva</h3>
   <p>Pink Diva has given birth to three Grand National 
winners.</p>

   <h3>Ring a Rosies</h3>
   <p>Ring a Rosies has won the Derby three times.</p>

   <h3>Chelsea's Fancy</h3>
   <p>Chelsea's Fancy has given birth to three Gold Cup 
winners.</p>

   <h2>Stallions</h2>
   <h3>Korah's Fury</h3>
   <p>Korah's Fury has fathered three champion race horses.</
p>

   <h3>Sea Pioneer</h3>
   <p>Sea Pioneer has won The Oaks three times.</p>

   <h3>Brown Biscuit</h3>
   <p>Brown Biscuit has fathered nothing of any note.</p>

   <p>All our horses come with full paperwork and a family 
tree.</p>
</div>
```

Figure 2: Our "Horses for sale" page, marked up using headings.

It’s as simple as that. The outline in figure 1 is created by the levels of the headings.

Just so you know that I’m not making this up, you should copy and paste the code above into Geoffrey Sneddon’s excellent outlining tool. Click the big “Outline this” button, et voila!

An outline created with heading content this way is said to consist of implicit, or implied, sections. Each heading creates its own implicit section, and any subsequent heading of a lower level starts another layer, of implicit sub-section, within it.

An implicit section is ended by a heading of the same level or higher. In our example, the “Mares” section is ended by the beginning of the “Stallions” section, and each section that contains details of an individual horse is ended by the beginning of the next one.

Figure 3 below is an example of an implicit section that ends with a heading of the same level. And figure 4 is an implicit section that ends with a heading of a higher level.

```
<h3>Sea Pioneer</h3><!-- start of implicit section -->
<p>Sea Pioneer has won The Oaks three times.</p>

<h3>Brown Biscuit</h3><!-- This heading starts a new implicit
section, so the previous Sea Pioneer section is closed -->
```

Figure 3: An implicit section being closed by a heading of the same level

```
<h3>Chelsea’s Fancy</h3><!-- start of implicit section -->
<p>Chelsea’s Fancy has given birth to 3 Gold Cup winners.</p>

<h2>Stallions</h2><!-- this heading starts a new implicit
section using a higher level heading, so Chelsea’s Fancy is
now closed -->
```

Figure 4: An implicit section being closed by a heading of a higher level.

Creating Outlines With Sectioning Content

Now that we know how heading content works in creating an outline, let’s mark up our horses page using some new HTML5 structural elements:

```
<div>
   <h6>Horses for sale</h6>

   <section>
      <h1>Mares</h1>

      <article>
         <h1>Pink Diva</h1>
         <p>Pink Diva has given birth to three Grand National
winners.</p>
      </article>

      <article>
         <h5>Ring a Rosies</h5>
         <p>Ring a Rosies has won the Derby three times.</p>
      </article>

      <article>
         <h2>Chelsea’s Fancy</h2>
         <p>Chelsea’s Fancy has given birth to three Gold Cup
winners.</p>
```

```
      </article>
   </section>

   <section>
      <h6>Stallions</h6>

      <article>
         <h3>Korah’s Fury</h3>
         <p>Korah’s Fury has fathered three champion race 
horses.</p>
      </article>

      <article>
         <h3>Sea Pioneer</h3>
         <p>Sea Pioneer has won The Oaks three times.</p>
      </article>

      <article>
         <h1>Brown Biscuit</h1>
         <p>Brown Biscuit has fathered nothing of any note.</
p>
      </article>
   </section>

   <p>All our horses come with full paperwork and a family 
tree.</p>
</div>
```

Figure 5: The horses page, marked up with some new HTML5 structural elements.

Now, I know what you’re thinking, but I haven’t taken leave of my senses with these crazy headings. I am making a very important point, which is that the outline is created by the sectioning content, not the headings.

Go ahead and copy and paste that code into the outliner, and you will see that the heading levels have absolutely no effect on the outline where sectioning content is used.

The **`section`**, **`article`**, **`aside`** and **`nav`** elements are what create the outline, and this time the sections are called explicit sections.

One of the most talked about features of HTML5 is that multiple **`h1`** elements are allowed, and this is why. It's not an open invitation to mark up every heading on the page as **`h1`**; rather, it's an acknowledgement that where sectioning content is used, *it* creates the outline, and that each explicit section has its own heading structure.

The part of the HTML5 spec that deals with headings and sections makes this clear:

> *Sections may contain headings of any rank, but authors are strongly encouraged to either use only **`h1`** elements, or to use elements of the appropriate rank for the section's nesting level.*

I would strongly advise that until browsers — and, more critically, screen readers — understand that sectioning content introduces a sub-section, using multiple **`h1`** elements is less safe than using a heading structure that reflects the level of each heading in the document, as shown in figure 6 below.

This means that user agents that haven't implemented the outlining algorithm can use implicit sectioning, and those that have implemented it can effectively ignore the heading levels and use sectioning content to create the outline.

At the time of this writing, no browsers or screen readers have implemented the outlining algorithm, which is why we need third-party testing tools such as the outliner. The latest versions of Chrome and Firefox style **`h1`** elements in nested sections differently, but that is very different from actually implementing the algorithm.

When most user agents finally do support it, using an **`h1`** in every explicit section will be the preferred option. It will allow syndication tools to handle articles without needing to reformat any heading levels in the original content.

```
<div>
   <h1>Horses for sale</h1>

   <section>
      <h2>Mares</h2>

      <article>
         <h3>Pink Diva</h3>
         <p>Pink Diva has given birth to three Grand National 
winners.</p>
      </article>

      <article>
         <h3>Ring a Rosies</h3>
         <p>Ring a Rosies has won the Derby three times.</p>
      </article>

      <article>
         <h3>Chelsea's Fancy</h3>
         <p>Chelsea's Fancy has given birth to three Gold Cup 
winners.</p>
      </article>
   </section>
   <section>
      <h2>Stallions</h2>

      <article>
         <h3>Korah’s Fury</h3>
         <p>Korah’s Fury has fathered three champion race 
horses.</p>
      </article>

      <article>
         <h3>Sea Pioneer</h3>
         <p>Sea Pioneer has won The Oaks three times.</p>
      </article>

      <article>
         <h3>Brown Biscuit</h3>
         <p>Brown Biscuit has fathered nothing of any note.</
p>
      </article>
   </section>

   <p>All our horses come with full paperwork and a family 
tree.</p>
</div>
```

Figure 6: Our horses page, marked up sensibly.

One other point worth noting here is the position of the paragraph “All our horses come with full paperwork and a family tree.” In the example that used headings to create the outline (figure 2), this paragraph is part of the implicit section created by the “Brown Biscuit” heading. Human readers will clearly see that this text applies to the whole document, not just Brown Biscuit.

Sectioning content solves this problem quite easily, moving it back up to the top level, headed by “Horses for sale.”

Mixing It Up

So, what happens when implicit sections and explicit sections are combined? As long as you remember that implicit sections can go inside explicit sections, but not the other way round, you will be fine. For example, the following works well and is perfectly valid:

```
<h1>Horses for sale</h1>

<section>
   <h2>Mares</h2>

   <h3>Pink Diva</h3>
   <p>Pink Diva has given birth to three Grand National 
winners.</p>

   <h3>Ring a Rosies</h3>
   <p>Ring a Rosies has won the Derby three times.</p>

   <h3>Chelsea’s Fancy</h3>
   <p>Chelsea’s Fancy has given birth to three Gold Cup 
winners.</p>
</section>
```

And it creates a sensible hierarchical outline:

1. Horses for sale
   1. Mares
      1. Pink Diva
      2. Ring a Rosies
      3. Chelsea’s Fancy

Figure 7: Implicit sections created by headings inside an explicit section.

However, if you hope to achieve the same outline by nesting an explicit section inside an implicit section, it won’t work. The sectioning element will simply close the implicit section created by the heading and create a very different outline, as shown below:

This would produce the following outline:

1. Horses for sale
   1. Mares
   2. Pink Diva
   3. Ring a Rosies
   4. Chelsea’s Fancy

Figure 8: Explicit sections can’t go inside implicit sections.

There is no way to make the explicit sections created by the **`article`** elements become sub-sections of the Mare’s implicit section.

You can use headings to split up the content of sectioning elements, but not the other way round.

Things To Watch Out For

UNTITLED SECTIONS

Until now we haven't really looked at **`nav`** and **`aside`**, but they work exactly the same as **`section`** and **`article`**. If you have secondary content that is generally related to your website — say, horse-training tips and industry news — you would mark it up as an **`aside`**, which creates an explicit section in the document outline. Similarly, major navigation would be marked up as **`nav`**, again creating an explicit section.

There is no requirement to use headings for **`aside`** and **`nav`**, so they can appear in the outline as untitled sections. Go ahead and try the following code in the outliner:

```
<nav>
   <ul>
      <li><a href="/">home</a></li>
      <li><a href="/about.html">about us</a></li>
      <li><a href="/horses.html">horses for sale</a></li>
    </ul>
</nav>
<h1>Horses for sale</h1>
<section>
   <h2>Mares</h2>
</section>
<section>
   <h2>Stallions</h2>
</section>
```

Figure 9: An untitled <nav>.

The **nav** appears as an untitled section. Now, this generally wouldn't be a problem and is not considered bad HTML5 code, although in his recent HTML5 Doctor article on outlining, Mike Robinson recommends using headings for all sectioning content in order to increase accessibility.

Untitled **section** and **article** elements, on the other hand, are generally to be avoided. In fact, if you're unsure whether to use a **section** or **article**, a good rule of thumb is to see whether the content has a natural, logical heading. If it doesn't, then you will more than likely be wiser to use a good old **div**.

Now, the spec doesn't actually require **section** elements to have a title. It says:

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

Your interpretation of this probably hinges on your understanding of the word "typically." I take it to mean that you need a damn good reason not to use headings with **section** elements. I do not take it to mean that you can ignore it whenever you feel the urge to use a new HTML5 element.

Where the `article` element is specified, the spec goes even further by showing an example of blog comments marked up as untitled **articles**, so there are exceptions. However, if you see an untitled **section** or **article** in the outline, make sure you have a good reason for not giving it a title.

If you are unsure whether your untitled section is a **nav**, **aside**, **section** or **article**, a very handy Opera extension will let you know which type of sectioning content you have left untitled. The tool will also let you view the outline without leaving the page, which can be hugely beneficial when you're debugging sections.

SECTIONING ROOT

The eagle-eyed among you will have noticed that when I said that sectioning content cannot create a sub-section of an implicit section, there was an **`h1`** ("Horses for sale") not in sectioning content immediately followed by a **`section`** ("Mares"), and that the sectioning content did actually create a sub-section of the **`h1`**.

The reason for this is sectioning root. As the spec says, sectioning elements create sub-sections of their nearest ancestor sectioning root or sectioning content.

> *Sectioning content elements are always considered subsections of their nearest ancestor sectioning root or their nearest ancestor element of sectioning content, whichever is nearest, regardless of what implied sections other headings may have created.*

The **`body`** element is sectioning root. So, if you paste the code from figure 7 into the outliner, the **`h1`** would be the sectioning root heading, and the **`section`** element would be a sub-section of the **`body`** sectioning root.

The **`body`** element is not the only one that acts as sectioning root. There are five others:

1. **`blockquote`**
2. **`details`**
3. **`fieldset`**

4. **`figure`**

5. **`td`**

The status of these elements as sectioning root has two implications. First, each can have its own outline. Secondly, the outline of nested sectioning root does not appear in, nor does it have an effect on, the outline of its parent sectioning root.

In practice, this means that headings inside any of the five sectioning root elements listed above do not affect the outline of the document that they are a part of.

The final thing (you'll be glad to hear) that I'll say about sectioning root is that the first heading in the document that is not inside sectioning content is considered to be the document title.

Try the following code in the outliner to see what happens:

```
<section>
   <h1>this is an h1</h1>
</section>
<h6>this h6 comes first in the source</h6>
<h1>this h1 comes last in the source</h1>
```

Figure 10: How heading levels at the root level affect the outline.

I won't try to explain this to you because it will probably only confuse both of us, so I'll let you play with it in the outliner. Hint: try using different heading levels for the implicit sections to see how the outline is affected; for example, **`h3`** and **`h4`**, or two **`h5`**s.

UNTITLED DOCUMENTS

If no heading is at the root level of the document (i.e. not inside sectioning content), then the document itself will be untitled. This is a pretty serious problem, and it can occur either through carelessness or, paradoxically, by thinking carefully about how sectioning content should be used.

Roger Johansson addresses this issue in his excellent article on document outlines and HTML5 and the follow-up article.

Johansson asks how a proper document outline is supposed to be created for a blog post or other news-type item using HTML5. If you subscribe to the belief that your logo or website name should not be in an **`h1`** element, you could mark up your blog post along the lines of the following:

```
<body>
   <article>
      <h1>Blog post title</h1>
      <p>Blog post content</p>
   </article>
</body>
```

The document is untitled. Somewhat reluctantly, Johansson settles on marking up the website's title in **`h1`** and using another **`h1`** to mark up the article's title. This is a sensible solution and is backed up by the results of the WebAIM screenreader user survey, in which the majority of respondents stated a preference for two top-level headings in exactly this format.

This same approach is also widely used on static pages that are built with HTML5 structural elements, and it could be very useful indeed for screen reader users. Imagine that you are using a screen reader to find a decent recipe for chicken pie, and you have a handful of recipe websites open for comparison. Being able to quickly find out which website you are on using the shortcut key for headings would be much more useful than seeing only "chicken pie" on each one.

Not too far behind two top-level headings in the screen reader user survey was one top-level heading for the document. This is probably my preferred option in most cases; but as we have already seen, it creates an untitled body, which is undesirable.

In my opinion, there is an easy way around this problem: don't use **`article`** as a wrapper for single-blog posts, news items or static page main content. Remember that **`article`** is sectioning content: it creates a sub-section of the document. But in these cases, the document is the content, and the content is the document. Setting aside the name of the element, why would we want to create a sub-section of a document before it has even begun?

Remember, you can still use div!

HGROUP

This is the final item in the list of things to watch out for, and it's very easy to understand. The **`hgroup`** element can contain only headings (**`h1`** to **`h6`**), and its purpose is to remove all but the highest-level heading it contains from the outline.

It has been and continues to be the subject of controversy, and its inclusion in the specification is by no means a given. However, for now, it does exactly what it says on the tin: it groups headings into one, as far as the outlining algorithm is concerned.

In Conclusion

The logic behind the document outlining algorithm can be hard to grasp, and the spec can sometimes feel like physics: understandable as you're reading it, but when you try to confirm your understanding, it dissolves and you find yourself re-reading it again and again.

But if you remember the basics — that **`section`**, **`article`**, **`aside`** and **`nav`** create sub-sections on Web pages — then you are 90% of the way there. Get used to marking up content with sectioning elements and to checking your pages in the outliner, because the more you practice creating well-outlined documents, the sooner you will grasp the algorithm.

I promise, you will have it cracked after only a handful of times, and you will never look back. And from that moment on, every Web page you create will be structured, semantic, robust, well-outlined content.

Our Pointless Pursuit Of Semantic Value

By Divya Manian

Allow me to paint a picture:

1. You are busy creating a website.
2. You have a thought, "Oh, now I have to add an element."
3. Then another thought, "I feel so guilty adding a **`div`**. Div-itis is terrible, I hear."
4. Then, "I should use something else. The **`aside`** element might be appropriate."
5. Three searches and five articles later, you're fairly confident that **`aside`** is not semantically correct.
6. You decide on **`article`**, because at least it's not a **`div`**.
7. You've wasted 40 minutes, with no tangible benefit to show for it.

This Just Straight Up Sucks

This is not the first time this topic has been broached. In 2004, Andy Budd wrote on semantic purity versus semantic realism.

If your biggest problem with HTML5 is the distinction between an aside and a blockquote or the right way to mark up addresses, then you are not using HTML5 the way it was intended.

Mark-up structures content, but your choice of tags matters a lot less than we've been taught for a while. Let's go through some of the reasons why.

THE WEB NO LONGER CONSISTS OF STRUCTURED CONTENT

In the golden days of the Web, Web pages were supposed to be repositories of information and meaning, nothing more. Today, the Web has content, but meaning is derived from users' interactions with it.

XML, RDFA, Dublin Core and other structured specifications have very solid use cases, but those use cases do not account for the majority of interactions on the Web. Heck, no website really has the purity of semantic mark-up that such specifications demand. Mark Pilgrim writes about this much better than I do.

If you have content that demands semantic purity — such as a library database, a document that needs a table of contents, or an online book (i.e. anything for which semantic purity makes sense) — then by all means stick to the HTML5 outlining algorithm, and split hairs on which element should be an **`article`** and which a **`section`**. No customer-facing tool exists that takes advantage of this algorithm by producing a table of contents. No browser seems to exploit such tools either.

IS IT REALLY ACCESSIBLE?

If accessibility is your reason for using semantic mark-up, then understand that accessibility and semantic mark-up have very little correlation, due to the massive abuse of HTML mark-up on the Web. (I would love to link to Mark Pilgrim's post on this, but it is dead, so this will have to do.)

The **`b`**, **`strong`**, **`i`** and `em` tags are equivalent to the **`span`** tag as far as the specification is concerned. And so are some of HTML5's tags.

As stated on HTML5 Accessibility, almost every new HTML5 element currently provides to assistive technology only as much semantic information as a **`div`** element. So, if you thought that using HTML5 elements would make your website more accessible, think again. (How much additional information do **`<figure>`** and **`<figcaption>`** bring? None.)

The recent debate (or debacle?) on the `<time>` element is just more proof of the impermanence of the semantic meanings associated with elements.

IS IT REALLY SEARCHABLE?

If SEO is your grand purpose for using semantic mark-up, then know that most search engines do not give more credence to a page just because of its mark-up. The only thing recommended in this SEO guide from Google is to use relevant headings and anchor links (other search engines work similarly). Your use of HTML5 elements or of **`strong`** or **`span`** tags will not affect how your content is read by them.

There is another way to provide rich data to search engines, and that is via micro-data. In no way does this make your website rank better on search engines; it simply adds value to the search result when a relevant one is found for your website.

IS IT REALLY PORTABLE?

Another much-touted advantage of the semantic Web is data portability. Miraculously, all devices are supposed to understand the semantic mark-up used everywhere and be able to parse the information therein with no effort. Aryeh Gregor puts that myth to sleep:

… +Manu Sporny said that semantic Web people had received feedback that out-of-band data was harder to keep in sync with content. I can attest that in MediaWiki's case this isn't true, though… The only times I can see where you'd want to use RDFa or microdata instead of separate RDF is if either you don't have good enough page-generation tools, or you want the metadata to be consumed by specific known clients that only support inline metadata (e.g. search engines supporting schema.org or such). If the page is being processed by a script anyway, and if the script author has ready access to server-side tools that can extract the metadata into a separate RDF stream, then it's normally going to be just as easy to publish as a separate stream as to publish inline. And it saves a lot of bloat on every page view.

What Now, Then?

- There is no harm using **`div`** elements; you can continue using them instead of **`section`** and **`article`**. I think we should use the new elements to make your mark-up readable, not for any inherent semantic advantage. If you want to use HTML5 **`section`** and **`article`** tags to enhance some particular textual documentation for a future document reader, do it.
- Tools exist today that take advantage of the **`nav`**, **`header`** and **`footer`** elements. NVDA now assigns implied semantics with these elements. The elements are straightforward to understand and use.
- There is good support for ARIA landmarks in screen readers, but be careful when using them with HTML5 elements.

- HTML5 has a host of new features. I think we should learn about them, use them, give feedback. Make these features more robust and stable. Yes, most of these features require that you understand and write JavaScript and expose features that create a richer experience for your audience. If that task sounds formidable to you, then start learning how to code, particularly JavaScript.

Pursuing Semantic Value

By Jeremy Keith

Disclaimer: This article by Jeremy Keith is a reactions to the article on the pursuit of semantic value by Divya Manian. Both articles are published in the *Opinion column* section in which we provide active members of the community with the opportunity to share their thoughts and ideas publicly.

Divya Manian, one of the super-smart web warriors behind *HTML5 Boilerplate*, has published an article called Our Pointless Pursuit Of Semantic Value.

I'm afraid I have to agree with Patrick's comment when he says that the abrasive title, the confrontational tone and strawman arguments at the start of the article make it hard to get to the real message.

But if you can get past the blustery tone and get to the kernel of the article, it's a fairly straightforward message: don't get too hung up on semantics to the detriment of other important facets of web development. Divya clarifies this in a comment:

> *Amen, this is the message the article gets to. Not semantics are useless but its not worth worrying over minute detail on.*

The specific example of **`divs`** and sectioning content is troublesome though. There **is** a difference between a **`div`** and a **`section`** or **`article`** (or **`aside`** or **`nav`**). I don't just mean the semantic difference (a **`div`** conveys no meaning about the contained content whereas a **`section`** element is specifically for enclosing *thematically-related* content). There are also practical differences.

A **`section`** element will have an effect on the generated outline for a document (a **`div`** will not). The new outline algorithm in HTML5 will make life a lot easier for future assistive technology and searchbots (as other people mentioned in the comments) but it already has practical effects today in some browsers in their default styling.

Download the HTML document I've thrown up at https://gist.github.com/1360458 and open it in the latest version of Safari, Chrome or Firefox. You'll notice that the same element (**`h1`**) will have different styling depending on whether it is within a **`div`** or within a **`section`** element (thanks to **`-moz-any`** and **`-webkit-any`** **CSS** declarations in the browser's default stylesheets).

So that's one illustration of the practical difference between **`div`** and **`section`**.

Now with that said, I somewhat concur with the conclusion of "when in doubt, just use a div". I see far too many documents where every **`div`** has been swapped out for a **`section`** or an **`article`** or a **`nav`** or an **`aside`**. But my reason for coming to that conclusion is the polar opposite of Divya's reasoning. Whereas Divya is saying there is effectively no difference between using a **`div`** and using sectioning content, the opposite is the case: it makes a big difference to the document's outline. So if you use a **`section`** or **`article`** or **`aside`** or **`nav`** without realizing the consequences, the results could be much worse than if you had simply used a **`div`**.

I also agree that there's a balance to be struck in the native semantics of HTML. In many ways its power comes from the fact that it is a limited—but universally understood by browsers—set of semantics. If we had an element for every possible type of content, the language would be useless. Personally, I’m not convinced that we need a **`section`** element *and* an **`article`** element: the semantics of those two elements are so close as to be practically identical.

And that’s the reason why right now is *exactly* the time for web developers to be thinking about semantics. The specification is still being put together and our collective voice matters. If we want to have well-considered semantic elements in the language, we need to take the time to consider the effects of every new element that could potentially be used to structure our content.

So I will continue to stop and think when it comes to choosing elements and class names just as much as I would sweat the details of visual design or the punctation in my copy or the coding style of my JavaScript.

The Semantic Grid System: Page Layout For Tomorrow

By Tyler Tate

CSS grid frameworks can make your life easier, but they're not without their faults. Fortunately for us, modern techniques offer a new approach to constructing page layouts. But before getting to the solution, we must first understand the three seemingly insurmountable flaws currently affecting CSS grids.

Problems

PROBLEM #1: THEY'RE NOT SEMANTIC

The biggest complaint I've heard from purists since I created The 1KB CSS Grid two years ago is that CSS grid systems don't allow for a proper separation of mark-up and presentation. Grid systems require that Web designers add `.grid_x` CSS classes to HTML elements, mixing presentational information with otherwise semantic mark-up.

Floated elements must also be cleared, often requiring unnecessary elements to be added to the page. This is illustrated by the "clearing" div that ships with 960.gs:

```
<div class="grid_3">
  220
</div>
<div class="grid_9">
  700
</div>
<div class="clear"></div>
```

PROBLEM #2: THEY'RE NOT FLUID

While CSS grids work well for fixed-width layouts, dealing with fluid percentages is trickier. While most grid systems do provide a fluid option, they break down when nested columns are introduced. In the 1KB CSS Grid example below, **`.grid_6`** would normally be set to a width of 50%, while **`.grid_3`** would typically be set to 25%.

But when **`.grid_3`** appears inside of a **`.grid_6`** cell, the percentages must be recalculated. While a typical grid system needs just 12 CSS rules to specify the widths of all 12 columns, a fluid grid would need 144 rules to allow for just one level of nesting: possible, but not very convenient.

```
<div class="column grid_6">
  <div class="row">
      <div class="column grid_3"> </div>
      <div class="column grid_3"> </div>
  </div>
</div>
```

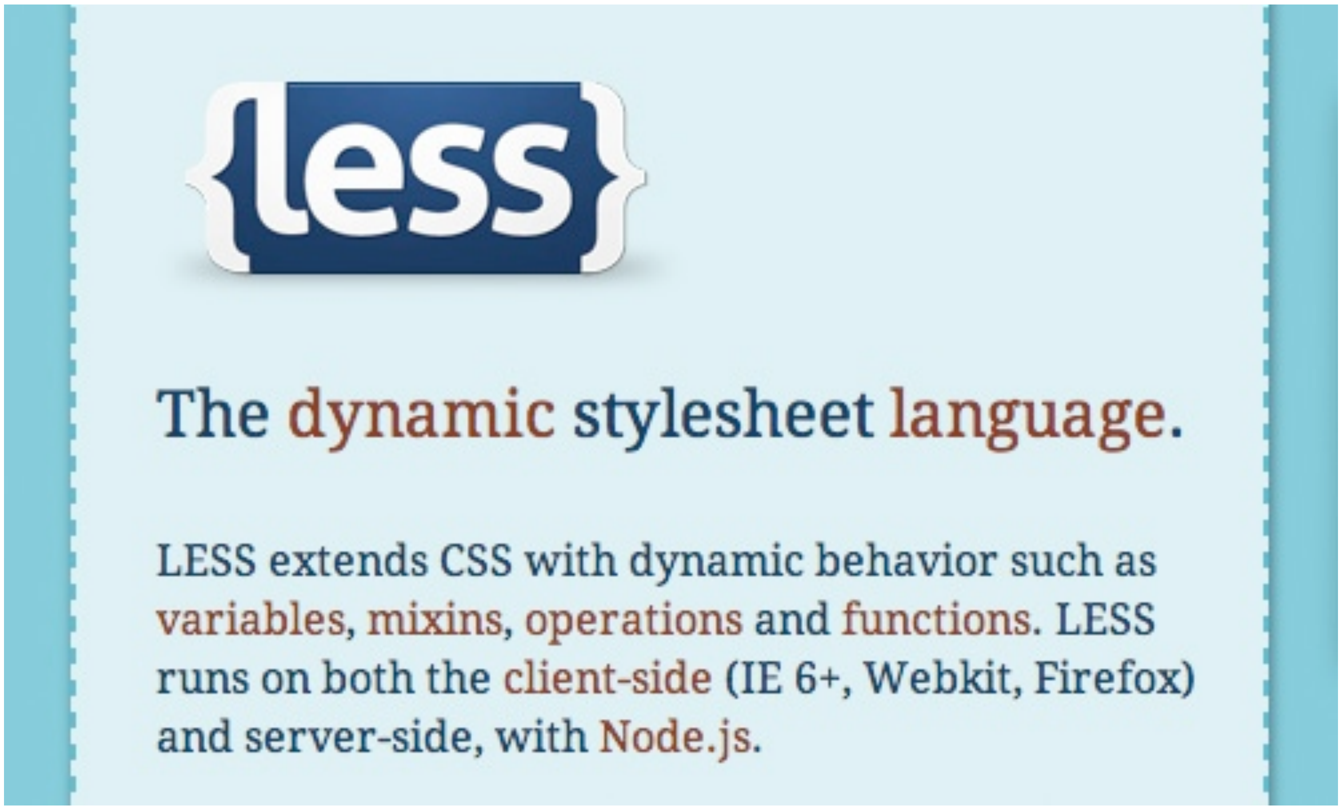
PROBLEM #3: THEY'RE NOT RESPONSIVE

Responsive Web design is the buzzword of the year. While new tools such as [1140 CSS Grid]() and [Adapt.js]() are springing up that enable you to alter a page's layout based on screen size or device type, an optimal solution has yet to arrive.

Blame It On The Tools

All three of these problems directly result from the limitations of our existing tools. CSS leaves us with the ultimatum of either compromising our principles by adding presentational classes to mark-up, or sticking to our guns and forgoing a grid system altogether. But, hey, we can't do anything about it, right?

Well, not so fast. While we wait for browsers to add native CSS support for this flawed grid layout module, a futuristic version of CSS is available *today* that's already supported by every CSS-enabled browser: LESS CSS.



The dynamic stylesheet language.

LESS extends CSS with dynamic behavior such as variables, mixins, operations and functions. LESS runs on both the client-side (IE 6+, Webkit, Firefox) and server-side, with Node.js.

LESS brings powerful new features to CSS.

LESS What?

You've probably heard of LESS but perhaps have never given it a try. Similar to SASS, LESS is *extends* CSS by giving you the ability to use variables, perform operations and develop reusable mixins. Below are a few examples of what it can do.

VARIABLES

Specify a value once, and then reuse it throughout the style sheet by defining variables.

```
// LESS
@color: #4D926F;

#header {
  color: @color;
}
```

The above example would compile as follows:

```
/* Compiled CSS */
#header {
  color: #4D926F;
}
```

OPERATIONS

Multiply, divide, add and subtract values and colors using operations.

```
// LESS
@border-width: 1px;
#header {
  border-left: @border-width * 3;
}
```

In this example, **`1px`** is multiplied by 3 to yield the following:

```
/* Compiled CSS */
#header {
  border-left: 3px;
}
```

MIXINS

Most powerful of all, mixins enable entire snippets of CSS to be reused. Simply include the class name of a mixin within another class. What's more, LESS allows parameters to be passed into the mixin.

```
// LESS
.rounded(@radius) {
    -webkit-border-radius: @radius;
    -moz-border-radius: @radius;
    border-radius: @radius;
}
#header {
    .rounded(5px);
}
```

Verbose, browser-specific CSS3 properties demonstrate the benefit that mixins bring:

```
/* Compiled CSS */
#header {
  -webkit-border-radius: 5px;
  -moz-border-radius: 5px;
  border-radius: 5px;
}
```

DOWNSIDES TO LESS

Having been skeptical of LESS at first, I'm now a strong advocate. LESS style sheets are concise and readable, and they encourage code to be reused. However, there are some potential downsides to be aware of:

1. It has to be compiled. This is one extra step that you don't have to worry about with vanilla CSS.
2. Depending on how LESS documents are structured, the compiled CSS file might be slightly larger than the equivalent hand-crafted CSS file.

A NOTE ON COMPILING LESS

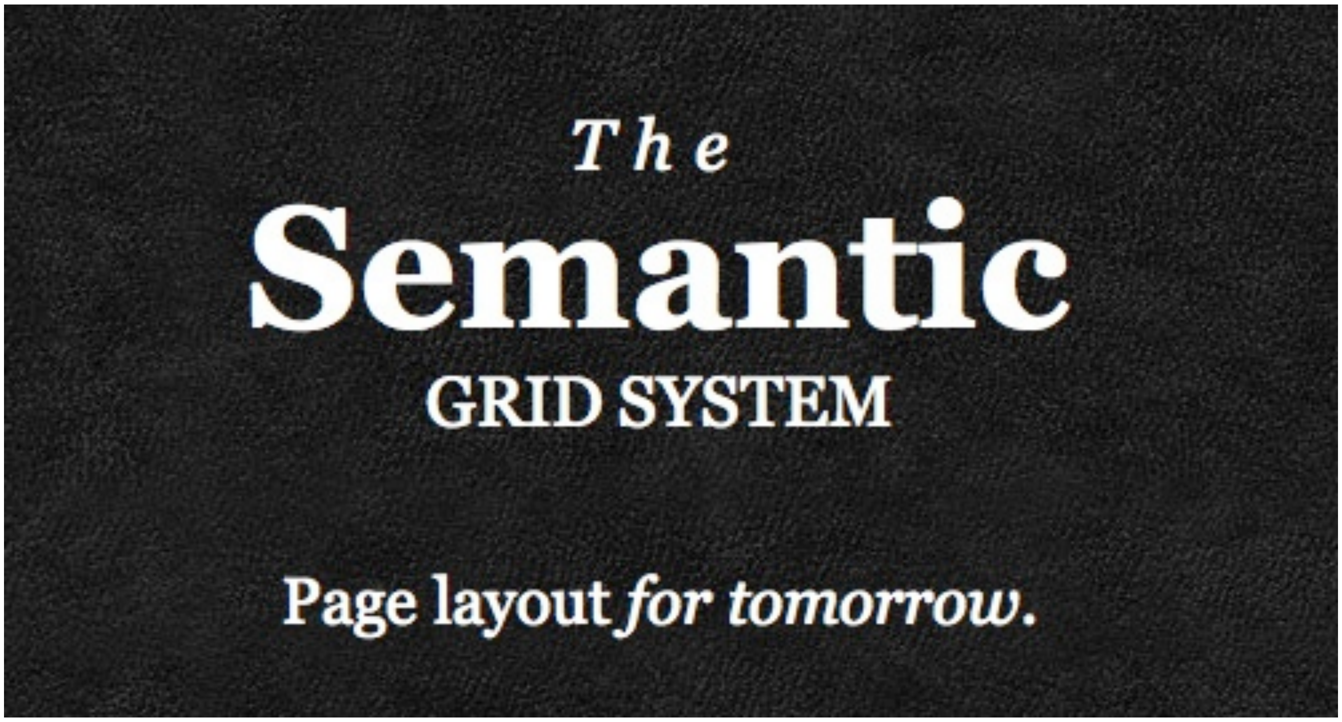
There are three approaches to compiling LESS style sheets into CSS:

- **Let the browser do the compiling.**
  As its name suggests, LESS.js is written in JavaScript and can compile LESS into CSS directly in the user's browser. While this method is convenient for development, using one of the next two methods before going into production would be best (because compiling in the browser can take a few hundred milliseconds).
- **Use a server-side compiler.**
  LESS.js can also compile server-side with Node.js, and it has been ported to several other sever-side languages.
- **Use a desktop app.**
  LESS.app is a Mac app that compiles local files as they're saved on your computer.

Introducing The Semantic Grid System

The innovations that LESS brings to CSS are the foundation for a powerful new approach to constructing page layouts. That approach is the The Semantic Grid System. This new breed of CSS grid shines where the others fall short:

1. It's semantic;
2. It can be either fixed or fluid;
3. It's responsive;
4. It allows the number of columns, column widths and gutter widths to be modified instantly, directly in the style sheet.



The Semantic Grid System uses LESS CSS to offer a new approach to page layout.

CONFIGURING THE GRID

Sounds too good to be true? Here's how it works.

First, import the semantic grid into your working LESS style sheet.

```
@import 'grid.less';
```

Next, define variables for the number of columns, and set the desired widths for the column and gutter. The values entered here will result in a 960-pixel grid system.

```
@columns: 12;
@column-width: 60;
@gutter-width: 20;
```

The grid is now configured and ready to be used for page layout.

USING THE GRID

Now that the grid has been configured, consider two elements on an HTML page that you would like to lay out side by side.

```
<body>
  <article>Main</article>
  <section>Sidebar</section>
</body>
```

The side-by-side layout can be achieved by passing the desired number of grid units to the **`.column()`** mixin (which is defined in the *grid.less* file).

```
// LESS
@import 'grid.less';

@columns: 12;
@column-width: 60;
@gutter-width: 20;
```

```
article {
  .column(9);
}
section {
  .column(3);
}
```

The above LESS would be compiled to CSS as the following:

```
/* Compiled CSS */
article {
  display: inline;
  float: left;
  margin: 0px 10px;
  width: 700px;
}
section {
  display: inline;
  float: left;
  margin: 0px 10px;
  width: 220px;
}
```

This page demonstrates the result. What makes this approach so different is that it does away with ugly **`.grid_x`** classes in the mark-up. Instead, column widths are set directly in the style sheet, enabling a clean separation between declarative mark-up and presentational style sheets. (It’s called the *semantic* grid for a reason, after all.)

SO, WHAT’S BEHIND THE CURTAIN?

For the curious among you, below are the mixins at the center of it all. Fortunately, these functions are hidden away in the *grid.less* file and need not ever be edited.

```
// Utility variable — you will never need to modify this
@_gridsystem-width: (@column-width*@columns) + (@gutter-
width*@columns) * 1px;

// Set @total-width to 100% for a fluid layout
@total-width: @_gridsystem-width;

// The mixins
.row(@columns:@columns) {
   display: inline-block;
   overflow: hidden;
   width: @total-width*((@gutter-width + @_gridsystem-width)/
@_gridsystem-width);
   margin: 0 @total-width*(((@gutter-width*.5)/@_gridsystem-
width)*-1);
}
.column(@x,@columns:@columns) {
   display: inline;
   float: left;
   width: @total-width*((((@gutter-width+@column-width)*@x)-
@gutter-width) / @_gridsystem-width);
   margin: 0 @total-width*((@gutter-width*.5)/@_gridsystem-
width);
}
```

Fluid Layouts

The example above demonstrates a fixed pixel-based layout. But fluid percentage-based layouts are just as easy. To switch from pixels to percentages, simply add one variable:

```
// LESS
@total-width: 100%;
```

With no other changes, the compiled CSS then becomes this:

```
/* Compiled CSS */
article {
  display: inline;
  float: left;
  margin: 0px 1.04167%;
  width: 72.9167%;
}
section {
  display: inline;
  float: left;
  margin: 0px 1.04167%;
  width: 22.9167%;
}
```

This example shows how the percentages are dynamically calculated using LESS operations, which also applies to nested columns.

Responsive Layouts

No modern grid system would be complete unless we had the ability to adapt the layout of the page to the size of the user's screen or device. With Semantic.gs, manipulating the grid using media queries couldn't be any easier:

```
article { .column(9); }
section { .column(3); }

@media screen and (max-width: 720px) {
  article { .column(12); }
  section { .column(12); }
}
```

Try It For Yourself

Just a couple of days ago Twitter released a project called Bootstrap which provides similar (but more limited) grid system built using LESS variable and mixins. The future of the CSS grid seems to be taking shape before us.

The Semantic Grid System delivers the best of both worlds: the power and convenience of a CSS grid and the ideal separation of mark-up and presentation. Download the grid for yourself, fork it on GitHub, and let us know what you think!

About The Authors

Bruce Lawson

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

Niels Matthijs

Niels Matthijs spends his spare time combining his luxury life with the agonizing pressure of blogging under his Onderhond moniker. As a front-end developer he is raised at Internet Architects, investing plenty of time in making the web a more accessible and pleasant place.

Derek Johnson

Derek builds websites at WebsiteNI and helps curate the HTML5 Gallery, where he gets particularly hung up about document outlines. His favourite place in the world is the Mourne Mountains but you are more likely to find him lurking on Twitter.

Divya Manian

Divya Manian is a web opener for Opera Software in Seattle. She made the jump from developing device drivers for Motorola phones to designing websites and has not looked back since. She takes her duties as an Open Web vigilante seriously which has resulted in collaborative projects such as HTML5 Boilerplate.

Jeremy Keith

Jeremy Keith is a Web developer and author living and working in Brighton, England.

Tyler Tate

Tyler Tate is a London-based user experience designer focused on making the complex feel simple. He leads UX at TwigKit, is the creator of the 1KB CSS Grid, and has led the design of big web applications from CMS systems to the Nutshell CRM. You can keep up with him on Twitter.