WRINKLEFREE

JS for HIPSTERS

by Matt Keas

HIGH
QUALITY
CODING

Wrinklefree JS for Hipsters

A cutthroat guide to architect, design, and oversee web application projects.

Matthew Keas

This book is for sale at http://leanpub.com/wrinklefree

This version was published on 2013-11-02

Tweet This Book!

Please help Matthew Keas by spreading the word about this book on Twitter!

The suggested tweet for this book is:

Wrinklefree JS for Hipsters, just got mine! :D #wrinklefreejs https://leanpub.com/wrinklefree

The suggested hashtag for this book is #wrinklefreejs .

Find out what other people are saying about the book by clicking on this link to search for this hashtag on Twitter:

https://twitter.com/search?q=#wrinklefreejs

Also By Matthew Keas

For Rebecca and Pip - Thank you for putting up with the numerous 2am bedtimes and the sleepless nights. I love you both.

Contents

1 Introduction

There are many books, articles and presentations that have influenced the way I approach the web today. Its always been one of my favorite parts of this community: a willingness and eagerness to share what we learn with each other. So many blog posts, online publications, and presentations continually challenged me to move forward by presenting new techniques and ways of thinking about the technology I used on a daily basis.

I believe that it is extremely important for us to ABC - "Always Be Coding". Moreover, we should always be learning. Empathy, Curiosity, and Drive are my personal engines, fueled with a hefty tank of Ambition.

We read tutorials and developer websites, implement new design styles ("Flat UI", anyone? How about Skeumorphism?) and recreate JavaScript frameworks and libraries like it is the next best thing since sliced bread. (Ember.js and Angular, I'm looking at you)

So where do we look at ourselves and try to improve, when we've learned all we can about jQuery and made more Twitter Bootstrap prototypes than we can count?

My answer: It's time to re-evaluate what we "know" about the web and the browser.

Can any single person stand up in a room and discuss the cacophony of technologies that a browser implements? (If you can, please call me. I have questions.)

I hope that answer for YOU, dear reader, is "almost".

The point I'm getting to is that coding a website is more than just knowing HTML, CSS, and JavaScript. It is more than begin familiar with WordPress and all-powerful with Node.

Try sitting back in your chair while at your computer. Now, envision twenty devices in front of you - each a different size, OS, or form factor.

Now ask yourself, "How do I create an EXPERIENCE that works with ALL of these?".

If you said "Responsive Web Design", I want you to pat your shoulder - but only once. This is because building cross-platform websites and web applications are part of a "Responsive Experience". This is a much larger beast.

My goal with this book is to address far more than some JavaScript techniques. I want to `require()` in myself all the experiences and struggles I have faced in coding and project management endeavors, and then `publish()` these thoughts to anyone who is willing to `listen()`.

Progressive Enhancement, and why I believe your code is really the last thing to fix.

Through many life, work, and learning optimization content, such as the 4 Hour Workweek, Zen Habits, LifeHacker, we hear or read about the 80/20 rule. More commonly known as Pareto's Principle, it governs that for many events, roughly 80% of the effects come from 20% of the causes.

That's a broad fucking spectrum of things to consider.

I prefer to focus on Parkinson's Law, which dictates on differing principles: "Work expand so as to fill the time available for its completion."

Why is this a better paradigm to focus on? Well, it means that we can be more effictive by diligently focusing and practicing to maximize our efforts. Depending on what you do – be it marketing, coding, or cooking – this effects the 80/20 Rule. The more focused and practiced you are at something, the better your ratio!

I recently did this myself, by diligently focusing while building my own single page app framework (viewable on my website: http://mkeas.org[1]) in a few hours because I wanted to prove that I could. This included a short prototyping and design phase, followed by implementation, server-side software vetting, and ultimately implementation.

Once I had all of the main features ready, I used the platform Heroku (http://heroku.com)[2] to publish… for free!

Guiding Principles

Before jumping into the first Act, I'd like to discuss some Guiding Principles (GP) to help keep perspective while you work through this book.

1. Keep it AVAILABLE

   Availability is the bedrock of a website. If the site's not available to people, game over, we might as well not even bother. And so our job is to make sure that everything we do is out there, working, available to everybody. This needs to be true regardless of where they are in the world, or what special circumstances they may be under — everything needs to be available and accessible. And I purposely used the word availability — you hear a lot of talk about accessibility, but availability applies to everybody, and it's an umbrella term that encompasses accessibility also. And so I like to think about just making the sites available to everybody, whatever that entails.

[1]http://mkeas.org
[2]http://heroku.com

2. Keep it RICH

   You're probably not the average user. We sit on top of a tremendously fat internet connections – everything's blazing fast, and many of us work on brand new hardware with lots of memory. As we're doing rich JavaScript development, and as I'm writing lots of software that gets executed in the browser, coding and testing may go very smoothly. However, we need to remember that different people have different sort of equipment, and bandwidth, and so forth. Remember that – that's another reason to build in the layers, and to make things feature-rich.
3. Keep it STABLE and SMALL

   The web is young; we don't know what's coming around the corner. We don't know what's going to be invented next, what technology's coming next. And so it's important to continually invest in stability, in strong infrastructure, in stable code, so that we have that strong platform to stand on as the next thing happens. So by focusing on stability, you're really investing in your future, and again, preparing for that future — whatever it might bring.
4. Keep it about the USER

   I often get asked at the beginning of a project: "which browsers do we need to support on this project? Are we going to support Netscape 4? Are we going to support IE 5 on the Mac?" And it was always asked in a binary sense, where the answer is either yes or no. And we realized over time that that's counter to our goal of maximum availability – if we're ever choosing not to support a particular browser, that means we're choosing to have less than complete availability. So that was the first thing that we had to understand, is that support is not binary. The second thing we had to understand is that support doesn't mean identical (and THAT IS THE MOST ANNOYING AND POINTLESS FUCKING THING TO HAVE TO DO, EVER). Forcing every pixel to be in the exact same place on every user agent in the world isn't what it means to support the user agent. Instead, to support a browser, we want to give the browser what it can handle, in the most efficient way possible. Your user will be happier knowing that the software being used is made for the platform they are on (because that means you were a smart developer and realized limitations).

To supplement this ideals and how you can adhere to them in day-to-day development, I will present 10 simple rules of writing JavaScript libraries:

1. Don't overstep boundaries

   I hold a firm belief that JavaScript libraries should be minimal – they should perform one small set of functions around a single context, and perform them well. This helps you use ALL of it or NONE of it in your application code; insuring that code utilization remains high and you aren't sending the browser 50KB of unused JavaScript.
2. Don't break the DOM

   Any `<script>` tags with the `async` attribute will not halt the rendering of a webpage in browsers newer than IE9 (Firefox, Safari, Chrome, Opera, etc all work). In the event that you are loading multiple JavaScript files which depend on something before it (jQuery and a .js file that will use jQuery), you must either use a script-loading library (such as mine: https://github.com/matthiasak/Loader[3] or RequireJS) or take out the `async` attribute and make sure the `<script>` tags are the very last items in the `<body>` element, to help the browser render faster! Initial time-to-glass (TTG) can mean a lot to users' perception on a website's speed.

3. Don't rely on frameworks

   We all like and love jQuery. I even wrote another book on using jQuery with HTML5 APIs. Why does that mean you shouldn't use it? You should, but be smart, and know how it works behind the scenes. I can write native JavaScript code that is 1000x faster than simple jQuery functions which works across all browsers and platforms newer than IE8. I will cover this in Act II.

4. Don't listen-and-forget-it

   You like using jQuery to listen for events (e.g. using `$.on()`). That's fine, I do as well! However, libraries like Backbone certainly got it right on the money by deleting those events if the Backbone view that these events are attached to is removed from the DOM. If you use a framework, or write your own code, make sure you follow these same guidelines to help your web app's performance.

5. Don't fuck with UX

   Every browser and platform has different default gestures and accessibility features. If your library is too narrow-minded and depends too much on how a device and browser works, or how the DOM / CSS currently looks, then you could be missing out on re-using the code for new devices in the future (like Google Glass).

6. Don't hog resources

   Always assume limited resources. I try to write JavaScript with the mindset of Mobile-first just so that I can force myself to consider possible performance hurdles in the future.

7. Don't overwrite my damn stylesheet

   Your code should almost never set a direct style on an element in the DOM. If you catch yourself coding `padding: 10px;` on en element in JavaScript, stop and refactor immediately. CSS is for that. So instead, add a CSS class to the element, and defer as much as you can to using CSS or HTML features instead of JavaScript.

8. Don't do desktop first

   This one has already been discussed at length, but you should minimize your JavaScript as much as possible, and polyfill to support older browsers and features. If you write a web app that is designed and architected for a desktop, you might have trouble reusing that code for a mobile experience. Thus, code mobile first, and enhance that features and design for larger/more powerful devices.

9. Don't ignore the 16.4ms loop

   16.4ms is the time you have to perform some functions in JavaScript before the screen will update (60fps). If your JavaScript is animating something in the browser, and it takes longer

[3]https://github.com/matthiasak/Loader

than 16.4ms to update the interface, then you are creating `jank`. Stay `jank` free, and read Act VI.

10. Don't ignore other developers

    You may not always be the only developer working on a project. Many of us work with other developers to get something done. If your code:

    - uses other libraries
    - is used by other libraries
    - is used by other developers
    - runs on the same page as other developers' code

    Then you should probably do your best to keep a small modular API, and keep your code as DRY as possible. For more details on this, check out Act V.

On Web Development and Shakespeare

The Oxford English Dictionary says "science" originally just meant knowledge. In the Middle Ages, the seven liberal arts (grammar, logic, rhetoric, arithmetic, music, geometry, astronomy) were also called the "seven liberal sciences." Shakespeare used science in that sense: "Cunning in Musicke, and the Mathematickes, To instruct… in those sciences."

We risk blinding ourselves with the science of creating websites (read: JavaScript frameworks like Angular), in their now narrower highly reliable form.

They are dazzling. But they also have limits.

Their tools don't fit all situations. Web development, especially the JavaScript kind, has history-like aspects. Their narratives might be woven with HTML, but not everything that is markup can be marked-up. They must deal with different kinds of change that (to paraphrase Shakespeare) were never dreamt of in our original code. Perhaps this limits JavaScript to moderately reliable maxims.

Even when I rub shoulders with paradigms such as not using `RequireJS` and instead writing my own damn script loader (https://github.com/matthiasak/Loader)[4], I find that my website (http://mkeas.org)[5] is far faster, far smoother, and far sexier than most Ember/Angular/Backbone/etc apps.

Thus, it's not always just the execution, it's the VISION.

Following the VISION

Remember that techniques have a limited shelf-life. Frameworks like Meteor, Ember, Backbone, and Angular are techniques that are attempting to follow a philosophy.

[4]https://github.com/matthiasak/Loader

[5]http://mkeas.org

As web technologies grow and change, you will probably find that your framework itself will change. But you didn't realize how to get client-side MVC routing or dynamic templating with cascaded views into your app without Backbone Marionette, did you?

Thank goodness you don't have to maintain that code 4 years from now. You'll be working somewhere else, right?

That sort of question doesn't need attention. The real key is to truly understand these "features" that are provided by frameworks. Frameworks always create these "ideal pictures" of an app, and it's support in a browser. Until SHIT HITS THE FAN. The only thing we can do is "pick-n-choose" frameworks wisely, build our own, or get the problem fixed if it arises.

Building robust sites typically includes a lot of painful testing, a lot of painful bugs, and a lot of painful iterations… JUST SO IT CAN WORK IN IE8.

Is there a cost associated in building robust sites? Yes, though it's not nearly as bad as many seem to think. As with anything, the more you learn the quicker the process will go. Eventually, it just becomes the way you work. I remember going through this with CSS. At first it was a beast and yeah, building with tables was easier. But as I learned more and more about the spec, about how browsers behaved, and about how to make my process efficient, that time gap gradually reduced itself to being minimal, at most. (We will cover some fun CSS optimizations, too.)

When topics like this come up, we focus a lot on the benefits for people with less than ideal situations (IE8). That appeals to some of us and less so to others. For those unmoved by that, consider this: by building something that can handle less than ideal circumstances, by removing assumptions from your development process, you make your site better equipped to handle the unpredictability of the web's future. For instance, do you know if your site work on Google Glass? ;)

We can build robust sites, but don't be blind to graceful feature/design degradation for the sake of uniformity. This can produce some pretty stifling problems in its wake.

In fact, I prefer to follow feature ENHANCEMENT. Start small, start Mobile first, and always assume limited computing resources. Animations that are fast on a mobile device will always be smooth on a desktop, too. However, animations that are smooth on a desktop will not always be fast on a mobile device!