acmqueue

# Dismantling the Barriers to Entry

**We have to choose to build a web that is accessible to everyone.**

Rich Harris

A war is being waged in the world of web development. On one side is a vanguard of toolmakers and tool users, who thrive on the destruction of bad old ideas ("old," in this milieu, meaning anything that debuted on Hacker News more than a month ago) and raucous debates about transpilers and suchlike.

On the other side is an increasingly vocal contingent of developers who claim – not entirely without justification – that the head-spinning rate of innovation makes it impossible to stay up to date, and that the web is disintegrating into a jumble of hacks upon opinions, most of which are wrong, and all of which will have changed by the time hot-new-thing.js reaches version 1.0.0.

This second group advocates a return to the basics, eschewing modern JavaScript libraries and frameworks in favor of untamed DOM APIs (the DOM being the closest we unwashed web developers ever get to "bare metal"). Let's call it the back-to-the-land movement. The back-to-the-landers argue that tools slow the web down, harm accessibility, and increase fragility. You can often find them linking to vanilla-js.com in the comments of programming blogs.

Here's Peter-Paul Koch, the creator of quirksmode.org, in a recent article[6] (emphasis original):

"The movement toward toolchains and ever more libraries to do ever less useful things has become hysterical, and with every day that passes I'm more happy with my 2006 decision to ignore tools and just carry on. **Tools don't solve problems any more, they have *become* the problem.**"

Setting aside the "get off my lawn" tone of much of this commentary, the movement does have valid concerns. But we expect more of the web than we used to – real-time collaboration, personalized apps, rich interactivity. We can't expect software engineers to build those experiences without tools any more than we expect civil engineers to build suspension bridges by hand. As Facebook's Sebastian Markbåge says in a direct response to Koch[7], "the only time you can say that the Web is "good enough" is when you're building for yesterday's Web."

As in any war, there are false dichotomies (simplicity versus power), hypocrisies (abandoning libraries then writing acres of app code that do the same thing, albeit without documentation or tests), and casualties. It's the casualties that I want to talk about.

## FRONT-ENDERS: AN ENDANGERED SPECIES?

Until relatively recently, "front-end developer" was a slightly derisive term for someone who could cobble together some HTML and CSS and sprinkle some JavaScript on top of it, perhaps after searching Stack Overflow for "how to hide element with jQuery." The front-ender was responsible for adding the Google Analytics script snippet to the CMS article template, and perhaps adding a

carousel of sliding images (the traditional cure for the marketing department's indecision about what to put on the homepage), but was never trusted with anything particularly important.

Then along came Backbone[1], which was the starting pistol in the race towards ever-more-elaborate JavaScript application frameworks. Many modern web apps push almost all the logic out to the client, the result being that as applications become more sophisticated, so must the tools – and the people using them.

As a consequence, many commentators have placed the traditional front-ender on extinction watch. Trek Glowacki, a core member of the Ember.js team (Ember is one of the aforementioned client-side application frameworks), wrote in response to a lament about build tools:

"I know everyone on Ember core sympathizes with web developers whose careers started during the 'download a zip, add some script tags, FTP into production' era for the 'front end' and now feel a bit startled that all their favorite tools are becoming increasingly complex. But, the fact remains, that era is ending."[5]

In other words, "get with the program." Glowacki isn't wrong, just like Koch isn't wrong, but there's a problem with modern tools – newcomers to the field, after they've been greeted with an overwhelming number of choices, are expected to learn a dizzying array of new concepts (insert joke about "transclusion" here) before they can actually build anything. The incredible power of those tools is only really available to a select few – those with the determination to ascend a steep learning curve, and the time and inclination to keep pace with our community's frantic innovation.

## "LEARN TO CODE" IS NOT THE ANSWER

Back when the web was a simpler place, it was a welcoming environment for newbie programmers. There were fewer tools, and the ones we had were a good deal less sophisticated, but we made up for it with the power of "view source." In those Wild West days, before we cared about *best practices*, it was surprisingly easy to reverse-engineer a lot of web software.

Web development has matured spectacularly in a few short years. But the tools that have supplanted "view source" (which is useless in an age of transpiled, minified code) are not accessible to the vast majority.

It's not simply a question of better training for those who would be professional software engineers. The power and beauty of the web was always that anyone could participate as a creator as well as a consumer – scientists, academics, artists, journalists, activists, entertainers, educators – most of whom have yet to unlock the thrilling possibilities of modern web technologies.

One way that we've tried to address this problem is with the "learn to code" movement, which has spawned an entire industry of startups (startup culture itself being one of the prime drivers of learn-to-code). Politicians love it because it makes them look forward-thinking, though no-one is quite sure if Michael Bloomberg ever did finish his Codecademy course[2].

There is plenty to admire about learn-to-code, of course. Many people have developed skills that would otherwise have been out of reach. But the movement rests on two odd assumptions – firstly that our priority should be to make more programmer talent rather than making programming more accessible, and secondly that "learning to code" consists of absorbing facts about programming languages and practicing the formation of correct syntax.

In reality, learning how to program is a process of developing the ability to model problems in such a way that a computer can solve them – something that only happens through experience. You don't learn a foreign language by learning how to conjugate verbs and pluralize nouns; you learn by picking up phrases and practicing them, and reading and listening to native speakers until it becomes natural. Every language teacher knows this, yet to a large extent it's not how we teach programming languages.

We don't need the 1,437th explanation of prototypal inheritance or JavaScript's `this` keyword. What we need are tools that allow novices to express their ideas without a complete knowledge of the process by which it happens.

## ENTER RACTIVE.JS

A few years ago I was in need of such a tool, having recently joined the interactive news team at theguardian.com. News interactives typically contain a lot of *state*, represented in several different visually rich forms, and have to handle many different modes of user interaction – a recipe for buggy code, especially when written against news industry deadlines (we laugh at the term "agile"). I was well aware that my jQuery spaghetti was always a few keystrokes away from implosion, but more advanced tools such as Angular were both too intimidating and yet somehow inadequate for the task at hand.

I had been looking forward to the day when someone would let me in on the secret to doing it properly, but that day never came. There simply weren't any tools designed to make my job easier, so I resolved to create one myself.

Laid bare, the problem is relatively simple to articulate. The state of a web app UI at any given moment can be described as a function of application state, and our task is to manipulate the DOM until the reality matches the intention.

On the server, it's easy: write a template, compile it to a function with a templating engine, call it with some data, and serve the resulting HTML to the client. But string templating is a bad technique once you're in the browser. Repeatedly generating HTML and inserting it into the document means trashing the existing DOM, which taxes the garbage collector and destroys state (such as which element is focused, and where the cursor is). Because of that, developers typically break their applications apart into microscopic chunks, with dedicated custom Model and View classes tied together with an events system. MVC duct tape is the new jQuery spaghetti.

Ractive.js[10] was designed to allow developers to use the declarative power of templates to their fullest extent without the sacrifices that come from string-based templating systems. The idea, novel at the time (though less so now, as other tools have adopted a similar approach), was that a template parser that understood both HTML and template tags could generate a tree structure that a data-binding engine could later use to manipulate the DOM with surgical precision. The developer need do nothing more than occasionally provide new data.

This isn't the virtual DOM diffing technique used by React.js and other similar libraries. That approach has some deeply interesting properties, but data-binding – i.e., updating the parts of the DOM that are known to correspond to particular values that have changed, rather than re-rendering everything and not updating the bits that *haven't* changed – is typically a great deal more performant.

Since then, Ractive has added (and in some cases pioneered) many new features: a component system, declarative animations and transitions, full SVG support, encapsulated CSS, server-side rendering, and more. In terms of mindshare, we're a minnow next to the likes of Angular, Ember, Meteor, and React, even though we have contributors from all around the world and Ractive is used for all kinds of websites, from e-commerce to enterprise monitoring software.

But the thing the team and I are most proud of is the way that it has allowed less-experienced developers to bring their ideas to life on the web.

A magazine article is a suboptimal place for code samples demonstrating an interactive UI library, but if you're curious you should visit http://learn.ractivejs.org for an interactive tutorial.

## LESSONS LEARNED

The question "will this make it easier or harder for novice developers to get started?" is always on our minds when we're building Ractive. Interestingly, we've never found that this has required us to sacrifice power for more experienced developers – there's no "dumbing down" in software development, only clear APIs versus convoluted APIs. By focusing on the beginner experience, we make life better for all of our users.

Over the years, we've distilled this mindset into a toolmaker's checklist. Some of these points are, frankly, aspirational. But we've found them to be useful guidelines even when we fall short, and they apply to tools of all kinds.

## README-DRIVEN DEVELOPMENT

Often, when we write code designed to be used by other people, we focus on the implementation first, then slap an interface on it as a final step. That's natural – figuring out the right algorithms and data structures is the interesting part, after all – but completely backwards.

When the API is an afterthought, you're going to get it wrong nine times out of ten. The same is true of the implementation, but there's a crucial difference – you can fix a lousy implementation in a subsequent release, but changing an API means breaking everyone else's code and thereby discouraging them from upgrading. (Worse, you could try to accommodate both the old and the new API, printing deprecation warnings where necessary, and causing Zalgo to appear in your codebase as a result. I speak from experience.)

Instead, try to write the first draft of your README, code samples and all, before writing any code. You'll often find that doing so forces you to articulate the problem you're trying to solve with a great deal more clarity. Your starting vocabulary will be richer, your thoughts will be better arranged, and you'll end up with a more elegant API.

The Ractive API for getting and setting data is a case in point. We were very clear that we wanted to allow users to use plain old JavaScript objects (POJOs), rather than insisting they wrap values in a Ractive-specific observable class (think `Backbone.Model` or `ko.observable`). That posed some implementation challenges, but it was unquestionably the right move. We're currently in the process of overhauling the internal architecture, which will deliver significant performance boosts to many users without breaking their apps.

The phrase "Readme-driven development" was coined, or at least popularized, by Tom Preston-Werner[9].

### ELIMINATE DEPENDENCIES

Dependency management in JavaScript is a pain, even for experts – especially in the browser. There are tools designed to make the situation easier, such as Browserify and RequireJS (or Webpack, Esperanto, and JSPM, if you're part of the revolutionary vanguard), but they all have steep learning curves and sometimes go wrong in ways that are spectacularly difficult to debug.

So the silent majority of developers use the tried-and-tested solution of manually adding `<script>` tags. This means that libraries must be included on the page *after* their dependencies (and *their* dependencies, and so on). Forgot to include underscore.js before backbone.js? Here you go n00b, have a cryptic "Cannot read property 'extend' of undefined" error.

Often, the dependencies aren't actually necessary – it's incredibly common to see libraries depend on jQuery for the sake of one or two easy-to-implement methods, for example. (Yes, it's probably already on the page. But which version?) When they *are* necessary, library authors should provide a version of the library with dependencies bundled alongside the version without. Don't worry about potential duplication; that's the least of our worries at this stage.

### DON'T OVER-MODULARIZE

Since the advent of node.js and npm, a vocal group of developers has evangelized the idea that code should only be released in the form of tiny modules that do very specific jobs. This is at least part of the reason npm has more packages than any other package manager.

On the face of it, this seems like an excellent idea, and a good way to cut down on the amount of imported-but-unused code in an app or library. But the end result is that the burden of thinking rigorously about architectural questions is pushed from toolmakers to app authors, who must typically write large amounts of glue code to get the various tiny modules to talk to each other.

No-one is going to build the next jQuery, because they would instantly be subjected to modularity shaming (an excellent phrase coined by Pete Hunt, formerly of the React.js team). And that's a crushing shame, because it means that we won't have any more libraries with the same level of learnability and philosophical coherence.

In case you think I'm overstating things, there is literally a package on npm called "no-op." Its source code is as follows:

```
module.exports = function noop() {}
```

It has had three releases. It has a test suite! At least it doesn't use Travis-CI for continuous integration, unlike the "max-safe-integer" package, which exports the number 9007199254740991. These packages are not jokes. They were created unironically by leading members of the JavaScript community.

Tiny modules can be just as bad as monolithic frameworks. As usual, there's a happy medium that we should aim for.

### UNIVERSAL MODULE DEFINITION (UMD)

Speaking of modules, you should ideally make your code consumable in as many different ways as possible. The three most common formats are AMD (used via RequireJS and its various clones),

CommonJS (used in node.js, or via Browserify), and browser globals.

The Universal Module Definition lets you target all three of these environments. There are a few different versions, but the basic pattern is this:

```
(function (global, factory) {
   typeof exports === 'object' && typeof module !== 'undefined' ? module.exports = factory()
:
   typeof define === 'function' && define.amd ? define(factory) :
   global.MyLibrary = factory()
}(this, function () {
   var MyLibrary = {};
   /* some code happens… */
   return MyLibrary;
}));
```

The first part detects a CommonJS environment, the second detects AMD, and if neither of those is found it falls back to creating a browser global.

### PROMINENT DOWNLOAD LINKS

It goes without saying these days that if you want to release an open source library, it should exist in a public VCS repository (GitHub being the de facto standard) and be published to npm. Both those are true, but it's important to have a download link available for users who aren't comfortable using git or npm, or who want to quickly try out a library without rigging up a new project with a package.json and a build step.

This needn't involve lots of manual labor or complex automation (though it's straightforward to set up with services like cdnjs.com). One easy way to provide a download link is to include the built library in the GitHub repo (e.g., dist/my-library.min.js) and tag specific commits so that it's easy to link to specific versions:

```
# create the dist files (npm run is a great task runner!)
npm run build

# create a version 0.2.0 tag and add it
# to the 'releases' tab on the repo
git tag -a v0.2.0 -m 'version 0.2.0'
git push origin v0.2.0
```

### GOOD ERROR MESSAGES

Error and warning messages will never be a source of joy, but they can at least be a source of enlightenment. A well-crafted error message is worth pages of documentation, because it appears exactly when the developer needs it.

On the Ractive team, we decided a few months ago that we were doing more harm than good by trying to shield developers from their mistakes. Now, we print verbose warnings to the console

explaining how they can guard against common bugs and make their applications more performant. (This can be disabled if the developer so wishes.) Where it makes sense, we include links to relevant documentation inside error messages. In most browsers, these turn into clickable hyperlinks.

At one stage, we had a class of bugs that were very difficult to unravel. We didn't know quite what was causing the problem, but we were able to detect the state that gave rise to it, so we started throwing errors when that state was reached that included a friendly "please raise an issue with a reproduction!" message, linking to our issues page. Users felt empowered to do something about what would otherwise have been a highly frustrating experience (in some cases becoming first-time GitHub contributors), and we gathered the test cases we needed to solve the bug.

### AVOID THE COMMAND LINE

This guideline only really applies to browser-based tools, but it's an important one: if your introductory instructions involve using the command line, you've already lost half your audience.

That might sound hyperbolic unless you've spent a lot of time with novice developers. But try to remember how *lost* you felt the first time you opened the terminal. GUIs make the things we're working with – folders and files and drives and servers – into almost physical, tangible things that our brains are well-evolved to understand, whereas the command line forces you to build a complex mental model.

Have you ever taken a wrong turn on the way to the restroom and ended up backstage? That's how most people feel when they open a terminal window – like they're behind the curtain, and not in a good way.

### EXAMPLES, EXAMPLES, EXAMPLES

Inviting people to consult the API documentation is polite developer-speak for "RTFM," but no-one wants to read the "fine" manual. What people really want – especially people who aren't yet experts in your domain, and haven't developed the right mental vocabulary – are examples.

I can't articulate it any better than Mike Bostock, the creator of d3[4], so I won't try. Instead I'll just recommend his article "For Example"[3]. The proliferation of copy-and-paste-able examples is one of the main reasons for d3's massive success.

### ELIMINATE JARGON

Naming things is hard, so don't bother. As far as possible, stick to vocabulary that people are already familiar with (but don't make any assumptions about prior knowledge). Favor the slightly wordy but universally comprehensible over terse jargon.

You might need a more complex vocabulary to describe the primitives *inside* your tool, but the less you force your users to become familiar with it, the better.

### EMPATHIZE

While this is the most nebulous item on the checklist, it's also the most important. The motivation to go the extra mile, and try to help people you don't know get the most out of your open source software, springs from empathy.

If your empathy reserves need a top-up, try reading a paper in a field with which you're unfamiliar. For most mortals, reading the *Communications of the ACM* front to back should suffice; you, dear reader, may need something stronger. Try Papers We Love[8]. The bewilderment you feel

closely matches that of the average human trying to learn web development – or, for that matter, a highly experienced developer coming to your domain of expertise for the first time.

## WE HAVE TO BUILD THE FUTURE WE WANT

It's depressingly common to hear people suggest that the increasing complexity of the web platform is inevitable, the price we pay for progress. This is a classic self-fulfilling prophecy – once we decide that it's true (or worse, *right*) that web development is best left to the professionals, we'll stop striving to make it more accessible for everyone else.

This would be a tragedy of the highest order were it to come to pass. The web has been a gateway drug for an entire generation of programmers (your present correspondent included), many of whom would never have otherwise experienced the sheer joy of computer science. There's no intrinsic reason it can't continue to be. But it's down to us: we have to choose to build a web that is accessible to everyone.

## REFERENCES


1. http://backbonejs.org
2. Bloomberg, M. 2012. https://twitter.com/mikebloomberg/status/154999795159805952
3. Bostock, M. 2013. For example. http://bost.ocks.org/mike/example/
4. http://d3js.org/
5. Glowacki, T. 2015. Comment on "Will there be continued support for people that do not want to use Ember-CLI?" http://discuss.emberjs.com/t/will-there-be-continued-support-for-people-that-do-not-want-to-use-ember-cli/7672/3
6. Koch, P.-P. 2015. Tools don't solve the Web's problems, they are the problem. http://www.quirksmode.org/blog/archives/2015/05/tools_dont_solv.html
7. Markbåge, S. 2015. Tooling is not the problem of the web. https://medium.com/@sebmarkbage/tooling-is-not-the-problem-of-the-web-cb0ae1fdbbc6
8. http://paperswelove.org/
9. Preston-Werner, T. 2010. Readme driven development. http://tom.preston-werner.com/2010/08/23/readme-driven-development.html
10. http://ractivejs.org


**LOVE IT, HATE IT? LET US KNOW**

feedback@queue.acm.org

**RICH HARRIS** is an award-winning interactive journalist at theguardian.com, where he uses web technologies to tell stories in new ways through interactivity and data visualization. He is the creator and lead author of a number of open source projects designed to make web developers' lives easier, and an occasional speaker at technology conferences. Originally from the north of England, he lives in Brooklyn with his wife Emma. He can occasionally be found rambling incoherently at twitter.com/rich_harris.