Thoughts on the web: the history, where we are and how we move forward

If you've been doing web development for any length of time you can see how much the web and its component technologies have evolved over time. I think it's interesting to see how the web and its component technologies have evolved since their creation and where can we go from there. The main question these reflections seek to answer is: ***Is the basic stack of the web: HTML, CSS and Javascript solid enough to build web applications?***

Is the web stack still solid?

The web has managed to survive 25 years and has evolved and endured fads and trends unscathed but not unchanged. Let's look at some of these changes. These vignetes are not meant as exhaustive histories of each element. Where possible, I provide links to (more) complete histories of the technologies. They are meant to highlight some aspects of the technologies and where I see problems and room for growth.

HTML

HTML, perhaps unsurprisingly, is the piece of the web that has gone through the most signigicant changes in the last 25 years.

The language has evolved though different versions and has resulted on the HTML we take for granted today. HTML endures because it's markup, not a programming language. The number of elements that you must know to create a well-formed HTML document is rather small and it hasn't changed much since the early days.

A basic document conforming to the 1997 HTML 3.2 specification looks like this:

```
<!DOCTYPE HTML PUBLIC
```

```
"-//W3C//DTD HTML 3.2 Final//EN">
<HTML>
  <HEAD>
    <TITLE>Title of the document</TITLE>
  </HEAD>
  <BODY>
    <H1>Level 1 heading</H1>

    <P>A paragraph of text
    <P>Another paragraph
  </BODY>
</HTML>
```

The same document expressed as XHTML 1.0 looks like this. Note the XML prologue and the way that uses an XML namespace to indicate the default name space of the document.

```
<?xml version="1.0" encoding="UTF-8"?>
<html xmlns="http://www.w3.org/1999/xhtml">
  <head>
    <title>Title of the document</title>
  </head>
  <body>
    <h1>Level 1 heading</h1>

    <p>a paragraph of text</p>
    <p>anoter paragraph</p>
  </body>
</html>
```

And the same document conforming to the current WHATWG HTML specification, and the corresponding HTML 5.3 W3C spec, looks like this as a comparison.

```
<!DOCTYPE html>
<html lang="eng">
  <head>
```

```
    <meta charset="utf-8">
    <title>Title of the document</title>
  </head>
  <body>
    <h1>Level 1 heading</h1>

    <p>a paragraph of text</p>
    <p>anoter paragraph</p>
</body>
</html>
```

For the sake of clarity I will refer to the latest HTML specification from the WHATWG, rather than the W3C because it updates more frequently.

> HTML introduces new structural elements:
>
> - `section` represents a generic document or application section
> - `article` represents an independent piece of content of a document, such as a blog entry or newspaper article
> - `main` represents the main content of the body of a document or application
> - `aside` represents a piece of content that is tangetially related to the rest of the page
> - `header` represents a group of introductory or navigational aids
> - `footer` represents a footer for a section and can contain information about the author, copyright information, etc
> - `nav` represents a section of the document intended for navigation
> - `figure` represents a piece of self-contained flow content, typically referenced as a single unit from the main flow of the document and the optional `figcaption` can be used as caption for the figure element
> - `template` can be used to declare fragments of HTML that can be cloned and inserted in the document by script

The other interesting set of new elements in HTML5 are media and media related elements:

- `video` and `audio` for multimedia content
  - one or more `source` elements for multiple streams of different types
  - zero or more `track` elements to provide captions, subtitles or other text tracks for the video element

- canvas is used for rendering dynamic bitmap graphics or WebGL-based experiences such as graphs or games

For more detailed information see the W3C's HTML5 Differences from HTML4.

Even with all these new tags available in the latest version of HTML the criticims remains that new elements are too slow to be incorporated into the standard and that, as Dave Rupert writes in Why <details> is Not an Accordion:

> At the risk of being a broken record; HTML really needs <accordion> , <tabs>, <dialog>, <dropdown>, and <tooltip> elements. Not more "low-level primitives" but good ol' fashioned, difficult-to-get-consensus-on elements. A new set of accessible controls for a modern era... except that these things have been in-use on nearly every major website and application for the last two decades and exist in every major design system.

Unfortunately, while I may agree with the sentiment there is also a big communication issue at work here. How do we agree on what should be in a new element? How do we agree what should be the default and what shoud be left for developers to add? How long is it OK for a feature to remain under development?

The problem is that, unless and until we get agreement about the shape these elements will take, we will continue to have versions of these elements in all frameworks and design systems.

Google tried to create a toast element using Blink's process for launching new features as with any other feature that has been released in Chrome for the last few years.

Because the initial communication was labeled as an ***Intent to ship*** (rather than an ***Intent to prototype***) people assumed this was Google just adding things to the web platform without solicitying input from people outside the company. As you can imagine, the idea had a poor reception (see Adrian Roselli, Dave Cramer and Terence Eden opinions of the new element) and it sparked a lot of discussion and debate among developers, most of the pieces critical of Google's approach.

So, while the language itself is healthy and has room for growth, the shape of this growth concerns me. What will it take for these new elements to reach

consensus when any one group can decide they are not going to implement it and break any consensus that may have been reached?

Releasing these standard elements like `toast`, `kv-storage`, or `virtual-scroller` as libraries defeats the purpose of wanting to incorporate them on to the platform directly; they add load time and bloat to Javascript and puts the onus on developers to incorporate them into their projects rather than have them baked into the browser so you only need to reference them, possibly with a fallback.

It is interesting that after pushback against toast and the idea of a standard library for the web, Google as a team, has pulled back from such projects: `kv-storage` and `virtual-scroller` are no longer under development.

The idea for a standard library is now being criticized by some of the people who used to champion it. See Domenic DeNicola's comment on the W3C TAG Design questions around pay-for-what-you-use issue

On the other hand, since most large companies have released their design systems and components as open source project nothing stops developers from adopting any of them if they like them... you're just locking yourself in to whatever the design system says it's OK.

Web components are a good opportunity to experiment with new elements and how they would work on browsers without having to wait for standard organizations to implement them.

This may be a good use for Web components. They offer a good opportunity to experiment with new elements and how they would work on browsers without having to wait for standard organizations to implement them.

Does CSS still hold?

CSS has evolved in interesting and different ways over the years. From a single specification to the current family of specifications and APIs covering a much broader spectrum of functionality that we ever thought was possible.

If you've worked on the web for a while you will remember the different types of layouts that we've worked with over the years. From no layout before CSS to tables, to floats to flexbox to grid and all the ancillary libraries that were needed to get the layouts we wanted; for the longest time we stayed in the same layouts and schemas but that's starting to change.

The introduction of Flexbox and CSS Grid presents new horizons to explore and new layout possibilities to use.

Flexbox and CSS Grid are unlikely to eliminate existing workarounds and tooling that makes use of these workarounds. If there is no need, why make changes that would update our code? The old adage “if it ain’t broken, don’t fix it” seems to hold particularly true for the web.

But we can hope things will change.

Bootstrap 5.1 introduced an experimental grid layout based on CSS Grid based on CSS Grid rather than Flexbox or floats like prior versions. Bootstrap still commands a respectable percentage of web usage (27.2% according to W3Techs) so, even if they are low-traffic sites it 's likely that the new CSS Grid in bootstrap will be adopted by existing sites in addition to new sites that adopt the library. It’s just a matter of waiting to see if it happens and how long it takes.

If you’ve worked with the web for a while you will also remember one of its worst ideas: vendor prefixes. In older web pages you would see code like this:

```
.myClass {
  -webkit-transition: all 1s linear;
  -moz-transition: all 1s linear;
  -ms-transition: all 1s linear;
  -o-transition: all 1s linear;
  transition: all 1s linear;
}
```

Where each line before the final one was a vendor-specific version that may or may not have had the same syntax or worked the exact same way. The last, unprefixed, version is the standard way according to the specification. The prefixes are:

- `-webkit-`: Safari, iOS Safari / iOS WebView
- `-moz-`: Firefox
- `-ms-`: Edge before adopting Chromium, Internet Explorer
- `-o-` Opera before adopting Chromium, Opera Mini
- Chromium (Chrome, Android, Edge, Opera), has not implemented vendor prefixes in blink but still uses older `-webkit-` prefixes

Initially, vendor prefixes were developed as a way to test new proposals and features without affecting what the feature would look like in a finalized form. Browser makers used the honor system and assumed that once a feature was finalized, developers would move away from prefixes and into the finalized version.

But it didn't happen.

Because different versions of different browsers implement features at different paces we couldn't remove the prefixes and, after a while, enough people were using them that it wasn't possible for browser makers to remove them without breaking sites.

CSS is not a complete programming language. Preprocessors like SASS/SCSS or LESS and tools like PostCSS have done a lot to "enchance" what we can do with CSS. Nesting, automating the addition of prefixes where necessary and some basic programming-like constructs have become a part of developer's CSS workflows. The following SCSS code works with multiple elements simultaneously:

```
$base-color: 663399;

@for $i from 1 through 3 {
  ul > li:nth--of-type(n + #{$i}) {
    background-color:
      lighten($base-color, $i - 5%);
  }
}
```

We'll set aside the fact that you need a compiler to turn the SCSS code into CSS that a browser can read, that's a whole other issue to be discussed later.

CSS is not stagnating.

In addition to new layout features and the ability to use containment as the basis for container queries (see Next Gen CSS: @container) and the Houdini APIs that enhance and supplement existing elements and APIS and allow developers to hook into the browser's rendering engine lifecycle to do custom work in a performant way.

Although they are at different stages of maturity, some of them are already

proving themselves to be beneficial. The two Houdini APIs I find most useful are the CSS Typed Object Model which allow for optimizations of CSS when used with Javascript and the CSS Properties and Values API Level 1 that allows CSS variables to have a type, a default value and choose whether children inherit the value or not. Houdini CSS Propertie and Values allows you to create custom properties using the following CSS syntax in supported browsers:

```
@property --my-color {
  syntax: '<color>';
  inherits: false;
  initial-value: #C0FFEE;
}
```

This is currently supported only on Chromium browsers (Chrome, Edge, Opera and others). Firefox and Safari have not implemented it and there's no signal that they will do so in the future.

Another API that's worth taking a look at is CSS nesting editor's draft. It's true that no browser currently implements this specification but the very fact that it exists is important. It means that the CSS Working Group has deemed nesting worth a look and eventual publication as a W3C Recommendation.

CSS, more so than other web technologies, needs implementation and adoption to promote further adoption. The more developers see a technology in production sites, the more they will implement the technlogy themselves.

Javascript

Javascript is one of those languages that developers either love or hate with equal passion.

Javascript was hampered from its inception by conflicting priorities. Its creator, Brendan Eich, was given ten days to complete it and was supposed to be an auxiliary language to Java, meant to be run on the client using Applets.

Some of the earliest Javascript I wrote were image roll over scripts where we display one image when the page loads and replace it with a different image when the user mouses over the image.

The script sets two identical functions that change the source of the image to

the one specified in the second parameter.

```
function move_in(img_name,img_src) {
  document[img_name].src=img_src;
}

function move_out(img_name,img_src) {
  document[img_name].src=img_src;
}
```

We would then use in HTML like this:

```
<a href="link.html"
  onMouseOver="document.image1.src='onImage.gif'"
  onMouseOut="document.image1.src='outImage.gif'">

  <img src="outImage.gif" name="image1">
</a>
```

Since these early scripts there has been evolution and new features both in terms of the language itself and in terms of what we can do with it.

The following table shows a brief synopsis of the history of Javascript

Version	Formal Name	Released On	Description
1.X	Javascript 1.X	1995	First version of the language. First released in Netscape Navigator 2.0
ES1	ECMAScript 1	1997	Standardized existing Javascript practices. Equivalent to Javascript 1.3
ES2	ECMAScript 2	1998	Editorial changes. Equivalent to Javascript 1.3
ES3	ECMAScript 3	1999	Added regular expressions and try/catch blocks. Equivalent to Javascript 1.5

Version	Formal Name	Released On	Description
ES 3.1	ECMAScript 3.1 "Harmony"		Additions to the ES3 syntax that were agreed to by the committee and are more modest than what ES4 would have required
ES4	ECMAScript 4		Not released and version skipped
ES5	ECMAScript 5	2010	Added `strict mode`, JSON support, `String.trim()`, `Array.isArray()`, and Array iteration methods. Think of it as a compromise between the massive changes of ES4 and a more gradual evolution of ES3 and 3.1
ES6	ECMAScript 2015	2015	Added `let` and `const`, default parameter values, `Array.find()`, and `Array.findIndex()`
ES7	ECMAScript 2016	2016	Added exponential operator and `Array.prototype.includes`
ES8	ECMAScript 2017	2017	Added string padding, `Object.entries`, `Object.values`, async functions, and shared memory
ES9	ECMAScript 2018	2018	Added rest / spread properties, asynchronous iteration, `Promise.finally()`, and RegExp
ES10	ECMAScript 2019	2019	
ES11	ECMAScript 2020	2020	
ES12	ECMAScript 2021	2021	
ES13	ECMAScript 2022	2022	

The most interesting aspect the history of Javascript is the ten year gap between ES3 and ES5 and the fact that there's no ES4 in the books. What happened?

There was strong disagreement over what direction should the language take.

The big idea for ES4 was to become more useful for programming beyond

what Javascript was capable of at the time. See Valdermar Horvath's JavaScript 2.0: Evolving a Language for Evolving Systems for an idea of where some people wanted to take ES4. Other members of the committee wanted a more gradual improvement of the language without the massive changes that were part of the ES4 proposal.

In an email titled ECMAScript Harmony Brendan Eich outline the solution that came out of the committe. He writes:

> The committee has resolved in favor of these tasks and conclusions:
>
> 1. Focus work on ES3.1 with full collaboration of all parties, and target two interoperable implementations by early next year.
> 2. Collaborate on the next step beyond ES3.1, which will include syntactic extensions but which will be more modest than ES4 in both semantic and syntactic innovation.
> 3. Some ES4 proposals have been deemed unsound for the Web, and are off the table for good: packages, namespaces and early binding. This conclusion is key to Harmony.
> 4. Other goals and ideas from ES4 are being rephrased to keep consensus in the committee; these include a notion of classes based on existing ES3 concepts combined with proposed ES3.1 extensions.

Several features from ES4 became part of ES5 and later versions of the standard so not all development work on ES4 was a loss.

The TC39 work itself became less cumbersome and adopted an annual release cadence for new editions of the language starting with ES2015 (ES6) and a process document that outlines the different stages of a feature from strawman to shipped.

Part of the requirements for stage four, the final step of a released fature, requires two interoperable implementations before they are added to the next edition of the specification. So we know at least some browsers will implement new features for current versions of Javascript but if there's consensus from all participants in the committee that the feature is worth implementing, we can only

hope that all the browser makers will implement it in a timely fashion.

Third party tools libraries and module systems

Libraries like jQuery, MooTools, the Dojo Toolkit came about as ways to smooth out the differences between browsers and as ways to enchance the Javascript developer experience.

jQuery, in particular, did a great job to smooth out the way browsers impmlemented features so you only had to write code once. They also implemented features that have now become part of the standard Javascript vocabulary. Yes, we have jQuery to thank for `querySelector`, `querySelectorAll`. You might not need jQuery shows equivalencies between jQuery code and code needed to support IE8, IE9, and IE10

We take them for granted today, but modules were not a part of the ECMAScript standard until 2015, leaving developers to adopt their own version of modules (AMD and CommonJS), leading to incompatibilities and the pain of modules in Node today.

But as the requirements for web applications and the demands from users have evolved so have frameworks and libraries evolved to match all through the Javascript ecosystem, from build system to front-end frameworks we keep seeing more and more ways to do the same task and have to choose which one we're most comfortable with.

Everywhere in the stack we are required to make choices and, sometimes, the sheer amount of choices developers must make can be overwhelming. We should be careful how we present these choices and how we provide alternatives when it's possible to do so.

It wasn't too long after Javascript was introduced that people started working on addressing the differences between browser implementations; this allowed developers to write code once and have it work in all browsers without you having to do anything special.

Transpilation is also another big thing in Javascript and its evolution. I group transpilers and compilers using Javascript as a compile target into four categories:

- Supersets of Javascript
    - Typescript
    - Coffee Script

- Compiling new features to current Javascript
    - Traceur
    - Babel
- Using javascript as a target compilation language
    - Dart
    - ClojureScript
    - Scala.js
- Any language that can be compiled to asm.js or WebAssembly (they may or may not be the same languages)

So now we have multiple ways to create content for the web. You accept the limitations of these technologies (more on that later).

Babel has moved from being a transpiler of modern Javascript to ES5 to being a testbed for features at stage two, three and four in the TC39 process. This way implementors and developers can provide feedback on the specification without waiting for browsers to implement it.

Furthermore, we can use babel to help us work with a sane ES2017-based Javascript baseline. I described one possible method in Baseline for Javascript development: modules in the browser without transpilation. Credit where credit is due, the idea for the post came from Publish, ship, and install modern JavaScript for faster applications

asm.js was the first attempt to create a strict subset of Javascript as a compilation target for programming languages allowing them to run on the web without plugins.

Using C/C++ as an example, you can tools like Emscripten to compile the C codebase into asm.js and then run the resulting code, plus some additional Javascript glue code, on the browser without plugins.

Web Assembly takes the ideas of asm.js even further. Rather than compiling to a subset of Javascript, WebAssembly (Wasm) compiles to a binary format that aims to be safe, language neutral and used in and out of the web.

Autodesk has ported AutoCAD to the web using Wasm. The following presentations from WebAssembly SF covers the process to create the web version of Autocad in more detail.

So Javascript is ubiquitous but it's no longer the only way to create content for the web. It's highly unlikely that WebAssembly apps will fully replace Javascript

since that's not what Web Assembly was designed for. According to the WebAssembly FAQ:

> **Is WebAssembly trying to replace JavaScript?**
>
> No! WebAssembly is designed to be a complement to, not replacement of, JavaScript. While WebAssembly will, over time, allow many languages to be compiled to the Web, JavaScript has an incredible amount of momentum and will remain the single, privileged (as described above) dynamic language of the Web
>
> Furthermore, it is expected that JavaScript and WebAssembly will be used together in a number of configurations:
>
> - Whole, compiled C++ apps that leverage JavaScript to glue things together
> - HTML/CSS/JavaScript UI around a main WebAssembly-controlled center canvas, allowing developers to leverage the power of web frameworks to build accessible, web-native-feeling experiences
> - Mostly HTML/CSS/JavaScript app with a few high-performance WebAssembly modules (e.g., graphing, simulation, image/sound/video processing, visualization, animation, compression, etc., examples which we can already see in asm.js today) allowing developers to reuse popular WebAssembly libraries just like JavaScript libraries today
> - When WebAssembly gains the ability to access garbage-collected objects, those objects will be shared with JavaScript, and not live in a walled-off world of their own.

Language version adoption versus usable features set and transpilation

While it's nice to have smaller, more predictable ECMAScript / Javascript releases it is not without its drawbacks. With anual releases we are presented with new features to consider for adoption and the consequent decisions developers have to make. They have to choose whether to use the latest language features and transpile them to older versions for wider support, or use older features that are more widely supported in the browsers on your users' computers and, as a result,

wouldn't require compilation?

In an ideal world, we would be able to use the newest language features as soon as they reach stage three in the TC39 process.

But we don't live in an ideal world. Some browsers have a much slower cadence of releases so, even if a new Javascript feature is implemented as soon as it reaches stage three, it may still be months before it's available in all production browsers.

Do we need tooling? Yes and no

It used to be that you could just put your HTML, CSS and Javascript on a server and let people visit the content. But that has changed significantly; A significant chunk of what we do for the web today requires tooling, whether we like it or not. Some items that required tooling:

- Javascript transpiling from React or other frameworks to ES5+
- CSS preprocessing with SASS, LESS or PostCSS
- Bundling Javascript with WebPack, Rollup, Parcel or others

Each of these steps presents further choices. Do we run each tool individually or do we automate the proccess? What tool do we use to automate the process with?

As with many other parts of the Javascript ecosystem, every step of choosing a build system is a tradeoff and switching means dealing with the same tradeoffs each time and each step of the way is littered with these types of choices. Here are some of the questions I've had to answer along the way:

- What task runner will you use if you have tasks that you run every time you build your site or app
- If you want to bundle your code, do you use WebPack or Rollup? (at the time I asked this question, those were the two most popular options)
- Are you including data visualizations in your project so what library will you use to visualize the data?
- Are you building a WebGL App? Do you use raw WebGL code or do you use a library? if you choose to work with a library, which one do you use?
- Do you want to use WASM to improve your app's performance? If you're porting an existing library then this is done for you but if you're writing your own code then what language will you write the code that you will port to WASM?

But things are changing and we are moving forward. Unless we are doing React or SASS or other specific things that require it, we don't need to bundle or transpile code and bundle code.

HTTP 2 makes it easier to ship unbundled code. It uses multiplexing to use a single connection for multiple requests and assigns IDs to each request so there won't be an issue if the server has to run the request again. It is not a perfect solution (HTTP 3 is already in the works and almost ready for release) but it's a start.

Depending on our target population and the features that we need to support, we might not need to transpile our code at all. See Publish, ship, and install modern JavaScript for faster applications for one way to creaate multiple bundles of code based on what a browser supports and what we want to use as developers.

Thoughts on where we are and how we may move forward

These are some thoughts based on the research for the first part of the post that reflect on some trends and technologies and where we may go moving foward.

Give the users footguns or protect them from themselves?

A 'footgun' is something you give a person that allows them to shoot themselves in the foot. In software the term is used to describe features that may be potentially dangerous if missued, either intentionally or accidentally.

For the context of this conversation we'll refer to the following APIs, available in Chromium browsers but opposedby Mozilla and Apple:

- Web Bluetooth — Allows websites to connect to nearby Bluetooth LE devices
    - Mozilla's position
- WebHID — Allows websites to retrieve information about locally connected Human Interface Device (HID) devices
    - No Mozilla position I could find regarding WebHID. I expect their position to be similar to those regarding Web Serial and Web USB

- Web Serial API — Allows websites to write and read data from serial interfaces, used by devices such as microcontrollers, 3D printers, and othes
    - Mozilla's position
- Web USB — Lets websites communicate with devices via USB
    - Mozilla's position

In New Standards to Access Device Hardware using JavaScript the author writes about the HID interfacce and briefly touches on Chromium's security considerations for poweful APIs that, apparently, are not enough for Mozilla and don't reduce Apple's privacy concerns.

While Apple declined to implement 16 Web APIs in Safari due to privacy concerns they have equivalent APIs for their native applications, have tight control of the only allowed App Store for iOS/iPadOS, and restrict other browsers to using their own rendering engine (which lags significantly behind on key web app features, PWA features, and properly implemented standards), it opens more questions than it answers.

Apple's concern for device fingerprinting is important but it doesn't stop fingerprinting libraries like Clientjs or fingerprintjs from working. The source for both of these libraries is on Github so we can look at how to ameliorate fingerprint generation without reducing current APIs or refusing to implement new ones.

Alex Russell presents an interesting analysis of the issue in Platform Adjacency Theory, particularly footnote 3.

There is always a risk in being the first one to implement a technology and, despite the process in place to incubate new features, there is an inherent distrust of Google as the dominant force in the browser market today so it's easy to think the worst of a company introducing a feature that is not part of a specification or a standard but it's intended to be part of one. In this case it's the concept of layered APIs, the high-level counterpart to the extensible web manifesto, that takes advantage of lower-level APIs to produce tools for the web platform.

Coming back to the APIs we're talking about, how much risk is too much? How much do you need to educate and communicate with developers and end users of the APIs in question?

Are these APIs as big a security risk as they are painted to be? Can we trust developers to be responsible in how they use them?

That's part of the learning involved in releasing these APIs and features; how

much do we need to train users and how powerful should implementors make these features to make them useful without making them dangerous.

Part of this process is to run Origin Trials for these new features.

Origin trials address the need to have developer feedback without baking the feature into the web platform because of overuse.

With Origin Trials developers are able to register for an experimental feature **for a specific origin for a fixed period of time (measured in months)**. In exchange, they agree to give feedback once the experiment ends.

The feature being tested is not permanent and ***can be automatically disabled for everyone participating in the trial if usage hits the Chrome deprecation threshold*** (< 0.5% of all Chrome page loads) so we won't be stuck with features because one large user decided to use the feature in production.

So yes, I do believe that we should trust developers and users and make the features available, as long as we follow a reasonable security protocol, for developers to play and experiment with and it should be their decision whether to use the feature in their applications. Worrying about hypotheticals helps no one in making the right decisions.

The Eternal Debate: Web versus Native

Ever since the web became accessible to phone and tablets there has been a debate between the web and native as the technology that developers should pursue.

It would be an easier choice if both the web and native supported the same set of APIs for developers to use.

But they don't support the same APIs and they have differences in how you make software available.

The web has the advantage that it requires zero installation beyond the web browser you use to access it. There is no download to run and no installation to speak of. The process becomes smoother when you make your application into a PWA (Progressive Web Application) that adds push notifications, offline capabilities and will run on most major browsers (Safari is the outlier where PWA won't necessarily work as expected or designed if they work at all in iOS/iPadOS)

As far as capabilities, efforts like Chromium's Capabilities Project (Project Fugu) are good steps in trying to bridge the gap between what you can do on the web versus native and, in the process, make the web a more compelling development platform.

We discussed some of the Fugu APIs (Web Bluetooth, WebHID, Web Serial and Web USB) when we discussed developer footguns but there is also a wider case to be made.

These APIs open new fields and types of experience for web users. There is no real reason why developers ought to maintain multiple code bases for the same product, keep the codebases in sync feature-wise, and have to go through the, some times expensive, store application processes, and it leverages technologies web developers are already familiar with.

One example of the possibilities these APIs open is Henrik Joreteg's work with medical devices that run through serial ports, as documented in Project Fugu: A New Hope.

Another example may be trying to recreate the Physical Web using PWAs and Web NFC or similar, open source, technologies.

But even if we can create these awesome experiences or come up with new ideas for projects no one has thought about, they will only work in Chromium browsers; they are the only ones who have implemented the Fugu APIs.

To some, this may be a lead to the bad old days of the browser wars, where browsers implemented proprietary technologies and left interoperability far behind as a concern and developers had to pick up the slack by branching code depending on the user agent in use. It is also easy to accuse Google and the Chromium project of monopoly by doing things no one else wants to do.

I think this situation is different; the technology is available for everyone to implement but concerns about privacy and fingerprinting (whether true or not) have prevented Firefox and Safari from adopting them even though iOS and iPadOS have native features equivalent to some of the Fugu APIs and have had it since the early days.

I'm willing to give Apple the benefit of the doubt but it's always easier to say ***I don't like something*** rather than say ***I don't like something and here's the alternative I'm proposing instead***. It's also easier to take the high road when

your native platform already has these features and makes them available to app developers. You're just widening the gap between native and the web.

There is no reason why the web can't have feature parity with native platforms like iOS/iPadOS, Android or desktop operating systems.

The web as a craft?

Doing web development used to be simple. All we needed was a text editor and either good knowledge of HTML, CSS and Javascript or bookmarks on your browser pointing to Eric Meyer's CSS resources on the CWRU server and links to the best tutorials you could find about CSS and Javascript with the differences for each browser and how to branch the code to acommodate each browser's idiosyncracies.

Those days are long gone. The web has become too complex for any one person, even a full stack developer, to handle.

Reading David Rupert's The tangled webs we weave reflected some of my own struggles when working with existing projects.

But even when working on our projects, starting a new project forces us to ask many questions before we even get started regarding the tools and technologies that we choose to use.

If you build web content on a regular basis you may have internalized this and have a set of scripts ready to run and generate the skeleton of a website project ready to go with all the tools you know you will use; a lot of the tools that you may consider using already provides these skeletons and starter projects, but it's still up to you to decide if the tool is appropriate.

But what happens to people new to frontend, backend, or full stack development or to people who have left the field for a while and now return to a drastically changed development landscape?

Two interesting counterpoints on this issue are Jessica Joy Kerr's Back when software was a craft and Justin Searls Twitter thread

Because of the diversity of tools available it's very hard to stay in a single tool or set of tools, particularly if you're adopting a project or tool, or are contributing to an existing one.

Consider for front end developers in 2012. Take articles like Rebecca Murphey's A Baseline for Front-End Developers and contrast it with the variety of tools that we have available today.

In Everything Easy is Hard Again Frank Chimero documents one developer's journey back into web development.

> So much of how we build websites and software comes down to how we think. The churn of tools, methods, and abstractions also signify the replacement of ideology. A person must usually think in a way similar to the people who created the tools to successfully use them. It's not as simple as putting down a screwdriver and picking up a wrench. A person needs to revise their whole frame of thinking; they must change their mind.

I experienced this when I first started working with task runners for processing web code. I used Grunt for a year or two but when I decided to use Polymer all the tools (with one or two exceptions) were written in Gulp, a completely different paradigm, and a more programmatic way to write the tasks that you need for your project.

I remember when looking at the source of a page was one of the best ways to learn was to look at the code of a page you liked to see how the markup was structured and how the scripts ran on the page. It was refreshing to see this note in the source code of Jeremy Keith's homepage

```
<!-- Ah, I see you want a peek at the source code. -->
<!-- That's the great thing about the web: -->
<!-- reverse engineering. -->
<!-- If you have any questions, feel free to write to me: -->
<!-- address redacted -->
```

But in our quest to use the best framework or produce the fastest possible web content we've hidden the core of what makes the web work and what makes that little language Tim Berner-Lee created in the late 1980s so powerful.

Links and Resources

Below are links and resources that I used while researching this post.

- Background
    - Information Management: A Proposal — Tim Berners-Lee
        - This is the document that proposes what became the World Wide Web to the powers-that-be at CERN
    - Evaluating Technology — Jeremy Keith | An Event Apart
- HTML
    - HTML Living Standard — WHATWG
    - DOM Living Standard — WHATWG
- CSS
    - SASS
    - CSS-TAG Houdini Editor Drafts
    - Next Gen CSS: @container
- Javascript
    - JavaScript: The First 20 Years — Allen Wirfs-Brock and Brendan Eich
    - TC 39 Process Document
    - Frameworks and libraries
        - jQuery
        - MooTools
        - Dojo Framework
            - Dojo Toolkit 1.x
            - Dojo 2.x and later
    - Alternatives and complementary solutions
        - Typescript
        - Coffeescript
    - WebAssembly
        - WebAssembly.org
        - Awesome WebAssembly Languages
        - C/C++
            - Emscripting a C library to Wasm
        - Go
            - Go and WebAssembly: running Go programs in your browser
            - WebAssembly — Golang Wiki
            - Compiling Go to WebAssembly
        - Rust
            - Why Rust and WebAssembly?
            - Tutorial: Conway's Game of Life
    - Video presentations
        - An introduction to WebAssembly
        - WebAssembly for Web Developers — Surma | Google I/O '19
        - Why the #wasmsummit Website isn't written in Wasm —

Ashley Williams
        - A Cartoon Intro to WebAssembly — Lin Clark | JSConf EU
        - How WebAssembly is Accelerating the Future of Web Development
        - WebAssembly as a video polyfill — Brion Vibber
    - ES4, AS3 and the fallout
        - ECMAScript Harmony — Brenda Eich
        - JavaScript 2.0: Evolving a Language for Evolving Systems — Valdermar Horvath
        - The Macromedia/Adobe angle
            - The Real Story Behind ECMAScript 4 — Sebastian Peyrott from Auth0
            - ActionScript 3 and ECMAScript 4 — Mike Chambers
            - JavaScript Stalled, AS3 Orphaned – Microsoft to Blame? — Grant Skinner
            - Ru Roh! Adobe Screwed By EcmaScript Standards Agreement — Hank Williams
- Give users footguns or protect them from themselves?
    - Criticism pushes the web forward
    - Progress Delayed Is Progress Denied — Alex Russell
    - Apple declined to implement 16 Web APIs in Safari due to privacy concerns
    - Controversial APIs
        - Web Bluetooth
            - Mozilla's position
        - WebHID
        - Web Serial API
            - Mozilla's position
        - Web USB
            - Mozilla's position
    - Controlling Access to Powerful Web Platform Features
    - New Standards to Access Device Hardware using JavaScript
    - Why <details> is Not an Accordion — Dave Rupert
    - Layered APIs
    - Extensible web manifesto
    - Origin Trials Motivationgo and Explainer
    - Platform Adjacency Theory — Alex Russell
    - Publish, ship, and install modern JavaScript for faster applications
- Performance as a core concern of the web
    - The Mobile Performance Inequality Gap, 2021 — Alex Russell
- The eternal debate: Web versus Native

    - Web Capabilities (Project Fugu 🐡) — Chromium
    - Bridging the native app gap — Chrome Dev Summit 2019
    - New capabilities status
    - Fugu API Tracker
    - Project Fugu: A New Hope — Henrik Joreteg
- The web as craft or the web as an assebly line? The tangled webs we weave — David Rupert
    - Back when software was a craft — Jessica Kerr
    - Justin Searls Twitter thread
    - Everything Easy is Hard Again — Frank Chimero
    - What Screens Want — Frank Chimero
    - The Web's Grain — Frank Chimero
    - The web is too damn complex — Robin Rendle
    - The arc of design