PUBLICLY SHARED

Towards a modern Web stack

January 2023 - Ian 'Hixie' Hickson

Introduction

Web pages today are built on 25-year-old technology: a markup language for scientific documents from 1991, a scripting language from 1995 whose first version was implemented "in ten days", a styling and layout model from 1996, and an API from 1997 whose initial design was based on combining the independent inventions of two teams with little regard to the developer experience. This stack has evolved over the past few decades and is now quite convoluted, but remains, at its core, a system based on a markup language, a scripting language, and a styling language all of which could be politely described as "quirky".

This isn't a priori a problem, but it presents an opportunity: the web could be modernized for a better developer and user experience[a].

The bulk of the web stack consists of high-level primitives that are expensive and difficult to configure. For example, the web stack assumes a vertical layout (and scrolling) model by default, which makes vertical centering more difficult than horizontal centering. The built-in widgets are limited in their configurability and so new widgets must be created using markup and layout primitives that were not originally designed to enable this. Scripts must be written in JavaScript. The input APIs assume certain gestures.

Developers have, over the years, developed techniques to work around these limitations[b][c][d][e][f], but this remains a challenging and frustrating task. As a result, users continue to suffer applications that are not quite what the developer intended (for example, it is common for a stray interaction to accidentally cause all text on a web page, including widgets, to be selected, something that would never happen on other platforms).

By providing low-level primitives instead, applications could ship with their own implementations of high-level concepts like layout, widgets, and gestures, enabling a much richer set of interactions and custom experiences with much less effort on the part of the developer.

As it happens, among the many high-level primitives, a few low-level APIs have been created and are available on the web today:

- WebAssembly (also known as Wasm) provides a portable compilation target for programming languages beyond JavaScript; it is being actively extended with features such as WasmGC.

- WebGPU provides an API (to JavaScript) that exposes a modern computation and rendering pipeline.

- Accessible Rich Internet Applications (ARIA) provides an ontology for enabling accessibility of arbitrary content.

Towards a modern Web stack (PUBLICLY SHARED)

This document proposes to enable browsers to render web pages that are served not as HTML files, but as Wasm files, skipping the need for HTML, JS, and CSS parsing in the rendering of the page, by having the WebGPU, ARIA, and WebHID APIs exposed directly to Wasm.[g][h][i][j] To enable developers to continue to use the wider range of APIs exposed on the web, a mechanism to "escape" to a JavaScript environment would need to be made available as well.

Opportunity

These APIs exist today, but require bootstrapping via JavaScript and require all Wasm use of these APIs to be done via JavaScript trampolines. The opportunity therefore is around simplifying the experience, and likely improving the performance as a result.

Proposal

Wasm

This proposal assumes a successful deployment of WasmGC, to extend the range of viable source languages to include Dart and Kotlin.

Bootstrapping

Loading a Wasm binary directly should cause the browser to compile and instantiate the Wasm module directly.

Wasm ABI

[k][l][m]

Like WASI or Emscripten, we define an ABI whose purpose is to expose the core set of services that are needed to make an application useful in a browser. Specifically, we expose:

- An ABI to launch a Wasm module on another thread, taking the existing Web Worker plus SharedArrayBuffer concept but enabling it without requiring JavaScript.
- A WebHID-based ABI.
- A WebGPU-based ABI.
- An ARIA-based ABI to describe the current accessibility tree.[n][o][p][q][r][s][t][u][v][w][x][y]
- An ABI to spawn a JavaScript environment (the opposite of the current JS API to spawn a Wasm environment). This enables access to the rest of the web stack without requiring additional ABIs.

WebHID considerations

The WebHID API would need to support providing access to keyboard and touch/mouse input by default

In practice, touch and mouse input may not be well suited to WebHID and an alternative may need to be provided instead.

Minimal viable product

In the original MVP, the WebGPU and ARIA APIs, and the API to spawn another thread, could be skipped so long as the API to spawn a JavaScript environment exists, since those API calls could be indirected through JS initially. Javascript shims could be created to make the API appear to Wasm code as it would once the APIs are directly exposed to Wasm as through an explicit ABI.

The WebHID API would still need to exist since the JS version of that API requires an explicit permission request from the user even for keyboard input, which would make it impractical.

Framework

This API alone is not useful for application developers, since it provides no high-level primitives. However, as with other platforms, powerful frameworks will be developed to target this set of APIs. A proof of concept already exists in Flutter, which is currently being ported to target Wasm GC and WebGPU. In time, other frameworks would likely find this to be a useful target. For example, porting game frameworks to Wasm would allow games using that framework to be used on the web. (Unity is an obvious example here, though a clear C#-to-Wasm story would presumably need to exist first.[z][aa][ab][ac][ad][ae][af][ag][ah])

Performance benefits

Early versions of this are unlikely to show significant performance benefits compared to the current Wasm-in-JavaScript model, in part because of the years of optimizations that JavaScript has backing it, compared to the minimal equivalent work that has so far been applied to Wasm itself and to Wasm ABI technologies. However, the benefits of simplicity will inevitably lead to performance improvements in the long term. For example, merely the ability to bootstrap right into Wasm rather than starting with HTML, then loading JavaScript, then loading the Wasm, can save many milliseconds[ai][aj][ak][al] in page load time.

The performance benefits are not required to prove the viability of the MVP.

PUBLICLY SHARED

[a]This is great. I've written a response essay that explores what an alternative non-web architecture might look like if you have the goal of really enabling frameworks like Flutter to shine, whilst preserving the feature set of the web. It's here - whilst I don't plan to implement this I hope it proves interesting to people. https://docs.google.com/document/d/1oDBw4fWyRNug3_f5mXWdlgDI4J5AbxVWKEeYr6hscT8/edit?usp=sharing
[b]see also my comment at https://news.ycombinator.com/item?id=34622514
[c] Markeren als opgelost
[d] Opnieuw geopend
[e] Markeren als opgelost
[f] Opnieuw geopend
[g]I propose another direction of development of a future replacement of HTML: https://science-dao.vporton.name/xml-for-publishing/ (my project is intended not to just develop "HTML6+", but to

[h]To be clear, this proposal isn't suggesting a replacement for HTML, merely a different entry-point in the browser's page loading logic. Much like a browser can open an image/jpeg file, a text/plain file, or a text/html file, this suggests also supporting opening a Wasm file.

[i]_Markeren als opgelost_

[j]_Opnieuw geopend_

[k]Am I understanding the idea correctly that these would be largely new ABIs defined in a standard body which would be sufficient for the needs of this new web stack? Would these go through standard bodies and seek cross browser approval?

What about all the functionality that aren't covered by this core set of ABIs? Like all the things Fugu has been working on https://fugu-tracker.web.app/ ?

[l]Yes to this being a standardization question, at least in the long term (experiments are probably justified before we try to standardize anything, otherwise, how do we know what to standardize).

It's a pretty minimal set of things we'd want to expose, though. If we expose a way to open a JS environment, then existing APIs could be exposed that way, and only the performance-sensitive ones (e.g. WebGPU), really commonly used ones (e.g. accessibility), or those without a good solution in JS land (e.g. some equivalent of WebHID, maybe some equivalent of WebWorkers) would be worth directly exposing as ABIs.

[m]Agreed that you could definitely be prototyped. I do think leveraging existing APIs would be essential to potentially do this. From my own experience standardizing dozens of APIs in the Fugu project, I think doing any of this in standards bodies (and getting Mozilla and Apple onboard) seems impossible (in my opinion).

[n]What about SEO? Accessibility functionality could help with SEO, but SEO today relies upon the HTML/CSS/JS content on a page. Shipping just a WASM would undercut this entirely and would force web crawlers to have to execute all that WASM, which doesn't seem practical.

This doc appears to just discuss the idea of putting a native application in a browser. Thus, is SEO not really considered here on purpose?

[o]Exactly my concern. There needs to be plans for indexability in this direction.

[p]SEO would work the same way it does today for pages that are all Wasm+canvas, which is to say, it wouldn't unless search engines start running and crawling Wasm apps the way they do JS apps. It's technically possible, but wouldn't work overnight, certainly.

[q]After you execute the JS app, you get text nodes that can easily be parsed. In case of something like Flutter, accessing that "text node" is not easy and in certain cases, it is possible that there is no "text", just bitmaps of text.

However, treating the whole app as a bitmap and applying text recognition on it would be much more simpler and effective in my opinion.

[r]Text recognition could work but is error-prone and compute-intensive for search engines (take a simple HTML parsing task and then compare it to an entire rendering and then text recognition pipeline--it just doesn't scale great). Pretty sure the way it would be done in practice is piggy-backing off of pre-existing accessibility support by parsing the accessibility tree, which you can think of as today's semantic HTML nodes (i.e. "article" and "input" elements specify what they do exactly).
This does require frameworks like Flutter to manage more data internally, but Flutter already does this today.

[s]_Markeren als opgelost_

[t]_Opnieuw geopend_

[u]@Riasat - you wouldn't grab text nodes, you would literally just use the accessibility tree as exported from the app via the (new) a11y ABI, in the same way that, say, if you wanted to crawl an Android app, you would crawl the accessibility tree exported via Android's a11y API.

[v]How does "cloaking" affect this scenario?

[w]Can you elaborate on your question?

[x]i'd suggest not calling this an "ARIA" anything. rather, just define that this environment you're envisaging, where the web simply works as the delivery mechanism for Wasm-based closed applications, provides a direct interface either to the platform's accessibility API itself, or a way to inject accessibility information into the browser which THEN passes this on, as an intermediary, to the

indexible HTML blob to robots.

[z]Unity has il2cpp whose output can be sent to emscripten to generate wasm.

[aa]There's also blazor, which IIRC compiles C# to WASM.

[ab]Both of those are what I would describe as "unclear stories". :-) But yes, they are presumably part of the foundation on which a Unity-to-Wasm model would be built, and I'm sure the Unity folks are already working on it.

[ac]It probably isn't important to you point, but to be pedantic, they added wasm support like 4 years ago. That's an official flow c#->il->cpp->wasm. It's just a flag you set when you hit build.

[ad]my impression was that you couldn't really do that and get a real web app out of a unity project yet. is that wrong?

[ae]They've had it since 2018: https://blog.unity.com/technology/webassembly-is-here I never shipped a product with it but I played with the asm.js predecessor.

[af]_Markeren als opgelost_

[ag]_Opnieuw geopend_

[ah]As far as I can tell, this doesn't seem to be entirely production-quality yet (e.g. I spoke to some folks in the game industry and they seemed very skeptical about the ability to use this to ship their game today). But yes, that kind of thing is what Unity would need.

[ai]Do you have any guesstimates on how many milliseconds? E.g., if a typical flutter/react app needs 2000ms to fully load, with this, will it be in the range of 1000ms or 100ms?

[aj]probably less than 100.

[ak]_Markeren als opgelost_

[al]_Opnieuw geopend_