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.

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, 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.
- WebHID provides an API (to JavaScript) that exposes the underlying input devices on the device.

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. 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

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.
- 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 without asking for permissions, and would need to be adjusted to only provide such data while the app is focused.

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.)

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 in page load time.

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