WebAssembly and the Wizards of Hogwarts

Daniel Burger[1]

[1]**Middlesex University London***
public@danielburger.online

7. November 2020

Abstract

There is a new kind in the web development hood: WebAssembly. It is fast, portable, supported by the big players, and should allow the World Wide Web to become the largest software platform in existence. It does not matter if a person is an experienced software engineer or a beginner web developer—everyone can profit from WebAssembly's existence.

In this essay, we will look at what WebAssembly is, how it works and how it works alongside its sibling JavaScript.

*In collaboration with SAE Institute Zürich.

Table of Contents

List of Figures

List of Tables

List of Listings

1 Introduction

On the 17th of June 2015, Brendan Eich—the inventor of JavaScript—and the teams behind Mozilla, Chrome, Edge and WebKit presented a new browser standard: WebAssembly, a portable and highly efficient byte-code compilation target for high-level languages such as C++ and Rust (Eich, 2015).

However, what does this mean? What is the reason that WebAssembly should exist in the first place? Should JavaScript developers be worried now? Furthermore, what do the wizards of Hogwarts have to do with it?

1.1 Web Development Back Then

Back in the day, when people could call themselves web developers because they only understood HTML, web development was a rather interactionless and static field of business. Netscape Communications, a pivotal company in the development of the modern web and the creator of the Netscape Browser (shown in Figure 1.1), quickly recognised that websites of the time lacked interactivity and dynamism. They wanted the web to be a new form of a distributed operating system rather than just a simple HTML document-accessing application on people's computers (Cassel, 2018).

Figure 1.1: The Netscape Navigator browser interface, circa mid-1990s. This screenshot captures the early stages of the World Wide Web, where Netscape Communications made significant contributions towards a more dynamic and interactive web (NPR, n.d.).

Marc Andreessen, founder of Netscape Communications, proposed that HTML needed some kind of "scripting language" that was approachable, something you could glue directly into the markup. A language that is easy to use by newbie programmers who do not want to handle compiler errors or strictly typed syntax.

That was why they hired the experienced programming language and network developer Brendan Eich. Brendan's first task was the nearly unachievable goal of creating

such a scripting language for the web—due in 10 days. They (later) called it JavaScript (Severance, 2012).

1.2 JavaScript's Destiny

One does not need to dig too deep into the history of the web to show one crucial pain point of today's web technology standards: JavaScript is a scripting language for the browser to interpret—not some fancy multi-paradigm system programming language that focuses on speed, security or code maintainability. It was supposed and designed to be an easy-to-understand dynamically typed scripting language to, e.g. give a website fast DOM manipulations and decorative animations.

Nevertheless, see what happened:

> *"Any application that can be written in JavaScript will eventually be written in JavaScript." (Atwood, 2007)*

This is a quote by Jeff Atwood (co-founder of Stack Exchange), commonly referred to as Atwood's Law. Ever since frameworks like Node.js or React Native became widely used, as depicted in Figure 1.2, JavaScript's possibilities have already crossed the border of just living on the client side inside of a browser. It is nearly everywhere.

Also, Node.js' NPM package manager is currently the most prominent and active package registry platform ever created. As an example: From the 10th of May to the 17th of May of 2018, JavaScript developers downloaded 5.2 billion Node.js packages from the NPM registry, setting a new record (NPM, 2018)

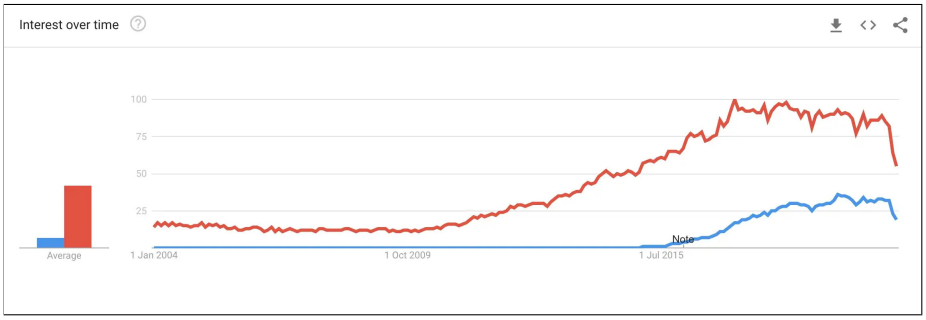

Figure 1.2: Google Trends data illustrating the ascent of Node.js (in red) and React Native (in blue) from January 2004 to 2018. The marked increase aligns with the period when JavaScript began to dominate server-side programming and mobile app development, demonstrating the language's versatility and the extensive ecosystem supported by NPM (Google, n.d.).

1.3 Treacherous Atwood's Law

There are thousands of courses, books and tutorials on how to learn JavaScript. Every computer-related school now or then teaches JavaScript. Nearly everyone can learn it

anywhere with a minimum amount of effort. The search term for "Learn JavaScript" on Google will get 2 220 000 000 search results. In comparison, the search term "Learn Java" will get 305 000 000 results.

Though, is that a good thing? Is an originally lightweight scripting language capable of ruling the world of computational web development? My opinion: No, it is not, and there will not be such a bright future for JavaScript as nearly everybody claims. Let me explain it with a Harry Potter analogy:

1.4 Muggles Entering Hogwarts

Do you know how it feels to watch web developers call themselves "full-stack software engineers" after they have simply learned Node.js? It feels like Muggles entering Hogwarts—a school full of wizards. Only in our case, these wizards are trained software engineers and computer scientists. These are the people who learn and work on all the hardcore-implemented algorithms and compilers, programming languages and operating systems. They know precisely how to build software (Might, 2011). Do web developers know how to write actual software? I would say we better not talk about it.

Muggles are not invited to Hogwarts because they have no magical ability. Web developers were not "invited" to software engineering, too, because they could not even do something with the languages they knew. What they wrote was not even considered software per se, just web pages. Nevertheless, thanks to all the cross-platform JavaScript runtimes, they are suddenly here to do some magic. However, imagine what if the wizards of Hogwarts, our beloved software engineers and computer scientists, could perform magic in the real world—or, in our case, the web? What would happen? What could go wrong?

2 WebAssembly Says Hello World

WebAssembly is the new player in the web development industry. It is fast, small, non-readable and not even an actual programming language. You literally cannot code in WebAssembly (Rourke, 2018). So, why should we all be excited about it? It is a compilation byte-format target for high-level languages used side-by-side with JavaScript, CSS and HTML, as illustrated in Figure 2.1.

You write your code in a high-level language like C or C++ and compile it to WebAssembly. The magic trick: It works in the browser and is super fast—sometimes about 5 to 20 times faster than JavaScript (Aboukhalil, 2019).

3 WebAssembly in a Nutshell

The real face behind the term "WebAssembly" is not as uniform as it is being marketed. WebAssembly itself is only a piece of a bigger technology chain of workflows and concepts. Several other key components are essential for delivering super-fast web applications. It is also good to know its current limitations and which use cases it is the best for. Nevertheless, first, let us talk about the five key components:

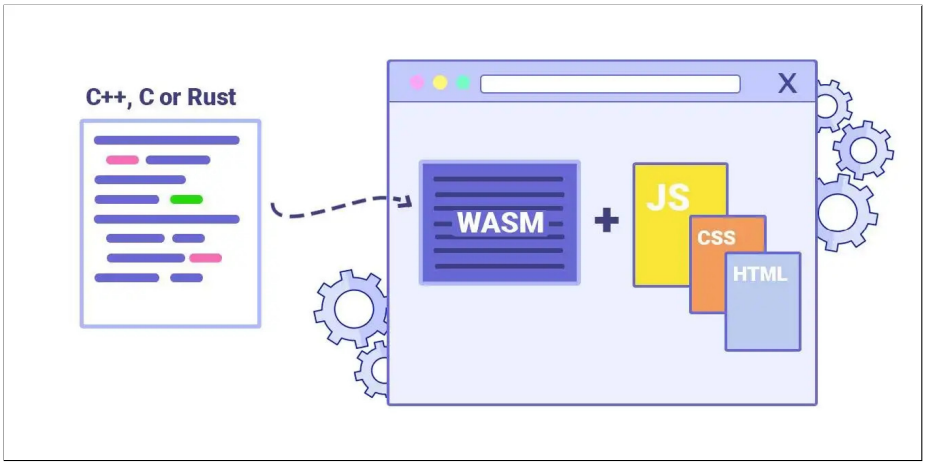

Figure 2.1: Conceptual diagram depicting the process of integrating WebAssembly with traditional web technologies. High-level programming languages like C++, C, or Rust are compiled into WebAssembly (WASM) modules, which are then delivered to the browser to work in conjunction with JavaScript (JS), Cascading Style Sheets (CSS), and HyperText Markup Language (HTML) (LogRocket, n.d.).

3.1 WAT — WebAssembly Text Format

This is a human-readable file format you will get when you compile your C, C++ or Rust code. It represents the abstract syntax tree (AST) from the source code of a programming language. An AST—or, in the WebAssembly case, a .wat file—may be verbose, but it does an excellent job describing the source code components. Representing source code in an AST makes verification and compilation simple and efficient. Here is a simple return function called `getDoubleNumber` written in C:

```c
int getDoubleNumber(int x) {
  return x * 2;
}
```

Listing 3.1: Code example in C.

Compiling this C code will return a `.wat` file containing the abstract syntax tree of our `getDoubleNumber` function. It looks like this:

```
(module
  (table 0 anyfunc)
  (memory $0 1)
  (export "memory" (memory $0))
  (export "getDoubleNumber" (func $getDoubleNumber))
  (func $getDoubleNumber (; 0 ;)
    (param $0 i32) (result i32)
  (i32 .shl
    (get_local $0)
```

```
      (i32.const 1)
     )
   )
 )
```

Listing 3.2: Code example from above compiled into the WebAssembly Text Format.

3.2 WASM — WebAssembly Binary Instruction Format

While in production, you probably will not send the text instruction format file to the client side. You will only send the binary instruction format (.wasm). This is the actual low-byte format file, written in non-readable hexadecimal code. Usually, they are referenced as WebAssembly modules (Mozilla, 2023a). Here is the example from the `getDoubleNumber` C function from above:

```
0061 736d 0100 0000 0186 8080 8000 0160 017f 017f 0382
8080 8000 0100 0484 8080 8000 0170 0000 0583 8080 8000
0100 0106 8180 8080 0000 079c 8080 8000 0206 6d65 6d6f
7279 0200 0f67 6574 446f 7562 6c65 475 6d62 6572 0000
0a8d 8080 8000 0187 8080 8000 0020 0041 0174 0b
```

Listing 3.3: Code example from above compiled into the WebAssembly Binary Instruction Format.

3.3 WASM Module Instantiating

If you want to access the C code within your website, you need to instantiate the WebAssembly .wat module inside JavaScript. This would look like this:

```
// Access the WebAssembly object
WebAssembly.instantiateStreaming(
  fetch("program.wasm"), imports)
  // Resolve the promise
  .then(_wasm => {
    // Log something if it worked
    console.info("WASM is ready")
  })
```

Listing 3.4: Instantiate the .wasm module in JavaScript.

3.4 WebAssembly Compilation

In order to get a .wasm or .wat file, you first need to compile your source code. There are already different compilers for the various languages out there. The most common one—and also the most famous one—is called Emscripten. Emscripten is described as a so-called LLVM source-to-source compiler with the main focus of compiling

C code straight to a subset of JavaScript known as asm.js. However, the recent rise of WebAssembly has pushed the team behind Emscripten to shift its focus to help make WebAssembly more accessible and easier to start with. You can then access the Emscripten compiler inside your command-line interface to easily compile selected source code.

3.5 Original Source Code

To write efficient software and compile it to WebAssembly, you must be proficient in C, C++ or the Rust programming language (or use AssemblyScript if you are familiar with TypeScript). The WebAssembly Working Group has plans to add more languages in the near future. However, they have other priorities and next steps on their roadmap. They primarily focus on the most significant pain points of WebAssembly.

4 Pain Points And Limitations

The WebAssembly workflow sounds too good to be true. Just write your code in a high-level language, compile it via Emscripten, and then instantiate it inside JavaScript—it is as easy as that. Well, no, it is not. There are still some considerable pain points for easily porting your desktop-like application to the web despite not having a garbage collector or exception handlers (W3C, 2019). The most significant pain point right now is, for sure, the lack of web APIs. Let us say you want C++ code that can listen if a user clicks a button and then render some data from your database into the client-side DOM view.

Well, I need to disappoint you; this will not work because WebAssembly cannot access the DOM API. The only way to access it is via JavaScript. If you want to develop actual WebAssembly-ready client-side software, you would need to write some JavaScript glue code, as depicted in Figure 4.1 (Mihaylov, 2018) that acts as the asynchronous bridge between your WASM module and the client-side HTML page.

5 Current Best Use Case

As of today, the best use case for WebAssembly under these circumstances is math-heavy calculations for graphics rendering inside an HTML5 canvas element. The canvas element is the only place to create a graphical user interface without accessing the DOM. This does not sound very interesting to many, but at least it is super promising for the gaming industry. Since games rely on computationally heavy shader calculations and simulations, compiling WebAssembly makes perfect sense. That is why Epic Games showcased a compiled Unreal Engine C++ high-resolution game inside the browser. Everything related to high-fidelity graphics or simulations is the most straightforward fit for WebAssembly right now—but for everything else that still needs to access or manipulate the DOM, use a glue code mixture between WebAssembly and JavaScript.

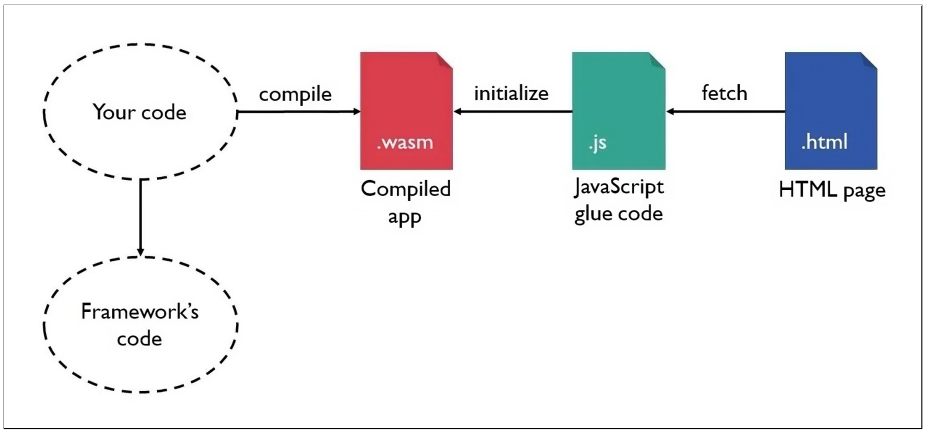

Figure 4.1: Workflow diagram illustrating the interaction between WebAssembly and JavaScript. Developer code is compiled into a WebAssembly (.wasm) module, which is initialised through JavaScript glue code. This JavaScript code provides the necessary interface for the WebAssembly module to interact with the HTML page, enabling access to the DOM API that WebAssembly cannot directly manipulate (Mihaylov, 2018).

6 Glue Code

But wait, this sounds familiar? Glue code? Wasn't this the original purpose of JavaScript? Well, yes, it was. However, since the WebAssembly Working Group is actively working on an official DOM API (Mozilla, 2023b), what is the primary purpose of JavaScript in the first place? If we can soon create very efficient full-stack web applications with just one high-level language, do we really need JavaScript in the near future?

Well, we do not know; you cannot kill a whole ecosystem with millions of developers overnight. This did not work for other "dead-talked" ecosystems like PHP and will not work for JavaScript either. However, different people have different predictions. Moreover, the following chapter describes some predictions for the future.

7 WebAssembly's Future

Since people with in-depth knowledge of software development and computer science—aka our Wizards from Hogwarts — soon have the ability to port their power to the client side, we will most likely see a completely new age of the World Wide Web as we know it today. Netscape Communications' original vision of the web as a distributed operating system could finally become a reality. Native apps and their app stores are going extinct, and whole software packages like the suites from Adobe and Autodesk will run entirely inside the browser. The web will push its borders far beyond what we now define as the information and entertainment industry epicentre.

In the future, JavaScript will still have an essential role, but not as crucial as today. It will act as a simple scripting language to provide websites with dynamic content and to create animations. The positioning of JavaScript will most likely be like the one it

initially was being created for, and WebAssembly will act as the binary format for the browser to do all software-like heavy-lifting as, e.g. the real-life uses cases in Table 7.1 present.

Company	Use Case
Figma	Browser-Based Interface Design Tool. Initially, it was written in C++ and compiled from C++ to asm.js. Switching to WebAssembly improved document loading times by over three times.
Autodesk	CAD Drawing Software. Autodesk AutoCAD's original code, older than the WWW, was compiled into WebAssembly as a web app.
eBay	Barcode Scanner. eBay improved their JavaScript barcode scanner by rebuilding it in C++ and compiling it to WebAssembly to enhance its success rate significantly.

Table 7.1: Real-life use cases for WebAssembly

References

Aboukhalil, R. (2019). How We Used WebAssembly To Speed Up Our Web App By 20X (Case Study). Publication Title: Smashing Magazine.
URL: https://www.smashingmagazine.com/2019/04/webassembly-speed-web-app (Accessed at: 2022-12-11)

Atwood, J. (2007). The Principle of Least Power. Publication Title: Coding Horror.
URL: https://blog.codinghorror.com/the-principle-of-least-power (Accessed at: 2022-12-11)

Cassel, D. (2018). Brendan Eich on Creating JavaScript in 10 Days, and What He'd Do Differently Today. Publication Title: The New Stack.
URL: https://thenewstack.io/brendan-eich-on-creating-javascript-in-10-days-and-what-hed-do-differently-today (Accessed at: 2022-12-11)

Eich, B. (2015). From ASM.JS to WebAssembly – Brendan Eich.
URL: https://brendaneich.com/2015/06/from-asm-js-to-webassembly (Accessed at: 2022-12-11)

Google (n.d.). Google Trends.
URL: https://trends.google.com/trends/explore?date=all&q=%2Fm%2F0bbxf89,%2Fg%2F11h03gfxy9&hl=en-GB (Accessed at: 2022-12-11)

LogRocket (n.d.). LogRocket Blog. Publication Title: LogRocket Blog.
URL: https://blog.logrocket.com (Accessed at: 2022-12-11)

Might, M. (2011). What every computer science major should know. Publication Title: Matt Might Blog.
URL: https://matt.might.net/articles/what-cs-majors-should-know (Accessed at: 2022-12-11)

Mihaylov, B. (2018). How WebAssembly influences existing JavaScript frameworks / Boyan Mihaylov.
URL: https://boyan.io/how-webassembly-influences-existing-javascript-frameworks/ (Accessed at: 2023-12-20)

Mozilla (2023a). WebAssembly Concepts - WebAssembly | MDN.
URL: https://developer.mozilla.org/en-US/docs/WebAssembly/Concepts (Accessed at: 2023-06-28)

Mozilla (2023b). WebAssembly.Module - WebAssembly | MDN.
URL: https://developer.mozilla.org/en-US/docs/WebAssembly/JavaScript_interface/Module (Accessed at: 2023-12-20)

NPM (2018). How well do you know your npm trivia? Publication Title: Jaxcenter.
URL: https://jaxenter.com/npm-trivia-144846 (Accessed at: 2022-12-11)

NPR (n.d.). Home Page Top Stories. Publication Title: NPR.org.
URL: https://www.npr.org (Accessed at: 2022-11-19)

Rourke, M. (2018). *Learn WebAssembly: Build web applications with native performance using Wasm and C/C++*. Packt Publishing.

Severance, C. (2012). JavaScript: Designing a Language in 10 Days. *Computer*, *45*(02), 7–8.
URL: https://www.computer.org/csdl/magazine/co/2012/02/mco2012020007/13rRUy08MzA (Accessed at: 2022-12-11)

W3C (2019). Roadmap - WebAssembly.
URL: https://webassembly.org/roadmap/ (Accessed at: 2023-12-20)