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 making the world wide web the biggest software platform in existence. It doesn't matter if you are an experienced software architect or a frontend developer—everyone will be able to 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 . . . . . **II**

List of Tables . . . . . **III**

List of Listings . . . . . **IV**

1 Introduction **1**

1.1 Web Development Back Then 1
1.2 JavaScript's Destiny 2
1.3 Treacherous Atwood's Law 2
1.4 Muggles Entering Hogwarts 3

2 WebAssembly Says Hello World **3**

3 WebAssembly in a Nutshell **4**

3.1 WAT — WebAssembly Text Format 4
3.2 WASM — WebAssembly Binary Instruction Format 4
3.3 WASM Module Instantiating 5
3.4 WebAssembly Compilation 5
3.5 Original Source Code 5

4 Pain Points And Limitations **6**

5 Current Best Use Case **6**

6 Glue Code **7**

7 WebAssembly's Future **7**

References **8**

List of Figures

1.1 Screenshot of the Netscape browser and start-page. 1
1.2 Google Trends curve about Node.js and React Native. 2
2.1 Illustration of how WebAssembly modules are being delivered to the browser. 3
4.1 Illustration of how WebAssembly works together with JavaScript. . . 6

List of Tables

7.1 Real-life use cases for WebAssembly 7

List of Listings

3.1 Code example in C. 4
3.2 Code example from above compiled into the WebAssembly Text Format. 4
3.3 Code example from above compiled into the WebAssembly Binary Instruction Format. 5
3.4 Instantiate the .wasm module in JavaScript. 5

1 Introduction

On the 17th of June 2015, Brendan Eich—the inventor of JavaScript—and the teams behind Mozilla, Chromium, 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? And what do the wizards of Hogwarts have to do with it?

1.1 Web Development Back Then

Back in the day when you could call yourself web developer because you only understood HTML, web development itself 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 your computer (Cassel, 2018).

Figure 1.1: Screenshot of the Netscape browser and start-page (NPR, n.d.).

Marc Andreessen, who was the 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 didn't want to handle compiler errors or strictly-typed syntax.

That was the reason they hired the experienced programming language and network code 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

I don't need to dig too deep into the history of the web to show you one crucial pain point of today's web technology standards: JavaScript is a scripting language for the browser to interpret. I repeat it: 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 give your website some cool 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)*

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

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

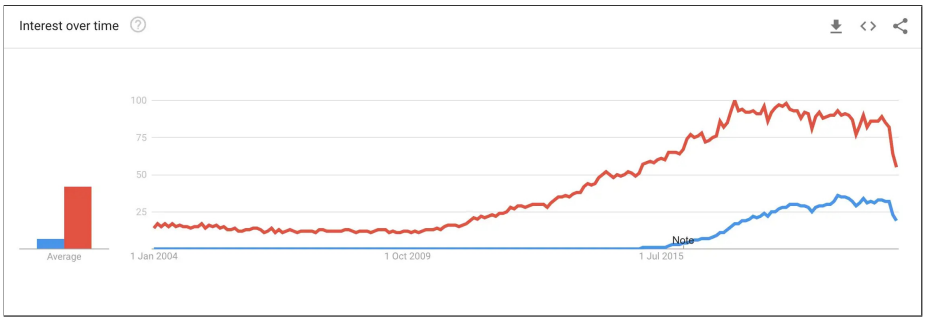

Figure 1.2: Google Trends curve about Node.js and React Native (Google, n.d.).

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 could be able to learn it anywhere with a minimum amount of effort. If you search for "Learn JavaScript" on Google, you'll get 2 220 000 000 search results. In comparison: When you search "Learn Java" you'll 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's not, and I believe there won't 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 frontend developers call themselves "full-stack software developer" after they've simply learned Node.js? It feels like Muggles entering Hogwarts—a school full of wizards. Only that in our case these wizards are trained software engineers and computer scientists. These are the people who learn all the hardcore-implemented algorithms and compilers, programming languages and operating systems. They know precisely how to build software (Might, 2011). Do frontend developers know how to write actual software? I'd say we better don't talk about it.

Muggles aren't invited to Hogwarts because they have no magical ability. Frontend developers weren't "invited" for software engineering too because they couldn't even do something with the languages they knew. Nevertheless, now—thanks to all the cross-platform JavaScript runtimes—they're suddenly here to do some magic. But imagine, what if the wizards of Hogwarts, our beloved software engineers and computer scientists, would be able to perform magic in the real world—or our case: the frontend? What would happen? What could go wrong?

2 WebAssembly Says Hello World

WebAssembly is the new player in the web development industry. It's fast, small, non-readable and not even a real programming language. Yes, you heard that right. You literally can't code in WebAssembly (Rourke, 2018). So I may hear you asking: Why should we all be excited about it? Well, as mentioned earlier, it's a compilation byte-format target for high-level languages.

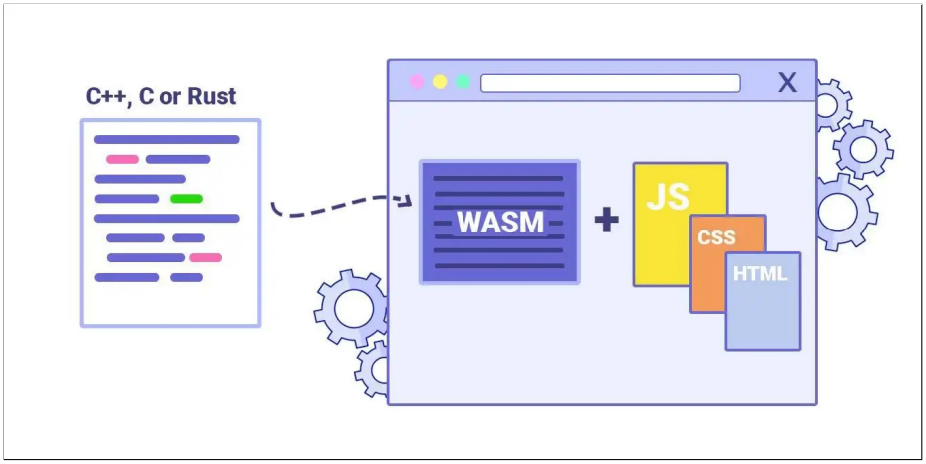

Figure 2.1: Illustration of how WebAssembly modules are being delivered to the browser (LogRocket, n.d.).

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

3 WebAssembly in a Nutshell

Enough with the marketing fuzz. The real face behind the term "WebAssembly" isn't that uniform as it's being marketed. WebAssembly itself is only a piece of a bigger technology chain of workflows and concepts. There are several other key components that are important for delivering super-fast web applications. It's also good to know what its current limitations are and for which use cases it's the best. But first, let me introduce you to the five key components:

3.1 WAT — WebAssembly Text Format

This is a human-readable file format you'll 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 at describing the components of source code. Representing source code in an AST makes verification and compilation simple and efficient. Here is a simple return function called `getDoubleNumber` written in C:

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

Listing 3.1: Code example in C.

If you compile this C code, it will return a .wat file which contains 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 being in production, you probably won't send the text instruction format file to the client-side. You'll only send the binary instruction format (.wasm). This is the actual low-byte format file, written in non-readable hexadecimal code. Usually, they're

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 of 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 making WebAssembly more accessible and easier to get started with. You can then access the Emscripten compiler inside of 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 need to be proficient in C, C++ or the rather new Rust programming language. The WebAssembly Working Group has its plans to add more languages in the near future. But, right now they have other priorities and next steps on their roadmap. They especially focus on the biggest pain points of WebAssembly.

4 Pain Points And Limitations

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

Well, I need to disappoint you, this isn't going to work because WebAssembly can't access the DOM API. The only way to access it is via JavaScript. If you would want to develop actual WebAssembly-ready client-side software, you would need to write some kind of JavaScript glue code (Mihaylov, 2018), that acts as the asynchronous bridge between your WASM module and the client-side HTML page.

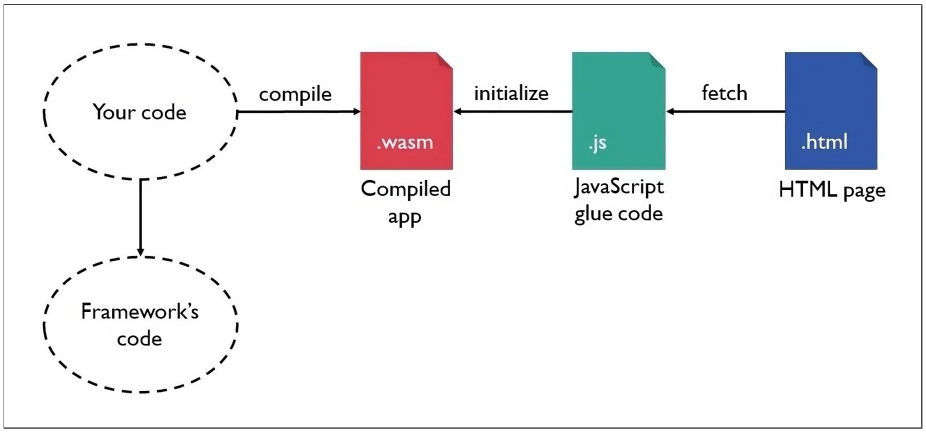

Figure 4.1: Illustration of how WebAssembly works together with JavaScript (Mihaylov, 2018).

5 Current Best Use Case

Currently, the best use case for WebAssembly under these circumstances would only be math-heavy calculations for graphics rendering inside of an HTML5 canvas element. This is because the canvas element is the only place where you can create a graphical user interface without accessing the DOM. This doesn't sound very interesting for lots of people, but at least it's super promising for the gaming industry. Since games rely on heavy shader calculations and simulations, the compilation to WebAssembly into a canvas makes perfect sense. That's 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 easiest 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.

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's the main purpose of JavaScript in the first place, though? If we soon can create very efficient full-stack web applications with just one high-level language, do we really need JavaScript in the near future?

Well, we don't know, you can't kill a whole ecosystem with millions of developers overnight. This didn't work for PHP and also won't work for JavaScript. However, different people have different predictions. Moreover, following chapter describes my 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, as well as their app stores, are going extinct and whole software packages like the suites from Adobe and Autodesk will entirely run inside the browser. The web will push its current borders far beyond what we now define as the epicentre for the information and entertainment industry.

In the future, JavaScript will probably still have an essential role, but not as important 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 originally 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 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 to WebAssembly for use 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)