awesome Wasm!

TLDR; WebAssembly (Wasm) is now the official[1] second language of the web, offering a compact and performant complement to JavaScript.

[1] Well, mostly official

awesome Wasm!

WebAssembly is the culmination of previous attempts to both achieve **near-native performance** in the browser as well as provide an improved **security model** for running ‘untrusted’ code.

Questions to answer…

… what’s the story ?

… why the new standard ?

… `.wat` is `.wasm` ?

… is it better+faster+stronger ?

… can we see it ?

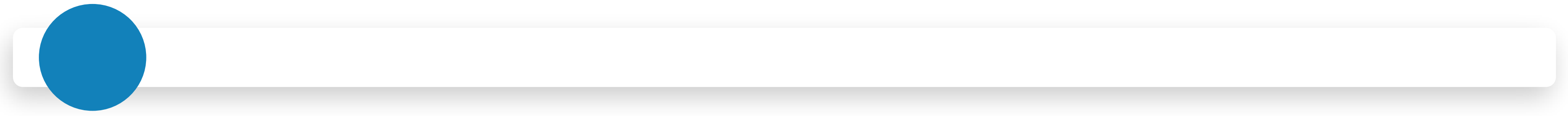
… who’s using it ?

… where’s it going ?

what’s the story?

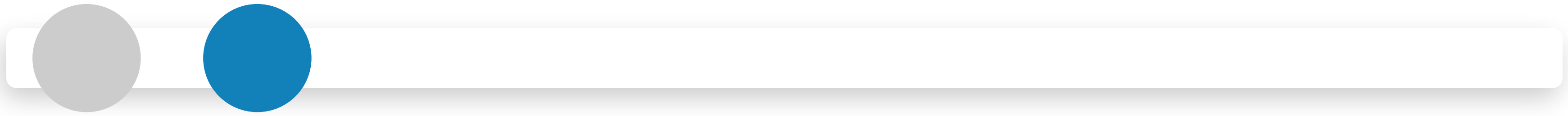
a comically brief, incomplete, and quite possibly inaccurate history of client-side code

what’s the story?



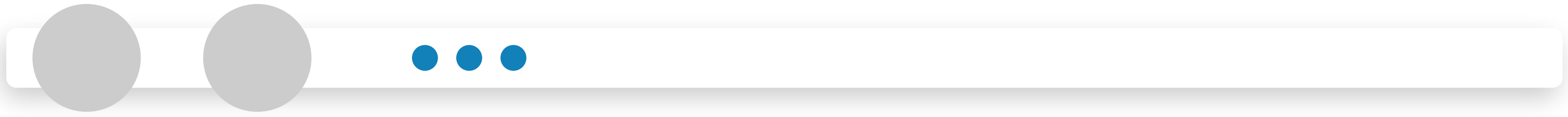
In **The Beginning** there was static text, and Things Were Simple™.

what’s the story?



Then, in **1995**, JavaScript was created to give page designers a simple ‘behavior layer’, and the intention was really just simple DOM interaction.

what’s the story?



JavaScript remained fully interpreted (read: slow) for **over a decade**, and this was accepted because Java applets were considered the ‘blessed’ path toward greater performance.

what’s the story?

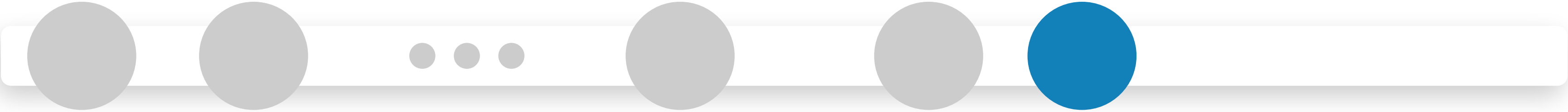
Google released V8 in **2008** and introduced the first JIT compiler for JavaScript, which set the language on a path toward remarkably faster (“10x”) execution speeds.[1]

[1] Lin Clark wrote an excellent article describing how these compilers work at a high level

what’s the story?

In **2011**, Google released a prototype tech for Chrome called Portable Native Client (PNaCl), which was a compile target for C(++) aiming to achieve near-native speed in the browser.

what's the story?



Mozilla released asm.js in **2013** as a compile target for C(++) that, unlike PNaCl, aimed to leverage the existing JavaScript VMs to execute more optimized code while also 'bypassing' the garbage collector.[1]

[1] asm.js would be made more powerful as different browsers introduced JIT optimizations tailored specifically for it

what's the story?

In **2015** engineers from Google, Mozilla, Microsoft, and Apple[1] announced the new joint WebAssembly standard, which was heavily inspired by asm.js and PNaCl.

[1] Those working on the standard became the [W3C WebAssembly Working Group]

what's the story?

The MVP design was completed in early **2017** and all major browsers[1] (including mobile) offered support by the end of the year.

we're here!

[1] Again, ~87% support, but who cares about IE these days anyway?

why the new standard?

`ie. why not just make JavaScript even moar better-er?`

why the new standard?

> *The maturation of the Web platform has given rise to sophisticated and demanding Web applications such as interactive 3D visualization, audio and video software, and games. With that, **efficiency** and **security of code** on the Web has become more important than ever. Yet JavaScript as the only built-in language of the Web is not well-equipped to meet these requirements, especially as a compilation target.*

Bringing the Web up to Speed with WebAssembly, 2017

why the new standard?

JIT compilation and optimization cycles radically improved JavaScript execution speed, but its fundamental nature rendered it unsuitable for certain types of applications.

- Source files require a non-trivial amount of **time to parse**, especially on mobile where <1MB apps can require 20-40s or more to fully parse into an AST on many devices

- Code paths are **compiled and re-compiled** based on how they are executed, decreasing the predictability of performance and essentially setting hard limits on execution speed

- The profiler needs to continually store baseline and optimized versions of different functions, increasing **memory overhead**

- asm.js relies on JavaScript engines having **specific optimizations** to reach its peak performance

why the new standard?

Beyond hard performance limits set by JIT, there are **no additional protections** around running untrusted (read: third-party) code beyond just the browser sandbox.

why the new standard?

To alleviate these limitations, the WebAssembly Working Group outlined their mission for a new technology.

> *Define a portable, size- and load-time-efficient **binary format** to serve as a **compilation target** which can be compiled to **execute at native speed** by taking advantage of common hardware capabilities available on a **wide range of platforms**, including mobile and IoT.*[1]

[WebAssembly | High Level Goals]

[1] Pretty generic, right? https://webassembly.org is basically the reason that Lin Clark's *A Cartoon Intro to WebAssembly* is so priceless - and necessary

.wat is **.wasm?**

abbreviations, acronyms, and contractions all the way down

`.wat` is `.wasm`?

The working group decided to design and implement Wasm in an **incremental** fashion. The current implementations[1] began with an MVP that included…

1. … an **instruction set** semantics for an abstract stack machine
2. … a **bytecode module** format (**`.wasm`**) that the browser will compile to machine code
3. … a **textual** format (**`.wat`**) to allow inspection and debugging
4. … an embeddable, **portable design** that would allow for non-browser applications

[1] There have already been some major post-MVP improvements to Wasm implementations, including fast calls between JS and streaming compilation

`.wat` is `.wasm`?

The essence of the embeddable design is the compiled `.wasm` module that…

- … is loaded, instantiated, and **executed by a host** (eg. JavaScript VM)
- … can only call Wasm instructions, **internal functions**, or **host imports**[1]
- … can only interact with **linear memory** arrays[2]

Modules are meant as a **compile target** for other languages (eg. Rust) but one can hand-write a human-readable `.wat` file to compile to a module

- Compilation[3] is **two-way**, meaning **binaries can always be converted to a readable format** regardless of source

[1] A WebAssembly module has no access to the outside world (including Web APIs) except for functions imported by the host
[2] Linear memory is why garbage collection is not necessary and why modules cannot reach beyond their own memory
[3] We can use the official command-line toolkit WABT (pronounced "wabbit") to compile

`.wat` is `.wasm`?

A sample `.wat` module to do basic math could look like…

```
(module
  (func $add (param $lhs i32) (param $rhs i32) (result i32)
    local.get $lhs
    local.get $rhs
    i32.add
  )
  (func $double (param $x i32) (result i32)
    (i32.add (local.get $x) (local.get $x))
  )
  (export "add" (func $add))
  (export "double" (func $double))
)
```

`.wat` is `.wasm`?

Wasm only supports **four primitive types**: `i32`, `i64`, `f32`, and `f64`.

- There is no distinction between signed and unsigned integers
- Instructions like **i32.add** treat an integer as *signed*, but you can change that through instructions like **i32.add_u**

This means that only numbers can be passed to or returned from Wasm functions.

- Anything more complex (eg. strings) requires marshalling to and from a **linear memory** and passing pointers around[1]
- Any language that can compile to Wasm should have facilities for generating this‘glue code, but that still does not remove the performance implications

[1] Like many current limitations, this will change with the introduction of reference types

`.wat` is `.wasm`?

Linear memory is essentially a contiguous array of bytes, which enables Wasm to avoid garbage collection while still isolating the accessible memory.

- Memory is indexed **by byte**, can grow as needed, and be written to, read from, and even imported and exported
- Modules declare contiguous memory in increments of 64kB if they want to store anything **outside of the stack**

`.wat` is `.wasm`?

For instance, if we wanted to store an internal list of 32-bit integers and have some (meager & useless) utility for setting and getting the third number, we would define a module like...

```
(module
  ;; Declare a single 64kB block of memory
  (memory $mem 1)

  ;; Save a number to the third spot in memory.
  (func $setThird (param $num i32)
    ;; Save directly to third 8-byte address in linear memory
    ;; First 8-byte number stored at byte address 0, second at 8, etc.
    (i32.store (i32.const 16) (get_local $num))
  )

  ;; Returns the 32-bit number stored at third position
  (func $getThird (result i32)
    (i32.load (i32.const 16))
  )
  (export "setThird" (func $setThird))
  (export "getThird" (func $getThird))
)
```

`.wat` is `.wasm`?

Okay, okay… but what about *actually loading* one of these modules?

- Loading will soon be possible via **`<script type="module">`** tag[1]
- For now, a module must be manually fetched, buffered, and instantiated…

[1] This tag (independent of Wasm) [only works in modern browsers](), so (again) no IE

`.wat` is `.wasm`?

… like so.

```html
<!doctype html>
<html>
  <head>
    <meta charset="utf-8"/>
  </head>
  <body>
    <script type="text/javascript">
      const response = fetch("mod.wasm")
        .then(resp => resp.arrayBuffer())
        .then(WebAssembly.instantiate)
        .then(obj => {
          const { setThird, getThird } = obj.instance.exports;

          console.info(`The third number starts at ${getThird()}`);
          console.info('Setting third number to be 123');

          setThird(123);

          console.info(`The third number is now ${getThird()}`);
        });
    </script>
  </body>
</html>
```

is it better+faster+stronger?

or were the promises all filthy, filthy lies?

is it better+faster+stronger?

One of the major stated goals of WebAssembly is **near-native performance**.

- Current benchmarks vary wildly, suggesting 50-80% the speed of native

is it better+faster+stronger?

One of the major stated goals of WebAssembly is **near-native performance**.

- Current benchmarks vary wildly, suggesting 50-80% the speed of native

Wasm generally outperforms* JavaScript and asm.js.

- 2018: Different implementations of WebAssembly differed dramatically in performance
- 2019: Browser implementations have begun to converge and Wasm outperforms asm.js by **a factor of 1.33 - 1.5x**[1, 2]
- Also: **64-bit integers**![3]

[1] https://www.usenix.org/system/files/atc19-jangda.pdf
[2] https://medium.com/@torch2424/webassembly-is-fast-a-real-world-benchmark-of-webassembly-vs-es6-d85a23f8e193
[3] Calling a Wasm function that returns an `i64` in JavaScript will be a `TypeError`

is it better+faster+stronger?

One of the major stated goals of WebAssembly is **near-native performance**.

- Current benchmarks vary wildly, suggesting 50-80% the speed of native

Wasm generally outperforms* JavaScript and asm.js.

- 2018: Different implementations of WebAssembly differed dramatically in performance
- 2019: Browser implementations have begun to converge and Wasm outperforms asm.js by a factor of 1.33 - 1.5x
- Also: **64-bit integers**![1]

In general, Wasm is **great** for computational tasks* and tight loops…

- … but (until reference types) performance will suffer pretty quickly when passing non-primitives back and forth

[1] Calling a Wasm function that returns an `i64` in JavaScript will be a `TypeError`

can we see it?

ETLP; enough talk, let’s play!

can we see it?

Game of Life in JS and Wasm

Soft-body physics: The Blob!

Epic's ZenGarden

who’s using it?

does anyone you’ve *actually heard of* use it?

who's using it?

https://www.figma.com/blog/webassembly-cut-figmas-load-time-by-3x/
https://blogs.autodesk.com/autocad/autocad-web-app-google-io-2018/
https://blogs.unity3d.com/2018/08/15/webassembly-is-here/

where’s it going?

in other words, why it’s actually exciting

where’s it going?

Wasm is **great as-is** but is lacking features necessary for it to be greater or **more widely appealing**.

- Reference types to allow better interop with host (eg. JS, DOM, Web APIs, etc.)
- Access to JS garbage collector to open up compilation from managed languages

Non-browser hosts.

- Wasm offers a potentially **unparalleled security model** for CLI tools, mobile, and **even desktop apps**
- WASI: WebAssembly System Interface

where's it going?

Some excellent resources:[1]

- Planned post-MVP features
- A "cartoon skill tree" of what works, what's in development, and what's necessary still

Writing Wasm with Rust:

- The Rust-Wasm book
- Rust, WebAssembly, and Javascript Make Three

Also, see the official white paper for more background and the current instruction set

[1] Lin Clark is much smarter than I am and a much better writer, hence all of the links to her articles