WebAssembly for Developers (web... or not)

Horacio Gonzalez

@LostInBrittany

Horacio Gonzalez

@LostInBrittany

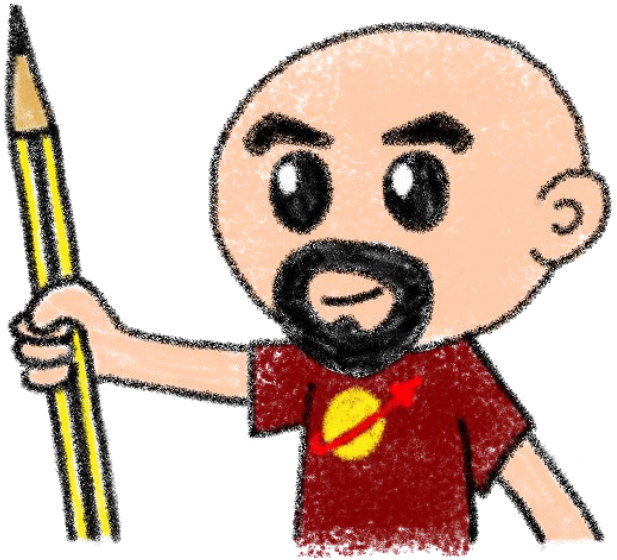
OVH
Team DevRel

Spaniard lost in Brittany, developer, dreamer and all-around geek

DevFest du Bout du Monde

Finist Devs

Google Developers Experts 2019
Web Technologies GDE



Did I say WebAssembly?

WASM for the friends...

WebAssembly, what's that?

Can I code webapps in Rust?

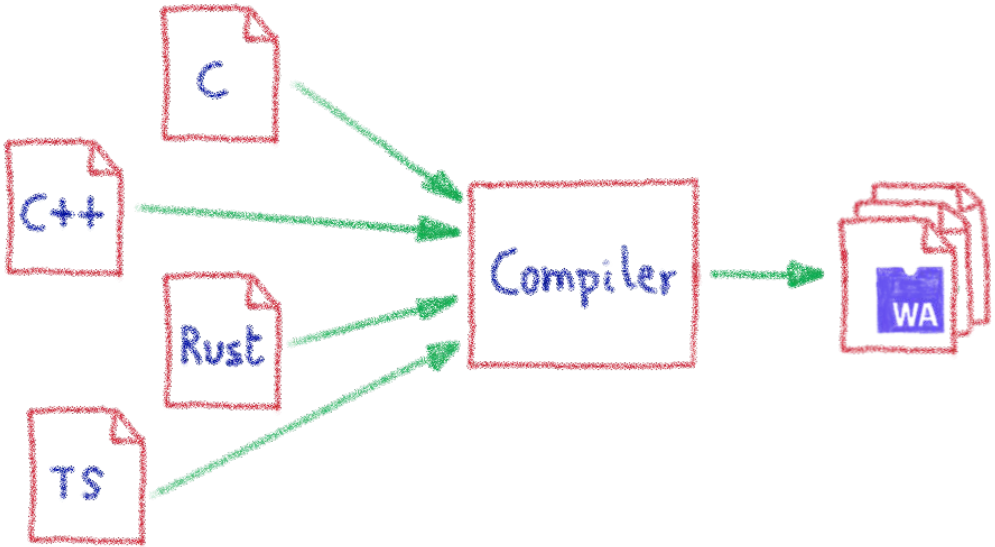
What's WASM ?

Does it replace JS ?

Is HTML/CSS/JS stack obsolete?

Let's try to answer those (and other) questions...

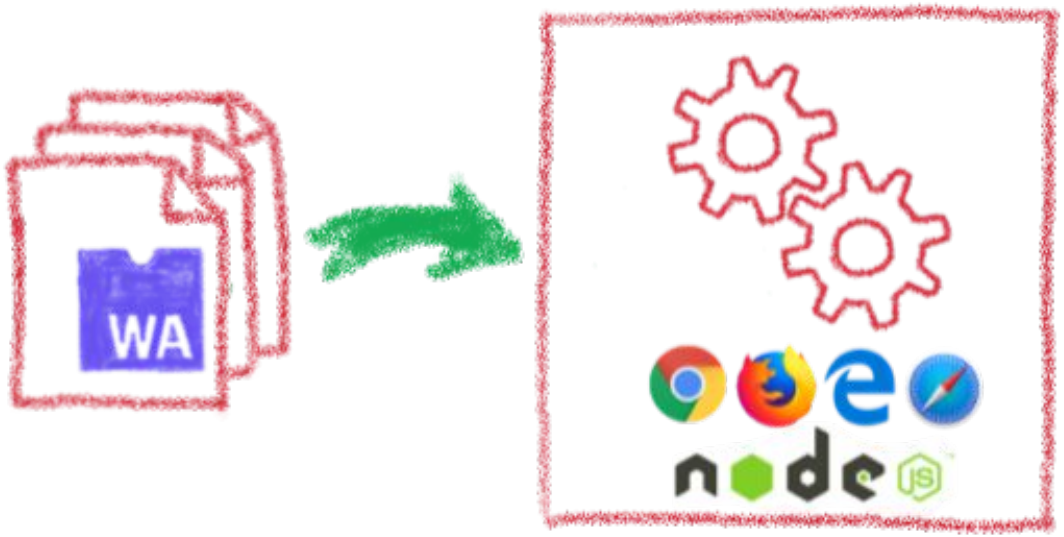
A low-level binary format for the web



Not a programming language

A compilation target

That runs on a stack-based virtual machine



A portable binary format that runs on all modern browsers...
but also on NodeJS!

With several key advantages

Fast & Efficient

Memory-safe & Sandboxed

Open & Deboggable

WWW Part of the Web Platform

But above all...

WebAssembly is not meant to replace JavaScript

Who is using WebAssembly today?

And many more others...

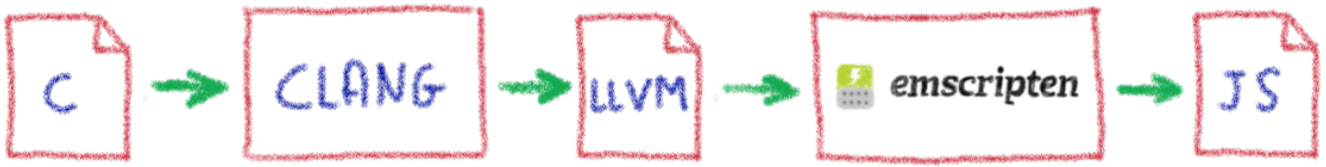
A bit of history

Remembering the past
to better understand the present

Executing other languages in the browser

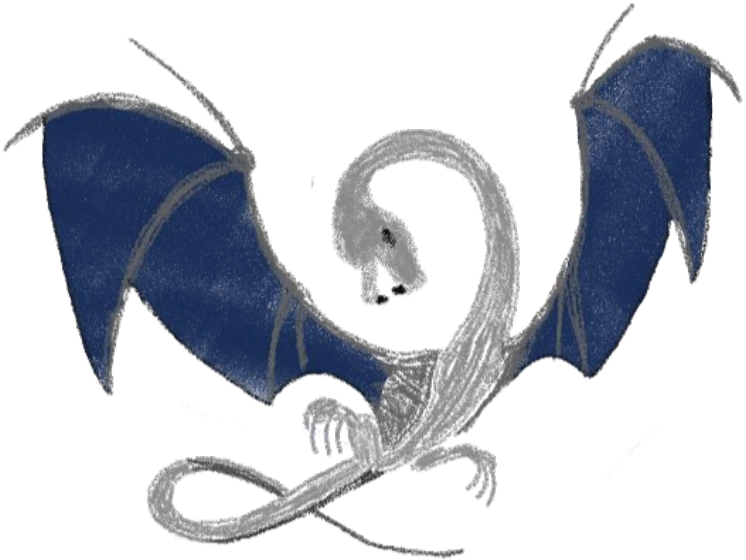
A long story, with many failures...

2012 - From C to JS: enter emscripten



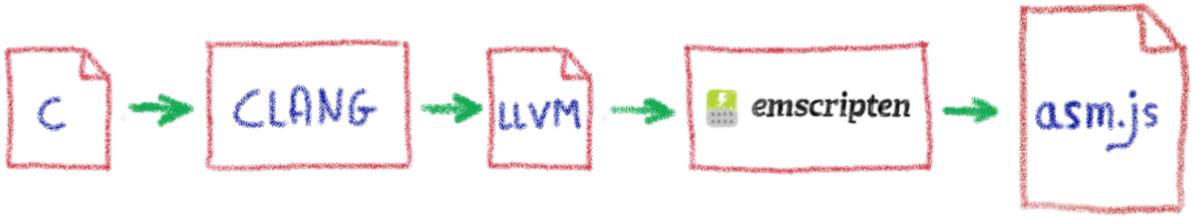
Passing by LLVM pivot

Wait, dude! What's LLVM?



A set of compiler and toolchain technologies

2013 - Generated JS is slow...



Let's use only a strict subset of JS: asm.js

Only features adapted to AOT optimization

WebAssembly project

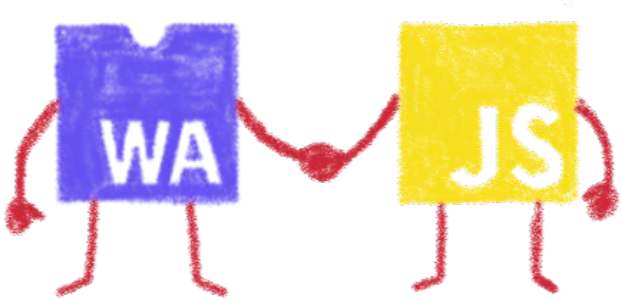
moz://a

Google

Microsoft

W3C

Joint effort



Hello W(ASM)orld

My first WebAssembly program

Do you remember your 101 C course?

```c
#include <stdio.h>

int main(int argc, char ** argv) {
  printf("Hello, world!\n");
}

```

A simple *HelloWorld* in C

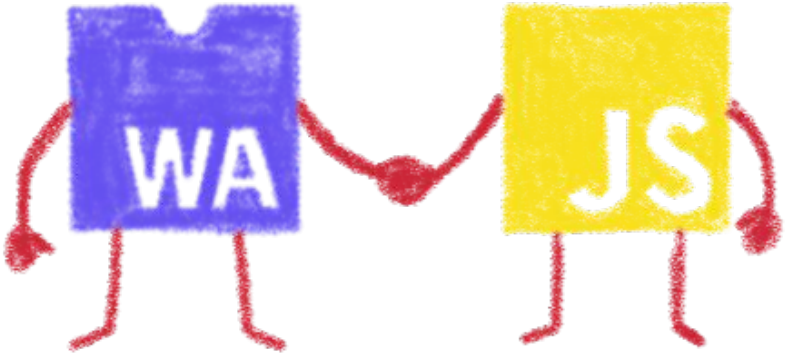
We compile it with emscripten

```
horacio@DESKTOP-6KHP1S2:~/git/wasm/hello_world$ emcc hello_world.c -o hello_world.html
cache:INFO: generating system asset: is_vanilla.txt... (this will be cached in "/home/horacio/.emscripten_cache/is_vanilla.txt" for subsequent builds)
cache:INFO:  - ok
shared:INFO: (Emscripten: Running sanity checks)
cache:INFO: generating system library: libcompiler_rt.bc... (this will be cached in "/home/horacio/.emscripten_cache/asmjs/libcompiler_rt.bc" for subsequent builds)
cache:INFO:  - ok
cache:INFO: generating system library: libc-wasm.bc... (this will be cached in "/home/horacio/.emscripten_cache/asmjs/libc-wasm.bc" for subsequent builds)
cache:INFO:  - ok
cache:INFO: generating system library: libdlmalloc.a... (this will be cached in "/home/horacio/.emscripten_cache/asmjs/libdlmalloc.a" for subsequent builds)
cache:INFO:  - ok
cache:INFO: generating system library: libpthreads_stub.bc... (this will be cached in "/home/horacio/.emscripten_cache/asmjs/libpthreads_stub.bc" for subsequent builds)
cache:INFO:  - ok
horacio@DESKTOP-6KHP1S2:~/git/wasm/hello_world$ ls
hello_world.c  hello_world.html  hello_world.js  hello_world.wasm
horacio@DESKTOP-6KHP1S2:~/git/wasm/hello_world$
```

We get a .wasm file...

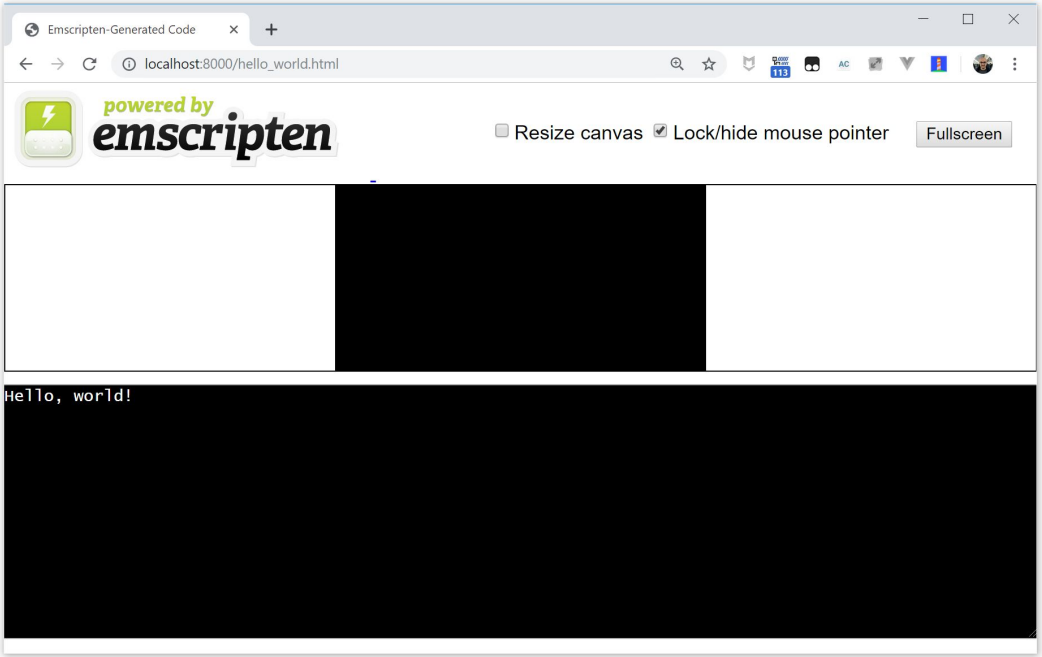
Binary file, in the binary WASM format

We also get a `.js` file...



Wrapping the WASM

And a .html file



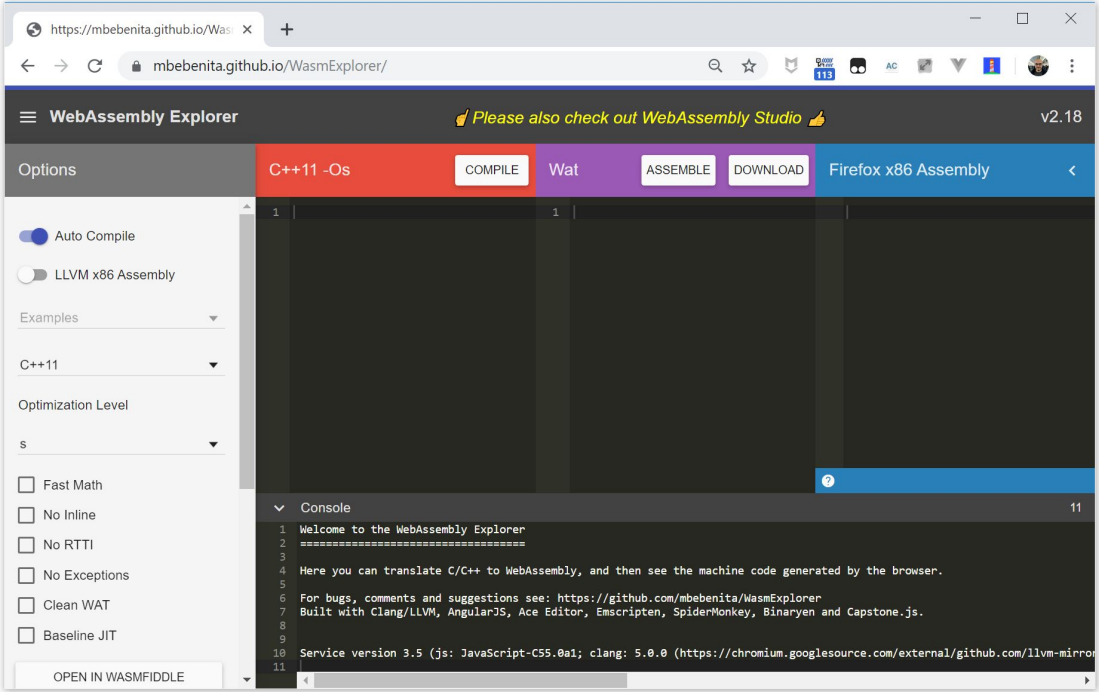
To quickly execute in the browser our WASM

And in a more Real World™ case?

A simple process:

- Write or use existing code
  - In C, C++, Rust, Go, AssemblyScript...
- Compile
  - Get a binary `.wasm` file
- Include
  - The `.wasm` file into a project
- Instantiate
  - Async JavaScript compiling and instantiating the `.wasm` binary

I think I need a real example now



Let's use WASM Explorer

https://mbebenita.github.io/WasmExplorer/

Let's begin with the a simple function

C++11 -Os | COMPILE

```
1  int squarer(int num) {
2    return num * num;
3  }
```

Wat | ASSEMBLE | DOWNLOAD

```
1  (module
2    (type $type0 (func (param i32)
       (result i32)))
3    (table 0 anyfunc)
4    (memory 1)
5    (export "memory" memory)
6    (export "_Z7squareri" $func0)
7    (func $func0 (param $var0 i32)
       (result i32)
8      get_local $var0
9      get_local $var0
10     i32.mul
11   )
12 )
```

Firefox x86 Assembly

```
wasm-function[0]:
  sub rsp, 8
  mov edx, edi
  mov ecx, edx
  mov eax, edx
  imul ecx, eax
  mov eax, ecx
  nop
  add rsp, 8
  ret
```

WAT: WebAssembly Text Format

Human readable version of the `.wasm` binary

Download the binary `.wasm` file

Now we need to call it from JS...

Instantiating the WASM

1. Get the `.wasm` binary file into an array buffer
2. Compile the bytes into a WebAssembly module
3. Instantiate the WebAssembly module

Instantiating the WASM

wasm > squarer > JS squarer.js > ...

```
3   var importObject = {
4       imports: {
5         imported_func: function(arg) {
6           console.log(arg);
7         }
8       }
9     };
10
11  async function loadWebAssembly() {
12      let response = await fetch('squarer.wasm');
13      let arrayBuffer = await response.arrayBuffer();
14      let wasmModule = await WebAssembly.instantiate(arrayBuffer, importObject);
15      squarer = await wasmModule.instance.exports._Z7squareri;
16      console.log('Finished compiling! Ready when you are...');
17  }
18
19  loadWebAssembly();
```

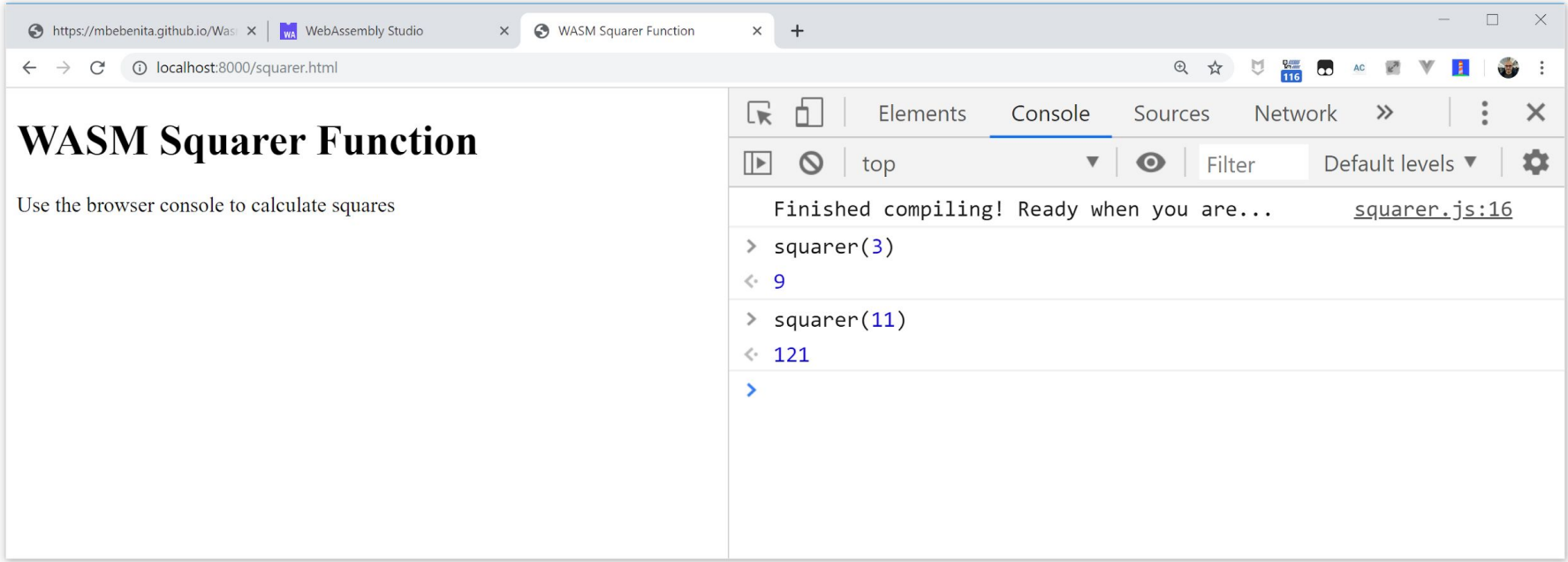
Loading the squarer function

wasm > squarer > squarer.html > ...

```
1  <!DOCTYPE html>
2  <html>
3  <head>
4      <meta charset="utf-8" />
5      <meta http-equiv="X-UA-Compatible" content="IE=edge">
6      <title>WASM Squarer Function</title>
7      <meta name="viewport" content="width=device-width, initial-scale=1">
8  </head>
9  <body>
10
11     <h1>WASM Squarer Function</h1>
12
13     <script src="squarer.js"></script>
14
15     <p>Use the browser console to calculate squares</p>
16 </body>
17 </html>
18
19
```

We instantiate the WASM by loading the wrapping JS

Using it!



Directly from the browser console (it's a simple demo...)

WASM outside the browser

Not only for web developers

Run any code on any client... almost

Languages compiling to WASM

Includes WAPM

wapm install optipng

Oh, like npm for WASM!

The WebAssembly Package Manager

Some Use Cases

What can I do with this?

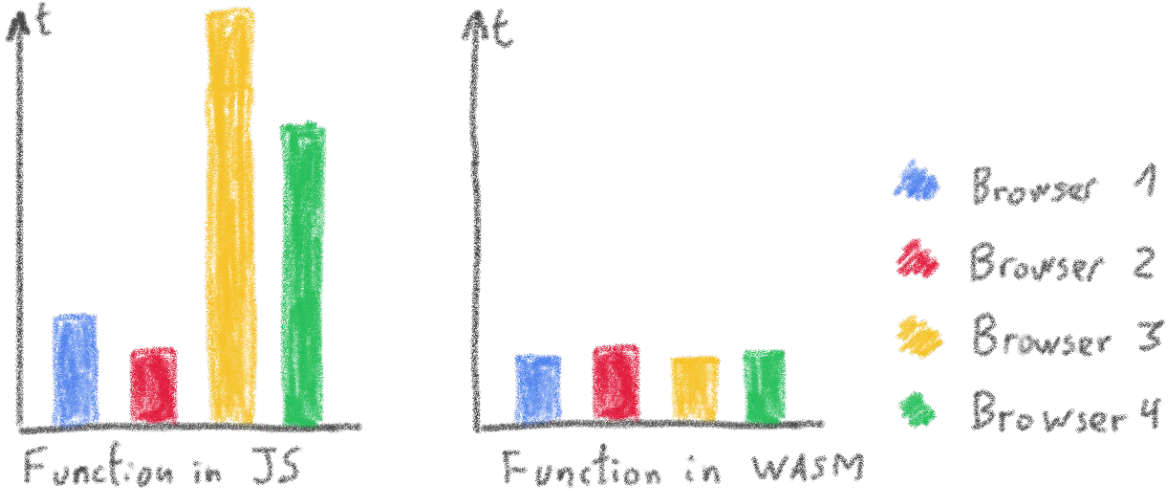
Tapping into other languages ecosystems

SQUOOSH.APP

OptiPNG (C)
Resize (Rust)
MozJPEG (C++)
webp (C)

Don't rewrite libs anymore

Replacing problematic JS bits



Predictable performance

Same peak performance, but less variation

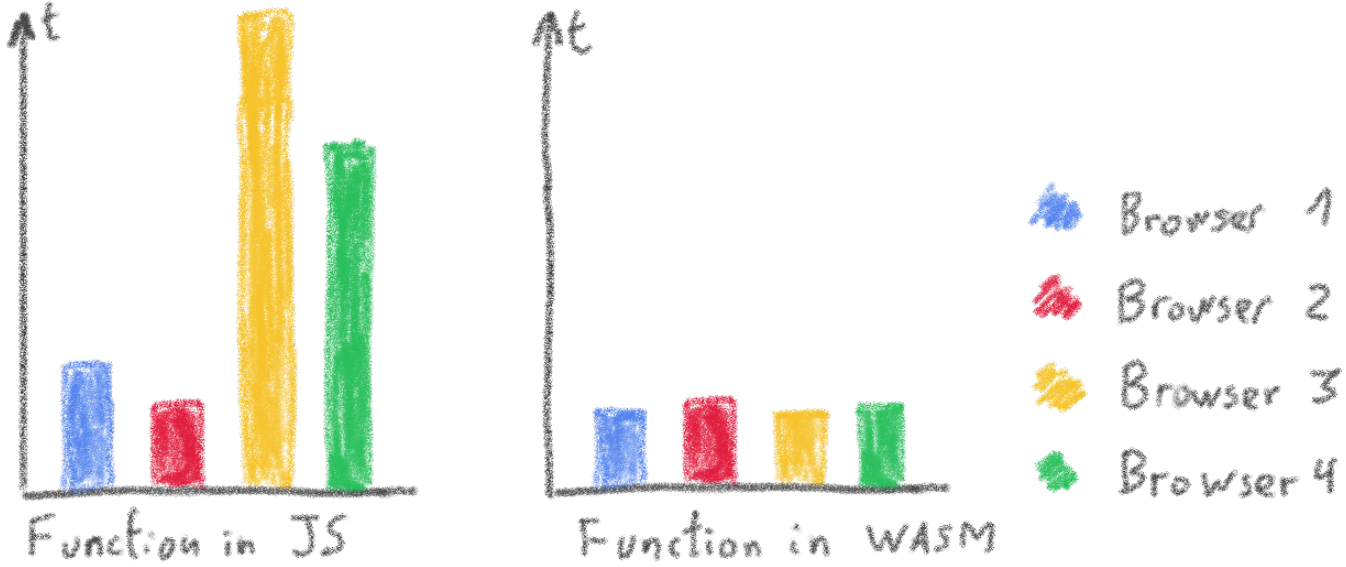
AssemblyScript

Writing WASM without learning a new language

TypeScript subset compiled to WASM

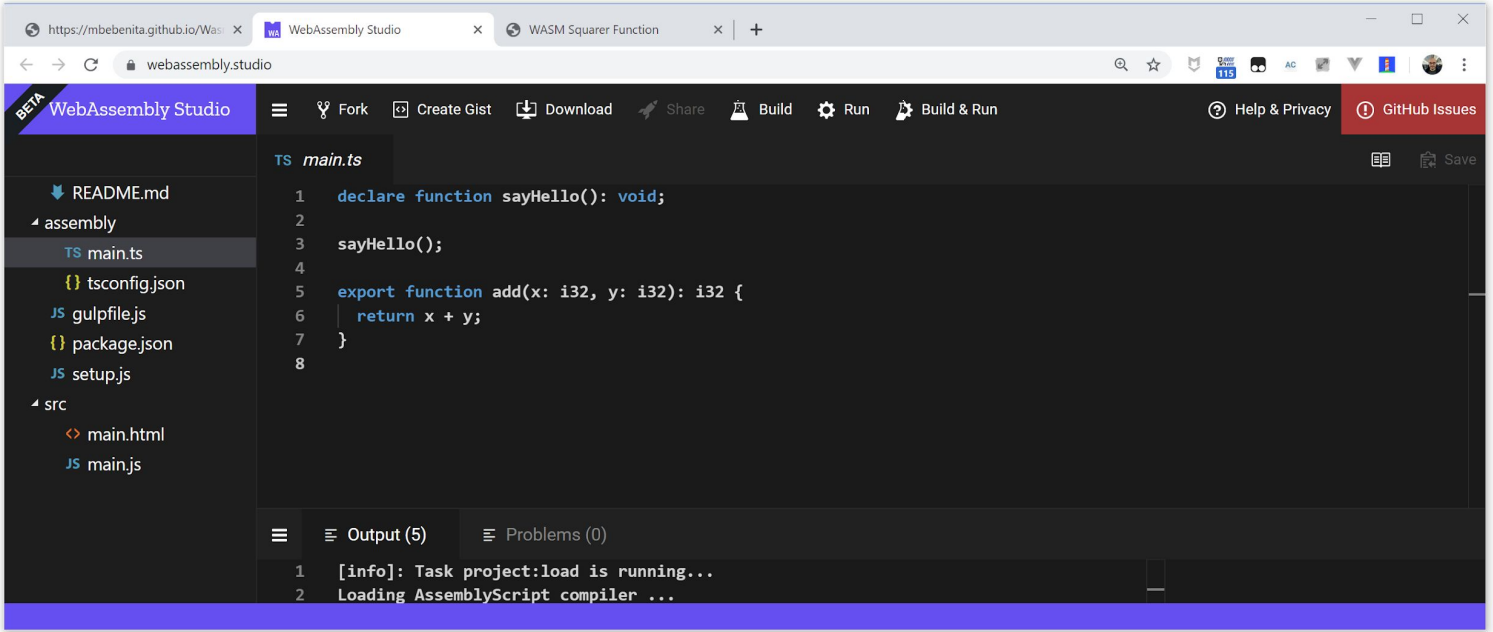
Why would I want to compile TypeScript to WASM?

Ahead of Time compiled TypeScript



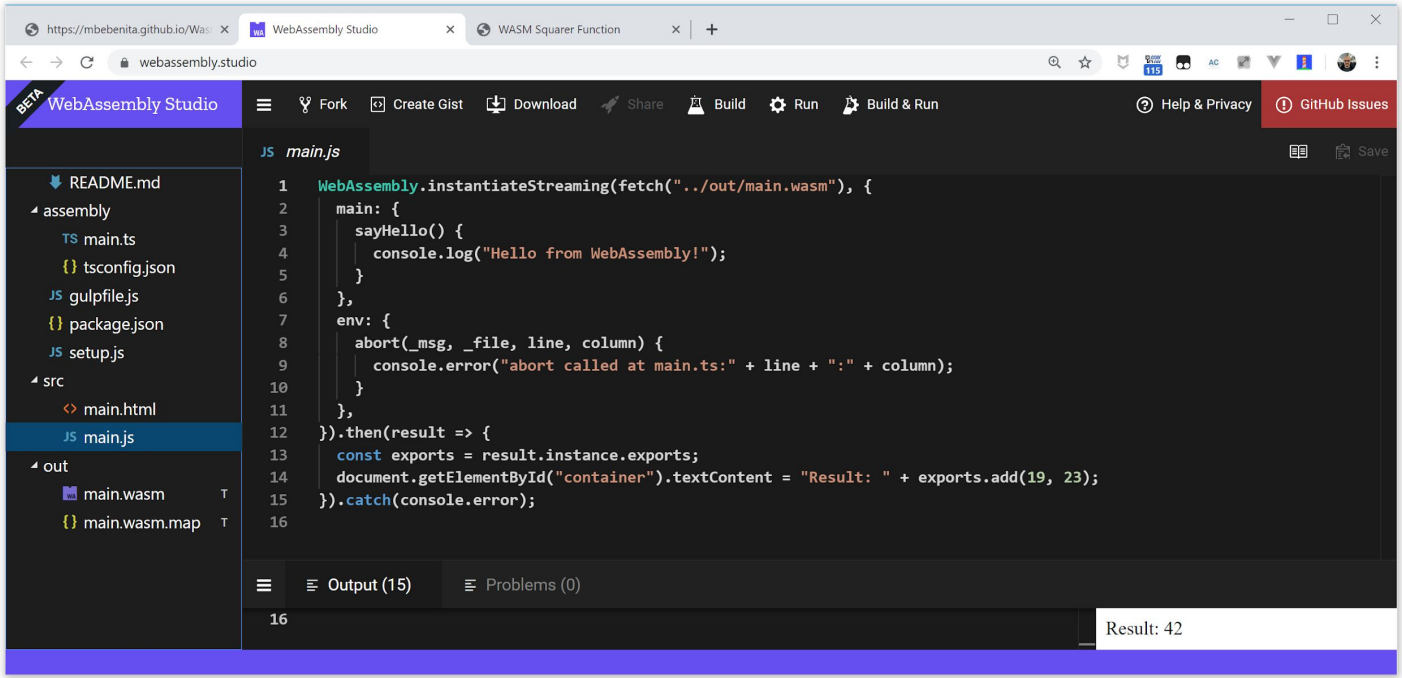
More predictable performance

Avoiding the dynamicness of JavaScrip



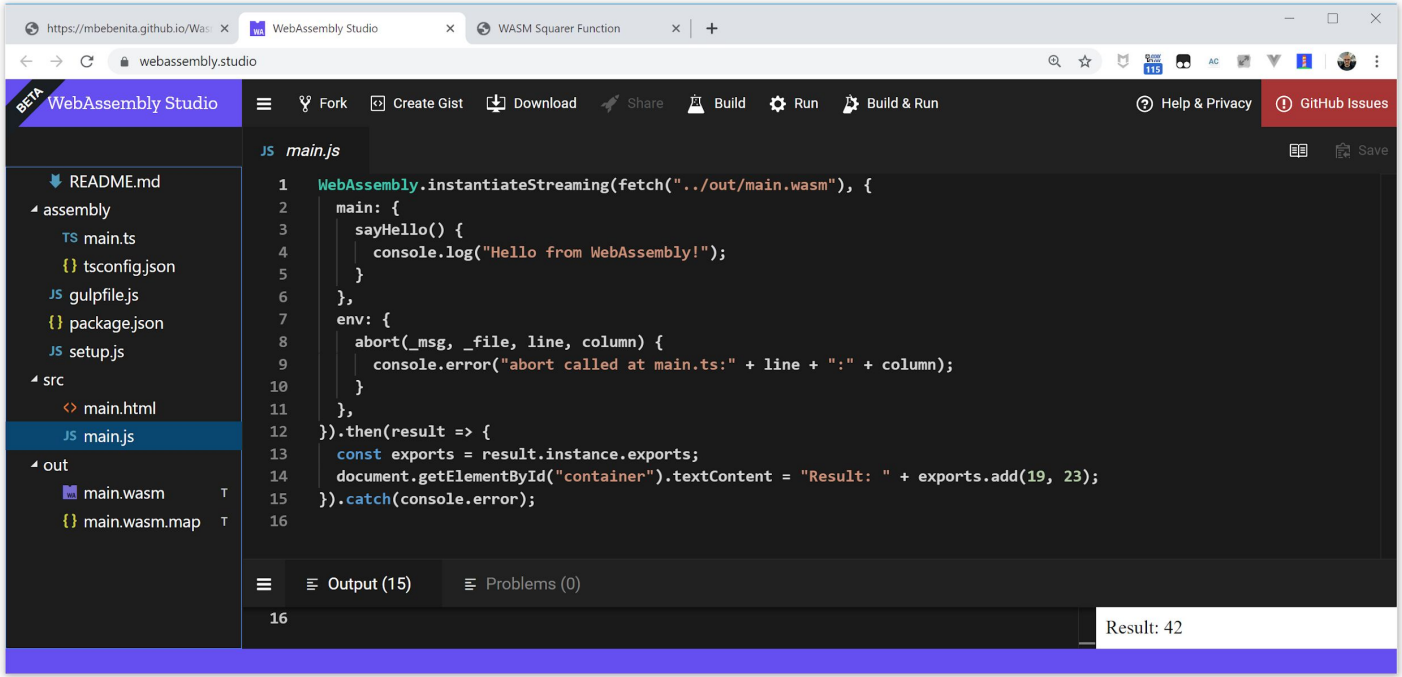
More specific integer and floating point types

Objects cannot flow in and out of WASM yet

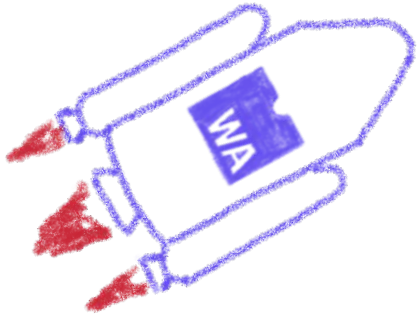


Using a loader to write/read them to/from memory

No direct access to DOM



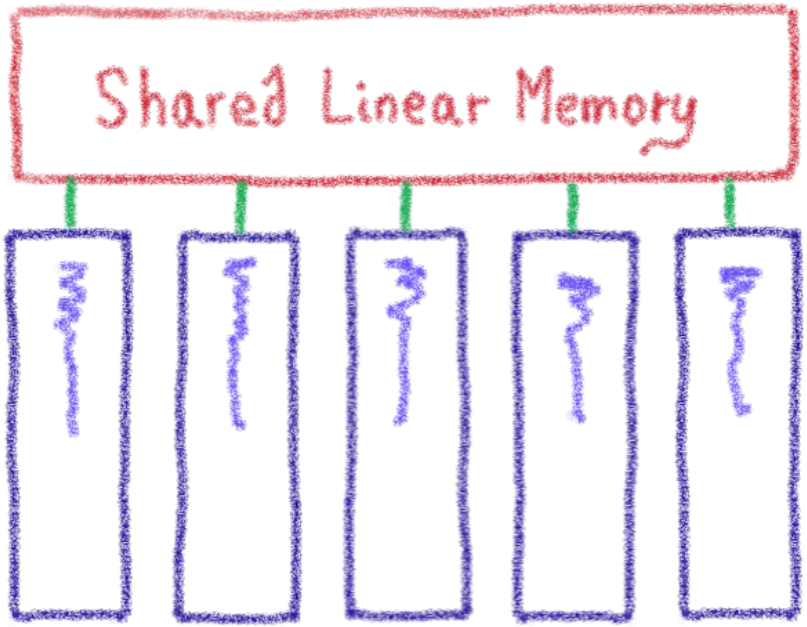
Glue code using exports/imports to/from JavaScript



Future

To the infinity and beyond!

WebAssembly Threads



Threads on Web Workers with shared linear memory

SIMD

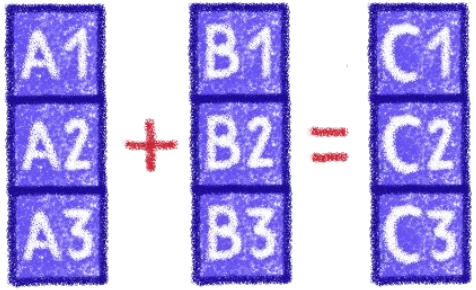
Multiple scalar operations

A1 + B1 = C1

A2 + B2 = C2

A3 + B3 = C3

Single vectorial operation



Already available in Wasmer

Single Instruction, Multiple Data

Garbage collector

And exception handling