

# DEVFEST NANTES 2019



## WebAssembly Codelab



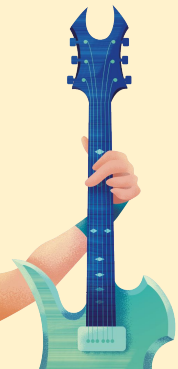
Stéphanie Moallic

@steffy\_29

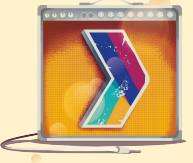


Horacio Gonzalez

@LostInBrittany

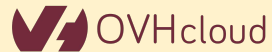


# Horacio Gonzalez




@LostInBrittany


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



# Stéphanie Moallic



 @steffy\_29

 Steffy29

“La dame du téléphone” - Quentin Adam



Passionnée d'informatique mais pas que...



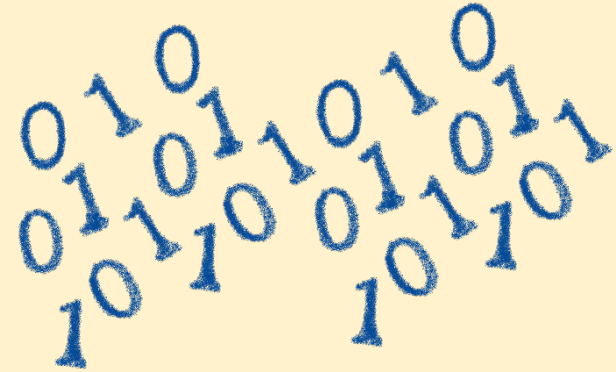
Duchess



Développeuse front Telecom chez OVHcloud

Organisatrice d'évènements pour les développeurs et les enfants.

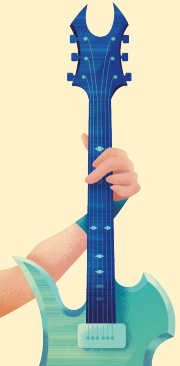
Prédilection pour le développement front ainsi que les gadgets et autres jouets.



# How is the codelab structured?

---

What are we coding today?



# A GitHub repository

A screenshot of a GitHub repository page for 'wasm-codelab' by user 'LostInBrittany'. The page shows the repository's overview, including 18 commits, 1 branch, and 0 releases. A table lists the commit history with columns for the commit message and the time since the commit. The 'README.md' file is selected and its content is displayed below, which includes the title 'DevFest Nantes 2019 WebAssembly Codelab' and introductory text about the tutorial's objectives and requirements.

Commit	Message	Time
Latest commit	Game of Life working, even in local	1 hour ago
	Step-02 and init Step-01	10 hours ago
	Project README done	14 hours ago
	Fixing typos	8 hours ago
	Fixing links	8 hours ago
	Fixing bugs and typos, adding offline version of step-04	6 hours ago
	Fixing bugs and typos, adding offline version of step-04	6 hours ago
	Game of Life working, even in local	1 hour ago
	Fixing links	8 hours ago

**DevFest Nantes 2019 WebAssembly Codelab**

We have built this [WebAssembly Codelab](#) as a quick entry point to [WebAssembly](#).

### What are the objectives of this tutorial

Follow the tutorial to learn the concepts behind WebAssembly (WASM), write your first WASM libraries, compile existing libraries to WASM and generally understand how WASM open new possibilities in the web development ecosystem.

### What do I need to use this tutorial?

The tools strictly needed for this tutorial are a modern web browser (ideally [Chrome](#) or [Chromium](#)), a text editor (we suggest the excellent [Visual Studio Code](#)), [Node JS](#), and a web-server to test your code.

<https://github.com/LostInBrittany/wasm-codelab>



# Nothing to install

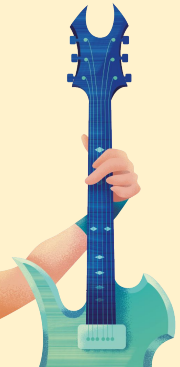


```
C++11 -Os      COMPILER  Wat      ASSEMBLER  DOWNLOAD  Firefox x86 Assembly  <
```

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

```
1 (module
2   (type $type0 (func (param i32)
3     (result i32)))
4   (table 0 anyfunc)
5   (memory 1)
6   (export "memory" memory)
7   (export "_Z7squarer_i" $func0)
8   (func $func0 (param $var0 i32)
9     (result i32)
10    get_local $var0
11    get_local $var0
12    i32.mul
13  )
14 )
```

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



Using WebAssembly Explorer  
and WebAssembly Studio

# Only additional tool: a web server



Because of the browser security model



# Procedure: follow the steps



Step by step



# But before coding, let's speak



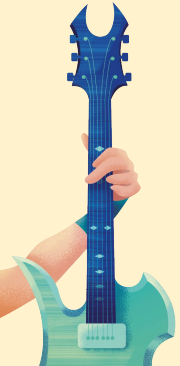
What's this WebAssembly thing?



# Did we say WebAssembly?

---

WASM for the friends...



# WebAssembly, what's that?



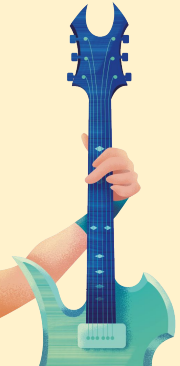
Can I code webapps in Rust?

What's WASM?

Does it replace JS?

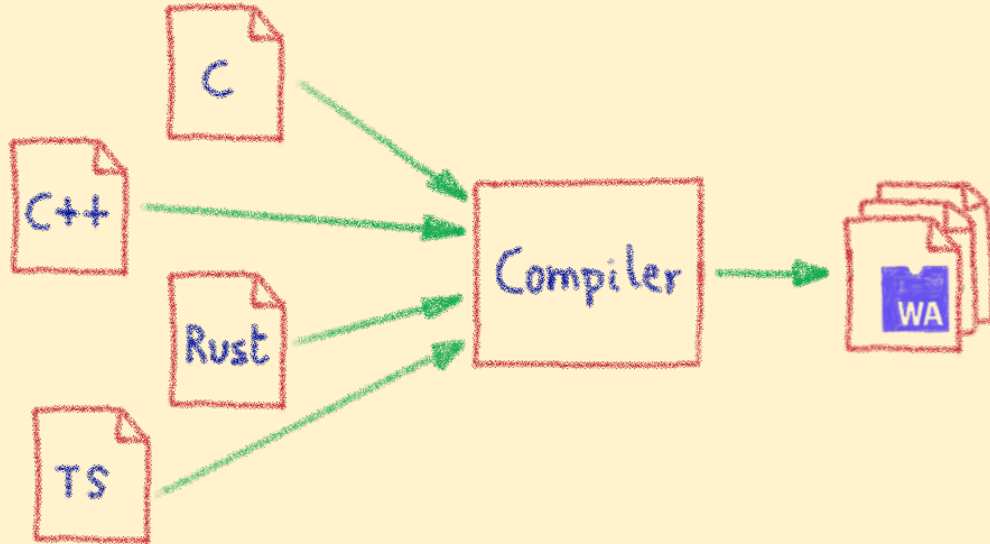


Is HTML/CSS/JS stack obsolete?



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

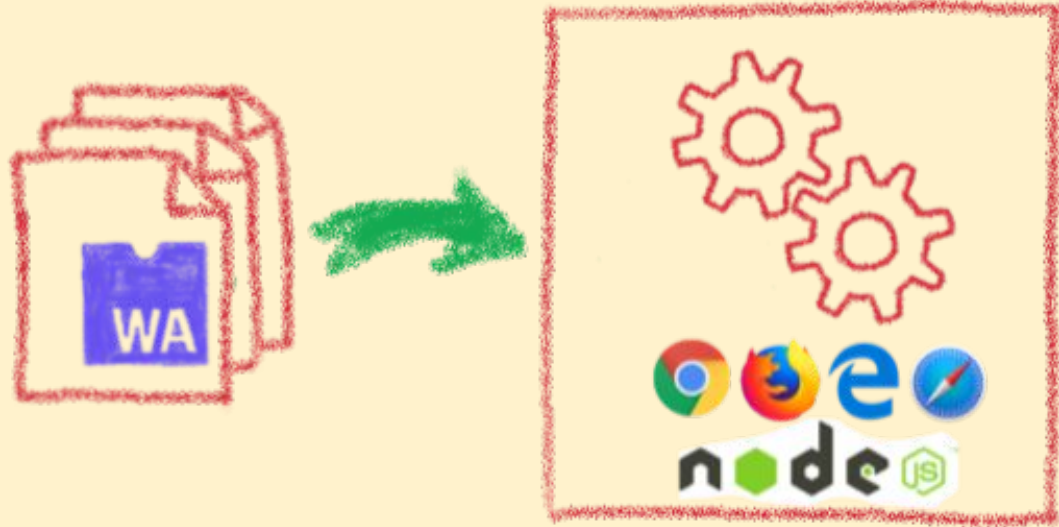
# A low-level binary format for the web



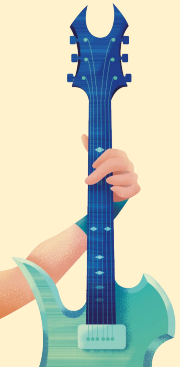
Not a programming language  
A compilation target



# 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 ⚡

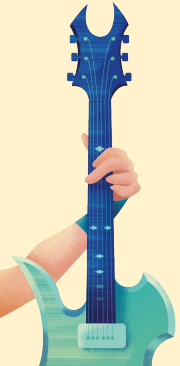
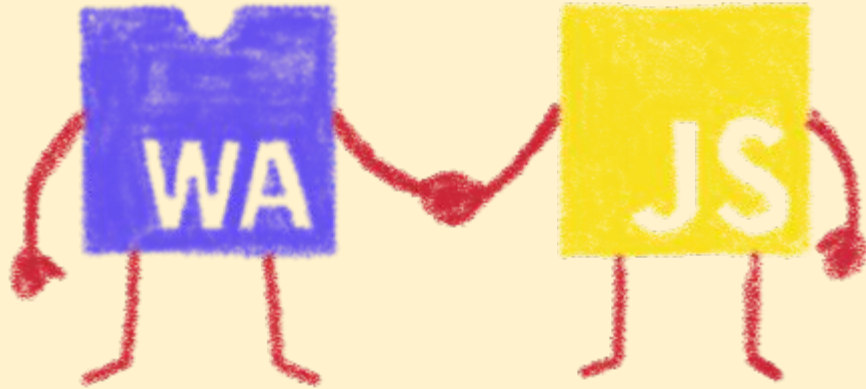
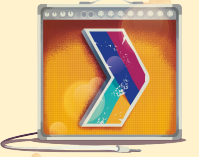
🔒 Memory-safe & Sandboxed

Open & Debuggable 📄

www Part of the Web Platform

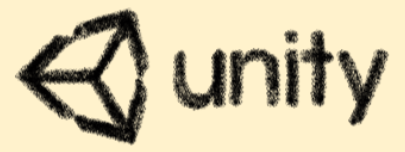
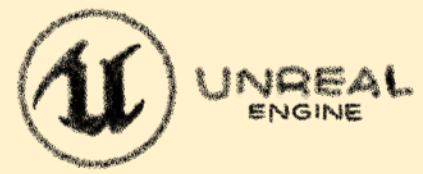


# But above all...



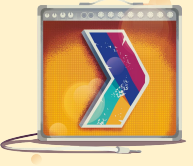
WebAssembly is not meant to replace JavaScript

# Who is using WebAssembly today?



And many more others...





# A bit of history

---

Remembering the past  
to better understand the present



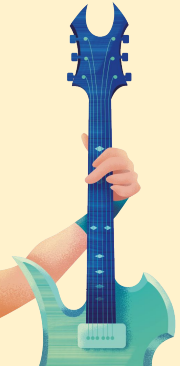
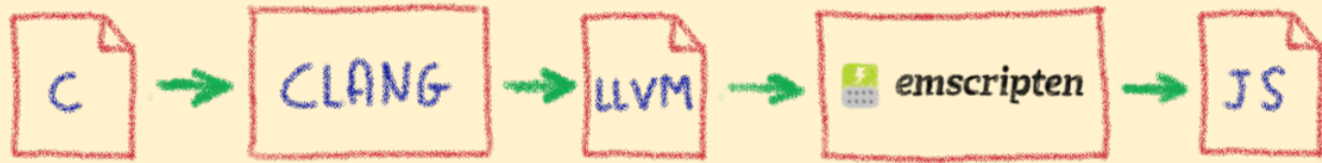
# Executing other languages in the browser



A long story, with many failures...

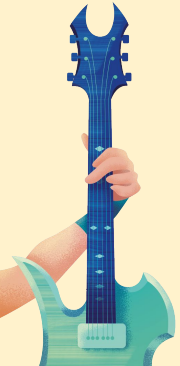
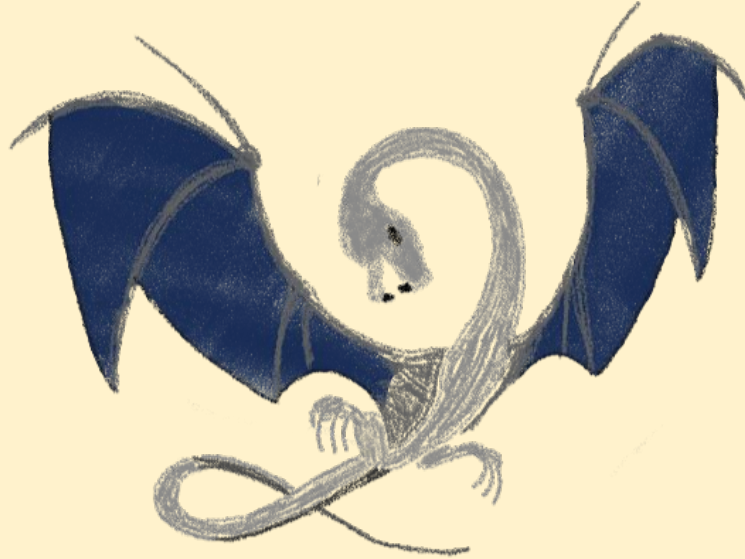
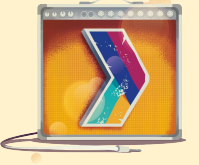


# 2012 - From C to JS: enter emscripten



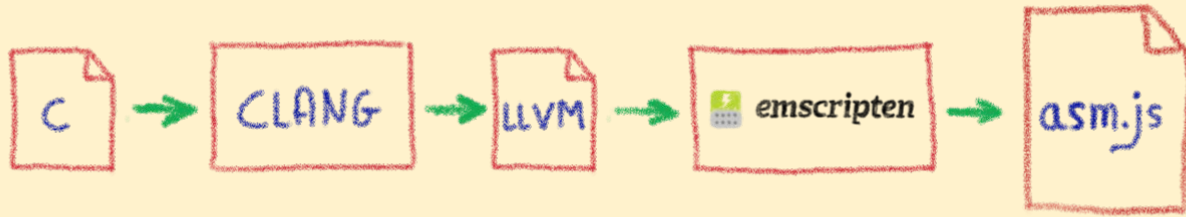
Passing by LLVM pivot

# Wait, dude! What's LLVM?

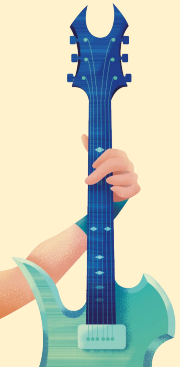


A set of compiler and toolchain technologies

# 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

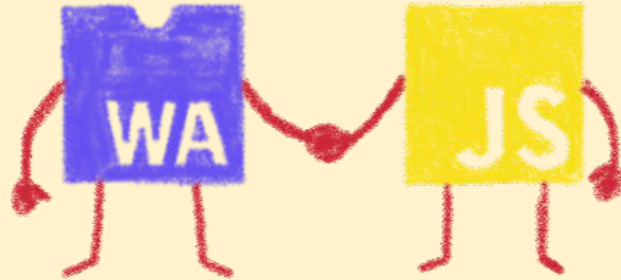
Google



W3C

Joint effort

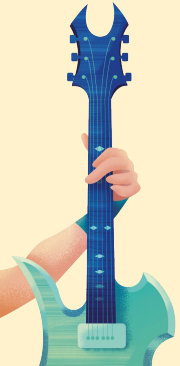




# Hello W(ASM)orld

---

My first WebAssembly program



# Do you remember your 101 C course?



```
1  #include <stdio.h>
2
3  int main(int argc, char ** argv) {
4  |  printf("Hello, world!\n");
5  |  }
6  |
```



A simple *HelloWorld* in C



# We compile it with emscripten



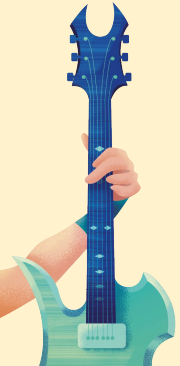
```
horacio@DESKTOP-6KHP1S2: ~/git/wasm/hello_world x horacio@DESKTOP-6KHP1S2: ~/git/emscripten x + v
horacio@DESKTOP-6KHP1S2:~/git/wasm/hello_world$ emcc hello_world.c -o hello_world.html
cache:INFO: generating system asset: is_vanilla.txt... (this will be cached in "/home/horacio/.emscripten_cache/is_vanilla.txt" 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: libpthread_stub.bc... (this will be cached in "/home/horacio/.emscripten_cache/asmjs/libpthread_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...

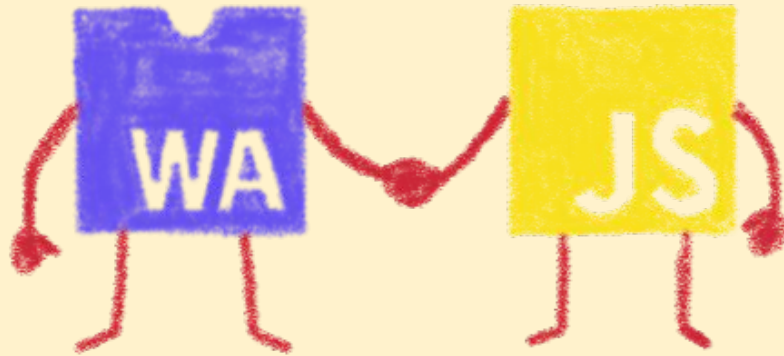


01011010  
01101101010  
101100110110



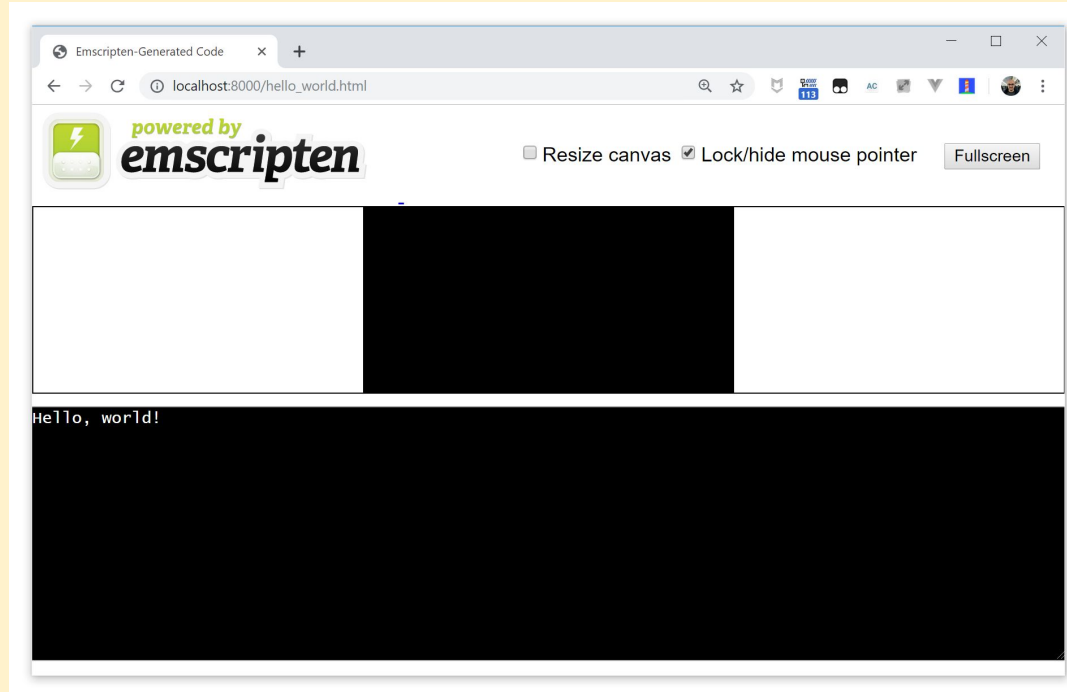
Binary file, in the binary WASM format

# 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?

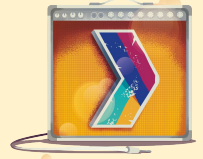


A simple process:

- 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

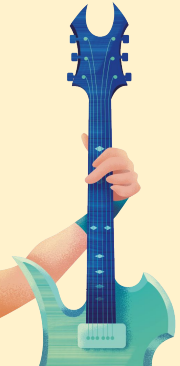


# I don't want to install a compiler now...

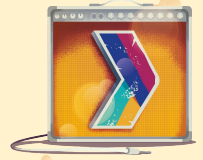
A screenshot of the WebAssembly Explorer web application. The browser address bar shows the URL <https://mbebenita.github.io/WasmExplorer/>. The application interface includes a navigation bar with 'WebAssembly Explorer' and a version number 'v2.18'. Below the navigation bar, there are tabs for 'C++11 -Os', 'Wasm', 'ASSEMBLE', and 'DOWNLOAD'. The main area is a code editor with a dark background, showing a welcome message in the console. The console output includes: '1 Welcome to the WebAssembly Explorer', '2 =====', '3', '4 Here you can translate C/C++ to WebAssembly, and then see the machine code generated by the browser.', '5', '6 For bugs, comments and suggestions see: https://github.com/mbebenita/WasmExplorer', '7 Built with Clang/LLVM, AngularJS, Ace Editor, Emscripten, SpiderMonkey, Binaryen and Capstone.js.', '8', '9', '10 Service version 3.5 (js: JavaScript-C55.0a1; clang: 5.0.0 (https://chromium.googlesource.com/external/github.com/llvm-mirror', '11'. On the left side, there are options for 'Auto Compile' (checked), 'LLVM x86 Assembly' (unchecked), 'Examples' (dropdown), 'C++11' (dropdown), 'Optimization Level' (dropdown), and several checkboxes for 'Fast Math', 'No Inline', 'No RTTI', 'No Exceptions', 'Clean WAT', and 'Baseline JIT'. At the bottom left of the options panel, there is a button labeled 'OPEN IN WASMFIDDLE'.

Let's use WASM Explorer

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



# Let's begin with the a simple function



C++11 -Os	COMPILE	Wat	ASSEMBLE	DOWNLOAD	Firefox x86 Assembly <
<pre>1 int squarer(int num) { 2     return num * num; 3 }</pre>		<pre>1 (module 2   (type \$type0 (func (param i32) 3     (result i32))) 4   (table 0 anyfunc) 5   (memory 1) 6   (export "memory" memory) 7   (export "_Z7squareri" \$func0) 8   (func \$func0 (param \$var0 i32) 9     (result i32) 10    get_local \$var0 11    get_local \$var0 12    i32.mul 13  ) 14 )</pre>			<pre>1 wasm-function[0]: 2   sub rsp, 8 3   mov edx, edi 4   mov ecx, edx 5   mov eax, edx 6   imul ecx, eax 7   mov eax, ecx 8   nop 9   add rsp, 8 10  ret</pre>

WAT: WebAssembly Text Format

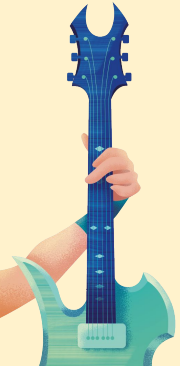
Human readable version of the .wasm binary



# Download the binary .wasm file



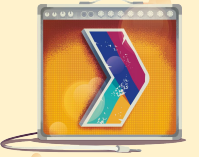
01011010  
0110110101  
1011001101



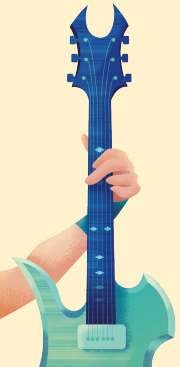
Now we need to call it from JS...



# 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();
20
```



# 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!



WASM Squarer Function

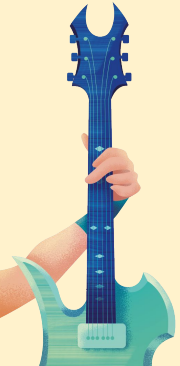
Use the browser console to calculate squares

```
Finished compiling! Ready when you are... squarer.js:16
> squarer(3)
< 9
> squarer(11)
< 121
>
```

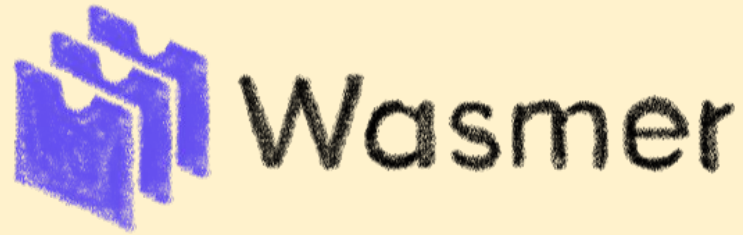
Directly from the browser console  
(it's a simple demo...)



# You can do steps 01 and 02 now



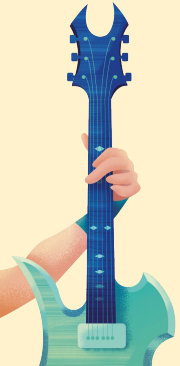
Let's code, mates!



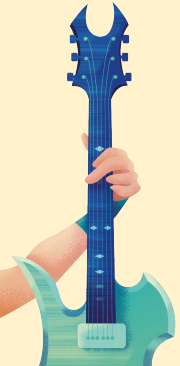
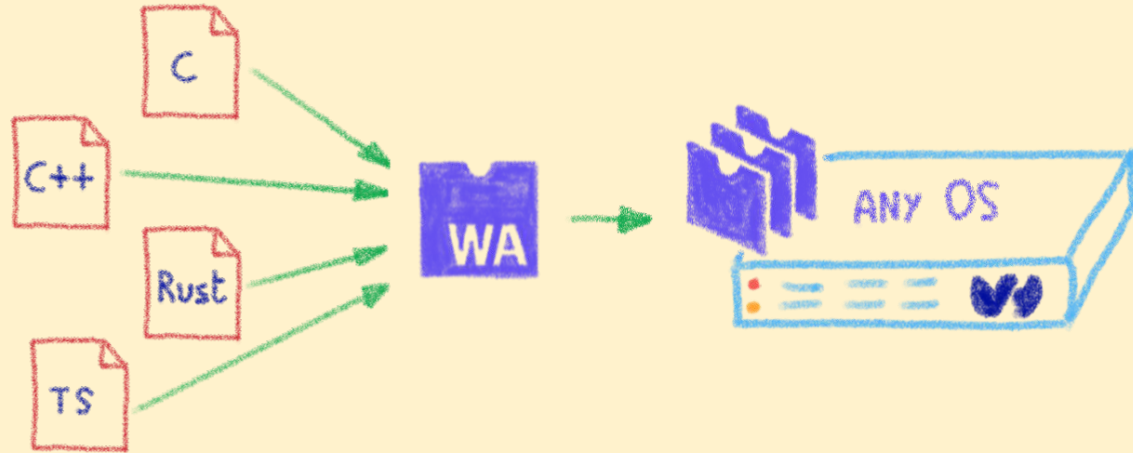
# WASM outside the browser

---

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!*



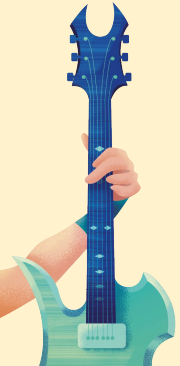
The WebAssembly Package Manager



# Some use cases

---

What can I do with it?



# Tapping into other languages ecosystems



SQUOSH.APP

OptiPNG (C)

Resize (Rust)

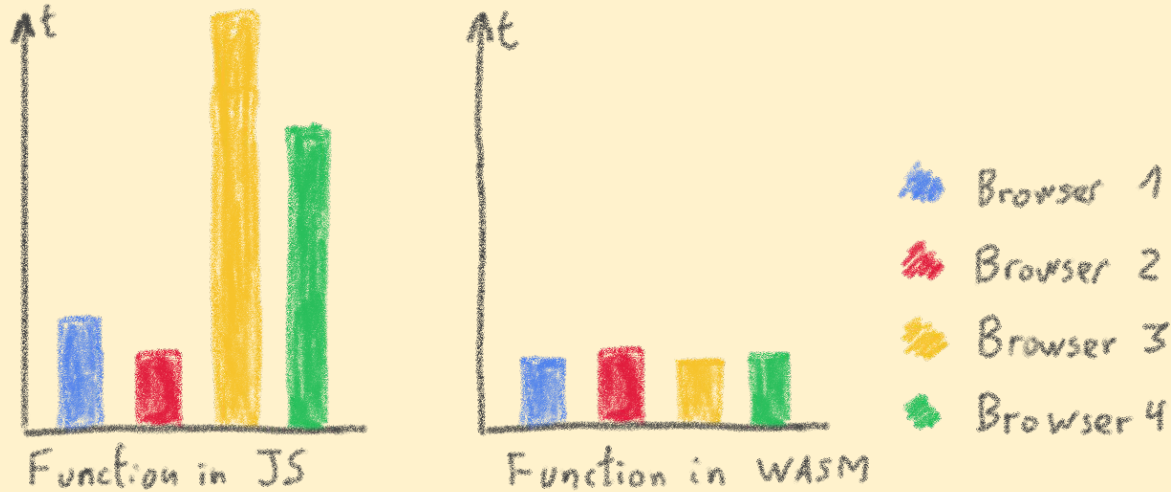
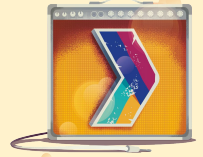
MozJPEG (C++)

webp (C)

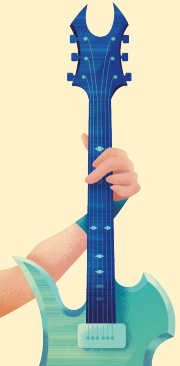
Don't rewrite libs anymore



# Replacing problematic JS bits



Predictable performance  
Same peak performance, but less variation

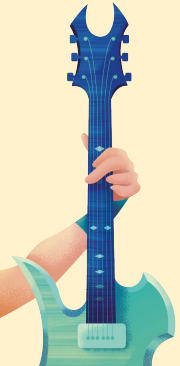




# Communicating between JS and WASM

---

Shared memory, functions...



# Native WASM types are limited



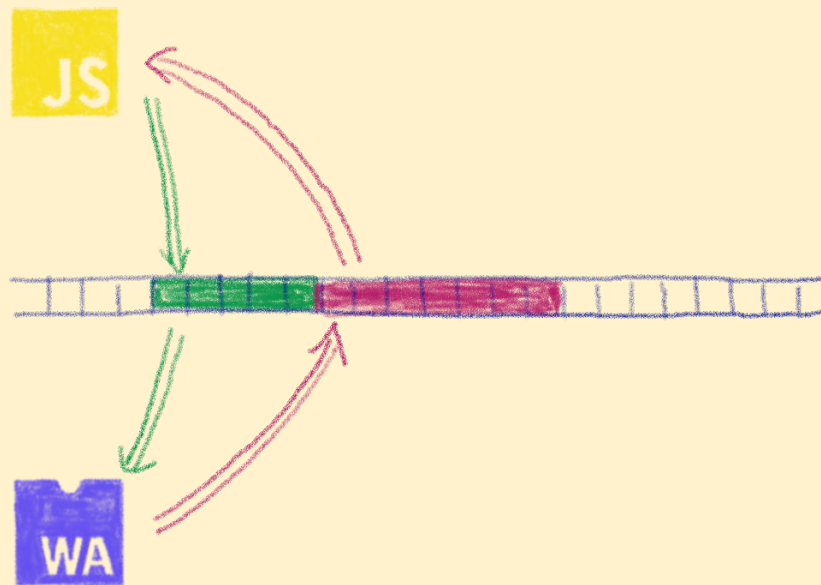
WASM currently has four available types:

- **i32**: 32-bit integer
- **i64**: 64-bit integer
- **f32**: 32-bit float
- **f64**: 64-bit float



Types from languages compiled to WASM are mapped to these types

# How can we share data?



Using the same data in WASM and JS?  
Shared linear memory between them!

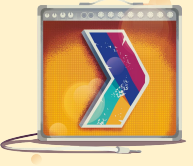


# You can do steps 03 and 04 now



Let's code, mates!





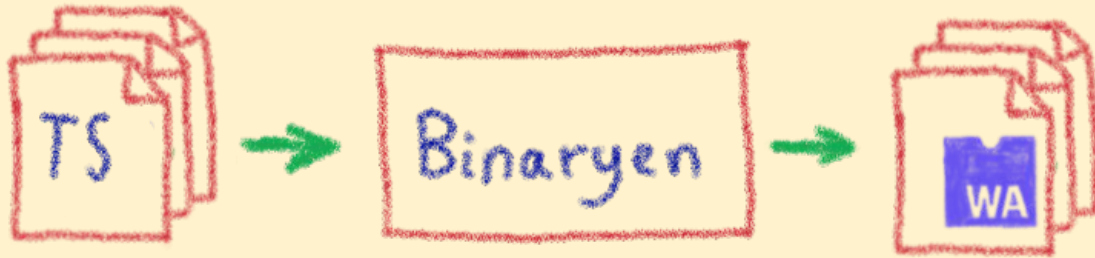
# AssemblyScript

---

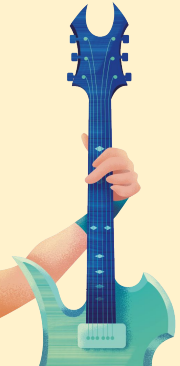
Writing WASM without learning a new language



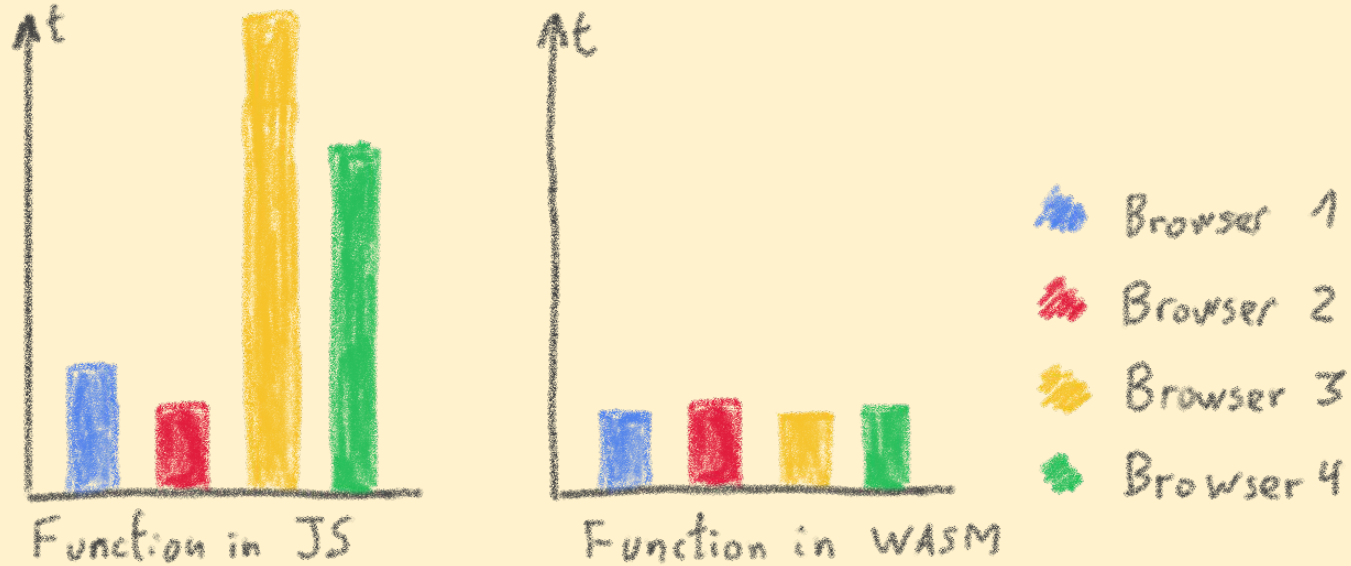
# TypeScript subset compiled to WASM



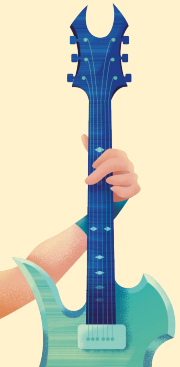
Why would I want to compile TypeScript to WASM?



# Ahead of Time compiled TypeScript



More predictable performance



# Avoiding the dynamicness of JavaScript



```
TS main.ts
1 declare function sayHello(): void;
2
3 sayHello();
4
5 export function add(x: i32, y: i32): i32 {
6     return x + y;
7 }
8
```

Output (5) Problems (0)

```
1 [info]: Task project:load is running...
2 Loading AssemblyScript compiler ...
```



More specific integer and floating point types

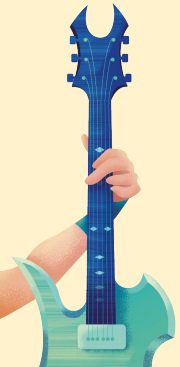
# Objects cannot flow in and out of WASM yet



```
1 WebAssembly.instantiateStreaming(fetch("../out/main.wasm"), {
2   main: {
3     sayHello() {
4       console.log("Hello from WebAssembly!");
5     }
6   },
7   env: {
8     abort(_msg, _file, line, column) {
9       console.error("abort called at main.ts:" + line + ":" + column);
10    }
11  },
12 }).then(result => {
13   const exports = result.instance.exports;
14   document.getElementById("container").textContent = "Result: " + exports.add(19, 23);
15 }).catch(console.error);
16
```

Output (15) Problems (0)

16 Result: 42



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

# No direct access to DOM



```
JS main.js
1  WebAssembly.instantiateStreaming(fetch("../out/main.wasm"), {
2    main: {
3      sayHello() {
4        console.log("Hello from WebAssembly!");
5      }
6    },
7    env: {
8      abort(msg, _file, line, column) {
9        console.error("abort called at main.ts:" + line + ":" + column);
10     }
11   },
12 }).then(result => {
13   const exports = result.instance.exports;
14   document.getElementById("container").textContent = "Result: " + exports.add(19, 23);
15 }).catch(console.error);
16
```

Output (15) Problems (0)

16 Result: 42

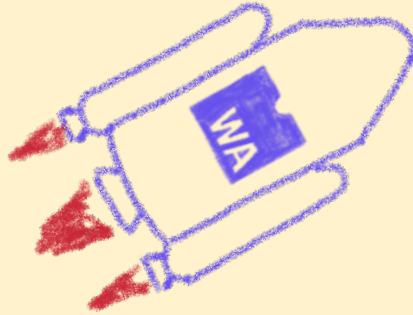


Glue code using exports/imports to/from JavaScript

# You can do step 05 now



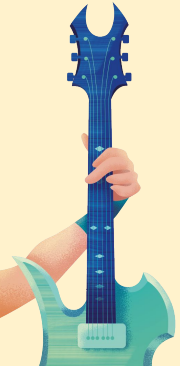
Let's code, mates!



# Future

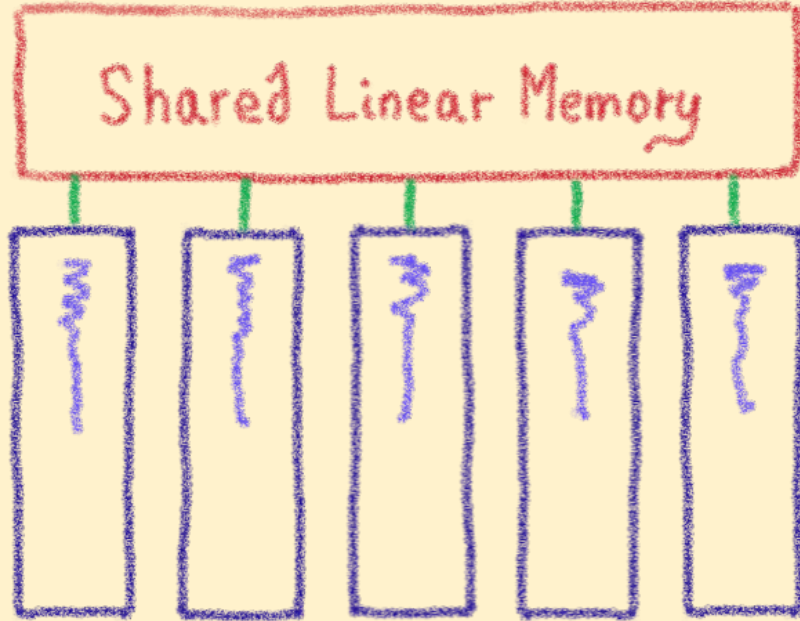
---

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

$$\begin{matrix} A1 \\ A2 \\ A3 \end{matrix} + \begin{matrix} B1 \\ B2 \\ B3 \end{matrix} = \begin{matrix} C1 \\ C2 \\ C3 \end{matrix}$$

Single Instruction, Multiple Data

Already available  
in  Wasmer



# Garbage collector



And exception handling

