QCon NYC, 26 June 2019

Rust, WebAssembly, and JavaScript make three: An FFI Story

@ag_dubs
Rust Core Team
RustWasm WG

An FFI Story

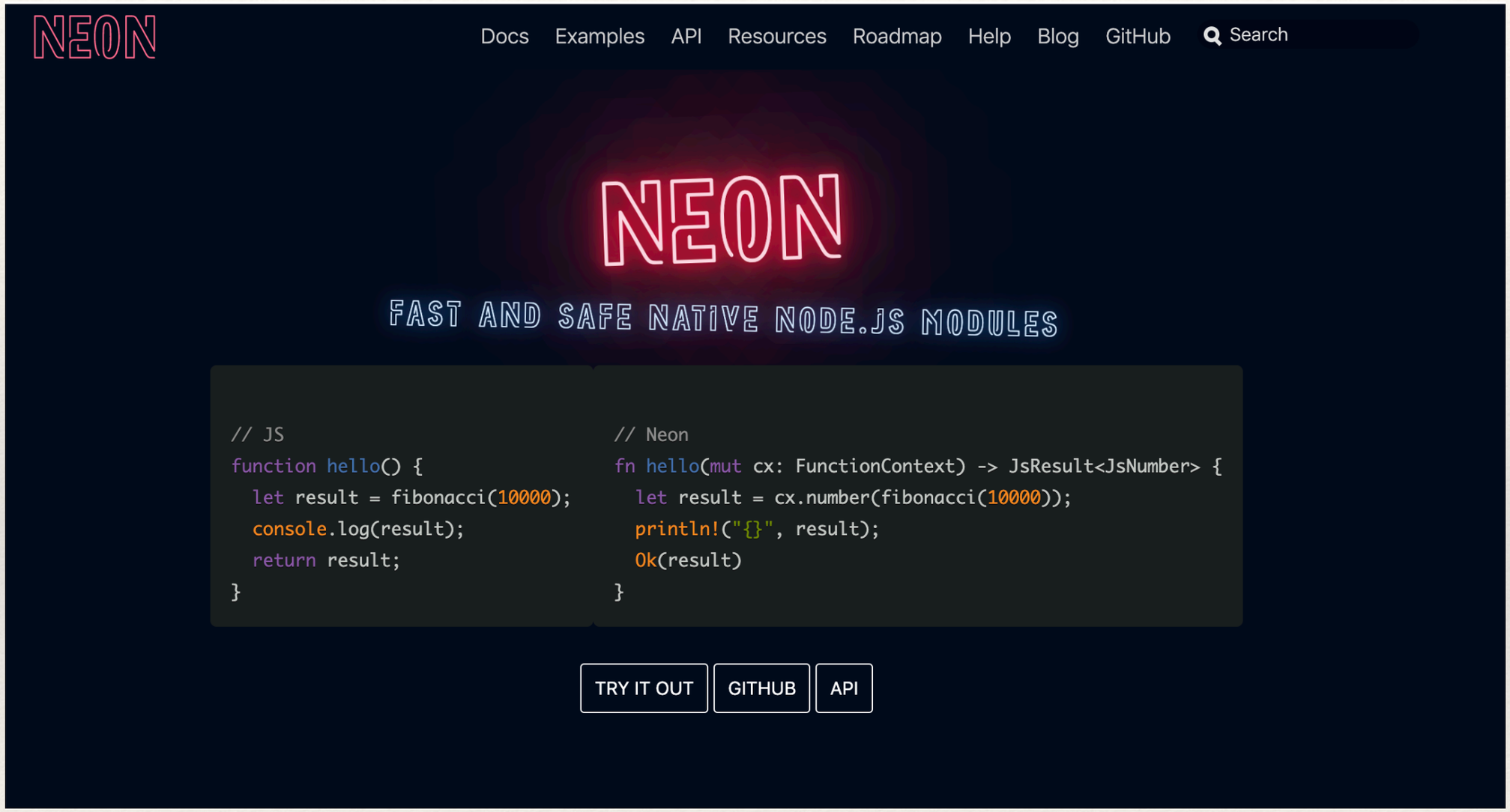
FFI: Foreign Function Interface

FFI: Foreign Function Interface

A mechanism by which a program written in one programming language can call routines or make use of services written in another.

Rust -> JavaScript

Rust -> JavaScript



Rust -> WebAssembly -> JavaScript

Ashley Williams

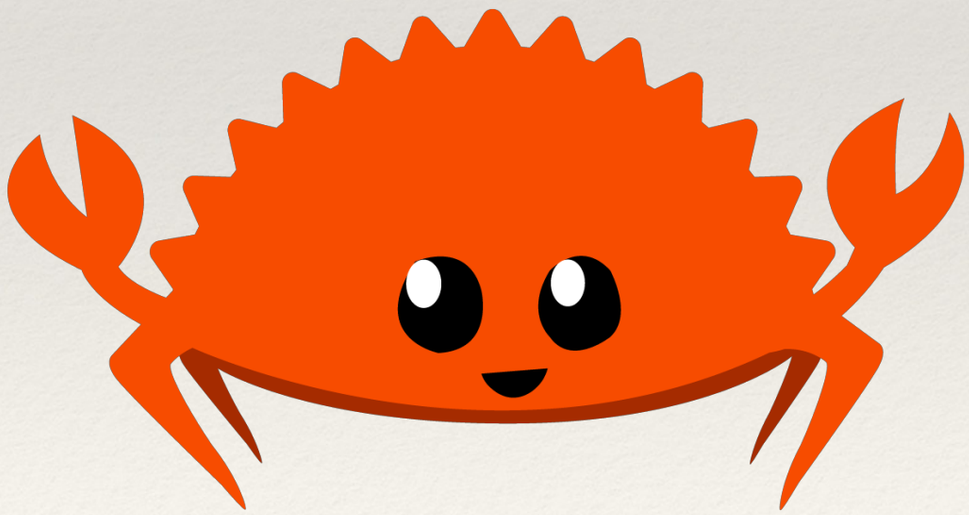
@ag_dubs

- Rust Core Team
- Rust Wasm Core Team
- wasm-pack author
- Former npm engineer
- Former Node.js Board
- Webassembler, Cloudflare

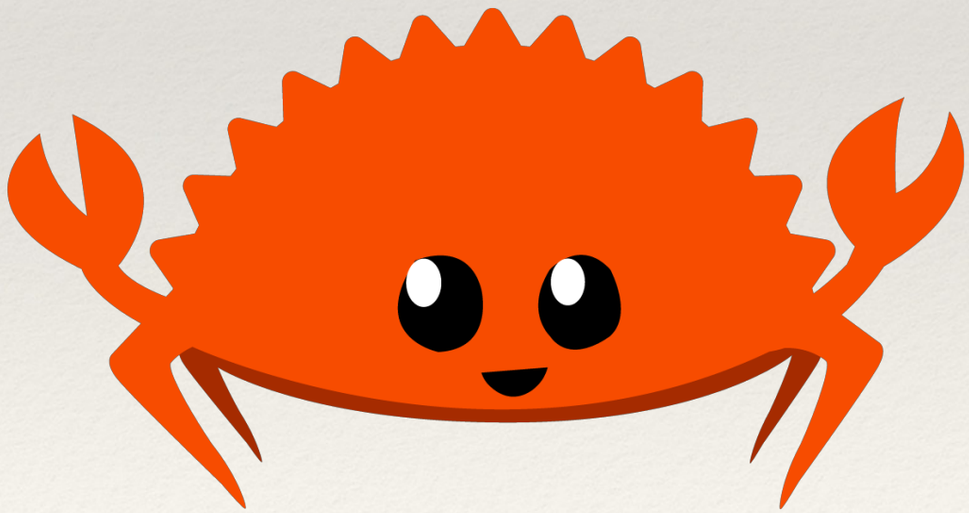
Rust -> WebAssembly -> JavaScript

What is Rust?

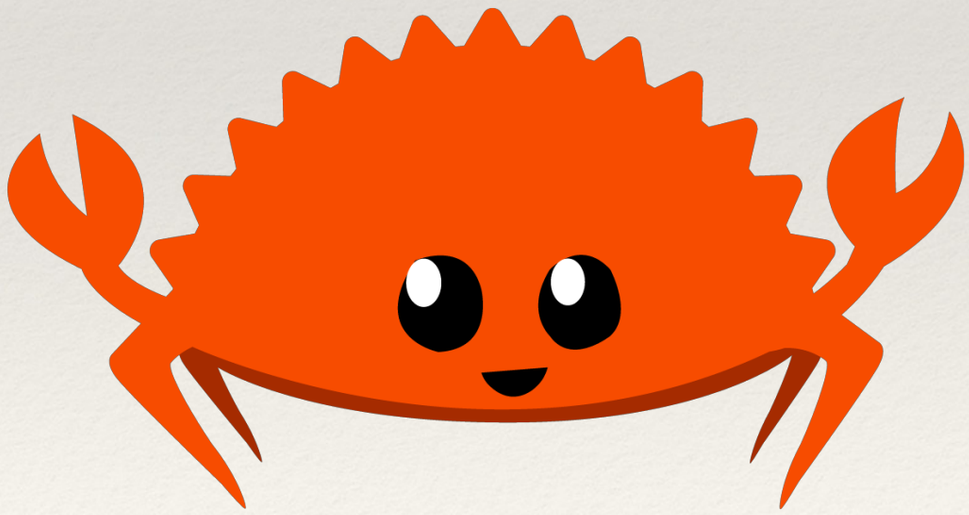
Rust is a programming language designed to empower everyone to build reliable and efficient software.



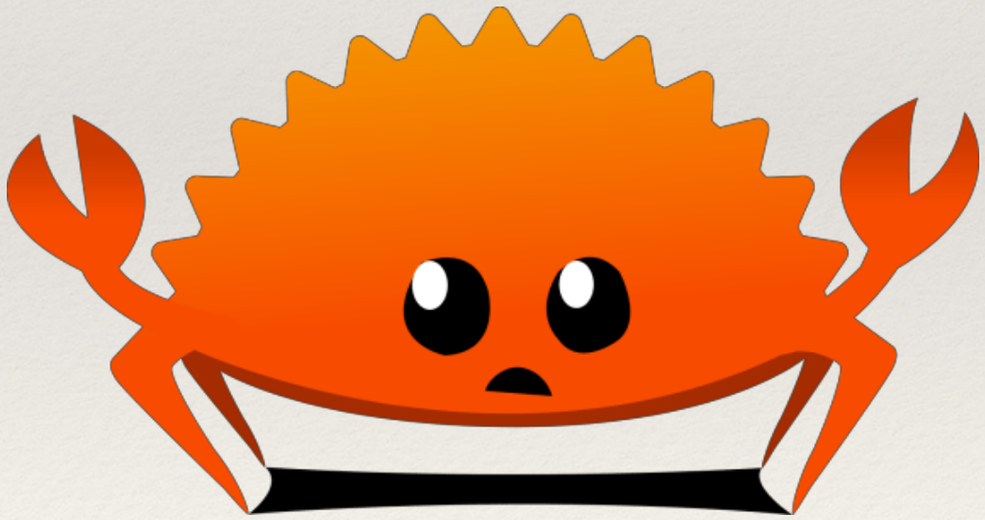
Statically Typed, with Type Inference



Guaranteed Memory Safety without a Garbage Collector



If Cloudbleed had been written in Rust it wouldn't have compiled.



What is WebAssembly?

A binary instruction format for a stack-based virtual machine.

A compilation target for running byte code on the web.

```
(module
  (func $addTwo (param i32 i32) (result i32)
    (i32.add
      (get_local 0)
      (get_local 1)))
  (export "addTwo" $addTwo))
```

```
0000000: 0061 736d                                 ; WASM_BINARY_MAGIC
0000004: 0b00 0000                                 ; WASM_BINARY_VERSION
; section "type"
0000008: 04                                        ; string length
0000009: 7479 7065                                 ; section id: "type"
000000d: 00                                        ; section size (guess)
000000e: 01                                        ; num types
; type 0
000000f: 40                                        ; function form
0000010: 02                                        ; num params
0000011: 01                                        ; param type
0000012: 01                                        ; param type
0000013: 01                                        ; num results
0000014: 01                                        ; result_type
000000d: 07                                        ; FIXUP section size
; section "function"
0000015: 08                                        ; string length
0000016: 6675 6e63 7469 6f6e                       ; section id: "function"
000001e: 00                                        ; section size (guess)
```

WHY??????

Bringing the Web up to Speed with WebAssembly

Andreas Haas Andreas Rossberg Derek L. Schuff* Ben L. Titzer
Google GmbH, Germany / *Google Inc, USA
{ahaas,rossberg,dschuff,titzer}@google.com

Michael Holman
Microsoft Inc, USA
michael.holman@microsoft.com

Dan Gohman Luke Wagner Alon Zakai
Mozilla Inc, USA
{sunfishcode,luke,azakai}@mozilla.com

JF Bastien
Apple Inc, USA
jfbastien@apple.com

Abstract

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.

Engineers from the four major browser vendors have risen to the challenge and collaboratively designed a portable

device types. By historical accident, JavaScript is the only natively supported programming language on the Web, its widespread usage unmatched by other technologies available only via plugins like ActiveX, Java or Flash. Because of JavaScript's ubiquity, rapid performance improvements in modern VMs, and perhaps through sheer necessity, it has become a compilation target for other languages. Through Emscripten [43], even C and C++ programs can be compiled to a stylized low-level subset of JavaScript called asm.js [4]. Yet JavaScript has inconsistent performance and a number of other pitfalls, especially as a compilation target.

AutoCAD
@AutoCAD

Follow

Hello from #GoogleIO! The AutoCAD product team is excited to be in Mountain View this week presenting their work with AutoCAD web app. Stay tuned for videos from the conference!

1:53 PM - 8 May 2018 from San Francisco, CA

11 Retweets **70** Likes

WebAssembly is a technology that invites new types of applications written in multiple different languages to be discovered and distributed on the web

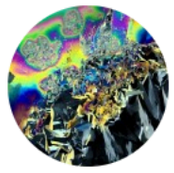

Google


Java

Speed and Size

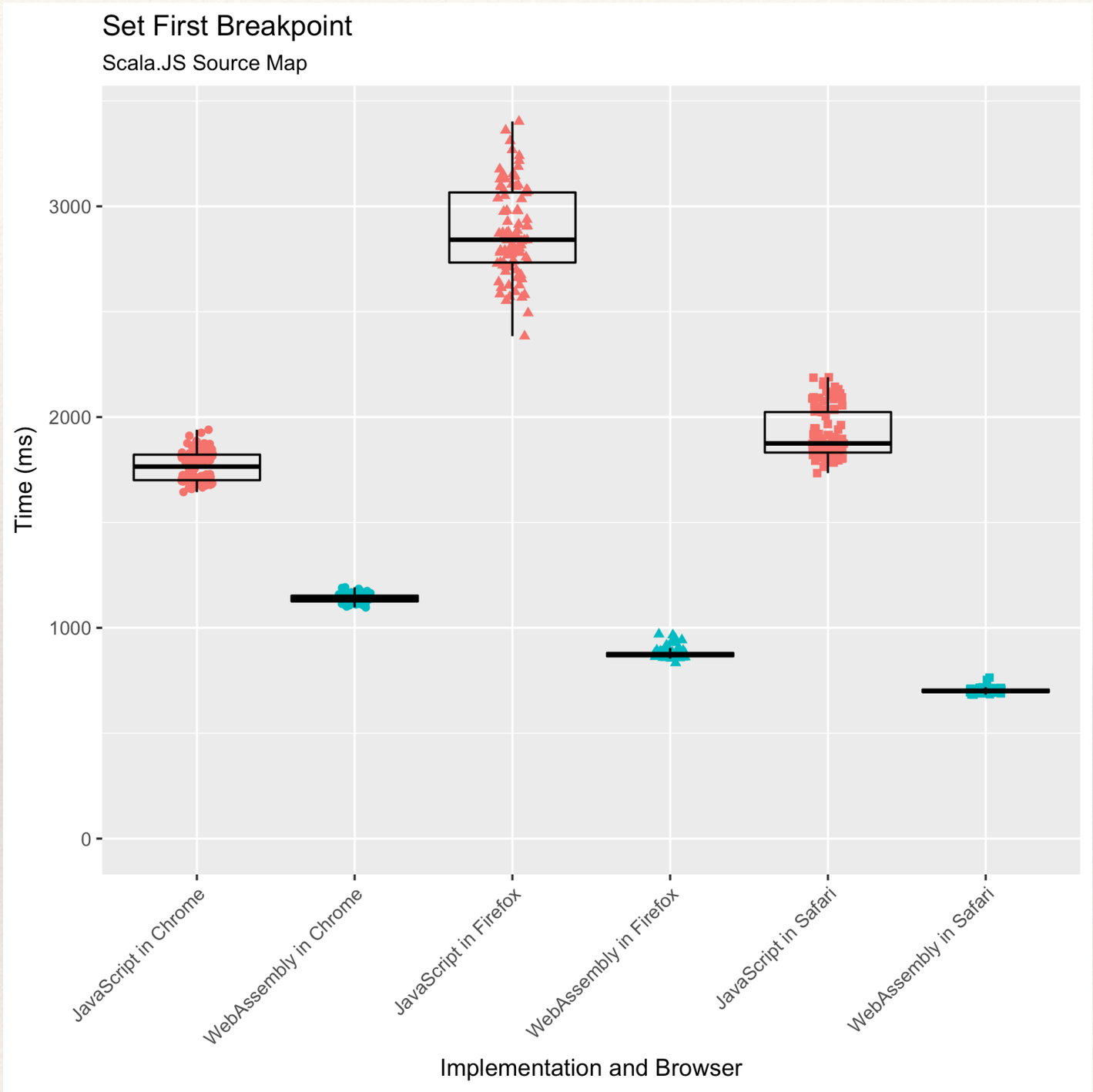
moz://a HACKS

Oxidizing Source Maps with Rust and WebAssembly



By Nick Fitzgerald

Posted on January 18, 2018 in Featured Article, Performance, Rust, and WebAssembly


Set First Breakpoint
Scala.JS Source Map
Time (ms)
3000
2000
1000
0
JavaScript in Chrome
WebAssembly in Chrome
JavaScript in Firefox
WebAssembly in Firefox
JavaScript in Safari
WebAssembly in Safari
Implementation and Browser

Maybe you don't need Rust and WASM to speed up your JS

Vyacheslav Egorov on 03 feb 2018

Few weeks ago I noticed a blog post "Oxidizing Source Maps with Rust and WebAssembly" making rounds on Twitter - talking about performance benefits of replacing plain JavaScript in the core of `source-map` library with a Rust version compiled to WebAssembly.

This post piqued my interest, not because I am a huge on either Rust or WASM, but rather because I am always curious about language features and optimizations missing in pure JavaScript to achieve similar performance characteristics.

So I checked out the library from GitHub and departed on a small performance investigation, which I am documenting here almost verbatim.

Nick Fitzgerald

Speed Without Wizardry

Feb 26, 2018

Vyacheslav Egorov, who goes by `mraleph` on the Web, wrote a response to my article "Oxidizing Source Maps with Rust and WebAssembly" titled "Maybe you don't need Rust and WASM to speed up your JS".

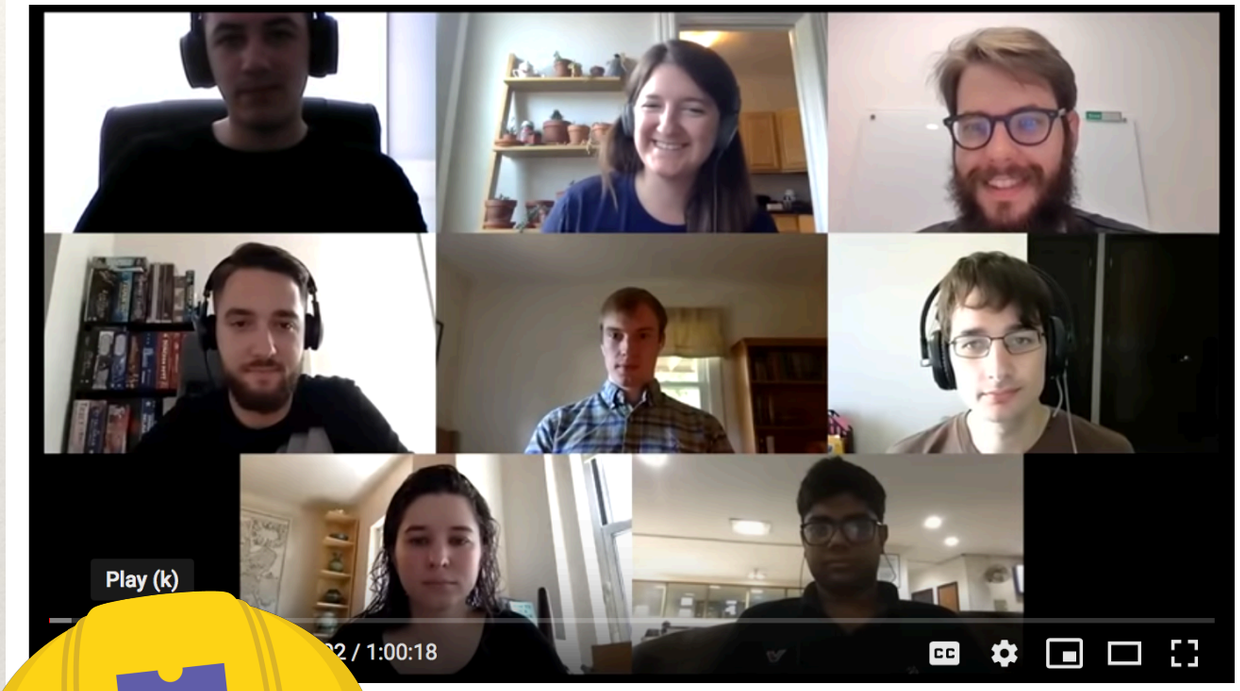
The "Oxidizing" article recounts my experience integrating Rust (compiled to WebAssembly) into the `source-map` JavaScript library. Although the JavaScript implementation was originally authored in idiomatic JavaScript style, as we profiled and implemented speed improvements, the code became hard to read and maintain. With Rust and

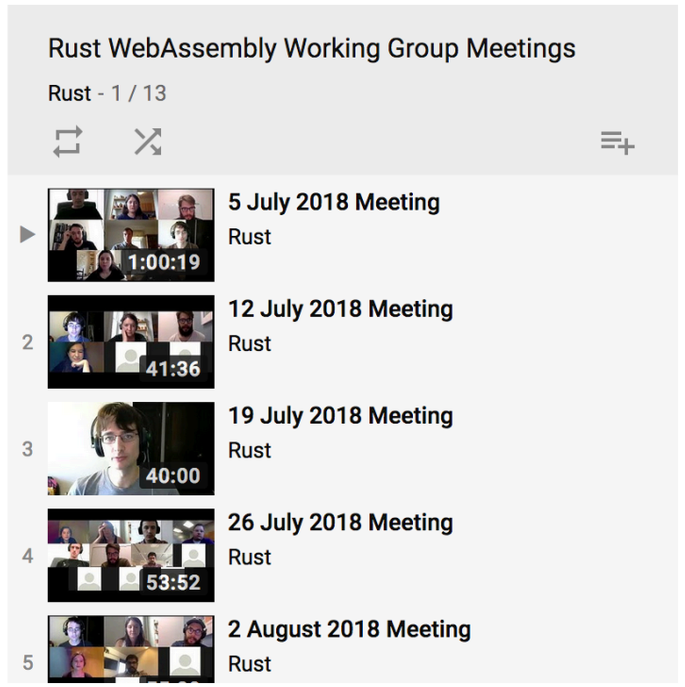
WebAssembly is not a replacement for JavaScript.

To be successful, WebAssembly needs to interoperate with JavaScript

- Be able to store and work with JavaScript Objects
- Integrate into the JS Ecosystem via modules and workflows, like npm

RustWasm Working Group





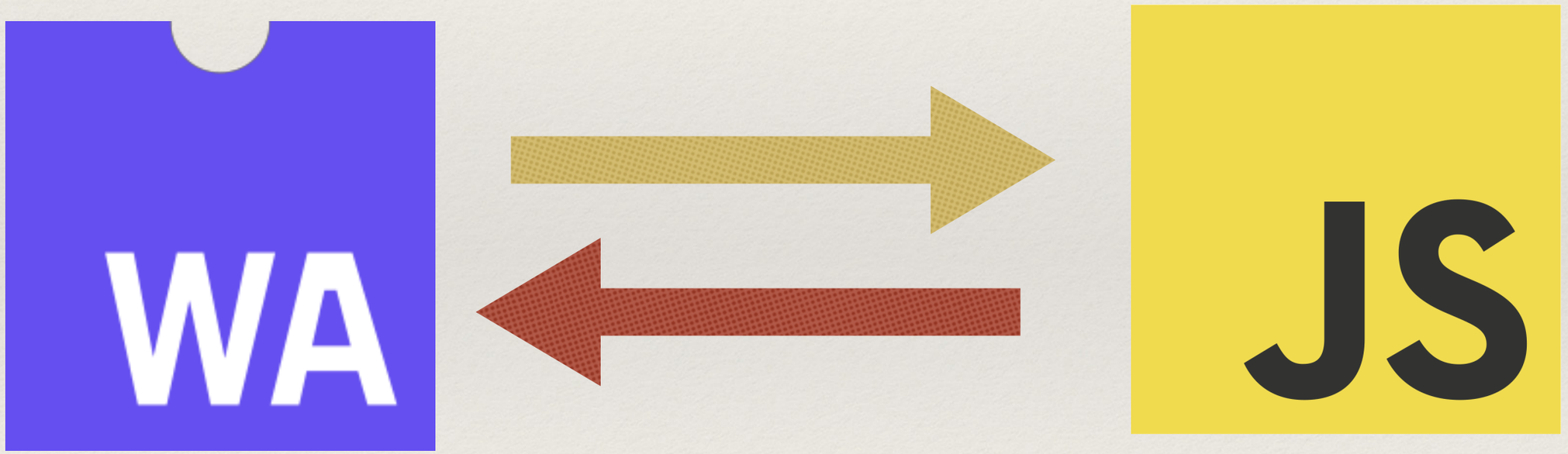
https://rustwasm.github.io

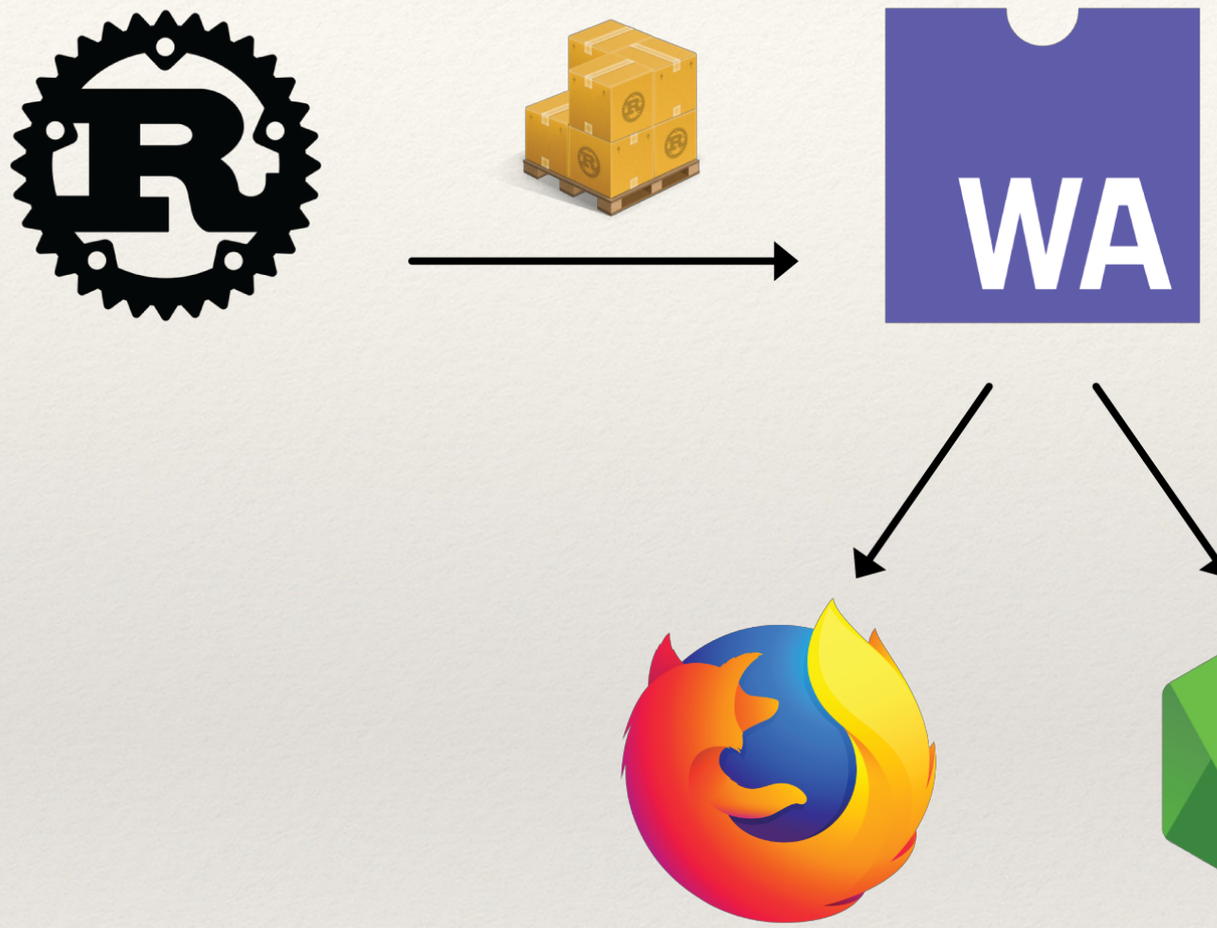
DEMO TIME!

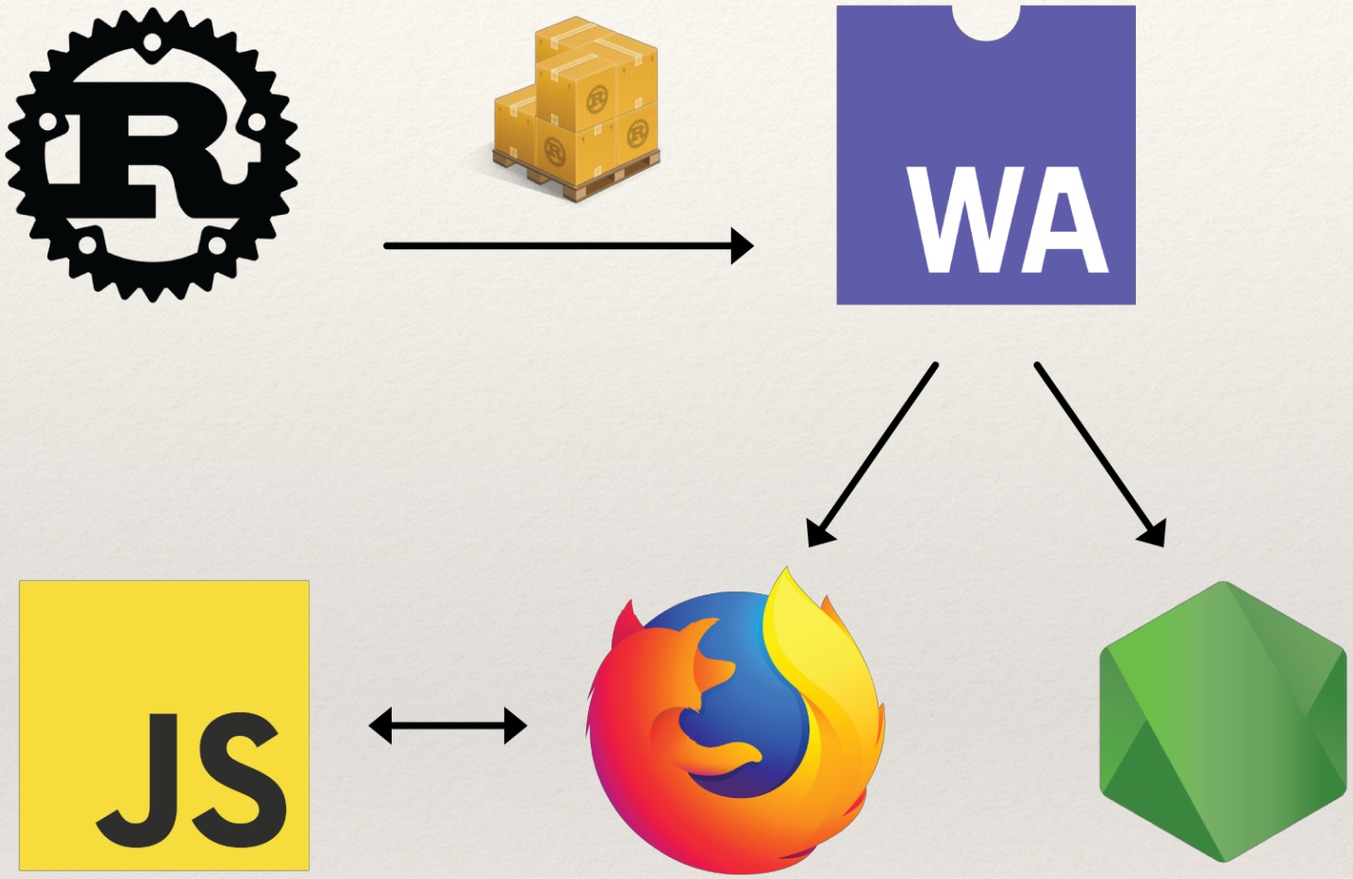
Design Principles

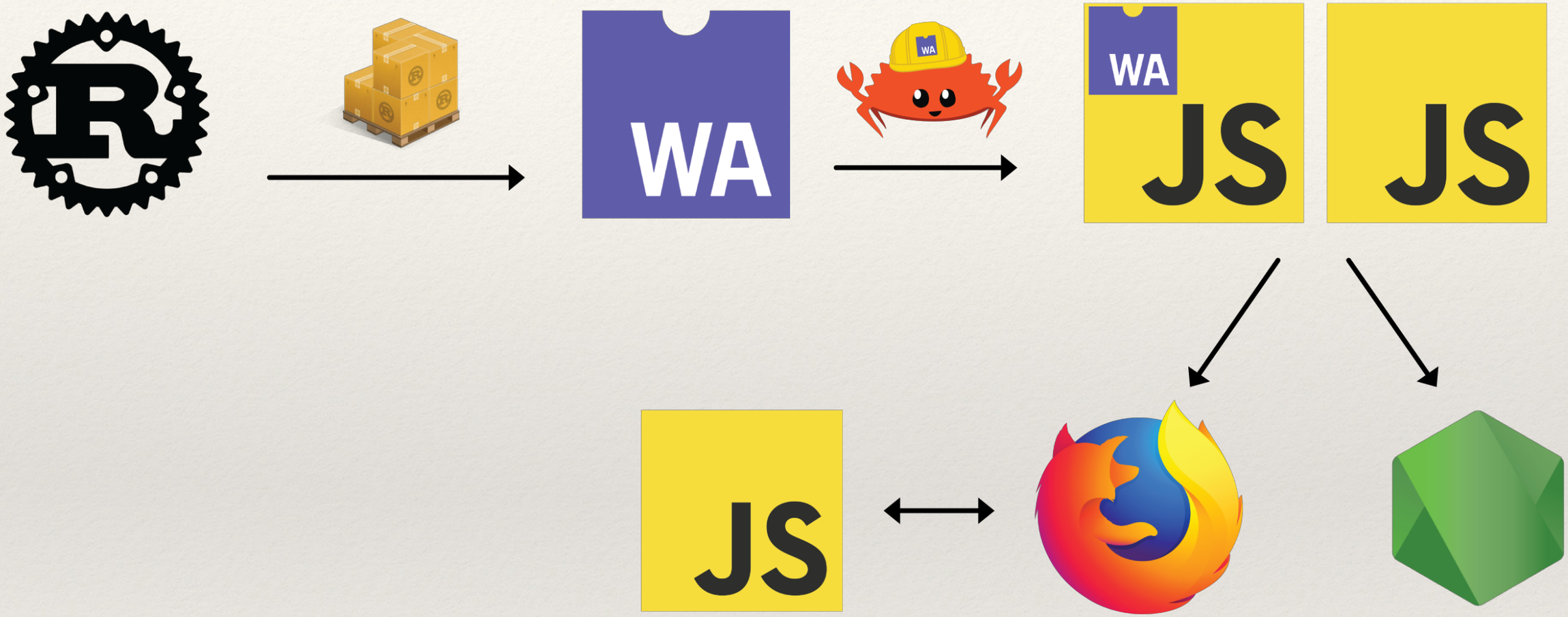
Rust -> WebAssembly -> JavaScript

Wasm should be an implementation detail



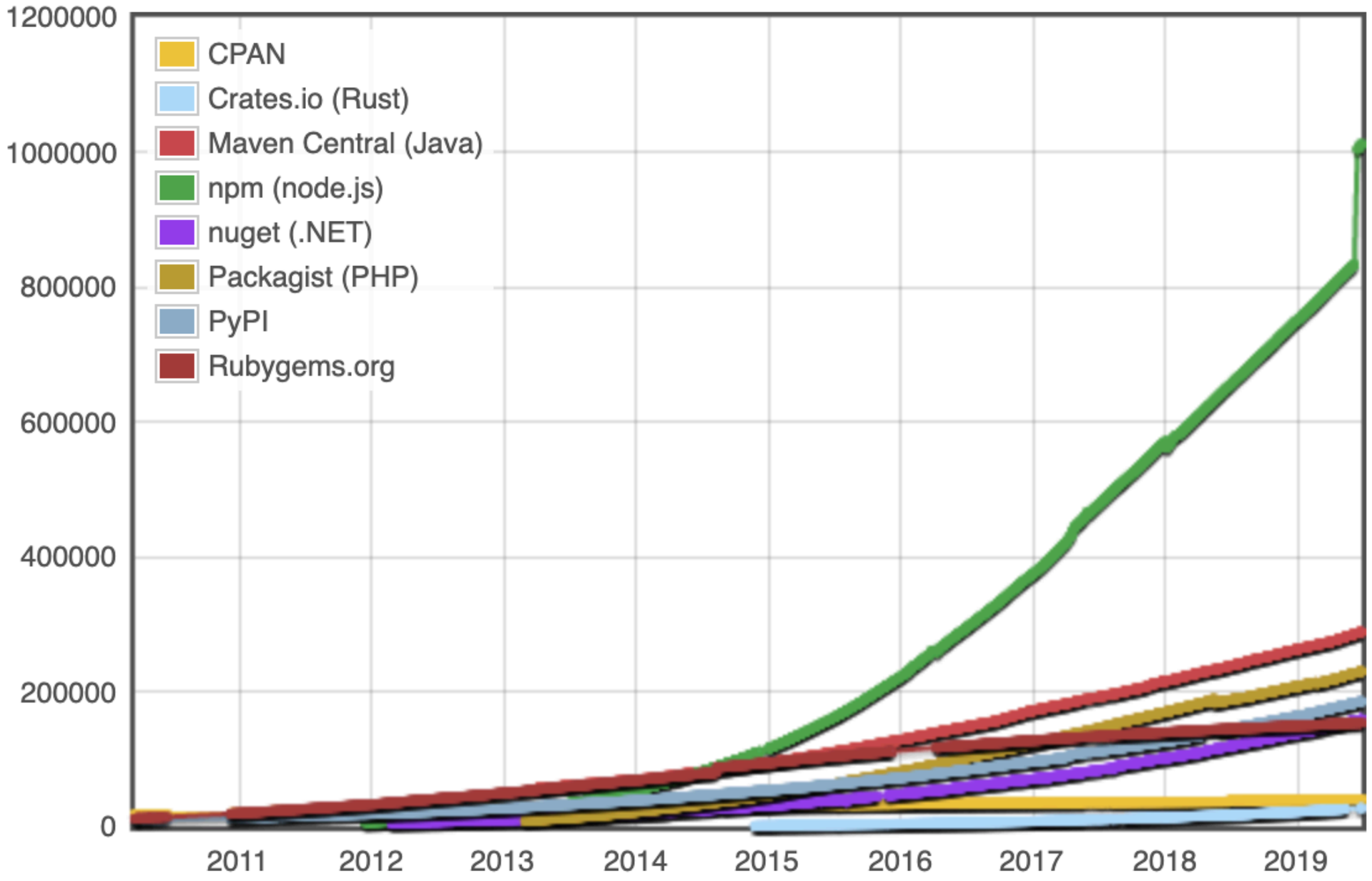

WA


WA
JS


WA
WA
JS
JS
JS

To be successful, WebAssembly needs to interoperate with JavaScript

Module Counts



There are around 9M* JavaScript developers in the world.

That's ~2.5M* more next most popular language

To be successful, WebAssembly needs to interoperate with JavaScript

Wasm support should be unobtrusive

```rust
pub fn greet(name: &str) -> String {
    // ...
}
```

```rust
#[wasm_bindgen]
pub fn greet(name: &str) -> String {
    // ...
}
```

We need to work with and pass JS Objects in Wasm

But current Wasm doesn’t support that

Current Wasm only supports integers and floats

We need to “enhance” the ABI of Wasm modules

“How do we shoehorn JS objects into a u32 for Wasm to use?”

Linear Memory

AKA “One Big Array”

A Polyfill for "JavaScript Objects in Wasm"

```rust
// foo.rs
#[wasm_bindgen]
pub fn foo(a: &JsValue) {
    // ...
}
```

```rust
// foo.rs
#[wasm_bindgen]
pub fn foo(a: JsValue) {
    // ...
}
```

Linear Memory

AKA “One Big Array”

```
const heap = new Array(32);
```

Short-lived JavaScript Objects

```rust
// foo.rs
#[wasm_bindgen]
pub fn foo(a: &JsValue) {
    // ...
}
```

JS Objects in Wasm

Short-lived JS Objects on the Stack

The Rust we write:

```rust
// foo.rs
#[wasm_bindgen]
pub fn foo(a: &JsValue) {
    // ...
}
```

The TS interface we want:

```ts
// foo.d.ts
export function foo(a: any);
```

```ts
// foo.js
import * as wasm from './foo_bg';

const heap = new Array(32);
heap.push(undefined, null, true, false);
let stack_pointer = 32;

function addBorrowedObject(obj) {
  stack_pointer -= 1;
  heap[stack_pointer] = obj;
  return stack_pointer;
}

export function foo(arg0) {
  const idx0 = addBorrowedObject(arg0);
  try {
    wasm.foo(idx0);
  } finally {
    heap[stack_pointer++] = undefined;
  }
```

```rust
// Generated Rust
// Original function is unmodified
pub fn foo(a: &JsValue) {
    // ...
}

// Wrapper function, unique name, actually exported
// from the Wasm
#[export_name = "foo"]
pub extern "C" fn
__wasm_bindgen_generated_foo(arg0: u32) {
    let arg0 = unsafe {

ManuallyDrop::new(JsValue::__from_idx(arg0))
    };
    let arg0 = &*arg0;
    foo(arg0);
}
```

JS Objects in Wasm

Short-lived JS Objects on the Stack

The Rust we write:

```rust
// foo.rs
#[wasm_bindgen]
pub fn foo(a: &JsValue) {
    // ...
}
```

The TS interface we want:

```ts
// foo.d.ts
export function foo(a: any);
```

```ts
// foo.js
import * as wasm from './foo_bg';

const heap = new Array(32);
heap.push(undefined, null, true, false);
let stack_pointer = 32;

function addBorrowedObject(obj) {
  stack_pointer -= 1;
  heap[stack_pointer] = obj;
  return stack_pointer;
}

export function foo(arg0) {
  const idx0 = addBorrowedObject(arg0);
  try {
    wasm.foo(idx0);
  } finally {
    heap[stack_pointer++] = undefined;
  }
```

```rust
// Generated Rust
// Original function is unmodified
pub fn foo(a: &JsValue) {
    // ...
}

// Wrapper function, unique name, actually exported
// from the Wasm
#[export_name = "foo"]
pub extern "C" fn
__wasm_bindgen_generated_foo(arg0: u32) {
    let arg0 = unsafe {

ManuallyDrop::new(JsValue::__from_idx(arg0))
    };
    let arg0 = &*arg0;
    foo(arg0);
}
```

JS Objects in Wasm

Short-lived JS Objects on the Stack

The Rust we write:

```rust
// foo.rs
#[wasm_bindgen]
pub fn foo(a: &JsValue) {
    // ...
}
```

The TS interface we want:

```ts
// foo.d.ts
export function foo(a: any);
```

```ts
// foo.js
import * as wasm from './foo_bg';

const heap = new Array(32);
heap.push(undefined, null, true, false);
let stack_pointer = 32;

function addBorrowedObject(obj) {
  stack_pointer -= 1;
  heap[stack_pointer] = obj;
  return stack_pointer;
}

export function foo(arg0) {
  const idx0 = addBorrowedObject(arg0);
  try {
    wasm.foo(idx0);
  } finally {
    heap[stack_pointer++] = undefined;
  }
```

```rust
// Generated Rust
// Original function is unmodified
pub fn foo(a: &JsValue) {
    // ...
}

// Wrapper function, unique name, actually exported
// from the Wasm
#[export_name = "foo"]
pub extern "C" fn
__wasm_bindgen_generated_foo(arg0: u32) {
    let arg0 = unsafe {

ManuallyDrop::new(JsValue::__from_idx(arg0))
    };
    let arg0 = &*arg0;
    foo(arg0);
}
```

JS Objects in Wasm

Short-lived JS Objects on the Stack

The Rust we write:

```rust
// foo.rs
#[wasm_bindgen]
pub fn foo(a: &JsValue) {
    // ...
}
```

The TS interface we want:

```ts
// foo.d.ts
export function foo(a: any);
```

```ts
// foo.js
import * as wasm from './foo_bg';

const heap = new Array(32);
heap.push(undefined, null, true, false);
let stack_pointer = 32;

function addBorrowedObject(obj) {
  stack_pointer -= 1;
  heap[stack_pointer] = obj;
  return stack_pointer;
}

export function foo(arg0) {
  const idx0 = addBorrowedObject(arg0);
  try {
    wasm.foo(idx0);
  } finally {
    heap[stack_pointer++] = undefined;
  }
```

```rust
// Generated Rust
// Original function is unmodified
pub fn foo(a: &JsValue) {
    // ...
}

// Wrapper function, unique name, actually exported
// from the Wasm
#[export_name = "foo"]
pub extern "C" fn
__wasm_bindgen_generated_foo(arg0: u32) {
    let arg0 = unsafe {

ManuallyDrop::new(JsValue::__from_idx(arg0))
    };
    let arg0 = &*arg0;
    foo(arg0);
}
```

JS Objects in Wasm

Short-lived JS Objects on the Stack

The Rust we write:

```rust
// foo.rs
#[wasm_bindgen]
pub fn foo(a: &JsValue) {
    // ...
}
```

The TS interface we want:

```ts
// foo.d.ts
export function foo(a: any);
```

```ts
// foo.js
import * as wasm from './foo_bg';

const heap = new Array(32);
heap.push(undefined, null, true, false);
let stack_pointer = 32;

function addBorrowedObject(obj) {
  stack_pointer -= 1;
  heap[stack_pointer] = obj;
  return stack_pointer;
}

export function foo(arg0) {
  const idx0 = addBorrowedObject(arg0);
  try {
    wasm.foo(idx0);
  } finally {
    heap[stack_pointer++] = undefined;
  }
```

```rust
// Generated Rust
// Original function is unmodified
pub fn foo(a: &JsValue) {
    // ...
}

// Wrapper function, unique name, actually exported
// from the Wasm
#[export_name = "foo"]
pub extern "C" fn
__wasm_bindgen_generated_foo(arg0: u32) {
    let arg0 = unsafe {

ManuallyDrop::new(JsValue::__from_idx(arg0))
    };
    let arg0 = &*arg0;
    foo(arg0);
}
```

Long-lived JavaScript Objects

```rust
// foo.rs
#[wasm_bindgen]
pub fn foo(a: JsValue) {
    // ...
}
```

JS Objects in Wasm

Long-lived JS Objects on the Stack

The Rust we write:

```rust
// foo.rs
#[wasm_bindgen]
pub fn foo(a: JsValue) {
    // ...
}
```

The TS interface we want:

```ts
// foo.d.ts
export function foo(a: any);
```

```ts
import * as wasm from './foo_bg'; // imports from wasm file

const heap = new Array(32);
heap.push(undefined, null, true, false);
let heap_next = 36;

function addHeapObject(obj) {
  if (heap_next === heap.length)
    heap.push(heap.length + 1);
  const idx = heap_next;
  heap_next = heap[idx];
  heap[idx] = obj;
  return idx;
}

export function foo(arg0) {
  const idx0 = addHeapObject(arg0);
  wasm.foo(idx0);
}

export function __wbindgen_object_drop_ref(idx) {
  heap[idx ] = heap_next;
  heap_next = idx;
}
```

```rust
// what the user wrote
pub fn foo(a: JsValue) {
    // ...
}

#[export_name = "foo"]
pub extern "C" fn
__wasm_bindgen_generated_foo(arg0: u32) {
    let arg0 = unsafe {
        JsValue::__from_idx(arg0)
    };
    foo(arg0);
}
```

JS Objects in Wasm

Long-lived JS Objects on the Stack

The Rust we write:

```rust
// foo.rs
#[wasm_bindgen]
pub fn foo(a: JsValue) {
    // ...
}
```

The TS interface we want:

```ts
// foo.d.ts
export function foo(a: any);
```

```ts
import * as wasm from './foo_bg'; // imports from
wasm file

const heap = new Array(32);
heap.push(undefined, null, true, false);
let heap_next = 36;

function addHeapObject(obj) {
  if (heap_next === heap.length)
    heap.push(heap.length + 1);
  const idx = heap_next;
  heap_next = heap[idx];
  heap[idx] = obj;
  return idx;
}

export function foo(arg0) {
  const idx0 = addHeapObject(arg0);
  wasm.foo(idx0);
}

export function __wbindgen_object_drop_ref(idx) {
  heap[idx ] = heap_next;
  heap_next = idx;
}
```

```rust
// what the user wrote
pub fn foo(a: JsValue) {
    // ...
}

#[export_name = "foo"]
pub extern "C" fn
__wasm_bindgen_generated_foo(arg0: u32) {
    let arg0 = unsafe {
        JsValue::__from_idx(arg0)
    };
    foo(arg0);
}
```

JS Objects in Wasm

Long-lived JS Objects on the Stack

The Rust we write:

```rust
// foo.rs
#[wasm_bindgen]
pub fn foo(a: JsValue) {
    // ...
}
```

The TS interface we want:

```ts
// foo.d.ts
export function foo(a: any);
```

```ts
import * as wasm from './foo_bg'; // imports from wasm file

const heap = new Array(32);
heap.push(undefined, null, true, false);
let heap_next = 36;

function addHeapObject(obj) {
  if (heap_next === heap.length)
    heap.push(heap.length + 1);
  const idx = heap_next;
  heap_next = heap[idx];
  heap[idx] = obj;
  return idx;
}

export function foo(arg0) {
  const idx0 = addHeapObject(arg0);
  wasm.foo(idx0);
}

export function __wbindgen_object_drop_ref(idx) {
  heap[idx ] = heap_next;
  heap_next = idx;
}
```

```rust
// what the user wrote
pub fn foo(a: JsValue) {
    // ...
}

#[export_name = "foo"]
pub extern "C" fn
__wasm_bindgen_generated_foo(arg0: u32) {
    let arg0 = unsafe {
        JsValue::__from_idx(arg0)
    };
    foo(arg0);
}
```

JS Objects in Wasm

Long-lived JS Objects on the Stack

The Rust we write:

```rust
// foo.rs
#[wasm_bindgen]
pub fn foo(a: JsValue) {
    // ...
}
```

The TS interface we want:

```ts
// foo.d.ts
export function foo(a: any);
```

```ts
import * as wasm from './foo_bg'; // imports from wasm file

const heap = new Array(32);
heap.push(undefined, null, true, false);
let heap_next = 36;

function addHeapObject(obj) {
  if (heap_next === heap.length)
    heap.push(heap.length + 1);
  const idx = heap_next;
  heap_next = heap[idx];
  heap[idx] = obj;
  return idx;
}

export function foo(arg0) {
  const idx0 = addHeapObject(arg0);
  wasm.foo(idx0);
}

export function __wbindgen_object_drop_ref(idx) {
  heap[idx ] = heap_next;
  heap_next = idx;
}
```

```rust
// what the user wrote
pub fn foo(a: JsValue) {
    // ...
}

#[export_name = "foo"]
pub extern "C" fn
__wasm_bindgen_generated_foo(arg0: u32) {
    let arg0 = unsafe {
        JsValue::__from_idx(arg0)
    };
    foo(arg0);
}
```

Exporting to JavaScript

```rust
#[wasm_bindgen]
pub fn greet(a: &str) -> String {
    format!("Hello, {}!", a)
}
```

I want a Wasm function that works like this…

```
export function greet(name) {
    alert(`Hello, ${name}!`);
}
```

…so I write Rust like this…

```
#[wasm_bindgen]
extern {
    fn alert(s: &str);
}

#[wasm_bindgen]
pub fn greet(name: &str) {
    alert(&format!("Hello, {}!", name));
}
```

…so I can use it in JS like this!

```
const rust = import("./wasm_greet");
rust.then(m => m.greet("World!"));
```

I write this…

```rust
#[wasm_bindgen]
extern {
    fn alert(s: &str);
}

#[wasm_bindgen]
pub fn greet(name: &str) {
    alert(&format!("Hello, {}!", name));
}
```

```rust
pub fn greet(name: &str) {
    alert(&format!("Hello, {}!", name));
}

#[export_name = "greet"]
pub extern fn __wasm_bindgen_generated_greet(arg0_ptr: *mut u8, arg0_len: usize) {
    let arg0 = unsafe { ::std::slice::from_raw_parts(arg0_ptr as *const u8, arg0_len) };
    let arg0 = unsafe { ::std::str::from_utf8_unchecked(arg0) };
    greet(arg0);
}
```

```rust
fn alert(s: &str) {
    #[wasm_import_module = "__wbindgen_placeholder__"]
    extern {
        fn __wbg_f_alert_alert_n(s_ptr: *const u8, s_len: usize);
    }
    unsafe {
        let s_ptr = s.as_ptr();
        let s_len = s.len();
        __wbg_f_alert_alert_n(s_ptr, s_len);
    }
}
```

…and it becomes this!

I write this…

```
const rust = import("./wasm_greet");
rust.then(m => m.greet("World!"));
```

```
import * as wasm from './wasm_greet_bg';

// ...

export function greet(arg0) {
    const [ptr0, len0] = passStringToWasm(arg0);
    try {
        const ret = wasm.greet(ptr0, len0);
        return ret;
    } finally {
        wasm.__wbindgen_free(ptr0, len0);
    }
}

export function __wbg_f_alert_alert_n(ptr0, len0)
{
    // ...
}
```

…and it calls this!

Exporting to JavaScript

Exporting a function to JavaScript

The Rust we write:

```rust
// foo.rs
#[wasm_bindgen]
pub fn greet(a: &str) -> String {
    format!("Hello, {}!", a)
}
```

The TS interface we want:

```typescript
// foo.d.ts
export function greet(a: string): string;
```

```javascript
import * as wasm from './foo_bg';

function passStringToWasm(arg) {
  const buf = new TextEncoder('utf-8').encode(arg);
  const len = buf.length;
  const ptr = wasm.__wbindgen_malloc(len);
  let array = new Uint8Array(wasm.memory.buffer);
  array.set(buf, ptr);
  return [ptr, len];
}

function getStringFromWasm(ptr, len) {
  const mem = new Uint8Array(wasm.memory.buffer);
  const slice = mem.slice(ptr, ptr + len);
  const ret = new
TextDecoder('utf-8').decode(slice);
  return ret;
}

export function greet(arg0) {
  const [ptr0, len0] = passStringToWasm(arg0);
  try {
    const ret = wasm.greet(ptr0, len0);
    const ptr = wasm.__wbindgen_boxed_str_ptr(ret);
    const len = wasm.__wbindgen_boxed_str_len(ret);
    const realRet = getStringFromWasm(ptr, len);
    wasm.__wbindgen_boxed_str_free(ret);
    return realRet;
  } finally {
    wasm.__wbindgen_free(ptr0, len0);
  }
}
```

```rust
pub extern "C" fn greet(a: &str) -> String {
    format!("Hello, {}!", a)
}

#[export_name = "greet"]
pub extern "C" fn __wasm_bindgen_generated_greet(
    arg0_ptr: *const u8,
    arg0_len: usize,
) -> *mut String {
    let arg0 = unsafe {
        let slice
= ::std::slice::from_raw_parts(arg0_ptr, arg0_len);
        ::std::str::from_utf8_unchecked(slice)
    };
    let _ret = greet(arg0);
    Box::into_raw(Box::new(_ret))
}
```

The Rust we write:

```rust
// foo.rs
#[wasm_bindgen]
pub fn greet(a: &str) -> String {
    format!("Hello, {}!", a)
}
```

The TS interface we want:

```typescript
// foo.d.ts
export function greet(a: string): string;
```

```javascript
import * as wasm from './foo_bg';

function passStringToWasm(arg) {
  const buf = new TextEncoder('utf-8').encode(arg);
  const len = buf.length;
  const ptr = wasm.__wbindgen_malloc(len);
  let array = new Uint8Array(wasm.memory.buffer);
  array.set(buf, ptr);
  return [ptr, len];
}

function getStringFromWasm(ptr, len) {
  const mem = new Uint8Array(wasm.memory.buffer);
  const slice = mem.slice(ptr, ptr + len);
  const ret = new
TextDecoder('utf-8').decode(slice);
  return ret;
}

export function greet(arg0) {
  const [ptr0, len0] = passStringToWasm(arg0);
  try {
    const ret = wasm.greet(ptr0, len0);
    const ptr = wasm.__wbindgen_boxed_str_ptr(ret);
    const len = wasm.__wbindgen_boxed_str_len(ret);
    const realRet = getStringFromWasm(ptr, len);
    wasm.__wbindgen_boxed_str_free(ret);
    return realRet;
  } finally {
    wasm.__wbindgen_free(ptr0, len0);
  }
}
```

```rust
pub extern "C" fn greet(a: &str) -> String {
    format!("Hello, {}!", a)
}

#[export_name = "greet"]
pub extern "C" fn __wasm_bindgen_generated_greet(
    arg0_ptr: *const u8,
    arg0_len: usize,
) -> *mut String {
    let arg0 = unsafe {
        let slice
= ::std::slice::from_raw_parts(arg0_ptr, arg0_len);
        ::std::str::from_utf8_unchecked(slice)
    };
    let _ret = greet(arg0);
    Box::into_raw(Box::new(_ret))
}
```

The Rust we write:

```rust
// foo.rs
#[wasm_bindgen]
pub fn greet(a: &str) -> String {
    format!("Hello, {}!", a)
}
```

The TS interface we want:

```typescript
// foo.d.ts
export function greet(a: string): string;
```

```javascript
import * as wasm from './foo_bg';

function passStringToWasm(arg) {
  const buf = new TextEncoder('utf-8').encode(arg);
  const len = buf.length;
  const ptr = wasm.__wbindgen_malloc(len);
  let array = new Uint8Array(wasm.memory.buffer);
  array.set(buf, ptr);
  return [ptr, len];
}

function getStringFromWasm(ptr, len) {
  const mem = new Uint8Array(wasm.memory.buffer);
  const slice = mem.slice(ptr, ptr + len);
  const ret = new
TextDecoder('utf-8').decode(slice);
  return ret;
}

export function greet(arg0) {
  const [ptr0, len0] = passStringToWasm(arg0);
  try {
    const ret = wasm.greet(ptr0, len0);
    const ptr = wasm.__wbindgen_boxed_str_ptr(ret);
    const len = wasm.__wbindgen_boxed_str_len(ret);
    const realRet = getStringFromWasm(ptr, len);
    wasm.__wbindgen_boxed_str_free(ret);
    return realRet;
  } finally {
    wasm.__wbindgen_free(ptr0, len0);
  }
}
```

The best part about all of this is that it's designed so that you Never Have to Think About it.


Gifon007.eu

WHY???

WebAssembly isn’t done.

We’re just getting started!

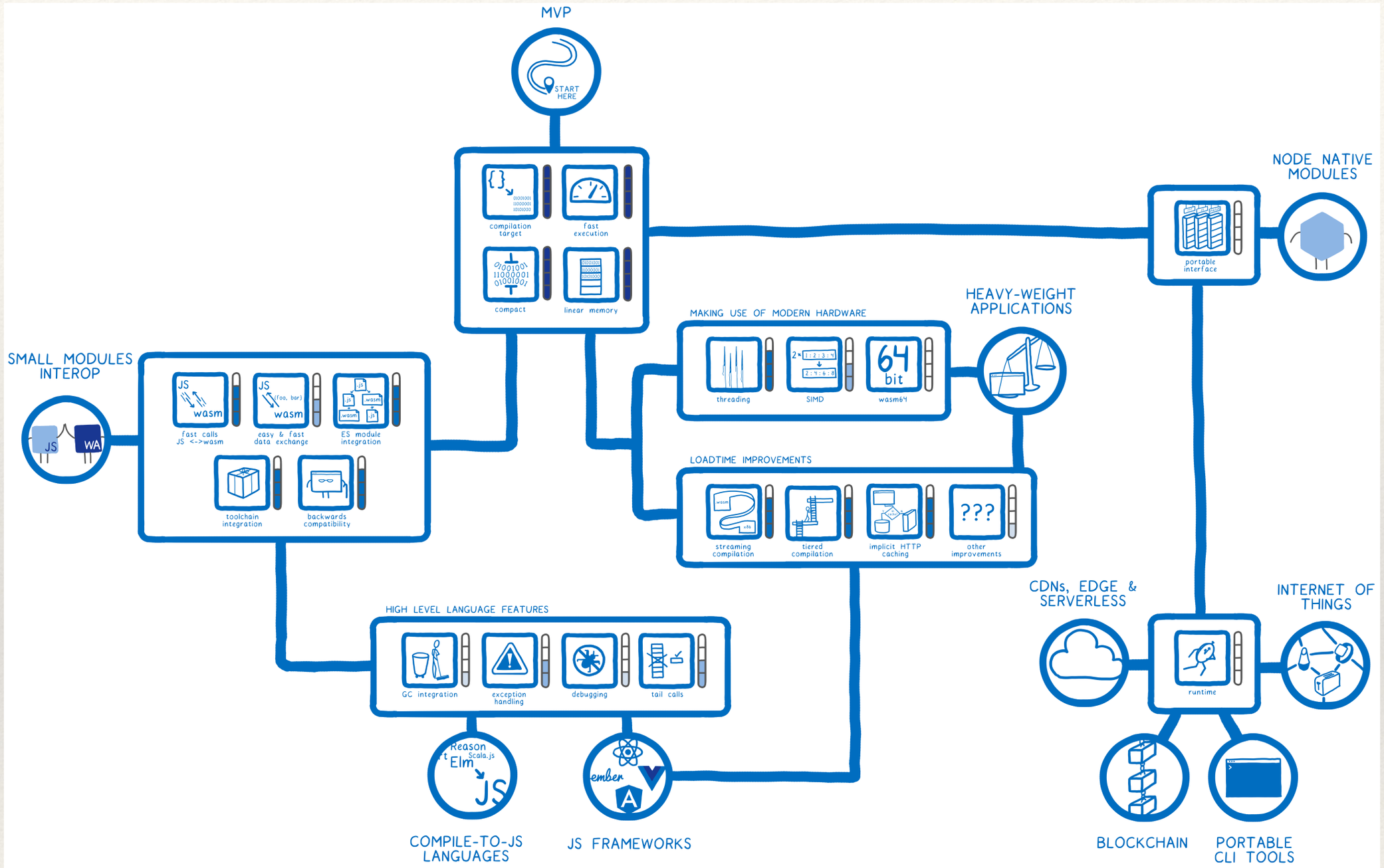
WebAssembly's post-MVP future: A cartoon skill tree

By Lin Clark, Till Schneidereit, Luke Wagner

Posted on October 22, 2018 in Code Cartoons, Featured Article, and WebAssembly

People have a misconception about WebAssembly. They think that the WebAssembly that landed in browsers back in 2017—which we called the minimum viable product (or MVP) of WebAssembly—is the final version of WebAssembly.

I can understand where that misconception comes from. The WebAssembly community group is really committed to backwards compatibility. This means that the WebAssembly that you create today **will** continue working on browsers into the future.

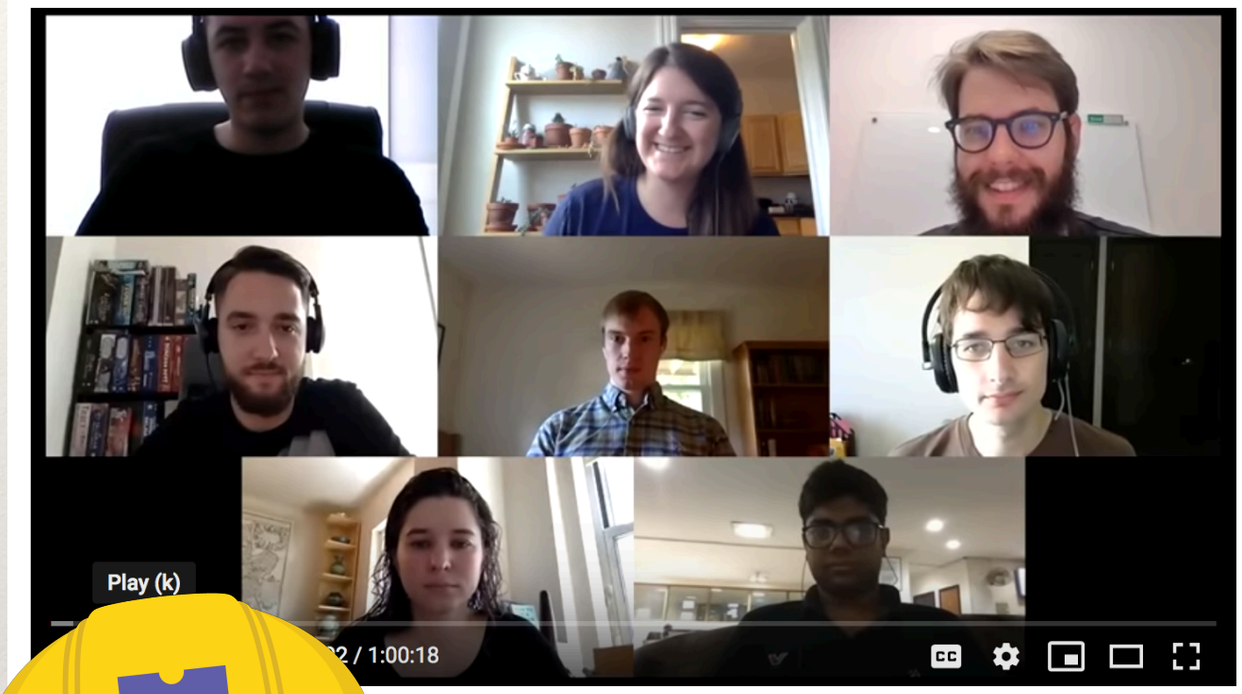


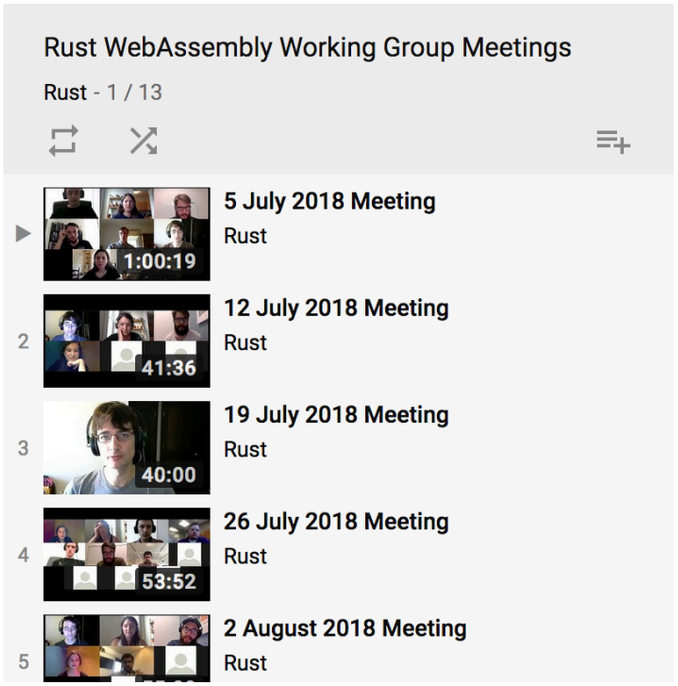
credit: Lin Clark, WebAssembly's post-MVP future: A cartoon skill tree

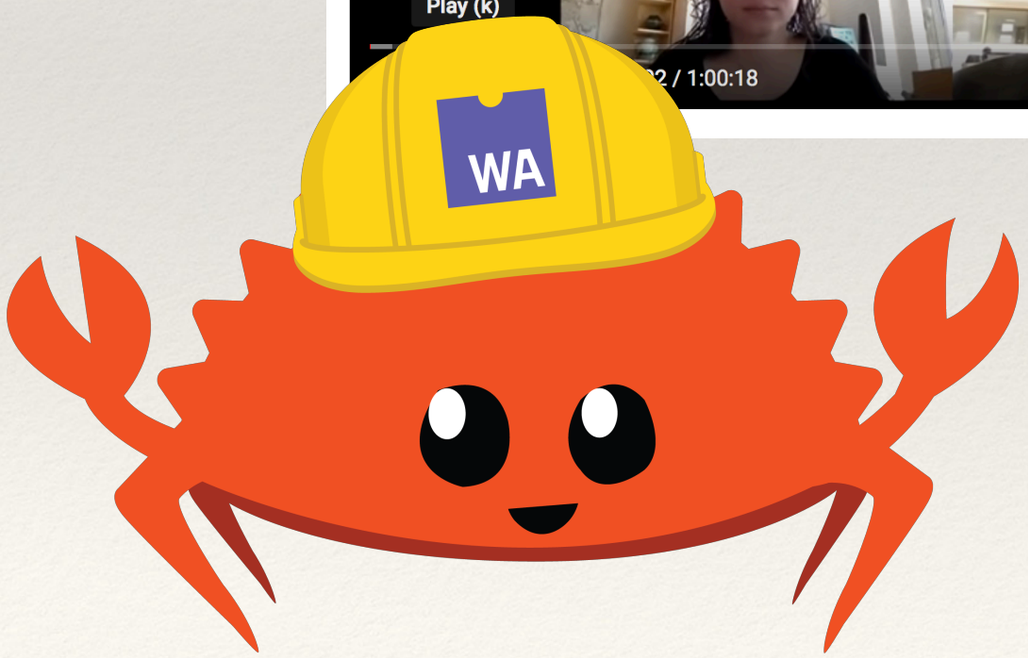
We want to build a WebAssembly for everyone.

So we need people to start using it now- so they can get involved in the design process!

RustWasm Working Group







https://rustwasm.github.io

Submitting Feedback & Issues

We welcome community and developer feedback on all aspects of WebAssembly, including the high-level design, binary format, JS API, developer experience, and browser implementations.

Please contribute your feedback or issues in the following forums:

- High level design feedback: WebAssembly/design
- Specification bugs / suggestions: WebAssembly/spec
- Test suite / reference interpreter issues: WebAssembly/spec
- Emscripten / Binaryen / LLVM issues: WebAssembly/binaryen
- WABT issues: WebAssembly/wabt
- V8 / Chrome bugs: crbug.com/v8
- SpiderMonkey / Firefox bugs: bugzilla.mozilla.org

WEBASSEMBLY COMMUNITY GROUP

The mission of this group is to promote early-stage cross-browser collaboration on a new, portable, size- and load-time-efficient format suitable for compilation to the web.

Note: Community Groups are proposed and run by the community. Although W3C hosts these conversations, the groups do not necessarily represent the views of the W3C Membership or staff.

No Reports Yet Published

Chairs, when logged in, may publish draft and final reports. Please see report requirements.

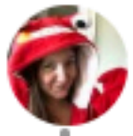
PUBLISH REPORTS

Tools for this group

- Mailing List
- IRC
- Github repository
- RSS
- Contact This Group

Resources

WHY???



achtung bitte
@ag_dubs

this but about developer technolgy

3:51 PM - 9 Apr 2019

178 Retweets **1,181** Likes

10 178 1.2K

HTML5 Spec, Design Principles

"In cases of conflict, consider users over authors over implementors over specifiers over theoretical purity."

rustwasm / wasm-bindgen

Watch 95 | Star 2,271 | Fork 263

Code | Issues 99 | Pull requests 5 | Actions | Security | Insights | Settings

Facilitating high-level interactions between wasm modules and JavaScript https://rustwasm.github.io/docs/wasm-... Edit

wasm javascript rust binding-generator rust-wasm Manage topics

2,715 commits | 3 branches | 56 releases | 1 environment | 145 contributors | View license

Branch: master | New pull request | Create new file | Upload files | Find File | Clone or download

alexcrichton Merge pull request #1625 from alexcrichton/less-return-ptr ... Latest commit 792ab40 1 hour ago

.cargo	Start running CI tests on Rust beta	9 months ago
benchmarks	Run fmt and clippy	29 days ago
ci	Attempt to fix CI	21 days ago
crates	Merge pull request #1625 from alexcrichton/less-return-ptr	1 hour ago
examples	Updating a couple examples	17 hours ago
guide	remove warning about caching	yesterday
releases	Add a template for release announcements	last year
src	Merge pull request #1625 from alexcrichton/less-return-ptr	1 hour ago

wasm-pack

📦✨ your favorite rust -> wasm workflow tool!

✨ INSTALL WASM-PACK 0.8.1 ✨

4 Apr 2019 | Release Notes

wasm-bindgen-futures 0.3.24

Homepage Documentation Repository Dependent crates

Cargo.toml `wasm-bindgen-futures = "0.3.24"`

Last Updated
7 days ago

Crate Size
8.79 kB

wasm-bindgen-futures

API Documention

This crate bridges the gap between a Rust `Future` and a JavaScript `Promise`. It provides two conversions:

1. From a JavaScript `Promise` into a Rust `Future`.
2. From a Rust `Future` into a JavaScript `Promise`.

See the API documentation for more info.

Authors

- The wasm-bindgen Developers

License

MIT/Apache-2.0

Owners

js-sys 0.3.24

Homepage Documentation Repository Dependent crates

Cargo.toml

```
js-sys = "0.3.24"
```

Last Updated
7 days ago

Crate Size
58.42 kB

js-sys

API documentation

Raw bindings to JS global APIs for projects using `wasm-bindgen`. This crate is handwritten and intended to work in *all* JS environments like browsers and Node.js.

Authors

- The wasm-bindgen Developers

License

MIT/Apache-2.0

Categories

web-sys 0.3.24

Homepage Documentation Repository Dependent crates

Cargo.toml

```
web-sys = "0.3.24"
```

`web-sys`

Raw bindings to Web APIs for projects using `wasm-bindgen`.

- The `web-sys` section of the `wasm-bindgen` guide
- API Documentation

Crate features

This crate by default contains very little when compiled as almost all of its exposed

Last Updated
7 days ago

Crate Size
0.2 MB

Authors

- The wasm-bindgen Developers

License

MIT/Apache-2.0

Owners

rustwasm / gloo

Unwatch 57 | Star 488 | Fork 23

Code | Issues 37 | Pull requests 4 | Actions | Security | Insights | Settings

A modular toolkit for building fast, reliable Web applications and libraries with Rust and Wasm

Edit

Manage topics

125 commits | 2 branches | 0 releases | 11 contributors | View license

Branch: master | New pull request | Create new file | Upload files | Find File | Clone or download

fitzgen Merge pull request #89 from rustwasm/fitzgen-patch-1 ··· Latest commit e8e505f 22 days ago

.ci	Generate `README.md`s for each crate from its top-level docs	last month
.github/ISSUE_TEMPLATE	Add a bit about the API's skeleton to the API proposal template	3 months ago
crates	Fix link to CI build in README template	22 days ago
guide	Include each crate's `README.md` in a crates reference guide section	last month
rfcs	get rid of separate FileList struct	2 months ago
src	Fixing all the issues with gloo-events	3 months ago
.README.tpl	Fix link to CI build in README template	22 days ago
.azure-pipelines.yml	Generate `README.md`s for each crate from its top-level docs	last month

🤠 The Wrangler CLI: Deploying Rust with WASM on Cloudflare Workers

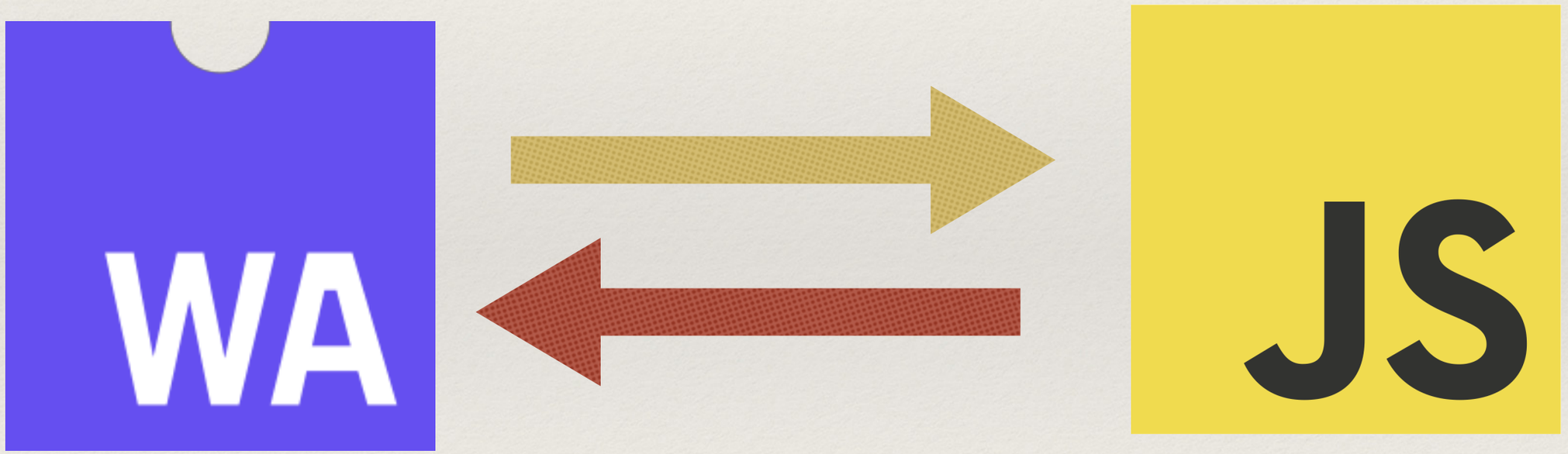
28 Mar 2019 by Ashley Williams.

Wrangler is a CLI tool for building Rust WebAssembly Workers

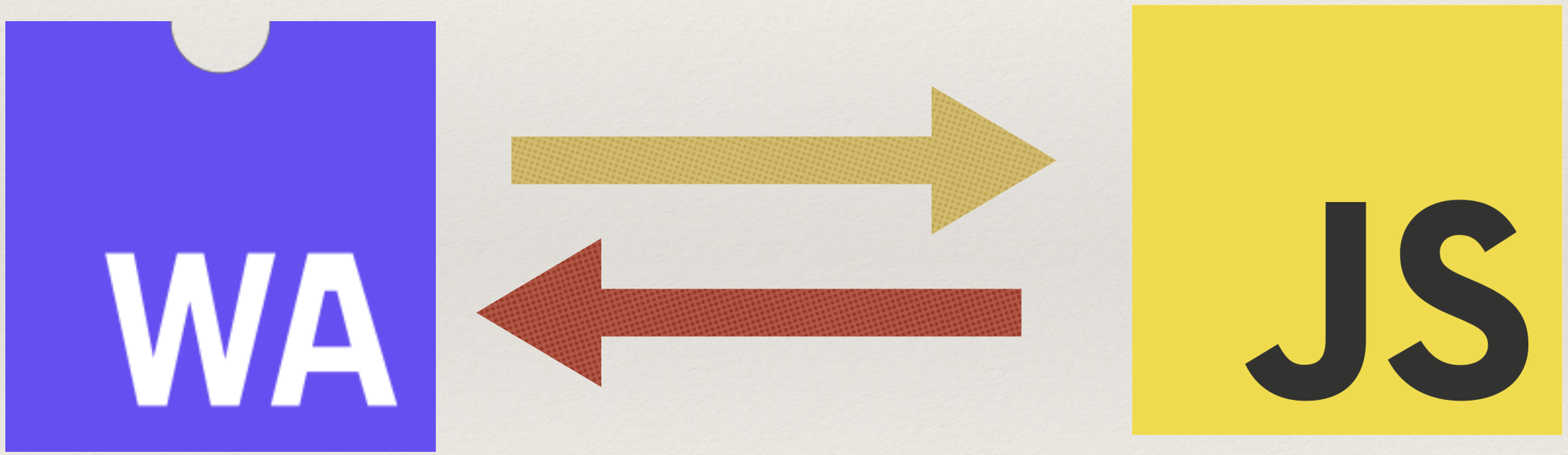
Today, we're open sourcing and announcing `wrangler`, a CLI tool for building, previewing, and publishing Rust and WebAssembly Cloudflare Workers.

If that sounds like some word salad to you, that's a reasonable reaction. All three of the technologies involved are relatively new and upcoming: WebAssembly, Rust, and Cloudflare Workers.

Wasm should be an implementation detail



Most people should never know they are even using WebAssembly



WebAssembly is a technology that invites new types of applications written in multiple different languages to be discovered and distributed on the web

Let’s go forth and make amazing things that further expand the web platform!

… or go forth and bridge the language divide between 2 or even 3 worlds.

You’ll learn so much.

Thanks!

QCon NYC, 26 June 2019

Rust, WebAssembly, and JavaScript make three: An FFI Story

@ag_dubs
Rust Core Team
RustWasm WG