# The Legend of WebAssembly

## And The Missing Link

# Who?

# Aaron Turner

@torch2424

AS

WasmBoy

WA

Wasm By Example

Language: AssemblyScript (TypeScript-like)

WebAssembly (Wasm) is an universal low level bytecc
languages like Rust, AssemblyScript (Typescript-like
offer a compact binary format, with predictable perform
shipped in all major browsers, and has runtimes mean
using WASI.

Made with WebAssembly

Home About

Search for a use case, technology, or project:

E.g Games, Rust, Wasm By Example ...

Google Earth
Google Earth renders a 3D representation of Earth based primarily on satellite imagery on the Web.

WasmBoy
Game Boy / Game Boy Color Emulator Library, written for WebAssembly using AssemblyScript...

VaporBoy
A Gameboy / Gameboy Color Emulator PWA for Android, iOS, Windows, MacOS, and Linux.

Wasm By Example
Wasm By Example is a website with a set of hands-on introduction examples and tutorials for We...

D3Wasm

fastly

# Agenda

# WebAssembly


Aaron Turner - @torch2424

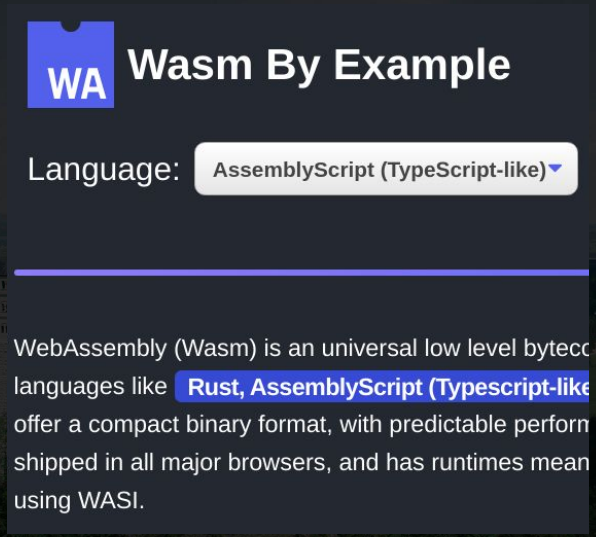
# WASI (WebAssembly System Interface)


Aaron Turner - @torch2424

# Explore WASI for Containerization


Aaron Turner - @torch2424

# Building WebAssembly (WASI) Modules




Aaron Turner - @torch2424

# Running WASI Modules

## Wasmtime: a WebAssembly Runtime

**A Bytecode Alliance project**

Wasmtime is a standalone wasm-only optimizing runtime for WebAssembly and WASI. It runs WebAssembly code outside of the Web, and can be used both as a command-line utility or as a library embedded in a larger application.

To get started, visit wasmtime.dev.


There are Rust, C, and C++ toolchains that can compile programs with WASI. See the WASI intro for more information, and the WASI tutorial for a tutorial on compiling and running programs using WASI and wasmtime, as well as an overview of the filesystem sandboxing system.

Wasmtime passes the WebAssembly spec testsuite. To run it, update the `tests/spec_testsuite` submodule with `git submodule update --remote`, and it will be run as part of `cargo test`.

Wasmtime does not yet implement Spectre mitigations, however this is a subject of ongoing research.

# What is WebAssembly?


Aaron Turner - @torch2424

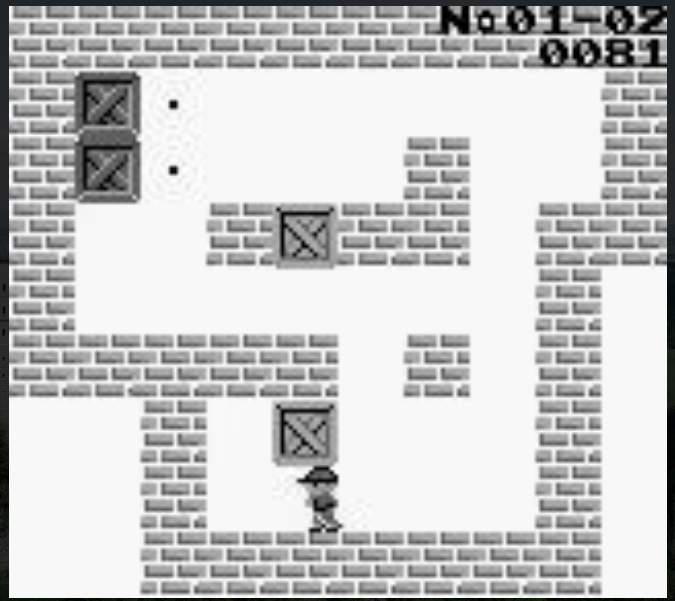
# Bytecode on the Web

Aaron Turner - @torch2424

# WebAssembly Offers Predictable Performance

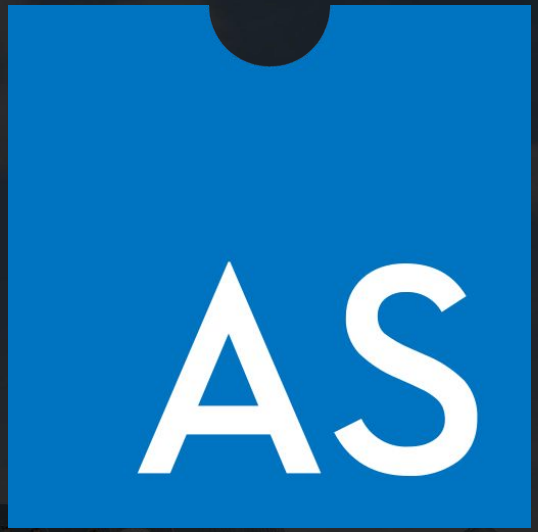

Aaron Turner - @torch2424

# WebAssembly is Portable



# Runs on:
# All Major Browsers
# Node.js
# Standalone Runtimes


Aaron Turner - @torch2424





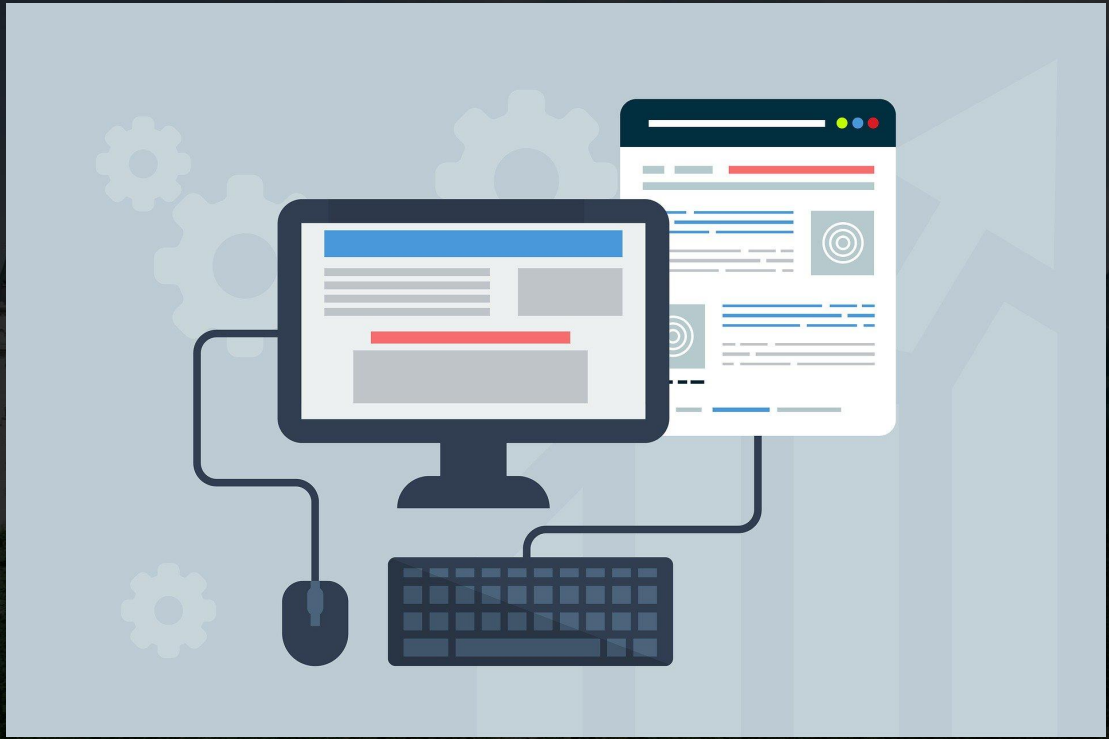
emscripten

Aaron Turner - @torch2424

# WebAssembly uses Linear Memory

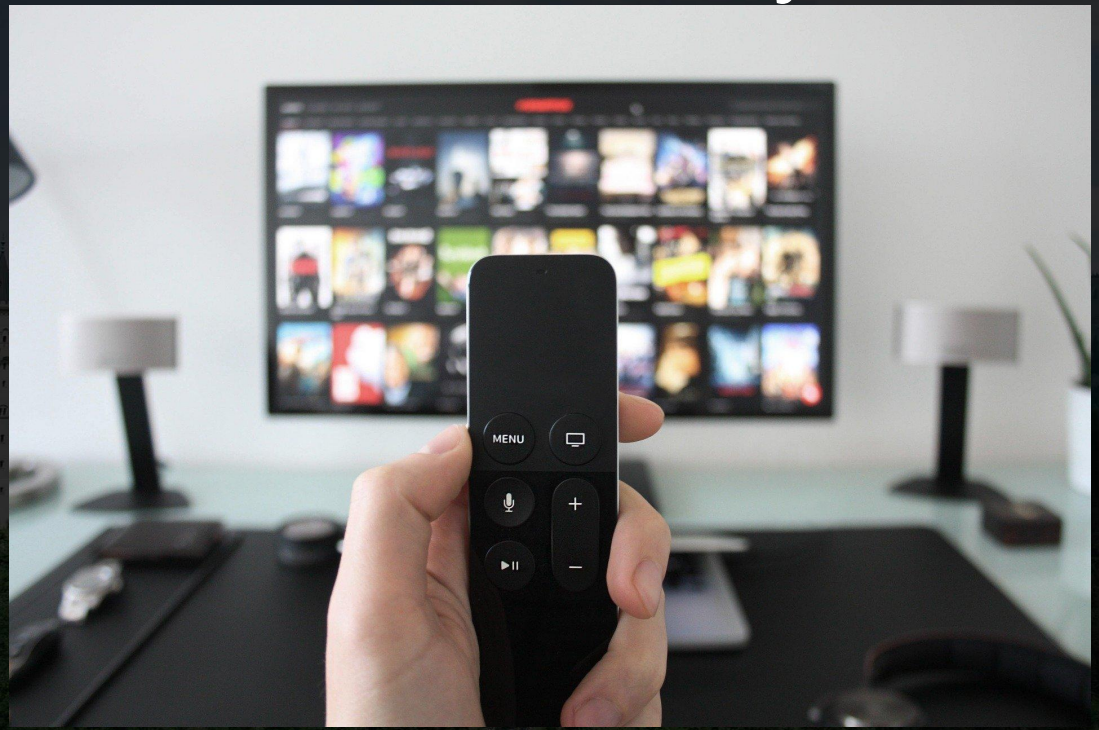
# WebAssembly Relies on Capabilities

# What is WASI?



# WASI is a System Interface for WebAssembly

# All the benefits of WebAssembly, with controlled access to System Calls



**Solomon Hykes**
@solomonstre

If WASM+WASI existed in 2008, we wouldn't have needed to created Docker. That's how important it is. Webassembly on the server is the future of computing. A standardized system interface was the missing link. Let's hope WASI is up to the task!

> **Lin Clark** @linclark · Mar 27, 2019
> WebAssembly running outside the web has a huge future. And that future gets one giant leap closer today with...
>
> 📢 Announcing WASI: A system interface for running WebAssembly outside the web (and inside it too)
>
> hacks.mozilla.org/2019/03/standa...
> Show this thread

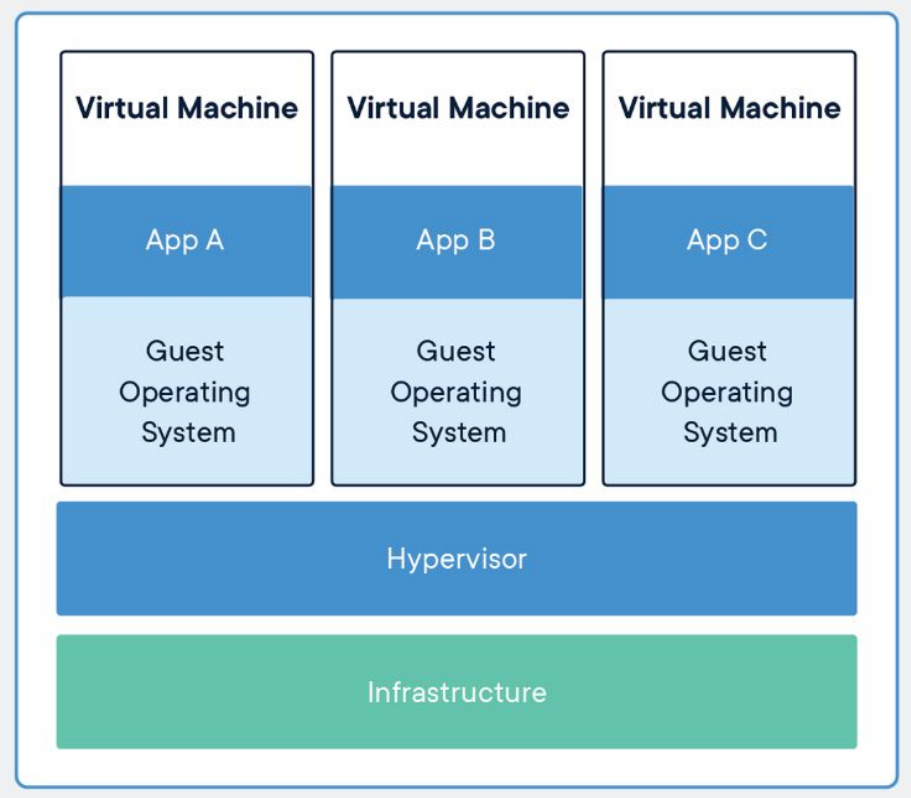
1:39 PM · Mar 27, 2019 · Twitter Web Client

**743** Retweets **1.7K** Likes

"Wow"

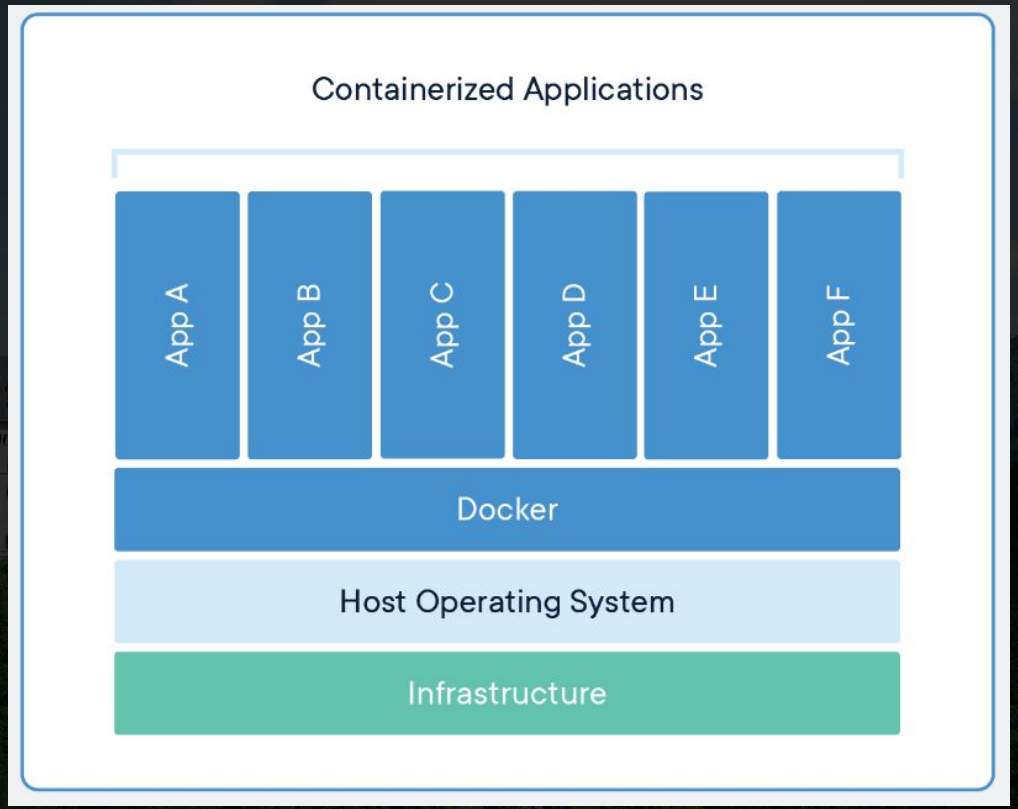
Aaron Turner - @torch2424

# Virtual Machine Architecture



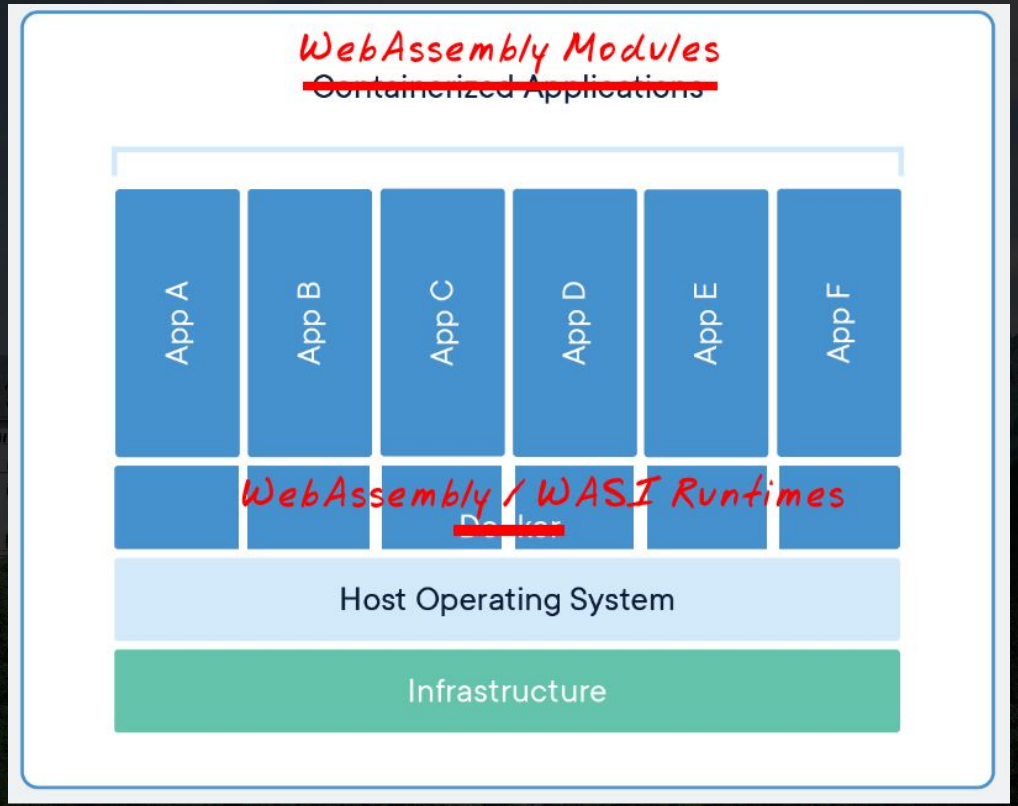

Source: https://www.docker.com/resources/what-container

# Container Architecture



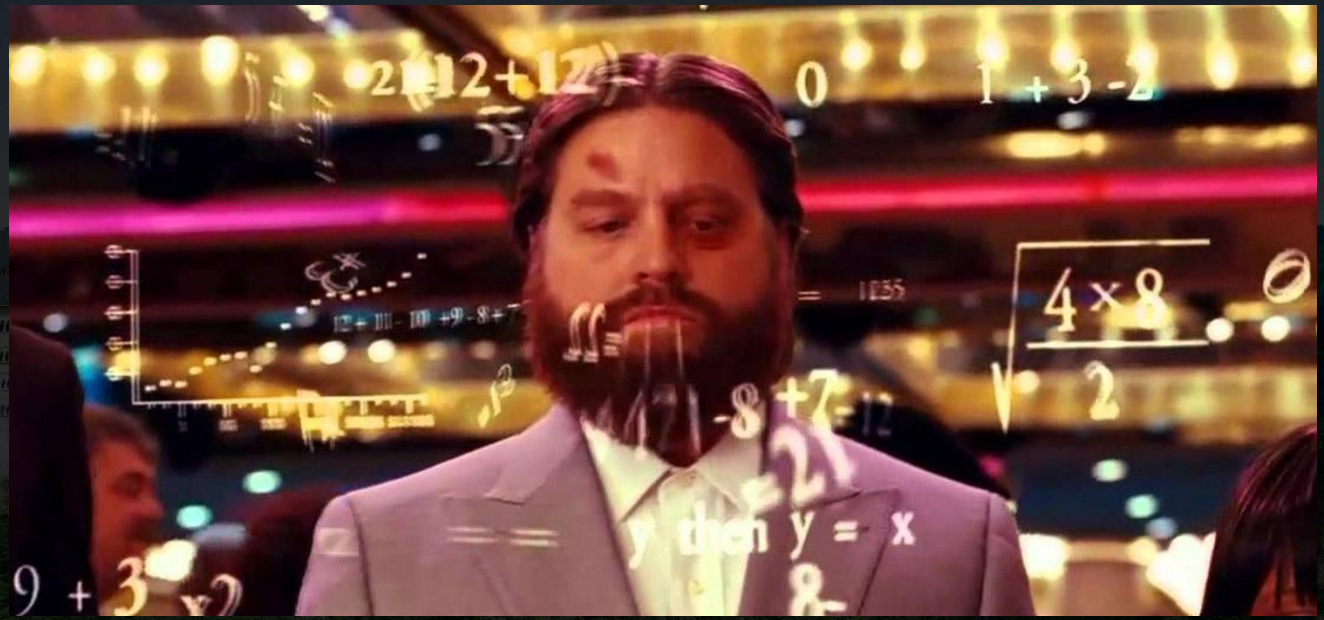

Source: https://www.docker.com/resources/what-container

# Wasm / Wasi “Containerized” Architecture




Source: https://www.docker.com/resources/what-container

# Okay, but how is this different from modern containerization?



# WebAssembly modules are small

# WebAssembly Runtimes are fast, and use very little memory


# WebAssembly + Docker

@tiborvass & @tonistiigi

WA SI

docker

WebAssembly SF Meetup @Docker
May 23, 2019

0:01 / 47:47

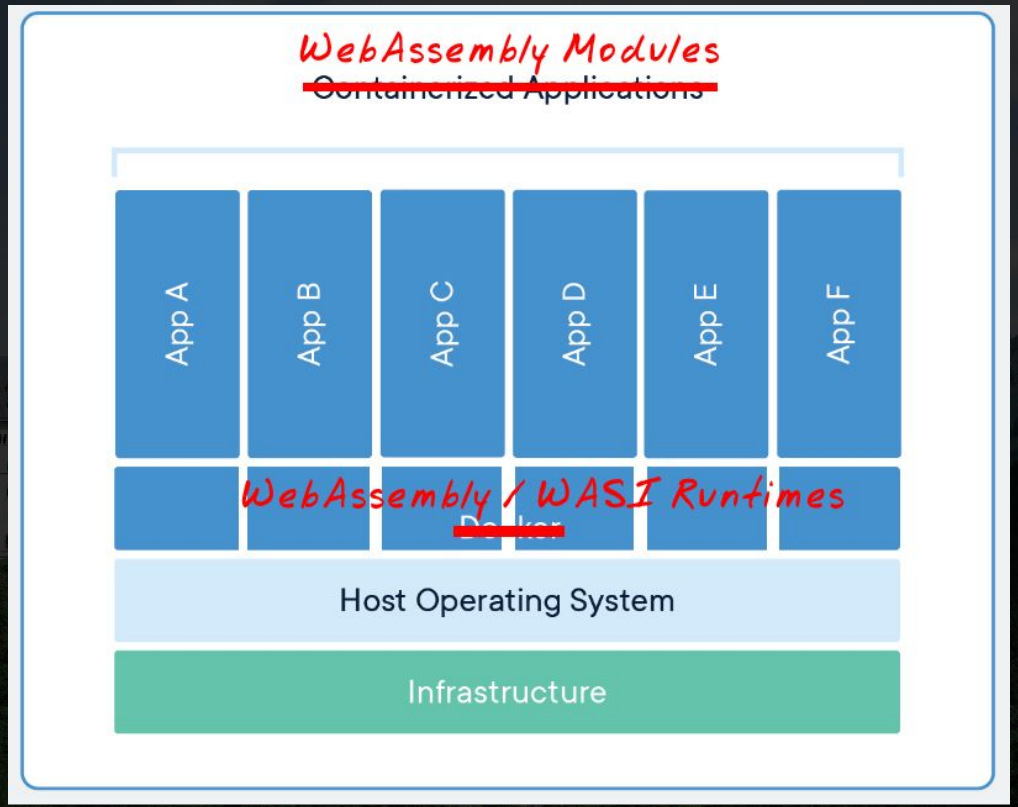
DOCKER INC

Docker + WebAssembly, Tõnis Tiigi & Tibor Vass

221 views • May 29, 2019

11 0 SHARE SAVE

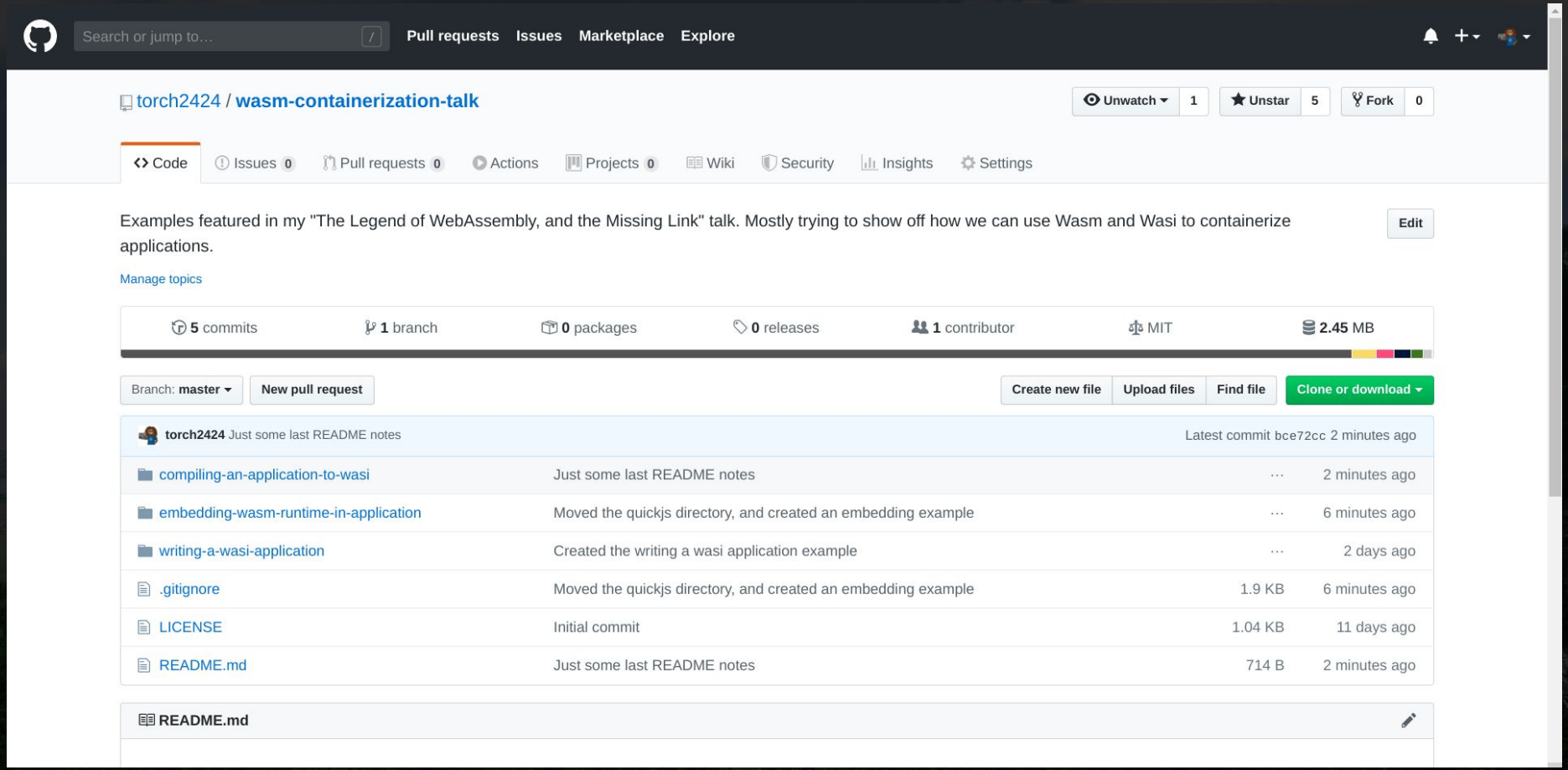
# Wasm / Wasi “Containerized” Architecture




Source: https://www.docker.com/resources/what-container

# Let’s write a Wasm / WASI application

And run it on our “server”

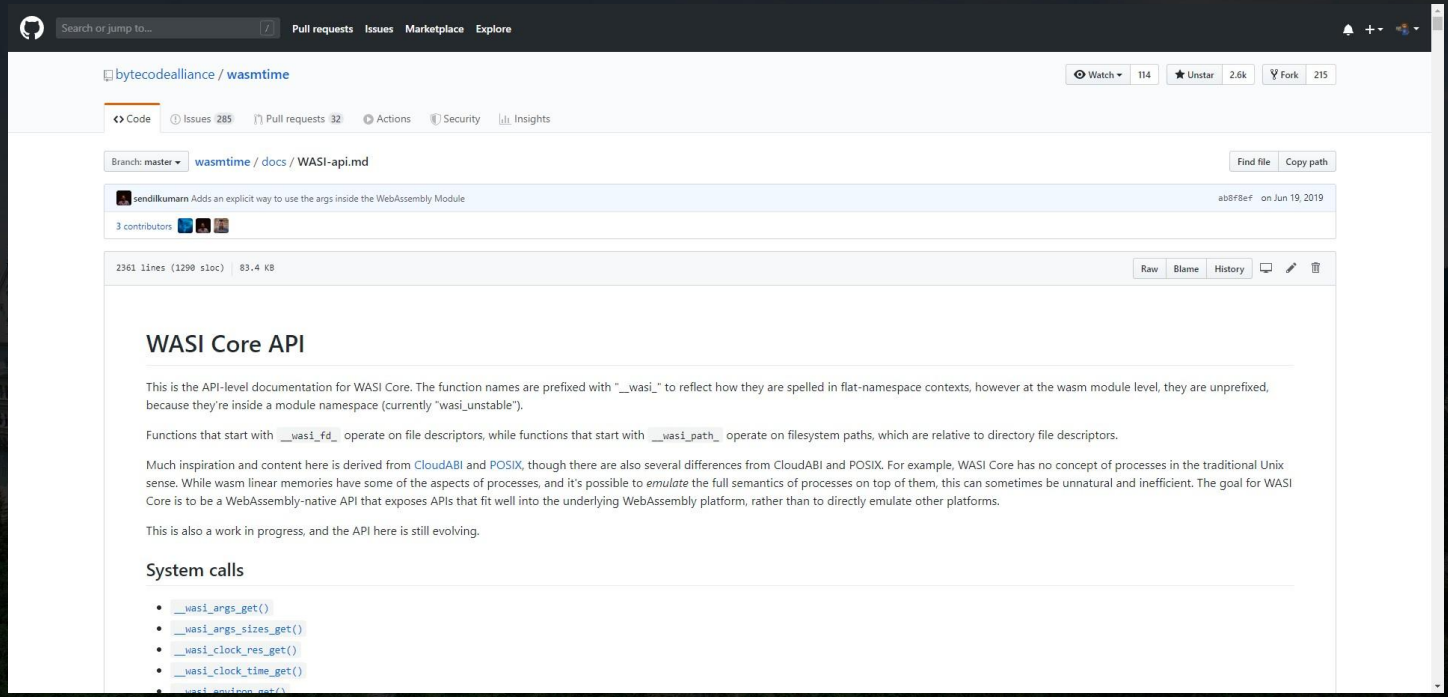

Search or jump to…
Pull requests
Issues
Marketplace
Explore
torch2424 / wasm-containerization-talk
Unwatch 1
Unstar 5
Fork 0
Code
Issues 0
Pull requests 0
Actions
Projects 0
Wiki
Security
Insights
Settings
Examples featured in my "The Legend of WebAssembly, and the Missing Link" talk. Mostly trying to show off how we can use Wasm and Wasi to containerize applications.
Edit
Manage topics
5 commits
1 branch
0 packages
0 releases
1 contributor
MIT
2.45 MB
Branch: master
New pull request
Create new file
Upload files
Find file
Clone or download
torch2424 Just some last README notes
Latest commit bce72cc 2 minutes ago
compiling-an-application-to-wasi Just some last README notes 2 minutes ago
embedding-wasm-runtime-in-application Moved the quickjs directory, and created an embedding example 6 minutes ago
writing-a-wasi-application Created the writing a wasi application example 2 days ago
.gitignore Moved the quickjs directory, and created an embedding example 1.9 KB 6 minutes ago
LICENSE Initial commit 1.04 KB 11 days ago
README.md Just some last README notes 714 B 2 minutes ago
README.md

Aaron Turner - @torch2424

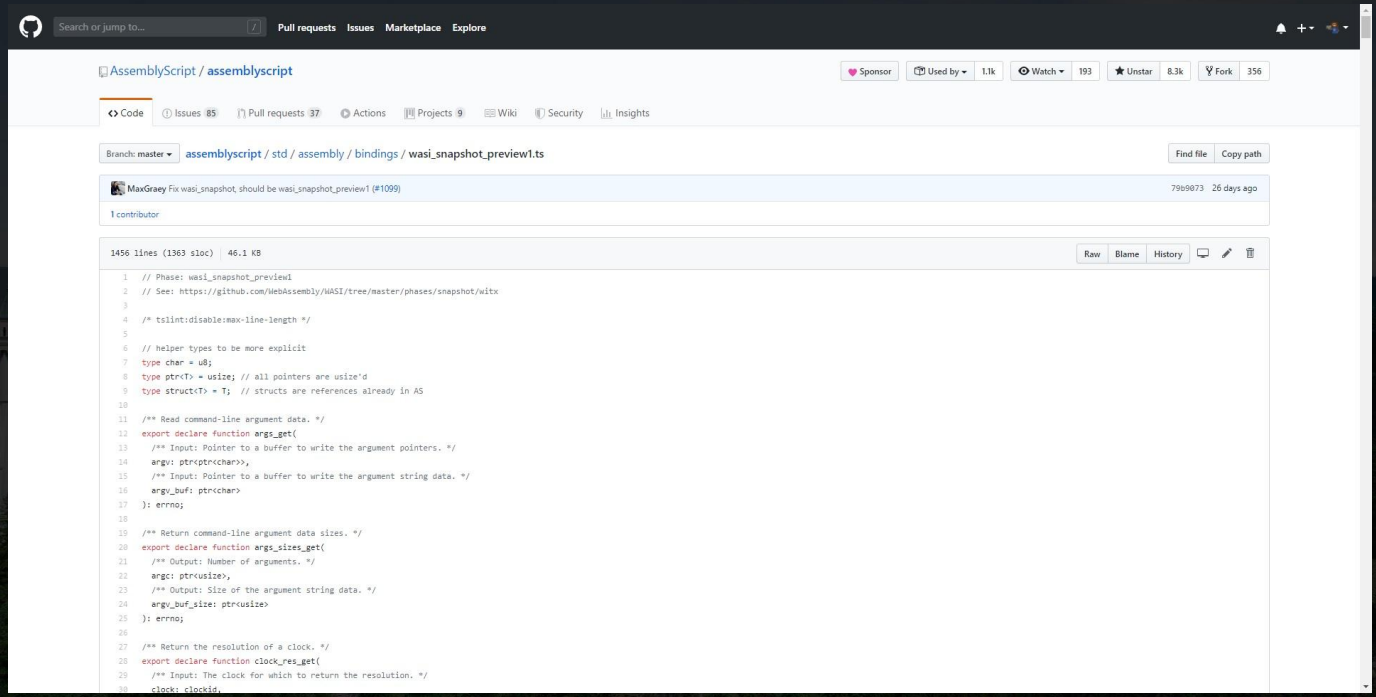
# WASI “Hello World” in AssemblyScript


Aaron Turner - @torch2424

# WASI API


Search or jump to...
Pull requests
Issues
Marketplace
Explore
bytecodealliance / wasmtime
Watch 114
Unstar 2.6k
Fork 215
Code
Issues 285
Pull requests 32
Actions
Security
Insights
Branch: master
wasmtime / docs / WASI-api.md
Find file
Copy path
sendilkumarn Adds an explicit way to use the args inside the WebAssembly Module
ab8f8ef on Jun 19, 2019
3 contributors
2361 lines (1290 sloc) 83.4 KB
Raw
Blame
History
WASI Core API
This is the API-level documentation for WASI Core. The function names are prefixed with "__wasi_" to reflect how they are spelled in flat-namespace contexts, however at the wasm module level, they are unprefixed, because they're inside a module namespace (currently "wasi_unstable").
Functions that start with __wasi_fd_ operate on file descriptors, while functions that start with __wasi_path_ operate on filesystem paths, which are relative to directory file descriptors.
Much inspiration and content here is derived from CloudABI and POSIX, though there are also several differences from CloudABI and POSIX. For example, WASI Core has no concept of processes in the traditional Unix sense. While wasm linear memories have some of the aspects of processes, and it's possible to emulate the full semantics of processes on top of them, this can sometimes be unnatural and inefficient. The goal for WASI Core is to be a WebAssembly-native API that exposes APIs that fit well into the underlying WebAssembly platform, rather than to directly emulate other platforms.
This is also a work in progress, and the API here is still evolving.
System calls
__wasi_args_get()
__wasi_args_sizes_get()
__wasi_clock_res_get()
__wasi_clock_time_get()

# WASI API - AssemblyScript Bindings

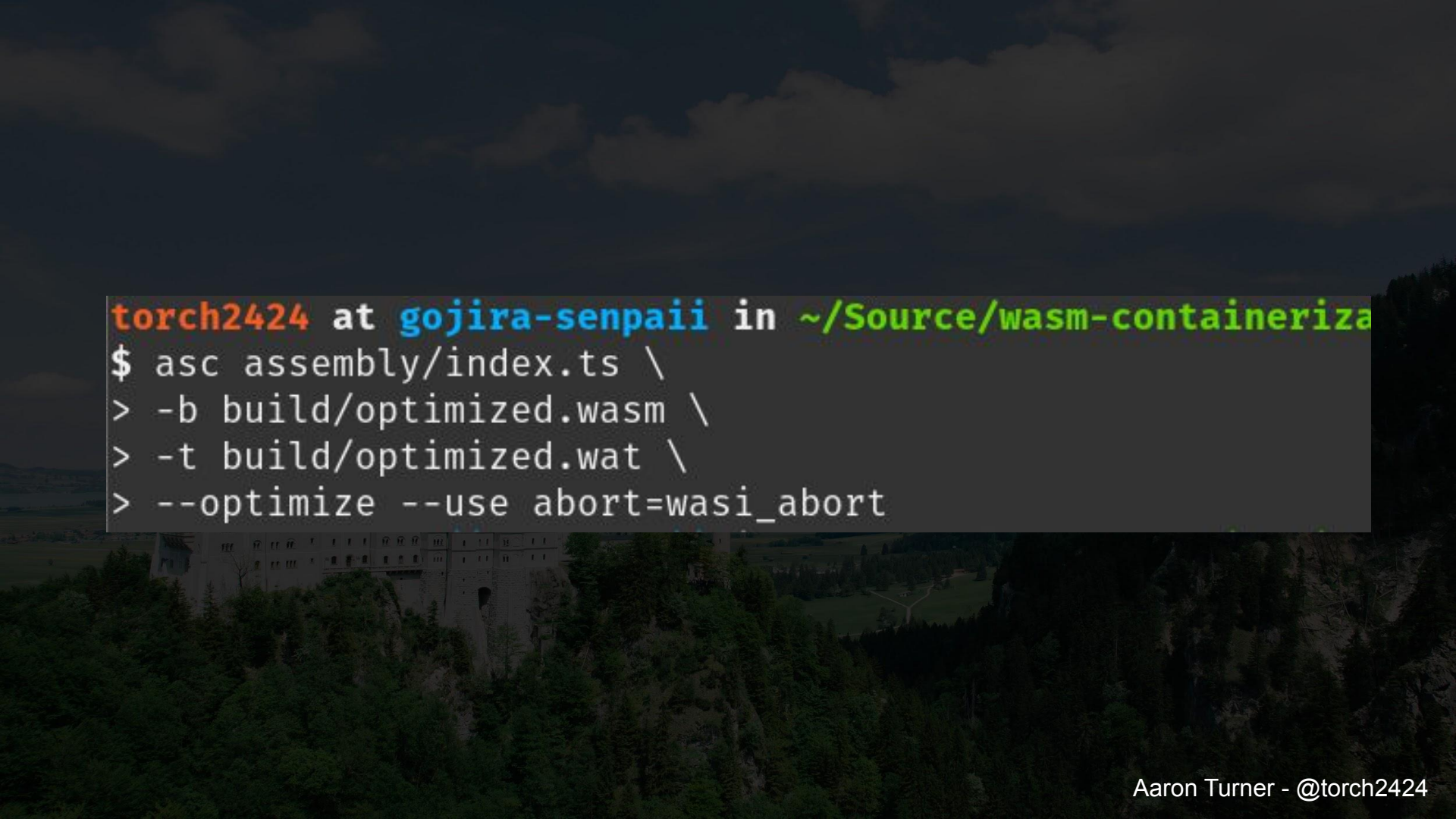


```ts
// Import our WASI Bindings
import {
  fd_write,
  proc_exit
} from "bindings/wasi";

function log(s: string): void {

  // Add a newline to the string
  s += "\n";

  // Convert the string into a UTF8 byte buffer
  let s_utf8_len: usize = String.UTF8.byteLength(s);
  let s_utf8 = changetype<usize>(String.UTF8.encode(s));

  // Allocate an array buffer in Wasm Memory
  let iov = changetype<usize>(new ArrayBuffer(2 * sizeof<usize>()));

  // Write the string buffer, into the allocated array buffer in Wasm
  store<u32>(iov, s_utf8);
  store<u32>(iov + sizeof<usize>(), s_utf8_len);

  // Get the length of an element in an array buffer
  let iovs_len = changetype<usize>(new ArrayBuffer(sizeof<usize>()));

  // Call the WASI fd_write syscall.
  // First parameter is the file descriptor to write to. (1 -> /dev/stdout)
  fd_write(1, iov, 1, iovs_len);
}

// Entry point into our module
export function _start(): void {
  log("Hello WASI!");
}

// Handle Aborting the program on errors.
@global
export function wasi_abort(
  message: string = "",
  fileName: string = "",
  lineNumber: u32 = 0,
  columnNumber: u32 = 0
): void {
  log("Abort!");
  proc_exit(255);
}
```


// Import our WASI Bindings
import {
fd_write,
proc_exit
} from "bindings/wasi";

```
function log(s: string): void {

  // Add a newline to the string
  s += "\n";

  // Convert the string into a UTF8 byte buffer
  let s_utf8_len: usize = String.UTF8.byteLength(s);
  let s_utf8 = changetype<usize>(String.UTF8.encode(s));

  // Allocate an array buffer in Wasm Memory
  let iov = changetype<usize>(new ArrayBuffer(2 * sizeof<usize>()));

  // Write the string buffer,
  // into the allocated array buffer in Wasm Memory
  store<u32>(iov, s_utf8);
  store<u32>(iov + sizeof<usize>(), s_utf8_len);

  // Get the length of an element in an array buffer
  let iovs_len = changetype<usize>(new ArrayBuffer(sizeof<usize>()));

  // Call the WASI fd_write syscall.
  // First parameter is the file descriptor to write to. (1 ->
/d fd_write)
  fd_write(1, iov, 1, iovs_len);
}
```


// Entry point into our module
export function _start(): void {
log("Hello WASI!");
}
Aaron Turner - @torch2424


torch2424 at gojira-senpaii in ~/Source/wasm-containeriza
$ asc assembly/index.ts \
> -b build/optimized.wasm \
> -t build/optimized.wat \
> --optimize --use abort=wasi_abort

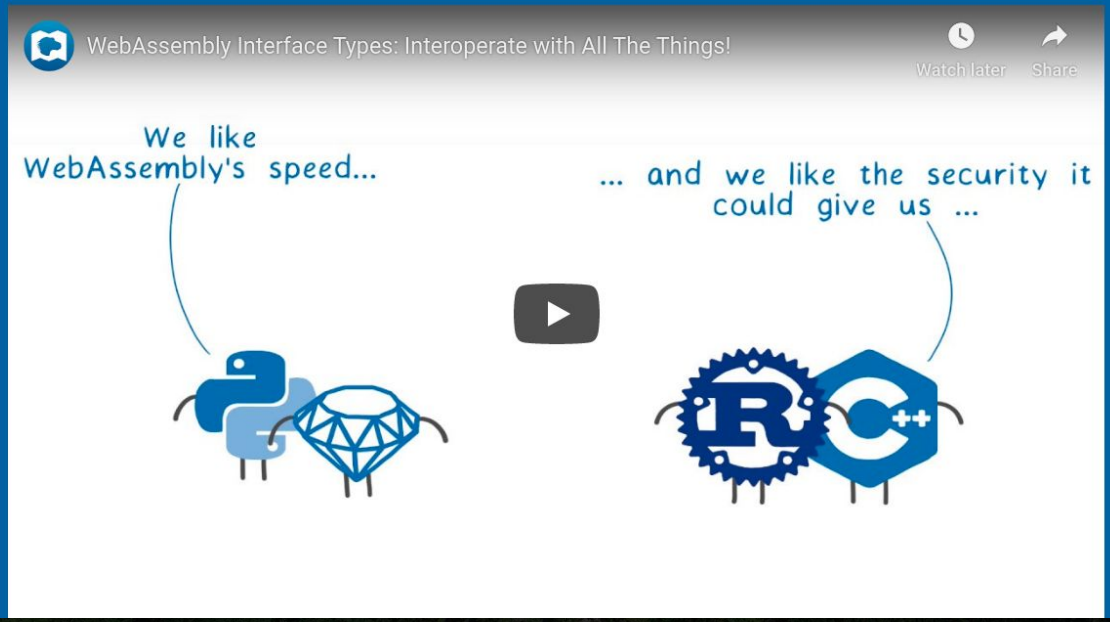
# But How do we run this Wasm File?

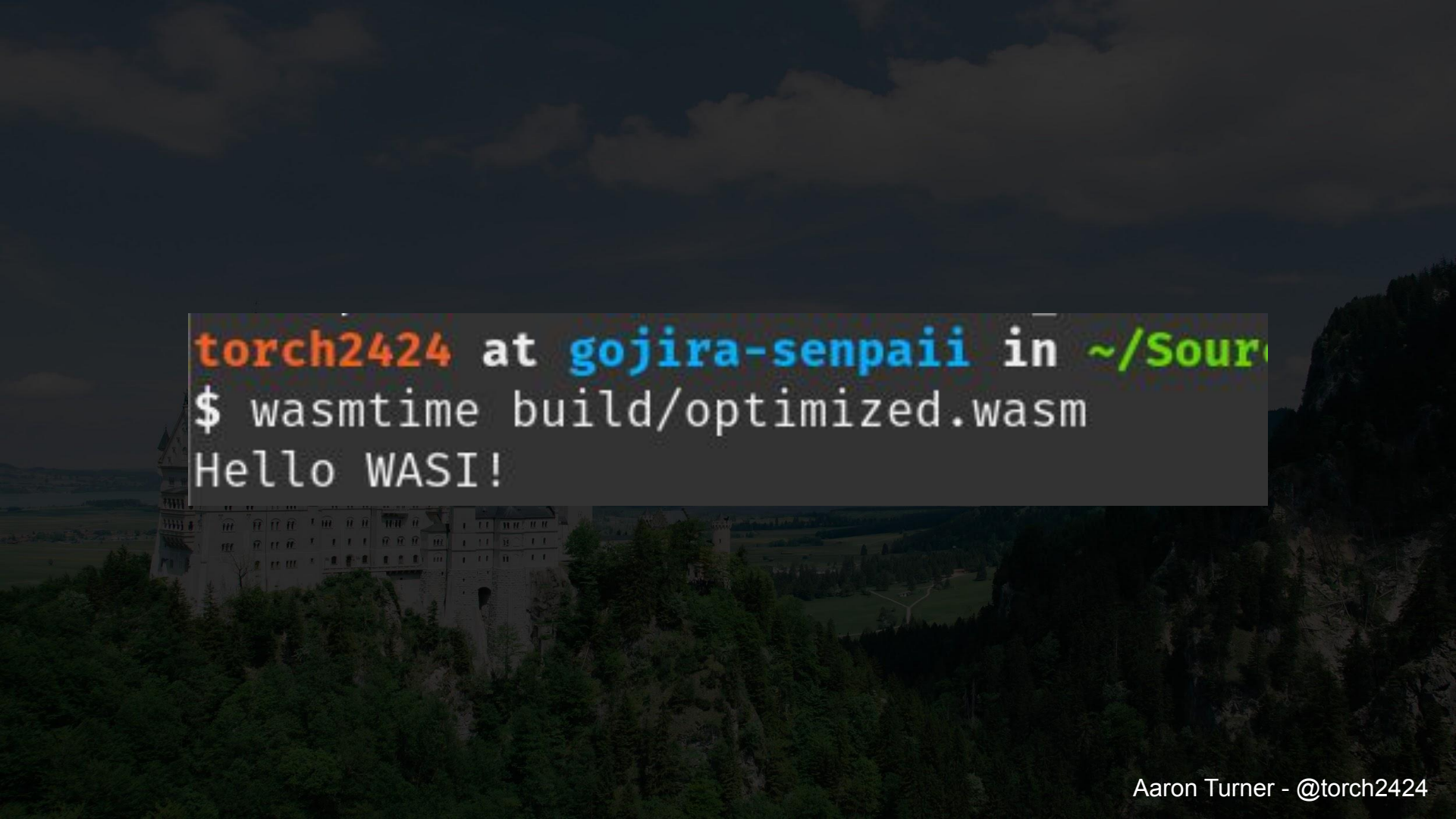
# Wasmtime

## A small and efficient runtime for WebAssembly & WASI

Start using the latest developments in WebAssembly and WASI with the Wasmtime runtime.

Wasmtime supports the WebAssembly Interface Types proposal:




torch2424 at gojira-senpaii in ~/Sour
$ wasmtime build/optimized.wasm
Hello WASI!

# Let’s compile an application to WebAssembly / WASI

## And run it on our “server”

# QuickJS

## QuickJS Javascript Engine

### News

- 2020-01-19:
  - New release with experimental BigDecimal support and updated operator overloading.
  - Small Javascript programs to compute one billion digits of pi.
- 2019-12-21:
  - New release
- 2019-10-27:
  - New release

### Introduction

QuickJS is a small and embeddable Javascript engine. It supports the ES2020 specification including modules, asynchronous generators, proxies and BigInt.

It optionally supports mathematical extensions such as big decimal floating point numbers (BigDecimal), big binary floating point numbers (BigFloat) and operator overloading.

Main Features:

- Small and easily embeddable: just a few C files, no external dependency, 210 KiB of x86 code for a simple `hello world` program.
- Fast interpreter with very low startup time: runs the 69000 tests of the ECMAScript Test Suite in about 95 seconds on a single core of a desktop PC. The complete life cycle of a runtime instance completes in less than 300 microseconds.
- Almost complete ES2019 support including modules, asynchronous generators and full Annex B support (legacy web compatibility). Many features from the upcoming ES2020 specification are also supported.
- Passes nearly 100% of the ECMAScript Test Suite tests when selecting the ES2019 features.
- Can compile Javascript sources to executables with no external dependency.
- Garbage collection using reference counting (to reduce memory usage and have deterministic behavior) with cycle removal.
- Mathematical extensions: BigDecimal, BigFloat, operator overloading, bigint mode, math mode.
- Command line interpreter with contextual colorization implemented in Javascript.
- Small built-in standard library with C library wrappers.

### Benchmark

### Online Demo

An online demonstration of the QuickJS engine with its mathematical extensions is available at numcalc.com. It was compiled from C to WASM/asm.js with Emscripten.

`qjs` and `qjscalc` can be run in JSLinux.

### Documentation

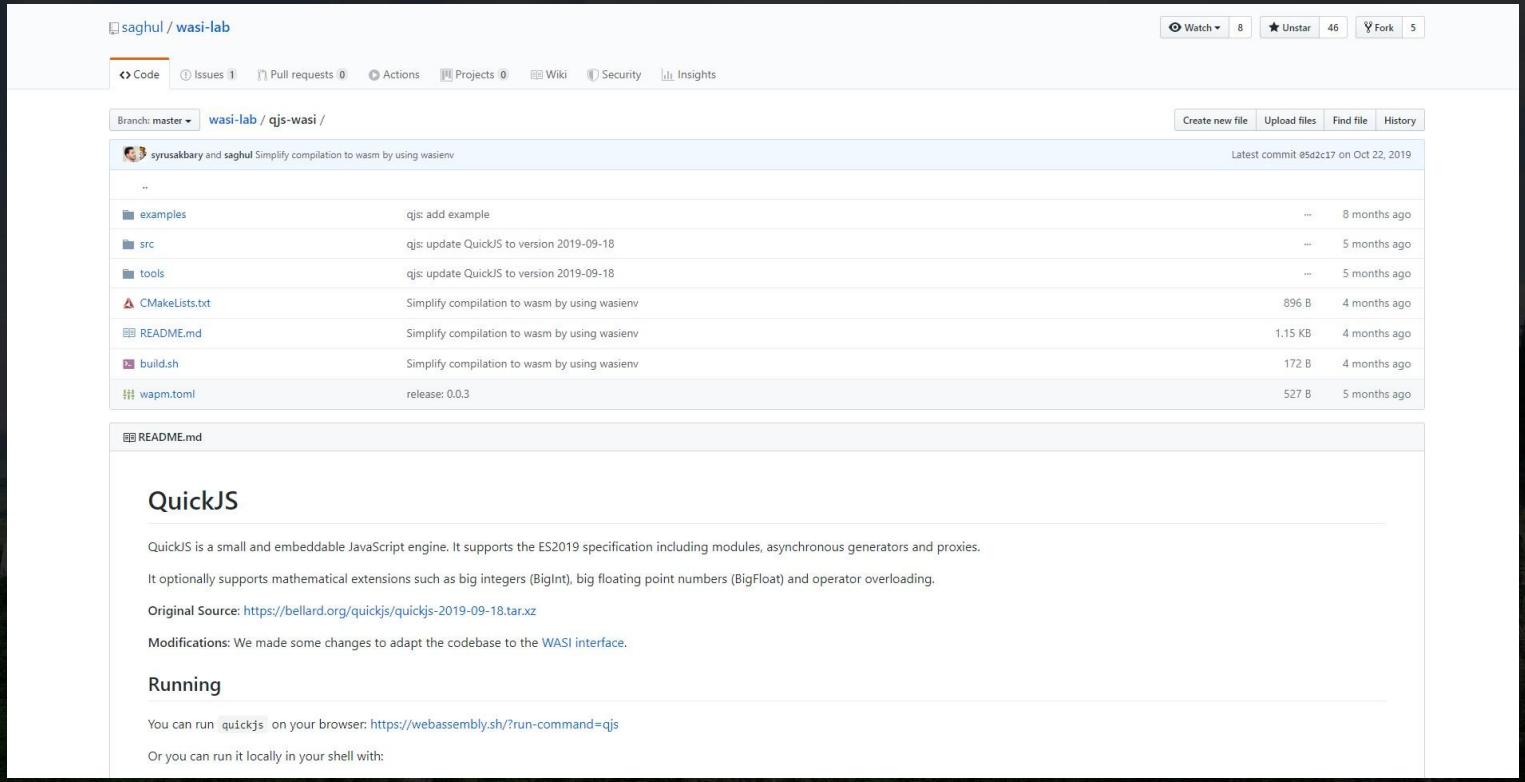
QuickJS documentation: HTML version, PDF version.

Specification of the JS Bignum Extensions: HTML version, PDF version.

### Download

- QuickJS source code: quickjs-2020-01-19.tar.xz
- QuickJS extras (contain the unicode files needed to rebuild the unicode tables and the bench-v8 benchmark): quickjs-extras-2020-01-19.tar.xz
- Unofficial git mirror

# QuickJS Compiled To WASI

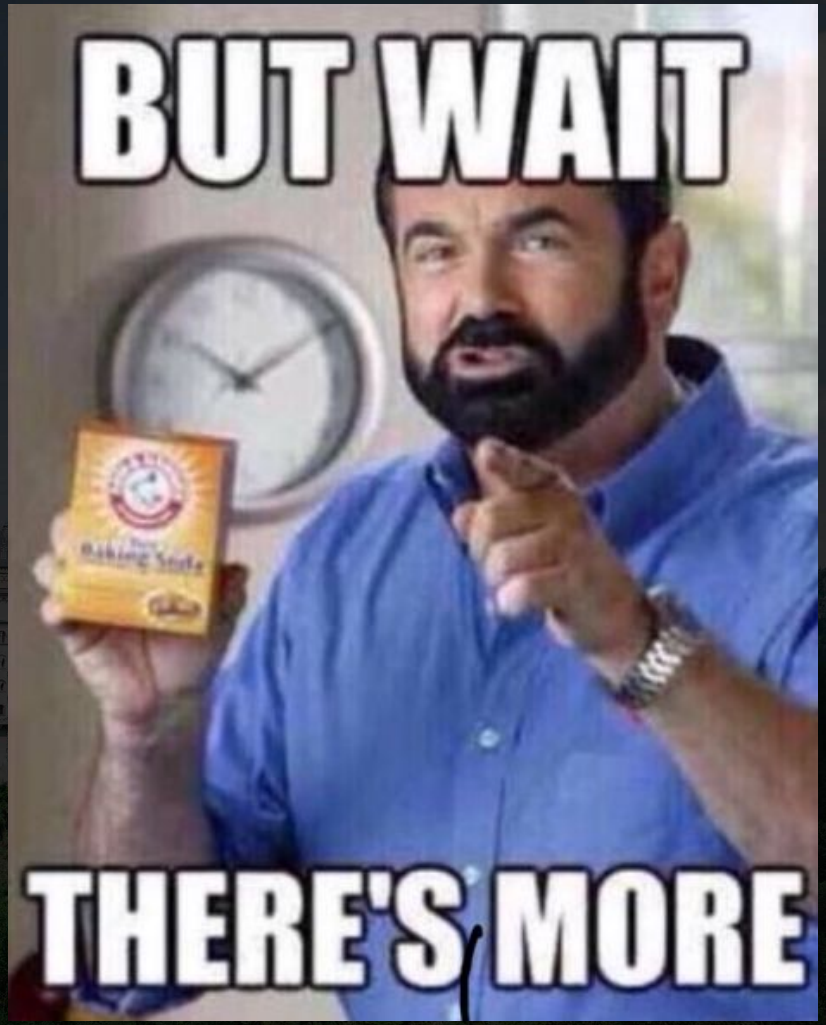


# Compile...

```
torch2424 at gojira-senpaii in ~/Source/wasm-containerization-talk-examples/compiling-an-application-to-wasi/qjs-wasi on master*
$ mkdir build
torch2424 at gojira-senpaii in ~/Source/wasm-containerization-talk-examples/compiling-an-application-to-wasi/qjs-wasi on master*
$ cd build/
torch2424 at gojira-senpaii in ~/Source/wasm-containerization-talk-examples/compiling-an-application-to-wasi/qjs-wasi/build on master*
$ wasimake cmake ..
-- The C compiler identification is Clang 9.0.0
-- The CXX compiler identification is Clang 9.0.0
-- Check for working C compiler: /home/torch2424/.wasienv/bin/wasicc
-- Check for working C compiler: /home/torch2424/.wasienv/bin/wasicc -- works
-- Detecting C compiler ABI info
-- Detecting C compiler ABI info - done
-- Detecting C compile features
-- Detecting C compile features - done
-- Check for working CXX compiler: /home/torch2424/.wasienv/bin/wasic++
-- Check for working CXX compiler: /home/torch2424/.wasienv/bin/wasic++ -- works
-- Detecting CXX compiler ABI info
-- Detecting CXX compiler ABI info - done
-- Detecting CXX compile features
-- Detecting CXX compile features - done
-- No build type selected, default to Release
-- Building in Release mode
-- Building with Clang 9.0.0 on Generic-1
-- Configuring done
-- Generating done
-- Build files have been written to: /home/torch2424/Source/wasm-containerization-talk-examples/compiling-an-application-to-wasi/qjs-wasi/build
torch2424 at gojira-senpaii in ~/Source/wasm-containerization-talk-examples/compiling-an-application-to-wasi/qjs-wasi/build on master*
$ cd ..
torch2424 at gojira-senpaii in ~/Source/wasm-containerization-talk-examples/compiling-an-application-to-wasi/qjs-wasi on master*
$ make -C build
make: Entering directory '/home/torch2424/Source/wasm-containerization-talk-examples/compiling-an-application-to-wasi/qjs-wasi/build'
make[1]: Entering directory '/home/torch2424/Source/wasm-containerization-talk-examples/compiling-an-application-to-wasi/qjs-wasi/build'
make[2]: Entering directory '/home/torch2424/Source/wasm-containerization-talk-examples/compiling-an-application-to-wasi/qjs-wasi/build'
Scanning dependencies of target qjs
make[2]: Leaving directory '/home/torch2424/Source/wasm-containerization-talk-examples/compiling-an-application-to-wasi/qjs-wasi/build'
```

# It Works!

```
torch2424 at gojira-senpaii in ~/Source/wasm-containerizati
$ cat hello-world.js
console.log('Hello World from QuickJS WASI!');
torch2424 at gojira-senpaii in ~/Source/wasm-containerizati
$ wasmtime qjs-wasi/build/qjs.wasm --dir=. hello-world.js
Hello World from QuickJS WASI!
```




Aaron Turner - @torch2424

# Embedding Wasmtime in Rust

This document shows how to embed Wasmtime using the Rust API, and run a simple wasm program.

## Create some wasm

Let's create a simple WebAssembly file with a single exported function that returns an integer:

```
(;; wat2wasm hello.wat -o $WASM_FILES/hello.wasm ;;)
(module
  (func (export "answer") (result i32)
     i32.const 42
  )
)
```

## Create rust project

```
$ cargo new --bin wasmtime_hello
$ cd wasmtime_hello
$ cp $WASM_FILES/hello.wasm .
```

We will be using the wasmtime engine/API to run the wasm file, so we will add the dependency to `Cargo.toml`:

```
[dependencies]
wasmtime = "<current version>"
```

# Let’s write a Rust/Wasm Module, that is Embedded into a Rust application


Aaron Turner - @torch2424

# Write the Guest Rust/Wasm Module

```rust
mod utils;
use wasm_bindgen::prelude::*;

// When the `wee_alloc` feature is enabled, use `wee_alloc` as the
// global allocator.
#[cfg(feature = "wee_alloc")]
#[global_allocator]
static ALLOC: wee_alloc::WeeAlloc = wee_alloc::WeeAlloc::INIT;

// Let's define our external function
#[wasm_bindgen]
extern "C" {
  #[wasm_bindgen(js_namespace = console)]
  fn host_import();
}

// Let's define and add import
#[wasm_bindgen]
pub fn add(a: i32, b: i32) -> i32 {
    return a + b;
}

// Let's define a function, to call our host import function
#[wasm_bindgen]
pub fn call_host_import() {
    host_import();
}
```

```
torch2424 at gojira-senpaii in ~/Source/wasm-containerization-talk-examples/em
bedding-wasm-runtime-in-application/guest-wasm-module on master
$ wasm-pack build
[INFO]: Checking for the Wasm target...
[INFO]: Compiling to Wasm...
warning: function is never used: `set_panic_hook`
 --> src/utils.rs:1:8
  |
1 | pub fn set_panic_hook() {
  |        ^^^^^^^^^^^^^^
  |
  = note: `#[warn(dead_code)]` on by default

    Finished release [optimized] target(s) in 0.02s
[INFO]: Installing wasm-bindgen...
[INFO]: Optimizing wasm binaries with `wasm-opt`...
[INFO]: Optional fields missing from Cargo.toml: 'description', 'repository',
and 'license'. These are not necessary, but recommended
[INFO]: :-) Done in 0.09s
[INFO]: :-) Your wasm pkg is ready to publish at /home/torch2424/Source/wasm-c
ontainerization-talk-examples/embedding-wasm-runtime-in-application/guest-wasm
-module/pkg.
```

# Embed Wasmtime to run the Guest Module

```rust
extern crate wasmtime;
use std::error::Error;
use wasmtime::*;

fn main() -> Result<(), Box<dyn Error>> {
    // Global store for the wasmtime instance
    let store = Store::default();

    // Get our guest wasm module
    let module = Module::from_file(
      &store,
      "../guest-wasm-module/pkg/guest_wasm_module_bg.wasm"
    )?;

    // Create our imported host function
    let host_import = Func::wrap0(&store, || {
        println!("host_import called!");
    });

    // Create our wasmtime instance,
    // and import our host function to our module
    let instance = Instance::new(&module, &[host_import.into()])?;

    // Use the wasmtime API to get our exported add function from
    // our guest wasm module
    let guest_add = instance.get_export("add")
        .expect("export named `add` not found")
        .func()
        .expect("export `add` was not a function");
    let guest_add = guest_add.get2::<i32, i32, i32>()?;

    // Call our add function from the guest wasm module
    let result = guest_add(24, 24)?;
    println!("Answer: {:?}", result);

    // Use the wasmtime API to get our exported call_host_import function
from// our guest wasm module
    let guest_call_host_import = instance.get_export("call_host_import")
        .expect("export named `call_host_import` not found")
        .func()
        .expect("export `call_host_import` was not a function");
    let guest_call_host_import = guest_call_host_import.get0::<_>()?;

    // Call our export call_host_import. This call, should then cause
    // host_import to be called, and log to the console
    guest_call_host_import()?;

    Ok(())
}
```

```rust
// Global store for the wasmtime instance
let store = Store::default();

// Get our guest wasm module
let module = Module::from_file(
  &store,
  "../guest-wasm-module/pkg/guest_wasm_module_bg.wasm"
)?;

// Create our imported host function
let host_import = Func::wrap0(&store, || {
  println!("host_import called!");
});

// Create our wasmtime instance,
// and import our host function to our module
let instance = Instance::new(&module, &[host_import.into()])?;
```

```rust
// Use the wasmtime API to get our exported add function from
// our guest wasm module
let guest_add = instance.get_export("add")
    .expect("export named `add` not found")
    .func()
    .expect("export `add` was not a function");
let guest_add = guest_add.get2::<i32, i32, i32>()?;

// Call our add function from the guest wasm module
let result = guest_add(24, 24)?;
println!("Answer: {:?}", result);

// Use the wasmtime API to get our exported call_host_import function
// from our guest wasm module
let guest_call_host_import = instance.get_export("call_host_import")
    .expect("export named `call_host_import` not found")
    .func()
    .expect("export `call_host_import` was not a function");
let guest_call_host_import = guest_call_host_import.get0::<_>()?;

// Call our export call_host_import. This call, should then cause
// host_import to be called, and log to the console
guest_call_host_import()?;
```

# Run the host application, and it works!

```
torch2424 at gojira-senpaii in ~/Source/wasm-containerization-talk-examples/embedding-wasm-runtime-in-application/host-application on master*
$ cargo run
   Compiling host-application v0.1.0 (/home/torch2424/Source/wasm-containerization-talk-examples/embedding-wasm-runtime-in-application/host-application)
    Finished dev [unoptimized + debuginfo] target(s) in 5.27s
     Running `target/debug/host-application`
Answer: 48
host_import called!
```

# In Conclusion...

- WebAssembly / WASI are exciting new technologies for performance, portability, and security.
- Compiling applications for WASI, can allow for an architecture of sandboxed, and “containerize” applications running in Wasm runtimes.
- Embedding Wasm Runtimes in your application, allow applications to run WebAssembly compiled libraries for code re-use and portability.

- Additional Resources
  - Learn WebAssembly
    - WasmByExample, Language Docs
  - Popular Wasm Runtimes
    - Wasmtime.dev
    - Lucet
  - Talks
    - Patrick Hamann | WebAssembly - To the browser and beyond! | performance.now() 2019
    - Kevin Hoffman | Building a Containerless Future with WebAssembly | Wasm Summit 2020

# Thank You!!!

@torch2424

Images: Pixabay, Memes, or taken by me.