Programming Webassembly with Rust

A Presentation By Collins Muriuki (c12i)

Get Started →

Get The Slides

rustwasm.c12i.xyz

Content

What is WASM?

Understanding WASM

According to webassembly.org, WebAssembly (WASM) a portable binary instruction format

for a stack based Virtual Machine.

It's designed to be portable and can be used as a compilation target for higher level programming languages like C++, Rust, Go etc.

Wasm enables deployment on the web for client and server applications offering near native performance while providing security through sandboxed execution.

WASM Architecture

"...a portable binary instruction format..."

The operations encoded in a wasm module are not tightly coupled to any hardware architecture or OS. They are just codes that a parser is able to interpret. In a very similar fashion to Python and Java bytecode.

"...for stack based Virtual Machine..."

This is the aspect that makes wasm *blazingly* fast, and is also the source of some of it's limitations. WASM uses a stack data structure for all its operations. Simplifies validation and ensures predictable execution behavior.

"...deployment on the web..."

Now this is a controversial definition and might potentially limit your understanding of wasm. This essentially means it's a portable format that can run anywhere you can build a host. Limiting to the scope of the web is a dis-service.

Host vs Guest in WASM

Host

- The environment running the wasm module, e.g; browsers, Node.js, Embedded runtimes
- Controls resources and provides APIs as host functions
- Manages memory allocation
- Handles I/O operations

Guest

- The WASM module itself from your compiled code
- Runs in a sandboxed environment
- Limited direct access to system resources*

WASM FAQs

Question: Is Wasm a transpile target for JavaScript?

No - Wasm is not a JS transpile target like JSX to JS. However, you can **compile** TypeScript to Wasm using AssemblyScript.

Question: Is Wasm meant to replace JavaScript?

While controversial, JS and Wasm have a symbiotic relationship. JavaScript is still needed to host Wasm in the browser.

Question: Is Wasm a programming language?

While you can write Wasm by hand, it’s impractical beyond basic examples. Understanding it can help with troubleshooting, but it’s not recommended for direct development.

Question: Can WebAssembly run independently?

No - like a video game disc needs a console, Wasm requires a host environment. Its sandboxed nature is actually one of its strengths.

WASM FAQs (ctd)

Question: Is WASM async?

WASM itself is synchronous - any async capabilities come from the host environment. In the browser, it's async via JavaScript.

Question: Does WASM support multithreading?

Yes, but through host APIs - for example, using Web Workers in the browser environment.

Question: Can I use Rust features like generics and trait objects in my WASM code?

Yes, but with limitations. Generics work with internal code, but any code exposed via the WASM module should use concrete types.

History and evolution

2015-16: Early Development

- asm.js influenced WebAssembly design
- First demos at Mozilla
- Initial specifications developed

2017: MVP Stable Release

- Core features established
- Major browser support achieved

2018-19: Ecosystem Growth

- W3C standardization
- Tools and frameworks emerge
- WASI initiative begins
- `wasm-pack` for Rust and `emscripten` for C++
- Commercial adoption increases

2020-21: Beyond the Browser

- WASI development matures
- Server applications emerge
- Standalone runtimes appear (wasmtime, wasmer)

2022-Present: New Frontiers

- Component model development
- Garbage collection proposal
- Interface types
- Threading model improvements

Applications and Use-Cases

- Web Applications
- Server Side computing
- Edge computing
- IOT/ Embedded systems
- Containerization

Solomon Hykes
@solomonstre · Follow

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
>
> 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...

11:39 PM · Mar 27, 2019

2.2K Reply Copy link

Read 29 replies

Writing Wasm by Hand & Interacting with JavaScript

WAT (WebAssembly Text Format)

- Human readable representation of wasm
- Uses S-expressions like Lisp
- Direct correlation with binary format

WAT (ctd.)

- Basic structure

```
(module
  ;; Memory declarations go here
  ;; Functions go here
  ;; Imports go here
  ;; Exports go here
)
```

WAT (ctd.)

- Simple Add function

```
(module
  (func $add (param $a i32) (param $b i32) (result i32)
    local.get $a
    local.get $b
    i32.add
  )
  (export "add" (func $add))
)
```

WAT (ctd.)

- Memory operations

```
(module
  (memory (export "memory") 1)  ;; Export 1 page of memory (64KB)

  (func $storeValue (param $value i32) (param $index i32)
    local.get $index        ;; Get index parameter
    local.get $value        ;; Get value parameter
    i32.store              ;; Store at memory[index]
  )

  (func $loadValue (param $index i32) (result i32)
    local.get $index       ;; Get index parameter
    i32.load              ;; Load from memory[index]
  )

  (export "store" (func $storeValue))
  (export "load" (func $loadValue))
)
```

WAT (ctd.)

- Compiling WAT to WASM

```
wat2wasm math.wat -o math.wasm
```

- You can install `wat2wasm` via Web Assembly Binary Toolkit (wabt) by following the instructions in the README.

WAT (ctd.)

- Loading in JS

```
(async () => {
    // Load the WebAssembly module
    async function init() {
      const response = await fetch("rust_wasm_memory_ops.wasm");
      const bytes = await response.arrayBuffer();

      // Instantiate the module
      const results = await WebAssembly.instantiate(bytes);
      return results.instance;
    }

    const wasmInstance = await init();
    const result = wasmInstance.exports.add(5, 3);
    console.log("5 + 3 =", result);
})();
```

Programming WASM in Rust

Project Setup

Setup

```
cargo new rust_math_wasm
cd rust_math_wasm
```

Add `wasm32-unknown-unknown` target

```
rustup target add wasm32-unknown-unknown
```

Project Setup

`Cargo.toml` modifications

```toml
[package]
name = "rust_math_wasm"
version = "0.1.0"
edition = "2021"

[lib]
crate-type = ["cdylib"] # Produces a compact dynamic library optimal for WebAssembly

[profile.release]
opt-level = "s"     # Optimize for size
lto = true          # Link time optimization
strip = true        # Remove debug symbols
```

Basic Math Functions

- Implementation

```
#[no_mangle]
pub extern "C" fn add(a: i32, b: i32) -> i32 {
    a + b
}

#[no_mangle]
pub extern "C" fn multiply(a: i32, b: i32) -> i32 {
    a * b
}
```

Memory Operations

```
static mut MEMORY: [i32; 100] = [0; 100]; // Static memory for storing values

#[no_mangle]
pub extern "C" fn store(index: i32, value: i32) {
    unsafe {
        MEMORY[index as usize] = value;
    }
}

#[no_mangle]
pub extern "C" fn load(index: i32) -> i32 {
    unsafe {
        MEMORY[index as usize]
    }
}
```

Array Operations

```
#[no_mangle]
pub extern "C" fn sum_array(ptr: *const i32, len: i32) -> i32 {
    let slice = unsafe { std::slice::from_raw_parts(ptr, len as usize) };
    slice.iter().sum()
}
```

Building the Rust WASM Module

```
# Build for WebAssembly target
cargo build --target wasm32-unknown-unknown --release

# Optional: Optimize the wasm file
wasm-gc target/wasm32-unknown-unknown/release/rust_math_wasm.wasm
```

The output file can be loaded directly in the browser!

Loading WASM in JavaScript

```
(async () => {
    // Load the WebAssembly module
    async function init() {
      const response = await fetch("rust_wasm_memory_ops.wasm");
      const bytes = await response.arrayBuffer();

      // Instantiate the module
      const results = await WebAssembly.instantiate(bytes);
      return results.instance;
    }

    const wasmInstance = await init();
    const addResult = wasmInstance.exports.add(5, 3);
    console.log("5 + 3 =", addResult);

    const multiplyResult = wasmInstance.exports.multiply(5, 3);
    console.log("5 x 3 =", multiplyResult);
})();
```

Memory Access from JavaScript

```javascript
// Get the memory buffer
const memory = wasmInstance.exports.memory;
const memoryArray = new Int32Array(memory.buffer);

// Store a value
function storeValue(index, value) {
    wasmInstance.exports.store(index, value);
}

// Load a value
function loadValue(index) {
    return wasmInstance.exports.load(index);
}
```

Array Operations from JavaScript

```javascript
// Create input array
const numbers = [1, 2, 3, 4, 5];
const memoryArray = new Int32Array(wasmInstance.exports.memory.buffer);

// Copy numbers to WebAssembly memory
numbers.forEach((num, i) => {
    memoryArray[i] = num;
});

// Call Wasm function
const sum = wasmInstance.exports.sum_array(0, numbers.length);
console.log('Sum:', sum);
```

Tips and Best Practices

- Keep functions simple and focused
- Use appropriate numeric types
- Be careful with memory management
- Test thoroughly
- Profile performance and optimize where necessary
- Handle errors gracefully

Introducing `wasm-bindgen`

- Raw WebAssembly is limited to numeric types
- Manual memory management is error-prone
- Complex types require tedious serialization
- JavaScript interop is verbose

`wasm-bindgen` provides:

- Rich type conversions
- Automatic memory management
- JavaScript class integration
- Promise/async support
- Error handling

Project Set Up with `wasm-pack`

Initialization

```
cargo install wasm-pack

# Create a new project
wasm-pack new my-wasm-project
```

Project Structure

```
test-project/
├── Cargo.lock
├── Cargo.toml
├── LICENSE_APACHE
├── LICENSE_MIT
├── README.md
├── src
│   ├── lib.rs     # your wasm bindings go here
│   └── utils.rs   # debug utils
└── tests
    └── web.rs
```

📚 Documentation

String Handling

```rust
use wasm_bindgen::prelude::*;
use web_sys::console; // `cargo add web_sys`

// Host function
#[wasm_bindgen]
extern "C" {
    fn alert(s: &str);
}

#[wasm_bindgen]
pub fn greet(name: &str) -> String {
    // Automatic string conversion! 🎉
    alert(&format!("Hello, {}!", name))
}

#[wasm_bindgen]
pub fn log_message(msg: String) {
    // Direct console access
    console::log_1(&msg.into());
}
```

String Handling (ctd.)

Compare to raw Wasm:

- No manual memory allocation
- No string length tracking
- No UTF-8 encoding handling

Compile Module

Build the module with `wasm-pack`

```
wasm-pack build --target web --release
```

- Your module is generated as a standalone local npm package in dir `pkg`
- TypeScript declarations automatically generated
- Publishable to `npm`

Import Module

First, import the wasm module `pkg` in your `package.json`

```
{
  "my-package": "file:../pkg
}
```

Then you can use your module in JS

```
import init, { greet, log_message } from "my-package";

init()
  .then(() => {
    greet("Foo");
    log_message("Hello World");
  })
  .catch(console.error);
```

Complex Types & Classes

Rust struct to JS class

```rust
#[wasm_bindgen]
pub struct User {
    name: String,
    age: u32,
}

#[wasm_bindgen]
impl User {
    #[wasm_bindgen(constructor)]
    pub fn new(name: String, age: u32) -> Self {
        Self {
            name, age,
        }
    }

    #[wasm_bindgen]
    pub fn greet(&self) -> String {
        format!("Hi, I'm {} and I'm {} years old!",
                self.name, self.age)
    }
}
```

Complex Types and Classes (ctd.)

Usage in JavaScript

```javascript
import init, { User } from "my-package";

init()
  .then(() => {
    const user = new User("Alice", 30);
    console.log(user.greet());  // "Hi, I'm Alice and I'm 30 years old!"
  })
  .catch(console.error);
```

Async/Promise Support

```rust
use wasm_bindgen::prelude::*;
use wasm_bindgen_futures::JsFuture;
use web_sys::window;
use js_sys::Promise;

#[wasm_bindgen]
pub async fn fetch_data(url: String) -> Result<JsValue, JsValue> {
    let window = window().unwrap();
    let resp = JsFuture::from(window
        .fetch_with_str(&url))
        .await?;
    Ok(resp)
}

#[wasm_bindgen]
pub fn create_promise() -> Promise {
    Promise::new(&mut |resolve, _reject| {
        resolve.call1(&JsValue::NULL, &JsValue::from_bool(true)).unwrap();
    })
}
```

Async/ Promise Support (ctd.)

```rust
use wasm_bindgen::prelude::*;
use wasm_bindgen_futures::JsFuture;
use web_sys::{Request, RequestInit, RequestMode, Response};

#[wasm_bindgen]
pub async fn fetch_github_user_stats(username: String) -> Result<JsValue, JsValue> {
    let mut opts = RequestInit::new();
    opts.method("GET");
    opts.mode(RequestMode::Cors);

    let url = format!("https://api.github.com/users/{}", username);
    let request = Request::new_with_str_and_init(&url, &opts)?;
    request.headers().set("Accept", "application/vnd.github.v3+json")?;

    let window = web_sys::window().unwrap();
    let response = JsFuture::from(window.fetch_with_request(&request)).await?;
    let response: Response = resp_value.dyn_into().unwrap();

    if !response.ok() {
        return Err(JsValue::from_str(&format!("HTTP error! status: {}", resp.status())));
    }

    Ok(JsFuture::from(response.json()?).await?)
}
```

Async/ Promise Support (ctd.)

Key Features

- Async/await syntax
- JavaScript Promise integration via wasm_bindgen_futures
- Automatic conversion
- Error propagation

Error Handling

Creating a custom error enum

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

#[derive(Debug, Clone)]
pub enum AppError {
    ValidationError(String),
    InternalError(String),
}

// Implement std::error::Error for proper error handling
impl std::error::Error for AppError {}

// Implement Display for error messages
impl core::fmt::Display for AppError {
    fn fmt(&self, f: &mut fmt::Formatter<'_>) -> fmt::Result {
        match self {
            AppError::ValiationError(msg) => write!(f, "Validation Error: {}", msg),
            AppError::InternalError(msg) => write!(f, "Internal Error: {}", msg),
        }
    }
}
```

Error Handling (ctd.)

```rust
// Internal function that returns our custom error type
#[wasm_bindgen]
fn internal_process() -> Result<(), AppError> {
    Err(AppError::NetworkError("Failed to connect".to_string()))
}

// Public WASM function that converts our error to JsError
#[wasm_bindgen]
pub fn process_data() -> Result<(), JsError> {
    internal_process()?;
    Ok(())
}
```

Supporting Libraries

`web-sys` : Browser API bindings

```
use web_sys::{Document, Element, HtmlElement};

let document = window().document().unwrap();
let element = document.create_element("div")?;
element.set_inner_html("Hello from web-sys!");
```

📚 Documentation

Supporting Libraries (cont.)

`js-sys` : JavaScript standard library bindings

```rust
use js_sys::{Array, Date, Object};

let array = Array::new();
array.push(&"Hello".into());
array.push(&42.into());
```

📚 Documentation

Supporting Libraries (cont.)

`gloo`: Ergonomic utilities for web development

```rust
use gloo::timers::callback::Timeout;
use gloo::events::EventListener;
use gloo::storage::{LocalStorage, Storage};

// Timer example
let timeout = Timeout::new(1000, move || {
    console::log_1(&"Timeout fired!".into());
});

// LocalStorage example
LocalStorage::set("key", &value)?;
```

📚 Documentation

Rust-WASM Based UI Libraries

Yew

- One of the oldest and most mature
- React-like development experience
- HTML macro syntax for components
- Strong community and ecosystem
- SSR support

```rust
use yew::prelude::*;

#[function_component]
fn App() -> Html {
    let counter = use_state(|| 0);
    html! {
        <button onclick={move |_| counter.set(*counter + 1)}>
            {*counter}
        </button>
    }
}
```

📚 Documentation

Rust-WASM Based UI Libraries (ctd.)

Dioxus

- Modern, also react like framework
- Cross platform (web, desktop, mobile)
- Strong focus on dev experience
- Unified runtime across platforms
- SSR support with `dioxus-ssr`

```rust
use dioxus::prelude::*;

fn App() -> Element {
    let count = use_state(|| 0);
    rsx! {
        button {
            onclick: move |_| count += 1,
            "{count}"
        }
    }
}
```

📚 Documentation

Rust-WASM Based UI Libraries (ctd.)

Leptos

- Full-stack framework with first class SSR support
- Fine grained reactivity with Signals
- Hydration support
- Server functions/ actions

```rust
use leptos::*;

#[component]
fn Counter() -> impl IntoView {
    let (count, set_count) = create_signal(0);
    view! {
        <button on:click=move |_| set_count.update(|n| *n + 1)>
            {count}
        </button>
    }
}
```

📚 Documentation, and 🎥 Watch This Talk

Rust-WASM Based UI Libraries (ctd.)

Other awesome libraries:

- Sycamore
- Hirola (Authored by our very own community member @geofmureithi)

I have also thrown my hat in the ring, though just tinkering for now

- eltr (A declarative macro based DSL for building UIs)

Alternative WASM Binding Libraries

Beyond `wasm-bindgen`, several other libraries offer unique approaches to WASM bindings

`extism`: Plugin System for WASM

- Designed specifically for plugin systems
- Host-agnostic architecture with controlled host access
- First-class support for multiple languages
- Built-in memory management with isolation
- Standardized plugin interface
- Write in any language

📚 Documentation

Core Concepts

- Host: Your main application
- Plugin: WASM module with extended functionality
- PDK: Plugin Development Kit
- Host SDK: Library to load and run plugin

```
Your App (Host) → Host SDK → Plugin (WASM) → PDK
```

Setting Up a `extism` plugin

Scaffold your plugin via the extism cli

Install

```
curl -s https://get.extism.org/cli | sh
```

Generate plugin (rust crate)

```
extism gen plugin -l Rust -o plugin
```

Basic Plugin Structure

Creating Exports

```rust
use extism_pdk::*;

#[plugin_fn]
pub fn greet(name: String) -> FnResult<String> {
    Ok(format!("Hello, {}!", name))
}

#[plugin_fn]
pub fn count_vowels(input: String) -> FnResult<i32> {
    let count = input.chars()
        .filter(|c| "aeiou".contains(*c))
        .count();
    Ok(count as i32)
}
```

Build plugin

```
cargo build --target wasm32-unknown-unknown
```

Using Extism Plugins

Via cli

```
extism call target/wasm32-unknown-unknown/debug/rust_pdk_template.wasm greet --input "Benjamin"
```

Using Extism Plugins

Host Application (JavaScript)

```
const createPlugin = require("@extism/extism")

const plugin = await createPlugin(
    'target/wasm32-unknown-unknown/debug/rust_pdk_template.wasm',
    { useWasi: true }
);

let out = await plugin.call("count_vowels", "Hello, World!");
console.log(out.text())
```

Using Extism Plugins (ctd)

Host Application (Rust)

```rust
use extism::*;

let wasm = Wasm::file("target/wasm32-unknown-unknown/debug/rust_pdk_template.wasm");
let manifest = Manifest::new([wasm]);

let plugin = Plugin::new(&manifest, [], true).unwrap();

// Call plugin functions
let result = plugin.call::<&str, &str>(
    "greet",
    "World".to_string()
)?;
println!("{}", result); // "Hello, World!"
```

Extism Features

Memory Management

```rust
#[plugin_fn]
pub fn process_data(input: Vec<u8>) -> FnResult<Vec<u8>> {
    // Automatic memory handling
    let mut output = input.clone();
    output.reverse();
    Ok(output)
}
```

Extism Features (ctd.)

Host Functions (pseudocode)

```
HostFunction {
    // The literal name of the function, how it would be called from a plug-in.
    name: string,

    // The types of the input arguments/parameters the plug-in caller will provide.
    input_param_types: Array<WasmValueType>,

    // The types of the output returned from the host function to the plug-in.
    output_return_types: Array<WasmValueType>,

    // An opaque pointer to an object from the host, accessible to the plug-in.
    // NOTE: it is the shared responsibility of the host and plug-in to cast/dereference
    // this value properly.
    user_data: void*,
}
```

`fp-bindgen`: Full-Stack WASM Plugins

- Protocol-first approach using Rust as the protocol format
- Works with stable serialization format (MessagePack)
- Can use existing Rust types directly
- Tight integration with Rust ecosystem
- Supports async functions

GitHub

Background

Setting Up `fp-bindgen`

```toml
# Cargo.toml
[dependencies]
fp-bindgen = "3.0.0"
serde = { version = "1.0", features = ["derive"] }
```

Protocol Definition

```rust
// Define data structures
#[derive(fp_bindgen::prelude::Serializable)]
pub struct MyStruct {
    pub foo: i32,
    pub bar: String,
}

// Define functions that can be called by the guest
fp_bindgen::prelude::fp_import! {
    fn my_imported_function(a: u32, b: u32) -> u32;
}

// Define functions that can be called by the host
fp_bindgen::prelude::fp_export! {
    fn my_exported_function(a: u32, b: u32) -> u32;
}
```

Data Structures

Functions can pass Rust structs and enums as their arguments and return values

```rust
#[derive(fp_bindgen::prelude::Serializable)]
pub struct MyStruct {
    pub foo: i32,
    pub bar: String,
}

fp_bindgen::prelude::fp_import! {
    fn my_function(data: MyStruct) -> MyStruct;
}
```

Async Support

Functions can also be `async`

```
fp_bindgen::prelude::fp_import! {
    async fn my_async_function(data: MyStruct) -> Result<MyStruct, MyError>;
}
```

Using existing structs to avoid unnecessary copies

```rust
use fp_bindgen::prelude::Serializable;

#[derive(Serializable)]
#[fp(rust_module = "my_crate::prelude")]
pub struct MyStruct {
    pub foo: i32,
    pub bar_qux: String,
}
```

Generating `fp-bindgen` Bindings

```rust
let bindings_type = fp_bindgen::BindingsType::RustWasmerRuntime;

fp_bindgen::prelude::fp_bindgen!(fp_bindgen::BindingConfig {
    bindings_type,
    path: &format!("bindings/{}", bindings_type)
});
```

Using the bindings

Using the Rust Plugin Bindings

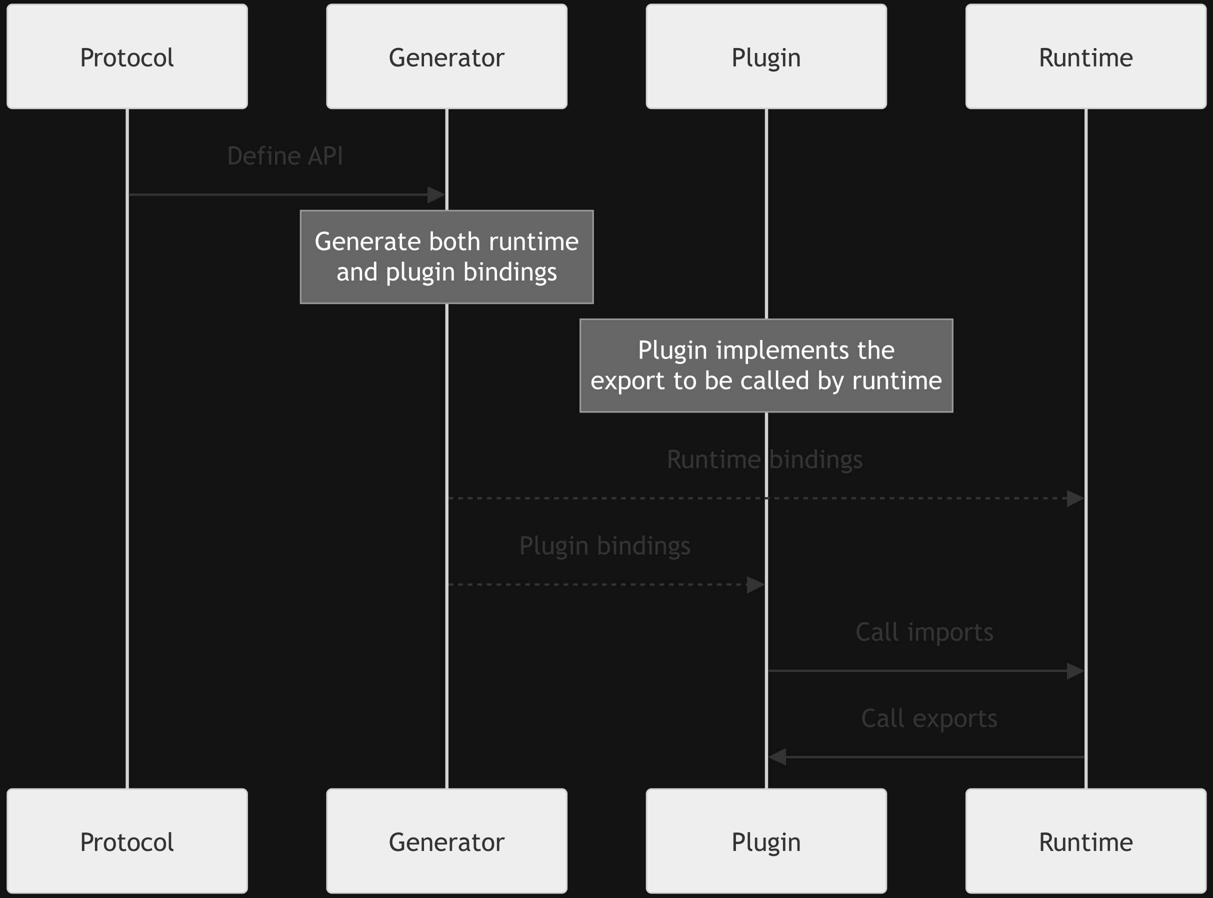
- The generator for the Rust plugin bindings generates a complete crate that allows to be linked against by plugins.
- In order to export the functions that are defined in the `fp_export!` block, use the exported `fp_export_impl` macro, like so:

```rust
#[fp_bindgen_macros::fp_export_impl(bindings_crate_path)]
fn my_exported_function(a: u32, b: u32) -> u32 {
    /* ... */
}
```

Using the bindings (ctd.)

- `bindings_crate_path` is expected to match with the module path from which the bindings crate itself is imported.
- The function signature must match exactly with one of the fp_export! functions.
- Remember to compile the plugin against `wasm32-unknown-unknown`

`fp-bindgen` workflow



Comparing Binding Libraries

Feature	`wasm-bindgen`	`fp-bindgen`	`extism`
Primary Use Case	Browser/JS integration	Full-stack plugins	Cross Language Plugin systems
Host environments	Browser, Node.js	Rust (Wasmer), TypeScript	Language-agnostic
Guest languages	Rust	Rust	Multiple
Serialization	JSON	MessagePack	Custom (via host SDKs)
Learning Curve	Low	High	Medium
Key Strength	DOM/Browser APIs	Complex state handling	Plugin isolation

Choosing the Right Binding Library

Use `wasm-bindgen` ***when***:

- Building web applications
- Need deep JavaScript integration
- Working with DOM/Browser APIs

Use `extism` ***when***:

- Building plugin systems
- Need language-agnostic hosting
- Want simplified memory management

Use `fp-bindgen` ***when***:

- Need strict type safety
- Building cross-platform applications
- Want protocol or interface driven development

Workshop: WASM on the browser in action

The Task

Build a WebAssembly module that processes images with various filters similar to Instagram, using Rust and `wasm-bindgen`. Implement this module in a frontend ui using a library of your choice or plain vanilla JS. A user should be able to upload an image, preview it and select filters to apply to the image from a list. The image filtering logic should be provided by wasm.

Focus on building something that works, not much on ui.

Learning Objectives

- Set up a Rust + WASM project using `wasm-pack`
- Understand image processing in Rust
- Handle binary data transfer between JS and WASM
- Implement common image processing algorithms (use either photon-rs or image)

Preview demo solution

Workshop (ctd.)

Pre-requisites:

- Rust installed
- wasm-pack installed
- Node.js/npm for the web interface
- Nice to have: some basic understanding of image processing concepts

Workshop (ctd.)

Clone starter code

```
# TODO
git clone https://github.com/c12i/image-filters.git
```

> Timebox: 30-45 minutes

WASM Beyond the Browser

WebAssembly System Interface (WASI)

What is WASI?

- System interface for WASM outside the browser
- Standardized API for system calls
- Security through capability-based permissions
- Portable across different operating systems

Using `wasm32-wasi` Target

```
rustup target add wasm32-wasi

# Build for WASI
cargo build --target wasm32-wasi
```

WASI Examples

File System Access

```rust
use std::fs;

fn main() {
    // WASI provides controlled access to files
    let contents = fs::read_to_string("input.txt")
        .expect("Unable to read file");
    println!("File contents: {}", contents);
}
```

WASI Examples (ctd.)

Environment Variables

```rust
use std::env;

fn main() {
    // Access env vars through WASI
    if let Ok(value) = env::var("MY_VAR") {
        println!("MY_VAR = {}", value);
    }
}
```

Raw WASM Modules

Minimal Rust Example

```rust
#[no_mangle]
pub extern "C" fn add(a: i32, b: i32) -> i32 {
    a + b
}
```

WASM Runtimes

Wasmtime

- Mozilla-backed runtime
- WASI reference implementation
- Focus on security and performance
- Good for production environments

Wasmer

- Universal WebAssembly runtime
- Multiple backends (LLVM, Cranelift)
- Package manager support
- Cross-platform support

WAMR (WebAssembly Micro Runtime)

- Lightweight runtime
- Ideal for embedded systems
- Small footprint

Using Different Runtimes

Wasmtime Example

```rust
use wasmtime::*;

// Create engine and module
let engine = Engine::default();
let module = Module::from_file(&engine, "example.wasm")?;

// Store (all wasm objects operate within the context of a store for host specific data)
let mut store = Store::new(&engine, ());

// Instantiate the module
let instance = Instance::new(&mut store, &module, &[])?;

// Get function from instance
let add = instance.get_func(&mut store, "add")
    .expect("add function export");

// Call function with parameters
let result = add.call(&mut store, &[Val::I32(5), Val::I32(37)])?;
println!("5 + 37 = {:?}", result[0].unwrap_i32()); // 42
```

Using Different Runtimes (ctd.)

Wasmer Example

```rust
use wasmer::{Store, Module, Instance, Value};

// Initialize store and module
let mut store = Store::default();
let module = Module::from_file(&store, "example.wasm")?;

// Create instance
let instance = Instance::new(&mut store, &module, &[])?;

// Get function from instance
let add = instance.exports.get_function("add")?;

// Call function with parameters
let result = add.call(&mut store, &[Value::I32(5), Value::I32(37)])?;
println!("5 + 37 = {:?}", result[0].i32()); // 42
```

The WebAssembly Component Model

Core Modules vs Components

- A core module is a single `.wasm` file with functions, memory, imports and exports
- Limited to basic types (integers and floats)
- Components are a means of extending core modules with rich type interfaces
- Uses `WIT` (WebAssembly Interface Types) as the interface language

📚 Documentation

🎥 Watch This

Benefits of The Component Model

Language Interop

- Portable across languages: same component runs everywhere
- Standardized interface (WIT): single source of truth
- Cannonical ABI for type translations: common type representation
- Language agnostic component communication: components need not know each other's implementation language

Benefits of The Component Model (ctd.)

Enhanced Safety

- Strong sandboxing: components are isolated by default and all interactions must go through defined interfaces. File and network capabilities.
- No direct memory exports: eliminates shared memory vulnerabilities
- Static analysis capabilities: resource usage and data flow can be tracked and verified and security properties checked at compile time
- Interface-based Reasoning: easier to audit component interactions

Interfaces in WIT

- An interface describes a single focus, a composable contract, through which components can interact with each other and with the hosts
- Describes the types and functions used to carry out this interaction

Worlds in WIT

- A higher level contract that describes a component’s capabilities and needs
- A world can:
- Describe the shape of the component i.e what the component exports and imports
- Defines the host environment for components i.e an environment in which a component can be installed and the functionalities that can be invoked

WIT (Wasm Interface Type)

- Used to define component model *interfaces* and *worlds*
- Not a general purpose programming language since it doesn't define behaviour, only contracts

WIT Interfaces

```
// Define types and functions for a component
interface file-manager {
    // Record type (struct)
    record file-info {
        name: string,
        size: u64,
    }

    // Variant type (enum with data)
    variant error {
        not-found,
        permission-denied(string),
    }

    // Functions
    read-file: func(path: string) -> result<list<u8>, error>;
    write-file: func(path: string, contents: list<u8>) -> result<_, error>;
}
```

WIT Worlds

```
// Define component boundaries
interface logger {
    log: func(message: string);
}

world file-system {
    // What the component provides
    export file-manager;
    export get-version: func() -> string;

    // What the component needs
    import logger;
}
```

The Component Model With Rust

Setup

- Install `cargo component`

```
cargo install cargo-component
```

- Project Setup

```
cargo component add --lib && cd add
```

- `cargo-component` will generate the necessary bindings as a module called bindings

- Update `Cargo.toml` to include the component package reference

```toml
[package.metadata.component]
package = "component:example"
```

The Component Model With Rust (ctd.)

Define Interface

```
# Define package name and version
package example:component;

# Define the interface with types and functions
interface math {
    add: func(a: u32, b: u32) -> u32;
}

# Define what the component exports/imports
world calculator {
    export math;
}
```

The Component Model With Rust (ctd.)

Implement in Rust

`cargo-component` generates a `Guest` trait that components should implement.

```rust
// add/src/lib.rs

mod bindings;

use bindings::Guest;

struct Component;

impl Guest for Component {
    fn add(x: i32, y: i32) -> i32 {
        x + y
    }
}
```

The Component Model With Rust (ctd.)

Build the component

```
cargo component build --release
```

Import from another component:

```rust
// In consumer component
use bindings::example::component::math::add;

fn calculate() {
    let result = add(5, 3);
    // Use result...
}
```

To learn more on the Component Model, Read the Docs

Component Model Debate

Supporters Say

- Needed for WASM ecosystem growth
- Solves real composition problems
- Enables better tooling
- Future-proofs module interaction

Component Model Debate (ctd.)

Critics Argue

- Too complex
- Overlaps with existing solutions
- Risk of over-standardization
- Implementation challenges
- Rustaceans are to blame?

End of Presentation

Questions?