Rust / Wasm

On Serverless & Frontend

David Wallace Croft, M.Sc.

Presented to the

Rust Miami Meetup

2025 Jan 09 Thu

WebAssembly

- WebAssembly (Wasm)
- Like Java bytecode "Write once, run anywhere"
  - But w/o intellectual property entanglements
- Wasm is one of the browser-supported languages
  - CSS, HTML, JavaScript, WebGL / GLSL
- Security sandbox
  - Limits on what it is allowed to do
  - When running in your browser

Rust

- Static typing like Java and TypeScript
  - Ada motto: "In strong typing we trust"
- High performance like C and C++
  - High performance = low electricity costs
- Memory safe like Go and Java
  - But without garbage collection spikes
- Wasm can be compiled from many languages
  - But Rust is best supported

Rust / Wasm Examples

- Java applets no longer supported in the browser
- Rewrote my Java applets using Rust / Wasm
- https://www.gamespawn.com/arcade/

- Bevy Game Engine
- Compiles Rust to Wasm for 3D in the browser
- https://bevyengine.org/examples/
- https://bevyengine.org/examples-webgpu/

Wasm Runtime

- Java jumps from browser applets to servlets
- JavaScript jumps from browser to Node.js
- Wasm jumps from browser to Wasm runtime
  - WasmEdge, Wasmer, Wasmtime, & more
- Security sandbox
  - Restrictions on transitive dependencies
- WebAssembly System Interface (WASI)
  - CLI, Clocks, Files, HTTP, Random, Sockets

Serverless

- No dedicated servers for the backend
  - Developers, not system administrators
- Server rented just while function executing
  - Charged by the millisecond
- Can have lengthy cold start times
  - 200+ milliseconds to seconds
  - Depending on language and size
  - Mitigation techniques

Fermyon Spin

- Virtual Machines to Containers to Serverless
- The next jump is to Serverless Wasm

- Fermyon Spin is Serverless Wasm
- Fast cold startups in less than 2 milliseconds
- Can run on top of Kubernetes
- Permits creating Wasms from other Wasms
- Transitive dependency security constraints
- Recently released 3.0

Single Page Application

- SPAs download a single index.html file
  - To bootstrap the JavaScript code
  - Updates Document Object Model (DOM)
- Renders the application web pages client-side
  - No download delays when switching pages
  - Everything just seems to pop instantly
- Served from a Content Delivery Network (CDN)
  - No server-side rendering = no servers
  - No servers = reduced costs

Pre-rendering & Hydration

- Web spiders search for text content in HTML
- No text content in the HTML for an SPA
  - Needs to run JavaScript to render content
- Static pre-rendering adds content to the HTML
  - Added before upload to the CDN
  - Good for bookmarked pages
  - Good for Search Engine Optimization (SEO)
- Hydration wires up JavaScript to pre-rendered

Jamstack

- JavaScript, APIs, and Markup (JAM)
  - J = SPA code in JavaScript or Wasm
  - A = Only data, not pages, to and from server
  - M = Static pre-rendering for SEO
- Static Site Generator (SSG)
  - Pre-renders HTML with content from markup
  - Markup can be Markdown or another language
- Jamstack uses SSG with data transfers via APIs

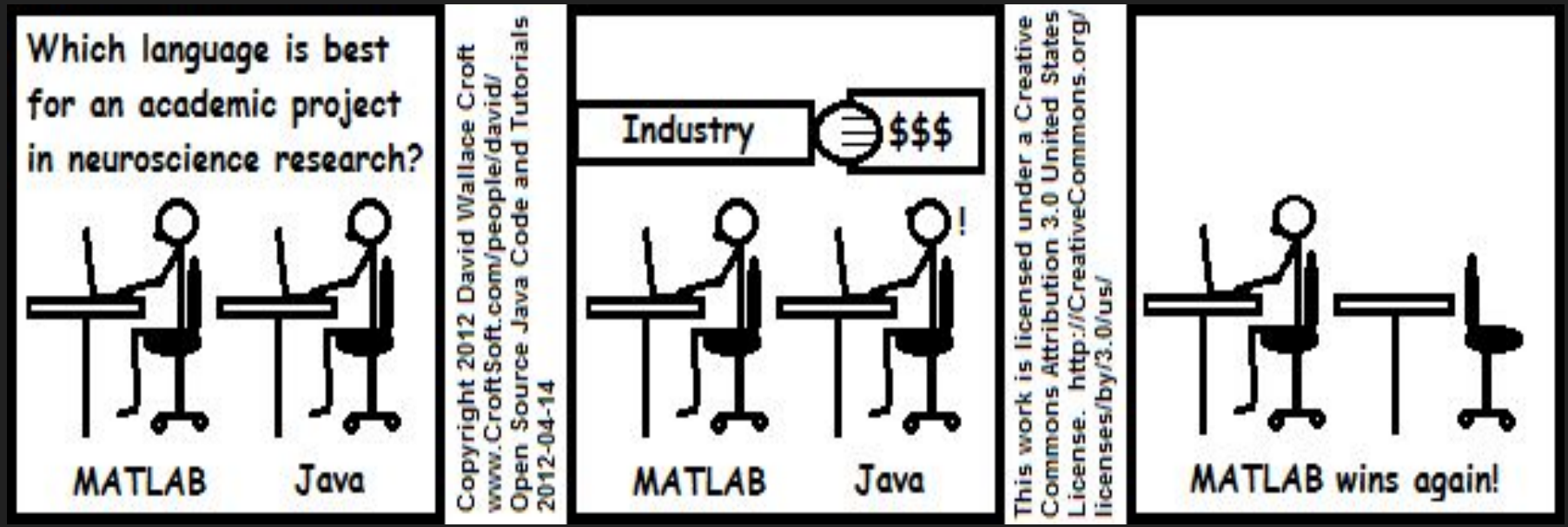
Dioxus

- SPA library for the Rust programming language
  - If you know React, you already know Dioxus
- Compiles to Wasm
  - Runs in all major browsers
  - Also desktop and mobile
- Recently released version 0.6
  - Improved support for SSG
  - Static pre-rendering and client-side hydration

Dioxus Examples

- All examples use static prerendering for SEO
- Dallas Rust
  - Mouse wheel input drives animation
  - https://www.DallasRust.org/
- Dioxus Prototype
  - Recently updated from Dioxus v0.4 to v0.6
  - Referenced tutorial not yet updated
  - https://www.persentia.com/
- Organs for Life
  - https://www.OrgansFor.Life/

Language Choice = Mobility



Rust / Wasm = Mobility

- Coding language choice limits job mobility
  - A bank with its own backend language
  - Backend vs frontend vs scripting
  - Platform-specific programming languages
- JavaScript / TypeScript (JS / TS) = Fullstack
  - Frontend, backend, and scripting
- Rust / Wasm is fullstack like JS / TS
  - Plus low level and performance applications
  - AI, Data, Embedded, Graphics, Systems

Advent of Spin

- Coding challenge hosted by Fermyon
- Uses Fermyon Spin for serverless Wasm
- Challenges teach you incrementally
- 2024 is my second year to participate
  - https://github.com/david-wallace-croft/advent-of-spin

Challenge 1

- Frontend data input form in any language
- Backend Wasm serverless from any language
- I choose Rust / Wasm for both
- Dioxus for my frontend Jamstack SPA

- Switched from RustRover back to VS Code
- My first time to use GitHub Copilot
- Used Amazon CodeWhisperer previous year

Challenge 1 Solution Demo

- https://challenge1-fbgn5xod.fermyon.app/

Solution to the Fermyon Advent of Spin 2024 Challenge 1
Both front-end and back-end use Rust compiled to WebAssembly (Wasm)

Author: David Wallace Croft, M.Sc.
Back-end: Fermyon Spin 3.0
Front-end: Dioxus 0.6
Image: Google Gemini / Imagen 3
Repository: https://github.com/david-wallace-croft/advent-of-spin/tree/main/2024

Name | First Gift | Second Gift | Third Gift | Add

Name	Items
John Doe	["Ugly Sweater", "Gingerbread House", "Stanley Cup Winter Edition"]
Jane	["puppy", "pony", "kitty"]
Billy-Bob Jones	["Apple Pie", "Gingerbread House", "Stanley Cup Winter Edition"]

Challenge 1 Solution Code

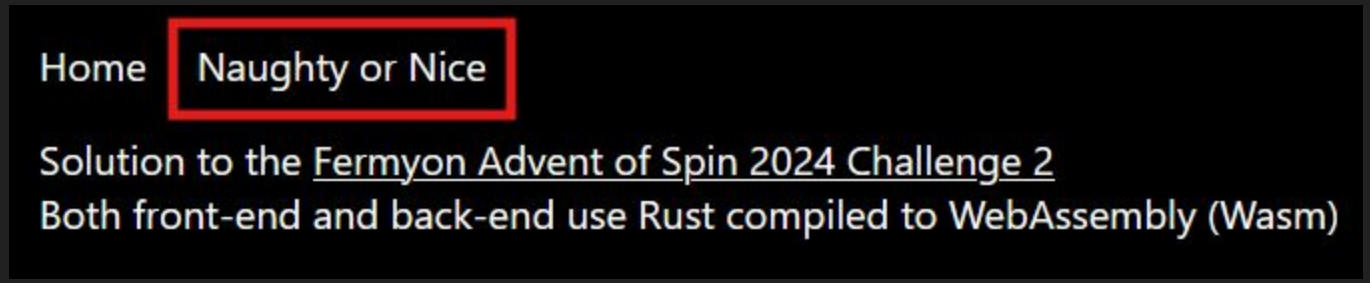
- https://github.com/david-wallace-croft/advent-of-spin/tree/main/2024/challenge1
- README.md
- spin.toml, Cargo.toml, lib.rs
- test.hurl, submit.hurl
- Dioxus.toml, Cargo.toml, main.rs
- assets/, components/, data/

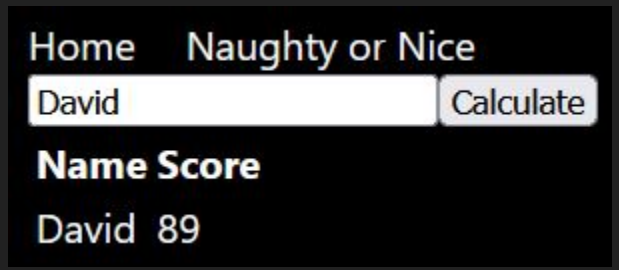
Challenge 2

- Compile Wasm from JavaScript or TypeScript
- Use the Wasm as a dependency library
- In a serverless Wasm compiled from Rust

Challenge 2 Solution Demo

- https://challenge2-xqnag9fm.fermyon.app/naughty-or-nice.html





Challenge 2 Solution Code

- https://github.com/david-wallace-croft/advent-of-spin/tree/main/2024/challenge2
- README.md
- calculator.js, component.wit, package.json
- .wit/, spin.toml, bindings/
- lib.rs

Challenge 3

- Compile Wasm from Python
- I gave up on this one after too much time on it
- componentize-py might be buggy on Windows

- But I learned a lot from the documentation
- The WebAssembly Component Model
  - https://component-model.bytecodealliance.org/

Future

- WebAssembly System Interface (WASI)
  - https://wasi.dev/
- Warg
  - https://warg.io/
- OAuth 2.0 / OpenID Connect (OIDC)
  - Authentication between Dioxus and Spin

Links

- Fermyon Spin: Serverless WebAssembly (Wasm)
  - https://www.fermyon.com/spin
- Dioxus: Rust-to-Wasm user interface library
  - https://dioxuslabs.com/
- Rust: high performance memory safe language
  - https://www.rust-lang.org/
- WebAssembly: bytecode for browser and server
  - https://webassembly.org/
- Jamstack: JavaScript, APIs, Markup architecture
  - https://jamstack.org/

Presenter

- David Wallace Croft, M.Sc.
  - https://www.CroftSoft.com/people/david/
- Organizer of the Dallas Rust User Meetup
  - https://www.DallasRust.org/
- Open source Rust projects
  - Animated interactive games and simulations that run in the browser using WebAssembly (Wasm)
  - Single page applications (SPAs) with static pre-rendering and client-side hydration using Dioxus
  - Serverless functions using Amazon Web Services (AWS) Lambda and Fermyon Spin
  - https://www.CroftSoft.com/people/david/research/rust-wasm/

Licenses

- Slides and code are © 2025 CroftSoft Inc
- This slide presentation is available under the terms of the Creative Commons Attribution 4.0 International License https://creativecommons.org/licenses/by/4.0/
- The code is available under the terms of the open source MIT License https://opensource.org/license/mit/