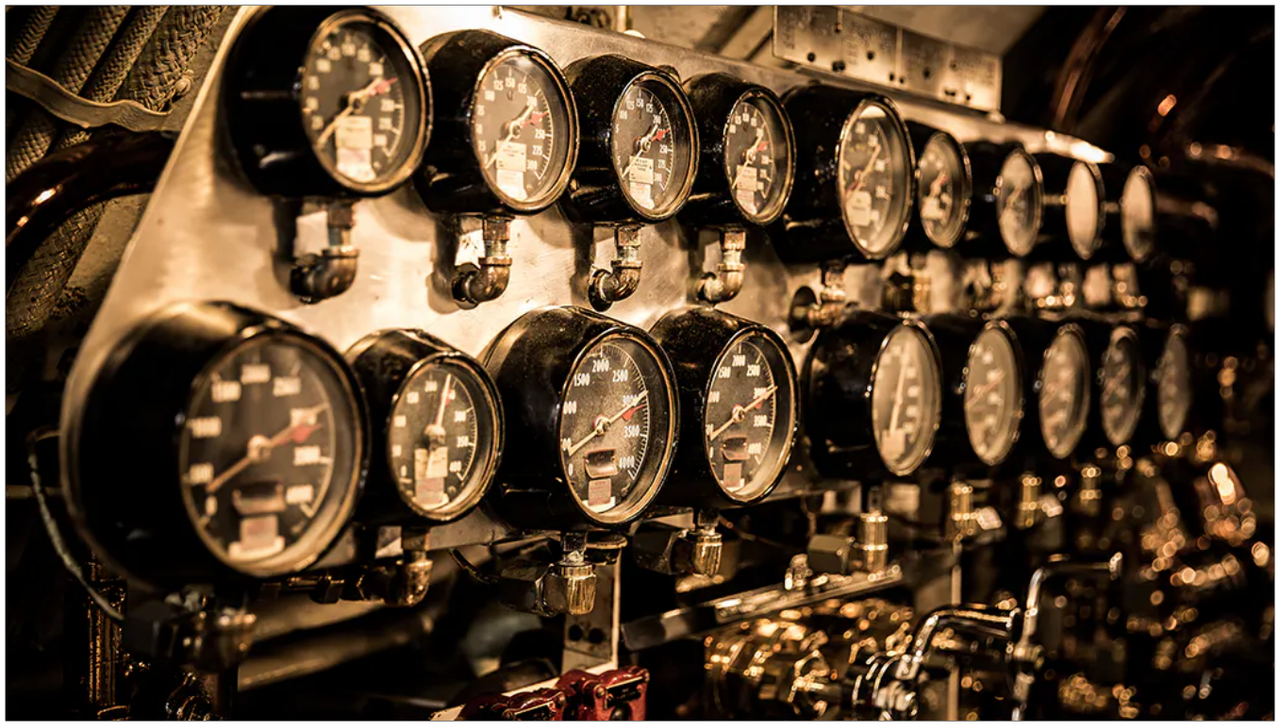
Web Developer's Guide to Rust

KCDC 2024

Shawn Strickland

introductions

- Father
- Musician
- Fan of Documentaries
- Fan of Documentation
- Not Boring
- Currently a Technical Lead @ Federal Reserve
  - *(Check out the Money Museum)*



This is not an in-depth Rust talk, more of a general overview.
It’s designed to whet your appetite for using Rust in your next web project or refactor.
Also, all links and slides will be provided in a QR at the end.

Getting to Know It (Rust)

introductions

- Very C-Like language allowing for easy pickup
- “Forces” you to write better code, known to shorten the gap between junior and senior devs
- Low-level language giving performance gains
  - No interpreter, VM, JIT Compiler, etc.
  - Compiles to executable for each machine (CPU arch)
    - Don’t even need Rust installed on the machine to run a program written in it!
      - Via “ahead-of-time” compilation

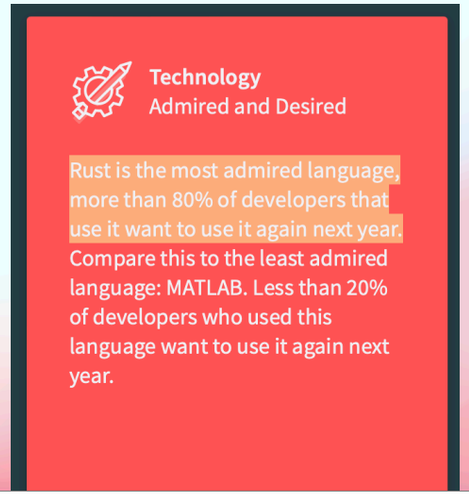
Why YAPL?

(Yet another programming language)

- Statically-typed
  - See: onset of Typescript, typing hints in Python, etc.
  - No more “cannot read property foo of null” errors (you catch those at compile time instead)
    - A really helpful compiler
- Must be intentional when you want to be mutative
- Lots of concurrency and parallel options available in Rust
- The White House likes its Memory Safety
- It has it’s own Package Manager

Interesting Facts

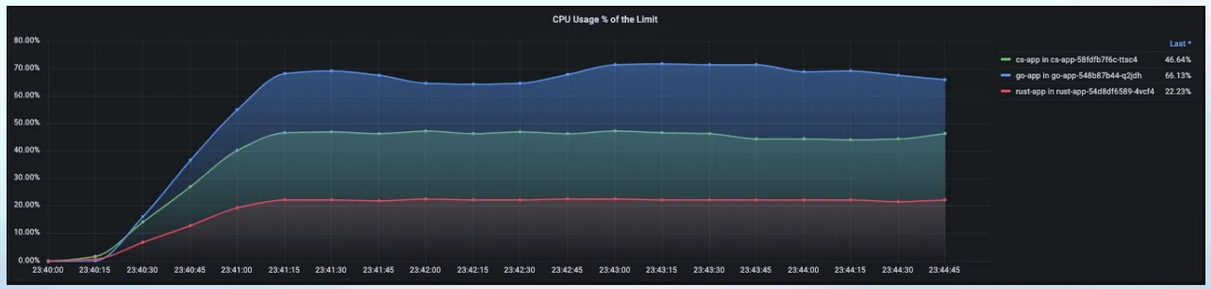
- "The most love programming language [YoY]"
  (StackOverflow developer survey)
  - 4x Champ
- Variables are immutable by default
- Memory safety w/o a garbage collector
  - Great resource: Understanding Rust's Borrow Checker
- Macros
  - `println!()`



Performance

CPU Usage

- POST 1M requests, 50 concurrent connections:

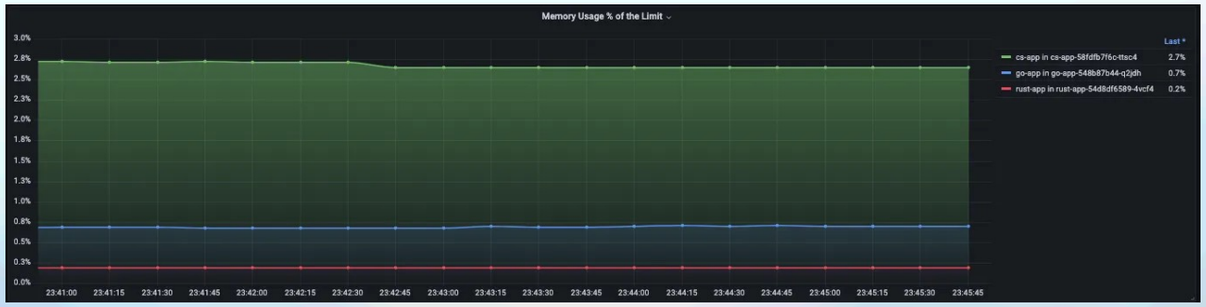


Lowest CPU usage vs go (blue) and c# (green)
Go peaks here at 70%, Rust holds steady ay 20%

Performance

Memory Usage

- POST 1M requests, 50 concurrent connections:

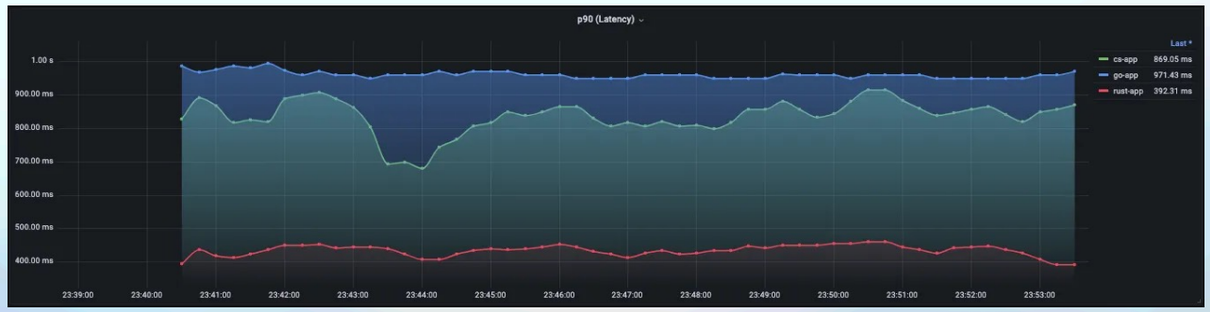


Lowest memory usage vs c# (green) and go (blue)

Performance

Latency (90th Percentile)

- POST 1M requests, 50 concurrent connections:



Lowest latency vs c# (green) and go (blue)
Go hovers at 1s latency while rust does it un 400ms

Rust + APIs

Rocket

- JSON serialization
- Forms
- Simple config

```rust
#[derive(Serialize, Deserialize)]
struct Message<'r> {
    contents: &'r str,
}

#[put("/<id>", data = "<msg>")]
fn update(db: &Db, id: Id, msg: Json<Message<'_>>) -> Value {
    if db.contains_key(&id) {
        db.insert(id, msg.contents);
        json!({ "status": "ok" })
    } else {
        json!({ "status": "error" })
    }
}
```

Rocket has first-class support for JSON.
Simply derive Deserialize or Serialize to receive or return JSON, respectively.
#[attribute] in Rust
- crate-level attribute
- function and module-level attribute
- conditional compilation like (cfg(target_os=“linux”))

Rust + APIs

Actix

- JSON
  - Responders
  - Extractors
- Forms
  - JSON and form data is deserialized into a struct
- Simple config
- FAST

```rust
use actix_web::{get, web, App, HttpServer, Responder};

#[get("/")]
async fn index() -> impl Responder {
    "Hello, World!"
}

#[get("/{name}")]
async fn hello(name: web::Path<String>) -> impl Responder {
    format!("Hello {}!", &name)
}

#[actix_web::main]
async fn main() -> std::io::Result<()> {
    HttpServer::new(|| App::new().service(index).service(hello))
        .bind(("127.0.0.1", 8080))?
        .run()
        .await
}
```

Actix also supports the same sort of features in it’s own way.
Serialize/Deserialize are “responders” and “extractors”
Form data is handled in similar way to Rocket crate.
Super fast, of a benchmarked top-10 web frameworks, it’s #7. Interestingly, of those 10, 5 are Rust.

Rust + Serverless

Functions go brrr

- Harness the power of cheap executions with Rust
  - In general, can run functions with lower execution times (cheaper) and lower memory thresholds (cheaper)
- `cargo lambda {LAMBDA_NAME}`
  - Extremely similar to dotnet lambda (c#), node-lambda (npm), etc.

More bang for you buck with that 1M lambda invocation free tier.

WebAssembly

a.k.a WASM (WebASseMbly)

- *We do get back to Rust...*
- Webpages go zoom (since we're talking "fast" programming languages)
  - Multithreaded (faster than JS's)
- Powers computationally-intensive web apps like **Microsoft Office Online, Figma, Abelton, Google Earth, AutoCAD**, etc.
- Interested? Check out "the book": The Art of WebAssembly *(https://wasmbook.com)*

Detour for WebAssembly here, because it jams so well with Rust.
Think of it as a super fast web language that all modern browsers can support.
Really nice part of it, is that many languages can compile down into WebAssembly for demanding web applications that just aren't cutting it with Javascript alone.

WebAssembly

How its structured

- S-Expressions
  - Symbolic Expression
  - think tree-based structure, branching, etc.
- Linear Instruction List
  - Essentially, dev keeps track of what's on the stack...

S-Expressions come from LISP, if any fans are in here for that.
But the rest of us can just think of that as “the normal if/else branching” we’re used to.

WebAssembly

"wat" it looks like

WASM

```
(module
  (func $add (param $lhs i32) (param $rhs i32) (result i32)
    local.get $lhs
    local.get $rhs
    i32.add)
  (export "add" (func $add))
)
```

i32 i64 f32 f64

Numbers in javascript are always 64-bit floating point, so a real enhancement comes in here with number-crunching.

WebAssembly

Calling WASM within Javascript

```
WebAssembly.instantiateStreaming(fetch("myModule.wasm"), importObject).then(
  (obj) => {
    // Call an exported function:
    obj.instance.exports.exported_func();

    // or access the buffer contents of an exported memory:
    const dv = new DataView(obj.instance.exports.memory.buffer);

    // or access the elements of an exported table:
    const table = obj.instance.exports.table;
    console.log(table.get(0)());
  },
);
```

Rust + WASM

Rust all in the frontend

- Harness the performance power of WASM without needing to BYO WASM Text
- Generate WASM from Rust which can be called and ran completely in the browser
- Demo Rust Game of Life

Rust + WASM + HTMX

...oh my

- Utilize modern browser features directly in HTML (no .js necessary)
  - Trigger events directly from actions in html elements
  - Neat: polling

```
<div hx-get="/news" hx-trigger="every 2s"></div>
```

- Respond with HTML, not JSON
- Progressive enhancement, accessibility, IE11+, etc.
- HTMX + Service Workers + WebAssembly + Rust

Rust + AWS

- Rust AWS SDK
- DynamoDB Crate example
- Rust Futures (more on await)

```rust
let request = client
    .put_item()
    .table_name(table)
    .item("username", user_av)
    .item("account_type", type_av)
    .item("age", age_av)
    .item("first_name", first_av)
    .item("last_name", last_av);

println!("Executing request [{request:?}] to add item...");

let resp = request.send().await?;

let attributes = resp.attributes().unwrap();

let username = attributes.get("username").cloned();
let first_name = attributes.get("first_name").cloned();
let last_name = attributes.get("last_name").cloned();
let age = attributes.get("age").cloned();
let p_type = attributes.get("p_type").cloned();

println!(
    "Added user {:?}, {:?} {:?}, age {:?} as {:?} user",
    username, first_name, last_name, age, p_type
);
```

“?” Is error propagation for the .await, just some syntactic sugar that avoid a few lines of handling errors traditionally.

Rust + Azure

- azure-sdk-for-rust
  - An *unofficial* SDK
- Azure Crates

```rust
// Using the prelude module of the Cosmos crate makes easier to use the Rust Azure SDK for Cosmo
use azure_data_cosmos::prelude::*;
use azure_core::Context;
use serde::{Deserialize, Serialize};
```

Rust + Azure

Continued

```rust
// Insert 10 documents
println!("Inserting 10 documents...");
for i in 0..10 {
    // define the document.
    let document_to_insert = MySampleStruct {
        id: format!("unique_id{}", i),
        string: "Something here".to_owned(),
        number: i * 100, // this is the partition key
    };

    // insert it
    collection
        .create_document(document_to_insert)
        .is_upsert(true)
        .await?;
}
```

Rust + Ground

For post-clouders (or pre-clouders)

- Diesel Crate
  - Highly-recommended ORM by the Rust community
- postgres Crate
  - Synchronous PostgreSQL client
- tokio_postgres Crate
  - Asynchronous PostgreSQL client

href

Links from the previous slides

Thank You!

Feedback is welcomed

Titanium Sponsors

Progress
ngrok
DON'T PAN!C LABS

Platinum Sponsors

Temporal
SUSE
Pulumi
snowflake
Procure SQL — DATA ARCHITECTURE AS A SERVICE
Particular Software
NAIC — NATIONAL ASSOCIATION OF INSURANCE COMMISSIONERS
NIPR — NATIONAL INSURANCE PRODUCER REGISTRY
LaunchDarkly
H&R BLOCK
ORACLE Database
LUNAVI
docker
pangea
CloudBees.
squid

Gold Sponsors

touchnet — A Global Payments Company
TEKsystems — Own change
SHAMROCK TRADING CORPORATION
QUEST ANALYTICS
EVENT STORE
BUILDERTREND
ADAPTIVE SOLUTIONS GROUP
honeycomb.io
accumatch CONSULTING
AMUSEMENT CONNECT
AVITURE
Federal Reserve Bank of Kansas City
VML
TURNBERRY SOLUTIONS
sonatype
yugabyteDB
LEAN TECHNIQUES

Speaker Dinner

Common Room

Friends of KCDC

outsystems
IowaComputer GURUS
Google Cloud
Gradle
JETBRAINS