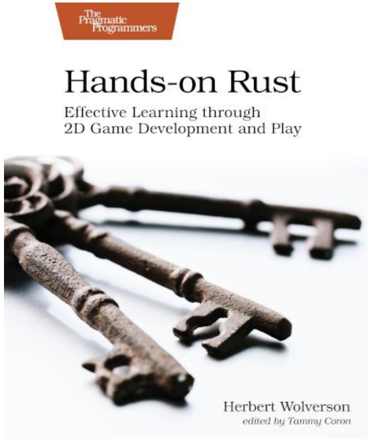
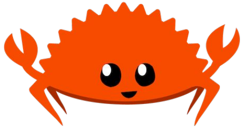
Introducing Rust into Your Company Ecosystem

herbert.wolverson@ardanlabs.com

About Herbert Wolverson

- Ardan Labs Rust Trainer & Consultant
- Author of *Hands-on Rust* and *Rust Brain Teasers*
- Author of the *Rust Roguelike Tutorial*
- Lead developer, *LibreQoS*, *bracket-lib*.
- Contributor to many open source projects.





All code used in this presentation is available here:

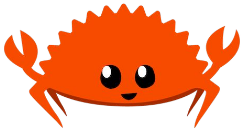
https://github.com/thebracket/ArdanLabs_RustInYourEnterprise

What We’re Going to Cover

- The benefits of Rust.
- Where does Rust fit in?
- How to Try Rust in your Enterprise.
- Rust in a Service-Oriented Architecture
- Rust in a Message-Oriented Architecture
- Rust inside Existing Services
- Transforming Legacy Code
- Q&A

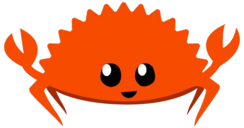
The Benefits of Rust (Part 1)

- Safety
  - Memory Safety: No buffer overflows, use-after-free.
  - Safe Concurrency: No data races.
  - Safe by Default - with “opt in” unsafe features when needed.
- Performance (Speed)
  - Compiles to native code, optimized by LLVM.
  - Techempower.com benchmarks show 5 of the top 10 performing web servers are written in Rust.
- Performance (Latency)
  - No Garbage Collection - but you can opt in to reference counting when you need it.
  - *Predictable* latency - a well architected Rust program will retain very consistent latency.
- Control
  - Opt-in to controlling allocations, threads, stacks and buffers *if* you need it.
  - Default settings work well.



The Benefits of Rust (part 2)

- Developers love it! Rust keeps winning Stack Overflow’s “most loved language” prize.
- Rust is very expressive: you can do a lot with a small amount of code.
- Rust has a large ecosystem, less reinventing the wheel.
- Together this amounts to: Rust is a very productive language ecosystem.



Where does Rust fit in?

- Rust really shines where you need raw performance.
- Rust “plays nicely” with other systems.
- If you have a CPU-bound problem, Rust is a great fit.
- If you have an I/O bound problem (waiting on other resources), Rust can offer some improvement - but probably won’t solve the problem completely.
- Fearless Concurrency - and the absence of data races - makes Rust a great choice for highly concurrent processing.
- If you need predictable latency, Rust is a *great* choice.
- For high-security, customer-facing applications, Rust can provide an improvement in safety.

Find an Itch - and Scratch it with Rust

Every system reaches a point in which one or two components just don’t quite meet expectations.

Find a niche that needs Rust.

Spend some time “scratching the itch” - solving your problem.

Test it.

Talk about your productivity gains and how Rust helped.

Avoid Forklifting!

Corollary: It’s far safer to start gradually.

Forklift-replacing your entire project is dangerous!

Rust in a Service-Oriented Architecture (SoA)

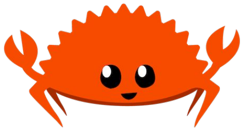
“Hello World” Webserver - in 16 lines of code

Cargo.toml

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

[dependencies]
axum = "0.6.18"
tokio = { version = "1.28.0", features = ["full"] }
```

main.rs

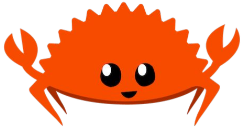
```rust
use axum::{routing::get, Router};
use std::net::SocketAddr;

#[tokio::main]
▶ Run | Debug
async fn main() {
    let app: Router = Router::new().route("/", get(handler: say_hello_text));
    let addr: SocketAddr = SocketAddr::from(([127, 0, 0, 1], 3000));
    axum::Server::bind(&addr)
        .serve(app.into_make_service())
        .await
        .unwrap();
}

async fn say_hello_text() -> &'static str {
    "Hello, world!"
}
```



Axum is fast - consistently in the top 10 on web-server performance benchmarks.

Add JSON Handling in 10 Lines of Code

```rust
use axum::{routing::get, Router, Json};
use serde::Serialize;
use std::net::SocketAddr;

#[tokio::main]
▶ Run | Debug
async fn main() {
    let app: Router = Router::new().route("/", get(handler: say_hello_json));
    let addr: SocketAddr = SocketAddr::from(([127, 0, 0, 1], 3000));
    axum::Server::bind(&addr)
        .serve(app.into_make_service())
        .await
        .unwrap();
}

#[derive(Serialize)]
1 implementation
struct HelloJson {
    message: String,
}

async fn say_hello_json() -> Json<HelloJson> {
    Json(HelloJson {
        message: "Hello, World!".to_string(),
    })
}
```



Just add a Database - Part 1, Migrations

Install the SQLX tool with “cargo install sqlx-cli”

Set an environment variable (or make a .env file):
DATABASE_URL="sqlite:C:/Users/Herbert/Rust/ArdanLabs_RustInYourEnterprise/hellodb_webservice/hello_db.db"

Run “sqlx migrate add initial”

Write the migration

```sql
CREATE TABLE IF NOT EXISTS messages
(
    id          INTEGER PRIMARY KEY NOT NULL,
    message     TEXT                NOT NULL
);

INSERT INTO messages (id, message) VALUES (1, 'Hello World!');
INSERT INTO messages (id, message) VALUES (2, 'Hello Galaxy!');
INSERT INTO messages (id, message) VALUES (3, 'Hello Universe!');
```

Add Database Support to Your Web Service

```rust
use axum::{routing::get, Router, Json, Extension};
use serde::{Serialize, Deserialize};
use std::net::SocketAddr;

#[tokio::main]
async fn main() {
    let pool = sqlx::SqlitePool::connect("sqlite:hello_db.db").await.unwrap();
    sqlx::migrate!("./migrations")
        .run(&pool)
        .await
        .expect("Unable to migrate database");

    let app = Router::new()
        .route("/", get(say_hello_json))
        .layer(Extension(pool));
    let addr = SocketAddr::from(([127, 0, 0, 1], 3000));
    axum::Server::bind(&addr)
        .serve(app.into_make_service())
        .await
        .unwrap();
}

#[derive(Serialize, Deserialize)]
struct HelloJson {
    id: i64,
    message: String,
}

async fn say_hello_json(Extension(pool): Extension<sqlx::SqlitePool>) -> Json<Vec<HelloJson>> {
    let result = sqlx::query_as!(HelloJson, "SELECT * FROM messages")
        .fetch_all(&pool)
        .await
        .unwrap();
    Json(result)
}
```


0:
id: 1
message: "Hello World!"
1:
id: 2
message: "Hello Galaxy!"
2:
id: 3
message: "Hello Universe!"

How fast is this tiny demo?

```
use serde::Deserialize;
use std::time::Instant;

#[derive(Deserialize, Debug)]
2 implementations
struct HelloJson {
    id: i64,
    message: String,
}

#[tokio::main]
▶ Run | Debug
async fn main() {
    const NUM_REQUESTS: usize = 10_1000;
    let mut results: Vec<u128> = vec![0; NUM_REQUESTS];

    let client: Client = reqwest::Client::new();
    for n in 0..NUM_REQUESTS {
        let now: Instant = Instant::now();
        let messages: Vec<HelloJson> = client Client
            .get(url: "http://localhost:3000/") RequestBuilder
            .send() impl Future<Output = Result<…>>
            .await Result<Response, Error>
            .unwrap() Response
            .json() impl Future<Output = Result<…>>
            .await Result<Vec<HelloJson>, Error>
            .unwrap();
        results[n] = now.elapsed().as_micros();
    }

    // Ignore the first result, it includes warm-up time
    results.remove(0);

    println!("Worst time: {} µs", results.iter().max().unwrap());
    println!("Best time: {} µs", results.iter().min().unwrap());
    println!("Average time: {} µs", results.iter().sum::<u128>() / NUM_REQUESTS as u128);
} fn main
```

Our tiny test application (35 lines of code) produces the following results on my test laptop:

Without SQLite: Average time: 53 µs

With SQLite: 99 µs

Just serializing JSON: 132 nanoseconds

We've not performed *any* optimization, or used concurrency!

How about direct TCP connections? (1 of 2)

Example boilerplate (use and structures) trimmed.

Command is an enumeration.

Each TCP connection is moved to its own Tokio worker (green threading).

The request is de-serialized, processed, and the results serialized and sent back.

```rust
#[tokio::main]
async fn main() {
    let listener = TcpListener::bind("127.0.0.1:3000")
        .await
        .expect("Unable to bind port");

    loop {
        let (mut socket, _address) = listener
            .accept()
            .await
            .expect("Unable to accept connection");
        spawn(async move {
            loop {
                let size = socket.read_u64().await.expect("Unable to read size");
                let mut buffer = vec![0; size as usize];
                socket.read_exact(&mut buffer).await.expect("Unable to read");
                let command: Command =
                    bincode::deserialize(&buffer).expect("Unable to deserialize");

                if let Command::SayHello = command {
                    let result = Hello {
                        message: "Hello".to_string(),
                    };
                    let result = bincode::serialize(&result).expect("Unable to serialize");
                    socket.write_u64(result.len() as u64).await.expect("Unable to write size");
                    socket.write_all(&result).await.expect("Unable to send");
                }
            }
        });
    }
}
```

Direct TCP Client (2/2)

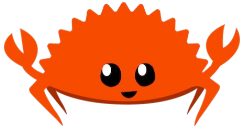
The client opens a socket connection, and sends 10,000 requests.

Serialization uses *bincode* from Mozilla, but you can easily use ProtoBufs, gRPC or other popular mechanisms.

Average time: 29 µs

No optimizations performed.

```rust
#[tokio::main]
async fn main() {
    const NUM_REQUESTS: usize = 100_000;
    let mut results = vec![0; NUM_REQUESTS];
    let command = Command::SayHello;
    let command = bincode::serialize(&command).expect("Unable to serialize");
    let mut socket = TcpStream::connect("127.0.0.1:3000")
        .await
        .expect("Unable to connect");

    for n in 0..NUM_REQUESTS {
        let now = Instant::now();
        socket
            .write_u64(command.len() as u64)
            .await
            .expect("Unable to write size");
        socket.write_all(&command).await.expect("Unable to send");
        let receive_size = socket.read_u64().await.expect("Unable to read size");
        let mut buffer = vec![0; receive_size as usize];
        socket.read_exact(&mut buffer).await.expect("Unable to read");
        let message: Hello = bincode::deserialize(&buffer).expect("Unable to deserialize");
        //println!("{message:?}");
        results[n] = now.elapsed().as_micros();
    }

    println!("Average time: {} µs", results.iter().sum::<u128>() / NUM_REQUESTS as u128);
}
```



Rust is Ready for the Service-Oriented Architecture

Rust lets you become productive fast in an SoA:

- Build an HTTP server in only 16 lines of Rust.
  - Add fast JSON support with 10 more lines of Rust.
  - 9 more lines of Rust adds SQL support, with migrations, compile-time SQL testing and dependency injected connection pools.
- Build a TCP-based RPC server in only 49 lines of Rust.

A few things to notice:

- We didn't do *any* manual management of memory, threads or resources.
- We didn't optimize, but it's *fast* by default:
  - Serialize a JSON message in 132 nanoseconds.
  - Round-trip HTTP with serialization in 53 µs.
  - Round-trip HTTP with a database query in 99 µs.
  - Streaming TCP service with a 29 µs response time.
- We didn't pay attention to safety, but Rust eliminates many common vulnerabilities automatically.

Rust in a Message-Oriented Architecture

Integration is available for most message queue architectures:

- Apache Kafka
- ZeroMQ
- RabbitMQ
- And many more (via crates.io)

The basic code is the same as a network service: you asynchronously receive messages and spawn workers to respond.

A ZeroMQ Client that Replies to Messages

13 lines of Rust is sufficient to reply to ZeroMQ messages.

You probably want to add some business logic!

```rust
use async_zmq::Result;
use futures::StreamExt;
use std::ops::Deref;

#[tokio::main]
async fn main() -> Result<()> {
    let mut zmq = async_zmq::reply("tcp://127.0.0.1:5555")?.bind()?;

    while let Some(msg) = zmq.next().await {
        for it in msg.unwrap().deref() {
            println!("message: {:?}", it.as_str());
        }
        zmq.send(vec!["Response for You"]).await?;
    }

    Ok(())
}
```

Integrate Rust into Existing Services

Speed up your Python Scripts with Rust/PyO3

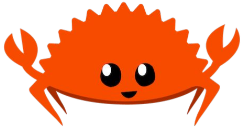
```rust
use pyo3::prelude::*;

#[pymodule]
#[pyo3(name = "mypylib")]
pub fn mypylib(_py: Python, m: &PyModule) -> PyResult<()> {
    m.add_wrapped(wrap_pyfunction!(say_hello))?;
    Ok(())
}

#[pyfunction]
pub fn say_hello(_py: Python) -> PyResult<String> {
    Ok("Hello, world!".to_string())
}
```

Take some simple Rust

Add PyO3 declarations

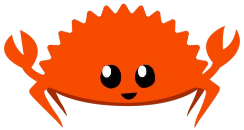
```python
#!/bin/python3
import mypylib
print(mypylib.say_hello())
```

Run it in Python



Call Rust from Go - Step 1, Write Some Rust

```rust
use std::ffi::CStr;

/// # Safety
/// Use a valid C-String!
#[no_mangle]
pub unsafe extern "C" fn hello(name: *const libc::c_char) {
    let name_cstr = unsafe { CStr::from_ptr(name) };
    let name = name_cstr.to_str().unwrap();
    println!("Hello {name}");
}
```



Call Rust from Go - Step 2, Build a Static LIbrary

Adjust Cargo.toml to produce a static library.

Either write a C header file, or use the "cbindgen" tool to do it for you.

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

[lib]
crate-type = ["staticlib"]

[dependencies]
libc = "0.2.2"
```

```c
// Normally you would want to use `cbindgen` to make this file for you.
void hello(char *name);
```

Call Rust from Go - Step 3, Write some Go!

You can use Rust libraries just like any other C library in Go. This example links statically.

For a faster, but more complicated version, see the GoRust project - https://words.filippo.io/rustgo/

```go
package main

/*
#cgo LDFLAGS: ./../target/debug/libsay_hello.a -ldl
#include "./lib/say_hello.h"
*/
regenerate cgo definitions
import "C"

import "fmt"

func main() {
    fmt.Println("Hello from GoLang!")
    C.hello(C.CString("from Rust!"))
}
```

Go to Rust Performance

Using the simplest possible Go benchmark to time function calls:

```go
func main() {
	start := time.Now()
	fmt.Println("Hello from GoLang!")
	duration := time.Since(start)
	fmt.Println(duration)
	start2 := time.Now()
	C.hello(C.CString("from Rust!"))
	duration2 := time.Since(start2)
	fmt.Println(duration2)
}
```

Running in an underpowered Linux VM on Hyper-V!

Hello from GoLang!
23.838µs

Hello from Rust!
27.563µs

3.7µs performance penalty for crossing language boundaries. Faster than a network boundary, slower than a native call.

Foreign Function Interface Pitfalls

- Marshalling Performance
  - Whenever you create a Python object in Rust, there's a short delay while the object is turned into Python format.
  - Calling out via CGo imposes overhead.
- Mitigate Marshalling Delays
  - Make sure that your task needs Rust!
  - Avoid lots of small calls into Rust.
  - Instead, try to wrap as much functionality into a single call as you.
  - This allows you to amortize the delay, and still benefit from the overall speed improvement.
- C Interfaces (C FFI Only - Not PyO3)
  - C interfaces aren't as rich as native Rust interfaces.
  - Conversion to/from C types can be tricky.

Transforming Legacy Code

Transforming Legacy Code to Rust

- Particularly useful for legacy C code.
- Can work with any language that supports external C linkage.
- You can also use a similar pattern for web services: build tests, replace remote procedure calls one at a time and tweak until all of the tests work.

The Process:

Wrap your legacy library in Rust

Write Rust unit tests that call the original library, testing every function you wish to port.

Create Rust functionality, one function at a time. Write a second set of tests to ensure that the Rust function produces the same/desired output.

Benefit from Rust safety and maintainability going forwards.

Example Legacy Migration (1 of 3)

Step 1: Add “cc” to Cargo.toml as build dependency:

```
[package]
name = "crust"
version = "0.1.0"
edition = "2021"

[dependencies]

[build-dependencies]
cc = "1"
```

Step 2: Add a “build.rs” script to your project:

```
fn main() {
    cc::Build::new()
        .file("src/crust.c")
        .compile("crust");
}
```

Example Legacy Migration (2 of 3)

Step 3: Here's the C file!

```c
// A simple function that doubles a number
int double_it(int x) {
    return x * 2;
}
```

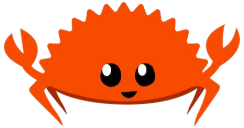
Step 4: Import the C and test it

```rust
extern "C" {
    fn double_it(x: i32) -> i32;
}

#[cfg(test)]
mod test {
    use super::double_it;

    #[test]
    fn test_double_it() {
        assert_eq!(unsafe { double_it(2) }, 4);
    }
}
```

Example Legacy Migration (3 of 3)

- Create a module for the Rust version to avoid namespace collisions.
- Port functions to Rust.
- Write unit tests to show that the new function gives the same result as the old function.
- When you are ready - you can stop importing the C function, and promote the Rust functions to the exported namespace.

```rust
extern "C" {
    fn double_it(x: i32) -> i32;
}

mod rust {
    pub fn double_it(x: i32) -> i32 {
        x * 2
    }
}

#[cfg(test)]
▶ Run Tests | Debug
mod test {
    use super::{double_it, rust};

    #[test]
    ▶ Run Test | Debug
    fn test_double_it() {
        assert_eq!(unsafe { double_it(2) }, 4);
    }

    #[test]
    ▶ Run Test | Debug
    fn test_c_rust() {
        assert_eq!(unsafe { double_it(2) }, rust::double_it(2));
    }
}
```



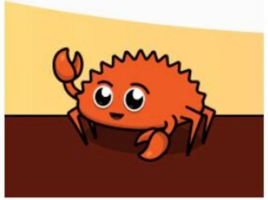
Ultimate Rust: Foundations

5 Classes

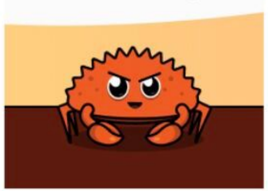
4-Hour Deep Dives

Choose Your Own Training Path!

CLASS
01
Getting Started with Rust



CLASS
02
Fearless System Thread Concurrency



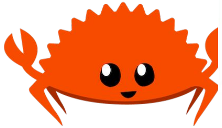
CLASS
03
Async/Await Concurrency

CLASS
04
Managing Memory & Resources

CLASS
05
Building Network Services

The More You Buy The More You Save

	Per Session	Regular Price	Early Bird
1 Session	$149	$149	$119
2 Sessions	$134	$268	$215
3 Sessions	$121	$362	$290
4 Sessions	$109	$434	$348
5 Sessions	$98	$489	$391



Questions?

All code used in this presentation is available here:

https://github.com/thebracket/ArdanLabs_RustInYourEnterprise