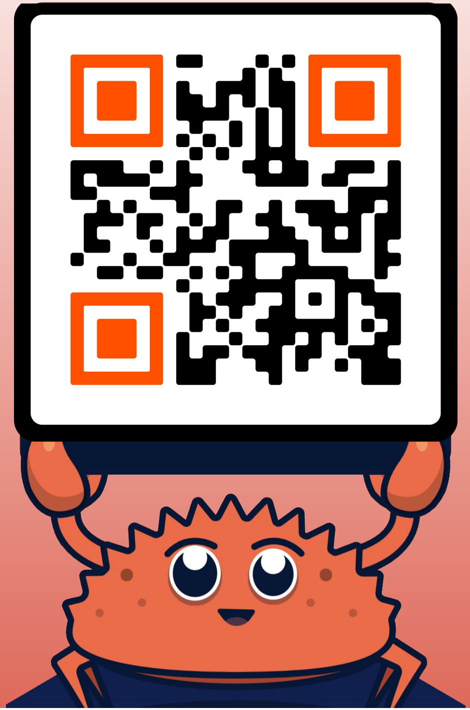
The Future of Tech Training has Arrived



Use our special code for conference attendees & receive $100 OFF!

Code: **HYBRIDGC23**

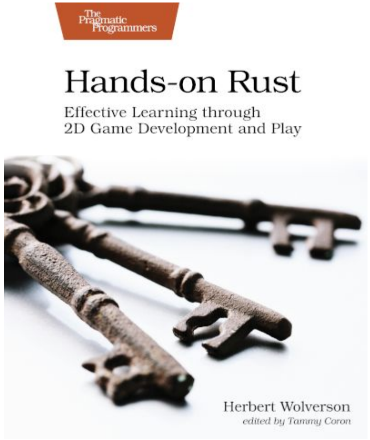
Rust Web Service Deep Dive

CRUD with SQLX + Axum

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.



Web Service Layers

We're going to build a full stack in an hour!

Database Layer	SQLite (in memory)
Data Model	SQLx
REST API	Axum
Web View	JavaScript + HTML
Deploy	Docker

Part 1: The Database (SQLite)

Project Setup

1. Create a new Rust project with `cargo init webinar_axum`
2. Install SQLX-CLI with `cargo install sqlx-cli`
3. Create a file named .env
    a. It contains `DATABASE_URL="sqlite::memory:"`
4. Create a migration with `sqlx migration add initial`

This sets up an in-memory database, and creates a file named `(timestamp)initial.sql` - ready for your database schema.

Populate the Migration File

```sql
CREATE TABLE books (
    id INTEGER PRIMARY KEY AUTOINCREMENT,
    title TEXT,
    author TEXT
);

INSERT INTO books (title, author) VALUES ('Hands-on Rust', 'Wolverson, Herbert');
INSERT INTO books (title, author) VALUES ('Rust Brain Teasers', 'Wolverson, Herbert');
```

Part 2: The Data Model (SQLX)

Add Dependencies

Add dependencies to Cargo.toml:

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

[dependencies]
tokio = { version = "1.32.0", features = ["full"] }
anyhow = "1.0.75"
dotenv = "0.15.0"
serde = { version = "1.0.188", features = ["derive"] }
sqlx = { version = "0.7.2", features = ["runtime-tokio", "sqlite"] }
axum = "0.6.20"

[dev-dependencies]
axum-test-helper = "0.3.0"
```

Tokio provides the async runtime.

Anyhow for simple error handling.

Dotenv for reading ".env" files.

Serde for Serialization/Deserialization.

SQLX for the Database.

Axum for the REST API (we'll need it later).

Axum-test-helper for convenient unit tests.

Create a skeleton main.rs

Setup an async Tokio main function:

```rust
mod db;
use crate::db::init_db;
use anyhow::Result;

#[tokio::main]
async fn main() -> Result<()> {
     // Load environment variables from .env if available
     dotenv::dotenv().ok();

     // Initialize the database and obtain a connection pool
     let connection_pool = init_db().await?;

     Ok(())
}
```

Create the db.rs module file and Book Structure

Create a new file, `db.rs` in the `src` directory.

This will contain the database API.

Let's start by defining a Book structure, representing a book from the database.

Deriving `Serialize` and `Deserialize` connects the structure to Serde — allowing for seamless transformation to/from JSON and other formats.

`FromRow` is an SQLX helper that makes it easy to SELECT data into a structure.

```rust
#[derive(Debug, Serialize, Deserialize, FromRow)]
pub struct Book {
    /// The book's primary key ID
    pub id: i32,
    /// The book's title
    pub title: String,
    /// The book's author (surname, lastname - not enforced)
    pub author: String,
}
```

Create the connection pool

Provides initialization for both tests and the main program.

Retrieves the db URL from environment.

Creates a connection pool.

Runs all SQLX migrations.

Since we’re using in-memory - it will run each time.

```rust
pub async fn init_db() -> Result<SqlitePool> {
    let database_url : String = std::env::var( key: "DATABASE_URL")?;
    let connection_pool : Pool<Sqlite> = SqlitePool::connect(&database_url).await?;
    sqlx::migrate!().run( migrator: &connection_pool).await?;
    Ok(connection_pool)
}
```

Building your Database API: Reading Data

SQLX's FromRow derivation allows you to use query_as to select data straight into a Book structure.

Data-binding prevents SQL Injection attacks.

```
pub async fn all_books(connection_pool: &SqlitePool) -> Result<Vec<Book>> {
    Ok(
        sqlx::query_as::<_, Book>( sql: "SELECT * FROM books ORDER BY title,author")
            .fetch_all( executor: connection_pool)
            .await?,
    )
}
```

```
pub async fn book_by_id(connection_pool: &SqlitePool, id: i32) -> Result<Book> {
    Ok(sqlx::query_as::<_, Book>( sql: "SELECT * FROM books WHERE id=$1")
        .bind( value: id)
        .fetch_one( executor: connection_pool)
        .await?)
}
```

Creating Records

SQLX isn't an ORM (Object Relational Mapper) - so it doesn't map structures back to the database.

Once again, binding makes the query safe and easy.

We use get(0) at the end to retrieve the new primary key we return from the query.

```rust
pub async fn add_book<S: ToString>(
    connection_pool: &SqlitePool,
    title: S,
    author: S,
) -> Result<i32> {
    let title = title.to_string();
    let author = author.to_string();
    Ok(
        sqlx::query("INSERT INTO books (title, author) VALUES ($1, $2) RETURNING id")
            .bind(title)
            .bind(author)
            .fetch_one(connection_pool)
            .await?
            .get(0),
    )
}
```

Updating & Deleting Records

Once again, we use data binding to keep the queries safe and easy to create.

We now have a full CRUD API: Create, Read, Update, Delete.

```rust
pub async fn update_book(connection_pool: &SqlitePool, book: &Book) -> Result<()> {
    sqlx::query("UPDATE books SET title=$1, author=$2 WHERE id=$3")
        .bind(&book.title)
        .bind(&book.author)
        .bind(&book.id)
        .execute(connection_pool)
        .await?;
    Ok(())
}
```

```rust
pub async fn delete_book(connection_pool: &SqlitePool, id: i32) -> Result<()> {
    sqlx::query("DELETE FROM books WHERE id=$1")
        .bind(id)
        .execute(connection_pool)
        .await?;
    Ok(())
}
```

Unit Test the Data Model

Since we're using an in-memory database, we get the luxury of a fresh database every time.

SQLX includes a number of helpers for unit testing: fixtures, migrations.

In this case, we're keeping it simple.

(Dive into the source code to see the other tests)

Good news: All the unit tests succeeded!

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

    Herbert Wolverson
    #[sqlx::test]
    async fn get_all() {
        dotenv::dotenv().ok();
        let cnn : SqlitePool = init_db().await.unwrap();
        let all_rows : Vec<Book> = all_books( connection_pool: &cnn).await.unwrap();
        assert!(!all_rows.is_empty());
    }
```

Part 3: REST API (Axum)

Setup the REST API

Create a new file `rest.rs` in the `src` directory.

Add `mod rest` to main.rs to include it.

Start by defining a function that exposes our REST API.

Routers are Axum's way of connecting URLs with a handler and a function.

- A get handler handles GET requests. post, patch and delete handle the associated HTTP verb of the same name.
- A handler takes a function as a parameter - and runs that function when a request matches.

```rust
pub fn books_service() -> Router {
    Router::new()
        .route( path: "/",  method_router: get( handler: get_all_books)) : Router<(), Body>
        .route( path: "/:id",  method_router: get( handler: get_book)) : Router<(), Body>
        .route( path: "/add",  method_router: post( handler: add_book)) : Router<(), Body>
        .route( path: "/edit",  method_router: put( handler: update_book)) : Router<(), Body>
        .route( path: "/delete/:id",  method_router: delete( handler: delete_book))
}
```

Connect the REST API to main

Create a router function that builds another Router. Using nest_service, we can attach another service with a base URL.

Calling .layer allows us to insert the connection pool as a “Tower Layer” into Axum - available for dependency injection into handlers.

So we:

1. Connect our database API.
2. Build a router with a connection pool.
3. Listen on port 3001
4. Start the web server.

```
fn router(connection_pool: SqlitePool) -> Router {
    Router::new()
        // Nest service allows you to attach another router to a URL base.
        // "/" inside the service will be "/books" to the outside world.
        .nest_service( path: "/books",  service: rest::books_service()) : Router<(), Body>
        // Add the connection pool as a "layer", available for dependency injection.
        .layer(Extension(connection_pool))
}
```

```
#[tokio::main]
async fn main() -> Result<()> {
    // Load environment variables from .env if available
    dotenv::dotenv().ok();

    // Initialize the database and obtain a connection pool
    let connection_pool : SqlitePool  = init_db().await?;

    // Initialize the Axum routing service
    let app : Router  = router(connection_pool);

    // Define the address to listen on (everything)
    let addr : SocketAddr  = SocketAddr::from( pieces: ([0, 0, 0, 0], 3001));

    // Start the server
    axum::Server::bind(&addr)
        .serve( make_service: app.into_make_service())
        .await?;

    Ok(())
}
```

Reading Books via REST

Get_all_books and get_book are thin wrappers over our database API.

Note the `Extension(cnn) : Extension<SqlitePool>` line. The handler will automatically have the connection pool injected when called.

Likewise, `Path(id)` will automatically extract from the named path entry in the route.

```
async fn get_all_books(
    Extension(cnn): Extension<SqlitePool>,
) -> Result<Json<Vec<Book>>, StatusCode> {
    if let Ok(books : Vec<Book> ) = all_books( connection_pool: &cnn).await {
        Ok(Json(books))
    } else {
        Err(StatusCode::SERVICE_UNAVAILABLE)
    }
}
```

```
async fn get_book(
    Extension(cnn): Extension<SqlitePool>,
    Path(id): Path<i32>,
) -> Result<Json<Book>, StatusCode> {
    if let Ok(book : Book ) = book_by_id( connection_pool: &cnn, id).await {
        Ok(Json(book))
    } else {
        Err(StatusCode::SERVICE_UNAVAILABLE)
    }
}
```

Creating, Updating and Deleting Books

Once again, we just make a thin wrapper around the database model.

Congratulations, you've implemented Create, Read, Update and Delete REST functions.

```rust
async fn add_book(
    Extension(cnn): Extension<SqlitePool>,
    extract::Json(book): extract::Json<Book>,
) -> Result<Json<i32>, StatusCode> {
    if let Ok(new_id: i32) = crate::db::add_book(connection_pool: &cnn, &book.title, &book.author).await {
        Ok(Json(new_id))
    } else {
        Err(StatusCode::SERVICE_UNAVAILABLE)
    }
}
```

```rust
async fn update_book(
    Extension(cnn): Extension<SqlitePool>,
    extract::Json(book): extract::Json<Book>,
) -> StatusCode {
    if crate::db::update_book(connection_pool: &cnn, &book).await.is_ok() {
        StatusCode::OK
    } else {
        StatusCode::SERVICE_UNAVAILABLE
    }
}
```

```rust
async fn delete_book(Extension(cnn): Extension<SqlitePool>, Path(id): Path<i32>) -> StatusCode {
    if crate::db::delete_book(connection_pool: &cnn, id).await.is_ok() {
        StatusCode::OK
    } else {
        StatusCode::SERVICE_UNAVAILABLE
    }
}
```

Unit Test the REST System: Framework

Start by building a test framework, and a setup_tests function that reads the connection URL, creates a database pool.

Using the TestClient from Axum, it creates a local test-version of the website and returns the client - ready for testing.

```
#[cfg(test)]
mod test {
    use super::*;
    use axum_test_helper::TestClient;

    5 usages  Herbert Wolverson
    async fn setup_tests() -> TestClient {
        dotenv::dotenv().ok();
        let connection_pool : SqlitePool = crate::init_db().await.unwrap();
        let app : Router = crate::router(connection_pool);
        TestClient::new( svc: app)
    }
```

Unit Testing the REST Framework

Using the test client, and tokio::test for an easy async testing environment - you can quickly test that each of your REST verbs works.

Let's take a quick look at the code for the other tests.

Good news: All the unit tests work!

```rust
#[tokio::test]
async fn get_all_books() {
    let client : TestClient = setup_tests().await;
    let res : TestResponse = client.get( url: "/books").send().await;
    assert_eq!(res.status(), StatusCode::OK);
    let books: Vec<Book> = res.json().await;
    assert!(!books.is_empty());
}
```

Part 4: Web View (JS+HTML)

Serve static content

Add a new file, `view.rs` and `mod view;` to your main file.

The view system embeds HTML at compile time, and serves it. You could easily use Tower's `ServeDir` feature for dynamic files.

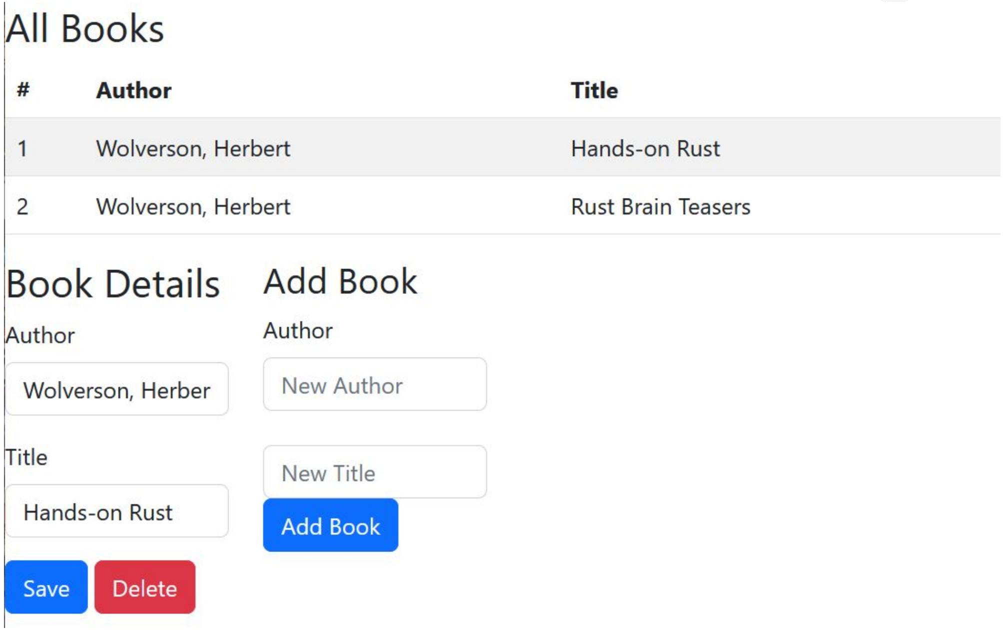
```rust
use axum::response::Html;
use axum::routing::get;
use axum::Router;

1 usage  Herbert Wolverson
pub fn view_service() -> Router {
    Router::new().route( path: "/",  method_router: get( handler: index_page))
}

1 usage  Herbert Wolverson
const INDEX_PAGE: &str = include_str!("index.html");

1 usage  Herbert Wolverson
async fn index_page() -> Html<&'static str> {
    Html(INDEX_PAGE)
}
```

Try it out!

Run the server with cargo run

Navigate to http://localhost:3001/

You can add/edit/remove books.



Part 5: Containerize in Docker

Build a docker file

Run “docker init”

Select Rust

Select server port 3001

Modify your Dockerfile:

- Add the DATABASE_URL environment variable
- Add migrations to the mounts list

```
# Set the DB URL
ENV DATABASE_URL="sqlite::memory:"
```

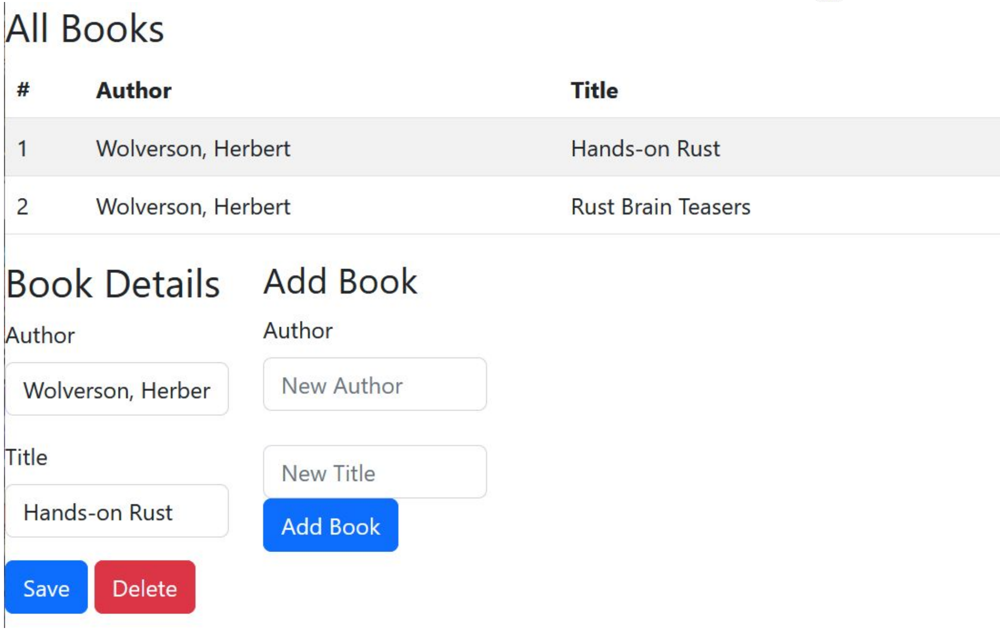
```
RUN --mount=type=bind,source=src,target=src \
    --mount=type=bind,source=Cargo.toml,target=Cargo.toml \
    --mount=type=bind,source=Cargo.lock,target=Cargo.lock \
    --mount=type=bind,source=migrations,target=migrations \
    --mount=type=cache,target=/app/target/ \
    --mount=type=cache,target=/usr/local/cargo/registry/ \
    <<EOF
```

Build your Container

docker build -t axum-test .

Start the image in Docker Desktop (don't forget to forward port 3001!) - navigate to http://localhost:3001/

And up comes the UI!



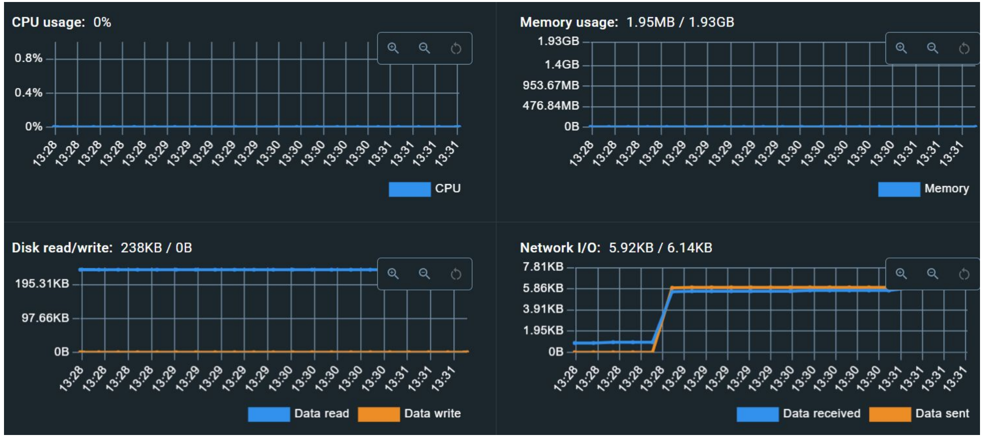
So far, so good!

In relatively short time, we've:

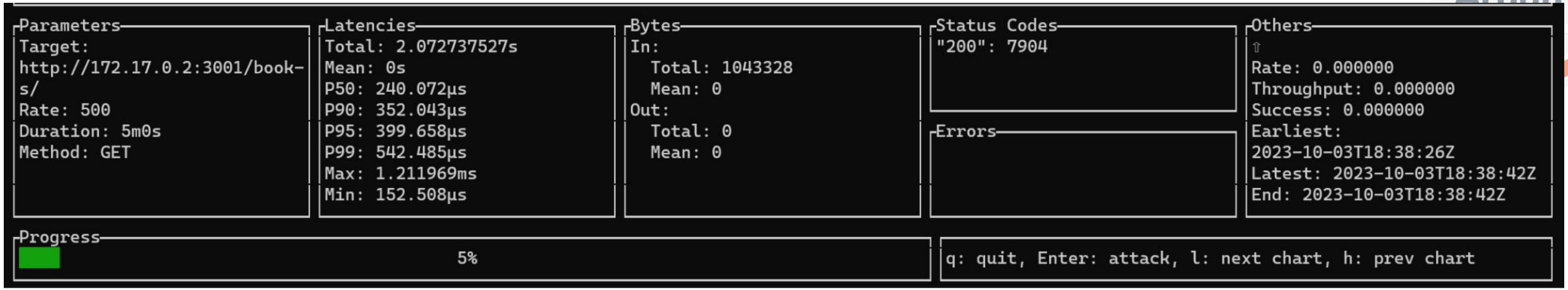
- Constructed a database layer with migration support
- Built a database model with a full range of CRUD operations and unit tests
- Created a REST API and fully unit tested it
- Created an HTML/JS view of the API in action

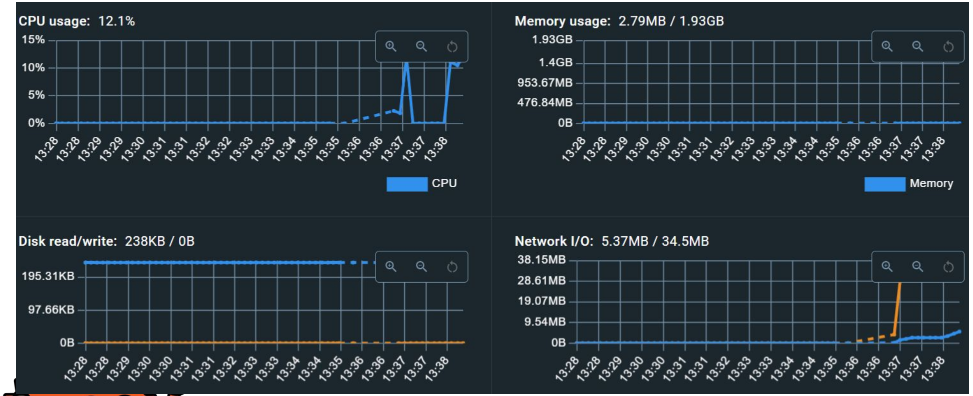
So how resource intensive is this quick application?

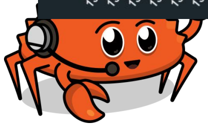
While mostly idle:



500 Requests per Second







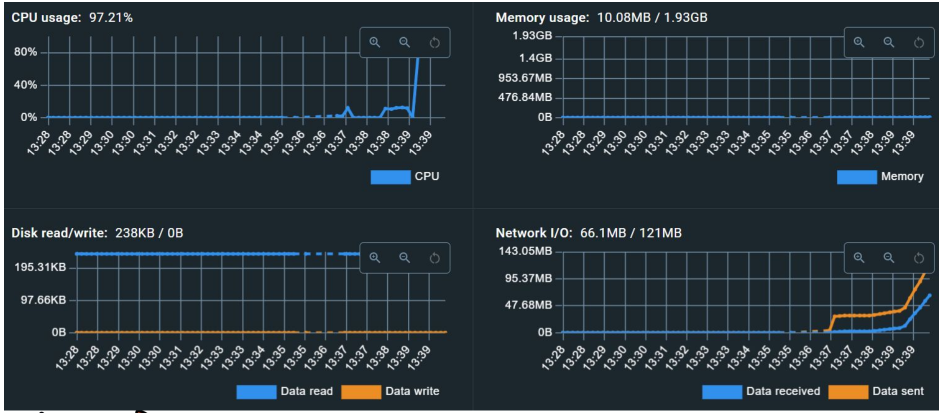
We're hardly using any memory (3 Mb!)

CPU is ticking upwards - and its mostly SQLite

My little MacBook Air can handle this. :-)

At 5,000 Requests per Second

```
Parameters                         Latencies                     Bytes                    Status Codes        Others
Target:                            Total: 58.801996113s          In:                      "200": 213143       ⇧
http://172.17.0.2:3001/books/      Mean: 0s                        Total: 28134876                            Requests: 213143
Rate: 5000                         P50: 192.188µs                  Mean: 0                                    Rate: 0.000000
Duration: 5m0s                     P90: 315.016µs                Out:                                         Throughput: 0.000000
Method: GET                        P95: 539.06µs                   Total: 0               Errors              Success: 0.000000
                                   P99: 2.315229ms                 Mean: 0                                    Earliest: 2023-10-03T18:39:29Z
                                   Max: 9.989046ms                                                            Latest: 2023-10-03T18:40:12Z
                                   Min: 104.259µs                                                             End: 2023-10-03T18:40:12Z
```



We're up to 10Mb of RAM! Oh no, however will we host that?

My Macbook Air is struggling a little at 97% CPU usage - but we haven't generated any errors yet.

Let’s Spend Some Cache

Let’s add a Cache

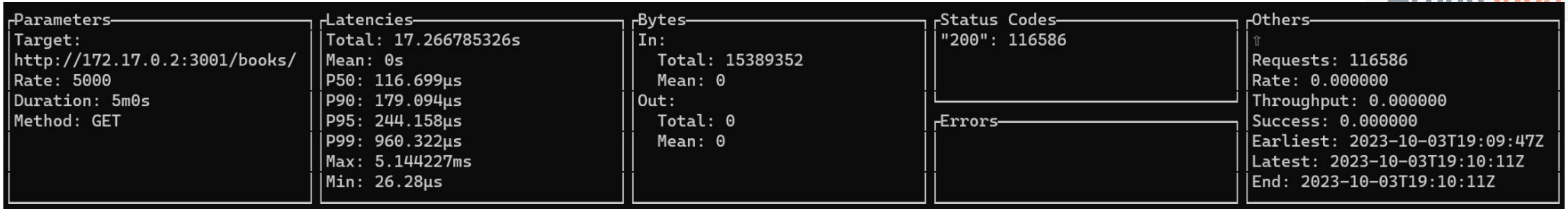
Hitting the database for a full-table retrieval for every request isn’t really necessary.

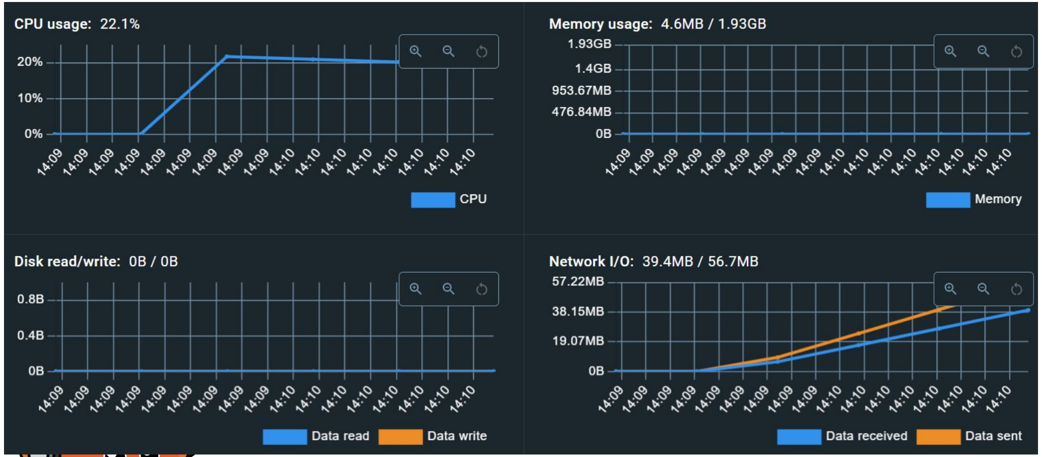
Rust is supposed to be all about Fearless Concurrency, so let’s make a self-invalidating, thread-safe cache for our database model layer.

The basic cache is pretty simple: a Read-Write lock wrapping a list of books, with invalidate and refresh methods.

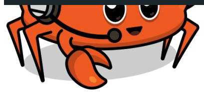
```
struct BookCache {
    all_books: RwLock<Option<Vec<Book>>>,
}

new *
impl BookCache {
    new *
    fn new() -> Self {
        Self {
            all_books: RwLock::new( value: None),
        }
    }

    new *
    async fn all_books(&self) -> Option<Vec<Book>> {
        let lock : RwLockReadGuard<Option<Vec<...>>> = self.all_books.read().await;
        lock.clone()
    }

    new *
    async fn refresh(&self, books: Vec<Book>) {
        let mut lock : RwLockWriteGuard<Option<Vec<...>>> = self.all_books.write().await;
        *lock = Some(books);
    }

    new *
    async fn invalidate(&self) {
        let mut lock : RwLockWriteGuard<Option<Vec<...>>> = self.all_books.write().await;
        *lock = None;
    }
}

5 usages   new *
static CACHE: Lazy<BookCache> = Lazy::new( f: BookCache::new);
```

Now how does it perform?



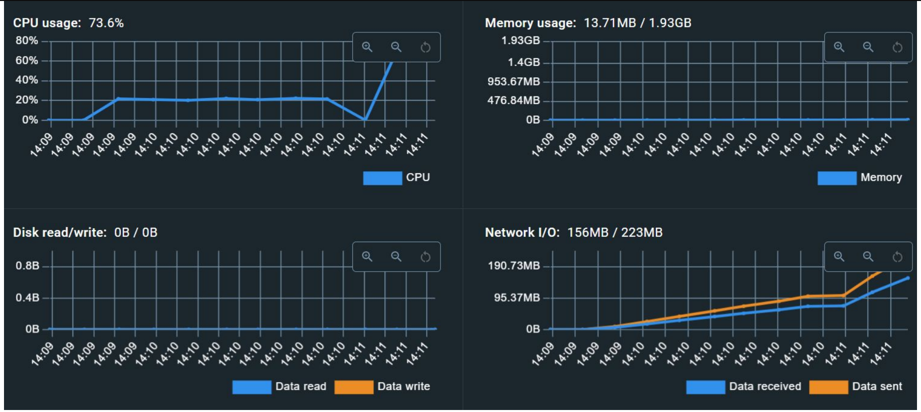


Now the Macbook Air is handling 5,000 requests per second without breaking a sweat - and still only using 5Mb of RAM!



How about 20,000 requests per second now?

```
Parameters
Target:
http://172.17.0.2:3001/books/
Rate: 20000
Duration: 5m0s
Method: GET

Latencies
Total: 17m49.93825839s
Mean: 0s
P50: 94.652µs
P90: 1.154262ms
P95: 5.396124ms
P99: 28.881083ms
Max: 281.897081ms
Min: 22.074µs

Bytes
In:
  Total: 112370412
  Mean: 0
Out:
  Total: 0
  Mean: 0

Status Codes
"200": 851291

Errors

Others
⇧
Requests: 851291
Rate: 0.000000
Throughput: 0.000000
Success: 0.000000
Earliest: 2023-10-03T19:11:04Z
Latest: 2023-10-03T19:11:46Z
End: 2023-10-03T19:11:46Z
```



We're hitting the upper limits of what we can run on a Macbook Air! Latencies are starting to worsen, CPU usage is high. But no errors!

In other words, we can survive bursts - but would need to scale up from a tiny laptop to handle this much live load.

What We’ve Learned

- SQLX provides a small, fast and expressive database layer with and migrations, test fixtures.
- Axum provides an expressive, enterprise-ready foundation for web services.
- Rust integrates easily with Docker for containerized builds - ready for your infrastructure.
- Rust glues it together nicely, and provides easy concurrency for things like caches.
- Rust is a great choice for line-of-business REST apps.

Next Steps

There's always improvements to be made! Here are some that you could make with relatively little effort - but not in this webinar.

- Tracing for logging and span timing.
- Tracing for OpenTelemetry support (1 crate and 1 line of code!)
- A proper database!
  - SQLX supports MySQL, PostreSQL, MS-SQL.
  - Rust drivers are available for most databases.
- A proper cache layer.

Most of the content covered today is included in *Ultimate Rust: Foundations*.