**Building a microservice with Rust and Actix-Web**

[[Rajat Nigam](https://rajatnigam89.medium.com/?source=post_page-----519e665063e6--------------------------------)](https://rajatnigam89.medium.com/?source=post_page-----519e665063e6--------------------------------)

[[Dev Genius](https://blog.devgenius.io/?source=post_page-----519e665063e6--------------------------------)](https://blog.devgenius.io/?source=post_page-----519e665063e6--------------------------------)

[Rajat Nigam](https://rajatnigam89.medium.com/?source=post_page-----519e665063e6--------------------------------)

·

Follow

Published in

[Dev Genius](https://blog.devgenius.io/?source=post_page-----519e665063e6--------------------------------)

·

5 min read

·

Dec 26, 2023

5

Rust in particular is becoming a popular choice among robust, reliable programming languages--especially at the system level. We will look at a simple but powerful RESTful API written in Rust with Actix-Web and MongoDB in this blog post. Now we dive into the codebase and talk about the frameworks involved, not to mention pointing out some of Rust’s advantages over Java, python, and Go.



Connaught Place from the early 90s…..Nostalgic !!

**Project Overview**

This project, available on [GitHub](https://github.com/rajat965ng/rust-account-info.git), is a Rust program for managing account information. The microservice **account\_info**, developed with the Actix-Web framework and MongoDB as a database, is one of many microservices. Here are the major elements and features.

**Frameworks Used**

The project leverages several libraries and frameworks to streamline development:

* **Actix-Web (v4.4.0)**: A high-performance, actor-based web framework for Rust offering support for asynchronous and concurrent web applications.
* **Tokio (v1.0.0)**: A runtime for writing safe, asynchronous applications in Rust.
* **Serde (v1.0.193)**: A Rust library for serializing and deserializing data structures.
* **MongoDB (v2.8.0)**: A NoSQL driver used for the storage of account information.
* **dotenv (v0.15.0)**: A library for loading configuration from a .env file.

**Project Setup**

The Cargo.toml has a list of all the project’s dependencies; they include Actix-Web, Tokio, Serde, Serde\_json, and dotenv. Actix-Web is a powerful, asynchronous web framework for Rust; Tokio provides runtime support to create asynchronous programs and Serde is used for serialization and deserialization. With dotenv, the project can access environment variables from a file.

[package]  
name = "account\_info"  
version = "0.1.0"  
edition = "2021"  
  
[dependencies]  
actix-web = "4.4.0"  
tokio = { version = "1.0.0", features = ["rt", "rt-multi-thread", "macros"] }  
serde = { version = "1.0.193", features = ["derive"] }  
serde\_json = "1.0.108"  
dotenv = "0.15.0"  
  
[dependencies.mongodb]  
version = "2.8.0"  
features = ["tokio-runtime"]

Here, the project uses Actix-Web as a web framework, Tokio for an asynchronous runtime, and Serde for serialization and deserialization. Also, MongoDB is integrated for the storage of documents.

**Main Application Logic**

The main.rs file is the entry point for the application. It creates a MongoDB client, connects to the “**account**” database, and begins an Actix-Web server. The server is set up to deal with routes defined in the controller module.

// ... (imports)  
  
#[actix\_web::main]  
async fn main() -> std::io::Result<()> {  
 dotenv().ok();  
 let client = Client::with\_uri\_str("mongodb://localhost:27017").await.unwrap();  
 let database = client.database("account");  
 HttpServer::new(move || {  
 App::new()  
 .configure(controller::config)  
 .app\_data(Data::new(database.clone()))  
 }).bind(("0.0.0.0", 8080))?.run().await  
}

**Controller, Service, and Model**

The project structure includes controller.rs, service.rs, and model.rs, respectively handling web requests, business logic, and data modeling.

**RESTful Endpoints**

controller.rsfile provides the RESTful endpoints for handling account information. N oteworthy endpoints include:

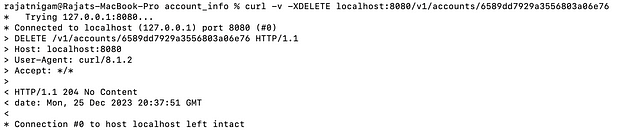
* POST /v1/accounts: Save a new account.



* GET /v1/accounts: Retrieve all accounts.



* DELETE /v1/accounts/{oid}: Delete an account by ID.



// ... (imports)  
  
#[get("")]  
async fn get\_all\_accounts(mut database: Data<Database>) -> HttpResponse {  
 // ... (retrieve and return all accounts)  
}  
  
#[post("")]  
async fn save\_accounts(mut database: Data<Database>, payload: Payload) -> HttpResponse {  
 // ... (save account and return the inserted ID)  
}  
  
#[delete("/{oid}")]  
async fn delete\_account(mut database: Data<Database>, path: Path<String>) -> HttpResponseBuilder {  
 // ... (delete account and return appropriate HTTP response)  
}  
  
pub fn config(conf: &mut ServiceConfig) {  
 let scope = web::scope("/v1/accounts")  
 .service(get\_all\_accounts)  
 .service(save\_accounts)  
 .service(delete\_account);  
  
 conf.service(scope);  
}

**Service Module**

The service module contains functions for interacting with the MongoDB database, such as finding all accounts or adding a new account. As Rust-safe asynchronous functions are find\_all, save, and delete.

**Model Module**

The model module defines the data structure for an account, including its ID, holder name, and status. The model.rs file defines the data model for the account, showcasing Rust's strong type system and its ability to handle complex data structures.

The model module defines the data structure of an account: its ID, holder’s name, and status. The model.rsfile defines the structure of the account, which shows off Rust’s efficient type system and its facility with complicated structures.

// ... (imports)  
  
#[derive(Clone, Debug, PartialEq, Eq, Deserialize, Serialize)]  
pub(crate) struct Account {  
 // ... (fields: id, holder\_name, status)  
}  
  
#[derive(Clone, Debug, PartialEq, Eq, Deserialize, Serialize)]  
pub(crate) enum Status {  
 ACTIVE,  
 INACTIVE  
}

**Rust vs. Java vs. Golang vs. Python**

Now, let’s briefly compare the Rust implementation with similar projects in Java, Golang, and Python:

**1. Performance**

Rust focuses on performance and memory safety. Through zero-cost abstractions, developers can write high-performance code without compromising safety. Other languages that do well include Java and Golang, but Rust’s emphasis on memory safety sets it apart.

**2. Concurrency**

With rust, the ownership system gives memory safety without sacrificing concurrency. Golang uses goroutines for concurrency while Java and Python have more traditional threading models.

**3. Developer Experience**

Rust’s borrow checker while it may be difficult for beginners, checks memory safety at compile time. Java gives the convenience of a rich environment with automatic garbage collection, while Golang offers simplicity. But Python, although dynamically typed, is known for its readability and ease of use.

**4. Ecosystem and Libraries**

The Java ecosystem is big, and libraries are pretty well established; Golang stresses simplicity and standard library. Now Rust is building a new environment around it, while Python with its large library is the choice for web development.

**Rust in Action**

The Java ecosystem is large, and the libraries are not bad; Golang emphasizes simplicity and standard libraries. Now Python with its big library is the choice for web development, while Rust is building a new environment around it.

Rust is beginning to influence several industries, and several organizations are using it for their application development. Some notable examples include:

* **Mozilla**: Rust was developed by Mozilla, best known for its Firefox browser. Many of Mozilla’s projects use it.
* **Dropbox**: Performance and Memory Safety The cloud storage service Dropbox has begun using Rust for parts of its core components.
* **AWS IoT Greengrass**: AWS uses Rust in its IoT Greengrass project for building secure and efficient edge computing solutions.
* **Cloudflare**: Cloudflare, a web infrastructure and security company, has added Rust to its systems for better performance and security.

These examples demonstrate the versatility of Rust and how well it’s suited for building stable, high-performance systems.

**Conclusion**

The “**account\_info**” project is a useful illustration of how to use Rust in web development by utilizing strong frameworks and tools to create an effective and scalable account information service. Rust’s influence on numerous sectors and applications is becoming more and more obvious as it gains more traction. Rust is an appealing option for contemporary software development because it combines performance, safety, and concurrency.

Investigate the [GitHub repository](https://github.com/rajat965ng/rust-account-info.git), try out Rust, and think about using it for your upcoming project to make use of the language’s advantages and create dependable and effective systems.