**Build CRUD REST API with Rust 🦀and MySQL using Axum & SQLx**

[[](https://medium.com/@raditzlawliet?source=post_page-----d7e50b3cd130--------------------------------)](https://medium.com/@raditzlawliet?source=post_page-----d7e50b3cd130--------------------------------)

[Radityo Hernanda](https://medium.com/@raditzlawliet?source=post_page-----d7e50b3cd130--------------------------------)

·

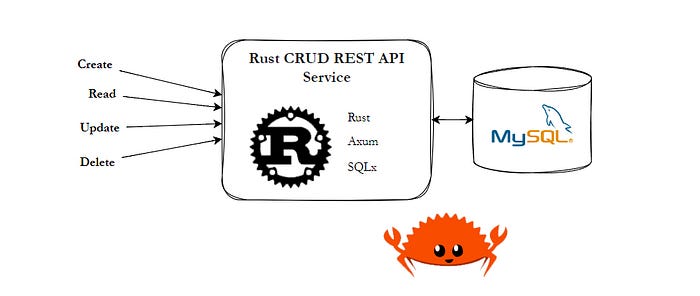
Follow

6 min read

·

4 days ago

1



Rust CRUD REST API

🦀**Rust Language**🦀have been hype in the few last year. So, let’s go straight to create a Rust Project for Simple CRUD REST API with Axum and SQLx.

For final result, you check on my [github repository](https://github.com/raditzlawliet/rust-notes-rest-axum-sqlx).

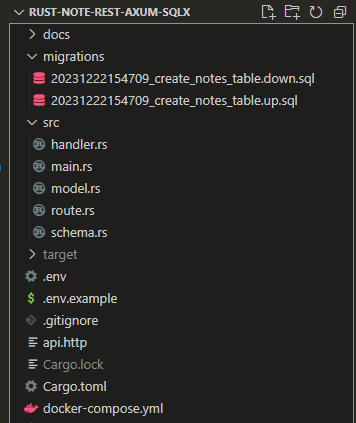
**Project Requirement & Scope**

Let create a **REST API Services for managing Notes**. This service will handle to perform *Create, Read, Update and Delete Notes*, and also *listing all notes with simple paginating*.

We build this on:  
- Rust 1.73

My Recommended IDE & Plugins when building this:  
- VSCode  
- VSCode Plugins: **Rust Analyzer**(Rust support language for VSCode)  
- VSCode Plugins: **REST Client**(a Simple sending HTTP Request to testing our REST API)

Our Final project structure will be looks like this:

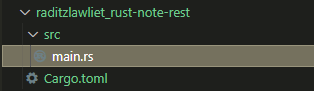


Final Project Structure

**Set-up Rust Server with Axum**

After installing rust (I’m assume you guys already install rust or see the [guide here](https://www.rust-lang.org/tools/install)), let’s create & initialize the project and set-up server with axum

# Init Project  
cargo init raditzlawliet\_rust-note-rest



Command above will create a folder and a new rust project inside. Then, we need add a depedency will be used in this project

# Depedency  
cargo add axum  
cargo add tokio -F full  
cargo add tower-http -F "cors"  
cargo add serde\_json  
cargo add serde -F derive  
cargo add chrono -F serde  
cargo add dotenv  
cargo add uuid -F "serde v4"  
cargo add sqlx --features "runtime-async-std-native-tls mysql chrono uuid"

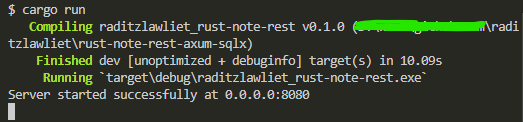
After running command above, the cargo.toml files will looks like this

Then, Let’s write our first API Service, open main.rs and modify into this

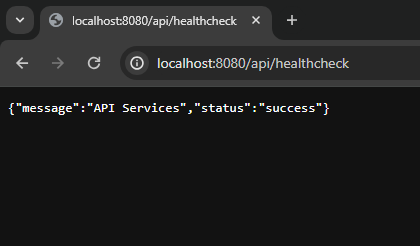
Code above will run a server with single API api/healthcheck and return a JSON response. Let’s try to run it out using

# Build & Run Project  
cargo build  
cargo run

The result may looks like this



Now you can test to call this API to <http://127.0.0.1:8080/api/healthcheck>



AND, our Rust Server API Service is complete!

Additionally, we can also re-run automatically every time the code change using cargo-watch

# CLI For Watch source when running & Automatically rebuild the project  
cargo install cargo-watch  
  
# Run with watch the src/  
cargo watch -q -c -w src/ -x run

**Set-up MySQL Server & Connect it with SQLx**

Let set-up MySQL, you can use your own MySQL or using Docker instead.

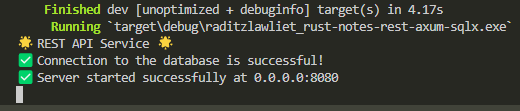
For Docker let create new environment file .env and docker compose file docker-compose.yml

Then run it to start MySQL on root project or in folder where docker-compose.yml file located.

# Run Docker Compose & Detach  
docker compose up -d  
  
# (Bonus! for stopping MySQL Docker)  
docker compose stop

Ok, Now update main.rs to be able connect the MySQL

Above code should looks like above figure. In this section we have able to connect our Rust Axum Server to MySQL and use that connection to perform our CRUD handler later.



Connected to Database ^\_^

**Database Migration**

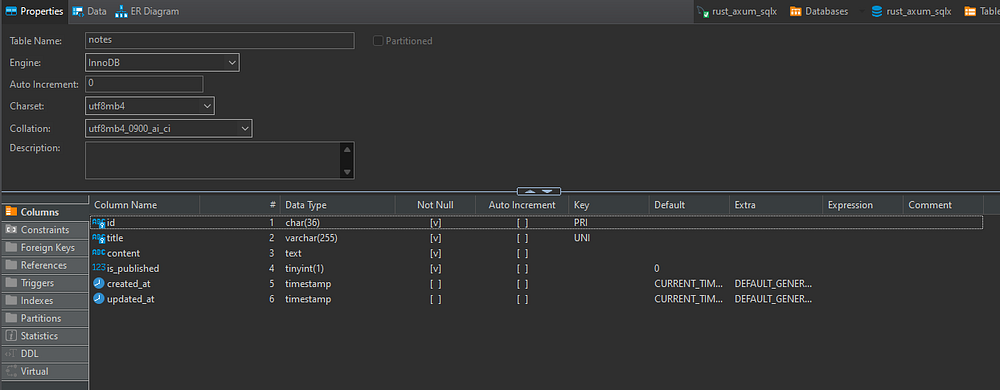
Create a Notes Table, you can use other method, but now let try to using migration with sqlx-cli . Install sqlx-cli and create a migration file

# CLI For migration  
cargo install sqlx-cli  
  
# create a migration  
sqlx migrate add -r create\_notes\_table

Command above will create 2 migration file for Up and Down/Revert. Update them to create notes table and drop notes table

Then we can run the migration via sqlx-cli (And if you need to revert, you can also do that). Finally our database is ready to be handle (you can check using Visual Tools like DBeaver or MySQL Workbench or else)

# perform migration up  
sqlx migrate run  
  
# (Bonus!, perform migration down/revert)  
sqlx migrate revert



**Define SQLx Model & Response**

Create model.rs and define **struct** the **Notes Model DB** and **Notes Model Response**. Later this model will be use for retrieve data from MySQL and parse to NoteModel, then we can adjust to NoteModelResponse for JSON response.

# Create file model.rs  
touch src/model.rs

* Notable: for Nullable column, you can use type Option<T> so it value can be Null or T
* Notable: BOOLEAN in MySQL is TINYINT(1) so we can use i8 to retrieve the record and later we can parse to Boolean

**Define API Request Schema**

Create schema.rs and define struct for Schema API Request

# Create schema.rs  
touch src/schema.rs

API will be plan like this:

* List will have pagination page and limit parameter and use FilterOptions e.g., GET <http://127.0.0.1:8080/api/notes?page=0&limit=10>
* Create and Update will use CreateNoteSchema and UpdateNoteSchema
* Read and Delete will have id param and use ParamOptions

**Implement CRUD Functionality**

Now we can create handler for all CRUD functionality and later we can attach them to our API.

Create handler.rs file

# create handler.rs  
touch src/handler.rs

Fill the imports on top before creating each handler (for easiness) and one function to parsing **NoteModel**to **NoteModelResponse**

**Implement handler to Fetch All Records**

Create a function to handle fetching all Notes records

* 1st step we need to define limit& offsetfrom param
* then we query with param and fetch all to notes
* convert NoteModel to NoteModelResponse and put it in note\_responses
* define json\_response and return it

**Implement handler to insert a record (Create)**

Create a function to handle inserting a record

* Because Note using UUID as ID, we need to create ID first let id = uuid::Uuid::new\_v4().to\_string();
* Query insert with payload (that already transform automatically into CreateNoteScheme)
* Check error (e.g., Duplicate Record)
* Get Inserted notes by using ID we create first
* return it to Json response

**Implement handler to get a record (Read)**

Create a function to handle getting a single record by ID

* ID will automatically transform from path route and pass to param variable id
* Query select by ID
* then we check error like Database error or no record found, otherwise return the Note Json Response

**Implement handler to update a record (Update)**

create a function to handle updating a record by ID

* We get single Note by ID
* parsing data (if needed), in our case we need to parsing is\_published from boolean to i8 and vice versa
* Query update and check if there error
* Get updated note and response it as JSON

**Implement handler to delete a record (Delete)**

create a function to handle deleting a record By ID

* Quite straightforward, using id from route path, we query delete by ID
* then return **NO\_CONTENT**as successor **NOT\_FOUND**when ID not found

**Register all handler to Axum Router**

Final handler.rs should be looks like this, (including moved other handler like health\_check\_handler)

Now create route.rs to connect route and handler

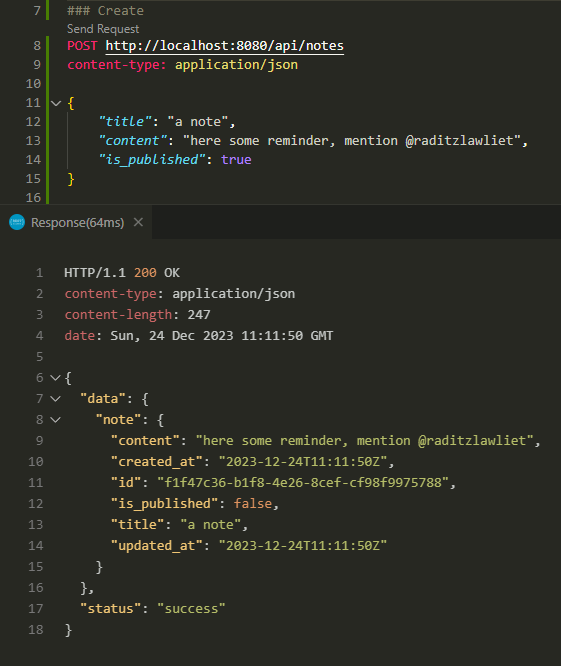
# Create route.rs  
touch src/route.rs

Then, update main.rs to connect them all and add CORS layer on it

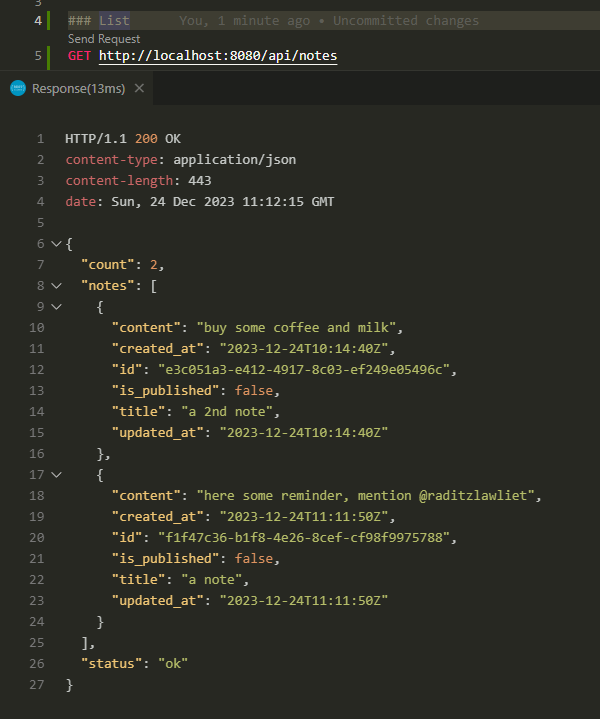
Now, you can try to test them out using REST Client or Postman or else.

* Create: POST http://localhost/8080/api/notes it will create a note with JSON Payload
* List: GET http://localhost/8080/api/notes listing all notes
* Get: GET http://localhost/8080/api/{ID} get a note by ID
* Update: PATCH http://localhost/8080/api/notes/{ID} update note by ID with JSON Payload
* Delete: DELETE http://localhost/8080/api/notes/{ID} delete a note by ID

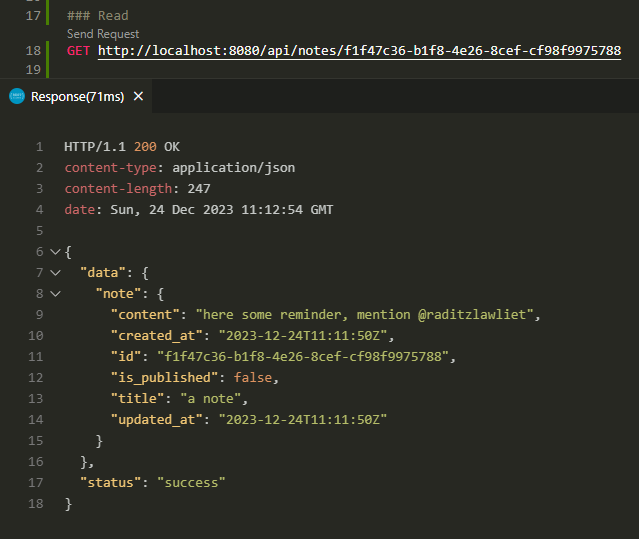
Bonus! You can use REST Client, a plugins on VSCode to test HTTP, create test.http and test them out



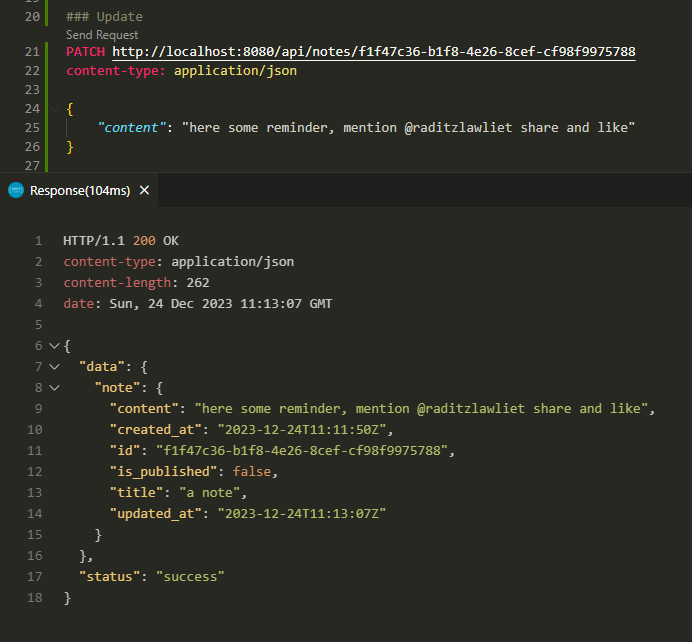
API Create Note



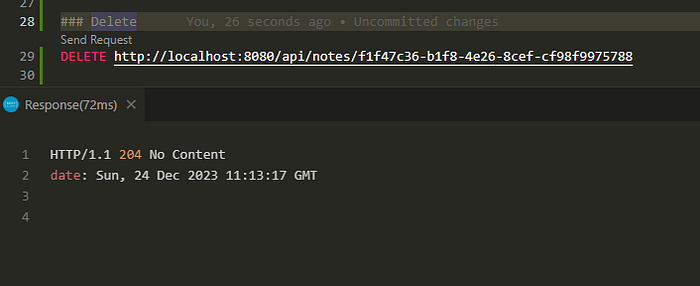
API List Notes



API Read Note



API Update Note



API Delete Note

**Conclusion**

Finally we complete a Rust Server REST API Service to perform Notes CRUD. You can more extend and do some real project with this.

I wish this can help you getting started creating a web service with Rust.

For final project, you find it on [my github repository](https://github.com/raditzlawliet/rust-notes-rest-axum-sqlx)

Thanks and happy coding!

[Rust](https://medium.com/tag/rust?source=post_page-----d7e50b3cd130---------------rust-----------------)

[Rest](https://medium.com/tag/rest?source=post_page-----d7e50b3cd130---------------rest-----------------)

[API](https://medium.com/tag/api?source=post_page-----d7e50b3cd130---------------api-----------------)

[Axum](https://medium.com/tag/axum?source=post_page-----d7e50b3cd130---------------axum-----------------)

[MySQL](https://medium.com/tag/mysql?source=post_page-----d7e50b3cd130---------------mysql-----------------)