Rust as a High-level Programming Language:

Backend Webdev with axum and Diesel

Ian & Han

Aug 21, 2025

Rust Malaysia @ Shortcut Asia

Agenda

1. axum web framework
2. Diesel ORM
3. Example & demo - e-wallet app
4. Q&A

axum web framework

Hello, World!

```rust
use axum::Router;
use axum::routing::get;
use tokio::net::TcpListener;

#[tokio::main]
async fn main() {
    // build our application with a single route
    let app = Router::new()
        .route("/", get(async || "Hello, World!"));

    // run our app with hyper, listening globally on port 3000
    let listener = TcpListener::bind("0.0.0.0:3000").await.unwrap();
    axum::serve(listener, app).await.unwrap();
}
```

Hello, World!

```
GET / HTTP/1.1
Host: localhost:3000
```

```
HTTP/1.1 200 OK
Content-Type: text/plain; charset=utf-8

Hello, World!
```


axum
tower
(middleware)
hyper
(HTTP)

Routing

```rust
use axum::Router;
use axum::routing::{get, post};

let app = Router::new()
    .route("/", get(get_root))
    .route("/cats", get(get_cats))
    .route("/cappuccino", post(post_cappuccino));

async fn get_root() {}
async fn get_cats() {}
async fn post_cappuccino() {}
```

Routing

```rust
use axum::Router;
use axum::routing::{get, post};

let cat_routes = Router::new()
    .route("/{cat_id}", get(get_cat));

let app = Router::new()
    .route("/", get(get_root))
    .nest("/cats", cat_routes)
    .route("/cappuccino", post(post_cappuccino));

async fn get_root() {}
async fn get_cat() {}
async fn post_cappuccino() {}
```

Nesting routes

Handlers

Matching path parameters

```
GET /cats/5e HTTP/1.1
Host: localhost:3000
```

```
HTTP/1.1 200 OK
Content-Type: text/plain; charset=utf-8

Cat 5e
```

Handlers

```rust
use axum::Router;
use axum::extract::Path;
use axum::routing::get;

let app = Router::new()
    .route("/cats/{cat_id}", get(get_cat));

async fn get_cat(
    Path(cat_id): Path<String>,
) -> String {
    format!("Cat {cat_id}")
}
```

Matching path parameters

Extracting query parameters

```
GET /greeting?name=Frieren HTTP/1.1
Host: localhost:3000
```

```
HTTP/1.1 200 OK
Content-Type: text/plain; charset=utf-8

Hello, Frieren!
```

Handlers

```rust
use axum::extract::Query;
use serde::Deserialize;

#[derive(Deserialize)]
struct GetGreetingParams {
    name: String,
}

async fn get_greeting(
    Query(GetGreetingParams {
        name,
    }): Query<GetGreetingParams>,
) -> String {
    format!("Hello, {name}!")
}
```

Extracting query parameters

Handlers

JSON request

```
POST /greeting HTTP/1.1
Host: localhost:3000
Content-Type: application/json

{
  "name": "Frieren"
}
```

```
HTTP/1.1 200 OK
Content-Type: text/plain; charset=utf-8

Hello, Frieren!
```

Handlers

```rust
use axum::extract::Json;
use serde::Deserialize;

#[derive(Deserialize)]
struct PostGreetingPayload {
    name: String,
}

async fn post_greeting(
    Json(PostGreetingPayload {
        name,
    }): Json<PostGreetingPayload>,
) -> String {
    format!("Hello, {name}!")
}
```

JSON request

Handlers

JSON response

```
POST /greeting HTTP/1.1
Host: localhost:3000
Content-Type: application/json

{ "name": "Frieren" }
```

```
HTTP/1.1 200 OK
Content-Type: application/json

{ "message": "Hello, Frieren!" }
```

Handlers

```rust
use axum::extract::Json;
use serde::{Deserialize, Serialize};

#[derive(Deserialize)]
struct PostGreetingPayload { name: String }

#[derive(Serialize)]
struct PostGreetingResponse { message: String }

async fn post_greeting(
    Json(PostGreetingPayload { name }): Json<PostGreetingPayload>,
) -> Json<PostGreetingResponse> {
    Json(PostGreetingResponse {
        message: format!("Hello, {name}!"),
    })
}
```

JSON response

Handlers

Optional query parameter with a fallback value

```
GET /greeting?name=Frieren HTTP/1.1
Host: localhost:3000
```

```
HTTP/1.1 200 OK
Content-Type: text/plain

Hello, Frieren!
```

```
GET /greeting HTTP/1.1
Host: localhost:3000
```

```
HTTP/1.1 200 OK
Content-Type: text/plain

Hello, stranger!
```

Handlers

```rust
use axum::extract::Query;
use serde::Deserialize;

#[derive(Deserialize)]
struct GetGreetingParams {
  name: Option<String>,
}

async fn get_greeting(
    Query(GetGreetingParams { name }): Query<GetGreetingParams>,
) -> String {
    let name = name.unwrap_or_else(|| "stranger".to_owned());

    format!("Hello, {name}!")
}
```

Optional query parameter with a fallback value

Error handling

```
GET /greeting?name=Frieren HTTP/1.1
Host: localhost:3000
```

```
HTTP/1.1 400 Bad Request
Content-Type: text/plain

nyan please
```

```
GET /greeting?name=Frienyan HTTP/1.1
Host: localhost:3000
```

```
HTTP/1.1 200 OK
Content-Type: text/plain

Hello, Frienyan!
```

Handlers

```rust
use axum::extract::Query;
use axum::http::StatusCode;
use axum::response::Result;
use serde::Deserialize;

#[derive(Deserialize)]
struct GetGreetingParams { name: String }

async fn get_greeting(
    Query(GetGreetingParams { name }): Query<GetGreetingParams>,
) -> Result<String> {
    if !name.ends_with("nyan") {
      return Err((StatusCode::BAD_REQUEST, "nyan please"))?;
    }

    Ok(format!("Hello, {name}!"))
}
```

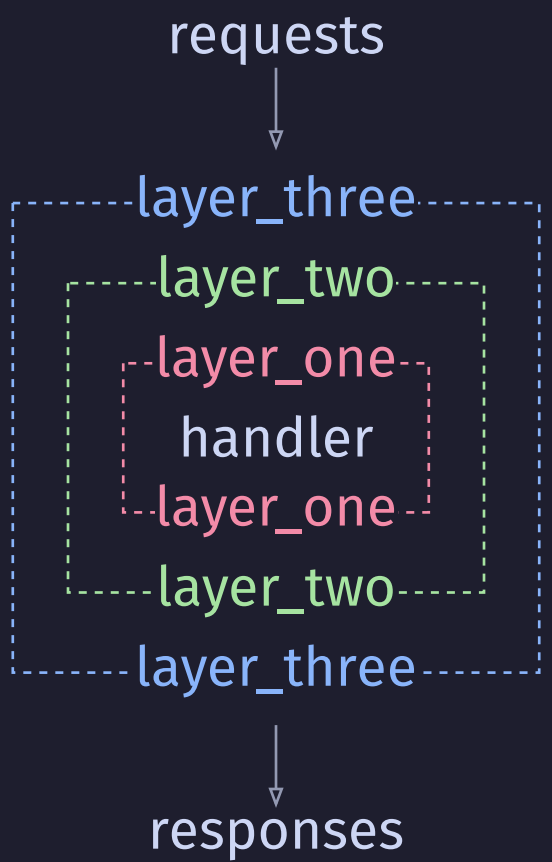
Error handling

Middleware

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

async fn handler() {}

let app = Router::new()
    .route("/", get(handler))
    .layer(layer_one)
    .layer(layer_two)
    .layer(layer_three);
```

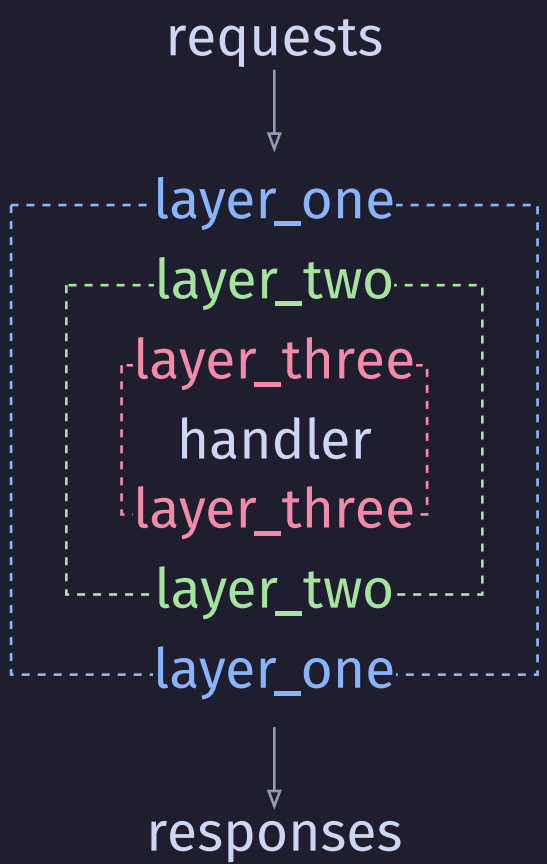


Middleware

```rust
use axum::Router;
use axum::routing::get;
use tower::ServiceBuilder;

async fn handler() {}

let app = Router::new()
    .route("/", get(handler))
    .layer(
        ServiceBuilder::new()
            .layer(layer_one)
            .layer(layer_two)
            .layer(layer_three),
    );
```



State

```rust
use axum::Router;
use axum::extract::State;
use axum::routing::get;

#[derive(Clone)]
struct AppState { cat: String }

let state = AppState { flavor: "Mocha".to_owned() };

let app = Router::new()
    .route("/cat", get(get_cat))
    .with_state(state);

async fn get_cat(State(flavor): State<String>) → String {
    format!("Catppuccin {flavor}")
}
```

State

```
GET /cat HTTP/1.1
Host: localhost:3000
```

```
HTTP/1.1 200 OK
Content-Type: text/plain; charset=utf-8

Catppuccin Mocha
```

Diesel ORM

Schema

```
DATABASE_URL=postgres://username:password@pg.example.com:5432/db_name

diesel print-schema > src/schema.rs
```

Schema

```
diesel::table! {
    cats (id) {
        id -> Uuid,
        name -> Nullable<Varchar>,
        legs -> Int4,
        purrs -> Bool,
        human_id -> Nullable<Uuid>,
    }
}

diesel::table! {
    humans (id) {
        id -> Uuid,
    }
}
```

Auto-generated from database schema

Schema

```
diesel::joinable!(cats → humans (human_id));

diesel::allow_tables_to_appear_in_same_query!(
    cats,
    humans,
);
```

Auto-detected relations from foreign key constraints

Models

```rust
use diesel::prelude::*;
use uuid::Uuid;

use crate::schema::cats;

#[derive(Debug, Queryable, Selectable)]
#[diesel(table_name = cats)]
#[diesel(check_for_backend(diesel::pg::Pg))]
pub struct Cat {
    pub id: Uuid,
    pub name: Option<String>,
    pub legs: i32,
    pub purrs: bool,
    pub human_id: Option<Uuid>,
}
```

Models

```rust
use diesel::prelude::*;
use uuid::Uuid;

use crate::schema::humans;

#[derive(Debug, Queryable, Selectable)]
#[diesel(table_name = humans)]
#[diesel(check_for_backend(diesel::pg::Pg))]
pub struct Human {
    pub id: Uuid,
}
```

Connection

```rust
use std::env;

use diesel_async::pooled_connection::deadpool::Pool;
use diesel_async::pooled_connection::AsyncDieselConnectionManager;
use diesel_async::AsyncPgConnection;

dotenvy::dotenv()?;
let db_url = env::var("DATABASE_URL")?;

let manager = AsyncDieselConnectionManager::<AsyncPgConnection>::new(
    db_url.as_str(),
);
let pool = Pool::builder(manager).build()?;
let mut conn = pool.get().await?;
```

CRUD

```rust
use diesel::prelude::*;
use diesel_async::{AsyncPgConnection, RunQueryDsl};

use crate::models::Cat;

pub async fn get_cats(
    conn: &mut AsyncPgConnection,
) -> QueryResult<Vec<Cat>> {
    use crate::schema::cats::dsl::*;

    cats
        .filter(purrs.eq(true))
        .limit(5)
        .select(Cat::as_select())
        .load(conn)
        .await
}
```

Models

```rust
use diesel::prelude::*;
use uuid::Uuid;

use crate::schema::cats;

#[derive(Debug, Default, Insertable)]
#[diesel(table_name = cats)]
pub struct NewCat {
    pub name: Option<String>,
    pub legs: i32,
    pub purrs: bool,
    pub human_id: Option<Uuid>,
}
```

CRUD

```rust
use diesel::prelude::*;
use diesel_async::{AsyncPgConnection, RunQueryDsl};

use crate::models::{Cat, NewCat};

pub async fn create_cat(conn: &mut AsyncPgConnection) -> QueryResult<Cat> {
    use crate::schema::cats;

    let new_cat = NewCat { legs: 3, ..Default::default() };

    diesel::insert_into(cats::table)
        .values(&new_cat)
        .returning(Cat::as_returning())
        .get_result(conn)
        .await
}
```

Models

```rust
use diesel::prelude::*;
use uuid::Uuid;

use crate::schema::cats;

#[derive(Debug, AsChangeset, Identifiable, Queryable, Selectable)]
#[diesel(table_name = cats)]
#[diesel(check_for_backend(diesel::pg::Pg))]
pub struct Cat {
    pub id: Uuid,
    pub name: Option<String>,
    pub legs: i32,
    pub purrs: bool,
    pub human_id: Option<Uuid>,
}
```

CRUD

```rust
use diesel::prelude::*;
use diesel_async::{AsyncPgConnection, RunQueryDsl};

use crate::models::Cat;

pub async fn update_cat(
    conn: &mut AsyncPgConnection,
    cat: &Cat,
) -> QueryResult<Cat> {
    use crate::schema::cats::dsl::*;

    diesel::update(cat)
        .set(purrs.eq(true))
        .returning(Cat::as_returning())
        .get_result(conn)
        .await
}
```

CRUD

```rust
use diesel::prelude::*;
use diesel_async::{AsyncPgConnection, RunQueryDsl};

use crate::models::Cat;

pub async fn get_cat(
    conn: &mut AsyncPgConnection,
    cat_id: uuid::Uuid,
) -> QueryResult<Option<Cat>> {
    use crate::schema::cats::dsl::*;
    cats
        .find(cat_id)
        .select(Cat::as_select())
        .first(conn)
        .await
        .optional()
}
```

CRUD

```rust
use diesel::prelude::*;
use diesel_async::{AsyncPgConnection, RunQueryDsl};

pub async fn delete_impostor_cats(
    conn: &mut AsyncPgConnection,
) -> QueryResult<usize> {
    use crate::schema::cats::dsl::*;

    diesel::delete(
        cats
            .filter(name.eq("Flerken").and(legs.gt(6))),
    )
        .execute(conn)
        .await
}
```

Example & demo - e-wallet app

https://github.com/ian-hon/axum-diesel-example

Demo time!

Q&A

Questions?

Thank You!