

Rust Pro Project: Scalable User Management API

Create a secure, production-grade User Management REST API in Rust using:

- actix-web: Web framework
- diesel: ORM for PostgreSQL
- jsonwebtoken: JWT authentication
- argon2: Password hashing
- dotenvy: Environment variable management
- serde, thiserror, anyhow: for serialization and error handling

File Structure Overview

rust_user_api/

```
|— src/
|   |— main.rs
|   |— config.rs
|   |— db.rs
|   |— models/
|   |   |— user.rs
|   |— schema.rs
|   |— handlers/
|   |   |— user_handler.rs
|   |— routes.rs
|   |— auth.rs
|   |— errors.rs
|— .env
```

└─ Cargo.toml

Cargo.toml

toml

Copy

Edit

[package]

name = "rust_user_api"

version = "0.1.0"

edition = "2021"

[dependencies]

actix-web = "4"

serde = { version = "1.0", features = ["derive"] }

serde_json = "1.0"

diesel = { version = "2.1", features = ["postgres", "r2d2", "chrono"] }

dotenvy = "0.15"

r2d2 = "0.8"

jsonwebtoken = "9.1"

argon2 = "0.5"

thiserror = "1.0"

anyhow = "1.0"

uuid = { version = "1", features = ["v4"] }

chrono = { version = "0.4", features = ["serde"] }

schema.rs — Diesel Schema

rust

Copy

Edit

```
diesel::table! {  
    users (id) {  
        id -> Uuid,  
        username -> Varchar,  
        email -> Varchar,  
        password_hash -> Varchar,  
        created_at -> Timestamp,  
    }  
}
```

models/user.rs — Data Models

rust

Copy

Edit

```
use chrono::NaiveDateTime;  
use serde::{Deserialize, Serialize};  
use uuid::Uuid;
```

```
#[derive(Queryable, Serialize)]
```

```
pub struct User {  
    pub id: Uuid,  
    pub username: String,  
    pub email: String,  
    pub password_hash: String,
```

```
    pub created_at: NaiveDateTime,  
}  

```

```
#[derive(Insertable, Deserialize)]
```

```
#[diesel(table_name = crate::schema::users)]
```

```
pub struct NewUser {  
    pub username: String,  
    pub email: String,  
    pub password_hash: String,  
}
```

db.rs — DB Setup with Diesel

rust

Copy

Edit

```
use diesel::pg::PgConnection;  
use diesel::r2d2::{self, ConnectionManager};  
  
pub type DbPool = r2d2::Pool<ConnectionManager<PgConnection>>;  
  
pub fn init_pool(database_url: &str) -> DbPool {  
    let manager = ConnectionManager::<PgConnection>::new(database_url);  
    r2d2::Pool::builder()  
        .build(manager)  
        .expect("Failed to create pool.")  
}
```

auth.rs — Password & JWT Auth

rust

Copy

Edit

```
use argon2::{self, Config};

use jsonwebtoken::{encode, decode, Header, Validation, EncodingKey, DecodingKey};

use serde::{Serialize, Deserialize};

use std::env;

#[derive(Debug, Serialize, Deserialize)]

struct Claims {

    sub: String,

    exp: usize,

}

pub fn hash_password(password: &str) -> Result<String, argon2::Error> {

    let salt = b"randomsalt";

    let config = Config::default();

    argon2::hash_encoded(password.as_bytes(), salt, &config)

}

pub fn verify_password(hash: &str, password: &str) -> bool {

    argon2::verify_encoded(hash, password.as_bytes()).unwrap_or(false)

}
```

```

pub fn create_jwt(user_id: &str) -> Result<String, jsonwebtoken::errors::Error> {
    let claims = Claims {
        sub: user_id.to_owned(),
        exp: chrono::Utc::now().timestamp() as usize + 3600,
    };

    encode(&Header::default(), &claims,
    &EncodingKey::from_secret(env::var("JWT_SECRET").unwrap().as_bytes()))
}

```

errors.rs — Custom Error Types

rust

Copy

Edit

```
use actix_web::{HttpResponse, ResponseError};
```

```
use thiserror::Error;
```

```
#[derive(Debug, Error)]
```

```
pub enum ApiError {
```

```
    #[error("Database error")]
```

```
    DbError(#[from] diesel::result::Error),
```

```
    #[error("Internal server error")]
```

```
    InternalError,
```

```
    #[error("Invalid credentials")]
```

```
    AuthError,
```

```
}
```

```

impl ResponseError for ApiError {

    fn error_response(&self) -> HttpResponse {

        match self {

            ApiError::DbError(_) => HttpResponse::InternalServerError().body("Database
Error"),

            ApiError::InternalServerError => HttpResponse::InternalServerError().body("Internal
Server Error"),

            ApiError::AuthError => HttpResponse::Unauthorized().body("Invalid
Credentials"),

        }

    }

}

```

handlers/user_handler.rs — Route Handlers

rust

Copy

Edit

```

use actix_web::{web, HttpResponse};

use diesel::prelude::*;

use crate::{schema::users::dsl::*, models::user::*, db::DbPool, auth, errors::ApiError};

pub async fn register_user(

    pool: web::Data<DbPool>,

    user_data: web::Json<NewUser>

) -> Result<HttpResponse, ApiError> {

    let mut user_data = user_data.into_inner();

```

```

user_data.password_hash = auth::hash_password(&user_data.password_hash)

.map_err(|_| ApiError::InternalError)?;

let conn = pool.get()?;
diesel::insert_into(users)
    .values(&user_data)
    .execute(&conn)?;

Ok(HttpResponse::Created().finish())
}

```

```

pub async fn login_user(
    pool: web::Data<DbPool>,
    creds: web::Json<(String, String)>
) -> Result<HttpResponse, ApiError> {
    let conn = pool.get()?;
    let (email_val, password_val) = creds.into_inner();

    let user = users
        .filter(email.eq(email_val.clone()))
        .first::<User>(&conn)?;

    if !auth::verify_password(&user.password_hash, &password_val) {
        return Err(ApiError::AuthError);
    }
}

```



```
    let token = auth::create_jwt(&user.id.to_string())

    .map_err(|_| ApiError::InternalServerError)?;

    Ok(HttpResponse::Ok().json(token))
}
```

routes.rs — API Routing

rust

Copy

Edit

```
use actix_web::web;

use crate::handlers::user_handler::{register_user, login_user};

pub fn config(cfg: &mut web::ServiceConfig) {
    cfg.service(
        web::scope("/api")
            .route("/register", web::post().to(register_user))
            .route("/login", web::post().to(login_user))
    );
}
```

config.rs — Env Loading

rust

Copy

Edit

```
use dotenvy::dotenv;
```

```
use std::env;
```

```
pub fn init_env() {  
    dotenv().ok();  
    env::set_var("RUST_LOG", "debug");  
}
```

main.rs — Main Entrypoint

```
rust
```

Copy

Edit

```
mod config;
```

```
mod db;
```

```
mod schema;
```

```
mod models;
```

```
mod handlers;
```

```
mod routes;
```

```
mod auth;
```

```
mod errors;
```

```
use actix_web::{App, HttpServer, middleware::Logger};
```

```
use db::init_pool;
```

```
#[actix_web::main]
```

```
async fn main() -> std::io::Result<()> {
```

```
    config::init_env();
```

```
env_logger::init();
```

```
let database_url = std::env::var("DATABASE_URL").expect("DATABASE_URL  
must be set");
```

```
let pool = init_pool(&database_url);
```

```
HttpServer::new(move || {  
    App::new()  
        .wrap(Logger::default())  
        .app_data(actix_web::web::Data::new(pool.clone()))  
        .configure(routes::config)  
    })  
    .bind(("127.0.0.1", 8080))?  
    .run()  
    .await  
}
```

.env File

ini

Copy

Edit

```
DATABASE_URL=postgres://user:password@localhost/userdb
```

```
JWT_SECRET=my_super_secret_key
```

Sample curl Requests

bash

Copy

Edit

Register a new user

```
curl -X POST http://localhost:8080/api/register \  
  
-H "Content-Type: application/json" \  
  
-d '{"username":"admin", "email":"admin@example.com",  
"password_hash":"password123"}'
```

Login

```
curl -X POST http://localhost:8080/api/login \  
  
-H "Content-Type: application/json" \  
  
-d '["admin@example.com", "password123"]'
```