// src/main.rs

mod handlers;

mod models;

mod routes;

mod state;

// 库模块导入

use axum::Router;

use tokio::net::TcpListener;

use tracing\_subscriber::fmt;

use sqlx::MySqlPool;

// 分离模块导入

use routes::create\_routes;

use state::AppState;

#[tokio::main]

async fn main() {

// 加载.env文件

dotenv::dotenv().ok();

let db\_url = std::env::var("DATABASE\_URL")

.expect("DATABASE\_URL must be set in .env");

// 初始化日志

fmt::init();

// 创建 MySQL 连接池

let db\_pool = MySqlPool::connect(&db\_url)

.await

.expect("Failed to create MySQL pool");

let state = AppState::new(db\_pool);

// 构建路由(注入状态)

let app = create\_routes().with\_state(state);

// 启动服务器

let listener = TcpListener::bind("0.0.0.0:3000").await.unwrap();

axum::serve(listener, app).await.unwrap();

}

// src/main.rs

mod handlers;

mod models;

mod routes;

mod state;

// 库模块导入

use axum::Router;

use tokio::net::TcpListener;

use tracing\_subscriber::fmt;

use sqlx::MySqlPool;

// 分离模块导入

use routes::create\_routes;

use state::AppState;

#[tokio::main]

async fn main() {

// 加载.env文件

dotenv::dotenv().ok();

let db\_url = std::env::var("DATABASE\_URL")

.expect("DATABASE\_URL must be set in .env");

// 初始化日志

fmt::init();

// 创建 MySQL 连接池

let db\_pool = MySqlPool::connect(&db\_url)

.await

.expect("Failed to create MySQL pool");

let state = AppState::new(db\_pool);

// 构建路由(注入状态)

let app = create\_routes().with\_state(state);

// 启动服务器

let listener = TcpListener::bind("0.0.0.0:3000").await.unwrap();

axum::serve(listener, app).await.unwrap();

}

// src/routes.rs

// 库模块导入

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

use tower\_http::cors::{CorsLayer, Any};

use axum::http::{Method, HeaderName};

// 分离模块导入

use super::handlers;

use crate::state::AppState;

// 构建路由并返回 Router 实例

pub fn create\_routes() -> Router<AppState> {

// CORS 中间件

let cors = CorsLayer::new()

.allow\_origin(Any)

.allow\_methods(vec![Method::GET, Method::POST])

.allow\_headers(vec![HeaderName::from\_static("content-type")]);

Router::new()

.route("/", get(handlers::root))

.route("/login", post(handlers::login))

.layer(cors)

}

// src/handlers.rs

// 库模块导入

use axum::{

http::StatusCode,

Json,

};

use serde::{Deserialize, Serialize};

use axum::extract::State;

use sqlx::MySqlPool;

use std::error::Error;

// 分离模块导入

use crate::{models::{LoginRequest, LoginResponse, User}, state::AppState};

// 根路径处理函数

pub async fn root() -> &'static str {

"Hello, World!"

}

// 登录处理函数

pub async fn login(

State(state): State<AppState>, // 注入状态

Json(payload): Json<LoginRequest>, // 解析为请求结构体

) -> Result<Json<LoginResponse>, StatusCode> {

// 1. 验证账号密码是否正确

match validate\_credentials(&state.db\_pool, &payload.account, &payload.password).await {

Ok(Some(username)) => {

// 2. 验证成功，生成响应结构体

Ok(Json(LoginResponse {

username,

}))

}

Ok(None) => {

// 3. 验证失败，返回 401

Err(StatusCode::UNAUTHORIZED)

}

Err(\_) => {

// 4. 数据库错误，返回 500

Err(StatusCode::INTERNAL\_SERVER\_ERROR)

}

}

}

async fn validate\_credentials(

db\_pool: &MySqlPool,

account: &str,

password: &str,

) -> Result<Option<String>, Box<dyn Error>> {

// 从数据库中查询对应用户账号的用户信息

let user = sqlx::query\_as::<\_, User>(

"SELECT \* FROM user\_info WHERE account = ?"

)

.bind(account)

.fetch\_one(db\_pool)

.await;

match user {

Ok(user) => {

// 验证密码

if let Some(user\_password) = &user.password {

if password == user\_password {

if let Some(username) = user.username {

return Ok(Some(username));

}

return Err("用户名未找到".into());

}

return Err("密码不匹配".into());

}

Err("用户密码未设置".into())

}

Err(sqlx::Error::RowNotFound) => Ok(None),

Err(e) => Err(e.into()),

}

}

// src/models.rs

// 库模块导入

use serde::{Deserialize, Serialize};

use sqlx::FromRow;

// 用户表模型

#[derive(Debug, Deserialize, Serialize, FromRow)]

pub struct User {

pub account: String,

pub password: Option<String>,

pub username: Option<String>,

}

// 登录请求模型

#[derive(Deserialize)]

pub struct LoginRequest {

pub account: String,

pub password: String,

}

// 登录响应模型

#[derive(Serialize)]

pub struct LoginResponse {

pub username: String,

}