Отчет по лабораторной работе №4 по диспиплине “Парадигмы и конструкции языков программирования”

errors/mod.rs

use actix\_web::{

error, get,

http::{header::ContentType, StatusCode},

App, HttpResponse,

};

use base64::display;

use derive\_more::{Display, Error};

#[derive(Debug, Display, Error)]

pub enum Errors {

#[display(fmt="internal error")]

InternalError,

#[display(fmt="bad request")]

BadClientData,

#[display(fmt="timeout")]

Timeout,

#[display(fmt="not found")]

NotFound,

#[display(fmt="unauthorized")]

Unauthorized,

#[display(fmt="forbidden")]

Forbidden,

}

impl error::ResponseError for Errors {

fn error\_response(&self) -> HttpResponse {

HttpResponse::build(self.status\_code())

.insert\_header(ContentType::html())

.body(self.to\_string())

}

fn status\_code(&self) -> StatusCode {

match \*self {

Errors::InternalError => StatusCode::INTERNAL\_SERVER\_ERROR,

Errors::BadClientData => StatusCode::BAD\_REQUEST,

Errors::Timeout => StatusCode::GATEWAY\_TIMEOUT,

Errors::NotFound => StatusCode::NOT\_FOUND,

Errors::Unauthorized => StatusCode::UNAUTHORIZED,

Errors::Forbidden => StatusCode::FORBIDDEN,

}

}

}

models/entities/users.rs

use sea\_orm::entity::prelude::\*;

use std::cmp::{*Eq*, *PartialEq*};

use uuid::Uuid;

use chrono::NaiveDate;

#[derive(Clone, Debug, PartialEq, Eq, DeriveEntityModel)]

#[sea\_orm(table\_name = "users")]

pub struct Model {

#[sea\_orm(primary\_key)]

pub **uuid**: Uuid,

pub **username**: String,

pub **email**: String,

pub **password\_hash**: String,

pub **registration\_date**: NaiveDate,

pub **is\_admin**: *bool*,

pub **is\_confirmed**: *bool*

}

#[derive(Copy, Clone, Debug, EnumIter, DeriveRelation)]

pub enum Relation {}

impl *ActiveModelBehavior* for ActiveModel {}

models/mod.rs

use sea\_orm::{Database, DbErr, *ConnectionTrait*, DbBackend, Statement};

mod migrator;

pub mod entities;

use migrator::Migrator;

use sea\_orm\_migration::prelude::\*;

pub const DATABASE\_URL: &*str* = "postgres://postgres:256128@0.0.0.0:5435";

pub const DATABASE\_NAME: &*str* = "users";

pub async fn run() -> Result<(), DbErr> {

let db = Database::connect(DATABASE\_URL).await?;

let db = &match db.get\_database\_backend() {

DbBackend::Postgres => {

db.execute(Statement::from\_string(

db.get\_database\_backend(),

format!("DROP DATABASE IF EXISTS \"{}\";", DATABASE\_NAME),

))

.await?;

db.execute(Statement::from\_string(

db.get\_database\_backend(),

format!("CREATE DATABASE \"{}\";", DATABASE\_NAME),

))

.await?;

let url = format!("{}/{}", DATABASE\_URL, DATABASE\_NAME);

Database::connect(&url).await?

},

DbBackend::MySql => {

db.execute(Statement::from\_string(

db.get\_database\_backend(),

format!("CREATE DATABASE IF NOT EXISTS `{}`;", DATABASE\_NAME),

))

.await?;

let url = format!("{}/{}", DATABASE\_URL, DATABASE\_NAME);

Database::connect(&url).await?

},

DbBackend::Sqlite => db,

};

let schema\_manager = SchemaManager::new(db);

Migrator::refresh(db).await?;

assert!(schema\_manager.has\_table("users").await?);

Ok(())

}

schemas/token.rs

use chrono::Utc;

use jsonwebtoken::{EncodingKey, Header};

use serde::{*Deserialize*, *Serialize*};

use uuid::Uuid;

use crate::models::entities::{users, users::Entity as Users};

pub static KEY: [*u8*; 63] = \*include\_bytes!("../secret.key");

static ONE\_WEEK: *i64* = 60 \* 60 \* 24 \* 7;

#[derive(Serialize, Deserialize)]

pub struct UserToken {

pub **iat**: *i64*,

pub **exp**: *i64*,

pub **user**: String

}

#[derive(Serialize, Deserialize)]

pub struct TokenBodyResponse {

pub **token**: String,

pub **token\_type**: String

}

impl UserToken {

pub fn generate\_token(user\_id: Uuid) -> String {

let max\_age: *i64* = ONE\_WEEK;

let now = Utc::now().timestamp\_nanos\_opt().unwrap();

let payload = UserToken {

**iat**: now,

**exp**: now + max\_age,

**user**: user\_id.to\_string()

};

jsonwebtoken::encode(

&Header::default(),

&payload,

&EncodingKey::from\_secret(&KEY)

)

.unwrap()

}

}

schemas/users.rs

use serde::{*Deserialize*, *Serialize*};

use uuid::Uuid;

use datetime::LocalDateTime;

use chrono::NaiveDate;

use crate::models::entities::{users, users::Entity as Users};

#[derive(Deserialize)]

pub struct UserCreate {

pub **username**: String,

pub **email**: String,

pub **password**: String

}

#[derive(Serialize)]

pub struct UserGet {

pub **uuid**: Uuid,

pub **username**: String,

pub **email**: String,

pub **password\_hash**: String,

pub **registration\_date**: NaiveDate,

pub **is\_admin**: *bool*,

pub **is\_confirmed**: *bool*

}

impl UserGet {

pub fn new(uuid: Uuid, username: String, email: String, password\_hash: String, registration\_date: NaiveDate, is\_admin: *bool*, is\_confirmed: *bool*) -> Self {

UserGet {

**uuid**,

**username**,

**email**,

**password\_hash**,

**registration\_date**,

**is\_admin**,

**is\_confirmed**

}

}

pub fn from\_model(user: users::Model) -> Self {

UserGet {

**uuid**: user.**uuid**,

**username**: user.**username**,

**email**: user.**email**,

**password\_hash**: user.**password\_hash**,

**registration\_date**: user.**registration\_date**,

**is\_admin**: user.**is\_admin**,

**is\_confirmed**: user.**is\_confirmed**,

}

}

}

#[derive(Debug, Deserialize)]

pub struct Params {

pub **page**: Option<*u64*>,

pub **page\_size**: Option<*u64*>,

}

#[derive(Debug, Serialize, Deserialize)]

pub struct UserLogin {

pub **username\_or\_password**: String,

pub **password**: String

}

service/users.rs

use actix\_web::HttpRequest;

use std::str::*FromStr*;

use uuid::Uuid;

use std::vec::Vec;

use crate::errors::Errors;

use crate::models::entities::{users, users::Entity as Users};

use crate::schemas::token::{TokenBodyResponse, UserToken};

use crate::schemas::users::{UserGet, UserLogin};

use crate::utils::hash;

use crate::utils::token::{is\_auth\_header\_valid, decode\_token};

use crate::constants;

use crate::utils::verify\_password;

use sea\_orm::\*;

pub struct Query;

impl Query {

pub async fn create\_user(db: &DbConn, username: &String, email: &String, password: &String) -> String {

let user\_id: Uuid = Uuid::new\_v4();

let hashed\_password: String = hash(password.as\_bytes()).await;

let new\_user = users::ActiveModel {

**uuid**: ActiveValue::Set(user\_id),

**username**: ActiveValue::Set(username.to\_owned()),

**email**: ActiveValue::Set(email.to\_owned()),

**password\_hash**: ActiveValue::Set(hashed\_password),

..*Default*::default()

};

let user\_res = Users::insert(new\_user).exec(db).await.unwrap();

match user\_res {

\_ => {format!("inserted")},

}

}

pub async fn get\_all\_users(db: &DbConn, page: *u64*, page\_size: *u64*) -> Result<(Vec<UserGet>, *u64*), DbErr> {

let paginator = Users::find()

.order\_by\_asc(users::Column::Uuid)

*// .into\_json()*

.paginate(db, page\_size);

let num\_pages = paginator.num\_pages().await?;

paginator.fetch\_page(page - 1).await.map(|p| {

let mut users: Vec<UserGet> = Vec::new();

for user in p {

users.push(UserGet::from\_model(user));

}

(users, num\_pages)

})

}

pub async fn get\_one\_user(db: &DbConn, user\_id: Uuid) -> Result<UserGet, DbErr> {

let user = Users::find\_by\_id(user\_id).one(db).await;

match user {

Ok(user\_model) => match user\_model {

Some(u) => Ok(UserGet::from\_model(u)),

None => Err(DbErr::RecordNotFound(String::from("Record not found")))

},

Err(err\_type) => Err(err\_type)

}

}

pub async fn login(db: &DbConn, login: UserLogin) -> Result<TokenBodyResponse, Errors> {

let user\_got = Users::find().filter(

Condition::any()

.add(users::Column::Email.eq(&login.**username\_or\_password**))

.add(users::Column::Username.eq(&login.**username\_or\_password**))

).one(db).await;

match user\_got {

Ok(user) => {

match user {

Some(user) => {

*// match verify\_password(login.password.as\_str(), password\_hash.as\_bytes()) {*

match verify\_password(login.**password**.as\_bytes(), user.**password\_hash**.as\_ref()) {

Ok(\_) => {

let token = UserToken::generate\_token(user.**uuid**);

return Ok(TokenBodyResponse{**token**: token, **token\_type**: String::from("bearer")})

},

Err(err) => {

println!("login error {}", err.to\_string());

Err(Errors::BadClientData)

}

}

},

None => return Err(Errors::NotFound)

}

}

Err(\_) => Err(Errors::NotFound)

}

}

pub async fn get\_current\_user(db: &DbConn, req: HttpRequest) -> Result<UserGet, Errors> {

if let Some(auth\_header) = req.headers().get(constants::AUTHORIZATION) {

if let Ok(auth\_str) = auth\_header.to\_str() {

if is\_auth\_header\_valid(auth\_header) {

let token = auth\_str[6..auth\_str.len()].trim();

if let Ok(token\_data) = decode\_token(&token.to\_string()) {

match Query::get\_one\_user(db, Uuid::from\_str(token\_data.**claims**.**user**.as\_str()).unwrap()).await {

Ok(login\_info) => return Ok(login\_info),

Err(\_) => return Err(Errors::NotFound)

}

} else {

return Err(Errors::BadClientData);

}

} else {

return Err(Errors::BadClientData);

}

} else {

return Err(Errors::BadClientData);

}

} else {

return Err(Errors::BadClientData);

}

}

}

utils/password.rs

use argon2::{

password\_hash::{rand\_core::OsRng, PasswordHash, *PasswordHasher*, *PasswordVerifier*, SaltString},

Argon2

};

#[tracing::instrument(name = "Hashing user password", skip(password))]

pub async fn hash(password: &[*u8*]) -> String {

let salt = SaltString::generate(&mut OsRng);

Argon2::default()

.hash\_password(password, &salt)

.expect("Unable to hash password")

.to\_string()

}

#[tracing::instrument(name = "Verifying user password", skip(password, hash))]

pub fn verify\_password(password: &[*u8*], hash: &*str*) -> Result<(), argon2::password\_hash::Error> {

let hash = PasswordHash::new(&hash)

.map\_err(|e| println!("hash error: {}", e)).unwrap();

let res = Argon2::default().verify\_password(password, &hash);

match res {

Ok(\_) => Ok(()),

Err(err) => {

println!("verify error {}", err.to\_string());

Err(argon2::password\_hash::Error::Crypto)

}

}

}

utils/token.rs

use uuid::Uuid;

use std::str::*FromStr*;

use sea\_orm::DbErr;

use actix\_web::{web, http::header::HeaderValue};

use jsonwebtoken::{DecodingKey, TokenData, Validation};

use sea\_orm::\*;

use crate::models::entities::{users, users::Entity as Users};

use crate::schemas::{token::{UserToken, KEY}, users::UserGet};

use crate::errors::Errors;

pub fn decode\_token(token: &String) -> jsonwebtoken::errors::Result<TokenData<UserToken>> {

jsonwebtoken::decode::<UserToken>(

token,

&DecodingKey::from\_secret(&KEY),

&Validation::default()

)

}

pub fn is\_auth\_header\_valid(authen\_header: &HeaderValue) -> *bool* {

if let Ok(authen\_str) = authen\_header.to\_str() {

return authen\_str.starts\_with("bearer") || authen\_str.starts\_with("Bearer");

}

return false;

}

pub async fn get\_current\_user(db: &DbConn, user\_token: &String) -> Result<UserGet, Errors> {

match decode\_token(user\_token) {

Ok(token\_data) => {

if token\_data.**claims**.**exp** > chrono::Utc::now().timestamp\_nanos\_opt().unwrap() {

return Err(Errors::Unauthorized)

}

let user\_id = Uuid::from\_str(token\_data.**claims**.**user**.as\_str()).unwrap();

let user = Users::find\_by\_id(user\_id).one(db).await;

match user {

Ok(user\_model) => match user\_model {

Some(u) => Ok(UserGet::from\_model(u)),

None => Err(Errors::NotFound)

},

Err(\_) => Err(Errors::NotFound)

}

},

Err(\_) => Err(Errors::BadClientData)

}

}

constants.rs

*// Messages*

pub const MESSAGE\_OK: &*str* = "ok";

pub const MESSAGE\_CAN\_NOT\_FETCH\_DATA: &*str* = "Can not fetch data";

pub const MESSAGE\_CAN\_NOT\_INSERT\_DATA: &*str* = "Can not insert data";

pub const MESSAGE\_CAN\_NOT\_UPDATE\_DATA: &*str* = "Can not update data";

pub const MESSAGE\_CAN\_NOT\_DELETE\_DATA: &*str* = "Can not delete data";

pub const MESSAGE\_SIGNUP\_SUCCESS: &*str* = "Signup successfully";

pub const MESSAGE\_SIGNUP\_FAILED: &*str* = "Error while signing up, please try again";

pub const MESSAGE\_LOGIN\_SUCCESS: &*str* = "Login successfully";

pub const MESSAGE\_LOGIN\_FAILED: &*str* = "Wrong username or password, please try again";

pub const MESSAGE\_USER\_NOT\_FOUND: &*str* = "User not found, please signup";

pub const MESSAGE\_LOGOUT\_SUCCESS: &*str* = "Logout successfully";

pub const MESSAGE\_PROCESS\_TOKEN\_ERROR: &*str* = "Error while processing token";

pub const MESSAGE\_INVALID\_TOKEN: &*str* = "Invalid token, please login again";

pub const MESSAGE\_INTERNAL\_SERVER\_ERROR: &*str* = "Internal Server Error";

*// Bad request messages*

pub const MESSAGE\_TOKEN\_MISSING: &*str* = "Token is missing";

pub const MESSAGE\_BAD\_REQUEST: &*str* = "Bad Request";

*// Headers*

pub const AUTHORIZATION: &*str* = "Authorization";

*// Misc*

pub const EMPTY: &*str* = "";

*// ignore routes*

pub const IGNORE\_ROUTES: [&*str*; 3] = ["/api/ping", "/api/auth/signup", "/api/auth/login"];

*// Default number of items per page*

pub const DEFAULT\_PER\_PAGE: *i64* = 10;

*// Default page number*

pub const DEFAULT\_PAGE\_NUM: *i64* = 1;

pub const EMPTY\_STR: &*str* = "";

*//Session key*

pub const SESSION\_SERVER\_PUBLIC\_KEY: &*str* = "spk";

pub const SESSION\_CLIENT\_PUBLIC\_KEY: &*str* = "cpk";

handlers/mod.rs

use sea\_orm::DatabaseConnection;

pub mod root;

pub mod users;

#[derive(Debug, Clone)]

pub struct AppState {

pub **conn**: DatabaseConnection,

}

handlers/root.rs

use actix\_web::{get, http::StatusCode, post, web, App, HttpRequest, HttpResponse, *Responder*};

#[get("/")]

async fn index() -> impl *Responder* {

HttpResponse::Ok().body("Hello, world!")

}

pub async fn page\_not\_found() -> impl *Responder* {

HttpResponse::NotFound().body("Error 404. Page not found.")

}

handlers/users.rs

use std::fmt::format;

use std::str::*FromStr*;

use actix\_web::{get, http::StatusCode, post, web, HttpResponse, HttpRequest, *Responder*, Result, Error};

use sea\_orm::\*;

use serde\_json::json;

use uuid::{uuid, Uuid};

use crate::schemas::users;

use crate::service::users::Query;

use super::AppState;

use crate::errors::Errors;

#[post("/")]

async fn create\_user(data: web::Json<users::UserCreate>, state: web::Data<AppState>) -> impl *Responder* {

let conn = &state.**conn**;

let username: String = data.**username**.clone();

let email: String = data.**email**.clone();

let password: String = data.**password**.clone();

Query::create\_user(conn , &username, &email, &password).await;

format!("created")

}

#[get("/")]

async fn get\_all\_users(req: HttpRequest, state: web::Data<AppState>) -> web::Json<serde\_json::Value> {

let conn = &state.**conn**;

let params = web::Query::<users::Params>::from\_query(req.query\_string()).unwrap();

let page = params.**page**.unwrap\_or(1);

let page\_size = params.**page\_size**.unwrap\_or(5);

let (users, num\_pages) = Query::get\_all\_users(conn, page, page\_size)

.await

.expect("Cannot find users in page");

web::Json(json!({

"users": users,

"num\_pages": num\_pages

}))

}

#[get("/{id}")]

async fn get\_user(path: web::Path<String>, state: web::Data<AppState>) -> Result<HttpResponse, Error> {

let conn = &state.**conn**;

let got\_id = Uuid::from\_str(path.into\_inner().as\_str());

match got\_id {

Ok(id) => {

let user = Query::get\_one\_user(conn, id).await;

match user {

Ok(user) => Ok(HttpResponse::Ok().json(user)),

Err(e) => {

println!("{:?}", e);

Ok(HttpResponse::NotFound().json("User not found"))

}

}

},

Err(\_) => Ok(HttpResponse::BadRequest().json("Invalid UUID"))

}

}

#[post("/login")]

async fn login(data: web::Json<users::UserLogin>, state: web::Data<AppState>) -> Result<HttpResponse, Errors> {

let conn = &state.**conn**;

match Query::login(conn, data.into\_inner()).await {

Ok(token) => Ok(HttpResponse::Ok().json(token)),

Err(\_) => Err(Errors::BadClientData)

}

}

#[get("/me")]

async fn get\_user\_me(req: HttpRequest, state: web::Data<AppState>) -> Result<HttpResponse, Errors> {

let conn = &state.**conn**;

match Query::get\_current\_user(conn, req).await {

Ok(user) => Ok(HttpResponse::Ok().json(user)),

Err(e) => Err(Errors::NotFound)

}

}

main.rs

use futures::executor::block\_on;

use actix\_web::{get, post, web, App, HttpRequest, HttpResponse, HttpServer, *Responder*};

use sea\_orm::{Database, DatabaseConnection};

pub mod handlers;

pub mod models;

pub mod schemas;

pub mod service;

pub mod utils;

pub mod middleware;

pub mod errors;

pub mod constants;

use handlers::{root, users, AppState};

use models::{DATABASE\_URL, DATABASE\_NAME};

#[actix\_web::main]

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

std::env::set\_var("RUST\_LOG", "info");

std::env::set\_var("RUST\_BACKTRACE", "1");

*// if let Err(err) = block\_on(models::run()) {*

*// panic!("{}", err);*

*// }*

let db\_url = format!("{DATABASE\_URL}/{DATABASE\_NAME}");

let conn = Database::connect(db\_url).await.unwrap();

let state = AppState {**conn**: conn};

println!("Created DB");

HttpServer::new(move || {

App::new()

.app\_data(web::Data::new(state.clone()))

.service(root::index)

.service(

web::scope("/users")

.service(users::create\_user)

.service(users::get\_all\_users)

.service(users::get\_user)

.service(users::login)

.service(users::get\_user\_me)

)

.default\_service(

web::route().to(root::page\_not\_found)

)

})

.bind(("127.0.0.1", 8083))?

.run()

.await

}