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

errors/mod.rs

```rust
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

```rust
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 {}
```

```rust
impl ActiveModelBehavior for ActiveModel {}
```

models/mod.rs

```rust
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
```

```rust
}



#[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,
```

```rust
    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;
```

```rust
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)
        })
    }
```

```rust
    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_passwor
d))

        ).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
};
```

```rust
#[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()

    )

}
```

```rust
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
```

```rust
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";
```

```rust
pub const SESSION_CLIENT_PUBLIC_KEY: &str = "cpk";
```

handlers/mod.rs

```rust
use sea_orm::DatabaseConnection;



pub mod root;

pub mod users;



#[derive(Debug, Clone)]

pub struct AppState {

    pub conn: DatabaseConnection,

}
```

handlers/root.rs

```rust
use actix_web::{get, http::StatusCode, post, web, App, HttpRequest,
HttpResponse, Responder};




#[get("/")]

async fn index() -> impl Responder {

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

}
```

```rust
pub async fn page_not_found() -> impl Responder {
    HttpResponse::NotFound().body("Error 404. Page not found.")
}
```

handlers/users.rs

```rust
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();
```

```rust
    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

    }))

}
```

```rust
#[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;
```

```rust
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
}
```