САНКТ-ПЕТЕРБУРГСКИЙ НАЦИОНАЛЬНЫЙ ИССЛЕДОВАТЕЛЬСКИЙ УНИВЕРСИТЕТ ИТМО

Дисциплина: Бэк-энд разработка

Отчет

Лабораторная работа 1

Выполнила:

Фролова Кристина

Группа К3339

Проверил:
Добряков Д. И.

Санкт-Петербург

2025 г.

Задача

Нужно написать свой boilerplate на express + TypeORM + typescript.

Должно быть явное разделение на:

- модели,
- контроллеры,
- роуты.

Должна быть реализована базовая пользовательская модель, jwt-авторизация и конфигурация через переменные среды.

Ход работы

Бойлерплейт создавался с использованием фреймворка tsoa. Фреймворк позволяет автоматически генерировать роуты, документацию, подвязывать авторизацию. Получившаяся структура проекта представлена на рисунке 1.

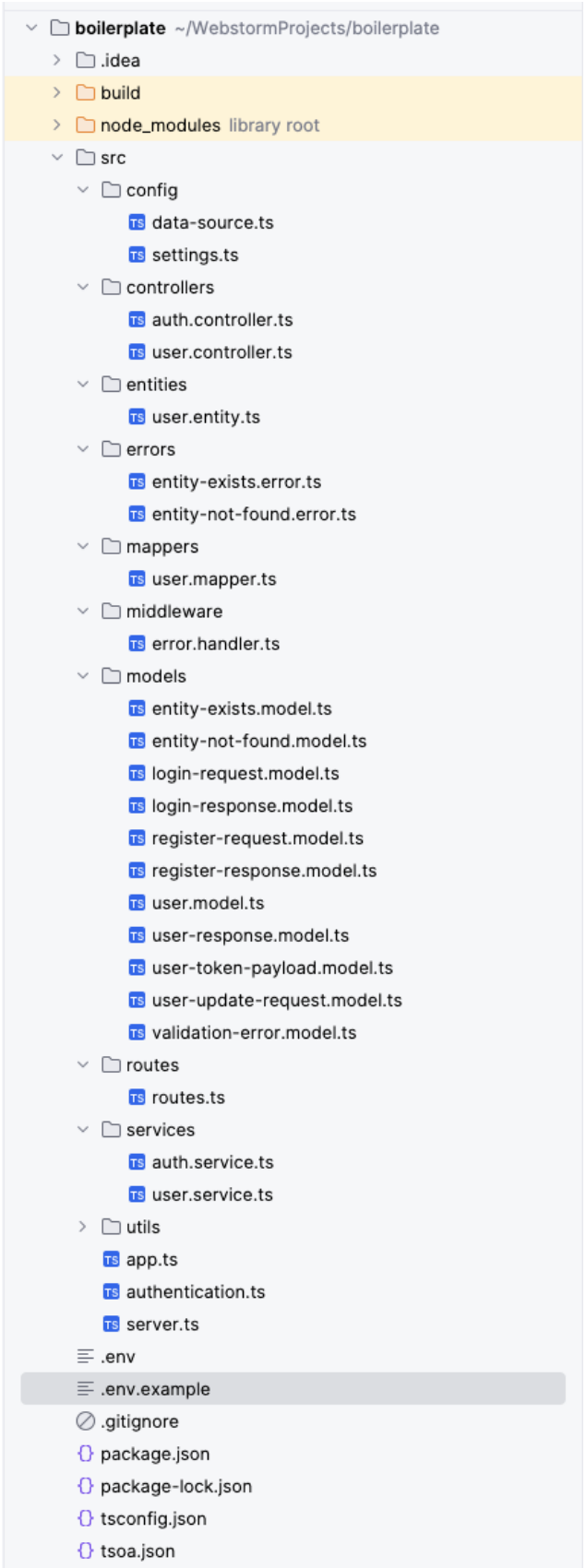

Рисунок 1 – Структура проекта

На рисунке 2 продемонстрирован UserController, на примере которого можно заметить особенности работы с фреймворком tsoa. Для защиты эндпоинта используется аннотация @Security, а для получения авторизованного пользователя из контекста достаточно добавить @Request() в параметры метода.

```
@Route("users") Show usages
@Tags("User")
export class UserController extends Controller {

    @Get() Show usages
    @SuccessResponse("200", "Ok")
    @Security("jwt")
    @Response<EntityNotFoundErrorResponse>(404, "Entity not found")
    public async getUsers(): Promise<UserResponseModel[]> {
        const users: User[] = await userService.getUsers();
        return users.map(toUserResponseModel)
    };

    @Get("me") Show usages
    @SuccessResponse("200", "Ok")
    @Security("jwt")
    @Response<EntityNotFoundErrorResponse>(404, "Entity not found")
    public async getUser(@Request() request: any): Promise<UserResponseModel> {
        const retrievedUser :any = await userService.getUserById(request.user.id);
        return toUserResponseModel(retrievedUser)
    };

    @Get("mail") Show usages
    @SuccessResponse("200", "Ok")
    @Response<EntityNotFoundErrorResponse>(404, "Entity not found")
    @Security("jwt")
    public async getUserByMail(@Query() mail: string): Promise<UserResponseModel> {
        const user :any = await userService.getUserByMail(mail);
        return toUserResponseModel(user);
    };

    @Get("{userId}") Show usages
    @SuccessResponse("200", "Ok")
    @Response<EntityNotFoundErrorResponse>(404, "Entity not found")
    @Security("jwt")
    public async getUserById(@Path() userId: number): Promise<UserResponseModel> {
        const user :any = await userService.getUserById(userId);
        return toUserResponseModel(user);
    };

    @Put("{userId}") Show usages
    @SuccessResponse("200", "Updated")
    @Security("jwt")
    @Response<EntityNotFoundErrorResponse>(404, "Entity not found")
    public async updateUser(
        @Path() userId: number,
        @Body() body: UpdateUserRequestModel
    ): Promise<UserResponseModel> {
        const updated :User = await userService.update(userId, body);
        return toUserResponseModel(updated);
    }

    @Delete("{userId}") Show usages
    @SuccessResponse("204", "Deleted")
    @Security("jwt")
    public async deleteUser(@Path() userId: number): Promise<void> {
```

Рисунок 2 – UserController

На рисунке 3 продемонстрирован AuthService, в котором реализована логика входа и регистрации.

```
class AuthService {  Show usages

    public async login(loginRequest: LoginRequestDto) : Promise<{ accessToken: string }>  {  Show usages
        const user : any  = await userService.getUserByMail(loginRequest.mail);
        const userPassword : any  = user.password;
        if (!checkPassword(userPassword, loginRequest.password)) {
            throw new BadRequest("Email or password is incorrect");
        }
        const userTokenPayload: UserTokenPayload = {
            id: user.id,
            mail: user.mail,
            firstName: user.firstName,
            lastName: user.lastName,
        };

        const accessToken : string  = jwt.sign(
            { user: userTokenPayload },
            SETTINGS.JWT_SECRET_KEY,
            {
                expiresIn: SETTINGS.JWT_ACCESS_TOKEN_LIFETIME,
            }
        );
        return {accessToken}
    }

    public async register(registerRequest: RegisterRequestModel) : Promise<User>  {  Show usages
        if (await userService.existsByEmail(registerRequest.mail)) {
            throw new EntityExistsError(UserEntity, registerRequest.mail, "mail");
        }
        const userEntity = new UserEntity();
        userEntity.mail = registerRequest.mail;
        userEntity.password = hashPassword(registerRequest.password);
        userEntity.firstName = registerRequest.firstName;
        userEntity.lastName = registerRequest.lastName;
        const savedUser : any  = await userService.create(userEntity);
        return toUser(savedUser);
    }
}
```

Рисунок 3 – AuthService

На рисунке 4 продемонстрирован файл settings.ts, в котором получают переменные окружения из файла .env.

```ts
import dotenv from 'dotenv';

dotenv.config();

class Settings {  Show usages
    // db connection settings
    DB_HOST : string  = process.env.DB_HOST || 'localhost';
    DB_PORT : number  = parseInt(process.env.DB_PORT || '15432');
    DB_NAME : string  = process.env.DB_NAME || 'maindb';
    DB_USER : string  = process.env.DB_USER || 'maindb';
    DB_PASSWORD : string  = process.env.DB_PASSWORD || 'maindb';
    DB_ENTITIES : string  = process.env.DB_ENTITIES || 'build/entities/*.entity.js';

    JWT_SECRET_KEY : string  = process.env.JWT_SECRET_KEY || 'secret';
    JWT_TOKEN_TYPE : string  = process.env.JWT_TOKEN_TYPE || 'Bearer';
    JWT_ACCESS_TOKEN_LIFETIME: number =
        parseInt(process.env.JWT_ACCESS_TOKEN_LIFETIME || '300');
}

const SETTINGS = new Settings();

export default SETTINGS;  Show usages
```

Рисунок 4 – settings.ts

На рисунке 5 продемонстрирован Swagger, который автоматически генерируется фреймворком tsoa, на основе контроллеров и моделей.

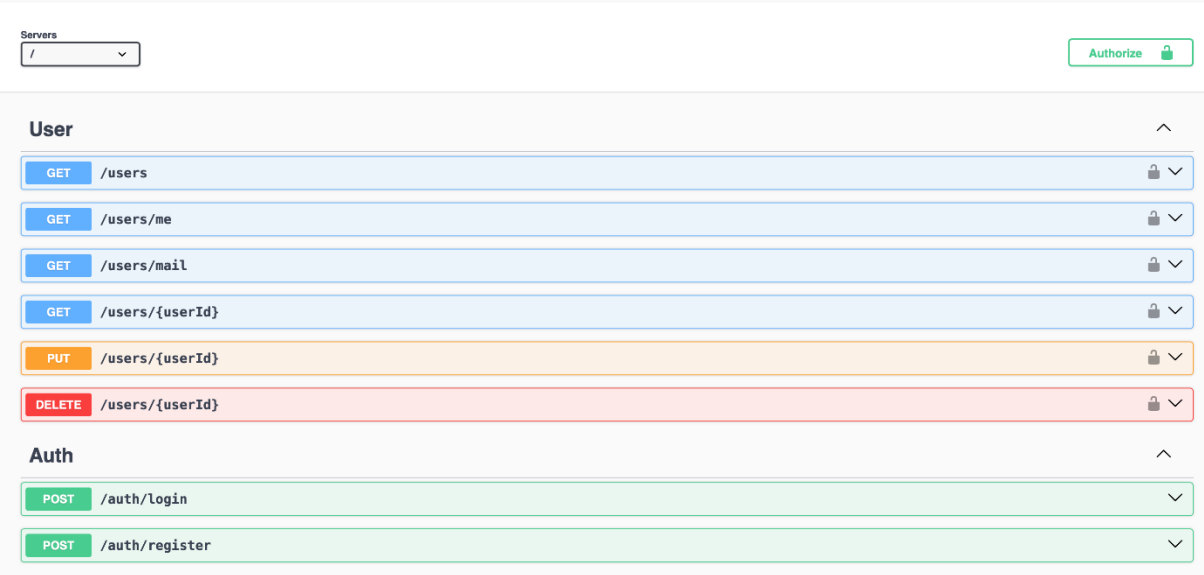

Рисунок 5 – Swagger

Вывод

В рамках работы был создан шаблон проекта, в основу которого закладывают express + TypeORM + typescript + tsoa.