

**САНКТ-ПЕТЕРБУРГСКИЙ НАЦИОНАЛЬНЫЙ  
ИССЛЕДОВАТЕЛЬСКИЙ УНИВЕРСИТЕТ ИТМО**

**Дисциплина:** Бек-энд разработка

**Отчет**

**Лабораторная работа №1**

Выполнила: Злотникова К.А.

Группа: К33392

Проверил:

Добряков Д. И.

Санкт-Петербург

2024 г.

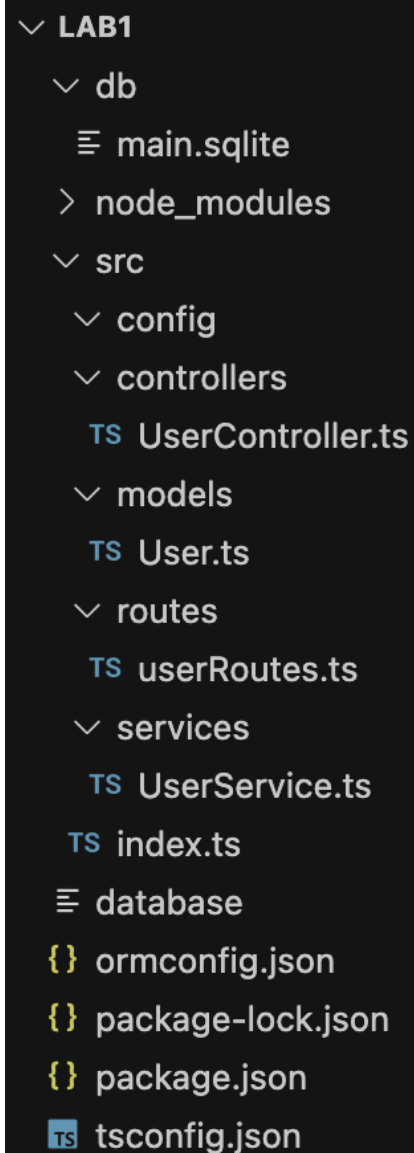
## Задача

Создать стартовый шаблон проекта (boilerplate) на основе Express, TypeORM и TypeScript. Проект должен иметь четкую структуру, разделенную на модели, контроллеры, роуты и сервисы, используя паттерн "репозиторий".

## Ход работы

### 1. Создание структуры проекта

Первым делом создаем структуру папок и файлов (Рисунок 1).



```

└─ LAB1
  └─ db
    └─ main.sqlite
  └─ node_modules
  └─ src
    └─ config
    └─ controllers
      └─ TS UserController.ts
    └─ models
      └─ TS User.ts
    └─ routes
      └─ TS userRoutes.ts
    └─ services
      └─ TS UserService.ts
  └─ TS index.ts
  └─ database
  └─ {} ormconfig.json
  └─ {} package-lock.json
  └─ {} package.json
  └─ TS tsconfig.json

```

Рисунок 1 - структура проекта

## 2. Инициализация проекта и установка зависимостей

Создаем `package.json` с помощью команды `npm init -y`, затем устанавливаем все нужные зависимости:

- `npm init -y`
- `npm install express typeorm reflect-metadata sqlite3`
- `npm install --save-dev typescript ts-node ts-node-dev @types/express @types/node`

Express: для создания сервера и обработки HTTP-запросов.

TypeORM: для работы с базой данных.

reflect-metadata: для декораторов TypeScript.

SQLite: как легковесная база данных.

TypeScript: основной язык разработки.

ts-node-dev: для запуска TypeScript кода без компиляции.

## 3. Настройка TypeScript

Создаем файл `tsconfig.json`, чтобы настроить TypeScript под наши нужды:

```
{  
  
  "compilerOptions": {  
  
    "target": "ES2020",  
  
    "module": "commonjs",  
  
    "outDir": "./dist",  
  
    "rootDir": "./src",  
  
    "strict": true,  
  
    "esModuleInterop": true,  
  
    "experimentalDecorators": true,  
  
    "emitDecoratorMetadata": true,  
  
    "skipLibCheck": true  
  }  
}
```

```
},  
  
"include": ["src/**/*"],  
  
"exclude": ["node_modules"]  
}
```

Эти настройки помогают TypeScript правильно компилировать наш код и поддерживать все нужные фичи, такие как декораторы.

## 4. Конфигурация TypeORM

Создаем файл `ormconfig.json` для настройки TypeORM:

```
{  
  
  "type": "sqlite",  
  
  "database": "./database",  
  
  "synchronize": true,  
  
  "logging": false,  
  
  "entities": ["src/models/**/*.ts"],  
  
  "migrations": ["src/migration/**/*.ts"],  
  
  "subscribers": ["src/subscriber/**/*.ts"],  
  
  "cli": {  
  
    "entitiesDir": "src/models",  
  
    "migrationsDir": "src/migration",  
  
    "subscribersDir": "src/subscriber"  
  }  
}
```

Эта конфигурация указывает TypeORM использовать SQLite и где искать наши сущности (модели).

## 5. Создание модели пользователя

Модель пользователя описывает, как будет выглядеть пользователь в базе данных. Создаем файл `src/models/User.ts`:

```
import { Entity, PrimaryGeneratedColumn, Column } from 'typeorm';

@Entity()

export class User {

  @PrimaryGeneratedColumn()

  id!: number;

  @Column()

  name!: string;

  @Column()

  email!: string;

}
```

Здесь мы используем декораторы `@Entity`, `@PrimaryGeneratedColumn` и `@Column`, чтобы указать, что это таблица в базе данных и какие у нее будут колонки.

## 6. Создание сервиса для работы с пользователями

Сервисы содержат логику работы с моделями. Создаем файл `src/services/UserService.ts`:

```
import { getRepository } from 'typeorm';

import { User } from '../models/User';

export class UserService {

  public async getAllUsers() {

    const userRepository = getRepository(User);

    return userRepository.find();

  }

}
```

```

    }

    public async createUser(user: Partial<User>) {

        const userRepository = getRepository(User);

        const newUser = userRepository.create(user);

        return userRepository.save(newUser);

    }

}

```

Этот сервис использует репозиторий TypeORM для взаимодействия с базой данных.

## 7. Создание контроллера для обработки запросов пользователей

Контроллеры отвечают за обработку HTTP-запросов. Создаем файл `src/controllers/UserController.ts`:

```

import { Request, Response } from 'express';

import { UserService } from '../services/UserService';

const userService = new UserService();

export const getAllUsers = async (req: Request, res: Response) => {

    const users = await userService.getAllUsers();

    res.json(users);

};

export const createUser = async (req: Request, res: Response) => {

    const user = await userService.createUser(req.body);

    res.status(201).json(user);

};

```

Контроллеры получают запросы, передают данные в сервисы и возвращают ответы клиентам.

## 8. Настройка роутов

Роуты связывают URL с соответствующими контроллерами. Создаем файл `src/routes/userRoutes.ts`:

```
import { Router } from 'express';

import { getAllUsers, createUser } from '../controllers/UserController';

const router = Router();

router.get('/users', getAllUsers);

router.post('/users', createUser);

export default router;
```

Роуты определяют, какие функции будут вызываться для конкретных HTTP-запросов.

## 9. Настройка главного файла приложения

Главный файл запускает сервер и настраивает основные middlewares. Создаем файл `src/index.ts`:

```
import 'reflect-metadata';

import express from 'express';

import { createConnection } from 'typeorm';

import userRoutes from '../routes/userRoutes';

const app = express();

const PORT = process.env.PORT || 3000;

app.use(express.json());

app.use('/api', userRoutes);

createConnection().then(() => {

  app.listen(PORT, () => {
```

```
console.log(`Server is running on port ${PORT}`);

});

}).catch(error => console.log(error));
```

Этот файл создает соединение с базой данных, запускает сервер и подключает маршруты.

## 10. Запуск проекта

Добавляем скрипт запуска в `package.json`:

```
"scripts": {

  "start": "ts-node-dev src/index.ts"

},
```

Теперь, чтобы запустить проект, используем команду: *npm start*

### Пример выполнения запросов:

Получение всех пользователей: GET `/api/users`

Создание нового пользователя: POST `/api/users` с JSON-данными в теле запроса:

- { "name": "John Doe", "email": "john.doe@example.com" }

### Вывод

Таким образом, мы создали базовую структуру для приложения на Express, TypeORM и TypeScript, которая легко расширяется и модифицируется под нужды любого проекта.