САНКТ-ПЕТЕРБУРГСКИЙ НАЦИОНАЛЬНЫЙ ИССЛЕДОВАТЕЛЬСКИЙ УНИВЕРСИТЕТ ИТМО

Дисциплина: Бэк-энд разработка

Отчет

Лабораторная работа № 1
“Typescript: основы языка”

Выполнил:

Коротин А.М.

K33392

Проверил:
Добряков Д. И.

Санкт-Петербург

2024 г.

Задача

Реализовать boilerplate на express + sequelize + typescript. Должно быть явное разделение на:

— модели,

— контроллеры

— роуты,

— сервисы для работы с моделями (реализуем паттерн “репозиторий”).

Ход работы

Boilerplate на express представляет собой базовые настройки проекта и файловую структуру. Данная часть сходна для всех проектов на заданном стеке технологий и может быть легко переиспользована или дополнена.

Для начала опишем структуру boilerplate проекта. Он должен содержать в себе:

1) настройки npm-пакета,
2) настройки typescript,
3) настройки окружения (файл .env для хранения конфигурационных параметров),
4) базовую настройку express и sequelize, включая модели, контроллеры, роуты и сервисы.

После тривиальной инициализации npm-пакета и установки всех необходимых зависимостей, реализуем npm-команду для сборки, проверки и запуска проекта (команды build, lint и start, соответственно). Реализованные команды приведены на рисунке 1.

```
"scripts": {
  "start": "node --env-file=./.env build/index.js",
  "build": "npx tsc",
  "lint": "npx eslint . --ext .ts",
```

Рисунок 1 — Реализованные npm-команды

Далее, создадим файловую структуру, отвечающую требованиям из пункта 4 (рисунок 2).

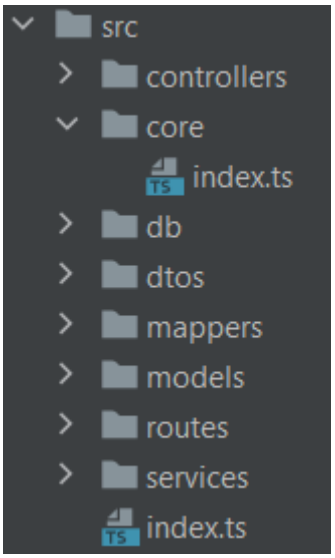

Рисунок 2 — Файловая структура проекта

Далее произведем настройку ORM Sequelize. Параметры конфигурации будем хранить в файле .env. На рисунке 3 приведена выдержка из данной конфигурации.

```
const sequelize :Sequelize  = new Sequelize( options: {
    database: process.env.database,
    dialect: "sqlite",
    host: process.env.host,
    username: process.env.username,
    password: process.env.password,
    storage: process.env.storage, // if embedded
    logging: console.log,
    repositoryMode: true
});
```

Рисунок 3 — Конфигурация ORM Sequelize

В качестве примера в boilerplate проекте была реализована модель User и ее атрибуты (рисунки 4 и 5).

```ts
export type UserAttributes = {
    id: number,
    name: string,
    email: string,
    password: string
}

export type UserCreationAttributes = Optional<UserAttributes, 'id'>;
```

Рисунок 4 — Атрибуты модели User

```ts
@Table
export class User extends Model<UserAttributes, UserCreationAttributes> {
    @Column
    name: string;

    @Unique
    @AllowNull( allowNull: false)
    @Column
    email: string;

    @AllowNull( allowNull: false)
    @Column
    password: string;

    1+ usages  Alexey Korotin
    @BeforeUpdate
    @BeforeCreate
    static hashPassword(instance: User): void {
        const {password : string } = instance;
        if (instance.changed( key: 'password')) {
            instance.password = bcrypt.hashSync(password, bcrypt.genSaltSync( rounds: 8));
        }
    }
}
```

Рисунок 5 — Модель User

Стоит отметить, что была реализована функция хеширования пароля при его изменении.

Далее, перейдем к сервисному слою. Был выделен параметризованный CRUD-интерфейс для всех сервисов, предлагающих данные операции (рисунок 6).

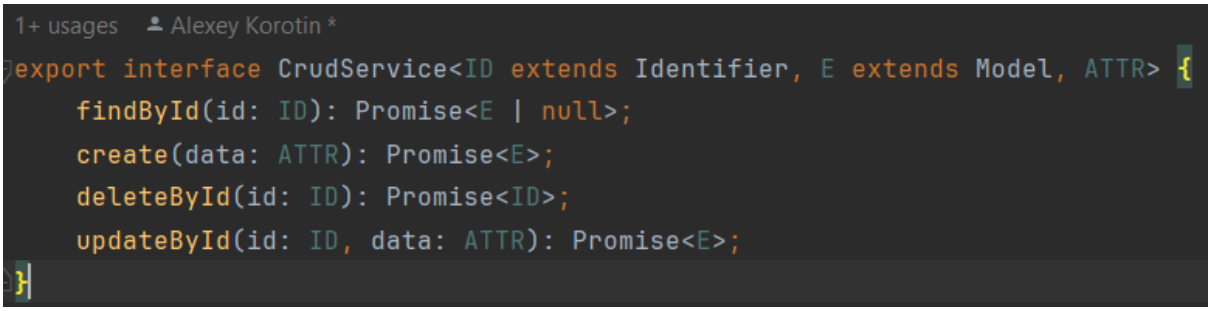

Рисунок 6 — Общий интерфейс для сервисов с CRUD-операциями

Также была предоставлена конкретная реализация данного интерфейса для работы с сущностью User (рисунок 7).

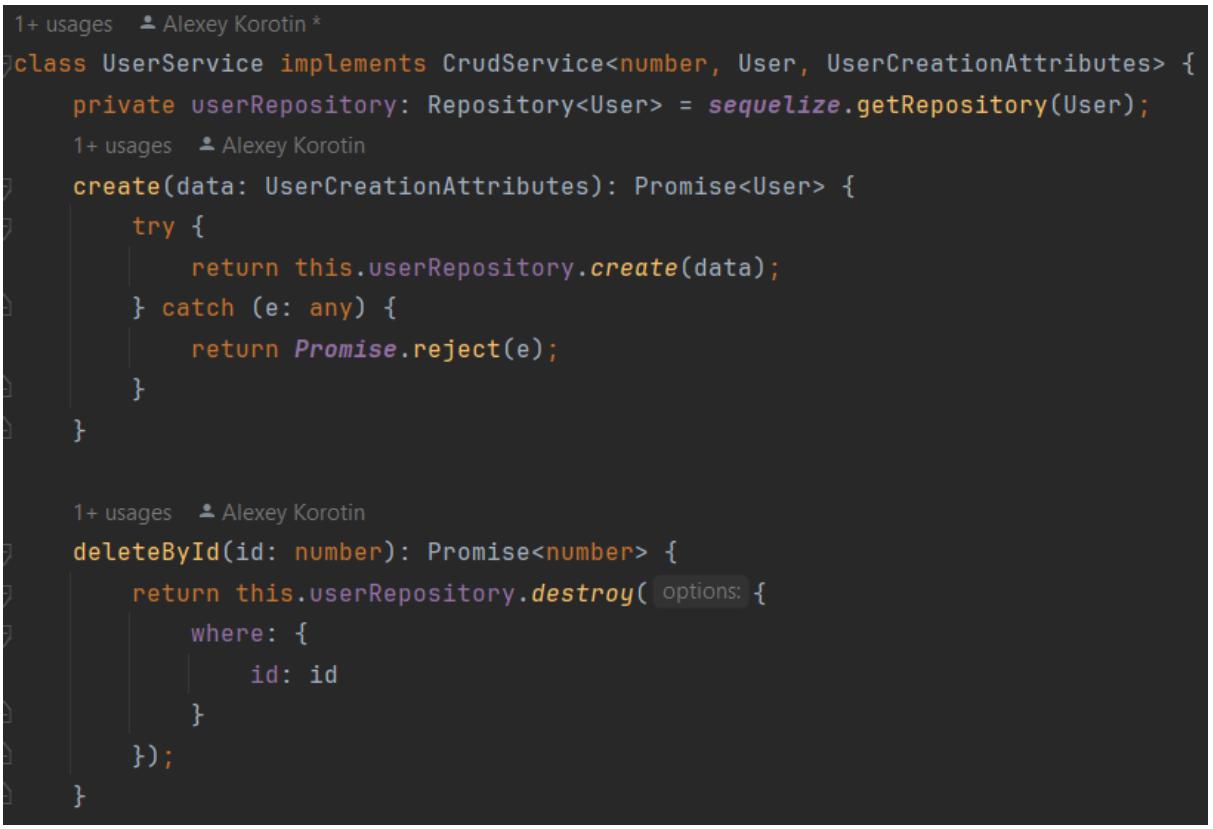

Рисунок 7 — Сервис для работы с сущностью User

Для корректной работы API также были предусмотрены DTO для моделей. Был реализован маркерный интерфейс Dto и интерфейс Mapper для преобразования модели в DTO (рисунок 8 и 9).

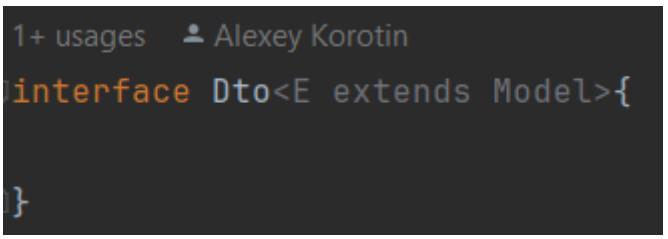

Рисунок 8 — Маркерный интерфейс Dto

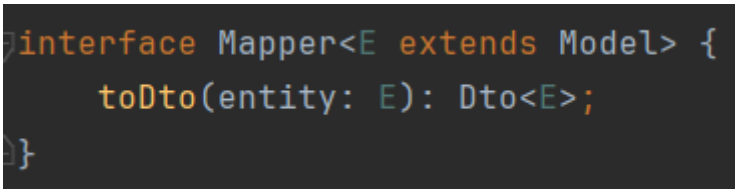

Рисунок 9 — Интерфейс Mapper

Для модели User были реализованы Dto и Mapper (рисунки 10 и 11).

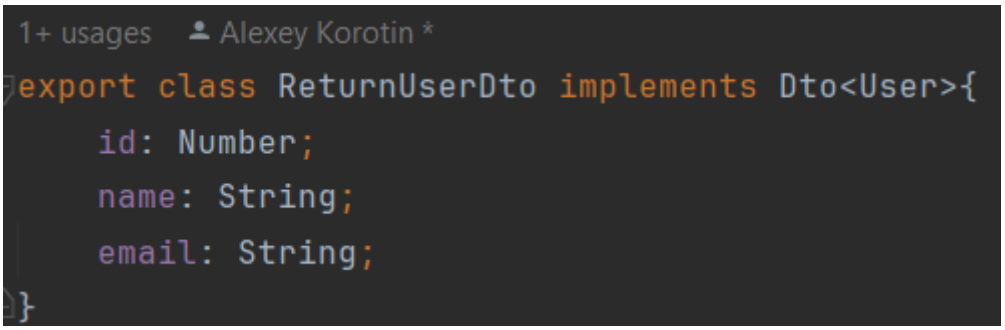

Рисунок 10 — UserDto

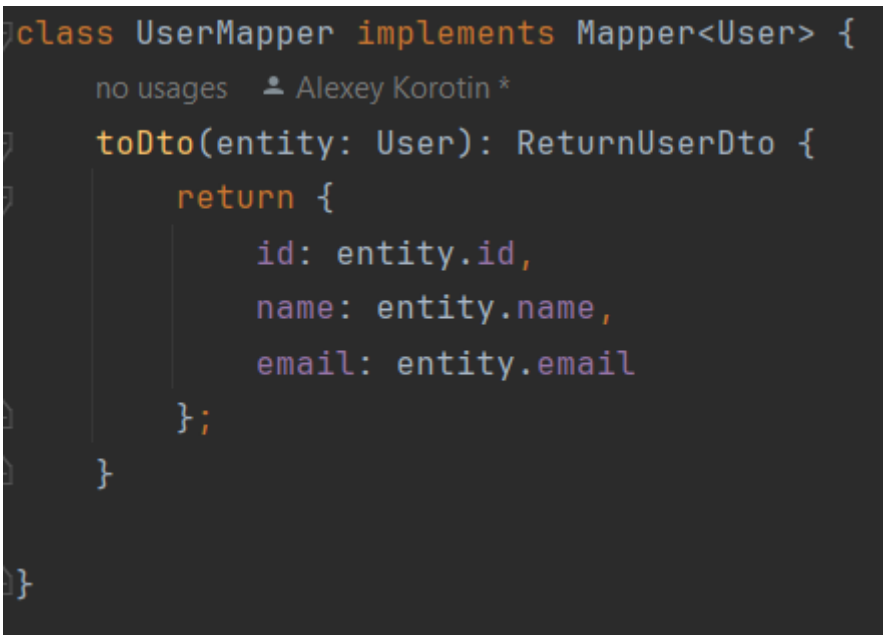

Рисунок 11 — UserMapper

Далее был реализован контроллер, представляющий CRUD-операции для модели User (фрагмент приведен на рисунке 12).

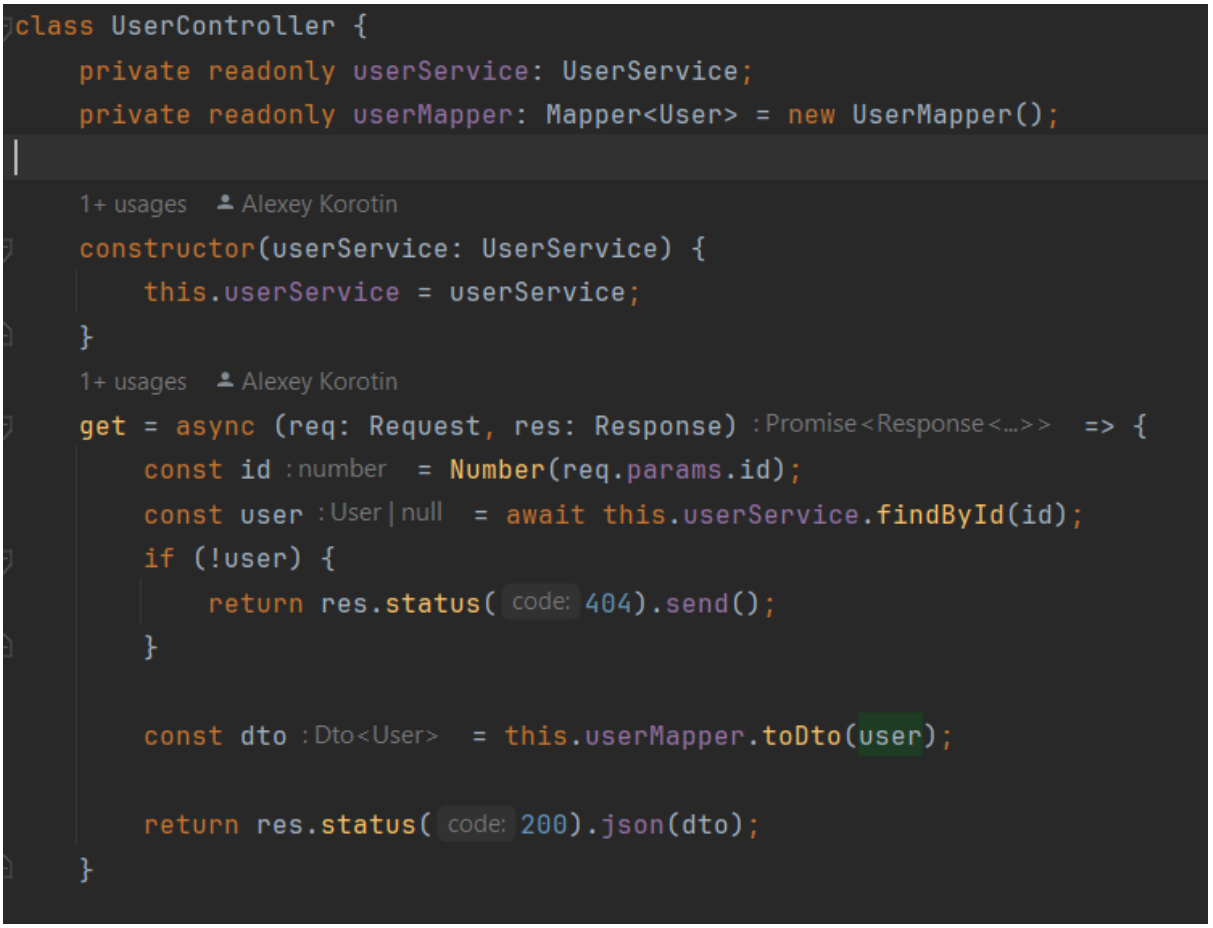

Рисунок 12 — Фрагмент UserController

Для маппинга входящих HTTP-запросов к соответствующим методам контроллера был создан роутер (рисунок 13).

```
const router :Router  = express.Router();
const controller :UserController  = new UserController(new UserService());

router.route( prefix: "/").post(controller.post);
router.route( prefix: "/:id").get(controller.get);
router.route( prefix: "/:id").delete(controller.delete);
```

Рисунок 13 — Контроллер для модели User

После был создан общий роутер, агрегирующий маршруты от всех роутеров низшего порядка. Роутер для модели User использовался с префиксом /users (рисунок 14).

```
const router :Router  = express.Router();

router.use("/users", userRouter);
```

Рисунок 14 — Агрегирующий роутер

Далее, был создан класс, ответственный за запуск express приложения. Фрагмент приведен на рисунке 15.

```
1+ usages  Alexey Korotin *
class App {
    public port: number;
    public host: string;

    private readonly app: express.Application;
    private server: Server;

    1+ usages  Alexey Korotin
    constructor(port : number  = 8000, host : string  = "localhost") {
        this.port = Number(process.env.port) || port;
        this.host = process.env.host || host;

        this.app = this.createApp();
        this.server = createServer(this.app);
    }

    1+ usages  Alexey Korotin
    private createApp(): express.Application {
        const app : Express  = express();
        app.use(cors());
        app.use(bodyParser.json());
        app.use('/v1', routes);

        return app;
    }
```

Рисунок 15 — Класс App

И, наконец, в файле index.ts создается и запускается express-приложение (рисунок 16)

```ts
import App from "./core";

const app :App  = new App();

app.start();
```

Рисунок 16 — Создание и запуск приложения

Проверим работоспособность приложения при помощи среды Postman. Попробуем создать, получить и удалить модель User.

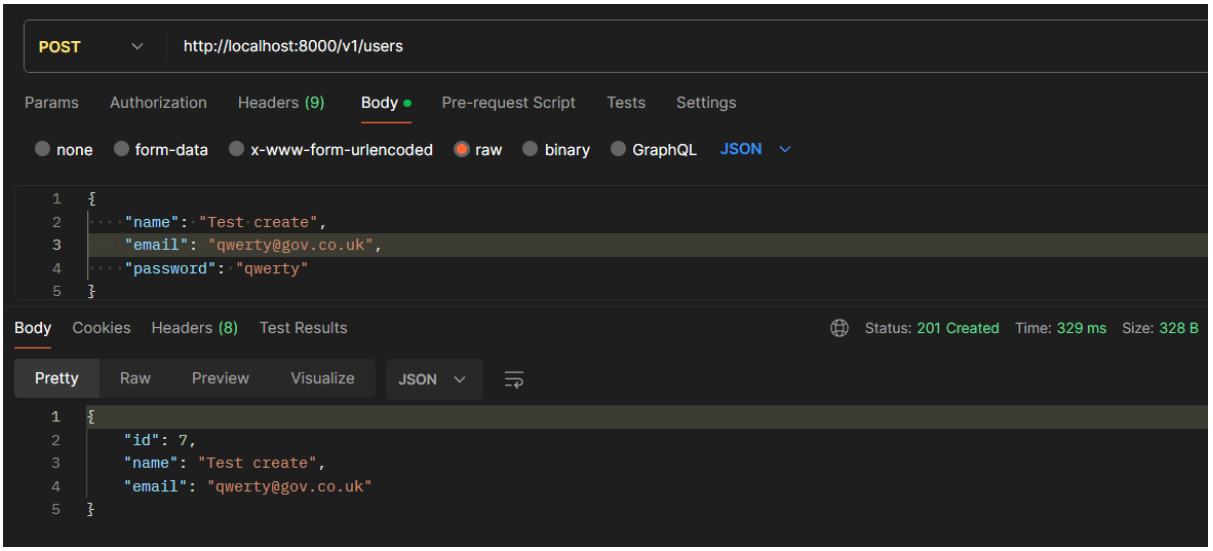

Рисунок 17 — Создание модели User

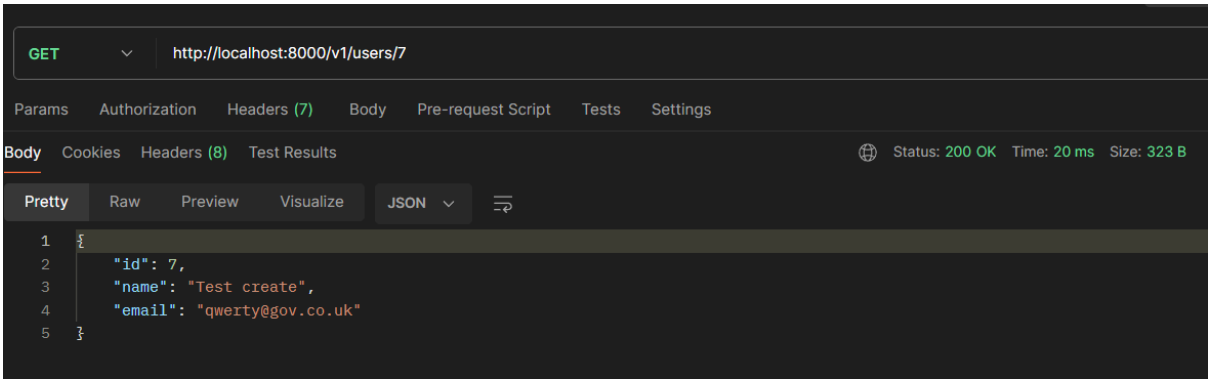

Рисунок 18 — Получение модели User

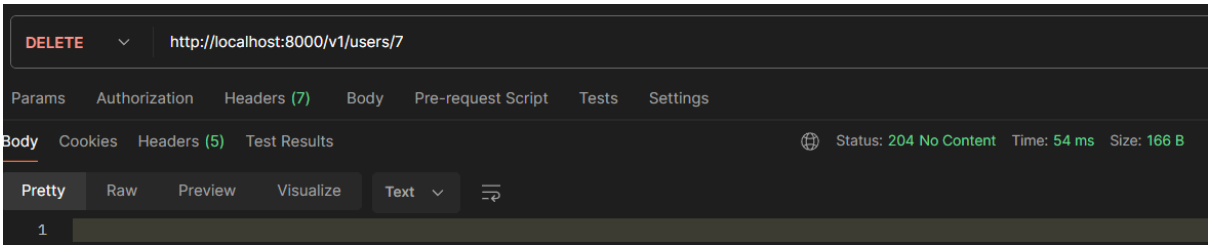

Рисунок 19 — Удаление модели User

Можем видеть, что операции были выполнены успешно, поэтому реализованный boilerplate можно считать рабочим.

Вывод

В ходе выполнения лабораторной работы были изучены основы языка TypeScript, работа с Sequelize-TypeScript и express. В результате был разработан рабочий шаблон приложения, который может быть использован для последующих лабораторных работ и проектов на данном стеке технологий.