САНКТ-ПЕТЕРБУРГСКИЙ НАЦИОНАЛЬНЫЙ ИССЛЕДОВАТЕЛЬСКИЙ УНИВЕРСИТЕТ ИТМО

Дисциплина: Бэк-энд разработка

Отчет

Домашняя работа 2

Выполнил:

Котовщиков Андрей

K3339

Проверил:
Добряков Д. И.

Санкт-Петербург

2022 г.

Задача

1. Реализовать все модели данных, спроектированные в рамках ДЗ1
2. Реализовать набор из CRUD-методов для работы с моделями данных средствами Express + TypeScript
3. Реализовать API-эндпоинт для получения пользователя по id/email

Ход работы

1. Реализация моделей данных

Первым делом все сущности (таблицы), спроектированные в рамках ДЗ1, были реализованы в виде классов с помощью ORM “TypeORM”. На рисунке 1 представлена модель резюме.

```
@Entity("resumes")
export class Resume {
    @PrimaryGeneratedColumn()
    id: string

    @Column({ nullable: false })
    title: string

    @Column({ name: "about_me", nullable: true })
    aboutMe: string

    @Column({ name: "photo_url", nullable: true })
    photoUrl: string

    @Column({ name: "is_hidden", nullable: false, default: false })
    isHidden: boolean

    @CreateDateColumn({ name: "created_at", nullable: false })
    createdAt: Date

    @UpdateDateColumn({ name: "updated_at", nullable: false })
    updatedAt: Date

    @ManyToOne(() => User, { nullable: false })
    @JoinColumn({ name: "user_id" })
    user: User

    @OneToOne(() => Contacts)
    contacts: Contacts

    @OneToMany(() => WorkPlace, (workPalce) => workPalce.resume)
```

Рисунок 1 – Модель “Резюме”

При помощи декоратора класса “Entity” мы указываем, что объявленный класс является ORM моделью, и в качестве аргумента передаем имя таблицы в БД. Декоратор “PrimaryGeneratedColumn” позволяет указать автоинкрементирующийся первичный ключ “id”. Декоратор “Column” нужен для маппинга неключевых атрибутов таблицы с полями класса. В качестве аргумента указывается имя атрибута в таблице, его тип непосредственно в СУБД (опционально), является ли атрибут обязательным (nullable), и значение по умолчанию. Для связей моделей друг с другом используются декораторы “ManyToOne”, “OneToMany” и “ManyToMany”.

2. Реализация CRUD-методов

Для созданных моделей были написаны базовые CRUD методы с помощью фреймворка express. На рисунке 2 можно видеть созданные для модели “Resume” CRUD контроллеры.

```
export const getAllResumes = async (req: Request, res: Response) => {
    const resumes = await dataSource.getRepository(Resume).find()
    res.status(200).json({ resumes })
}

export const getResumeById = async (req: Request, res: Response) => {
    const resume = await dataSource.getRepository(Resume).findOneBy({ id: req.params.id })
    if (!resume) {
        res.status(404).json({ message: "Resume not found" })
        return
    }

    res.status(200).json(resume)
}

export const createResume = async (req: Request, res: Response) => {
    const resume = dataSource.getRepository(Resume).create(req.body)
    const results = await dataSource.getRepository(Resume).save(resume)
    res.status(201).json(results)
}

export const updateResume = async (req: Request, res: Response) => {
    const resume = await dataSource.getRepository(Resume).findOneBy({ id: req.params.id })
    if (!resume) {
        res.status(404).json({ message: "Resume not found" })
        return
    }

    dataSource.getRepository(Resume).merge(resume, req.body)
    const results = await dataSource.getRepository(Resume).save(resume)
    res.status(200).json(results)
}

export const deleteResumeById = async (req: Request, res: Response) => {
    const results = await dataSource.getRepository(Resume).delete({ id: req.params.id })
    res.status(200).json(results)
}
```

Рисунок 2 – CRUD методы

Всего для каждой модели было написано по 5 контроллеров: получить список моделей, получить конкретную модель по id, создать модель, обновить существующую модель, удалить модель.

3. API-эндпоинт для получения пользователя по id/email

В качестве заключительного задания были также реализованы два API-эндпоинта для модели пользователя: получение пользователя по id, получение пользователя по email (рисунок 3).

```
export const getUserById = async (req: Request, res: Response) => {
    const user = await dataSource.getRepository(User).findOneBy({ id: req.params.id })
    if (!user) {
        res.status(404).json({ message: "User not found" })
        return
    }

    res.status(200).json(user)
}

export const getUserByEmail = async (req: Request, res: Response) => {
    const { email } = req.query
    if (!email) {
        res.status(400).json({ message: "Missing email query param" })
        return
    }

    const user = await dataSource.getRepository(User).findOneBy({ email: email as string })
    if (!user) {
        res.status(400).json({ message: "User not found" })
        return
    }

    res.status(200).json(user)
}
```

Рисунок 3 – API-эндпоинты для пользователя

При поиске по id пользователя мы берем его из path параметров, в то время как email необходимо получить из query параметров.

Вывод

В ходе выполнения работы все таблицы, описанные в предыдущем задании, были интегрированы в приложение с использованием TypeORM. Для каждой модели были реализованы стандартные CRUD-операции и API-эндпоинты с применением фреймворка express.