

**САНКТ-ПЕТЕРБУРГСКИЙ НАЦИОНАЛЬНЫЙ
ИССЛЕДОВАТЕЛЬСКИЙ УНИВЕРСИТЕТ ИТМО**

Дисциплина: Бэк-энд разработка

Отчет

Практическая работа 2

Выполнил:

Котовщиков Андрей

К3339

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.

```
61 @Entity("resumes")
62 export class Resume {
63     @PrimaryGeneratedColumn()
64     id: string
65
66     @Column({ nullable: false })
67     title: string
68
69     @Column({ name: "about_me", nullable: true })
70     aboutMe: string
71
72     @Column({ name: "photo_url", nullable: true })
73     photoUrl: string
74
75     @Column({ name: "is_hidden", nullable: false, default: false })
76     isHidden: boolean
77
78     @CreateDateColumn({ name: "created_at", nullable: false })
79     createdAt: Date
80
81     @UpdateDateColumn({ name: "updated_at", nullable: false })
82     updatedAt: Date
83
84     @ManyToOne(() => User, { nullable: false })
85     @JoinColumn({ name: "user_id" })
86     user: User
87
88     @OneToOne(() => Contacts)
89     contacts: Contacts
90
91     @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.

```
5   export const getAllResumes = async (req: Request, res: Response) => {
6       const resumes = await dataSource.getRepository(Resume).find()
7       res.status(200).json({ resumes })
8   }
9
10  export const getResumeById = async (req: Request, res: Response) => {
11      const resume = await dataSource.getRepository(Resume).findOneBy({ id: req.params.id })
12      if (!resume) {
13          res.status(404).json({ message: "Resume not found" })
14          return
15      }
16
17      res.status(200).json(resume)
18  }
19
20  export const createResume = async (req: Request, res: Response) => {
21      const resume = dataSource.getRepository(Resume).create(req.body)
22      const results = await dataSource.getRepository(Resume).save(resume)
23      res.status(201).json(results)
24  }
25
26  export const updateResume = async (req: Request, res: Response) => {
27      const resume = await dataSource.getRepository(Resume).findOneBy({ id: req.params.id })
28      if (!resume) {
29          res.status(404).json({ message: "Resume not found" })
30          return
31      }
32
33      dataSource.getRepository(Resume).merge(resume, req.body)
34      const results = await dataSource.getRepository(Resume).save(resume)
35      res.status(200).json(results)
36  }
37
38  export const deleteResumeById = async (req: Request, res: Response) => {
39      const results = await dataSource.getRepository(Resume).delete({ id: req.params.id })
40      res.status(200).json(results)
41  }
```

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).

```
5 export const getUserById = async (req: Request, res: Response) => {
6   const user = await dataSource.getRepository(User).findOneBy({ id: req.params.id })
7   if (!user) {
8     res.status(404).json({ message: "User not found" })
9     return
10  }
11
12  res.status(200).json(user)
13 }
14
15 export const getUserByEmail = async (req: Request, res: Response) => {
16   const { email } = req.query
17   if (!email) {
18     res.status(400).json({ message: "Missing email query param" })
19     return
20   }
21
22   const user = await dataSource.getRepository(User).findOneBy({ email: email as string })
23   if (!user) {
24     res.status(400).json({ message: "User not found" })
25     return
26   }
27
28   res.status(200).json(user)
29 }
```

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.