САНКТ-ПЕТЕРБУРГСКИЙ НАЦИОНАЛЬНЫЙ ИССЛЕДОВАТЕЛЬСКИЙ УНИВЕРСИТЕТ ИТМО

Дисциплина: Бек-энд разработка

Отчет

Лабораторная работа 1: Typescript: основы языка

Выполнил:

Скороходова Елена

Группа
К33392

Проверил:
Добряков Д. И.

Санкт-Петербург

2024 г.

Задачи
Нужно написать свой boilerplate на express + sequelize / TypeORM + typescript.

Должно быть явное разделение на:

- модели
- контроллеры
- роуты
- сервисы для работы с моделями (реализуем паттерн “репозиторий”)

Ход работы

Для начала создадим модель user с полями айди, имени и почты с использованием экземпляра sequelize для взаимодействия с базой данных:

```
import { DataTypes, Model } from 'sequelize';
import { sequelize } from '../config/database';

class User extends Model {
  public id!: number;
  public name!: string;
  public email!: string;
}

User.init(
  {
    id: {
      type: DataTypes.INTEGER,
      autoIncrement: true,
      primaryKey: true,
    },
    name: {
      type: DataTypes.STRING,
      allowNull: false,
    },
    email: {
      type: DataTypes.STRING,
      allowNull: false,
      unique: true,
    },
  },
  {
    sequelize,
    modelName: 'User',
  }
);

export { User };
```

Далее реализуем UserController, который представляет собой контроллер, который обрабатывает HTTP-запросы для взаимодействия с пользователями в приложении. Здесь реализованы методы по обработке запросов на создание, получение и удаление пользователя.

```
import { Request, Response } from 'express';
import { UserService } from '../services/UserService';

class UserController {
  public static async createUser(req: Request, res: Response): Promise<void> {
    const { name, email } = req.body;
    try {
      const user = await UserService.createUser(name, email);
      res.status(201).json(user);
    } catch (error) {
      console.error('Error creating user:', error);
      res.status(500).send('Error creating user');
    }
  }

  public static async getUserById(req: Request, res: Response): Promise<void> {
    const userId = parseInt(req.params.id);
    try {
      const user = await UserService.getUserById(userId);
      if (!user) {
        res.status(404).send('User not found');
        return;
      }
      res.json(user);
    } catch (error) {
      console.error('Error getting user by id:', error);
      res.status(500).send('Error getting user');
    }
  }

  public static async deleteUser(req: Request, res: Response): Promise<void> {
    const userId = parseInt(req.params.id);
    try {
      const success = await UserService.deleteUser(userId);
      if (!success) {
        res.status(404).send('User not found');
        return;
```

Далее пишем репозиторий для работы с моделью User. Репозиторий предоставляет методы для выполнения операций CRUD (создание, чтение, обновление, удаление) над данными пользователей в базе данных.

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

class UserRepository {
  public static async createUser(name: string, email: string): Promise<User> {
    try {
      const user = await User.create({ name, email });
      return user;
    } catch (error: any) {
      throw new Error('Error creating user: ' + error.message);
    }
  }

  public static async getUserById(id: number): Promise<User | null> {
    try {
      const user = await User.findByPk(id);
      return user;
    } catch (error: any) {
      throw new Error('Error getting user by id: ' + error.message);
    }
  }

  public static async deleteUser(id: number): Promise<boolean> {
    try {
      const deletedCount = await User.destroy({ where: { id } });
      return deletedCount > 0;
    } catch (error: any) {
      throw new Error('Error deleting user: ' + error.message);
    }
  }
}

export { UserRepository };
```

После идет сервисный слой, который предоставляет интерфейс для выполнения операций с пользователями, используя функции, предоставленные в UserRepository

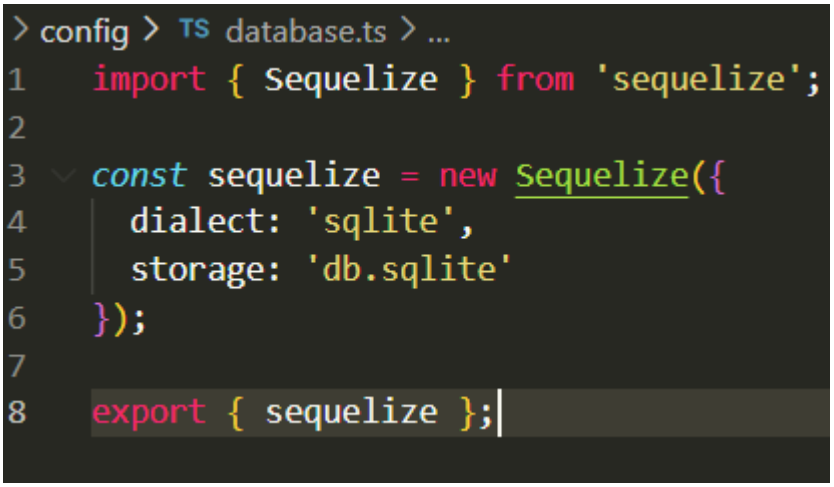
```ts
import { UserRepository } from '../repositories/UserRepository';
import { User } from '../models/User';

class UserService {
  public static async createUser(name: string, email: string): Promise<User> {
    return UserRepository.createUser(name, email);
  }

  public static async getUserById(id: number): Promise<User | null> {
    return UserRepository.getUserById(id);
  }

  public static async deleteUser(id: number): Promise<boolean> {
    return UserRepository.deleteUser(id);
  }
}

export { UserService };
```

И, наконец, пути для CRUD операций с пользователями. Маршруты предоставляют API для управления пользователями в приложении, обеспечивая возможность создания, чтения и удаления пользовательских данных.

```ts
import express from "express";
import { UserController } from "../controllers/UserController";

const router = express.Router();

router.post('/users', UserController.createUser);
router.get('/users/:id', UserController.getUserById);
router.delete('/users/:id', UserController.deleteUser);

export { router as userRoutes };
```

Для подключения бд и миграций создаем config

И файл подключения бд. Скрипт создает экземпляр Sequelize - объекта, который представляет собой основной интерфейс для работы с базой данных с помощью Sequelize ORM. В данной лр я использовала SQLite, предварительно создав файл самой бд.



А также создаем скрипт app.ts, который создает и настраивает сервер Express для обработки HTTP-запросов. Он использует библиотеку Express для управления маршрутами и обработчиками запросов, middleware body-parser для обработки данных запроса в формате JSON и маршруты, определенные в файле userRoutes, для обработки запросов, связанных с пользователями.

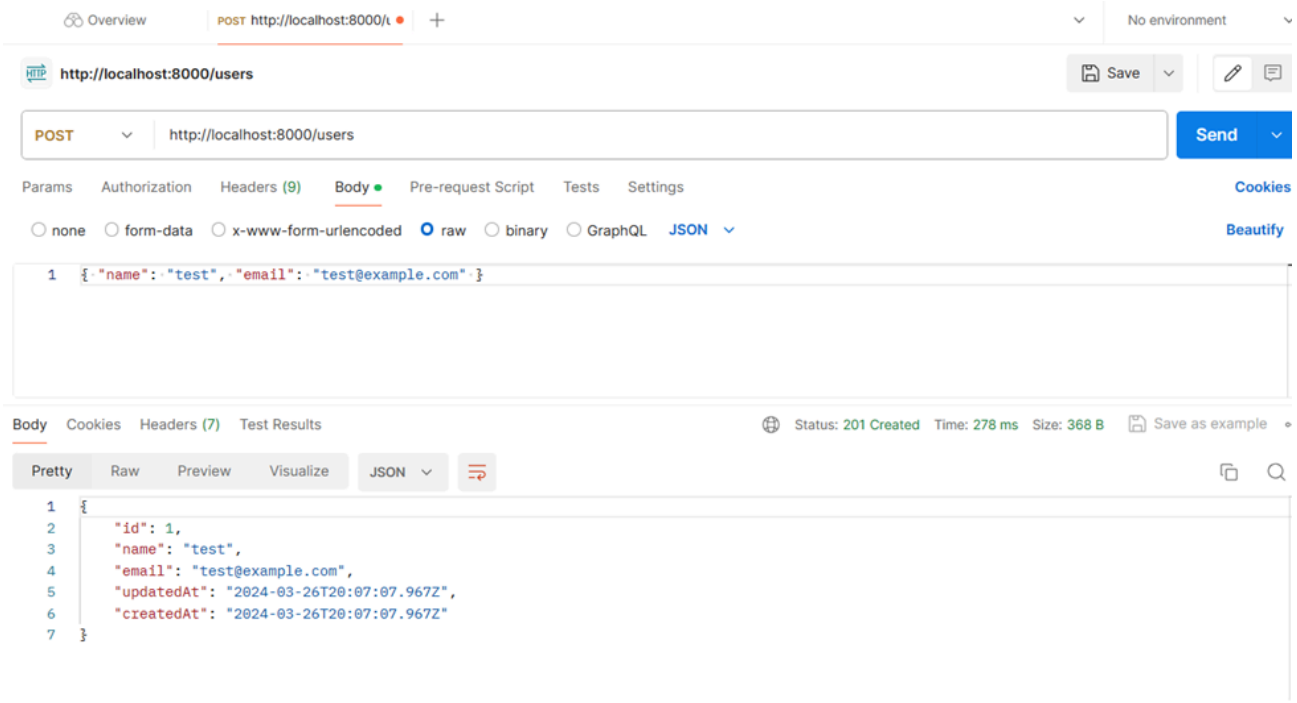
```ts
import express from 'express';
import bodyParser from 'body-parser';
import { userRoutes } from './src/routes/userRoutes';

const app = express();

app.use(bodyParser.json());
app.use(userRoutes);

const PORT = process.env.PORT || 8000;

app.listen(PORT, () => {
  console.log(`Server is running on port ${PORT}`);
});
```

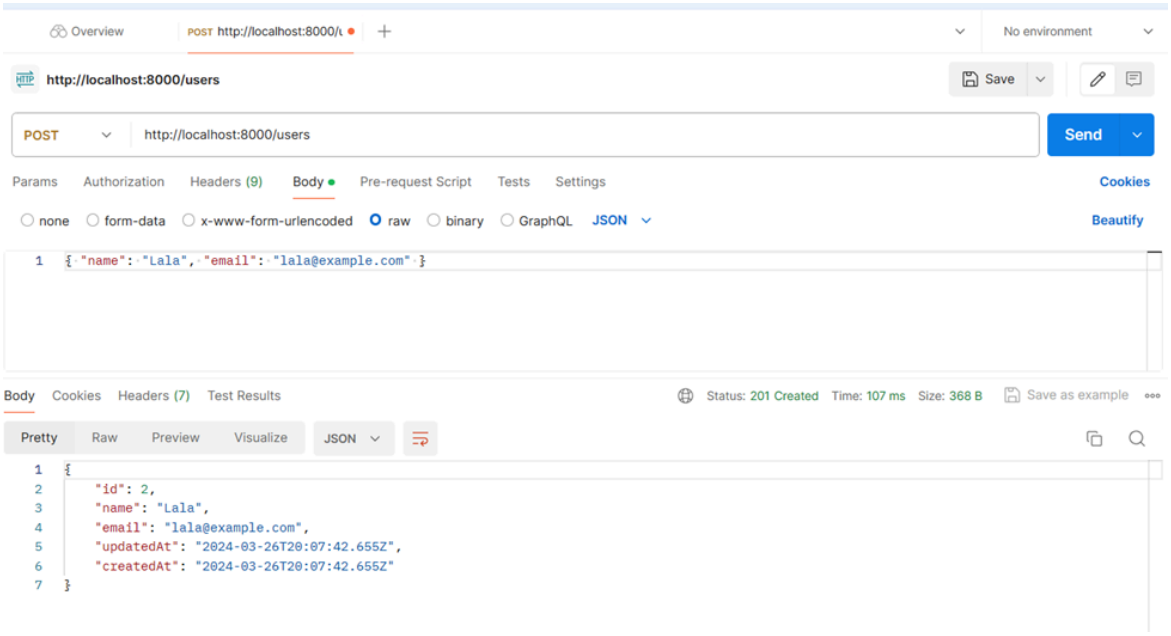
Создаем миграции, применяем и запускаем сервер при помощи команды npm start.

Теперь проверим работу в Postman:

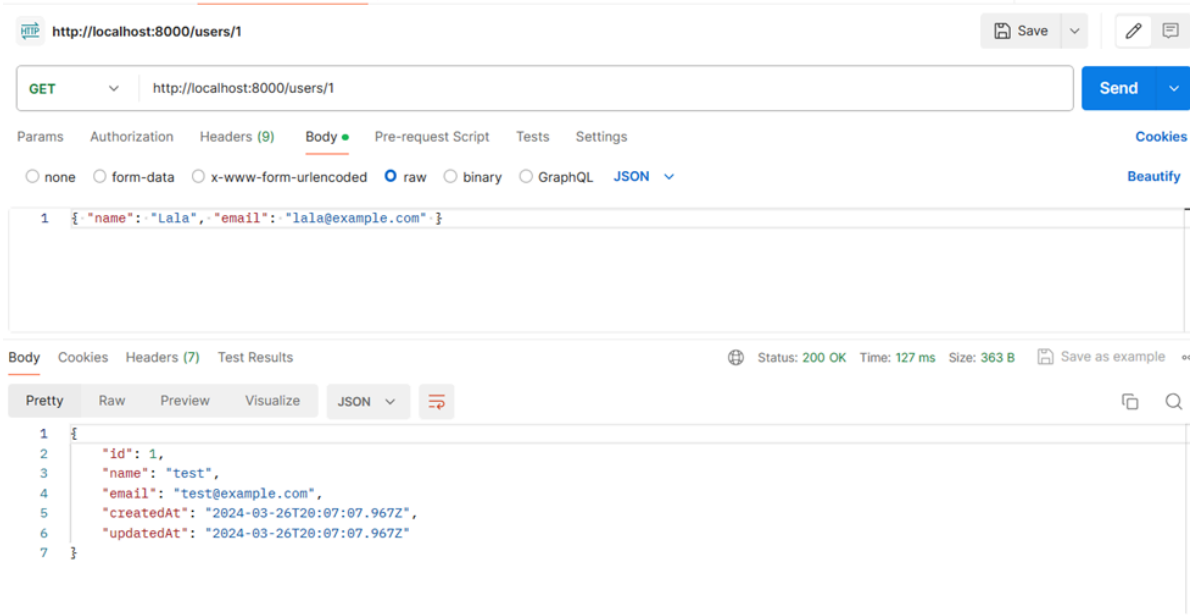
Выполним запросы на добавление юзера, получение и удаление

POST

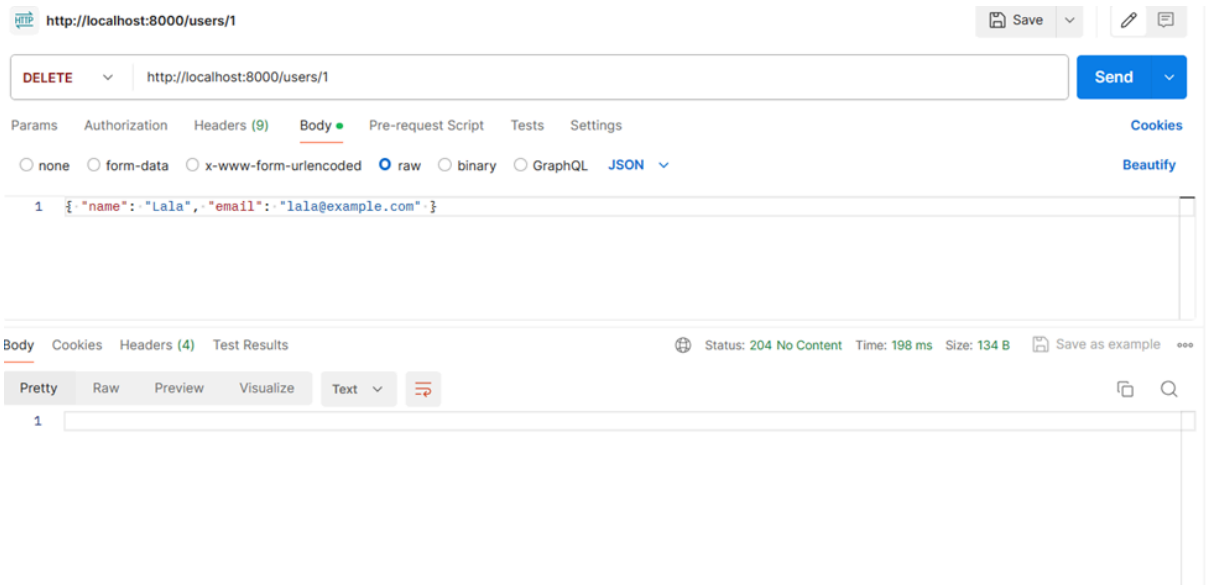



Overview
POST http://localhost:8000/
No environment
http://localhost:8000/users
Save
POST
http://localhost:8000/users
Send
Params
Authorization
Headers (9)
Body
Pre-request Script
Tests
Settings
Cookies
none
form-data
x-www-form-urlencoded
raw
binary
GraphQL
JSON
Beautify
{ "name": "Lala", "email": "lala@example.com" }
Body
Cookies
Headers (7)
Test Results
Status: 201 Created
Time: 107 ms
Size: 368 B
Save as example
Pretty
Raw
Preview
Visualize
JSON
{
"id": 2,
"name": "Lala",
"email": "lala@example.com",
"updatedAt": "2024-03-26T20:07:42.655Z",
"createdAt": "2024-03-26T20:07:42.655Z"
}

GET


http://localhost:8000/users/1
Save
GET
http://localhost:8000/users/1
Send
Params
Authorization
Headers (9)
Body
Pre-request Script
Tests
Settings
Cookies
none
form-data
x-www-form-urlencoded
raw
binary
GraphQL
JSON
Beautify
{ "name": "Lala", "email": "lala@example.com" }
Body
Cookies
Headers (7)
Test Results
Status: 200 OK
Time: 127 ms
Size: 363 B
Save as example
Pretty
Raw
Preview
Visualize
JSON
{
"id": 1,
"name": "test",
"email": "test@example.com",
"createdAt": "2024-03-26T20:07:07.967Z",
"updatedAt": "2024-03-26T20:07:07.967Z"
}

DELETE




```
D:\CharmTasks\Express_HW>npm start

> express_app@1.0.0 start
> tsc && node dist/app.js

Server is running on port 8000
Executing (default): INSERT INTO `Users` (`id`,`name`,`email`,`createdAt`,`updatedAt`) VALUES (NULL,$1,$2,$3,$4);
Executing (default): INSERT INTO `Users` (`id`,`name`,`email`,`createdAt`,`updatedAt`) VALUES (NULL,$1,$2,$3,$4);
Executing (default): SELECT `id`, `name`, `email`, `createdAt`, `updatedAt` FROM `Users` AS `User` WHERE `User`.`id`
;
Executing (default): DELETE FROM `Users` WHERE `id` = 1
```

Нас скринах видно, что запросы проходят - значит все работает.

Вывод

В первой лабораторной работе удалось познакомиться и поработать с TypeScript, а также изучить его основы и Sequelize и express. В результате был разработан рабочий boilerplate.