САНКТ-ПЕТЕРБУРГСКИЙ НАЦИОНАЛЬНЫЙ ИССЛЕДОВАТЕЛЬСКИЙ УНИВЕРСИТЕТ ИТМО

Дисциплина: Бек-энд разработка

Отчет
Лабораторная работа №2

Выполнила: Злотникова К.А.
Группа: K33392

Проверил:
Добряков Д. И.

Санкт-Петербург

2024 г.

Задача
Создать RESTful API для работы с магазином одежды, используя Express и TypeScript на основе ранее созданного boilerplate. Реализовать микросервисы для авторизации/регистрации и для управления товарами.

Ход работы

1. **Создание структуры проекта**

Создаем структуру проекта, чтобы было легко найти и организовать нужные файлы:

```
├── auth-service
│   ├── db
│   │   └── auth.sqlite
│   ├── node_modules
│   ├── src
│   │   ├── controllers
│   │   │   └── UserController.ts
│   │   ├── entities
│   │   │   └── User.ts
│   │   ├── routes
│   │   │   └── authRoutes.ts
│   │   └── index.ts
│   ├── ormconfig.json
│   ├── package.json
│   └── tsconfig.json
└── shop-service
    ├── db
    ├── node_modules
```

```
├── src
│   ├── controllers
│   │   ├── ItemController.ts
│   │   ├── OrderController.ts
│   │   └── RatingController.ts
│   ├── entities
│   │   ├── Item.ts
│   │   ├── Order.ts
│   │   └── Rating.ts
│   ├── middleware
│   │   └── authMiddleware.ts
│   ├── routes
│   │   ├── itemRoutes.ts
│   │   ├── orderRoutes.ts
│   │   └── ratingRoutes.ts
│   └── index.ts
├── ormconfig.json
├── package.json
└── tsconfig.json
```

2. Настройка микросервиса для авторизации и регистрации

2.1. Настройка TypeScript и TypeORM

Создаем файлы tsconfig.json и ormconfig.json для настройки TypeScript и TypeORM:

tsconfig.json:

```
{

 "compilerOptions": {
```

```
    "target": "ES6",
    "module": "commonjs",
    "strict": true,
    "esModuleInterop": true,
    "skipLibCheck": true,
    "outDir": "./dist",
    "experimentalDecorators": true,
    "emitDecoratorMetadata": true
  },
  "include": ["src/**/*"],
  "experimentalDecorators": true,
  "emitDecoratorMetadata": true
}
```

ormconfig.json:

```
{
  "type": "sqlite",
  "database": "db/main.sqlite",
  "synchronize": true,
  "logging": false,
  "entities": [
    "src/entities/**/*.ts"
  ]
}
```

2.2. Модель пользователя

Создаем модель пользователя в файле src/entities/User.ts:

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

@Entity()

export class User {

    @PrimaryGeneratedColumn()
```

```
    id!: number;

    @Column({ unique: true })

    username!: string;

    @Column()

    password!: string;

}
```

2.3. Контроллер пользователя

Создаем контроллер для регистрации и авторизации пользователей в файле src/controllers/UserController.ts:

```
import { Request, Response } from 'express';

import { getRepository } from 'typeorm';

import bcrypt from 'bcryptjs';

import jwt from 'jsonwebtoken';

import amqp from 'amqplib';

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

class UserController {

    static async signUp(req: Request, res: Response) {

        try {

            const userRepository = getRepository(User);

            const hashedPassword = await bcrypt.hash(req.body.password, 8);

                const newUser = userRepository.create({ username: req.body.username,
password: hashedPassword });

            await userRepository.save(newUser);

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

        } catch (error: any) {
```

```
            res.status(400).json({ message: error.message });
        }
    }
    static async signIn(req: Request, res: Response) {
        try {
            const userRepository = getRepository(User);
                  const user = await userRepository.findOne({ where: { username: req.body.username } });
            if (!user) {
                throw new Error('User not found');
            }
                   const isPasswordValid = await bcrypt.compare(req.body.password, user.password);
            if (!isPasswordValid) {
                throw new Error('Invalid password');
            }
                  const token = jwt.sign({ userId: user.id },'super_secret_key', { expiresIn: '1h' });
            res.status(200).json({ token });
        } catch (error: any) {
            res.status(400).json({ message: error.message });
        }
    }
    static verifyToken(token: string) {
        try {
            return jwt.verify(token,'super_secret_key');
        } catch (error) {
```

```
            throw new Error('Invalid token');

        }

    }

    static async sendUserData(userId: number) {

        const userRepository = getRepository(User);

        const user = await userRepository.findOne({ where: { id: userId } });

        if (!user) throw new Error('User not found');

        const connection = await amqp.connect('amqp://localhost');

        const channel = await connection.createChannel();

        const queue = 'auth_queue';

        await channel.assertQueue(queue, { durable: false });

        channel.sendToQueue(queue, Buffer.from(JSON.stringify(user)));

        setTimeout(() => {

            connection.close();

        }, 500);

    }

}

export default UserController;
```

2.4. Роуты для авторизации

Создаем файл src/routes/authRoutes.ts:

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

import UserController from '../controllers/UserController';

const router = Router();

router.post('/register', UserController.signUp);

router.post('/login', UserController.signIn);

export default router;
```

2.5. Настройка сервера

Создаем главный файл для запуска сервера src/index.ts:

```
import 'reflect-metadata';

import express from 'express';

import bodyParser from 'body-parser';

import { createConnection } from 'typeorm';

import authRoutes from './routes/authRoutes';

const app = express();

app.use(bodyParser.json());

app.use('/auth', authRoutes);

createConnection().then(() => {

   app.listen(4000, () => {

       console.log('Auth service is listening on port 4000');

   });

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

3. Настройка микросервиса для управления товарами

3.1. Настройка TypeScript и TypeORM

Создаем файлы tsconfig.json и ormconfig.json для настройки TypeScript и TypeORM:

tsconfig.json:

```
{

 "compilerOptions": {

   "target": "ES6",

   "module": "commonjs",

   "strict": true,

   "esModuleInterop": true,
```

```
    "skipLibCheck": true,
    "outDir": "./dist",
    "experimentalDecorators": true,
    "emitDecoratorMetadata": true
  },
  "include": ["src/**/*"],
  "experimentalDecorators": true,
  "emitDecoratorMetadata": true
}
```

ormconfig.json:

```
{
  "type": "sqlite",
  "database": "db/main.sqlite",
  "synchronize": true,
  "logging": false,
  "entities": [
    "src/entities/**/*.ts"
  ]
}
```

3.2. Модели

Создаем модели для товаров, заказов и рейтингов.

src/entities/Item.ts:

```
import { Entity, PrimaryGeneratedColumn, Column, OneToMany } from 'typeorm';
import { Rating } from './Rating';
@Entity()
```

```
export class Item {

   @PrimaryGeneratedColumn()

   id!: number;

   @Column()

   name!: string;

   @Column('decimal')

   price!: number;

   @Column('int')

   stock!: number;

   @Column('decimal', { default: 0 })

   discount!: number;

   @OneToMany(() => Rating, rating => rating.item)

   ratings!: Rating[];

}
```

src/entities/Order.ts:

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

import { Item } from './Item';

@Entity()

export class Order {

   @PrimaryGeneratedColumn()

   id!: number;

   @Column()

   userId!: number;

   @ManyToOne(() => Item)

   item!: Item;

   @Column('int')
```

```
    quantity!: number;

    @Column()

    date!: Date;

}
```

src/entities/Rating.ts:

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

import { Item } from './Item';

@Entity()

export class Rating {

    @PrimaryGeneratedColumn()

    id!: number;

    @Column()

    userId!: number;

    @ManyToOne(() => Item, item => item.ratings)

    item!: Item;

    @Column('int')

    rating!: number;

    @Column()

    comment!: string;

}
```

3.3. Контроллеры

Создаем контроллеры для управления товарами, заказами и рейтингами.

src/controllers/ItemController.ts:

```
import {Request, Response} from 'express';

import {getRepository} from 'typeorm';

import {Item} from '../entities/Item';
```

```
class ItemController {

   static async addItem(req: Request, res: Response) {

       try {

           const itemRepository = getRepository(Item);

           const newItem = itemRepository.create(req.body);

           await itemRepository.save(newItem);

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

       } catch (error: any) {

           res.status(400).json({message: error.message});

       }

   }

   static async modifyItem(req: Request, res: Response) {

       try {

           const itemRepository = getRepository(Item);

           await itemRepository.update(req.params.id, req.body);

                   const updatedItem = await itemRepository.findOne({where: {id:
Number(req.params.id)}});

           res.status(200).json(updatedItem);

       } catch (error: any) {

           res.status(400).json({message: error.message});

       }

   }

   static async removeItem(req: Request, res: Response) {

       try {

           const itemRepository = getRepository(Item);

           await itemRepository.delete(req.params.id);
```

```
            res.status(204).send();

        } catch (error: any) {

            res.status(400).json({message: error.message});

        }

    }

    static async getItemStock(req: Request, res: Response) {

        try {

            const itemRepository = getRepository(Item);

                        const item = await itemRepository.findOne({where: {id:
Number(req.params.id)}});

            res.status(200).json({stock: item?.stock});

        } catch (error: any) {

            res.status(400).json({message: error.message});

        }

    }

    static async applyOffer(req: Request, res: Response) {

        try {

            const itemRepository = getRepository(Item);

                            await itemRepository.update(req.params.id, {discount:
req.body.discount});

                    const updatedItem = await itemRepository.findOne({where: {id:
Number(req.params.id)}});

            res.status(200).json(updatedItem);

        } catch (error: any) {

            res.status(400).json({message: error.message});

        }

    }
```

```
}

export default ItemController;
```

src/controllers/OrderController.ts:

```
import { Request, Response } from 'express';

import { getRepository } from 'typeorm';

import { Item } from '../entities/Item';

import { Order } from '../entities/Order';

class OrderController {

   static async placeOrder(req: Request, res: Response) {

       try {

           const { userId, itemId, quantity } = req.body;

           const itemRepository = getRepository(Item);

           const orderRepository = getRepository(Order);

           const item = await itemRepository.findOne({ where: { id: itemId } });

           if (!item || item.stock < quantity) {

               throw new Error('Insufficient stock');

           }



           item.stock -= quantity;

           await itemRepository.save(item);

           const newOrder = orderRepository.create({

               userId,

               item,

               quantity,

               date: new Date()

           });
```

```
            await orderRepository.save(newOrder);

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

        } catch (error: any) {

            res.status(400).json({ message: error.message });

        }

    }

    static async getOrderHistory(req: Request, res: Response) {

        try {

            const userId = Number(req.params.userId);

            const orderRepository = getRepository(Order);

                  const orders = await orderRepository.find({ where: { userId },
relations: ['item'] });

            res.status(200).json(orders);

        } catch (error: any) {

            res.status(400).json({ message: error.message });

        }

    }

}

export default OrderController;
```

src/controllers/RatingController.ts:

```
import {Request, Response} from 'express';

import {getRepository} from 'typeorm';

import {Rating} from '../entities/Rating';

import {Item} from '../entities/Item';

class RatingController {

    static async addRating(req: Request, res: Response) {
```

```
        try {
            const ratingRepository = getRepository(Rating);
            const itemRepository = getRepository(Item);
            const {itemId, rating, comment} = req.body;
            const item = await itemRepository.findOne({where: {id: itemId}});
            if (!item) {
                throw new Error('Item not found');
            }
            const newRating = ratingRepository.create({
                userId: req.body.userId,
                item,
                rating,
                comment
            });
            await ratingRepository.save(newRating);
            res.status(201).json(newRating);
        } catch (error: any) {
            res.status(400).json({message: error.message});
        }
    }
    static async getRatings(req: Request, res: Response) {
        try {
            const ratingRepository = getRepository(Rating);
            const itemId = Number(req.params.itemId);
                const ratings = await ratingRepository.find({where: {item: {id:
itemId}}, relations: ['item']});
```

```
            res.status(200).json(ratings);

        } catch (error: any) {

            res.status(400).json({message: error.message});

        }

    }

    static async updateRating(req: Request, res: Response) {

        try {

            const ratingRepository = getRepository(Rating);

            const itemId = Number(req.params.itemId);

            const userId = req.body.userId;

            const {rating, comment} = req.body;

               const existingRating = await ratingRepository.findOne({where: {item:
{id: itemId}, userId}});

            if (!existingRating) {

                throw new Error('Rating not found');

            }

            existingRating.rating = rating;

            existingRating.comment = comment;

            await ratingRepository.save(existingRating);

            res.status(200).json(existingRating);

        } catch (error: any) {

            res.status(400).json({message: error.message});

        }

    }

    static async deleteRating(req: Request, res: Response) {

        try {
```

```
            const ratingRepository = getRepository(Rating);

            const { itemId } = req.params;

            const userId = req.body.user.userId;

               const rating = await ratingRepository.findOne({ where: { item: { id:
Number(itemId) }, userId } });

            if (!rating) {

                return res.status(404).json({ message: 'Rating not found' });

            }

            await ratingRepository.delete(rating.id);

            res.status(204).send();

        } catch (error: any) {

            res.status(400).json({ message: error.message });

        }

    }

}

export default RatingController;
```

3.4. Роуты

Создаем роуты для товаров, заказов и рейтингов.

src/routes/itemRoutes.ts:

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

import ItemController from '../controllers/ItemController';

import authenticate from '../middleware/authMiddleware';

const router = Router();

router.post('/items', authenticate, ItemController.addItem);

router.put('/items/:id', authenticate, ItemController.modifyItem);

router.delete('/items/:id', authenticate, ItemController.removeItem);
```

```
router.get('/items/:id/stock', authenticate, ItemController.getItemStock);

router.patch('/items/:id/offer', authenticate, ItemController.applyOffer);

export default router;
```

src/routes/orderRoutes.ts:

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

import OrderController from '../controllers/OrderController';

import authenticate from '../middleware/authMiddleware';

const router = Router();

router.post('/order', authenticate, OrderController.placeOrder);

router.get('/order/history/:userId',                                    authenticate,
OrderController.getOrderHistory);

export default router;
```

src/routes/ratingRoutes.ts:

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

import RatingController from '../controllers/RatingController';

import authenticate from '../middleware/authMiddleware';

const router = Router();

router.post('/ratings', authenticate, RatingController.addRating);

router.get('/ratings/:itemId', authenticate, RatingController.getRatings);

router.put('/ratings/:itemId', authenticate, RatingController.updateRating);

router.delete('/ratings/:itemId', authenticate, RatingController.deleteRating);

export default router;
```

3.5. Настройка сервера

Создаем главный файл для запуска сервера src/index.ts:

```
import 'reflect-metadata';

import express from 'express';
```

```
import bodyParser from 'body-parser';

import { createConnection } from 'typeorm';

import itemRoutes from './routes/itemRoutes';

import orderRoutes from './routes/orderRoutes';

import ratingRoutes from "./routes/ratingRoutes";

const app = express();

app.use(bodyParser.json());

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

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

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

createConnection().then(() => {

   app.listen(3000, () => {

       console.log('Main service is listening on port 3000');

   });

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

4. Настройка аутентификации

4.1. Middleware для проверки токена

Создаем файл src/middleware/authMiddleware.ts:

```
import { Request, Response, NextFunction } from 'express';

import amqp, { Channel, Connection, ConsumeMessage } from 'amqplib';

import jwt from 'jsonwebtoken';

interface User {

   id: number;

   username: string;

}

const jwtSecret = 'super_secret_key';
```

```
let amqpChannel: Channel;

const authenticate = async (req: Request, res: Response, next: NextFunction) => {

   try {

       const authHeader = req.headers.authorization;

       if (!authHeader) {

           return res.status(403).json({ message: 'Token not provided' });

       }

       const [bearer, token] = authHeader.split(' ');

       if (bearer !== 'Bearer' || !token) {

               return res.status(403).json({ message: 'Invalid token' });

       }

       const decoded: any = jwt.verify(token, jwtSecret);

       const userId = decoded.userId;

       const connection: Connection = await amqp.connect('amqp://localhost');

       amqpChannel = await connection.createChannel();

       const queue = 'auth_queue';

       await amqpChannel.assertQueue(queue, { durable: false });

       let user: User | null = null;

       await amqpChannel.consume(queue, (msg: ConsumeMessage | null) => {

           if (msg !== null) {

               user = JSON.parse(msg.content.toString());

               amqpChannel.ack(msg);

           }

       });

       await sendUserId(userId);

       await amqpChannel.close();
```

```
        await connection.close();

        setTimeout(() => {

            if (user) {

                req.body.user = user;

                next();

            } else {

                return res.status(403).json({ message: 'Invalid token' });

            }

        }, 500);

    } catch (error) {

        return res.status(403).json({ message: error });

    }

};

const sendUserId = async (userId: number) => {

    if (!amqpChannel) return;

    const queue = 'auth_queue';

    amqpChannel.sendToQueue(queue, Buffer.from(JSON.stringify({ userId })));

};

export default authenticate;
```

5. Запуск проекта

5.1. Скрипты запуска

Добавляем скрипты запуска в package.json для обоих микросервисов.

Для auth-service/package.json:

```
"scripts": {

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

 },
```

Для shop-service/package.json:

```
"scripts": {

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

},
```

Теперь, чтобы запустить проект, используем команду:

- npm start

Пример выполнения запросов

- Регистрация пользователя: POST /auth/register:
  {
  "username": "user1",
  "password": "password123"
  }
- Авторизация пользователя: POST /auth/login
  {
  "username": "user1",
  "password": "password123"
  }
- Добавление товара: POST /api/items
  {
  "name": "Shirt",
  "price": 19.99,
  "stock": 100
  }
- Получение всех товаров:
  GET /api/items

Вывод

Таким образом, мы создали микросервисную архитектуру для магазина одежды с использованием Express, TypeORM и TypeScript. Авторизация и регистрация вынесены в отдельный микросервис, а управление товарами, заказами и рейтингами находится в основном микросервисе.