САНКТ-ПЕТЕРБУРГСКИЙ НАЦИОНАЛЬНЫЙ ИССЛЕДОВАТЕЛЬСКИЙ УНИВЕРСИТЕТ ИТМО

Дисциплина: Бэк-энд разработка

Отчет

Практическая/Лабораторная работа

Выполнил:

Габов Михаил

К3340

Проверил:
Добряков Д. И.

Санкт-Петербург

2022 г.

Цель работы:

Реализовать автодокументирование средствами swagger, реализовать документацию API средствами Postman.

1. Реализация автодокументирования средствами Swagger

Для автоматического создания документации OpenAPI (Swagger) был использован фреймворк **tsoa**

1.1. Настройка проекта и зависимости

Первым шагом была установка необходимых npm-пакетов. В файле package.json можно видеть добавленные зависимости:

- tsoa: Основной инструмент для генерации маршрутов и спецификации OpenAPI.
- swagger-ui-express: Middleware для Express.js для отображения Swagger UI.
- concurrently: Используется для одновременного запуска нескольких процессов, в данном случае - сервера nodemon и генератора tsoa.

package.json (фрагмент)

```
"dependencies": {
   // ... другие зависимости
   "swagger-ui-express": "^5.0.0",
   "tsoa": "^6.2.0"
},
"devDependencies": {
   // ... другие зависимости
   "@types/swagger-ui-express": "^4.1.6",
   "concurrently": "^8.2.2"
}
```

1.2. Конфигурация tsoa

Для управления процессом генерации был создан конфигурационный файл

tsoa.json. В нем указаны ключевые параметры:

- entryFile: Точка входа в приложение (src/index.ts).
- controllerPathGlobs: Шаблон для поиска файлов с контроллерами API.
- spec и routes: Настройки для выходных файлов - спецификации swagger.json и сгенерированных маршрутов routes.ts.

tsoa.json

```
{
 "entryFile": "src/index.ts",
 "noImplicitAdditionalProperties": "throw-on-extras",
 "controllerPathGlobs": ["src/controllers/**/*.ts"],
 "spec": {
  "outputDirectory": "src/generated",
  "specVersion": 3
 },
 "routes": {
  "routesDir": "src/generated"
 }
}
```

1.3. Использование декораторов в контроллерах

tsoa использует декораторы в коде контроллеров для описания эндпоинтов, их параметров, ответов и т.д. Это позволяет держать документацию в актуальном состоянии прямо рядом с кодом.

Пример из src/controllers/user.controller.ts:
В этом примере декораторы @Route, @Tags, @Post, @SuccessResponse, @Body и @Security описывают эндпоинт для регистрации нового пользователя.

- @Route("users"): Задает базовый путь для всех методов контроллера.
- @Tags("User"): Группирует эндпоинты в Swagger UI под тегом "User".
- @Post("/register"): Определяет HTTP-метод и путь.
- @SuccessResponse("201", "Created"): Описывает успешный ответ (статус

201).

- @Body(): Указывает, что данные для userDto должны быть взяты из тела запроса.
- @Security("jwt"): Указывает, что для доступа к эндпоинту требуется аутентификация

src/controllers/user.controller.ts (фрагмент)

```
import { Body, Controller, Post, Route, SuccessResponse, Tags, Get, Path,
Security, Request } from 'tsoa';
import { CreateUserDto, UserDto } from '../dtos/user.dto';
import { AppDataSource } from '../data-source';
import { User } from '../models';
import * as bcrypt from 'bcryptjs';
import * as jwt from 'jsonwebtoken';

@Route("users")
@Tags("User")
export class UserController extends Controller {

  // ...

  @SuccessResponse("201", "Created")
  @Post("/register")
  public async register(@Body() userDto: CreateUserDto): Promise<void> {
    const userRepository = AppDataSource.getRepository(User);
    const hashedPassword = await bcrypt.hash(userDto.password, 10);
    const newUser = userRepository.create({
      ...userDto,
      password: hashedPassword,
    });
    await userRepository.save(newUser);
    this.setStatus(201);
    return;
```

```
  }

  // ...
}
```

1.4. Интеграция Swagger UI в Express

В главном файле приложения src/index.ts настроен middleware swagger-ui-express, который обслуживает сгенерированный swagger.json и предоставляет удобный UI для взаимодействия с API.

src/index.ts (фрагмент)

```
import express from 'express';
import "reflect-metadata";
import swaggerUi from 'swagger-ui-express';
import { RegisterRoutes } from './generated/routes';

// ...

const app = express();
app.use(express.json());

// ...

try {
    const swaggerDocument = require('./generated/swagger.json');
    app.use('/docs', swaggerUi.serve, swaggerUi.setup(swaggerDocument));
} catch (err) {
    console.error('Unable to load swagger.json', err);
}

RegisterRoutes(app);
```

// ...

1.5. Результат

После запуска приложения и перехода по адресу http://localhost:3000/docs открывается интерактивная документация Swagger UI.

В этом интерфейсе можно:

- Видеть все доступные эндпоинты, сгруппированные по тегам.
- Раскрывать каждый эндпоинт, чтобы увидеть детали: HTTP-метод, URL, параметры, ожидаемые тела запросов.
- Просматривать модели данных (схемы DTO).
- **Отправлять тестовые запросы** прямо из браузера и видеть реальные ответы от сервера.

2. Реализация документации API средствами Postman

2.1. Импорт спецификации OpenAPI

Процесс создания документации в Postman был значительно упрощен благодаря наличию файла swagger.json, сгенерированного tsoa.

Шаги для импорта:

1. Запустить локальный сервер, чтобы файл swagger.json был доступен по URL, например: http://localhost:3000/swagger.json.
2. В приложении Postman выбрать Import.
3. Переключиться на вкладку Link и вставить URL файла swagger.json.
4. Postman автоматически анализирует файл и предлагает создать новую коллекцию на его основе.

2.2. Структура коллекции

После импорта Postman создает коллекцию, которая полностью повторяет структуру API:

- Название коллекции соответствует названию API из swagger.json.
- Запросы сгруппированы в папки, соответствующие тегам (@Tags) из

контроллеров.

- Каждый запрос уже содержит:
  - Правильный HTTP-метод (GET, POST, etc.).
  - URL с плейсхолдерами для path-параметров (например, /:id).
  - Предустановленные заголовки (например, Content-Type: application/json).
  - Пример тела запроса (request body), сгенерированный на основе DTO.

Выводы

В ходе выполнения работы были применены два подхода к документированию API.