# САНКТ-ПЕТЕРБУРГСКИЙ НАЦИОНАЛЬНЫЙ ИССЛЕДОВАТЕЛЬСКИЙ УНИВЕРСИТЕТ ИТМО

Дисциплина: Бэк-энд разработка

Отчет

Домашняя работа №3

Выполнил:

Корчагин Вадим

Группа К3341

Проверил: Добряков Д. И.

Санкт-Петербург

2025 г.

## Тема

Платформа для фитнес-тренировок и здоровья.

## Задача

- реализовать автодокументирование средствами swagger;
- реализовать документацию API средствами Postman.

#### Технологии:

- Swagger (routing-controllers-openapi, swagger-ui-express)
- Postman (импорт через openapi2postmanv2)
- curl, jq автоматизация через Makefile
- Makefile сборка документации, выгрузка и синхронизация с Postman API

## Ход работы

### **Swagger**

В проекте используется генерация спецификации OpenAPI из метаданных routing-controllers и class-validator. Это обеспечивается следующим:

#### Модули:

- getMetadataArgsStorage сбор метаданных контроллеров
- validationMetadatasToSchemas генерация схем DTO
- routingControllersToSpec генерация спецификации OpenAPI 3.0
- swagger-ui-express отображение UI
- 1. Декораторы в контроллерах влияют на описание маршрутов
- @Get("/") метод HTTP и путь
- @OpenAPI(...) описание для Swagger (summary, description, теги, параметры)
- @ResponseSchema(...) тип возвращаемых данных (генерирует 200 ОК и структуру JSON)

```
@JsonController("/blog-posts")
export class BlogPostController extends BaseController<BlogPost> {
    private readonly blogPostService: BlogPostService;

    constructor() {
        super(new BlogPostService());
        this.blogPostService = this.service as BlogPostService;
    }

    @Get("/")
    @OpenAPI({ summary: "Get all blog posts" })
    @ResponseSchema(BlogPostResponseDto, { isArray: true })
    async getAll() {
        return this.blogPostService.findAllWithRelations();
    }
}
```

#### 2. DTO и class-validator влияют на входные схемы

Swagger получает метаинформацию и создаёт JSON Schema: типы, обязательность, описание.

- @IsEmail, @IsString → влияют на Swagger как требования к полям
- @Type(() => String)  $\rightarrow$  гарантирует правильный тип в спецификации

```
export class BlogPostResponseDto {
 @IsInt()
   @Type(() => Number)
 id: number:
 @IsInt()
   @Type(() => Number)
 author_id: number;
 @IsString()
   @Type(() => String)
 title: string;
 @IsString()
   @Type(() => String)
 content: string;
 @IsDate()
   @Type(() => Date)
 created_at: Date;
 @IsDate()
   @Type(() => Date)
 updated_at: Date;
```

#### 3. validationMetadatasToSchemas генерирует схемы из DTO

Эта часть создаёт раздел components.schemas в swagger.json — там описаны все DTO.

```
export function useSwagger(app: Application, options?: { controllers: any[] }) {
   try {
      const schemas = validationMetadatasToSchemas({
            classTransformerMetadataStorage: defaultMetadataStorage,
            refPointerPrefix: "#/components/schemas/",
      });
```

## 4. routingControllersToSpec → собирает всё в спецификацию

Это главный шаг — он создаёт весь swagger.json.

- controllers список всех контроллеров
- components.schemas DTO и валидации
- securitySchemes схемы авторизации (Bearer)

```
export function useSwagger(app: Application, options?: { controllers: any[] }) {
    const spec = routingControllersToSpec(
     storage,
        controllers: options?.controllers || [],
       defaultErrorHandler: false,
        components: {
         schemas,
         securitySchemes: {
           bearerAuth: {
             type: "http",
             scheme: "bearer",
             bearerFormat: "JWT",
           },
        },
        security: [{ bearerAuth: [] }],
        info: {
         title: "Fitness API",
         version: "1.0.0",
         description: "API documentation for the Fitness application",
        servers: [
            url: "http://localhost:3000",
           description: "Local server",
```

## 5. swagger-ui-express отображает интерфейс

Swagger доступен по адресу /docs

Можно визуально тестировать каждый маршрут

Можно вставить JWT-токен в UI и использовать защищённые @Authorized() методы

```
app.get("/swagger.json", (_req, res) => {
    res.json(spec);
});
app.use("/docs", swaggerUi.serve, swaggerUi.setup(spec));
} catch (error) {
    console.error("Error setting up Swagger:", error);
}

You, 6 days ago • feat: Lab1, Lab2, HW3, HW4 are done
```

#### **6.** В Swagger также настраивается авторизация

B Swagger UI появляется кнопка Authorize

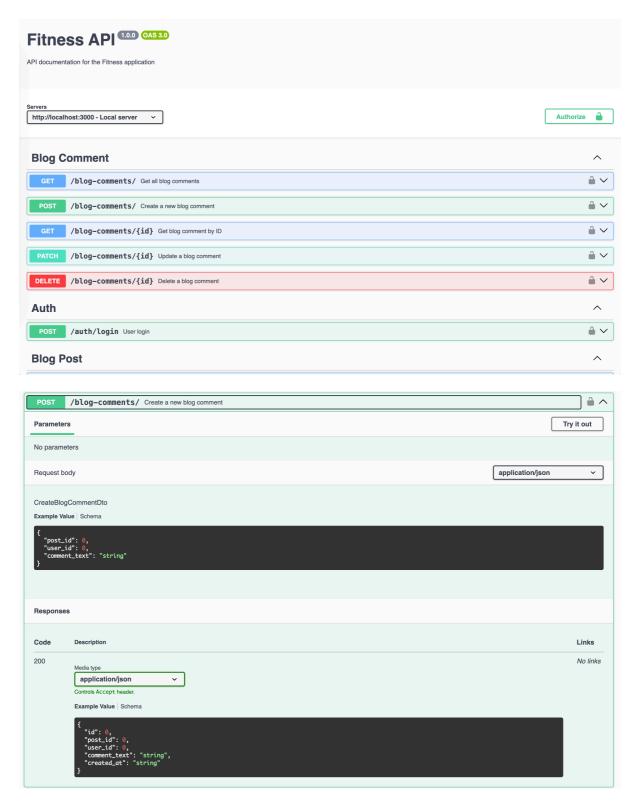
Токен передаётся автоматически в заголовке Authorization: Bearer ...

```
securitySchemes: {
    bearerAuth: {
        type: "http",
        scheme: "bearer",
        bearerFormat: "JWT",
     },
    },
    security: [{ bearerAuth: [] }],
```

## 7. Использование Swagger

```
useSwagger(app, {
    controllers: []
    BlogCommentController,
    AuthController,
    BlogPostController,
    UserController,
    OrderController,
    PaymentController,
    RoleController,
    WorkoutController,
    TrainingPlanController,
    UserTrainingPlanController,
    UserProgressController,
    TrainingPlanWorkoutController,
    TrainingPlanWorkoutController,
    TrainingPlanWorkoutController,
    TrainingPlanWorkoutController,
    TrainingPlanWorkoutController,
    TrainingPlanWorkoutController,
```

### Результаты:



#### **Postman**

Postman используется как инструмент для:

- Тестирования REST API вручную
- Подготовки и демонстрации коллекций запросов
- Документирования API для внешних команд (frontend, QA, заказчики)
- Интеграции с CI/CD и автотестами

Вместо ручного создания запросов в Postman, используется генерация коллекции из OpenAPI-спецификации (swagger.json).

**Инструмент - openapi2postmanv2** — утилита для конвертации спецификации OpenAPI → Postman Collection.

#### Makefile: Автоматизация процесса

Процесс автоматизирован с помощью Makefile-команд. Это исключает ручные ошибки и упрощает поддержку.

#### 1. Генерация swagger.json

Получает актуальную спецификацию АРІ с сервера.

```
swagger.json:
     curl http://localhost:3000/swagger.json -o
swagger.json
```

## 2. Конвертация в Postman Collection

```
postman.json: swagger.json
    openapi2postmanv2 -s swagger.json -o postman.json -p
```

Генерирует postman.json — коллекцию, которую можно открыть в Postman.

Флаг -р означает: "добавлять примеры" на основе схем из Swagger.

## 3. Обёртка в формат Postman API

```
wrap:
    jq '{ collection: . }' postman.json > wrapped-
postman.json
```

Оборачивает JSON в объект collection, как того требует Postman API.

#### 4. Синхронизация с Postman по API

Обновляет опубликованную коллекцию на Postman Cloud с помощью APIключа.

#### sync:

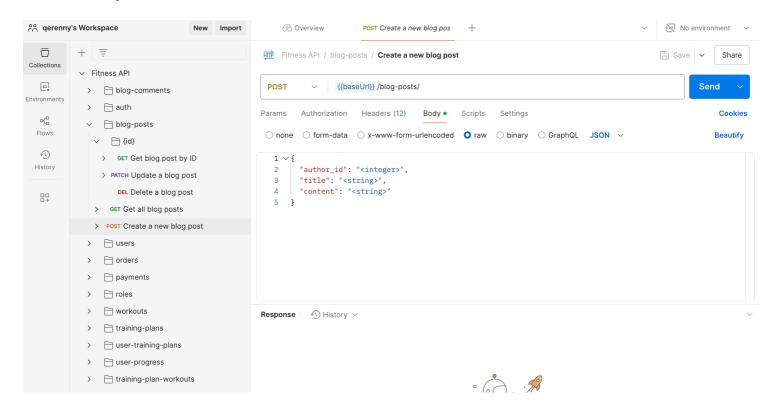
```
curl -X PUT https://api.getpostman.com/collections/$
(COLLECTION_UID) \
    -H "X-Api-Key: $(POSTMAN_API_KEY)" \
    -H "Content-Type: application/json" \
    -d @wrapped-postman.json
```

#### После выполнения команд:

- Автоматически создаётся актуальная коллекция Postman
- Swagger-спецификация и Postman остаются в синхронизации
- Все маршруты, тела запросов, ответы и авторизация уже настроены

```
labs > lab1 > № makefile
      You, 6 days ago | 1 author (You)
      POSTMAN_API_KEY := $(POSTMAN_API_KEY)
      COLLECTION_UID := $(COLLECTION_UID)
      swagger.json:
          curl http://localhost:3000/swagger.json -o swagger.json
      postman.json: swagger.json
          openapi2postmanv2 -s swagger.json -o postman.json -p
          jq '{ collection: . }' postman.json > wrapped-postman.json
      sync: swagger.json
          openapi2postmanv2 -s swagger.json -o postman.json -p
          jq '{ collection: . }' postman.json > wrapped-postman.json
          curl -X PUT https://api.getpostman.com/collections/$(COLLECTION_UID) \
            -H "X-Api-Key: $(POSTMAN_API_KEY)" \
            -H "Content-Type: application/json" \
            -d @wrapped-postman.json
           rm -f swagger.json postman.json wrapped-postman.json
```

## Результаты:



## Выводы