САНКТ-ПЕТЕРБУРГСКИЙ НАЦИОНАЛЬНЫЙ ИССЛЕДОВАТЕЛЬСКИЙ УНИВЕРСИТЕТ ИТМО

Дисциплина: Бэк-энд разработка

Отчет

Лабораторная работа №2. Реализация REST API на основе boilerplate

Выполнил:

Хиспметдинова Динара

Группа K3341

Проверил:
Добряков Д. И.

Санкт-Петербург

2025 г.

Задача

По выбранному варианту необходимо будет реализовать RESTful API средствами express + typescript (используя ранее написанный boilerplate).

Ход работы

Пул технологий:

NestJS (Layer-based)

TypeORM + PostgreSQL

Swagger (документация API)

JWT (авторизация)

bcrypt (хеширование пароля)

class-validator (валидация DTO)

Guards (JWT + роли)

Decorators (кастомный @Roles и @Public)

В соответствии с ER-диаграммой были реализованы следующие модели:

- User

- Appointment

- Psychologist

- Message

- Chat

- Review

- Specialization

- Schedule

- PsychologistSpecialization

Пример реализации CRUD для модели Appointment

Была реализована сущность Appointment, DTO классы для создания и обновления записей,

сервис и контроллер с методами: - POST /appointments — создание встречи - GET /appointments — получение всех - GET /appointments/:id — получение по ID - PUT /appointments/:id — обновление - DELETE /appointments/:id — удаление

С помощью декораторов @ApiProperty и @ApiTags API документировано. Все эндпоинты и

схемы запросов/ответов отображаются в Swagger UI по адресу http://localhost:3000/api.

Архитектура авторизации

JWT стратегия

Была реализована с использованием passport-jwt через JwtStrategy, которая:

Извлекает токен из заголовка Authorization (Bearer TOKEN)

Проверяет подпись токена через секрет JWT_SECRET из .env

Валидирует payload и передаёт его в контроллер как req.user

```
import { Injectable } from '@nestjs/common';
import { PassportStrategy } from '@nestjs/passport';
import { ExtractJwt, Strategy } from 'passport-jwt';
import { ConfigService } from '@nestjs/config';
import { UserService } from '../services/user.service';
import { JwtPayload } from '../types/jwt-payload.type';

@Injectable()
export class JwtStrategy extends PassportStrategy(Strategy) {
  constructor(
    configService: ConfigService,
    private readonly userService: UserService,
  ) {
    const jwtSecret = configService.get<string>('JWT_SECRET');
    if (!jwtSecret) {
      throw new Error('JWT_SECRET is undefined');
    }
    super({
      jwtFromRequest: ExtractJwt.fromAuthHeaderAsBearerToken(),
      secretOrKey: jwtSecret,
      ignoreExpiration: false,
    });
  }

  async validate(payload: JwtPayload): Promise<JwtPayload> {
    return await Promise.resolve(payload);
  }
}
```

Local стратегия

Используется при логине (/auth/login):

Извлекает email и password из тела запроса

Вызывает AuthService.validateUser()

Сравнивает пароль с хешем через bcrypt.compare

Если валидно — передаёт пользователя в контроллер

strategies > TS local.strategy.ts > ...

```ts
import { Injectable, UnauthorizedException } from '@nestjs/common';
import { PassportStrategy } from '@nestjs/passport';
import { Strategy as LocalStrategyBase } from 'passport-local';
import { AuthService } from '../services/auth.service';

@Injectable()
export class LocalStrategy extends PassportStrategy(
  LocalStrategyBase,
  'local',
) {
  constructor(private authService: AuthService) {
    super({ usernameField: 'email' });
  }

  async validate(email: string, password: string) {
    const user = await this.authService.validateUser(email, password);
    if (!user) {
      throw new UnauthorizedException();
    }
    return user;
  }
}
```

Guard'ы

JwtAuthGuard — проверяет наличие и валидность JWT.

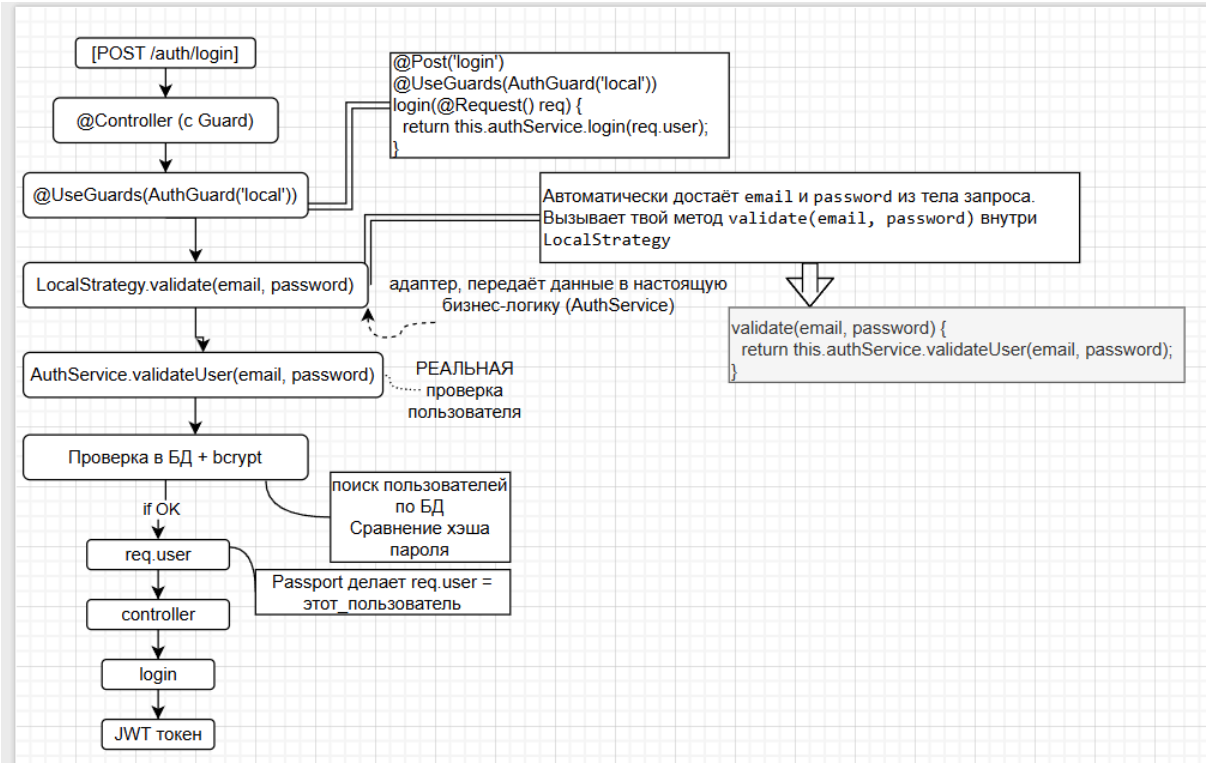
guards > TS jwt-auth.guard.ts > ...

```ts
import { ExecutionContext, Injectable } from '@nestjs/common';
import { Reflector } from '@nestjs/core';
import { AuthGuard } from '@nestjs/passport';
import { IS_PUBLIC_KEY } from '../decorators/public.decorator';

@Injectable()
export class JwtAuthGuard extends AuthGuard('jwt') {
  constructor(private reflector: Reflector) {
    super();
  }

  canActivate(context: ExecutionContext) {
    const isPublic = this.reflector.getAllAndOverride<boolean>(IS_PUBLIC_KEY, [
      context.getHandler(),
      context.getClass(),
    ]);
    if (isPublic) return true;
    return super.canActivate(context);
  }
}
```

RolesGuard — проверяет, входит ли пользователь в разрешённые роли для эндпоинта.

@Roles(Role.Administrator) — используется для ограничения доступа (например, создание специализаций).

@Public() — отключает защиту Guard'ом на отдельных маршрутах (register, login).



Управление ролями

- По умолчанию при регистрации пользователю присваивается роль user.
- Только администраторы могут:
  - Создавать/редактировать/удалять специализации
  - Удалять пользователей
  - Видеть всех пользователей

Guard на основе `@Roles` проверяет наличие роли в `req.user.role`.

Реализация сущностей

User: email, password_hash, full_name, phone, role

Psychologist: связанный 1:1 с User, имеет опыт, цену, bio

Specialization: доступно только для администратора

PsychologistSpecialization: связь многие-ко-многим

Appointment: клиент записывается к психологу

Review: клиент оставляет отзыв

Schedule: график приёма психолога

Chat / Message: структура для общения

src/

├── controllers/

├── dto/

├── enums/

├── guards/

├── models/

├── routes/

├── services/

├── strategies/

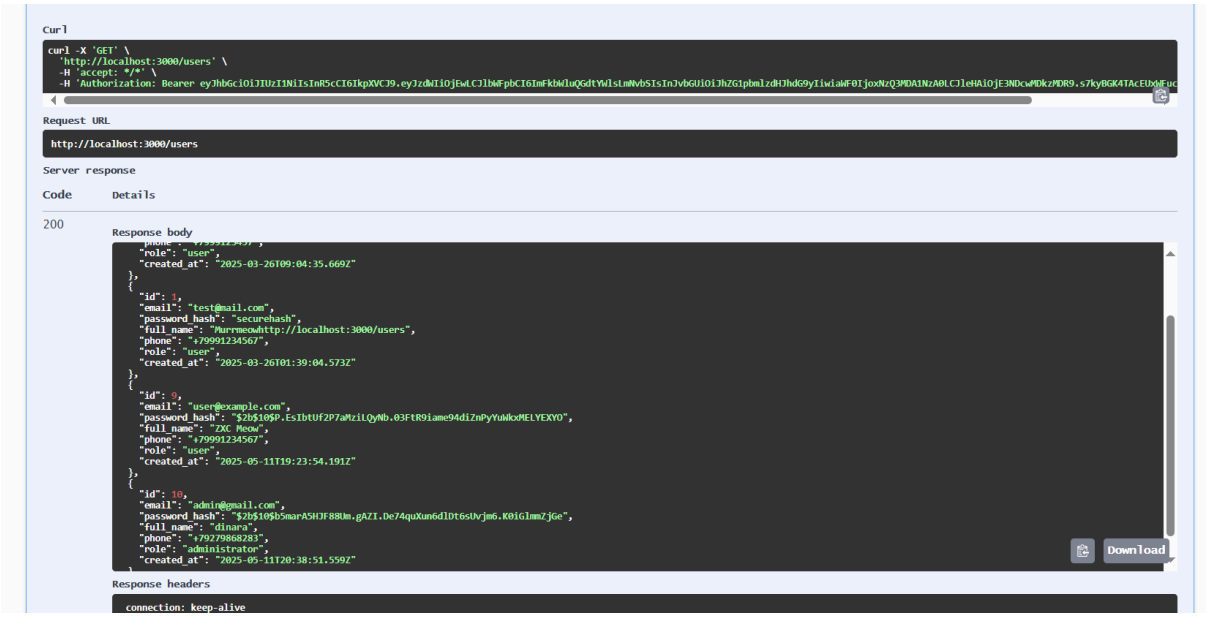
├── types/

└── main.ts

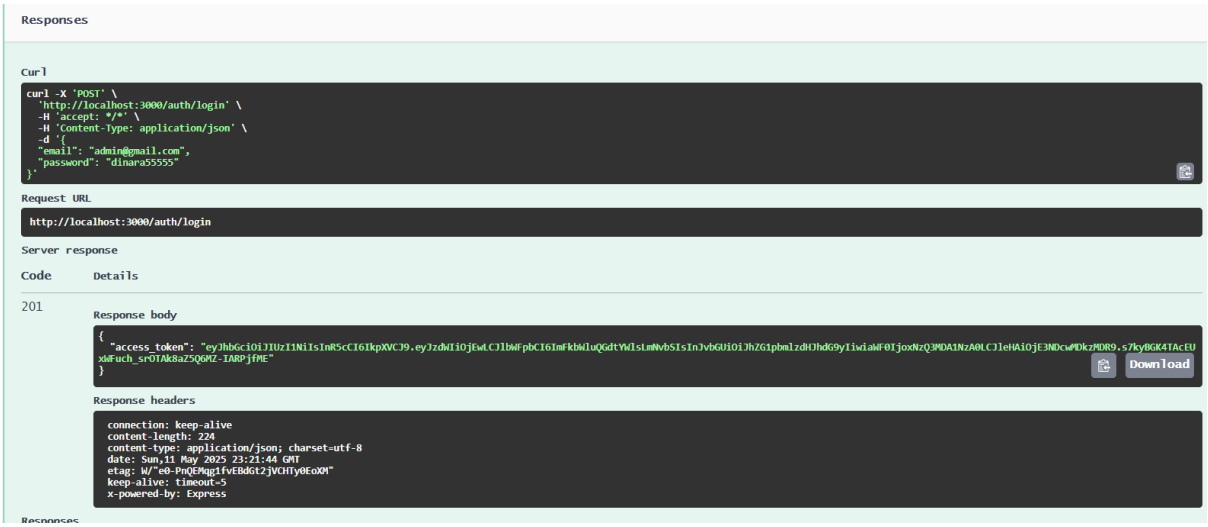
Тестирование API

Тестирование производилось через Swagger и Postman:

- Все защищённые маршруты требуют токен (`Authorize → Bearer`)



- Публичные (`/auth/register`, `/auth/login`) доступны без авторизации
- При попытке доступа к `POST /specializations` без токена → 401 Unauthorized



Особенности реализации

- Использован APP_GUARD в AppModule, чтобы глобально применять JwtAuthGuard и RolesGuard.
- Для Swagger требовалась дополнительная настройка @ApiBearerAuth() и @ApiBody().
- Ошибки eslint@typescript-eslint/no-unsafe-* устранялись за счёт явной типизации (: Promise<Specialization[]>, и т.п.)

- Role вынесен в enum, используется в декораторах и DTO.

Вывод

В ходе работы:

- Реализована безопасная аутентификация на основе JWT
- Защита маршрутов через Guard’ы с учётом ролей
- Описаны все сущности и CRUD-операции
- Обеспечено документирование API с помощью Swagger
- Проведено тестирование авторизации и ролей