**Документация по аутентификации и авторизации с использованием Access и Refresh токенов**

**Введение**

Документация описывает процесс регистрации, авторизации, генерации и обновления Access и Refresh токенов в приложении на Nest.js с использованием MongoDB. Использование WebSocket для создания двустороннего обмена данными между клиентом и сервером.

1. Регистрация пользователя

**Цель**

Регистрация нового пользователя в системе с сохранением данных в базе данных.

**Процесс**

1. Получение данных от пользователя:

- Пользователь отправляет email и пароль через эндпоинт `/auth/register`.

2. Проверка существования пользователя:

- Проверяется, существует ли пользователь с указанным email.

- Если пользователь существует, возвращается ошибка.

3. Хеширование пароля:

- Пароль шифруется с помощью bcrypt (алгоритм хеширования).

4. Создание пользователя:

- Сохраняются данные в коллекцию users MongoDB с ролью по умолчанию `user`.

**Пример запроса**

POST /auth/register

Body:

{

"email": "user@example.com",

"password": "securepassword123"

}

**Пример ответа**

{

"\_id": "64e5d1a2e8e0a1b9c3f0d8e9",

"email": "user@example.com",

"passwordHash": "$2b$10$...",

"role": "user"

}

2. Авторизация пользователя

**Цель**

Авторизация существующего пользователя и генерация Access и Refresh токенов.

**Процесс**

1. Проверка учетных данных:

- Пользователь отправляет email и пароль через эндпоинт `/auth/login`.

- Валидируется email и сравнивается хэш пароля с сохраненным в БД.

2. Генерация токенов:

- Access Token: Короткий срок действия (например, 15 минут).

- Refresh Token: Длительный срок действия (например, 7 дней).

3. Сохранение Refresh Token:

- Refresh Token сохраняется в HTTP-кookie с флагом `httpOnly` для безопасности.

**Пример запроса**

POST /auth/login

Body:

{

"email": "user@example.com",

"password": "securepassword123"

}

**Пример ответа**

json

{

"accessToken": "eyJhbGciOiJIUzI1NiIsInR5cCI6IkpXVCJ9..."

}

Cookie:

- `refreshToken=eyJhbGciOiJIUzI1NiIsInR5cCI6IkpXVCJ9...` (доступен только на сервере).

3. Access Token

**Описание**

- Цель: Используется для аутентификации в защищенных эндпоинтах.

- Срок действия: Краткосрочный (например, 15 минут).

- Структура:

json

{

"sub": "64e5d1a2e8e0a1b9c3f0d8e9", // ID пользователя

"email": "user@example.com",

"exp": 1717023200 // Unix timestamp

}

**Использование**

- Передается в заголовке `Authorization: Bearer <accessToken>`.

- При истечении срока действия необходимо обновить через Refresh Token.

4. Refresh Token

**Описание**

- Цель: Обновление Access Token без повторной авторизации.

- Срок действия: Длительный (например, 7 дней).

- Хранение:

- Сохраняется в HTTP-кookie с флагом `httpOnly` (недоступен клиентскому JavaScript).

- Лучше хранить в базе данных для черного списка (blacklist).

**Процесс обновления**

1. Отправка запроса:

- Клиент отправляет Refresh Token через эндпоинт `/auth/refresh`.

2. Валидация токена:

- Проверяется подпись и срок действия.

3. Генерация новых токенов:

- Создаются новые Access и Refresh токены.

4. Обновление Refresh Token:

- Старый Refresh Token добавляется в черный список (или обновляется в БД).

**Пример запроса**

POST /auth/refresh

**Пример ответа**

json

{

"accessToken": "eyJhbGciOiJIUzI1NiIsInR5cCI6IkpXVCJ9..."

}

Cookie:

- Обновленный `refreshToken` (если требуется).

5. Безопасность

**Основные меры**

- Хеширование паролей: Используется bcrypt с солью.

- HTTP-Only Cookie: Refresh Token недоступен клиентскому JavaScript, защищая от XSS-атак.

- Blacklist Refresh Token: Устаревшие/небезопасные токены добавляются в черный список.

- HTTPS: Всегда используйте защищенный протокол для передачи токенов.

6. Схема взаимодействия

**Пользователь**

↓ **Регистрация**

Создание аккаунта → Хеширование пароля → Сохранение в MongoDB

↓ **Авторизация**

Ввод email/password → Проверка → Генерация Access/Refresh Token → Отправка Access в ответ + Refresh в Cookie

↓

**Использование защищенных** **эндпоинтов** (с заголовком Authorization: Bearer <Access>)

↓ **При истечении Access**

Отправка Refresh Token → Проверка → Генерация новых токенов → Обновление Cookie

7. Валидация DTO

**Register DTO**

import { IsEmail, IsString, MinLength } from 'class-validator';

export class RegisterDto {

@IsEmail()

email: string;

@IsString()

@MinLength(6)

password: string;

}

**Login DTO**

import { IsEmail, IsString } from 'class-validator';

export class LoginDto {

@IsEmail()

email: string;

@IsString()

password: string;

}

8. Важные моменты

- Срок действия токенов:

- Access Token: Устанавливается в конфигурации JWT (`expiresIn: '15m'`).

- Refresh Token: Устанавливается в конфигурации JWT (`expiresIn: '7d'`).

- Черный список Refresh Token:

- После использования Refresh Token его следует добавлять в черный список, чтобы предотвратить повторное использование.

- Обновление Refresh Token:

- При каждом обновлении генерируйте новый Refresh Token и обновите его в БД/cookie.

9. Примеры кода

Генерация токенов (auth.service.ts)

generateTokens(user: User): Promise<{ accessToken: string; refreshToken: string }> {

return Promise.all([

this.jwtService.signAsync({ sub: user.\_id, email: user.email }, {

secret: 'accessSecret',

expiresIn: '15m',

}),

this.jwtService.signAsync({ sub: user.\_id }, {

secret: 'refreshSecret',

expiresIn: '7d',

}),

]);

}

Обработка Refresh Token (auth.controller.ts)

@Post('refresh')

@UseGuards(RefreshGuard) // Стратегия для проверки Refresh Token

async refresh(@Response() res): Promise<{ accessToken: string }> {

const user = await this.authService.getCurrentUser(); // Получение пользователя из стратегии

const tokens = await this.authService.generateTokens(user);

res.cookie('refreshToken', tokens.refreshToken, { httpOnly: true });

return { accessToken: tokens.accessToken };

}

Краткая инструкция по запуску проекта

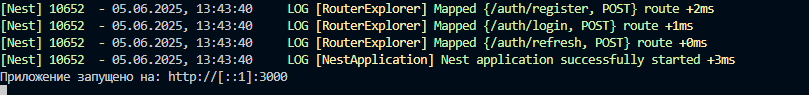
Запуск mongoDB и делаем connect к базе test

Заходим в папку с проектом и в консоле прописываем команды

npm install

npm run start:dev

Сервер запущен



Заходим в папку frontend. Открываем страницу registration – для регистрации, index – для входа.

БД: test