ФЕДЕРАЛЬНОЕ ГОСУДАРСТВЕННОЕ АВТОНОМНОЕ ОБРАЗОВАТЕЛЬНОЕ УЧРЕЖДЕНИЕ ВЫСШЕГО ОБРАЗОВАНИЯ

МОСКОВСКИЙ ПОЛИТЕХНИЧЕСКИЙ УНИВЕРСИТЕТ

Факультет Информационных технологий
Кафедра Информатики и информационных технологий

направление подготовки
09.03.02 «Информационные системы и технологии»

ЛАБОРАТОРНАЯ РАБОТА № 13

Дисциплина: «Backend»

Тема: *Добавление авторизации и аутентификации в веб-приложение на платформе ASP.NET Core*

Выполнил: студент группы: 231-339

Карапетян Нвер Каренович
(Фамилия И.О.)

Дата, подпись: 13.04.25 _____
(Дата) (Подпись)

Проверил: _____ _____
(Фамилия И.О., степень, звание) (Оценка)

Дата, подпись _____ _____
(Дата) (Подпись)

Москва
2025

Цель:

Освоить процесс добавления механизмов аутентификации и авторизации в веб-приложение на платформе ASP.NET Core для обеспечения безопасности приложения и данных пользователей.

Задачи:

- Настроить механизм аутентификации для приложения, используя встроенные средства ASP.NET Core.
- Реализовать механизм авторизации с установкой различных уровней доступа для различных ролей пользователей.
- Использовать атрибуты авторизации для ограничения доступа к определенным частям приложения для разных пользователей.
- Протестировать работу механизмов аутентификации и авторизации на разных уровнях доступа.

Ход работы

В основе проекта лежит простое веб-приложение, использующее ASP.NET Core, в котором данные пользователей хранятся в базе PostgreSQL с управлением через Entity Framework Core. Реализована регистрация новых пользователей, аутентификация с помощью JWT-токенов и механизм обновления токенов. Для разграничения доступа к API применяется атрибут авторизации, который разрешает доступ определённым эндпоинтам только пользователям с заданной ролью (например, «Admin»).

Модель данных

Основной сущностью является класс User, который представляет пользователя системы. Помимо стандартных полей (идентификатора, имени пользователя и хэшированного пароля) модель содержит поле для роли, а также данные, связанные с refresh-токеном и временем его истечения.

Листинг 1. Модель User.

```
public class User
{
    public Guid Id { get; set; }
    public string Username { get; set; } = string.Empty;
    public string PasswordHash { get; set; } = string.Empty;
    public string Role { get; set; } = string.Empty;
    public string? RefreshToken { get; set; } = string.Empty;
    public DateTime? RefreshTokenExpiryTime { get; set; }
}
```

Для регистрации и входа в систему используется DTO (Data Transfer Object), который содержит минимальный набор данных (имя пользователя и пароль):

Листинг 2. DTO модель UserDto.

```
public class UserDto
{
    public string Username { get; set; } = string.Empty;
    public string Password { get; set; } = string.Empty;
}
```

Реализация авторизации и аутентификации

Контроллер `AuthorizationController` объединяет логику для регистрации, аутентификации, обновления токенов и проверки уровня доступа.

1. **Регистрация пользователя.**

При регистрации пользователь отправляет на сервер данные (имя и пароль) через DTO. В контроллере запрос перенаправляется в сервис регистрации.

Метод проверяет, существует ли уже пользователь с заданным именем в базе данных. Это предотвращает дублирование учетных записей. Если пользователь отсутствует, перед сохранением пароль хэшируется с использованием класса `PasswordHasher<User>`. Это повышает безопасность, поскольку в базе данных сохраняется не сам пароль, а его хэш.

В конечном счете новый пользователь добавляется в контекст базы данных и сохраняется с помощью метода `SaveChangesAsync()`.

Листинг 3. POST-метод регистрации в AuthorizationController.

```
[HttpPost("register")]
public async Task<ActionResult<User>> Register(UserDto request)
```

```
{
    var user = await _authService.RegisterAsync(request);

    if (user is null)
        return BadRequest("Username already exists!");

    return Ok(user);
}
```

В сервисе регистрации (скрипт `AuthService.cs`) код выглядит следующим образом:

Листинг 4. Метод регистрации из сервиса AuthService.

```
public async Task<User?> RegisterAsync(UserDto request)
{
    var userExists = await _context.Users.AnyAsync(user => user.Username == re-
quest.Username);

    if (userExists is true)
        return null;

    var user = new User();

    var hashedPassword = new PasswordHasher<User>()
        .HashPassword(user, request.Password);

    user.Username = request.Username;
    user.PasswordHash = hashedPassword;

    await _context.Users.AddAsync(user);
    await _context.SaveChangesAsync();

    return user;
}
```

Здесь основное внимание уделено проверке уникальности имени пользователя и безопасному хранению пароля. Если регистрация прошла успешно, создается запись в таблице `Users` с информацией о новом пользователе.

2. **Вход в систему (логин).**

При входе в систему клиент отправляет имя и пароль. Приложение ищет в базе данных пользователя с заданным именем, а введённый пароль сравнивается

с хэшированным паролем из базы с помощью метода `VerifyHashedPassword`. Если пароли совпадают, проверка успешна.

При успешной аутентификации генерируется токен, который включает Claim'ы пользователя (идентификатор, имя и роль). Этот токен возвращается клиенту.

Листинг 5. POST-метод входа в систему в AuthorizationController.

```
[HttpPost("login")]
public async Task<ActionResult<TokenResponseDto>> Login(UserDto request)
{
    var response = await _authService.LoginAsync(request);

    if (response is null)
        return BadRequest("Invalid username and/or password!");

    return Ok(response);
}
```

А вот фрагмент из сервиса `AuthService`, который отвечает за вход:

Листинг 6. Метод входа в систему из сервиса AuthService.

```
public async Task<TokenResponseDto?> LoginAsync(UserDto request)
{
    var user = await _context.Users.FirstOrDefaultAsync(u => u.Username == re-
quest.Username);

    if (user is null)
        return null;

    if (new PasswordHasher<User>().VerifyHashedPassword(user, user.PasswordHash,
request.Password) == PasswordVerificationResult.Failed)
        return null;

    return await CreateTokenResponse(user);
}
```

После проверки учетных данных, если данные верны, вызывается создание токенов. JWT-токен содержит информацию, необходимую для дальнейшей авторизации, а refresh-токен используется для обновления access-токена при его истечении.

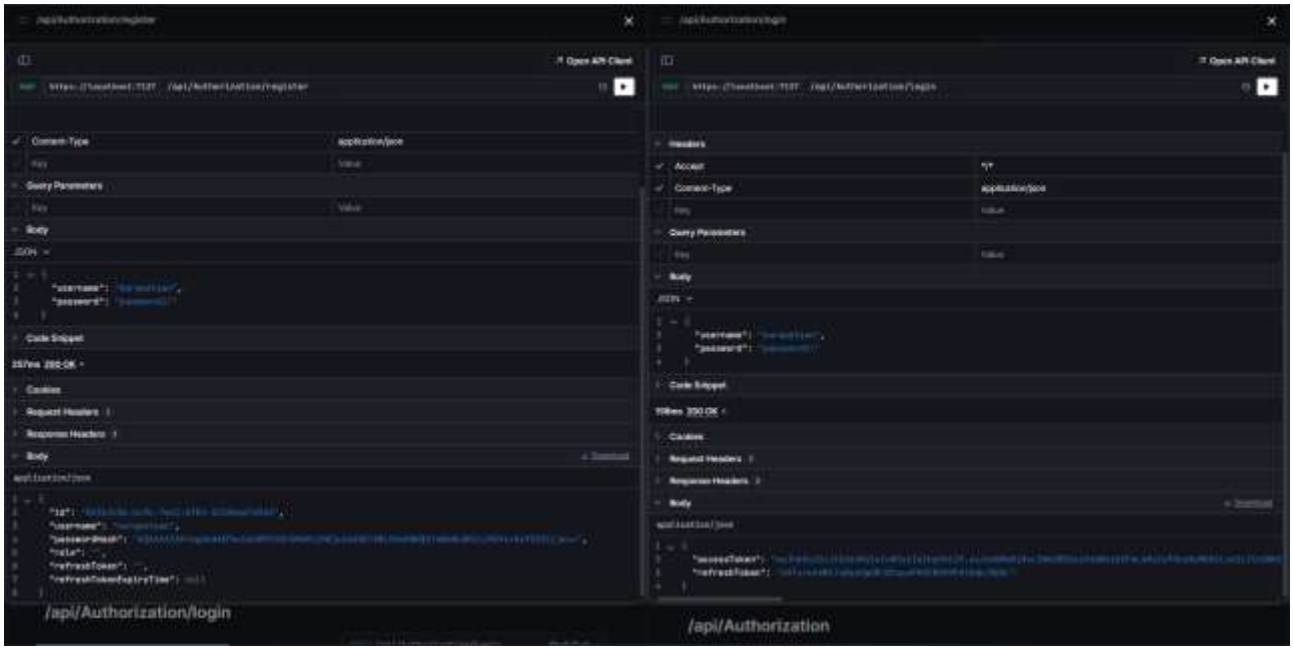

Рисунок 1. Успешно выполненные запросы регистрации (слева) и входа в систему (справа).

Доступ к защищенным эндпоинтам

Приложение имеет эндпоинты, доступ к которым ограничен с помощью атрибутов авторизации.

Листинг 7. Защищенные HTTP-запросы из контроллера AuthorizationController.

```
[HttpGet]
[Authorize]
public IActionResult AuthenticatedOnlyEndpoint()
{
    return Ok("You are authenticated!");
}

[HttpGet("admin-only")]
[Authorize(Roles = "Admin")]
public IActionResult AdminOnlyEdpoint()
{
    return Ok("You are admin!");
}
```

Эндпоинты, помеченные атрибутом `[Authorize]`, доступны только для аутентифицированных пользователей. Если пользователь отправляет запрос без валидного токена, сервер возвращает ошибку авторизации.

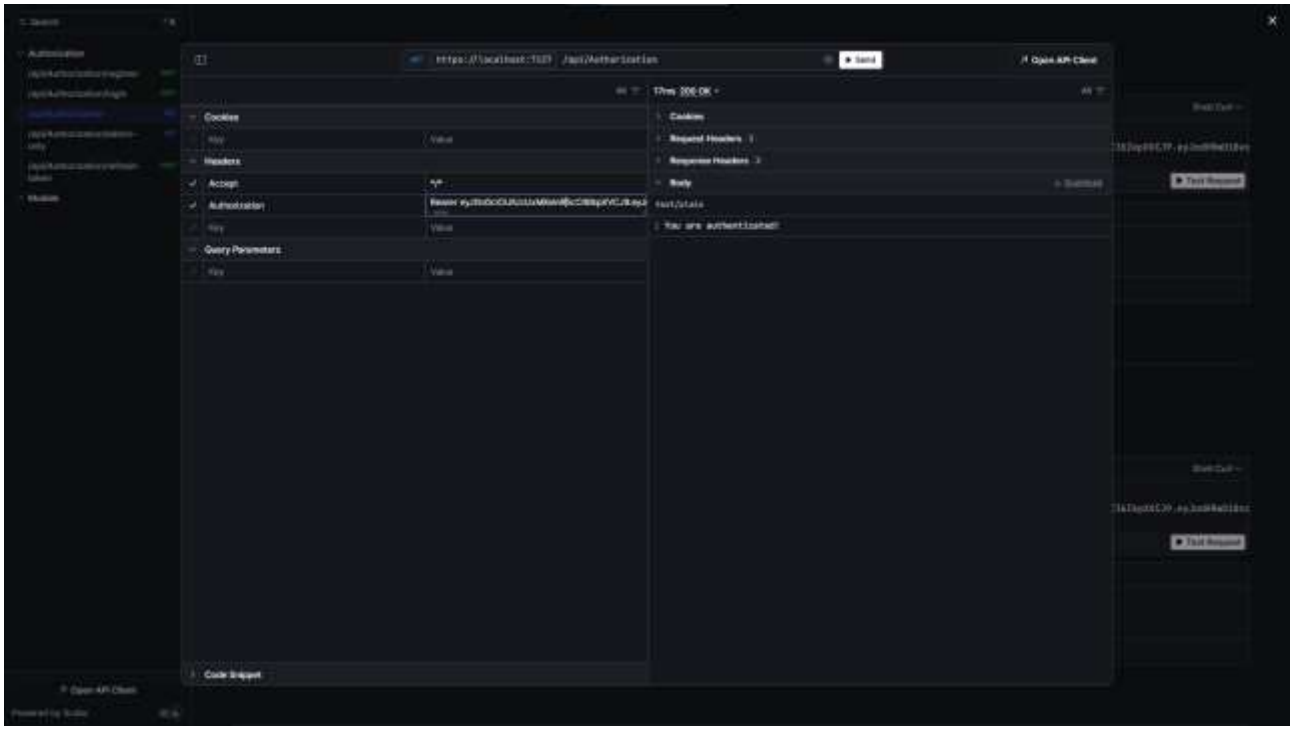

Рисунок 2. Успешно отправленный запрос к защищенному эндпоинту, помеченному атрибутом Authorize.

Эндпоинт с атрибутом `[Authorize(Roles = "Admin")]` доступен только пользователям, чьи токены содержат Claim с ролью «Admin». Это позволяет разграничивать права доступа.

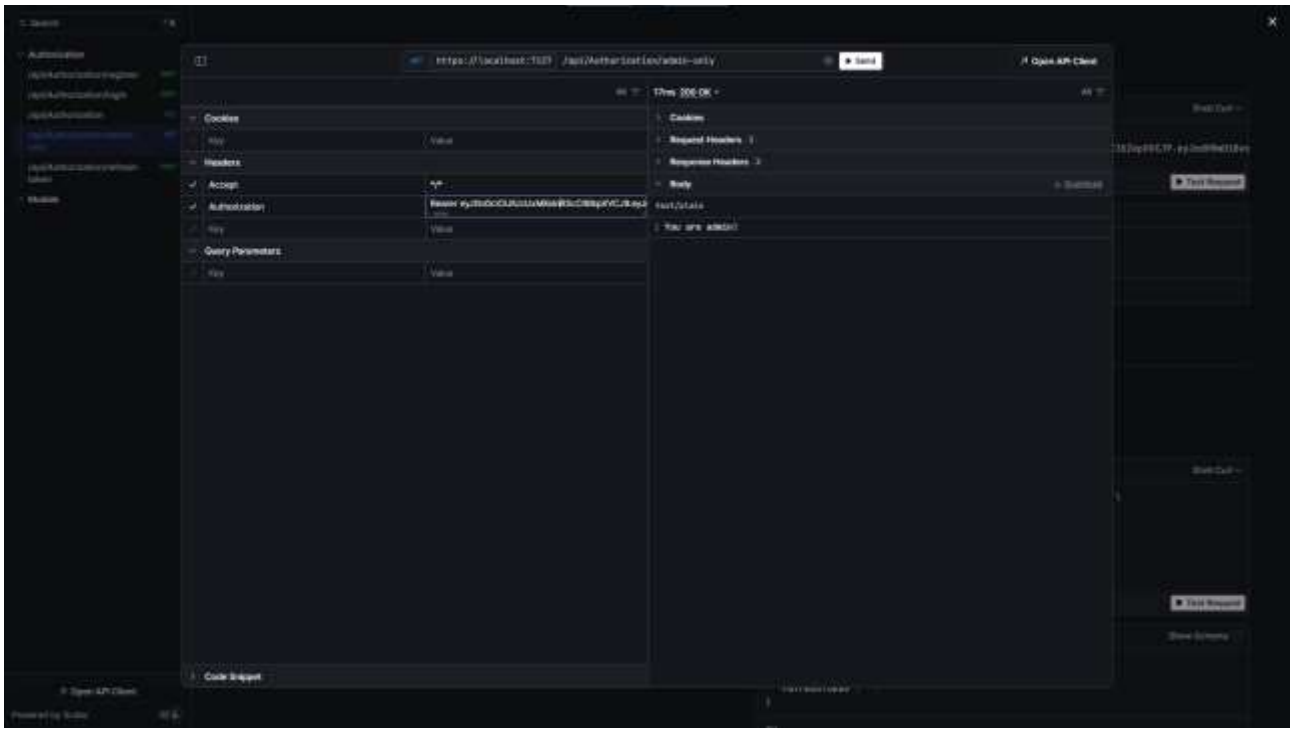

Рисунок 3. Успешно отправленный запрос к защищенному эндпоинту, доступному только администрам.

При обращении к вышеописанным эндпоинтам происходит проверка в middleware ASP.NET Core, которая анализирует HTTP-заголовок `Authorization`.

Если токен валиден и роль пользователя соответствует требуемой (в случае `admin-only`), запрос обрабатывается, и клиент получает соответствующий ответ. В противном случае сервер вернёт ошибку 401 (Unauthorized) или 403 (Forbidden).

Генерация и валидация токенов

Сервис `AuthService` содержит всю бизнес-логику, связанную с регистрацией, входом в систему и обновлением токенов. При входе проверяются данные пользователя, после чего формируется JWT-токен с использованием секретного ключа и заданным временем истечения. Также реализована генерация refresh-токена, позволяющего обновить access-токен без повторного ввода учетных данных.

Основные этапы работы скрипта создания токена

1. **Формирование Claim'ов** — создаются Claim'ы, содержащие информацию о пользователе (идентификатор, имя, роль). Это позволяет системе в дальнейшем проверять права доступа при обращении к защищённым ресурсам.
2. **Подготовка ключа для подписи** — секретный ключ, указанный в конфигурации, используется для создания симметричного ключа, который затем применяется для подписи токена с алгоритмом HMACSHA512. Это гарантирует целостность и подлинность токена.
3. **Формирование JWT-токена** — с помощью `JwtSecurityToken` создаётся токен, в который включены все данные, а также указываются параметры издателя, получателя и время истечения токена. После этого токен сериализуется в строку и возвращается клиенту.

Листинг 8. Метод, отвечающий за создание токена из скрипта AuthService.

```
private string CreateToken(User user)
{
    var claims = new List<Claim>
    {
        new Claim(ClaimTypes.NameIdentifier, user.Id.ToString()),
        new Claim(ClaimTypes.Name, user.Username),
        new Claim(ClaimTypes.Role, user.Role)
    };
```

```
    var key = new SymmetricSecurityKey(Encoding.UTF8.GetBytes(_configuration.Get-
Value<string>("JwtSettings:SigningKey")!));
    var creds = new SigningCredentials(key, SecurityAlgorithms.HmacSha512);

    var tokenDescriptor = new JwtSecurityToken(
        issuer: _configuration.GetValue<string>("JwtSettings:Issuer"),
        audience: _configuration.GetValue<string>("JwtSettings:Audience"),
        claims: claims,
        expires: DateTime.UtcNow.AddMinutes(3),
        signingCredentials: creds);

    return new JwtSecurityTokenHandler().WriteToken(tokenDescriptor);
}
```

На сайте jwt.io можно декодировать содержимое JWT-токена и подробно изучить его содержимое:

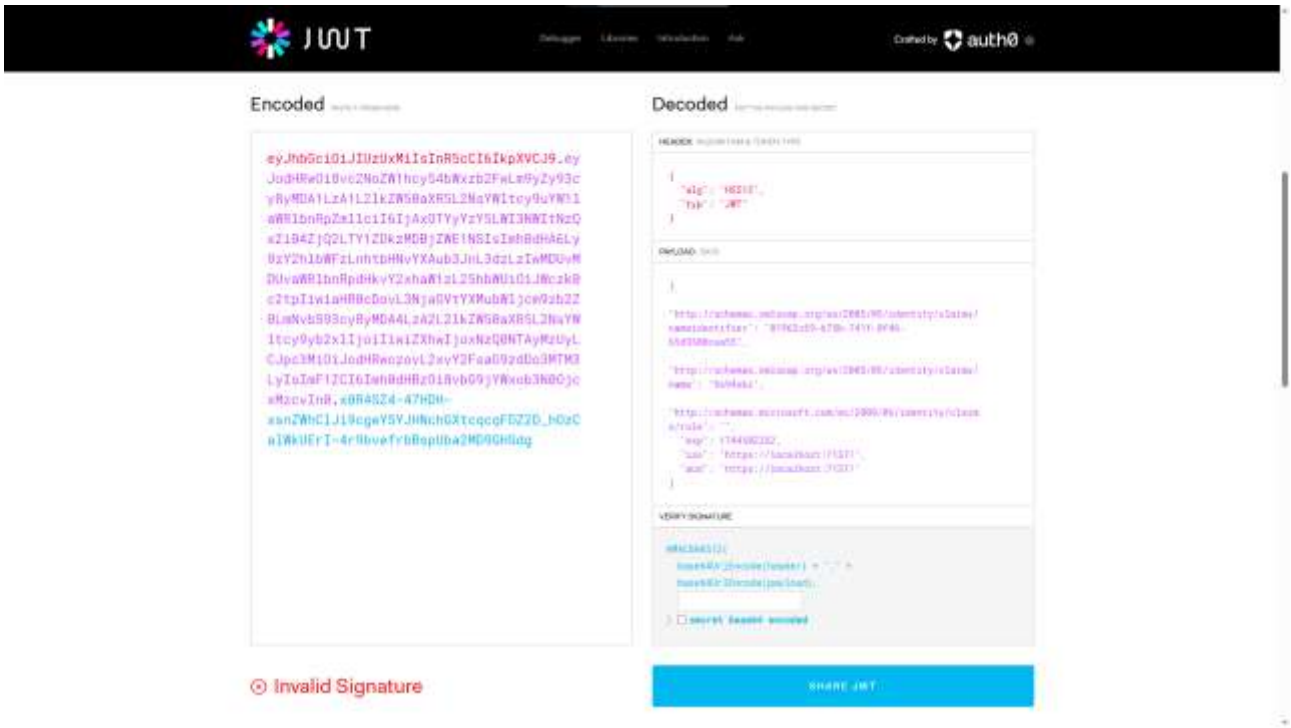

Рисунок 4. Расшифровка сгенерированного JWT-токена.

Конфигурация приложения

Основная настройка приложения производится в файле `Program.cs`, где регистрируются контроллеры, сервисы и настраивается подключение к базе данных. Здесь же осуществляется настройка схемы аутентификации, которая использует JWT Bearer токены.

Листинг 9. Фрагмент настройки аутентификации из Program.

```
builder.Services.AddAuthentication(JwtBearerDefaults.AuthenticationScheme)
    .AddJwtBearer(options =>
    {
        options.TokenValidationParameters = new TokenValidationParameters
        {
            ValidateIssuer = true,
            ValidIssuer = builder.Configuration["JwtSettings:Issuer"],
            ValidateAudience = true,
            ValidAudience = builder.Configuration["JwtSettings:Audience"],
            ValidateLifetime = true,
            IssuerSigningKey = new SymmetricSecurityKey(Encoding.UTF8.Get-
Bytes(builder.Configuration["JwtSettings:SigningKey"]!)),
            ValidateIssuerSigningKey = true
        };
    });
```

При этом в конфигурационном файле `appsettings.json` задаются все необходимые параметры, такие как строка подключения к PostgreSQL, параметры JWT (SigningKey, Issuer, Audience) и настройки логирования.

Листинг 10. Конфигурационный файл appsettings.

```
{
  "ConnectionStrings": {
    "Default": "Host=localhost;Port=5432;Database=Users;Username=postgres;Pass-
word=SpiLlett777"
  },
  "Logging": {
    "LogLevel": {
      "Default": "Information",
      "Microsoft.AspNetCore": "Warning"
    }
  },
  "AllowedHosts": "*",
  "JwtSettings": {
    "SigningKey":
"d5be7e771f8fd195502ec156b43d76f4b2335da20d423cccc3f44df1b2404f70f48954e0096facb4cf
d19094eb71209b4b108182f8a29f664f9959a0e9aac9bd",
    "Issuer": "https://localhost:7137/",
    "Audience": "https://localhost:7137/"
  }
}
```

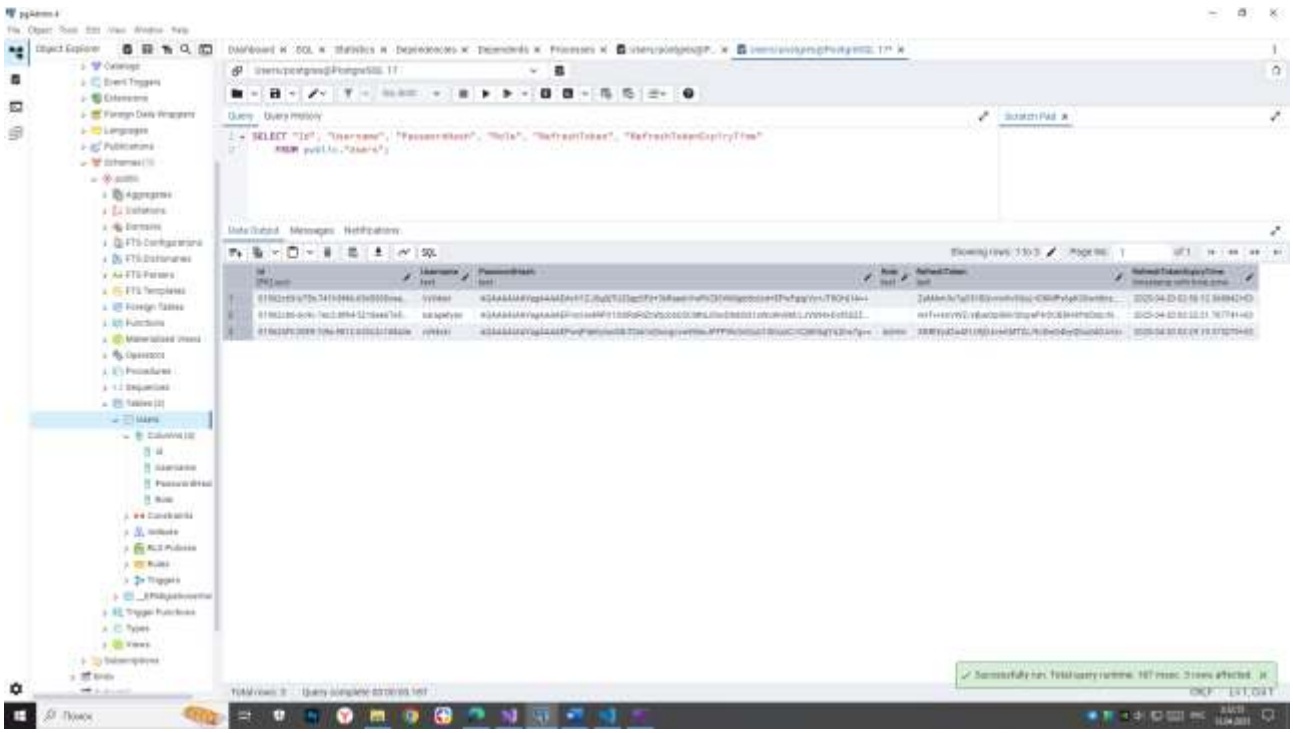

Рисунок 5. Записи в базе данных.