ASP.NET Core. Веб-приложения

Аутентификация и авторизация



# На этом уроке

1. Освоим основы безопасности
2. Узнаем, что такое аутентификация и авторизация и чем они отличаются
3. Разберёмся, зачем нужны токен обновления и токен доступа
4. Потренируемся с созданием токенов в консольном приложении
5. Рассмотрим работу с токенами в Web API сервисе

# 

# 

# 

# 

# Введение в безопасность

Безопасность приложения – один из самых важных его элементов. Никогда не стоит ею пренебрегать, иначе последствия могут быть необратимы. Например, похищение злоумышленниками пользовательской информации и других критических для бизнеса данных. В этом уроке познакомимся с базовыми понятиями по аутентификации и авторизации пользователя в приложении на базе JWT token. Это один из самых распространённых методов авторизации и аутентификации в наши дни, если не считать самописные методы, которые крайне не рекомендуется использовать.

Для начала нужно разобраться с определениями:

**Аутентификация** – это процесс проверки подлинности пользователя: «кто ты есть», а именно проверяет пару логин и пароль.

**Авторизация** – это процесс выдачи прав прав пользователю на те или иные действия. Во время него система проверяет, может ли он совершить запрошенное им действие.

Ни в коем случае не стоит путать эти два термина, особенно на собеседовании.

# Где и как хранятся пароли

Пароли и логины пользователей в основном хранятся в базах данных. Сами логины хранятся не в зашифрованном виде, а вот пароли – в зашифрованном. Почему так? Ответ достаточно прост: если базу все же украдет злоумышленник, то у него не будет возможности быстро расшифровать пользовательские пароли, что уменьшит вероятность дальнейшей атаки.

|  |
| --- |
| create table "Geekbrains"."Users"  (  id serial not null,  "Login" text not null,  "Password" text not null  );    create unique index users\_id\_uindex  on "Test"."Users" (id);    create unique index users\_login\_uindex  on "Test"."Users" ("Login");    alter table "Test"."Users"  add constraint users\_pk  primary key (id); |

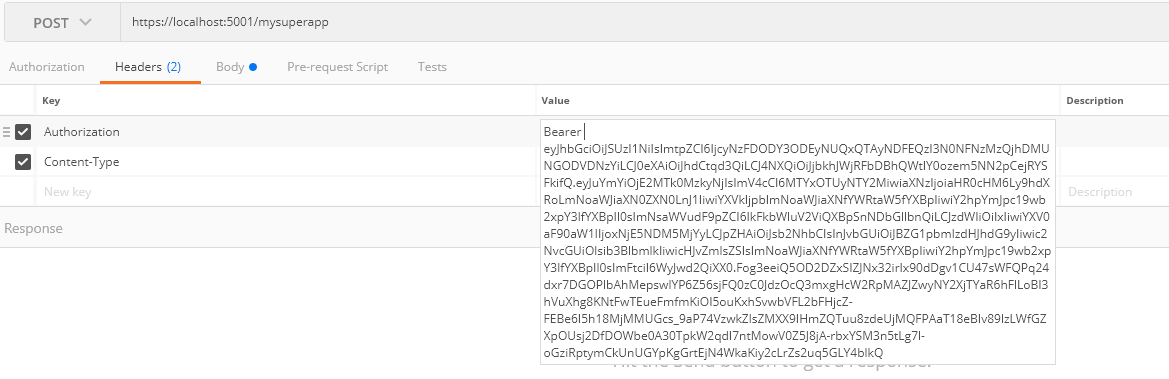
Это типичная базовая таблица для хранения пользовательских данных. Уникальный идентификатор пользователя является главным ключом для связки с другими данными. Логин — уникальное текстовое поле (чаще всего это либо номер телефона, либо почта пользователя). А пароль – обычное текстовое поле. Все шифрование происходит на стороне кода приложения, а не в базе.

Нельзя ни в коем случае делать разграничения прав через пользователей баз данных! В базу данных приложение должно логиниться только через одного пользователя с максимально урезанными правами и никак не через администратора (sa или root).

По мере развития вашего приложения данная таблица может обрасти дополнительными полями. Это нормально.

# Введение в токены

Так что же такое токены? Если говорить упрощенно, то это обычные строковые данные, которые сложно читаемы для человека. Токен помещаются в HEADER запроса по ключу Authorization.



Пример того, как выглядит токен с запросом POST.

Как правило, токенов бывает несколько видов: токены доступа и токены обновления. Именно взаимодействие в паре этих токенов приводит к тому, что безопасность вашего приложения будет выше, чем если бы вы использовали только один или самописный подход.

Токен доступа – это такой токен, с помощью которого можно получить доступ к защищенным ресурсам. Он короткоживущий и многоразовый.

Токен обновления – это такой токен, который выполняет единственную роль – получение нового токена доступа. Данный токен долгоживущий, но одноразовый. Благодаря ему пользователю не нужно постоянно перелогиниваться, чтобы обновить свой токен доступа.

Стоит отметить, что JWT расшифровывается как Jason Web Token. Токены используют преимущества цифровой подписи, что не даёт кому-либо подделать их без информации о секретном ключе. Но токен могут украсть. И именно благодаря токену обновления, воспользоваться им можно будет только в ограниченное время.

Давайте разберем полную схему работы с токенами:

1. Пользователь проходит аутентификацию в приложение с помощью пары логина и пароля.

2. Если аутентификация прошла успешна, приложение выдает пользователю пару токенов: доступ и обновление.

3. При работе с приложением пользователь постоянно передает свой токен доступа, который проверяется на валидность. В случае успеха приложение предоставляет запрашиваемый функционал.

4. В то время, как токен доступа становится невалидным, пользователь отправляет токен обновления, что приводит к генерации новых токенов доступа и обновления.

5. Если токен обновления истек, то пользователь должен заново пройти аутентификацию.

# Наш первый токен

Для того, чтобы создать наш первый токен, нужно подготовить консольное приложение на базе .net core 3.1 или выше. Какие nuget пакеты нам потребуются:

|  |  |
| --- | --- |
| **Название** | **Описание** |
| System.IdentityModel.Tokens.Jwt | Сборка, которая служит для создания, сериализации и проверки JWT токенов |

Создадим простенький класс под названием UserService:

|  |
| --- |
| internal sealed class UserService  {  private IDictionary<string, string> \_users = new Dictionary<string, string>()  {  {"test", "test"}  };    private const string SecretCode = "THIS IS SOME VERY SECRET STRING!!! Im blue da ba dee da ba di da ba dee da ba di da d ba dee da ba di da ba dee";    public string Authenticate(string user, string password)  {  if (string.IsNullOrWhiteSpace(user) || string.IsNullOrWhiteSpace(password))  {  return string.Empty;  }  int i = 0;  foreach (KeyValuePair<string,string> pair in \_users)  {  i++;  if (string.CompareOrdinal(pair.Key, user) == 0 && string.CompareOrdinal(pair.Value, password) == 0)  {  return GenerateJwtToken(i);  }  }  return string.Empty;  }    private string GenerateJwtToken(int id)  {  JwtSecurityTokenHandler tokenHandler = new JwtSecurityTokenHandler();    byte[] key = Encoding.ASCII.GetBytes(SecretCode);    SecurityTokenDescriptor tokenDescriptor = new SecurityTokenDescriptor  {  Subject = new ClaimsIdentity(new Claim[]  {  new Claim(ClaimTypes.Name, id.ToString())  }),  Expires = DateTime.UtcNow.AddMinutes(15),  SigningCredentials = new SigningCredentials(new SymmetricSecurityKey(key), SecurityAlgorithms.HmacSha256Signature)  };    SecurityToken token = tokenHandler.CreateToken(tokenDescriptor);    return tokenHandler.WriteToken(token);  }  } |

Это тестовый класс, который эмулирует некий сервис для проверки пользователя и выдачу первого токена. Пользователи хранятся в словаре и, пока он единственный, – test с аналогичным паролем. Логика достаточно проста, при совпадении логина и пароля происходит генерация токена, который действует всего 15 минут.

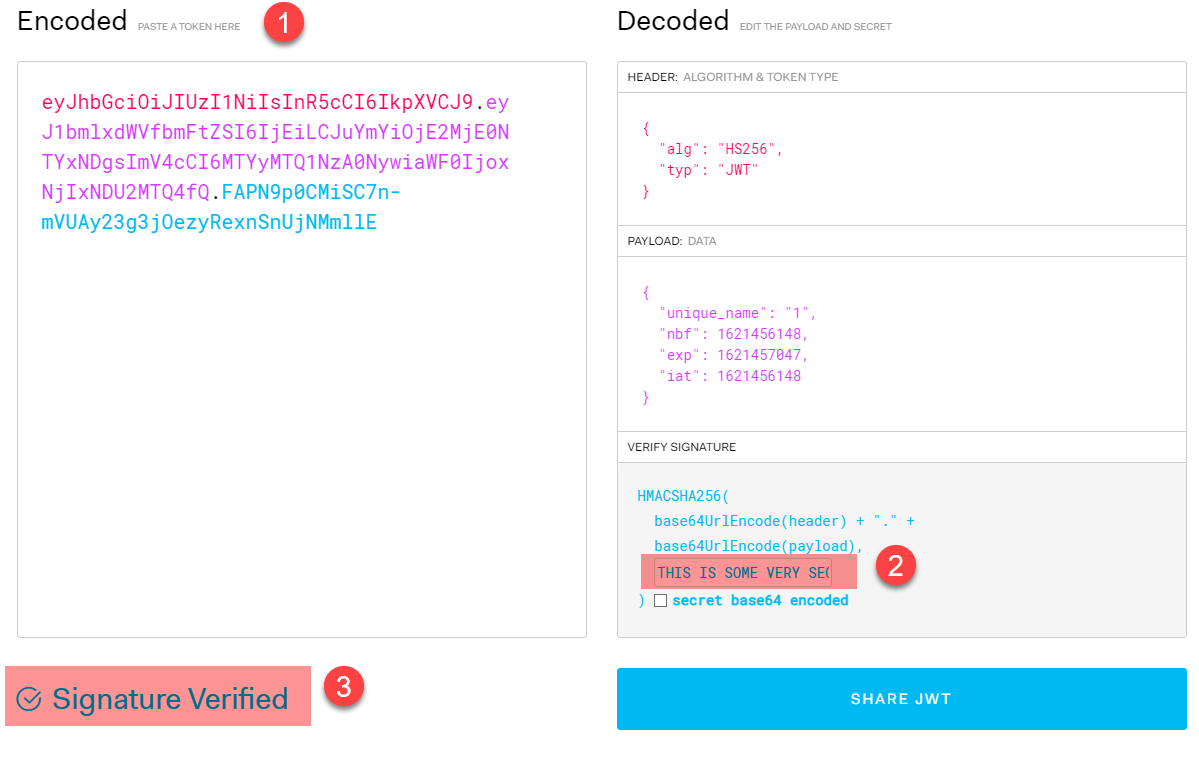
Дополним логику ввода пользователя и пароля в класс Program:

|  |
| --- |
| class Program  {  static void Main(string[] args)  {  Console.WriteLine("Enter user name: ");  string userName = Console.ReadLine();  Console.WriteLine("Enter user password: ");  string userPassword = Console.ReadLine();  UserService userService = new UserService();  string token = userService.Authenticate(userName, userPassword);  Console.WriteLine(token);  }  } |

При вводе правильных данных (test/test) на консоле произойдет вывод нового токена, к примеру:

|  |
| --- |
| eyJhbGciOiJIUzI1NiIsInR5cCI6IkpXVCJ9.eyJ1bmlxdWVfbmFtZSI6IjEiLCJuYmYiOjE2MjE0NTYxNDgsImV4cCI6MTYyMTQ1NzA0NywiaWF0IjoxNjIxNDU2MTQ4fQ.FAPN9p0CMiSC7n-mVUAy23g3jOezyRexnSnUjNMmllE |

Стоит обратить внимание, что создание токена происходит посредством подписи секретного кода, который указан в нашем классе UserService в константе. Если скопировать данный токен и перейти на официальный сайт JWT токена по адресу [https://jwt.io](https://jwt.io/), то его можно расшифровать.



В поле номер 1 нужно ввести наш токен, а в поле 2 нужно ввести наш секретный код. Если все правильно, то в поле 3 будет информация о том, что подпись проверена.

Это и есть вся процедура генерации токена. Она не сложна, но местами запутана.

# Работа в ASP.NET

В консольном приложении все было достаточно просто. Именно этот базовый функционал даёт хорошую основу для того, чтобы интегрировать его в дальнейшем в любое другое приложение. Давайте рассмотрим, как можно интегрировать его в приложение, основанное на REST технологии.

После создания нового приложения Web API нужно добавить nuget пакеты

|  |  |
| --- | --- |
| **Название** | **Описание** |
| System.IdentityModel.Tokens.Jwt | Сборка, которая служит для создания, сериализации и проверки JWT токенов |
| Microsoft.AspNetCore.Authentication.JwtBearer | Поддержку JWT токенов в ASP.NET Core приложениях |

После всех этих пунктов наступает этап настройки приложения для работы с токенами. Для начала модифицируем наш предыдущий сервис, сделаем секретный код публичной константой и добавим наследования от интерфейса:

|  |
| --- |
| public interface IUserService  {  TokenResponse Authenticate(string user, string password);  }  internal sealed class UserService : IUserService  {  // skipped  public const string SecretCode = "THIS IS SOME VERY SECRET STRING!!! Im blue da ba dee da ba di da ba dee da ba di da d ba dee da ba di da ba dee";  // skipped  } |

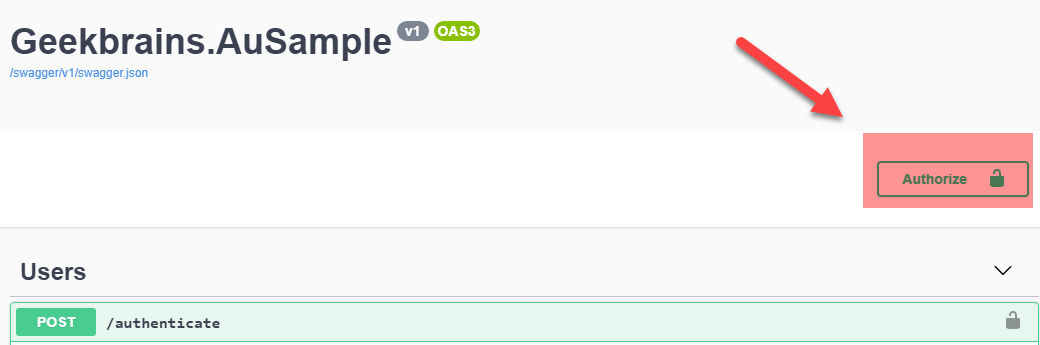
Обратите внимание, что большинство кода убрано, т. к. он идентичен предыдущему примеру.

Следующим шагом нужно сконфигурировать наши сервисы в классе Startup:

|  |
| --- |
| public void ConfigureServices(IServiceCollection services)  {  services.AddSingleton<IUserService, UserService>();  services.AddCors();  services.AddControllers();  services.AddAuthentication(x =>  {  x.DefaultAuthenticateScheme = JwtBearerDefaults.AuthenticationScheme;  x.DefaultChallengeScheme = JwtBearerDefaults.AuthenticationScheme;  })  .AddJwtBearer(x =>  {  x.RequireHttpsMetadata = false;  x.SaveToken = true;  x.TokenValidationParameters = new TokenValidationParameters  {  ValidateIssuerSigningKey = true,  IssuerSigningKey = new SymmetricSecurityKey(Encoding.ASCII.GetBytes(UserService.SecretCode)),  ValidateIssuer = false,  ValidateAudience = false,  ClockSkew = TimeSpan.Zero  };  });    services.AddSwaggerGen(c =>  {  c.SwaggerDoc("v1", new OpenApiInfo {Title = "Geekbrains.AuSample", Version = "v1"});  c.AddSecurityDefinition("Bearer", new OpenApiSecurityScheme  {  Description = "JWT Authorization header using the Bearer scheme (Example: 'Bearer 12345abcdef')",  Name = "Authorization",  In = ParameterLocation.Header,  Type = SecuritySchemeType.ApiKey,  Scheme = "Bearer"  });  c.AddSecurityRequirement(new OpenApiSecurityRequirement  {  {  new OpenApiSecurityScheme  {  Reference = new OpenApiReference  {  Type = ReferenceType.SecurityScheme,  Id = "Bearer"  }  },  Array.Empty<string>()  }  });  });  } |

Первым делом регистрируется наш сервис IUserService. Затем мы разрешаем делать межсайтовые запросы к нашему сервису посредством добавления поддержки CORS. И уже после добавления контроллеров происходит настройка аутентификации в сервисе на базе JWT токенов, где как раз и указывается та константа, которая является нашим секретным кодом.

Последним шагом идет настройка грамотного отображения Swagger страницы, чтобы поддержать отправку запросов вместе с токеном.



Большая часть работы сделана. Осталось только пару штрихов в методе Configure:

|  |
| --- |
| public void Configure(IApplicationBuilder app, IWebHostEnvironment env)  {  if (env.IsDevelopment())  {  app.UseDeveloperExceptionPage();  app.UseSwagger();  app.UseSwaggerUI(c => c.SwaggerEndpoint("/swagger/v1/swagger.json", "Geekbrains.AuSample v1"));  }  app.UseHttpsRedirection();  app.UseRouting();  app.UseCors(x => x  .SetIsOriginAllowed(origin => true)  .AllowAnyMethod()  .AllowAnyHeader()  .AllowCredentials());  app.UseAuthentication();  app.UseAuthorization();  app.UseEndpoints(endpoints => { endpoints.MapControllers(); });  } |

Здесь добавляется поддержка аутентификации и авторизации, а также происходит тонкая настройка политики CORS.

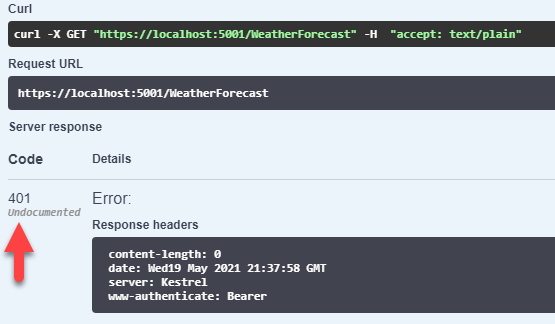
В новых приложениях часто формируется тестовые контроллеры. В нашем проекте тоже должен быть один такой под названием WeatherForecastController. Для проверки того, что все было сделано правильно, добавьте на него атрибут Authorize, как сделано ниже:

|  |
| --- |
| [ApiController]  [Authorize]  [Route("[controller]")]  public class WeatherForecastController : ControllerBase  {  private static readonly string[] Summaries = new[]  {  "Freezing", "Bracing", "Chilly", "Cool", "Mild", "Warm", "Balmy", "Hot", "Sweltering", "Scorching"  };  private readonly ILogger<WeatherForecastController> \_logger;  public WeatherForecastController(ILogger<WeatherForecastController> logger)  {  \_logger = logger;  }  [HttpGet]  public IEnumerable<WeatherForecast> Get()  {  var rng = new Random();  return Enumerable.Range(1, 5).Select(index => new WeatherForecast  {  Date = DateTime.Now.AddDays(index),  TemperatureC = rng.Next(-20, 55),  Summary = Summaries[rng.Next(Summaries.Length)]  })  .ToArray();  }  } |

После этого запустите приложение. Если все правильно, то вы увидите похожую страницу. Красным выделено, то, что должно быть обязательно.



Если вы попытаетесь выполнить вызов WeatherForecast, то увидите ошибку под номером 401, что означает нехватку прав для выполнения этого метода:



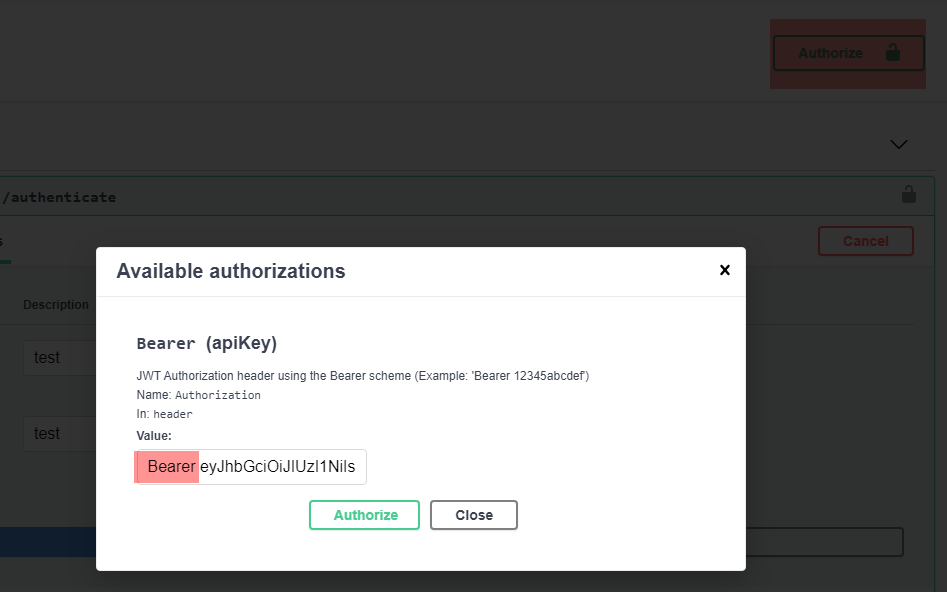
Осталось совсем чуть-чуть. Нужно добавить контроллер, который возьмёт на себя генерацию токенов.

Добавьте новый контроллер UserController, как показано ниже:

|  |
| --- |
| public sealed class UsersController : ControllerBase  {  private readonly IUserService \_userService;    public UsersController(IUserService userService)  {  \_userService = userService;  }  [AllowAnonymous]  [HttpPost("authenticate")]  public IActionResult Authenticate([FromQuery] string user, string password)  {  string token = \_userService.Authenticate(user, password);    if (string.IsNullOrWhiteSpace(token))  {  return BadRequest(new {message = "Username or password is incorrect"});  }  return Ok(token);  }  } |

Стоит обратить внимание, что на единственном методе Authenticate был добавлен атрибут AllowAnonymous, что означает доступ даже анонимным пользователям. Соответственно, такой атрибут удобен для всех тех методов, которые не подразумевают проверку пользователя и прав.

После запуска приложения у вас появится новый метод, который спокойно можно выполнить, отправив туда test/test для генерации токена. А после получения токена, нужно авторизоваться в Swagger, нажав на кнопку Authorize:



Внимательно посмотрите, как вводится токен: сначала идет слово Bearer, а потом уже тон сам.

|  |
| --- |
| Bearer eyJhbGciOiJIUzI1NiIsInR5cCI6IkpXVCJ9.eyJ1bmlxdWVfbmFtZSI6IjEiLCJuYmYiOjE2MjE0NjA2NjAsImV4cCI6MTYyMTQ2MTU2MCwiaWF0IjoxNjIxNDYwNjYwfQ.bx7zk6qd7-yUJXdZFqFjLn\_Nyc8\_vV4ePqxpkhtcm6M |

Если все сделано правильно, то последующий вызов контроллера WeatherForecast будет успешен.

# 

# 

# Поддержка RefreshToken

Token обновления по факту является тем же токеном, только с большим временем жизни. Если в предыдущем примере для токена доступа было задано время жизни 15 минут, то для токена обновления это значение может быть намного больше. В нашем примере установим 360 минут.

Добавим несколько моделей, которые будем использовать для хранения данных – токенов.

|  |
| --- |
| public sealed class RefreshToken  {  public string Token { get; set; }    public DateTime Expires { get; set; }    public bool IsExpired => DateTime.UtcNow >= Expires;  } |

Это модель, которая будет хранить последний из токенов обновления. Следующая модель будет хранить связку пароль к последнему токену обновления пользователя:

|  |
| --- |
| internal sealed class AuthResponse  {  public string Password { get; set; }    public RefreshToken LatestRefreshToken { get; set; }  } |

И последняя модель для удобной передачи из сервиса в контроллер двух токенов при аутентификации:

|  |
| --- |
| public sealed class TokenResponse  {  public string Token { get; set; }  public string RefreshToken { get; set; }  } |

Так как связка у нас стала более усложненной, то требуется полностью реработать весь UserService и его интерфейс:

|  |
| --- |
| public interface IUserService  {  TokenResponse Authenticate(string user, string password);    string RefreshToken(string token);  } |

Теперь при аутентификации мы возвращаем модель с двумя токенами и добавляем метод обновления токена. Сам сервис UserService потерпит большее изменение:

|  |
| --- |
| internal sealed class UserService : IUserService  {  private IDictionary<string, AuthResponse> \_users = new Dictionary<string, AuthResponse>()  {  {"test", new AuthResponse() { Password = "test"}}  };  public const string SecretCode = "THIS IS SOME VERY SECRET STRING!!! Im blue da ba dee da ba di da ba dee da ba di da d ba dee da ba di da ba dee";  public TokenResponse Authenticate(string user, string password)  {  if (string.IsNullOrWhiteSpace(user) || string.IsNullOrWhiteSpace(password))  {  return null;  }  TokenResponse tokenResponse = new TokenResponse();  int i = 0;  foreach (KeyValuePair<string,AuthResponse> pair in \_users)  {  i++;  if (string.CompareOrdinal(pair.Key, user) == 0 && string.CompareOrdinal(pair.Value.Password, password) == 0)  {  tokenResponse.Token = GenerateJwtToken(i, 15);  RefreshToken refreshToken = GenerateRefreshToken(i);  pair.Value.LatestRefreshToken = refreshToken;  tokenResponse.RefreshToken = refreshToken.Token;  return tokenResponse;  }  }  return null;  }  public string RefreshToken(string token)  {  int i = 0;  foreach (KeyValuePair<string,AuthResponse> pair in \_users)  {  i++;  if (string.CompareOrdinal(pair.Value.LatestRefreshToken.Token, token) == 0  && pair.Value.LatestRefreshToken.IsExpired is false)  {  pair.Value.LatestRefreshToken = GenerateRefreshToken(i);  return pair.Value.LatestRefreshToken.Token;  }  }  return string.Empty;  }  private string GenerateJwtToken(int id, int minutes)  {  JwtSecurityTokenHandler tokenHandler = new JwtSecurityTokenHandler();  byte[] key = Encoding.ASCII.GetBytes(SecretCode);    SecurityTokenDescriptor tokenDescriptor = new SecurityTokenDescriptor  {  Subject = new ClaimsIdentity(new Claim[]  {  new Claim(ClaimTypes.Name, id.ToString())  }),  Expires = DateTime.UtcNow.AddMinutes(minutes),  SigningCredentials = new SigningCredentials(new SymmetricSecurityKey(key), SecurityAlgorithms.HmacSha256Signature)  };  SecurityToken token = tokenHandler.CreateToken(tokenDescriptor);  return tokenHandler.WriteToken(token);  }  public RefreshToken GenerateRefreshToken(int id)  {  RefreshToken refreshToken = new RefreshToken();  refreshToken.Expires = DateTime.Now.AddMinutes(360);  refreshToken.Token = GenerateJwtToken(id, 360);  return refreshToken;  }  } |

Хранение пользователей теперь ведется не просто с паролем, но и последним токеном обновления. Сам токен обновления создаётся ровно по такому же принципу, но время его жизни увеличивается на 360 минут, чего вполне хватит, чтобы не сильно раздражать пользователя частым перелогином в приложение. Остался последний штрих – обновить контроллер и сохранить токен обновления в специальном хранилище:

|  |
| --- |
| public sealed class UsersController : ControllerBase  {  private readonly IUserService \_userService;  public UsersController(IUserService userService)  {  \_userService = userService;  }  [AllowAnonymous]  [HttpPost("authenticate")]  public IActionResult Authenticate([FromQuery] string user, string password)  {  TokenResponse token = \_userService.Authenticate(user, password);  if (token is null)  {  return BadRequest(new {message = "Username or password is incorrect"});  }  SetTokenCookie(token.RefreshToken);  return Ok(token);  }  [Authorize]  [HttpPost("refresh-token")]  public IActionResult Refresh()  {  string oldRefreshToken = Request.Cookies["refreshToken"];  string newRefreshToken = \_userService.RefreshToken(oldRefreshToken);    if (string.IsNullOrWhiteSpace(newRefreshToken))  {  return Unauthorized(new { message = "Invalid token" });  }  SetTokenCookie(newRefreshToken);  return Ok(newRefreshToken);  }  private void SetTokenCookie(string token)  {  var cookieOptions = new CookieOptions  {  HttpOnly = true,  Expires = DateTime.UtcNow.AddDays(7)  };  Response.Cookies.Append("refreshToken", token, cookieOptions);  }  } |

Токен обновления кладется в хранилище Cookies. Оно часто используется для хранения небольших данных и отправляется каждый раз при запросе.

# Практическое задание

В ваше приложение из предыдущего урока необходимо добавить таблицу пользователей вместе с колонкой для токена обновления. Нужно поддержать генерацию пары токенов, перевести все контроллеры на поддержку аутентификации и авторизации.

# Дополнительные материалы

1. Pro ASP.NET Core MVC 2, Adam Freeman
2. <https://docs.microsoft.com/ru-ru/aspnet/core/mvc/overview?view=aspnetcore-5.0#web-apis>
3. Ссылка на репозиторий проекта Timesheets: <https://github.com/evgshk/gb-ts-app>

# Используемые источники

1. [https://docs.microsoft.com/ru-ru/aspnet/core/security/cors?view=aspnetcore-5.0](https://docs.microsoft.com/ru-ru/aspnet/core/security/cors?view=aspnetcore-5.0#:~:text=CORS%20%E2%80%94%20%D1%8D%D1%82%D0%BE%20%D1%81%D1%82%D0%B0%D0%BD%D0%B4%D0%B0%D1%80%D1%82%20%D0%BA%D0%BE%D0%BD%D1%81%D0%BE%D1%80%D1%86%D0%B8%D1%83%D0%BC%D0%B0%20W3C,%D0%BE%D1%81%D0%BB%D0%B0%D0%B1%D0%B8%D1%82%D1%8C%20%D0%BF%D0%BE%D0%BB%D0%B8%D1%82%D0%B8%D0%BA%D1%83%20%D1%82%D0%BE%D0%B3%D0%BE%20%D0%B6%D0%B5%20%D0%B8%D1%81%D1%82%D0%BE%D1%87%D0%BD%D0%B8%D0%BA%D0%B0)
2. <https://jwt.io>
3. <https://devblogs.microsoft.com/aspnet/jwt-validation-and-authorization-in-asp-net-core/>