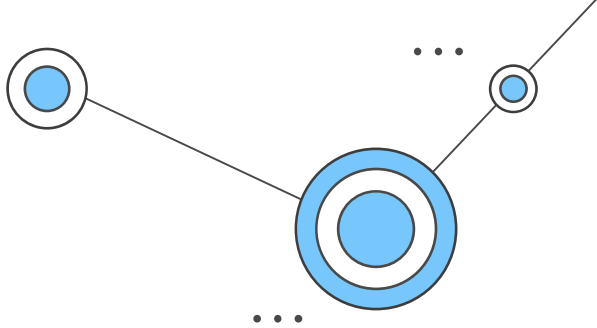
JWT Authentication Entity Framework

Manea Nicolas-Sebastian
Marin Alin-Andrei
Neagoe Filip

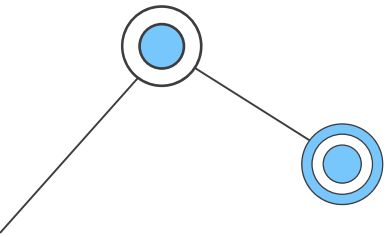


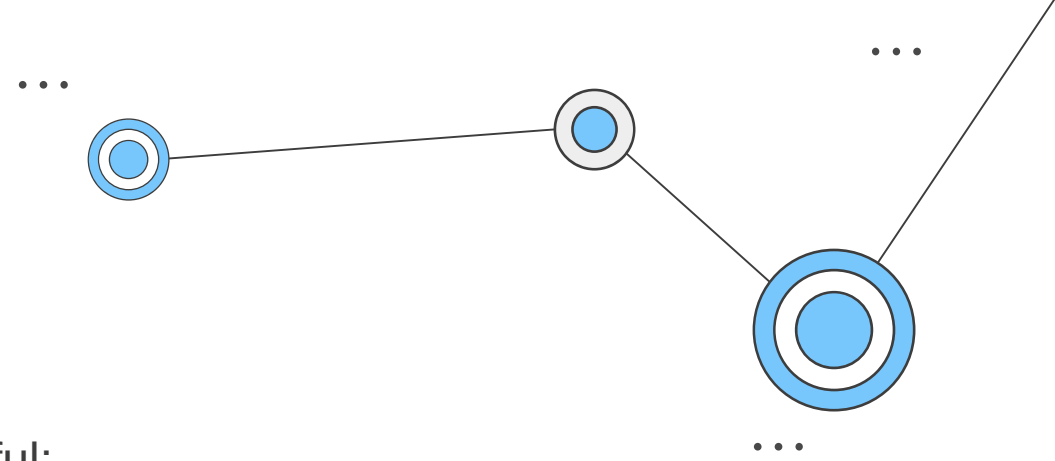
JWT

What is JSON Web Token?

JSON Web Token (JWT) is an open standard that defines a compact and self-contained way for securely transmitting information between parties as a JSON object. This information can be verified and trusted because it is digitally signed. JWTs can be signed using a secret (with the HMAC algorithm) or a public/private key pair using RSA or ECDSA.

Although JWTs can be encrypted to also provide secrecy between parties, we will focus on signed tokens. Signed tokens can verify the integrity of the claims contained within it, while encrypted tokens hide those claims from other parties.





JWT

When should you use JSON Web Tokens?

Here are some scenarios where JSON Web Tokens are useful:

Authorization: This is the most common scenario for using JWT. Once the user is logged in, each subsequent request will include the JWT, allowing the user to access routes, services, and resources that are permitted with that token.

Information Exchange: JSON Web Tokens are a good way of securely transmitting information between parties. Because JWTs can be signed - for example, using public/private key pairs - you can be sure the senders are who they say they are. Additionally, as the signature is calculated using the header and the payload, you can also verify that the content hasn't been tampered with.

JWT

What is the JSON Web Token structure?

In its compact form, JSON Web Tokens consist of three parts separated by dots (.), which are:

- ***Header***
- ***Payload***
- ***Signature***

Therefore, a JWT typically looks like the following:

xxxxx.yyyyy.zzzzz

JWT

Header

The header typically consists of two parts: the type of the token, which is JWT, and the signing algorithm being used, such as HMAC SHA256 or RSA.

For example:

```
{
  "alg": "HS256",
  "typ": "JWT"
}
```

JWT

Payload

The second part of the token is the payload, which contains the claims. Claims are statements about an entity (typically, the user) and additional data. There are three types of claims: registered, public, and private claims.

Registered claims: These are a set of predefined claims which are not mandatory but recommended, to provide a set of useful, interoperable claims. Some of them are: *iss* (issuer), *exp* (expiration time), *sub* (subject), *aud* (audience), and others.

An example payload could be:

```
{
  "sub": "1234567890",
  "name": "John Doe",
  "admin": true
}
```

JWT

Signature

To create the signature part you have to take the encoded header, the encoded payload, a secret, the algorithm specified in the header, and sign that.

For example if you want to use the HMAC SHA256 algorithm, the signature will be created in the following way:

```
HMACSHA256(
  base64UrlEncode(header) + "." +
  base64UrlEncode(payload),
  secret)
```

JWT

How do JSON Web Tokens work?

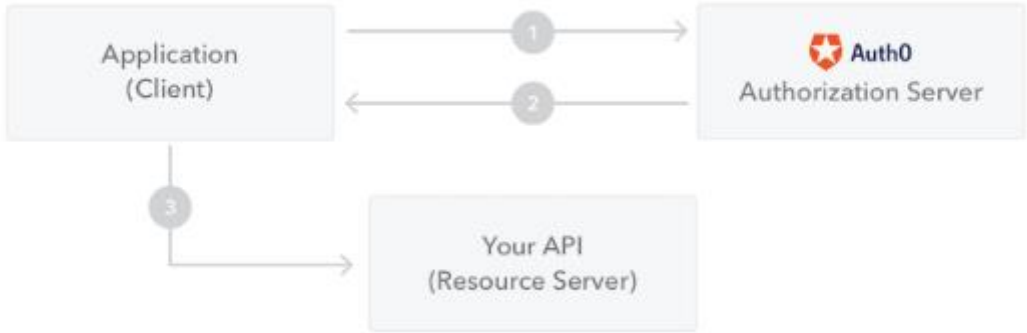
In authentication, when the user successfully logs in using their credentials, a JSON Web Token will be returned. Since tokens are credentials, great care must be taken to prevent security issues. In general, you should not keep tokens longer than required.

Whenever the user wants to access a protected route or resource, the user agent should send the JWT, typically in the Authorization header using the Bearer schema. The content of the header should look like the following:

```
Authorization: Bearer <token>
```

JWT

The following diagram shows how a JWT is obtained and used to access APIs or resources:

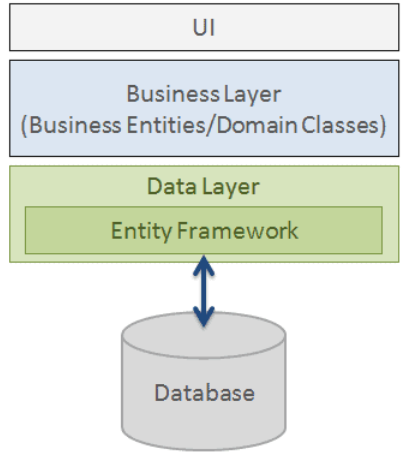


1. The application or client requests authorization to the authorization server. This is performed through one of the different authorization flows. For example, a typical OpenID Connect compliant web application will go through the */oauth/authorize* endpoint using the authorization code flow.
2. When the authorization is granted, the authorization server returns an access token to the application.
3. The application uses the access token to access a protected resource (like an API).

Entity Framework

Entity Framework is an open-source ORM framework for .NET applications supported by Microsoft. It enables developers to work with data using objects of domain specific classes without focusing on the underlying database tables and columns where this data is stored. With the Entity Framework, developers can work at a higher level of abstraction when they deal with data, and can create and maintain data-oriented applications with less code compared with traditional applications.

Entity Framework fits between the business entities (domain classes) and the database. It saves data stored in the properties of business entities and also retrieves data from the database and converts it to business entities objects automatically.

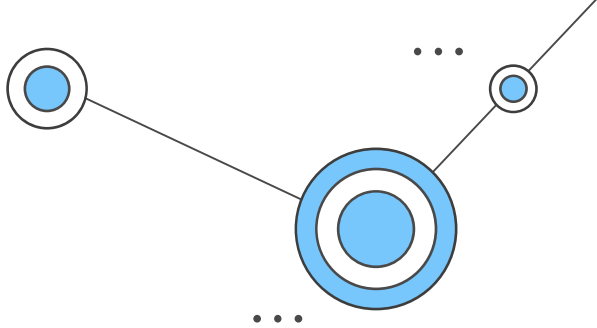


Postman

Postman is an API platform for building and using APIs. Postman simplifies each step of the API lifecycle and streamlines collaboration so you can create better APIs faster.

The Postman API client is the foundational tool of Postman, and it enables you to easily explore, debug, and test your APIs while also enabling you to define complex API requests for HTTP, REST, SOAP, GraphQL, and WebSockets.

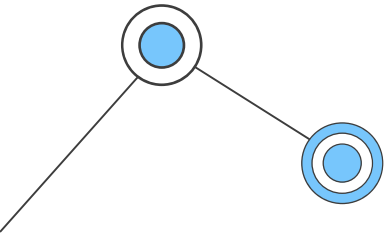
The API client automatically detects the language of the response, links, and format text inside the body to make inspection easy. The client also includes built-in support for authentication protocols like OAuth 1.2/2.0, AWS Signature, Hawk, and many more.



Swagger UI

Swagger UI allows anyone to visualize and interact with the API’s resources without having any of the implementation logic in place. It’s automatically generated from your OpenAPI (formerly known as Swagger) Specification, with the visual documentation making it easy for back end implementation and client side consumption.

Swagger UI generates an interactive API console for users to quickly learn about your API and experiment with requests.



JWT API

The first part of our project is to create an Api for authentication using JWT.

For this we used .NET 6.0 as technology.
For the user interface we used Swagger UI.

```
app.UseSwagger();
app.UseSwaggerUI(c =>
{
    c.SwaggerEndpoint("/swagger/v1/swagger.json", "Jwt Api v1");
});
```

JWT API

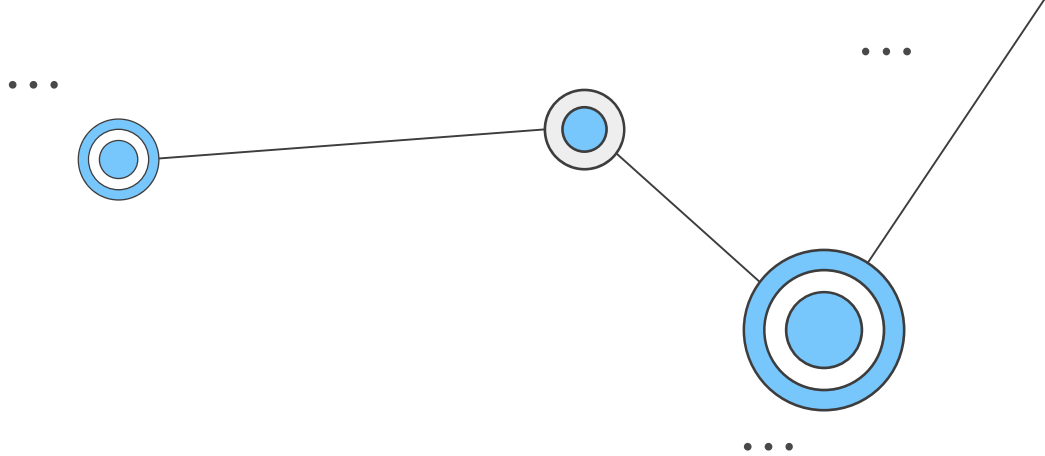
Launch Settings

When launching the application, Swagger will be used

The URL of the application will be the address https://localhost:7175

```
{
  "iisSettings": {
    "windowsAuthentication": false,
    "anonymousAuthentication": true,
    "iisExpress": {
      "applicationUrl": "http://localhost:38765",
      "sslPort": 44357
    }
  },
  "profiles": {
    "JwtApi": {
      "commandName": "Project",
      "dotnetRunMessages": true,
      "launchBrowser": true,
      "launchUrl": "swagger",
      "applicationUrl": "https://localhost:7175;http://localhost:5175",
      "environmentVariables": {
        "ASPNETCORE_ENVIRONMENT": "Development"
      }
    },
    "IIS Express": {
      "commandName": "IISExpress",
      "launchBrowser": true,
      "launchUrl": "swagger",
      "environmentVariables": {
        "ASPNETCORE_ENVIRONMENT": "Development"
      }
    }
  }
}
```

JWT API



App Settings

Database connection

The secret key. We must take into account that it must have at least 16 characters, because the HS256 algorithm requires a key greater than 128 bits.

Claims

```
{
  "Logging": {
    "LogLevel": {
      "Default": "Information",
      "Microsoft.AspNetCore": "Warning"
    }
  },
  "AllowedHosts": "*",
  "ConnectionStrings": {
    "DefaultConnection": "Server=(localdb)\\mssqllocaldb;Database=JwtApiDB;Trusted_Connection=True;MultipleActiveResultSets=true"
  },
  "Jwt": {
    "Key": "This is my jwt authentication key",
    "Issuer": "JwtApiAuthenticationServer",
    "Audience": "JwtApiServicePostmanClient",
    "Subject": "JwtApiServiceAccessToken"
  }
}
```

JWT API

Token Controller

In the Post method it is the place where the token is created.

First time the claims are created: the subject (Sub), JWT Id (Jti), the time the JWT was issued at (Iat), then the user credentials.

Then the key is created, followed by the signature, using a HmacSha256 algorithm, and then the token is generated.

```
namespace JwtApi.Controllers
{
    [Route("api/[controller]")]
    [ApiController]
    1 reference
    public class TokenController : ControllerBase
    {
        public IConfiguration _configuration;
        public readonly ApplicationDbContext _context;
        0 references
        public TokenController(IConfiguration configuration, ApplicationDbContext context)
        {
            _configuration = configuration;
            _context = context;
        }

        [HttpPost]
        0 references
        public async Task<IActionResult> Post(UserInfo userInfo)
        {
            if (userInfo != null && userInfo.UserName != null && userInfo.Email != null && userInfo.Password != null)
            {
                var user = await GetUser(userInfo.UserName, userInfo.Email, userInfo.Password);
                if (user != null)
                {
                    var claims = new[]
                    {
                        new Claim(JwtRegisteredClaimNames.Sub, _configuration["Jwt:Subject"]),
                        new Claim(JwtRegisteredClaimNames.Jti, Guid.NewGuid().ToString()),
                        new Claim(JwtRegisteredClaimNames.Iat, DateTime.UtcNow.ToString()),
                        new Claim("Id", user.UserId.ToString()),
                        new Claim("UserName", user.UserName),
                        new Claim("Email", user.Email),
                        new Claim("Password", user.Password)
                    };

                    var key = new SymmetricSecurityKey(Encoding.UTF8.GetBytes(_configuration["Jwt:Key"]));
                    var signIn = new SigningCredentials(key, SecurityAlgorithms.HmacSha256);
                    var token = new JwtSecurityToken(
                        _configuration["Jwt:Issuer"],
                        _configuration["Jwt:Audience"],
                        claims,
                        expires: DateTime.Now.AddMinutes(20),
                        signingCredentials: signIn);

                    return Ok(new JwtSecurityTokenHandler().WriteToken(token));
                }
                else
                {
                    return BadRequest("Invalid Credentials");
                }
            }
            else
            {
                return BadRequest();
            }
        }

        [HttpGet]
        1 reference
        public async Task<UserInfo> GetUser(string userName, string email, string password)
        {
            return await _context.UserInfo.FirstOrDefaultAsync(u => u.UserName == userName && u.Email == email && u.Password == password);

        }
    }
}
```

JWT API

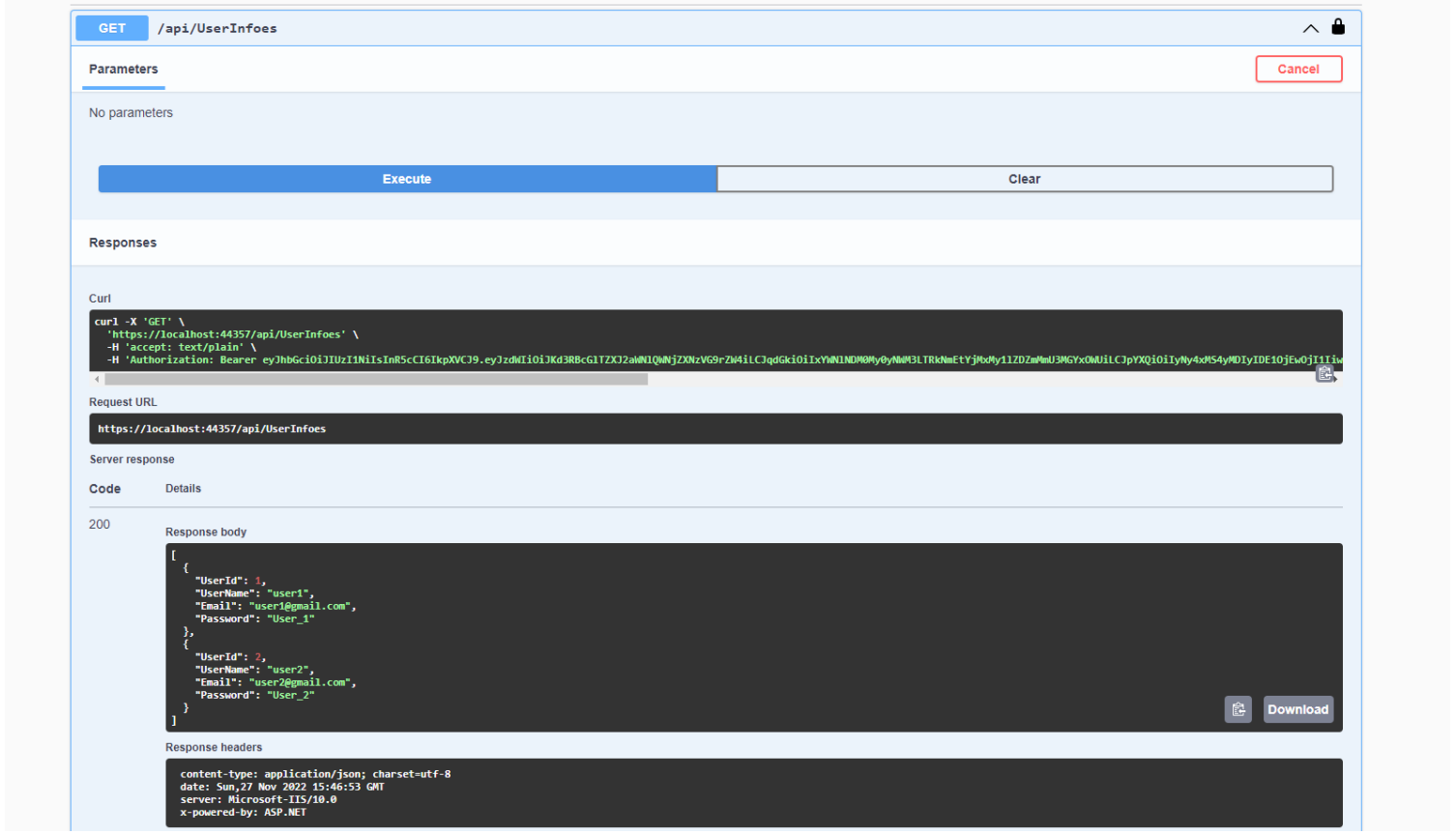
Add Authentication – using the Bearer schema

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

JWT API

Add a new User using Swagger UI

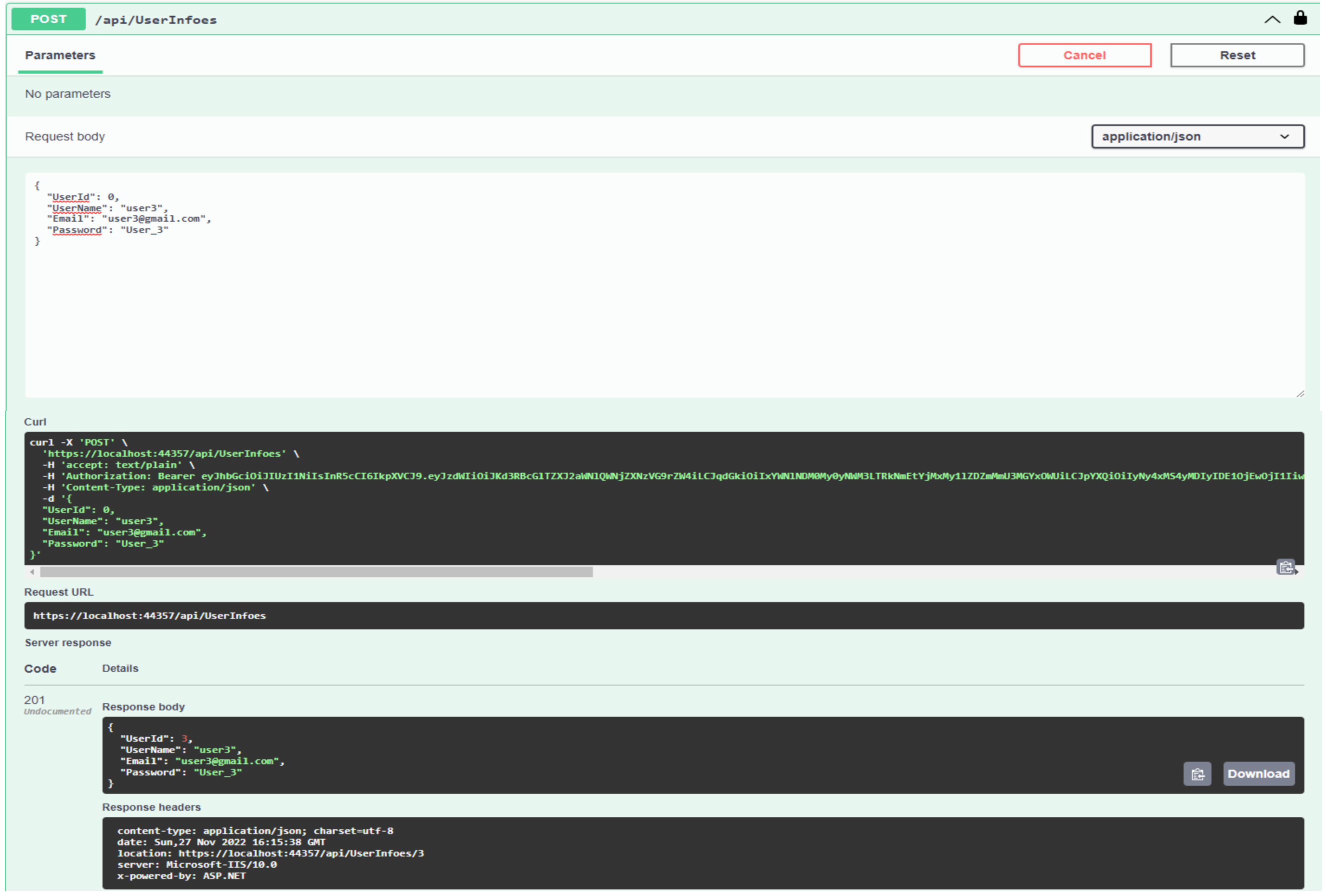
1. GET existing users



JWT API

Add a new User using Swagger UI

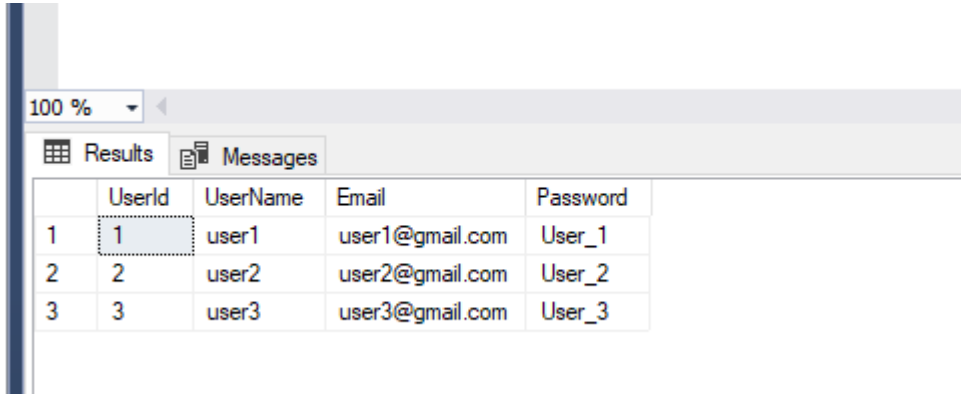
2. POST new user



JWT Api

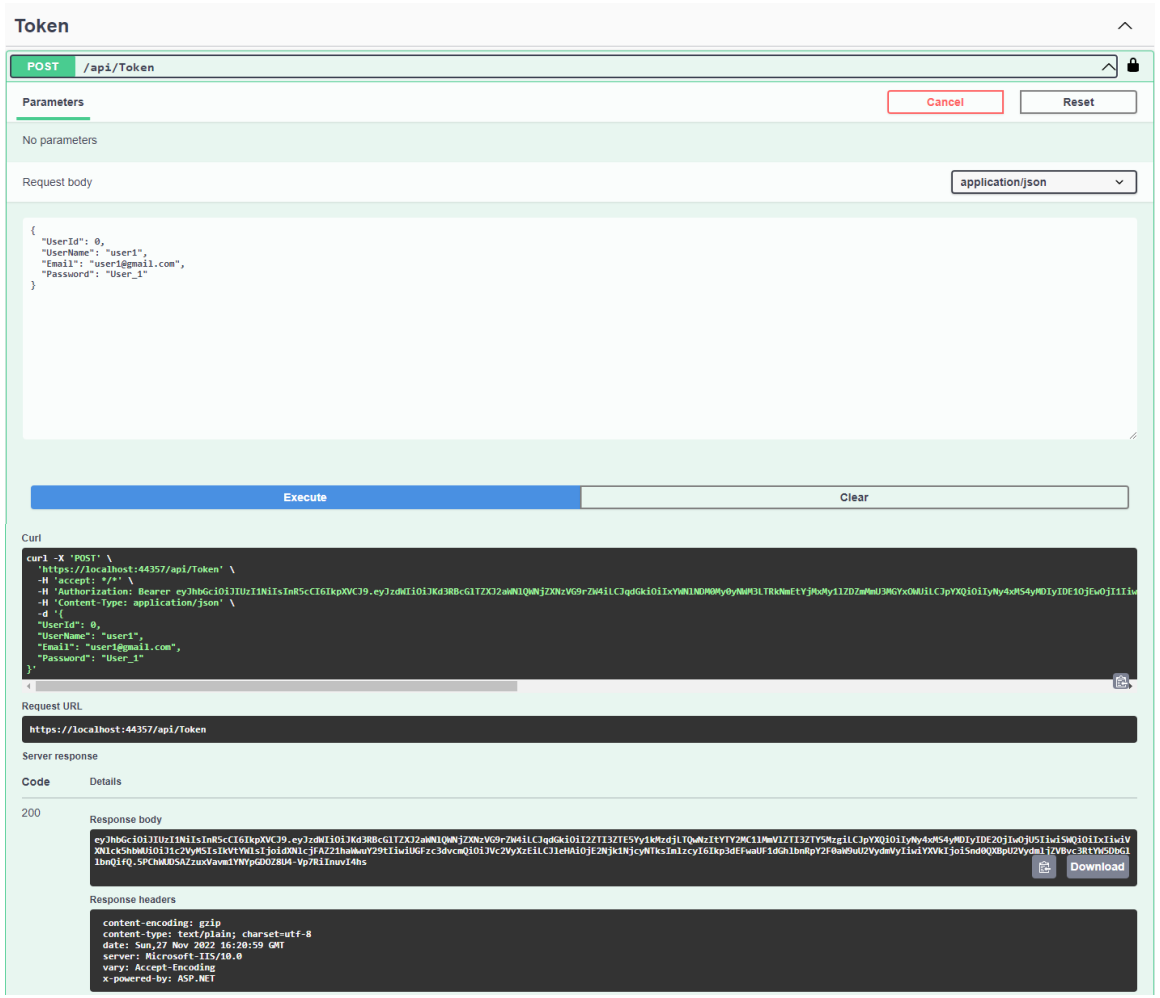
Add a new User using Swagger UI

3. Check database



JWT API

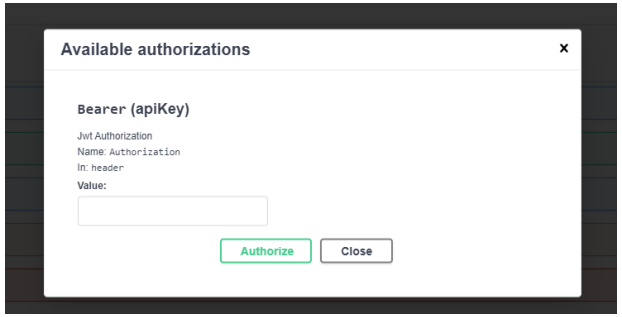
Generate a token using Swagger UI

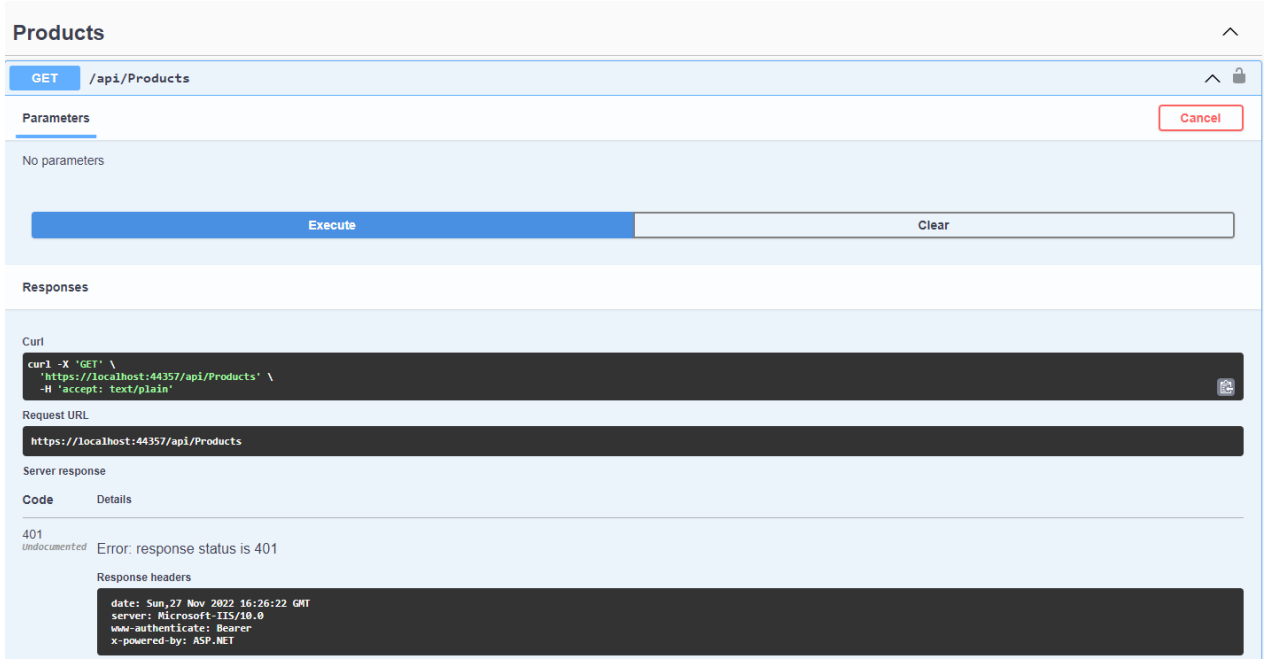


JWT API

Get products list – without authentication

To do any of the CRUD operations on the products, a user must be authenticated. When we try to view the list with an unauthenticated user, we will receive a response with the status 401 - Unauthorized Access.

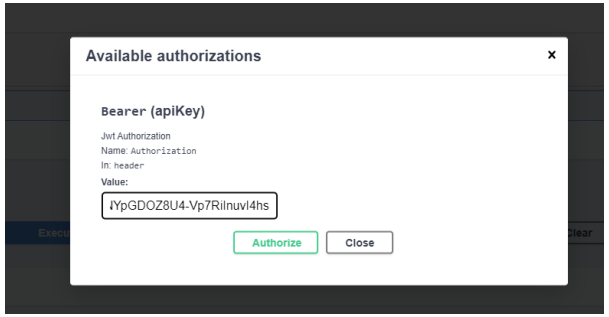


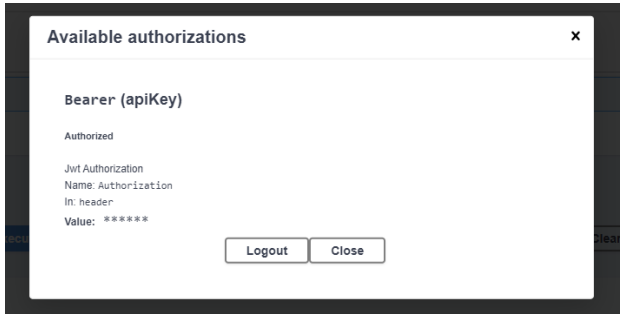


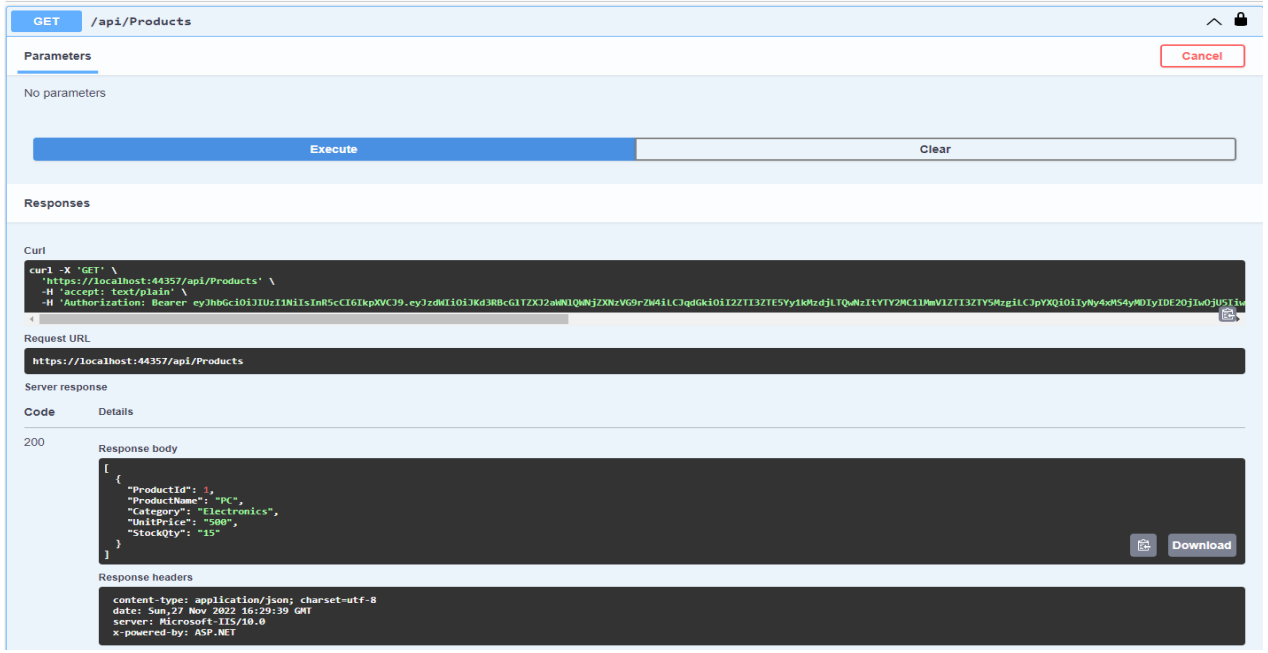
JWT API

Get products list – with authentication

If instead we authenticate using the generated token, we will be able to access the list of products. Authentication is done in Swagger UI by entering the token preceded by "Bearer" in the value field.

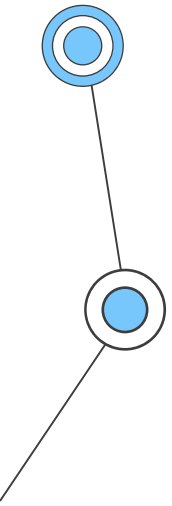






JWT Client Application

The second part of the project is a test application, which involves the authentication of a user and the performance of CRUD operations on the products in the database.



JWT Client Application

Home Controller

The Login and Logout methods are defined here.
When you log in, the token is generated, and when you log out, the token is deleted.

```
0 references
public async Task<ActionResult> LoginUser(UserInfo user)
{
    using (var httpClient = new HttpClient())
    {
        StringContent content = new StringContent(JsonConvert.SerializeObject(user), Encoding.UTF8, "application/json");
        using (var response = await httpClient.PostAsync("https://localhost:7175/api/token", content))
        {
            string token = await response.Content.ReadAsStringAsync();
            if(token == "Invalid credentials")
            {
                ViewBag.Message = "Incorrect credentials!";
                return Redirect("~/Home/Index");
            }
            HttpContext.Session.SetString("JWToken", token);
        }
        return Redirect("~/Dashboard/Index");
    }
}

0 references
public IActionResult LogOff()
{
    HttpContext.Session.Clear(); //clear token
    return Redirect("~/Home/Index");
}
```

JWT Client Application

Products Controller

We need token to access the protected api. All methods (get, create, edit, delete) will need this token to be accessed.

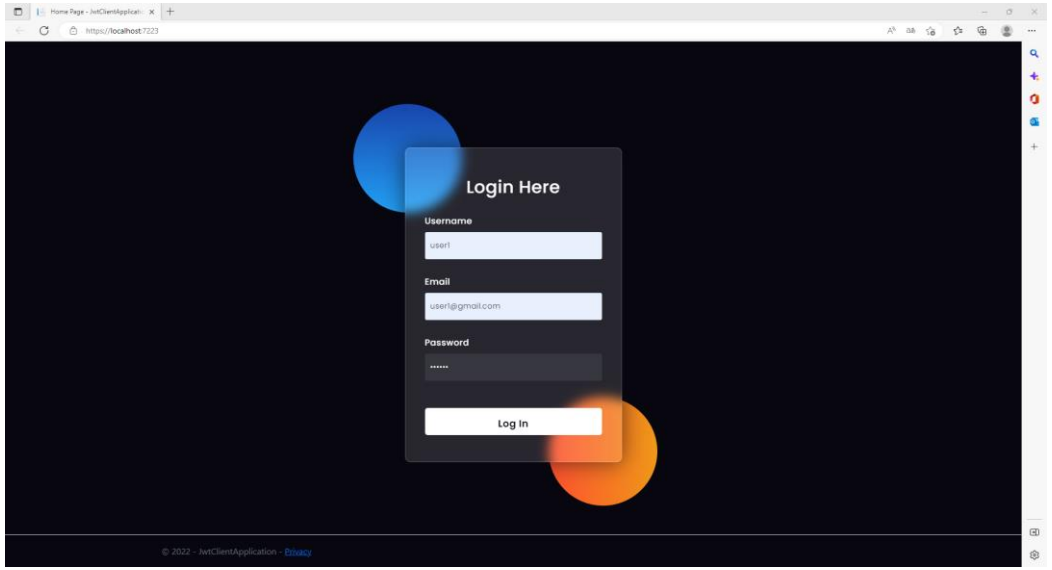
```
[HttpGet]
1 reference
public async Task<List<Products>> GetProducts()
{
    //use the access token to call a protected web api
    var accessToken = HttpContext.Session.GetString("JWToken");
    var url = baseUrl;
    HttpClient client = new HttpClient();
    client.DefaultRequestHeaders.Authorization = new AuthenticationHeaderValue("Bearer", accessToken);
    string jsonString = await client.GetStringAsync(url);

    var res = JsonConvert.DeserializeObject<List<Products>>(jsonString).ToList();

    return res;
}
```

JWT Client Application

User Login




```
0 references
public async Task<ActionResult> LoginUser(UserInfo user)
{
    using (var httpClient = new HttpClient())
    {
        StringContent content = new StringContent(JsonConvert.SerializeObject(user), Encoding.U
        using (var response = await httpClient.PostAsync("https://localhost:7175/api/token", co
        {
            string token = await response.Content.ReadAsStringAsync();
            if(token == "Invalid credentials") ≤ 3ms elapsed
            {
                ViewBag.Message = "Incorrect credentials!";
                return Redirect("~/Home/Index");
            }
            HttpContext.Session.SetString("JWToken", token);
        }
        return Redirect("~/Dashboard/Index");
    }
}
```

Autos

Search (Ctrl+E) Search Depth: 3

Name	Value	Type
response.Content	{System.Net.Http.HttpConnectionResponseContent}	System.Net.Http.Http...
this	{JwtClientApplication.Controllers.HomeController}	JwtClientApplication....
token	"eyJhbGciOiJIUzI1NiIsInR5cCI6IkpXVCJ9.eyJzdWIiOiJKd3RBc... View	string

JWT Client Application

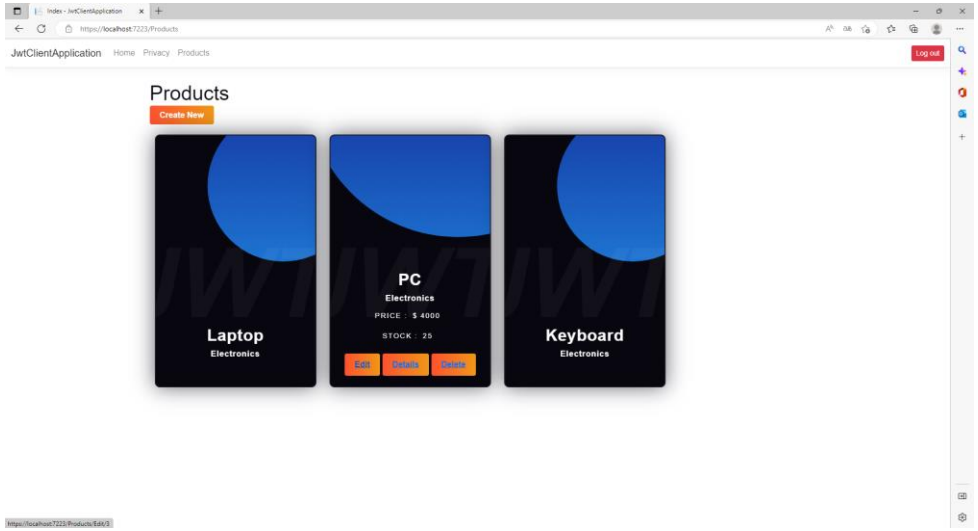
User Logout

Log out

```
0 references
public IActionResult LogOff()
{
    HttpContext.Session.Clear(); //clear token  ≤ 95.679ms elapsed
    return Redirect("~/Home/Index");
}
```

JWT Client Application

CRUD Operations for protected API




```
54
0 references
55  public async Task<IActionResult> Edit(int? id)
56  {
57      if (id == null)
58      {
59          return NotFound();
60      }
61
62      var accessToken = HttpContext.Session.GetString("JWToken");
63      var url = baseUrl + "/" + id;   ≤ 1ms elapsed
64      HttpClient client = new HttpClient();
65      client.DefaultRequestHeaders.Authorization = new AuthenticationHeaderValue("Bearer", accessToken);
66      string jsonString = await client.GetStringAsync(url);
67
68      var res = JsonConvert.DeserializeObject<Products>(jsonString);
69
70      if (res == null)
71      {
72          return NotFound();
73      }
74
75      return View(res);
76  }
0 references
```