CLASE 7

Seguridad

Introducción

Toda aplicación de cierta complejidad tiene que tener en consideración temas de seguridad ya sea autenticación/autorización, encriptación, CORS y HTTPS.

En este módulo veremos los aspectos de seguridad de un web API. Comenzaremos implementando **autenticación y autorización** para que sólo usuarios logueados en el sistema puedan consumir endpoints. Luego veremos la importancia de **CORS** para permitir que clientes de orígenes externos ajenos al contexto de seguridad del web API puedan consumir recursos.

También se puede trabajar con **encriptación** usando el servicio Data Protector Provider, y **claves de protección de datos** con la importancia de considerarlas en caso que en la resolución utilicemos encriptación, y aparte **funciones hash** para implementar un servicio hash en la aplicación (estos tres temas: encriptación, claves de protección de datos y funciones hash quedan fuera del alcance de este curso).

Y finalmente analizaremos cuándo es necesario usar **HTTPS**.

Autenticación y Autorización

Hasta ahora todos los envíos al web API han estado disponibles para cualquiera que quiera acceder. Esto no es bueno dado que habrá endpoints a los que queremos que solamente accedan determinados usuarios. Para eso usamos autenticación y autorización.

La **autenticación** se trata que un usuario muestre credenciales para verificar su identidad. Típicamente esas credenciales son un nombre de usuario y una contraseña.

La **autorización** se refiere a lo que un usuario tiene permitido hacer en la aplicación.

Existen distintos esquemas de autenticación HTTP. Estos no son exclusivos de ASP.Net Core sino que son estándares de cómo se debe manejar el flujo de autenticación de un usuario.

Podemos destacar algunos esquemas de autenticación:

- **Anónimo**: que quiere decir que cualquiera pueda acceder a un endpoint. Este es el esquema por omisión que hemos usado hasta ahora.
- **Basic**: a la hora de enviar un nombre de usuario y la contraseña al Web API, la contraseña se va a transformar en **Base64**. Esto implica que cualquiera con

acceso a estos datos puede obtener el nombre del usuario y la contraseña. Si este esquema va a ser utilizado es imprescindible también usar **SSL** (Secure Socket Layer).

- **Bearer**: este esquema está basado en tokens. Cuando el usuario se auténtica, el servidor le retorno un token como string el cual el usuario puede utilizar en subsiguientes peticiones HTTP al servidor. Este es uno de los esquemas de autenticación más populares.

Nota: Algunos links de intereses sobre esquemas de autenticación:

https://developer.mozilla.org/es/docs/Web/API/WindowBase64/Base64_codificando_y_decodificando

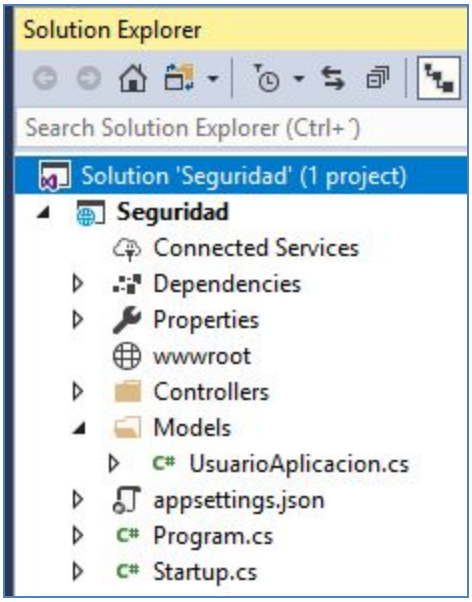
https://www.hostinger.com.ar/tutoriales/ssl-tls-https/

https://desarrolloweb.com/articulos/autenticacion-token.html

Veremos ahora cómo implementar autenticación en el web API, específicamente el esquema **Bearer**. Lo primero que vamos a necesitar es un conjunto de tablas con las cuales manejaremos el sistema de usuarios. ASP.Net Core ya ofrece una forma estandarizada de trabajar con esto, no es obligatorio hacerlo de esta manera pero ya que estamos utilizando un framework es razonable que aprovechemos sus capacidades.

Comenzaremos habilitando el sistema de usuarios. El primer paso que no es obligatorio pero es bastante conveniente es crear una clase que representa un usuario. A través de esta clase podremos guardar información extra de cualquier usuario y no solo las que el framework indique.

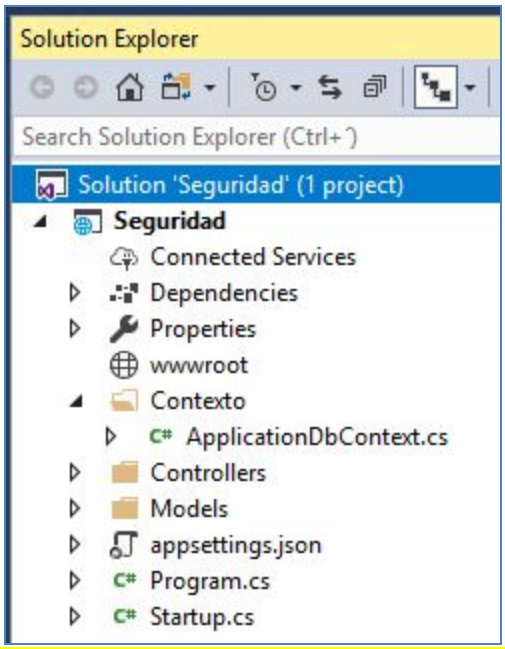
Creación de tablas base para un Sistema de Login



Vamos a trabajar con un nuevo proyecto WebApi y en él creamos una carpeta llamada **Models** y en dicha carpeta crear la clase llamada **UsuarioAplicacion** que hereda de la clase base a **IdentityUser** con el namespace correspondiente:

```
using Microsoft.AspNetCore.Identity;
using System;
using System.Collections.Generic;
using System.Linq;
using System.Threading.Tasks;

namespace Seguridad.Models
{
```

```
    public class UsuarioAplicacion: IdentityUser
    {
    }
}
```



Y también creamos el contexto de la aplicación, para ello creamos una carpeta **Contexto** con una clase llamada **ApplicationDbContext** y en vez de heredar de **DbContext** como hemos hecho hasta ahora para trabajar con Entity Framework, heredará de **IdentityDbContext** y le pasamos el **UsuarioAplicacion** que recién creamos con los namespace correspondientes:

```
using Microsoft.AspNetCore.Identity.EntityFrameworkCore;
using Seguridad.Models;
using System;
using System.Collections.Generic;
using System.Linq;
using System.Threading.Tasks;

namespace Seguridad.Contexto
{
    public class ApplicationDbContext:
IdentityDbContext<UsuarioAplicacion>
    {
    }
}
```

El **IdentityDbContext** es un contexto de datos especial que ya viene preconfigurado para trabajar con las tablas básicas y esenciales de un sistema de login. Se lo usa como si fuera un **DbContext** normal.

Y le agregamos el constructor y le pasamos un **DbContextOptions** con el nombre de esta clase del contexto **ApplicationDbContext** y le pasamos esto a la clase base.

```
using Microsoft.AspNetCore.Identity.EntityFrameworkCore;
using Microsoft.EntityFrameworkCore;
using Seguridad.Models;
```

```
using System;
using System.Collections.Generic;
using System.Linq;
using System.Threading.Tasks;

namespace Seguridad.Contexto
{
    public class ApplicationDbContext:
IdentityDbContext<UsuarioAplicacion>
    {
        public
ApplicationDbContext(DbContextOptions<ApplicationDbContext> options)
            : base(options)
        {

        }
    }
}
```

En esta clase también podríamos ubicar los **DbSet** correspondientes de las tablas de negocio pero en este caso solamente estamos haciendo un ejemplo de un sistema de login.

En la clase **startup** configuramos lo siguiente antes del servicio de MVC:

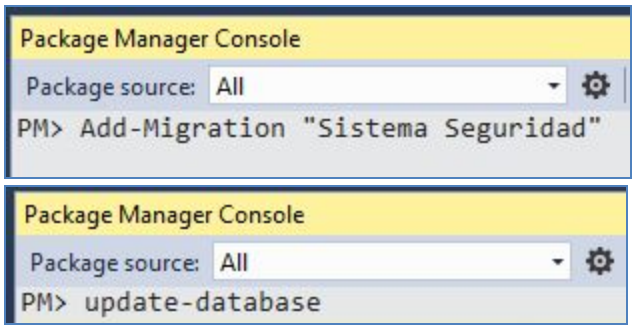
```
        public void ConfigureServices(IServiceCollection services)
        {
            services.AddDbContext<ApplicationDbContext>(options =>

options.UseSqlServer(Configuration.GetConnectionString("defaultConnection
")));

            services.AddIdentity<UsuarioAplicacion, IdentityRole>()
                .AddEntityFrameworkStores<ApplicationDbContext>()
                .AddDefaultTokenProviders();


services.AddMvc().SetCompatibilityVersion(CompatibilityVersion.Version_2_
1);
        }
```

Para configurar el sistema de autenticación de usuarios usamos el método **AddIdentity** y le pasamos la clase **UsuarioAplicacion** que representa los datos del usuario y la clase **ApplicationDbContext** que es el contexto de datos, e indicamos el uso del proveedor **Bearer** basado en tokens con el método **AddDefaultTokenProviders** (No olvide agregar los Namespaces necesarios). Esta es la configuración por omisión de **Identity Core** que cubre las necesidades de muchas aplicaciones que construiremos.

Y también configuramos el **DbContext**, le indicamos que usaremos Sql Server y le pasamos una cadena de conexión, que ubicamos en el archivo **appsettings.json**:

```
{
  "connectionStrings": {
    "defaultConnection": "Data Source=(localdb)\\mssqllocaldb;Initial
Catalog=Seguridad;Integrated Security=True"
  },
  "Logging": {
    "LogLevel": {
      "Default": "Warning"
    }
  },
  "AllowedHosts": "*"
}
```

Agregamos una migración de EF y actualizamos la base de datos:



Previo a ver la base de datos, veamos el código de la migración donde se crea un conjunto de tablas e índices, roles, usuarios, claims de roles, etc.:

```
using System;
using Microsoft.EntityFrameworkCore.Metadata;
using Microsoft.EntityFrameworkCore.Migrations;

namespace Seguridad.Migrations
{
    public partial class SistemaSeguridad : Migration
    {
        protected override void Up(MigrationBuilder migrationBuilder)
        {
            migrationBuilder.CreateTable(
                name: "AspNetRoles",
                columns: table => new
                {
                    Id = table.Column<string>(nullable: false),
```

```
                    Name = table.Column<string>(maxLength: 256, nullable:
true),
                    NormalizedName = table.Column<string>(maxLength: 256,
nullable: true),
                    ConcurrencyStamp = table.Column<string>(nullable:
true)
                },
                constraints: table =>
                {
                    table.PrimaryKey("PK_AspNetRoles", x => x.Id);
                });

            migrationBuilder.CreateTable(
                name: "AspNetUsers",
                columns: table => new
                {
                    Id = table.Column<string>(nullable: false),
                    UserName = table.Column<string>(maxLength: 256,
nullable: true),
                    NormalizedUserName = table.Column<string>(maxLength:
256, nullable: true),
                    Email = table.Column<string>(maxLength: 256,
nullable: true),
                    NormalizedEmail = table.Column<string>(maxLength:
256, nullable: true),
                    EmailConfirmed = table.Column<bool>(nullable: false),
                    PasswordHash = table.Column<string>(nullable: true),
                    SecurityStamp = table.Column<string>(nullable: true),
                    ConcurrencyStamp = table.Column<string>(nullable:
true),
                    PhoneNumber = table.Column<string>(nullable: true),
                    PhoneNumberConfirmed = table.Column<bool>(nullable:
false),
                    TwoFactorEnabled = table.Column<bool>(nullable:
false),
                    LockoutEnd = table.Column<DateTimeOffset>(nullable:
true),
                    LockoutEnabled = table.Column<bool>(nullable: false),
                    AccessFailedCount = table.Column<int>(nullable:
false)
                },
                constraints: table =>
                {
                    table.PrimaryKey("PK_AspNetUsers", x => x.Id);
                });

            migrationBuilder.CreateTable(
                name: "AspNetRoleClaims",
```

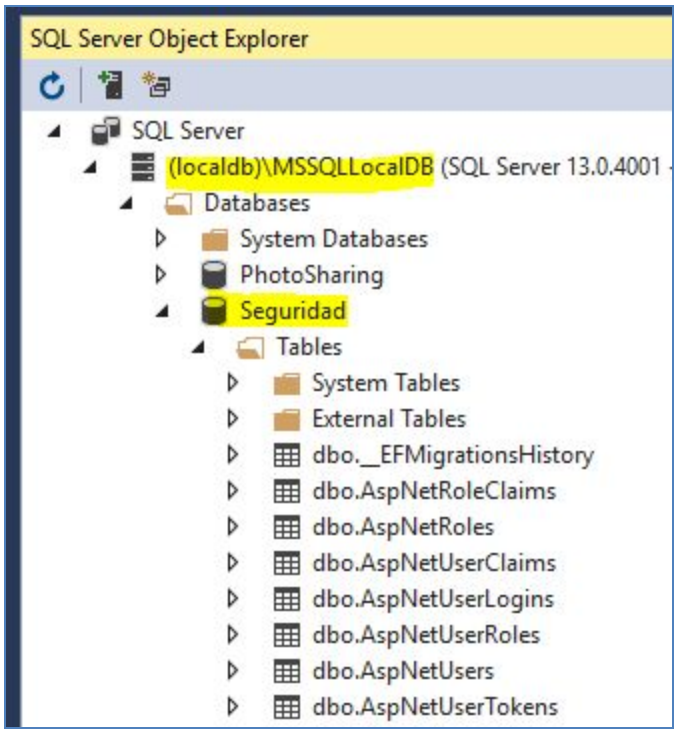
```
                columns: table => new
                {
                    Id = table.Column<int>(nullable: false)
                        .Annotation("SqlServer:ValueGenerationStrategy",
SqlServerValueGenerationStrategy.IdentityColumn),
                    RoleId = table.Column<string>(nullable: false),
                    ClaimType = table.Column<string>(nullable: true),
                    ClaimValue = table.Column<string>(nullable: true)
                },
                constraints: table =>
                {
                    table.PrimaryKey("PK_AspNetRoleClaims", x => x.Id);
                    table.ForeignKey(
                        name: "FK_AspNetRoleClaims_AspNetRoles_RoleId",
                        column: x => x.RoleId,
                        principalTable: "AspNetRoles",
                        principalColumn: "Id",
                        onDelete: ReferentialAction.Cascade);
                });

            migrationBuilder.CreateTable(
                name: "AspNetUserClaims",
                columns: table => new
                {
                    Id = table.Column<int>(nullable: false)
                        .Annotation("SqlServer:ValueGenerationStrategy",
SqlServerValueGenerationStrategy.IdentityColumn),
                    UserId = table.Column<string>(nullable: false),
                    ClaimType = table.Column<string>(nullable: true),
                    ClaimValue = table.Column<string>(nullable: true)
                },
                constraints: table =>
                {
                    table.PrimaryKey("PK_AspNetUserClaims", x => x.Id);
                    table.ForeignKey(
                        name: "FK_AspNetUserClaims_AspNetUsers_UserId",
                        column: x => x.UserId,
                        principalTable: "AspNetUsers",
                        principalColumn: "Id",
                        onDelete: ReferentialAction.Cascade);
                });

            migrationBuilder.CreateTable(
                name: "AspNetUserLogins",
                columns: table => new
                {
                    LoginProvider = table.Column<string>(nullable:
false),
```

```
                    ProviderKey = table.Column<string>(nullable: false),
                    ProviderDisplayName = table.Column<string>(nullable:
true),
                    UserId = table.Column<string>(nullable: false)
                },
                constraints: table =>
                {
                    table.PrimaryKey("PK_AspNetUserLogins", x => new {
x.LoginProvider, x.ProviderKey });
                    table.ForeignKey(
                        name: "FK_AspNetUserLogins_AspNetUsers_UserId",
                        column: x => x.UserId,
                        principalTable: "AspNetUsers",
                        principalColumn: "Id",
                        onDelete: ReferentialAction.Cascade);
                });

            migrationBuilder.CreateTable(
                name: "AspNetUserRoles",
                columns: table => new
                {
                    UserId = table.Column<string>(nullable: false),
                    RoleId = table.Column<string>(nullable: false)
                },
                constraints: table =>
                {
                    table.PrimaryKey("PK_AspNetUserRoles", x => new {
x.UserId, x.RoleId });
                    table.ForeignKey(
                        name: "FK_AspNetUserRoles_AspNetRoles_RoleId",
                        column: x => x.RoleId,
                        principalTable: "AspNetRoles",
                        principalColumn: "Id",
                        onDelete: ReferentialAction.Cascade);
                    table.ForeignKey(
                        name: "FK_AspNetUserRoles_AspNetUsers_UserId",
                        column: x => x.UserId,
                        principalTable: "AspNetUsers",
                        principalColumn: "Id",
                        onDelete: ReferentialAction.Cascade);
                });

            migrationBuilder.CreateTable(
                name: "AspNetUserTokens",
                columns: table => new
                {
                    UserId = table.Column<string>(nullable: false),
```

```
                    LoginProvider = table.Column<string>(nullable:
false),
                    Name = table.Column<string>(nullable: false),
                    Value = table.Column<string>(nullable: true)
                },
                constraints: table =>
                {
                    table.PrimaryKey("PK_AspNetUserTokens", x => new {
x.UserId, x.LoginProvider, x.Name });
                    table.ForeignKey(
                        name: "FK_AspNetUserTokens_AspNetUsers_UserId",
                        column: x => x.UserId,
                        principalTable: "AspNetUsers",
                        principalColumn: "Id",
                        onDelete: ReferentialAction.Cascade);
                });

            migrationBuilder.CreateIndex(
                name: "IX_AspNetRoleClaims_RoleId",
                table: "AspNetRoleClaims",
                column: "RoleId");

            migrationBuilder.CreateIndex(
                name: "RoleNameIndex",
                table: "AspNetRoles",
                column: "NormalizedName",
                unique: true,
                filter: "[NormalizedName] IS NOT NULL");

            migrationBuilder.CreateIndex(
                name: "IX_AspNetUserClaims_UserId",
                table: "AspNetUserClaims",
                column: "UserId");

            migrationBuilder.CreateIndex(
                name: "IX_AspNetUserLogins_UserId",
                table: "AspNetUserLogins",
                column: "UserId");

            migrationBuilder.CreateIndex(
                name: "IX_AspNetUserRoles_RoleId",
                table: "AspNetUserRoles",
                column: "RoleId");

            migrationBuilder.CreateIndex(
                name: "EmailIndex",
                table: "AspNetUsers",
                column: "NormalizedEmail");
```

```
            migrationBuilder.CreateIndex(
                name: "UserNameIndex",
                table: "AspNetUsers",
                column: "NormalizedUserName",
                unique: true,
                filter: "[NormalizedUserName] IS NOT NULL");
        }
```

Y en la base de datos, vemos los objetos creados:



JSon Web Tokens

Teniendo las tablas comenzamos a ver el proceso de autenticación con el esquema **Bearer** basado en tokens.

El formato de token que utilizaremos es de la tecnología **Json Web Tokens**. En síntesis los **JWT** nos permitirán tener datos confiables de los clientes de la aplicación. Estos datos fiables vienen en forma de **claims** que es información confiable acerca de un usuario y el claim es confiable porque son emitidos por alguien en quien confiamos.

Es importante destacar que el sistema de login va a ser válido para trabajar con clientes que sean aplicaciones web como aplicaciones de Angular, React o cualquier otro framework o librería, aplicaciones móviles como Android o iOS y también aplicaciones de escritorio. Y esto funciona porque el **JWT** es simplemente un string

que el Web API va a retornar, y cualquiera que presente dicho token es validado como un usuario autenticado de nuestro sistema.

Siempre que usamos un sistema de login debemos proteger los datos de los usuarios con TLS o SSL, con esto podemos evitar un ataque. El token que el usuario recibe no debe tener mucha información sensible del usuario, sino que sólo debe tener lo mínimo para permitirnos trabajar; datos como la contraseña del usuario bajo ningún concepto deberían estar presentes en el token.

Crearemos un nuevo controlador por el cual los usuarios van a poder registrarse y loguearse al web API. Para ello creamos dos clases en la carpeta **Models**: clase **UsuarioInfo** con email y contraseña, y **UsuarioToken** con el token y la fecha de expiración:

```
using System;
using System.Collections.Generic;
using System.Linq;
using System.Threading.Tasks;

namespace Seguridad.Models
{
    public class UsuarioInfo
    {
        public string Email { get; set; }
        public string Password { get; set; }
    }
}
```

```
using System;
using System.Collections.Generic;
using System.Linq;
using System.Threading.Tasks;

namespace Seguridad.Models
{
    public class UsuarioToken
    {
        public string Token { get; set; }
        public DateTime Expiracion { get; set; }
    }
}
```

Los token no son permanentes necesariamente sino que expiran, y el usuario debe volver a proveer su usuario y contraseña o antes de eso se debe refrescar el token.

La clase **UsuarioInfo** la usaremos para que el usuario pueda proveer el email y la contraseña para loguearse o registrarse.

Creamos un nuevo controlador llamado **CuentaController** con el siguiente código:

```
using Microsoft.AspNetCore.Identity;
using Microsoft.AspNetCore.Mvc;
using Microsoft.Extensions.Configuration;
using Microsoft.IdentityModel.Tokens;
using System;
using System.Collections.Generic;
using System.IdentityModel.Tokens.Jwt;
using System.Linq;
using System.Security.Claims;
using System.Text;
using System.Threading.Tasks;
using Seguridad.Models;

namespace Seguridad.Controllers
{
    [Route("api/[controller]")]
    [ApiController]
    public class CuentaController : ControllerBase
    {
        private readonly UserManager<UsuarioAplicacion> _userManager;
        private readonly SignInManager<UsuarioAplicacion> _signInManager;
        private readonly IConfiguration _configuration;

        public CuentasController(
            UserManager<UsuarioAplicacion> userManager,
            SignInManager<UsuarioAplicacion> signInManager,
            IConfiguration configuration)
        {
            _userManager = userManager;
            _signInManager = signInManager;
            _configuration = configuration;
        }

        [HttpPost("Crear")]
        public async Task<ActionResult<UsuarioToken>>
CrearUsuario([FromBody] UsuarioInfo model)
        {
            var user = new UsuarioAplicacion { UserName = model.Email,
Email = model.Email };
            var result = await _userManager.CreateAsync(user,
model.Password);
            if (result.Succeeded)
```

```
            {
                return CrearToken(model, new List<string>());
            }
            else
            {
                return BadRequest("Usuario o contraseña inválida");
            }

        }

        [HttpPost("Ingresar")]
        public async Task<ActionResult<UsuarioToken>> Ingresar([FromBody]
UsuarioInfo userInfo)
        {
            var result = await
_signInManager.PasswordSignInAsync(userInfo.Email, userInfo.Password,
isPersistent: false, lockoutOnFailure: false);
            if (result.Succeeded)
            {
                var usuario = await
_userManager.FindByEmailAsync(userInfo.Email);
                var roles = await _userManager.GetRolesAsync(usuario);
                return CrearToken(userInfo, roles);
            }
            else
            {
                ModelState.AddModelError(string.Empty, "Ingreso inválido
");
                return BadRequest(ModelState);
            }
        }

        private UsuarioToken CrearToken(UsuarioInfo userInfo,
IList<string> roles)
        {
            var claims = new List<Claim>
            {
        new Claim(JwtRegisteredClaimNames.UniqueName, userInfo.Email),
        new Claim("miValor", "Lo que yo quiera"),
        new Claim(JwtRegisteredClaimNames.Jti, Guid.NewGuid().ToString())
    };

            foreach (var rol in roles)
            {
                claims.Add(new Claim(ClaimTypes.Role, rol));
            }
```

```
            var key = new
SymmetricSecurityKey(Encoding.UTF8.GetBytes(_configuration["JWT:key"]));
            var creds = new SigningCredentials(key,
SecurityAlgorithms.HmacSha256);

            // Tiempo de expiración del token. En nuestro caso lo hacemos
de una hora.
            var expiration = DateTime.UtcNow.AddYears(1);

            JwtSecurityToken token = new JwtSecurityToken(
               issuer: null,
               audience: null,
               claims: claims,
               expires: expiration,
               signingCredentials: creds);

            return new UsuarioToken()
            {
                Token = new JwtSecurityTokenHandler().WriteToken(token),
                Expiracion = expiration
            };
        }
    }

}
```

Básicamente lo que estamos haciendo es crear dos endpoint, uno de creación de usuario y otro de ingreso de usuario.

Creación de usuario (Register)

En el endpoint de crear recibimos un **UsuarioInfo** donde el usuario envía su email y contraseña, e instanciamos de un nuevo **UsuarioAplicacion** y le pasamos esta información para crear el usuario.

Luego si esto fue exitoso creamos un token, con el método **CrearToken** que construye un token instanciando claims. En los claims podemos incluir campos como el nombre único del usuario que en este caso es su email, y también podemos otros valores y la información que guarda es literalmente lo que uno quiera. También podemos utilizar campos especiales como JTI que sirve para identificar en forma única un token, que puede servir para si en algún momento se desea implementar una lógica para invalidar un token. Si detectamos que un usuario con un token está haciendo algo indebido en la página, se puede usar JCI para invalidar ese token y no permitir acceso a su usuario.

Luego podemos se está construyendo una clave simétrica de seguridad que sirve para poder garantizar la autenticidad de la información que estamos recibiendo.

Estamos usando configuration es decir que en algún proveedor de configuración se va a guardar la información de la clave y su web token **JWT** e internamente va a haber un campo llamado **key**.

Vamos a colocar esta información en el **appsettings.json** pero dado que es información muy importante y confidencial de nuestra aplicación lo ideal sería colocarlo en una variable de ambiente pero para mantener este ejemplo simple, lo ubicamos en el **appsettings.json**:

```
{
  "connectionStrings": {
    "defaultConnection": "Data Source=(localdb)\\mssqllocaldb;Initial
Catalog=WebApi2_2_Modulo7;Integrated Security=True"
  },
  "JWT": {
    "key":
"aKLMSLK3I4JNKNDKJFNKJN545N4J5N4J54H4G44H5JBSSDBNF3453S2223KJNF"
  },
  "Logging": {
    "LogLevel": {
      "Default": "Warning"
    }
  },
  "AllowedHosts": "*"
}
```

La clave a colocar debe ser una clave bastante compleja así el algoritmo de encriptación no falla. Se recomiendan claves largas como el string que indicamos.

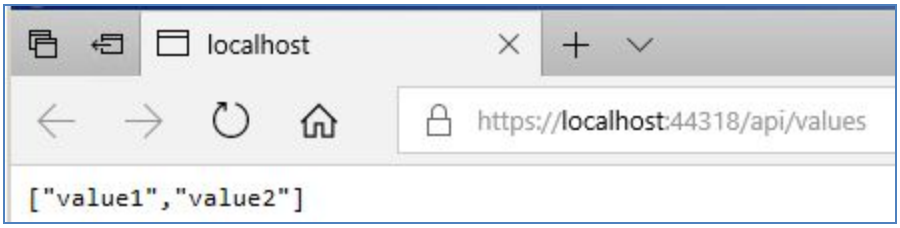
Regresamos al controlador vemos que se instancian credenciales, se define el tiempo de expiración, por ejemplo una hora. La unidad de tiempo se fija según el nivel de seguridad que tenga lo lógica de negocio y la naturaleza de la aplicación, por ejemplo, piense en un homebanking, seguro los tokens no duran días ni horas, tienen duración corta.

Luego instanciamos el token de seguridad, se le pasa información entre ellas los claims, fecha de expiración, credenciales, y finalmente se instancia el objeto de tipo **UsuarioToken**, que es el nuevo token que retorna la función.

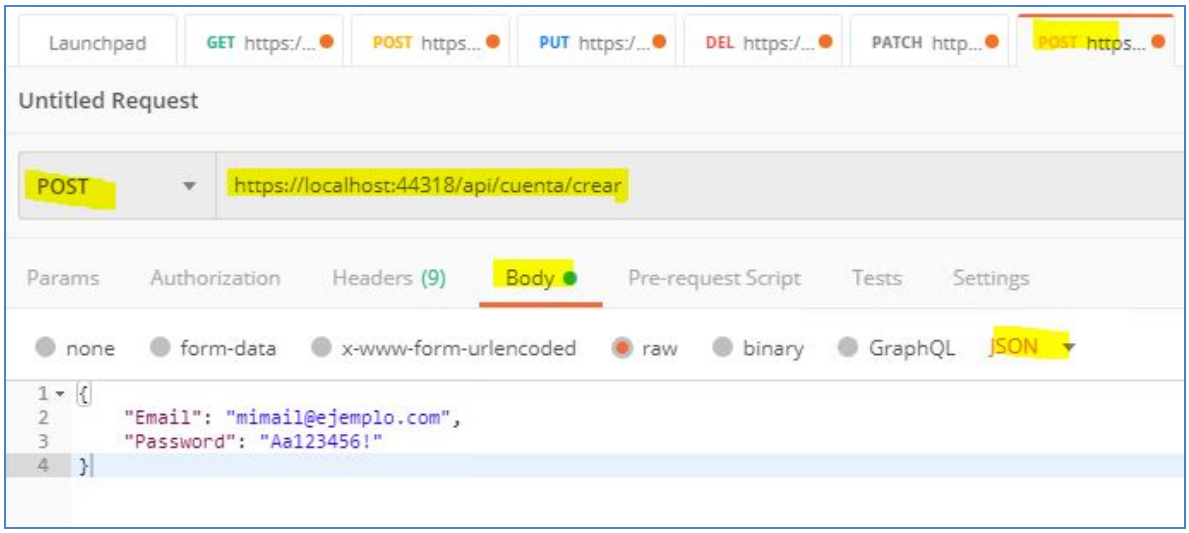
Ingreso de usuario (Login)

Este método es parecido al de creación de usuario, recibe un objeto **UsuarioInfo** y usamos el SignIn Manager para hacer el login pasandole el email y la contraseña. Si esto es exitoso se crea un token y se hace el mismo proceso que hicimos para la parte de creación pero en esta caso solamente para el ingreso del usuario.

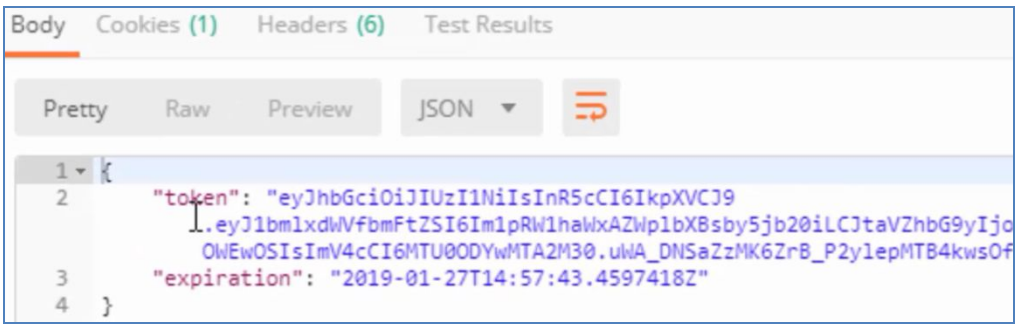
Vamos a probar la aplicación, pulsamos <F5> para dejar la api en ejecución y vamos a Postman para probar los endpoints:



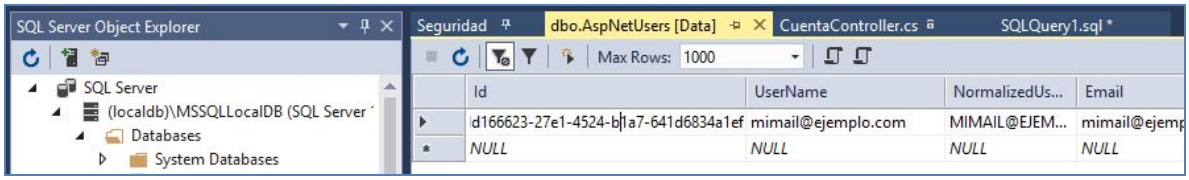
Y en Postman:



El token generado tiene este aspecto:



Y en la base de datos el registro del usuario:



Autenticación de usuarios con tokens

Veremos la autenticación de usuarios pero antes debemos configurar el web API para que entienda que cuando se le pida una autenticación o una autorización debe de tratar leer el token, descifrarlo y verificar que es válido.

En la clase **startup** en el método **ConfigureServices** agregar la configuración (no olvide agregar los Namespaces necesarios:

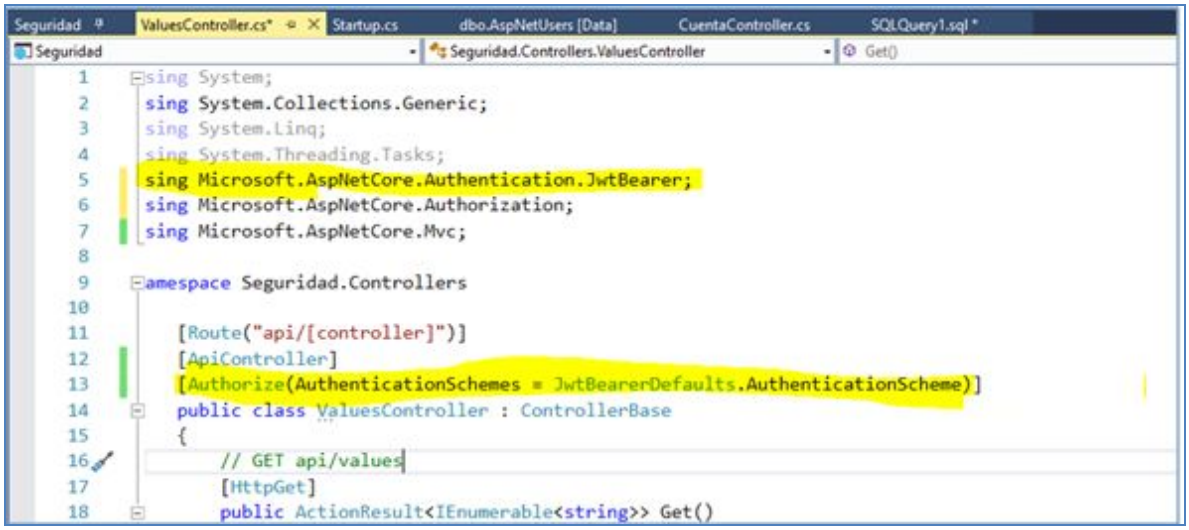
```
        public void ConfigureServices(IServiceCollection services)
        {
            services.AddDbContext<ApplicationDbContext>(options =>

options.UseSqlServer(Configuration.GetConnectionString("defaultConnection
")));

            services.AddIdentity<UsuarioAplicacion, IdentityRole>()
                .AddEntityFrameworkStores<ApplicationDbContext>()
                .AddDefaultTokenProviders();


services.AddMvc().SetCompatibilityVersion(CompatibilityVersion.Version_2_
1);

services.AddAuthentication(JwtBearerDefaults.AuthenticationScheme)
                .AddJwtBearer(options =>
                    options.TokenValidationParameters = new
TokenValidationParameters
                    {
                        ValidateIssuer = false,
                        ValidateAudience = false,
                        ValidateLifetime = true,
                        ValidateIssuerSigningKey = true,
                        IssuerSigningKey = new
SymmetricSecurityKey(Encoding.UTF8.GetBytes(Configuration["jwt:key"])),
                        ClockSkew = TimeSpan.Zero
                    });
        }
```

Indicamos que use el esquema de autenticación basado en tokens (JWT), y le pasamos parámetros para que los valide Estamos indicando también la clave simétrica estamos usando para escribir el token. Básicamente necesitamos la misma clave para escribir y leer el token ya que si las claves no son las mismas entonces no vamos a poder leer el token que recibamos del usuario.

Luego en el método **Configure**, configuramos un middleware de autenticación antes de MVC para que cuando lleguemos al middleware de MVC el sistema de autenticación ya está activado.

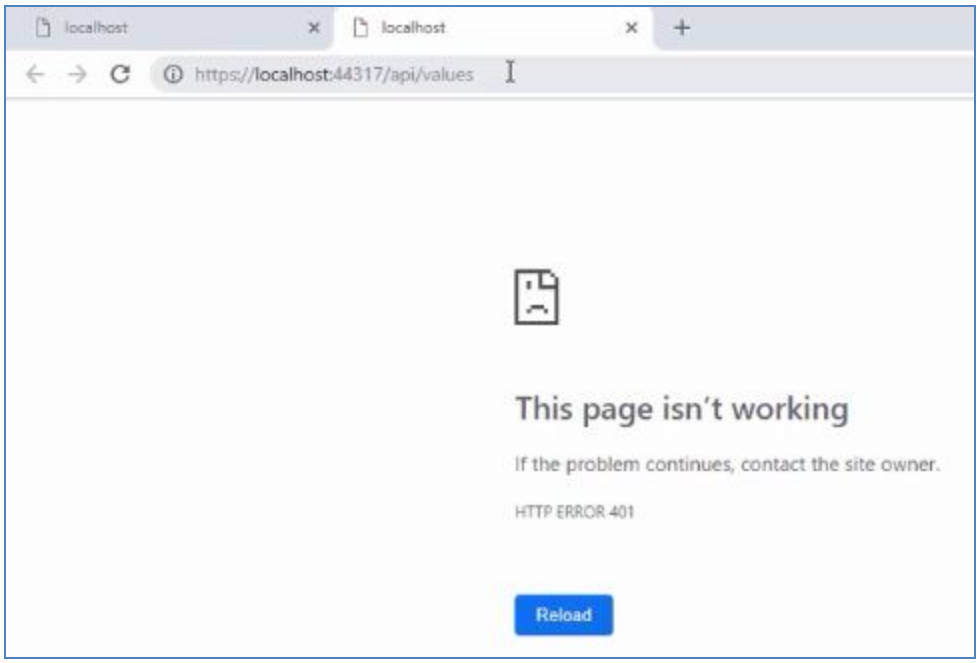
```
        public void Configure(IApplicationBuilder app,
IHostingEnvironment env)
        {
            if (env.IsDevelopment())
            {
                app.UseDeveloperExceptionPage();
            }
            else
            {
                app.UseHsts();
            }

            app.UseHttpsRedirection();
            app.UseAuthentication();
            app.UseMvc();
        }
```

Paso final, en el controlador y a nivel de controlador o a nivel de una acción ubicar el atributo **[Autorize]** indicarle el esquema de autenticación que se va a usar:



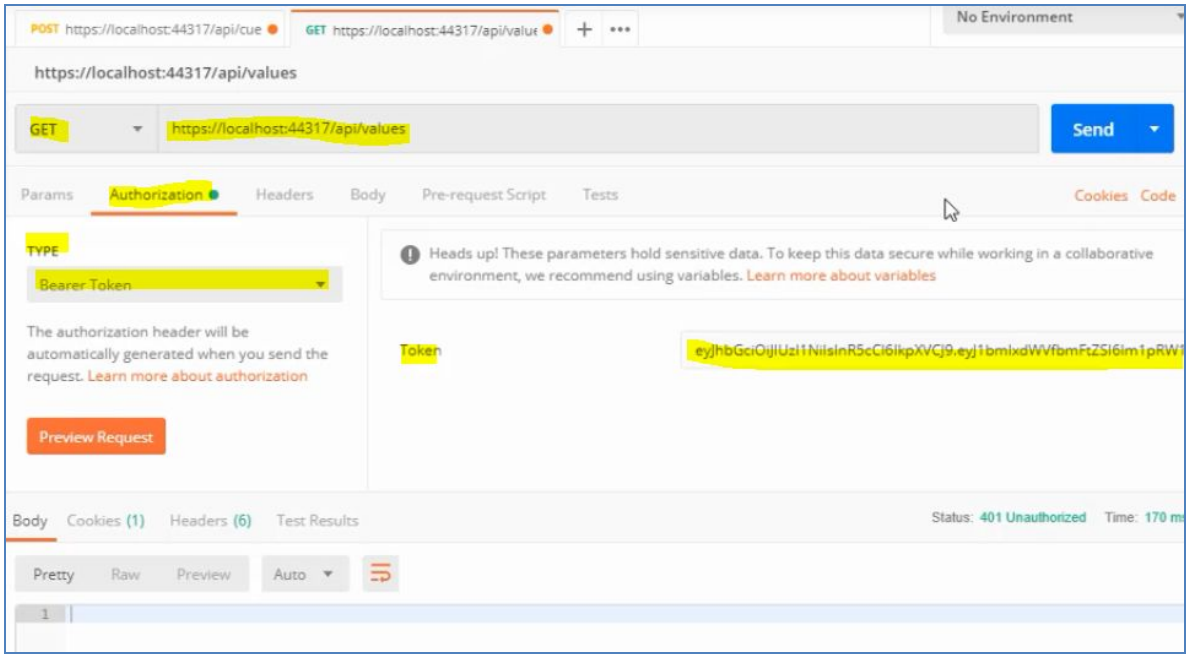
Esto quiere decir entonces que todos los endpoints del controlador van a estar protegidos por el sistema de autenticación JWT.

Probando esto si intentamos acceder al controlador Values obtendremos un error 401 indicando que no estamos autorizados:

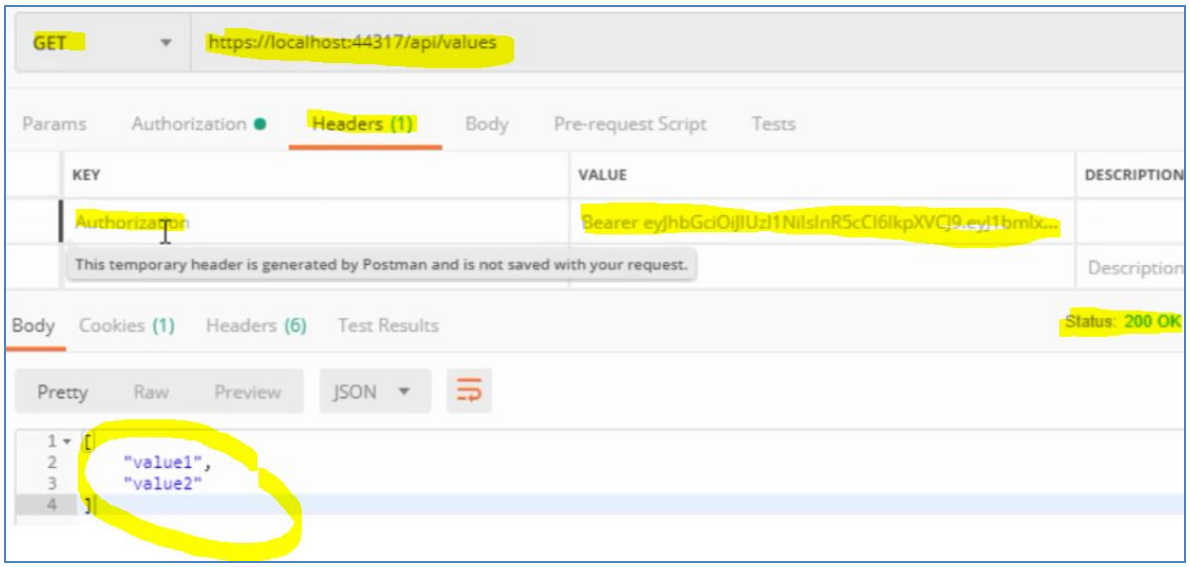


Para solucionar esto debemos enviar en una cabecera HTTP el token que recibimos.

En Postman probando la acción Get con el endpoint, le enviamos el token que obtuvimos antes y le indicamos el tipo de esquema de autenticación:



Luego obtenemos un status 200 ok.



Básicamente esto está colocando un Header que se llama autorization, este header fue creado automáticamente cuando indicamos valores en la solapa Authorization de Postman, y con esto se podrá consumir los endpoints donde se tenga autorización de acceso.

Nota: dejamos aquí otro ejemplo completo sobre autenticación con jwt para Asp.Net Core versión 2.2 y 3.1

https://jasonwatmore.com/post/2019/10/11/aspnet-core-3-jwt-authentication-tutorial-with-example-api

CORS (Cross-origin resource sharing)

Existe un standard de seguridad que por omisión impide que desde un navegador en una página de un origen determinado, hagamos una petición HTTP a un endpoint de otro origen. Simplificando…, si estamos usando el navegador en la página **midominio.com** e intentamos hacer una petición a HTTP vía JavaScript usando Ajax…(u otra tecnología web), a un endpoint del dominio **midominiootro.com**, obtendremos un error.

El nombre de esta política de seguridad es **CORS** y traducido es algo así como "**política de seguridad del mismo origen**".

Aclaramos que ésta es una política de seguridad de navegadores, no habrá este problema si se hace una petición HTTP desde una aplicación móvil o desde un servidor web en general. Además la política se aplica a nivel de origen y no solo de dominio.

Un origen es la combinación de:

Esquema URI + nombre de host + número de puerto

Las siguientes URLs de origen son distintas a **http://miDominio.com**:

Origen	Razón de que sea diferente
http://www.miDominio.com	El subdominio WWW
https://miDominio.com	El protocolo HTTPS
http://miDominio.com:1234	El puerto 1234
http://miDominio.net	El dominio es diferente

Y con más detalle decimos:

- http://www.miDominio.com es diferente es por el subdominio **www**.
- https://miDominio.com es diferente porque usamos otro protocolo, usamos el protocolo seguro https.
- http://miDominio.com:1234 es diferente porque especificamos un número de puerto.
- http://miDominio.net es diferente porque tenemos .net, es otro dominio de nivel superior distinto a .com (https://www.redeszone.net/2018/07/08/dominio-nivel-superior-tld/).

La razón de esta política es prevenir el acceso de un script malicioso en una página a los recursos de otra página. Todo esto es correcto en general pero no es una regla, habrá ocasiones en las que necesitaremos permitir que se hagan peticiones HTTP sobre nuestros endpoints aún cuando vengan de otros orígenes. Esto es frecuente cuando se trabaja con web apis en arquitectura de **microservicios**.

Nota: recomendamos este breve artículo para introducir el tema microservicios https://docs.microsoft.com/es-es/azure/architecture/guide/architecture-styles/microservices

En las aplicaciones .Net que desarrollemos, habilitamos el intercambio de recursos de origen cruzado CORS para permitir peticiones desde otros orígenes hacia el web API. Podemos hacerlo de dos formas:

- A nivel de middleware desde la clase **startup** en **ConfigureServices**
- A nivel de atributo

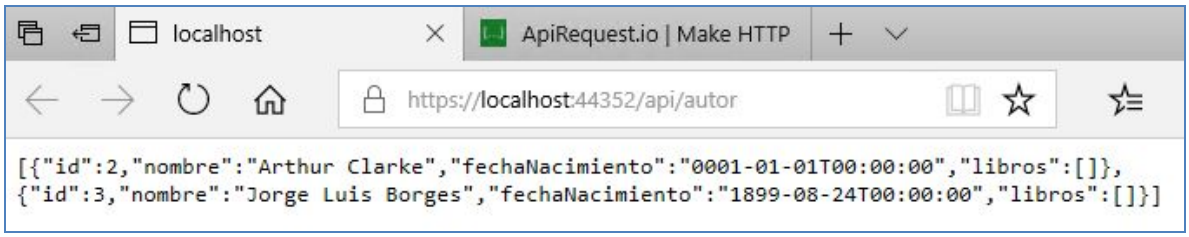
A nivel de middleware

En la clase **startup** en **ConfigureServices** y para iniciar el servicios CORS usamos el método **AddCors**, y luego debemos configurarlos; podemos hacerlo de dos formas, por middleware o a nivel de controladores y acciones.

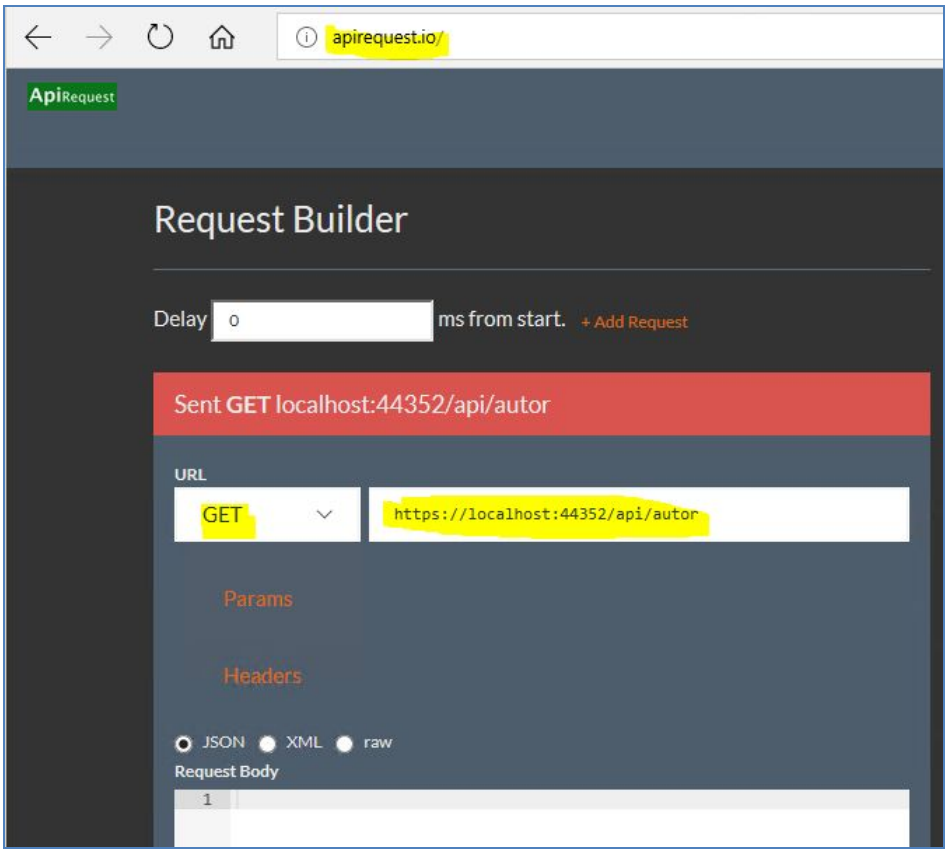
```
public void ConfigureServices(IServiceCollection services)
{
    services.AddCors();
```

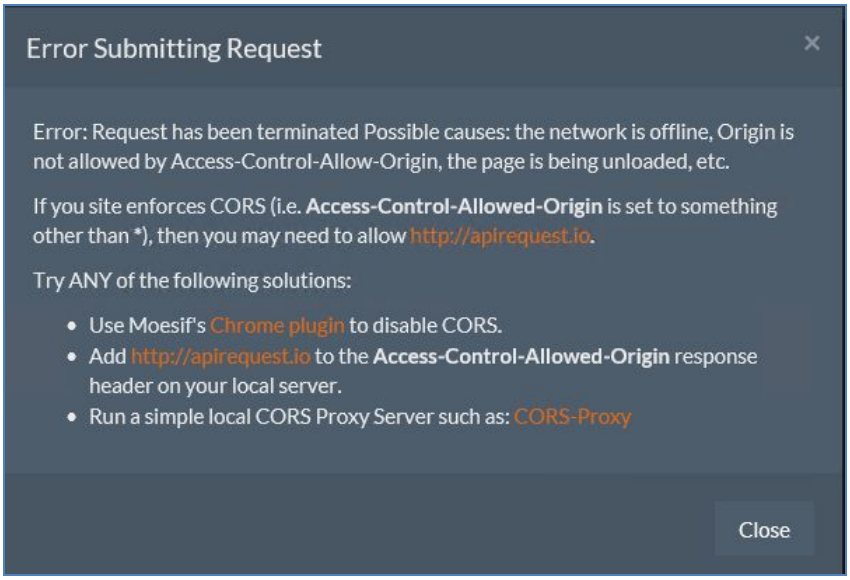
Si lo hacemos por middleware usamos la función **UseCORS** y debemos ubicar este middleware antes de cualquier endpoint que definamos en alguno de nuestros middlewares, en nuestro caso antes del middleware de MVC.

Un ejemplo práctico, si desde el entorno de desarrollo ejecutamos la api de autores de clases anteriores, vemos que accedemos a los endpoints sin problema.

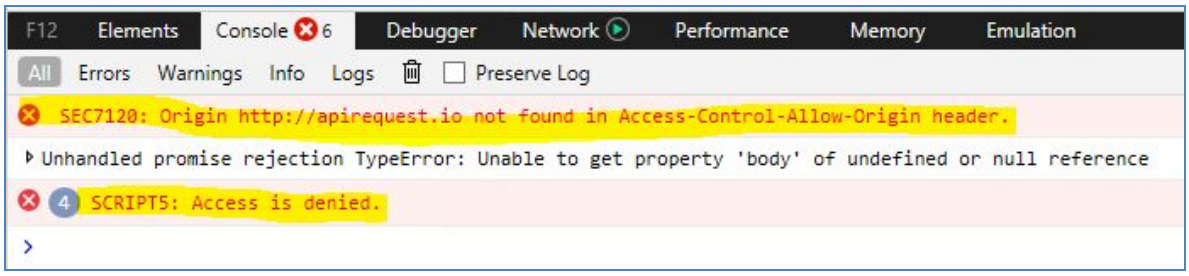


Pero si lo hacemos desde otro origen, por ejemplo apirequest.io da error al no tener la política CORS habilitada:





Y desde la consola F12 del desarrollador vemos también el error:



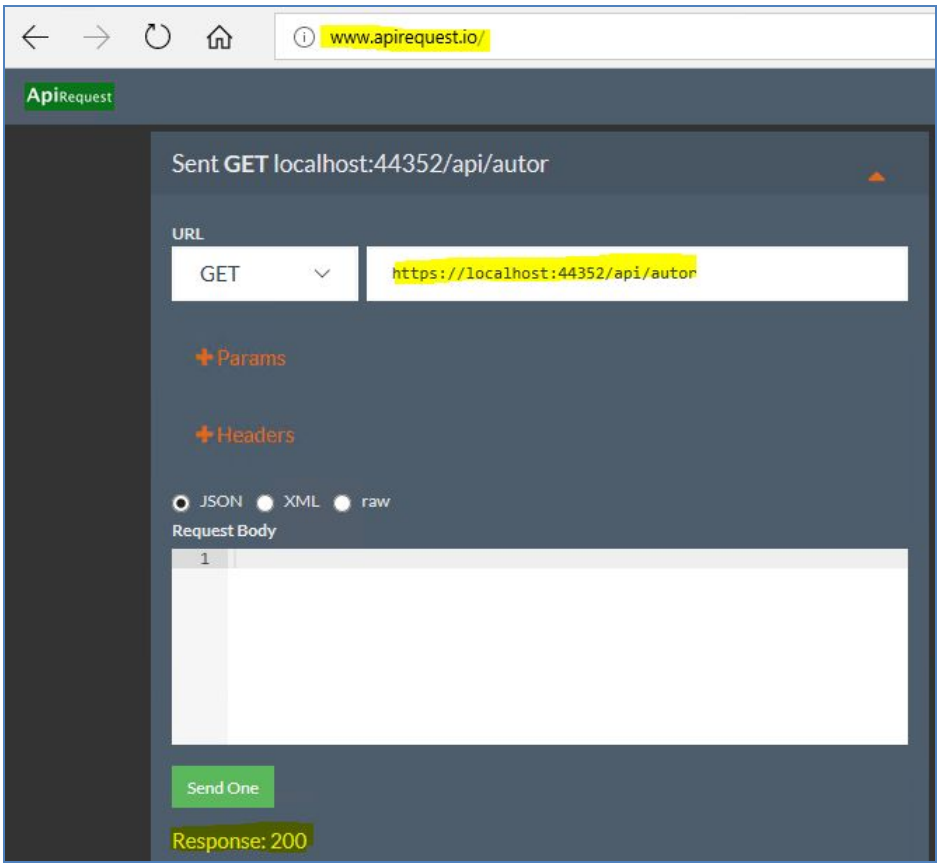
Hemos sido bloqueados por una política de CORS, porque el web API está en el dominio-puerto **https://localhost:44352** y la petición la hacemos desde el dominio **http://apirequest.io/**.

Entonces para configurar CORS por middleware indicamos:

```
        public void Configure(IApplicationBuilder app,
IHostingEnvironment env)
        {
            if (env.IsDevelopment())
            {
                app.UseDeveloperExceptionPage();
            }
            else
            {
                app.UseHsts();
            }

            app.UseHttpsRedirection();
            app.UseAuthentication();
            app.UseCors(builder =>
builder.WithOrigins("http://www.apirequest.io");
            app.UseMvc();
        }
```

En el builder WithOrigins podemos indicar los orígenes que queremos permitir puedan hacer peticiones HTTP a nuestro web API.

Probamos y vemos que ahora si podemos acceder desde otro origen y en el body los datos:



Si queremos refinar más aún, podemos indicar qué acciones específicas puede un origen ejecutar sobre el web API. Por ejemplo supongamos que solamente queremos permitir realizar peticiones get y post, indicaríamos lo siguiente:

```
app.UseCors(builder =>
builder.WithOrigins("http://www.apirequest.io").WithMethods("GET",
"POST").AllowAnyHeader());
```

También podríamos indicar * (asterisco) en los métodos para indicar todos los métodos caso contrario los enumeramos con comas, y * (asterisco) en los orígenes para indicar todos los dominios. Más detalles en este link https://docs.microsoft.com/es-es/aspnet/core/security/cors?view=aspnetcore-3.1

A nivel de atributos

También podemos manejar CORS a través de atributos a nivel de controladores y acciones. Esto se hace usando políticas de seguridad específicas para CORS que permitirán centralizar configuraciones de CORS en un solo lugar, de tal forma que más adelante podamos asignar esta política de seguridad para distintos endpoints del web api.

Con las políticas de seguridad podemos realizar las mismas configuraciones de origen, métodos HTTP y Headers con el middleware **UseCors**.

Para crear la política de seguridad a la que le podemos poner un nombre como *PermitirApiRequest*, vamos al método **ConfigureServices** e indicamos:

```
public void ConfigureServices(IServiceCollection services)
        {
            services.AddCors(options =>
            {
                options.AddPolicy("PermitirApiRequest",
                    builder =>
builder.WithOrigins("http://www.apirequest.io").WithMethods("GET",
"POST").AllowAnyHeader());
            });
```

Y en el método Configure solo **UseCors** sin parámetros:

```
        public void Configure(IApplicationBuilder app,
IHostingEnvironment env)
        {
            if (env.IsDevelopment())
            {
                app.UseDeveloperExceptionPage();
            }
            else
            {
                app.UseHsts();
            }

            app.UseHttpsRedirection();
            app.UseAuthentication();
            app.UseCors();
            app.UseMvc();
        }
```

Y en un controlador indicamos la política de seguridad que puede estar a nivel del controlador para que aplique a todos los endpoints, o de nivel de acciones específicas. El atributo es el mismo y se le pasa por parámetro el nombre de la política de seguridad, en este ejemplo *PermitirApiRequest*:

A nivel de controlador:

```
using System;
using System.Collections.Generic;
using System.Linq;
using System.Threading.Tasks;
using Microsoft.AspNetCore.Authentication.JwtBearer;
using Microsoft.AspNetCore.Authorization;
using Microsoft.AspNetCore.Cors;
using Microsoft.AspNetCore.Mvc;

namespace Seguridad.Controllers
{
    [Route("api/[controller]")]
    [ApiController]
    [EnableCors("PermitirApiRequest")]

    public class ValuesController : ControllerBase
    {
```

A nivel de acción:

```
        [HttpGet]
        [EnableCors("PermitirApiRequest")]
        public ActionResult<IEnumerable<string>> Get()
        {
            return new string[] { "value1", "value2" };
        }
```

HTTPS

TLS (Transport Layer Security) o en español *seguridad de la capa de transporte* es un protocolo estándar para establecer una conexión segura entre un servidor web y un cliente web. Bajo este protocolo queda encriptada la información que se envía a través de la red. Esto da seguridad que un tercero no podrá "espiar" los datos que se transmiten entre un servidor web y el cliente. Hoy en día se espera que cualquier servidor web use TLS para cuidar la privacidad de nuestros usuarios.

Cuando nos conectamos a un web api con TLS usamos el protocolo **HTTPS** (Hyper Text Transfer Protocol Secure). Al crear el proyecto con .Net Core, se crearon middlewares para esto: **UseHttpsRedirection** que sirve para redirigir peticiones HTTP a HTTPS, y **UseHsts** sirve para enviar a nuestros clientes la cabecera Strict-Transport- Security (HSTS), que le indica a los navegadores que nuestro sitio web sólo debería ser accedido usando HTTPS y no HTTP; esta opción HSTS es relativamente buena pero sólo funciona si los clientes respetan la cabecera HTTPS lo que es normal en navegadores web pero no en todos los clientes de un web API.

La idea de estos dos middleware es pedir a los clientes que usen HTTPS para conectarse al web api. En ASP.Net Core existe el atributo **RequiredHTTPS** que sirve para requerir HTTPS sobre un endpoint, es decir obligar a que se use el protocolo seguro. Esto está bien para navegadores web pero no es suficiente para web APIs.

De hecho la documentación oficial en la actualidad hace hincapié en no utilizar el atributo HTTPS en web APIs, porque por ejemplo no protege la información de un HTTP POST en caso que un usuario envíe esta petición HTTP a un endpoint.

Entonces, la mejor opción en estos casos es solamente permitir conexiones HTTPS al web API. Esto se logra con una configuración a nivel del servidor, es decir configurar el servidor para que rechace todas las conexiones HTTP. Esto va a variar según donde el servidor donde se publique el Web API.