Redes de computadoras - Uncuyo
Augusto Robles - 11737 | Martin Farres - 13495

UNCUYO UNIVERSIDAD NACIONAL DE CUYO | FACULTAD DE INGENIERÍA Licenciatura en Ciencias de la Computación

Proyecto Integrador - Año 2024

Autenticación de API REST con Token JWT en NET CORE

Conocimientos integrados

- **Capa de aplicación**: Servidor HTTP, peticiones GET, POST, visualizacion dinamica de la información en el cliente (Swagger)

- **Capa de seguridad:** Autenticación y autorización de aplicaciones, firmas digitales y algoritmos de seguridad HMACSHA512

1 DESCRIPCIÓN

Este documento describe la forma de implementar un servidor de autenticación web api utilizando JWT.

2 CONFIGURACIÓN INICIAL

2.1 Paquetes necesarios

Para comenzar a trabajar en la implementación de nuestro servidor de autenticación con JWT necesitamos primero instalar los siguientes paquetes (Utilizando el NuGet Packet Manager de Visual Studio:

- **EntityFrameworkCore**
- **EFCore.Design**
- **EFCore.SqlServer**
- **AspNetCore.Identity.EFCore**
- **AspNetCore.Authentication.JwtBearer**

¿Qué provee el paquete Identity en dotnet?

1. **Gestión de Usuarios:**
    - Creación, actualización y eliminación de cuentas de usuario.
    - Recuperación de cuentas, incluyendo el restablecimiento de contraseñas y la recuperación de nombres de usuario.
2. **Gestión de Roles:**
    - Creación y gestión de roles de usuario.
    - Asignación de usuarios a roles específicos.
    - Verificación de roles de usuario para la autorización.
3. **Autenticación:**

- Proporciona métodos para registrar y autenticar usuarios.
- Soporte para autenticación basada en cookies y tokens.
- Integración con proveedores de autenticación externa como Google, Facebook, Microsoft y otros mediante OAuth.

4. **Autorización:**
   - Control de acceso basado en roles y políticas.
   - Capacidades para crear reglas de autorización personalizadas.
   - Integración con políticas de autorización para recursos específicos.
5. **Gestión de Contraseñas:**
   - Reglas y políticas de complejidad de contraseñas.
   - Soporte para el cambio y restablecimiento de contraseñas.
   - Bloqueo de cuentas después de múltiples intentos fallidos de inicio de sesión.
6. **Seguridad:**
   - Almacenamiento seguro de contraseñas usando hashing.
   - Protección contra ataques de fuerza bruta y enumeración de cuentas.
   - Generación y verificación de tokens para acciones seguras (como la confirmación de cuenta y el restablecimiento de contraseñas).
7. **Requisitos de Validación:**
   - Validación de usuarios y contraseñas según reglas definidas.
   - Personalización de los requisitos de validación para cumplir con las políticas de seguridad específicas de la aplicación.
8. **Reivindicaciones y Tokens:**
   - Soporte para el manejo de reivindicaciones (claims) que se pueden usar para autorización basada en políticas.
   - Generación de tokens de acceso y refresh tokens para autenticación basada en JWT.

2.2 Conexión a BD y tablas Identity

1. Crearemos el contexto para nuestra base de datos, el cual heredará de IdentityDbContext. Ya que este implementa los DbSet y configuraciones necesarias para crear las tablas de usuario, roles, tokens, entre otras.

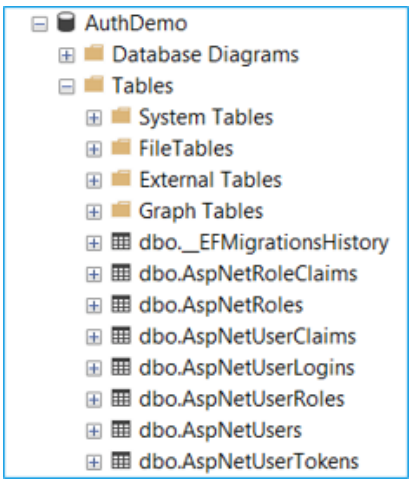
Ver documentación oficial: [Use Identity to secure a Web API backend for SPAs | Microsoft Learn]


```
public class AuthDemoDbContext : IdentityDbContext
{
    0 references
    public AuthDemoDbContext(DbContextOptions options) : base(options)
    {

    }

}
```

2. Configuramos el connection string para nuestra base de datos en el appsettings.json y agregamos el servicio en el Program.cs de nuestra app

```
builder.Services.AddDbContext<AuthDemoDbContext>(options =>
{
    options.UseSqlServer(builder.Configuration.GetSection("ConnectionStrings:DefaultConnection").Value);
});
```

3. Realizamos la primera migración y update de nuestra base de datos. Veremos que EntityFramework Core generará las siguientes tablas en la base de datos:



- **AspNetUsers:** Esta tabla almacena la información básica de los usuarios, como el nombre de usuario, la dirección de correo electrónico, el número de teléfono, etc.
- **AspNetRoles:** Esta tabla contiene los diferentes roles que se pueden asignar a los usuarios, como "Administrador", "Usuario regular", etc.
- **AspNetUserRoles:** Esta tabla es una tabla de unión que mapea los usuarios con sus respectivos roles.

- **AspNetUserClaims:** Esta tabla almacena las reclamaciones (claims) de los usuarios, que son afirmaciones sobre el usuario, como su dirección de correo electrónico, su número de teléfono, etc.
- **AspNetUserLogins:** Esta tabla almacena los proveedores de autenticación externos que un usuario ha utilizado para iniciar sesión, como Google, Facebook, etc.
- AspNetUserTokens: Esta tabla almacena los tokens de restablecimiento de contraseña y tokens de confirmación de correo electrónico para los usuarios.
- **AspNetRoleClaims:** Esta tabla almacena las reclamaciones (claims) de los roles.

Puedes personalizar estas tablas agregando más propiedades o modificando las existentes según tus necesidades. Además, Entity Framework Core Identity proporciona varias clases y métodos para interactuar con estas tablas y manejar la autenticación y autorización de usuarios en tu aplicación.

Es importante tener en cuenta que estas tablas son específicas de Entity Framework Core Identity y pueden variar dependiendo de la versión de .NET Core y de las configuraciones que hayas realizado en tu aplicación.

3 CONTROLLERS

Implementaremos 2 controladores, AuthController y TestController. El primero para el proceso de registro y login, mientras que el segundo será utilizado para probar y verificar la lógica de autenticación y autorización a recursos dentro de nuestra app.

3.1 AuthController

El auth controller implementará 2 actions methods para sus 2 endpoints. Un action method para Login y otro para Register. Es importante notar la necesidad de la creación de DTOs necesarios tanto para registro y login.

```
[HttpPost("login")]
0 references
public async Task<IActionResult> Login([FromBody] LoginDto loginDto)
{
    if (!ModelState.IsValid)
    {
        return BadRequest(ModelState);
    }

    var result = await _authService.Login(loginDto);

    if (!result.Succeeded)
    {
        return Unauthorized();
    }

    var tokenString = _authService.GenerateTokenString(loginDto);
    return Ok(tokenString);
}
```

La lógica de negocio de cada endpoint será delegada a authService, servicio que se inyecta en el constructor de cada Controlador. De esta manera trabajamos con una arquitectura en capas, modularizada y escalable.

```
[HttpPost("register")]
0 references
public async Task<IActionResult> RegisterUser([FromBody] RegisterDto registerUser)
{
    if (!ModelState.IsValid)
    {
        return BadRequest(ModelState);
    }

    var result = await _authService.RegisterUser(registerUser);

    if (!result.Succeeded)
    {
        return BadRequest("Something went wrong");
    }

    return Ok("Successfully Registered!");
}
```

Se adjunta a continuación la estructura básica de los DTOs utilizados en dichos endpoints:

```
public class RegisterDto
{
    [Required]
    [EmailAddress]
    1 reference
    public string Email { get; set; }

    [Required]
    1 reference
    public string Username { get; set; }

    [Required]
    [MinLength(8)]
    1 reference
    public string Password { get; set; }
}
```

```
public class LoginDto
{
    [Required]
    2 references
    public string Username { get; set; }
    [Required]
    1 reference
    public string Password { get; set; }
}
```

3.2 TestController

Tendrá un solo endpoint con un filtro de autorización para verificar el funcionamiento de la seguridad de nuestra API.

```
[Route("api/[controller]")]
[ApiController]
[Authorize]
0 references
public class TestController : ControllerBase
{
    [HttpGet]
    0 references
    public string Get()
    {
        return "You hit me!";
    }
}
```

4 CONFIGURACIÓN DE LA AUTENTICACIÓN

Antes de construir nuestro AuthService es necesario agregar y configurar los servicios provistos por Identity, como también el Middleware de autenticación.

Estos servicios y configuraciones se realizaran en el archivo Program.cs de nuestra aplicación.

4.1 Identity Service

```
builder.Services.AddIdentity<IdentityUser, IdentityRole>(options =>
{
    options.Password.RequiredLength = 8;

}).AddEntityFrameworkStores<AuthDemoDbContext>()
.AddDefaultTokenProviders();
```

Al agregar el servicio se pueden configurar varias opciones para personalizar el comportamiento del sistema de identidad y autenticación. Algunas de las opciones más comunes son:

- **Opciones de contraseña**
- **Opciones de usuario**
- **Opciones de bloqueo**
- **Opciones de tokens**
- **Opciones de inicio de sesión**
- **Opciones de almacenamiento**

Para profundizar en las opciones y configuraciones: [Configure ASP.NET Core Identity | Microsoft Learn]

4.2 Middleware de authenticacion

```
builder.Services.AddAuthentication(options =>
{
    options.DefaultAuthenticateScheme = JwtBearerDefaults.AuthenticationScheme;
    options.DefaultChallengeScheme = JwtBearerDefaults.AuthenticationScheme;
}).AddJwtBearer(options =>
{
    options.TokenValidationParameters = new TokenValidationParameters()
    {
        ValidateActor = true,
        ValidateIssuer = true,
        ValidateAudience = true,
        RequireExpirationTime = true,
        ValidateIssuerSigningKey = true,

        ValidIssuer = builder.Configuration.GetSection("Jwt:Issuer").Value,
        ValidAudience = builder.Configuration.GetSection("Jwt:Audience").Value,

        IssuerSigningKey = new SymmetricSecurityKey(
            Encoding.UTF8.GetBytes(builder.Configuration.GetSection("Jwt:Key").Value))
    };
});
```

Esta sección del código está configurando la autenticación basada en JSON Web Tokens (JWT) en la aplicación ASP.NET Core. Aquí hay una explicación detallada de lo que está sucediendo:

1. `builder.Services.AddAuthentication(options => {...})`: Esta línea agrega el servicio de autenticación a la aplicación y configura las opciones del esquema de autenticación predeterminado.

2. `options.DefaultAuthenticateScheme = JwtBearerDefaults.AuthenticationScheme;` y `options.DefaultChallengeScheme = JwtBearerDefaults.AuthenticationScheme;`: Estas líneas establecen el esquema de autenticación predeterminado como "JwtBearer", lo que indica que la aplicación utilizará el esquema de autenticación JWT Bearer.

3. `.AddJwtBearer(options => {...})`: Esta línea agrega el middleware de autenticación JWT Bearer a la aplicación y configura las opciones del middleware.

4. `options.TokenValidationParameters = new TokenValidationParameters() {...}`: Aquí se configuran los parámetros de validación del token JWT. Esto es crucial para garantizar la integridad y la seguridad del token.

- `ValidateActor`: Validará el valor del campo "actor" del token JWT.
- `ValidateIssuer`: Validará que el "issuer" (emisor) del token coincida con el valor esperado.

- `ValidateAudience`: Validará que el "audience" (audiencia) del token coincida con el valor esperado.

- `RequireExpirationTime`: Validará que el token tenga una hora de expiración establecida.

- `ValidateIssuerSigningKey`: Validará que la clave de firma del token sea válida.

5. `ValidIssuer = builder.Configuration.GetSection("Jwt:Issuer").Value`: Esta línea lee el valor del "issuer" válido desde el archivo de configuración.

6. ValidAudience = builder.Configuration.GetSection("Jwt:Audience").Value: Esta línea lee el valor de la "audiencia" válida desde el archivo de configuración.

7. IssuerSigningKey = new SymmetricSecurityKey(Encoding.UTF8.GetBytes(builder.Configuration.GetSection("Jwt:Key").Value))`: Esta línea crea una clave de seguridad simétrica a partir de una clave secreta obtenida del archivo de configuración. Esta clave se utilizará para firmar y validar los tokens JWT.

En resumen, esta sección del código configura la autenticación JWT Bearer en la aplicación. Establece los parámetros de validación del token, como el emisor, la audiencia y la clave de firma, que se leen desde el archivo de configuración. Esta configuración es esencial para garantizar la seguridad y la integridad de los tokens JWT utilizados para la autenticación en la aplicación.

Al terminar de realizar dicha configuración, no nos olvidemos de agregar en nuestro Program.cs los middleware:

- App.UseAuthentication()
- App.UseAuthorization()

5 AUTH SERVICE

Ya configurados los middlewares de Identity, Authentication y JwtBearer, podemos trabajar en la construcción de la capa de servicios para la autenticación, es decir, AuthService. Este servicio se encargará de toda la lógica referida al login, registro y generación del token JWT.

A continuación las dependencias que inyectaremos en nuestro AuthService, las cuales son requeridas para la busqueda de usuarios en la BD, registro, login y validación de passwords y credenciales.


```
namespace AuthDemo.Services
{
    2 references
    public class AuthService : IAuthService
    {
        private readonly UserManager<IdentityUser> _userManager;
        private readonly SignInManager<IdentityUser> _signInManager;
        private readonly IConfiguration _config;

        0 references
        public AuthService(
            UserManager<IdentityUser> userManager,
            SignInManager<IdentityUser> signInManager,
            IConfiguration config)
        {
            _userManager = userManager;
            _signInManager = signInManager;
            _config = config;
        }
```

Para profundizar en los metodos provistos por Identity de dotnet core: [UserManager SignInManager in ASP.NET Core Identity - Dot Net Tutorials]

Los métodos principales de nuestro servicio serán dos, uno para el registro y otro para el login de nuestros usuarios. Tener en cuenta la definición de las interfaces correspondientes contemplando la posterior inyección de este servicio en nuestro controlador de autenticación.

```
2 references
public async Task<IdentityResult> RegisterUser(RegisterDto registerDto)
{
    var identityUser = new IdentityUser
    {
        UserName = registerDto.Username,
        Email = registerDto.Email
    };

    var result = await _userManager.CreateAsync(identityUser, registerDto.Password);
    return result;
}
```

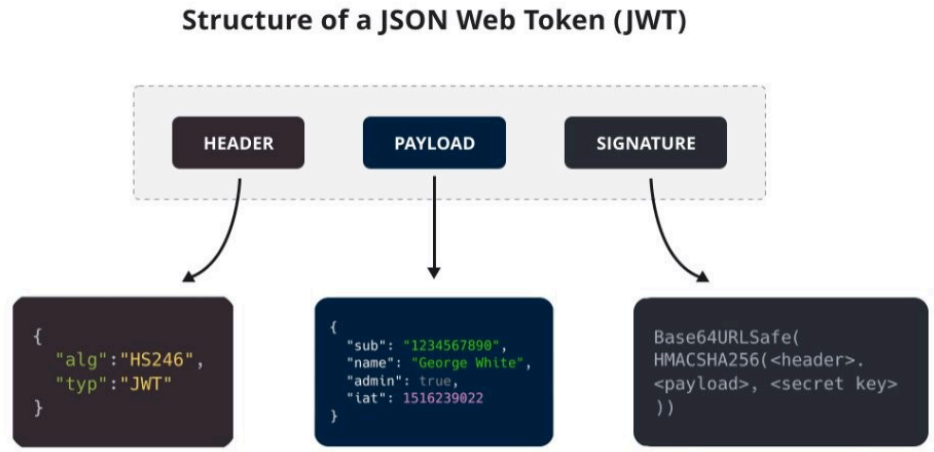
```
public async Task<SignInResult> Login(LoginDto loginDto)
{
    var user = await _userManager.FindByNameAsync(loginDto.Username);

    if (user == null)
    {
        return SignInResult.Failed;
    }

    var result = await _signInManager.PasswordSignInAsync(user, loginDto.Password, false, false);

    return result;
}
```

UNCUYO UNIVERSIDAD NACIONAL DE CUYO | FACULTAD DE INGENIERÍA Licenciatura en Ciencias de la Computación

6 GENERACIÓN DEL TOKEN JWT

6.1 Token JWT

Un token JWT (JSON Web Token) es un estándar abierto (RFC 7519) que define una forma compacta y segura de transmitir información entre partes como un objeto JSON. Es ampliamente utilizado para la autenticación y la transmisión de datos en aplicaciones web y API.



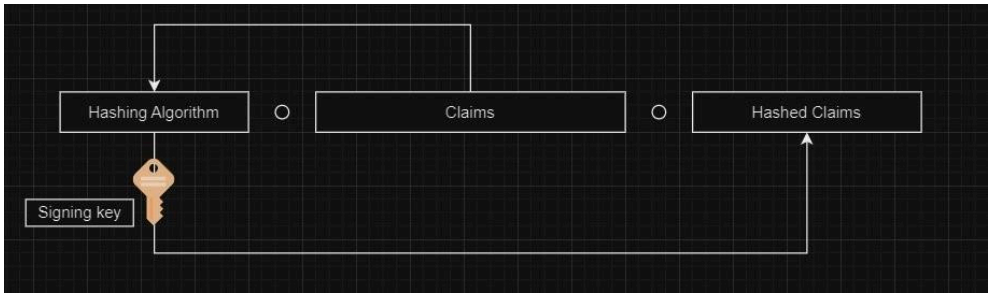
- **Encabezado (Header):** Contiene metadatos sobre el tipo de token y el algoritmo de cifrado utilizado. Por ejemplo: {"alg": "HS256", "typ": "JWT"}.

- **Carga útil (Payload):** Contiene las "claims" (afirmaciones o datos), que son las piezas de información codificadas en el token. Por ejemplo, podría contener el nombre de usuario, una marca de tiempo de emisión, una marca de tiempo de expiración, etc.

- **Firma (Signature):** Es una porción codificada que permite validar la autenticidad del token. Se crea mediante la codificación del encabezado, la carga útil y una clave secreta con el algoritmo especificado en el encabezado.

Documentación sobre JWT: [JSON Web Tokens (auth0.com)]

6.2 Generación del JWT



Al crear un token JWT con JwtSecurityToken, se proporciona el objeto SigningCredentials con la clave y el algoritmo adecuados. Luego, el token se firma utilizando estas credenciales antes de ser serializado y enviado al cliente.

Cuando se recibe un token JWT en el lado del servidor, el objeto SigningCredentials se utiliza para validar la firma del token y asegurar que no haya sido modificado. Si la validación falla, el token se considera no válido y se rechaza.

Por lo tanto crearemos un nuevo método en nuestro AuthService, este será llamado antes de retornar Ok() en el login. Para así poder enviar el JWT en dicha respuesta.

¿Cómo creamos el Token y lo devolvemos?

Punto de partida en el metodo:

```
public string GenerateTokenString(LoginDto loginDto)
{

    string tokenString = new JwtSecurityTokenHandler().WriteToken(securityToken);
    return tokenString;
}
```

El handler para escribir el token necesita recibir un SecurityToken, esta es una clase abstracta, que en nuestro caso usaremos su clase derivada JwtSecurityToken.

Creación del JwtSecurityToken

```
var securityToken = new JwtSecurityToken(
    signingCredentials: signingCreds,
    claims: claims,
    issuer: _config.GetSection("Jwt:Issuer").Value,
    audience: _config.GetSection("Jwt:Audience").Value,
    expires: DateTime.Now.AddMinutes(60)
    );
```

Al ver los parametros del objeto vemos que también necesitaremos:

- **SigningCredentials:**
- **Claims**
- **Issuer y Audience**
- **Expires (Tiempo de vida del token)**

SigningCredentials

El objeto SigningCredentials desempeña un papel crucial en la firma y verificación de tokens JWT. Su función principal es proporcionar las credenciales necesarias para firmar o validar la integridad de un token JWT.

```
var signingCreds = new SigningCredentials(
    securityKey,
    SecurityAlgorithms.HmacSha512Signature);
```

Contiene dos componentes principales:

- **Clave de firma (Key):** Esta es la clave secreta utilizada para firmar y validar el token JWT. Puede ser una clave simétrica (compartida entre las partes que firman y validan el token) o una clave asimétrica (una clave privada para firmar y una clave pública para validar).

- **Algoritmo de firma (Algorithm):** Este es el algoritmo criptográfico utilizado para firmar y validar el token JWT. Los algoritmos comunes incluyen HMAC (como HS256, HS384, HS512) para claves simétricas y RSA (como RS256, RS384, RS512) para claves asimétricas.

Claims

Los claims son afirmaciones o declaraciones sobre una entidad (generalmente un usuario o un cliente) que se incluyen dentro del payload (cuerpo) de un JWT. Estas declaraciones contienen información sobre el sujeto del token, como su identidad, roles, permisos, datos personalizados, etc.

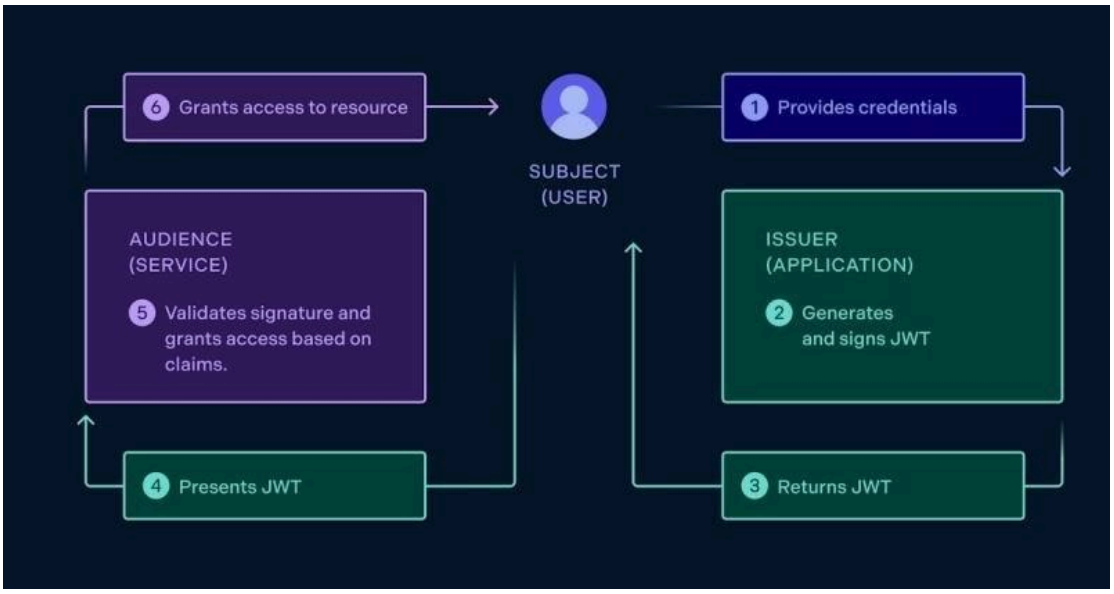
```
var claims = new List<Claim>
{
    new Claim(ClaimTypes.GivenName, loginDto.Username),
    new Claim(ClaimTypes.Role, "Admin")
};
```

Ejemplo de Claims Comunes

- **sub: (subject)** El identificador del usuario.
- **name:** El nombre del usuario.
- **email:** La dirección de correo electrónico del usuario.
- **role:** El rol del usuario, como "admin" o "user".
- **iat:** (issued at) Fecha y hora en la que se emitió el token.
- **exp:** (expiration) Fecha y hora en la que expira el token.

UNCUYO UNIVERSIDAD NACIONAL DE CUYO | FACULTAD DE INGENIERÍA Licenciatura en Ciencias de la Computación

Issuer y Audience



Issuer (Emisor): Identifica al emisor o la entidad que generó el JWT.

Su propósito principal es permitir que el servicio que recibe y valida el token pueda verificar si el emisor es una entidad de confianza.

Esto ayuda a prevenir ataques de suplantación de identidad, ya que un token emitido por una entidad no confiable sería rechazado.

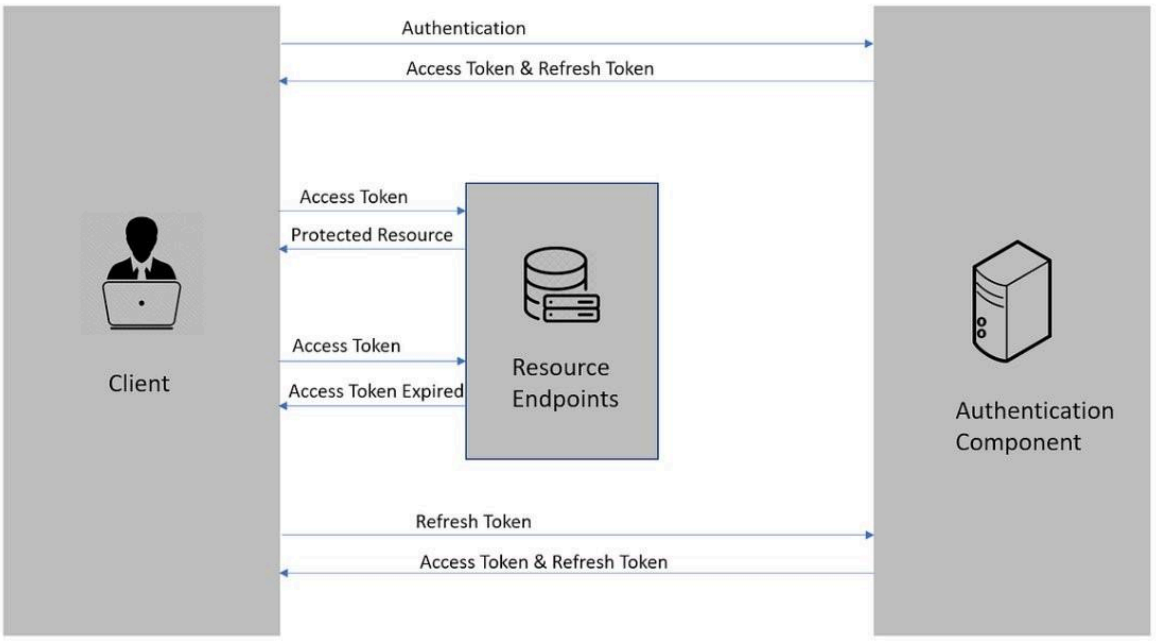
Audience (Audiencia): Define la audiencia o los destinatarios previstos del JWT.

Esto permite que el servicio que recibe el token verifique si está destinado a ser consumido por él mismo o por un conjunto específico de servicios.

Esta propiedad ayuda a proteger contra ataques de "confusión de destino", donde un token válido se envía al destinatario incorrecto.

Expires (Tiempo de vida del Token)

La propiedad Expires (o exp como se conoce en el estándar JWT) cumple una función crucial en el JwtSecurityToken al establecer el tiempo de vida o la fecha de expiración del token.



La función principal de Expires es definir un límite de tiempo después del cual el token JWT se considerará inválido y no se aceptará por los servicios que lo validan. Esto aporta seguridad y control sobre la validez de los tokens emitidos.

7 CONFIGURACION SWAGGER

Este código se ejecuta durante la configuración de la aplicación ASP.NET Core y utiliza la biblioteca Swashbuckle.AspNetCore para configurar Swagger (una herramienta de documentación y prueba de APIs).

UNCUYO UNIVERSIDAD NACIONAL DE CUYO | FACULTAD DE INGENIERÍA Licenciatura en Ciencias de la Computación

```
builder.Services.AddSwaggerGen(options =>
{
    options.SwaggerDoc("v1", new OpenApiInfo
    {
        Version = "v1",
        Title = "AuthDemo",
        Description = "Authentication|Authorization|JWT Demo API"
    });

    options.AddSecurityDefinition("Bearer", new OpenApiSecurityScheme
    {
        Name = "Authorization",
        Description = "Enter JWT Token",
        In = ParameterLocation.Header,
        Type = SecuritySchemeType.Http,
        Scheme = "Bearer",
        BearerFormat = "JWT"
    });

    options.AddSecurityRequirement(new OpenApiSecurityRequirement
    {
        {
            new OpenApiSecurityScheme
            {
                Reference = new OpenApiReference
                {
                    Type = ReferenceType.SecurityScheme,
                    Id = "Bearer"
                }
            },
            Array.Empty<string>()
        }
    });
});
```

Se configura la información básica de la API, como la versión, el título y la descripción, utilizando options.SwaggerDoc.

Se agrega una definición de seguridad para autenticación Bearer (JWT) utilizando options.AddSecurityDefinition. Esto permite que los usuarios ingresen un token JWT en Swagger para probar los endpoints que requieren autenticación. Se configura el nombre del encabezado de autorización, la descripción del token, la ubicación del token (encabezado HTTP), el tipo de esquema de autenticación (HTTP Bearer) y el formato del token (JWT).

Se agrega un requisito de seguridad utilizando options.AddSecurityRequirement. Esto indica que todos los endpoints de la API requieren autenticación Bearer (JWT). Se hace referencia a la definición de seguridad Bearer configurada anteriormente.

Al configurar Swagger de esta manera, la documentación de la API generada incluirá una sección para ingresar un token JWT Bearer, y todos los endpoints estarán marcados como protegidos por autenticación Bearer. Esto permite a los desarrolladores probar y depurar la API de manera más sencilla, ya que pueden autenticarse e interactuar con los endpoints protegidos directamente desde la interfaz de Swagger.

Para profundizar en el uso de Swagger para documentar Web APIs: [Get started with Swashbuckle and ASP.NET Core | Microsoft Learn]