Funcionamiento de JWT y Users

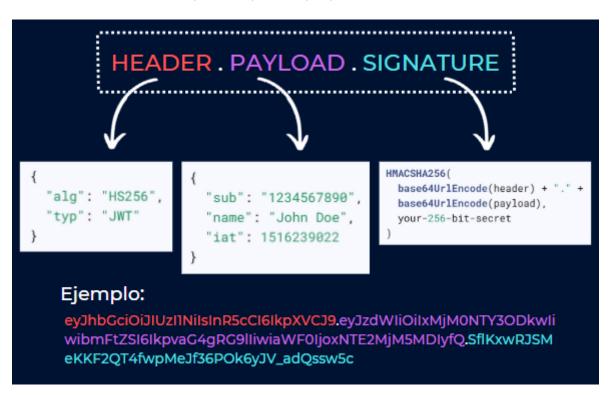
JWT

¿Qué es JWT?

JWT (por sus siglas en inglés JSON Web Token) es un estándar que define una manera segura de transmitir información entre dos partes a través de JSON. Los JWT son especialmente populares en los procesos de autentificación.

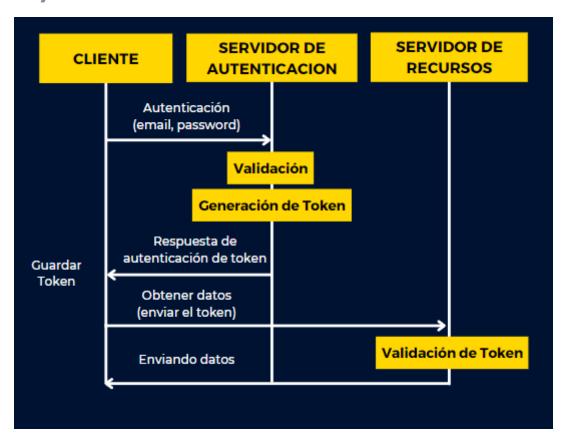
Estructura JSON Web Token

Un JWT firmado consta de 3 partes separadas por puntos.



- HEADER Consta generalmente de dos valores y proporciona información importante sobre el token. Contiene el tipo de token y el algoritmo de la firma.
- PAYLOAD Contiene la información real que se transmitirá a la aplicación. Aquí se definen algunos estándares que determinan qué datos se transmiten y cómo. La información se proporciona como pares Key/Value (Clave/Valor). Las claves se denominan claims en JWT. Hay tres tipos diferentes de claims (registrados, públicos, privados).
- SIGNATURE La firma de un JSON Web Token se crea utilizando la codificación Base64 del header y del payload, así como el método de firma o cifrado especificado.

Flujo basico de JWT



https://dev.to/gdcodev/introduccion-a-json-web-token-3mjf

Login y JWT en la API

La generación de JWT (JSON Web Tokens) en este archivo se realiza en el endpoint de inicio de sesión ("/auth/login"). Aquí están los pasos detallados:

1. Primero, se obtiene el usuario de la base de datos utilizando el correo electrónico proporcionado en la solicitud de inicio de sesión.

```
var user = userService.GetUser(request.Email);
```

2. Luego, se verifica si el usuario existe y si la contraseña proporcionada coincide con la del usuario.

```
if (user != null && request.Password == user.Password)
```

3. Si la verificación es exitosa, se procede a la generación del JWT. Primero, se crean las reclamaciones (claims) que se incluirán en el token. En este caso, se está incluyendo el correo electrónico del usuario.

```
var claims = new List<Claim>() { new(ClaimTypes.Name, request.Email), };
```

4. Luego, se crea una clave de seguridad utilizando la clave secreta proporcionada. Esta clave se utilizará para firmar el token.

5. A continuación, se crea el token JWT utilizando el emisor, el público, las reclamaciones, la fecha de vencimiento y las credenciales de firma.

6. Finalmente, se escribe el token JWT en una cadena y se devuelve como parte de la respuesta.

```
var accessToken = new JwtSecurityTokenHandler().WriteToken(jwtSecurityToken);
return Results.Ok(new { AccessToken = accessToken });
```

Si la verificación del usuario falla, se devuelve un error 400 (BadRequest).

```
return Results.BadRequest();
```

Configuración de JWT en la API

La declaración de servicios para la generación de JWT en el archivo Program.cs se realiza en varias partes:

1. Primero, se configura el servicio de autenticación para usar JWT (JSON Web Tokens) como el esquema de autenticación predeterminado. Esto se hace utilizando

```
AddAuthentication(JwtBearerDefaults.AuthenticationScheme) .
```

builder.Services.AddAuthentication(JwtBearerDefaults.AuthenticationScheme)

2. Luego, se configura el servicio JWT Bearer. Aquí es donde se establecen los parámetros de validación del token. Se valida el emisor del token (ValidateIssuer = true), se establece la clave de firma del token (IssuerSigningKey) y se definen el emisor válido (ValidIssuer) y la audiencia válida (ValidAudience).

```
.AddJwtBearer(options =>
{
    options.TokenValidationParameters = new TokenValidationParameters
    {
        ValidateIssuer = true,
        IssuerSigningKey = new SymmetricSecurityKey(Encoding.UTF8.GetBytes("aopsjfp0aoisjf[poajsf[poajsvalidIssuer = "https://www.surymartinez.com",
        ValidAudience = "Minimal APIs Client"
    };
})
```

3. Después de configurar la autenticación, se agrega el servicio de autorización con AddAuthorization().

```
builder.Services.AddAuthorization();
```

4. Finalmente, se configura Swagger para usar JWT. Se agrega una definición de seguridad para JWT y se agrega un requisito de seguridad que hace referencia a esa definición.

```
builder.Services.AddSwaggerGen(options =>
   options.AddSecurityDefinition(JwtBearerDefaults.AuthenticationScheme,
       new OpenApiSecurityScheme
        {
           Type = SecuritySchemeType.ApiKey,
           In = ParameterLocation.Header,
           Name = HeaderNames.Authorization,
           Description = "Insert the token with the 'Bearer' prefix",
       });
   options.AddSecurityRequirement(new OpenApiSecurityRequirement
    {
        {
           new OpenApiSecurityScheme
                Reference = new OpenApiReference
                    Type = ReferenceType.SecurityScheme,
                    Id = JwtBearerDefaults.AuthenticationScheme
            },
            new string[] { }
        }
   });
```

Estas configuraciones permiten que la aplicación genere y valide JWT para la autenticación de usuarios.

Users

Declaracion de la clase o record User

El UserDto es un record en C#, introducido en C# 9.0. Un record es un tipo de referencia que tiene características de inmutabilidad y comportamiento de valor. Esto significa que una vez que un record es creado, no puede ser modificado. Cualquier modificación resultará en la creación de un nuevo record.

Aquí está el UserDto:

```
public record UserDto(string Email, string Password);
```

Este record tiene dos propiedades, Email y Password, que son inmutables. Esto significa que una vez que se crea una instancia de UserDto, no puedes cambiar el Email o Password.

La principal diferencia entre un record y una clase es que un record es inmutable y tiene comportamiento de valor. Esto significa que dos records con los mismos valores serán considerados iguales. En contraste, dos instancias de una clase con los mismos valores no son iguales a menos que se sobrescriba el método Equals .

Además, los records proporcionan funcionalidades incorporadas para copiar y comparar objetos. Por ejemplo, puedes crear una copia de un record con algunas propiedades modificadas utilizando la sintaxis with .

En resumen, debes usar records cuando quieras modelos inmutables con comportamiento de valor, y clases cuando necesites objetos con estado mutable.

Servicio de usuarios

El archivo UserService.cs contiene una clase llamada UserService que se utiliza para manejar las operaciones relacionadas con los usuarios en la aplicación. Aquí está un desglose de su funcionamiento:

1. La clase UserService se inicializa con una lista de UserDto . Esta lista se utiliza como una base de datos en memoria para almacenar los usuarios.

```
public class UserService(List<UserDto> users)
{
    private readonly List<UserDto> _users = users;
```

2. La función CreateUser toma un UserDto como argumento, lo agrega a la lista de usuarios y luego lo devuelve.

```
public UserDto CreateUser(UserDto newUser)
{
    _users.Add(newUser);
    return newUser;
}
```

3. La función GetAllusers devuelve todos los usuarios en la lista.

```
public List<UserDto> GetAllUsers()
{
    return _users;
}
```

4. La función GetUser toma un userId como argumento, busca un usuario con ese ID en la lista y lo devuelve. Si no se encuentra ningún usuario, devuelve null .

```
public UserDto? GetUser(string userId)
{
    return _users.Find(u => u.Email == userId);
}
```

5. La función UpdateUserPassword toma un userId y un UserDto como argumentos. Busca un usuario con ese ID en la lista y, si lo encuentra, actualiza su contraseña con la del UserDto proporcionado y lo devuelve. Si no se encuentra ningún usuario, devuelve null .

```
public UserDto? UpdateUserPassword(string userId, UserDto updatedUser)
{
   var user = _users.Find(u => u.Email == userId);
   if (user != null)
   {
      user = updatedUser;
      return user;
   }
   return null;
}
```

En resumen, UserService es una clase simple que proporciona funcionalidades para crear, obtener y actualizar usuarios en una base de datos en memoria.

Endpoints de usuarios

Los endpoints /users en el archivo Program.cs interactúan con la clase UserService para realizar operaciones CRUD (Crear, Leer, Actualizar, Eliminar) en los usuarios.

app.MapPost("/users", (UserDto newUser) => {...}) : Este es un endpoint POST que se utiliza para crear un nuevo usuario. Toma un UserDto como cuerpo de la solicitud, lo pasa al método CreateUser de UserService , y luego devuelve el usuario creado. Si la creación es exitosa, devuelve un estado 201 (Created) con la ubicación del nuevo recurso y el recurso creado.

```
app.MapPost("/users", (UserDto newUser) =>
{
   var createdUser = userService.CreateUser(newUser);
   return Results.Created($"/users/{createdUser.Email}", createdUser);
}).WithName("CreateUser").WithOpenApi();
```

2. app.MapGet("/users", () => {...}) : Este es un endpoint GET que se utiliza para obtener todos los usuarios. Llama al método GetAllUsers de UserService y devuelve la lista de todos los usuarios.

```
app.MapGet("/users", () => userService.GetAllUsers()).WithName("GetAllUsers").WithOpenApi();
```

3. app.MapGet("/users/{userId}", (string userId) => {...}) : Este es un endpoint GET que se utiliza para obtener un usuario específico por su ID (en este caso, el correo electrónico). Toma un userId como parámetro de ruta, lo pasa al método GetUser de UserService, y luego devuelve el usuario si se encuentra. Si no se encuentra el usuario, devuelve un estado 404 (Not Found).

```
app.MapGet("/users/{userId}", (string userId) =>
{
   var user = userService.GetUser(userId);
   if (user == null)
   {
      return Results.NotFound($"User with ID {userId} not found.");
   }
   return Results.Ok(user);
});
```

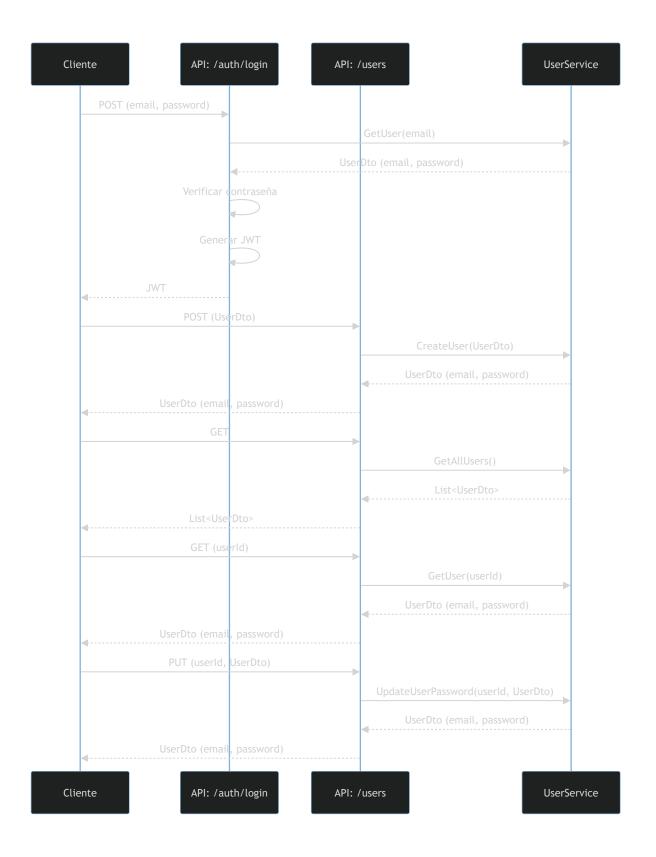
4. app.MapPut("/users/{userId}", (string userId, UserDto updatedUser) => {...}) : Este es un endpoint PUT que se utiliza para actualizar la contraseña de un usuario específico. Toma un userId como parámetro de ruta y un UserDto como cuerpo de la solicitud, los pasa al método UpdateUserPassword de UserService , y luego devuelve el usuario actualizado si se encuentra. Si no se encuentra el usuario, devuelve un estado 404 (Not Found).

```
app.MapPut("/users/{userId}", (string userId, UserDto updatedUser) =>
{
   var user = userService.UpdateUserPassword(userId, updatedUser);
```

```
if (user == null)
{
    return Results.NotFound($"User with ID {userId} not found.");
}
return Results.Ok(user);
}).WithName("UpdateUserPassword").WithOpenApi();
```

En resumen, estos endpoints proporcionan una interfaz HTTP para interactuar con la clase UserService , permitiendo a los clientes de la API realizar operaciones CRUD en los usuarios.

Funcionamiento de los endpoints de la API



WebApplication +Services: IServiceProvider +CreateBuilder(args: string[]):: WebApplicationBuilder +Build():: WebApplication +UseSwagger():: WebApplication +UseSwaggerUI():: WebApplication +UseHttpsRedirection():: WebApplication +UseAuthentication():: WebApplication +UseAuthorization():: WebApplication +MapGet(pattern: string, handler: Delegate) :: WebApplication +MapPost(pattern: string, handler: Delegate) : : WebApplication +MapPut(pattern: string, handler: Delegate):: WebApplication +Run():: void Creates WebApplicationBuilder +Services: IServiceCollection Uses

+Build():: WebApplication Adds to services

UserService

+CreateUser(newUser: UserDto):: UserDto

+GetAllUsers()::List

+GetUser(userId: string) : : UserDto?

+UpdateUserPassword(userId: string, updatedUser: UserDto):: UserDto?

