

Autenticación con JWT en .Net

Modificación de strings

Codificación

Es una técnica en la que los datos se transforman de un formato a otro, para que puedan ser entendidos y utilizados por diferentes sistemas. No se utiliza ninguna “clave” en la codificación.

El mismo algoritmo se utiliza para decodificar los datos que se utilizaron para codificarlos en primer lugar. Por esta razón, es muy fácil para un atacante decodificar los datos si tiene la información codificada. El ejemplo de tales algoritmos son ASCII, Unicode, Base64, etc.

Para entender esto, tomemos un ejemplo de la técnica de codificación BASE64. Durante los primeros años de los correos electrónicos, los datos se basan principalmente en texto. Entonces, para transmitirlos a través de Internet, los datos de texto se convirtieron a formato binario. Y, al final del receptor, estos datos binarios se volvieron a convertir nuevamente a formato de texto.

Pero a medida que pasaba el tiempo, surgió la necesidad de enviar archivos adjuntos y multimedia junto con el correo electrónico basado en texto. La probabilidad de que los datos binarios de estos archivos adjuntos se dañaran era alta en su forma original. Esto se debió principalmente a que un binario sin formato contiene algunos caracteres que en algunos lenguajes se tratan de manera diferente.

A modo de ejemplo presentamos una pequeña tabla de codificación de caracteres del alfabeto a BASE64

Value	Char	Value	Char	Value	Char	Value	Char
0	A	16	Q	32	g	48	w
1	B	17	R	33	h	49	x
2	C	18	S	34	i	50	y
3	D	19	T	35	j	51	z
4	E	20	U	36	k	52	0
5	F	21	V	37	l	53	1
6	G	22	W	38	m	54	2
7	H	23	X	39	n	55	3
8	I	24	Y	40	o	56	4
9	J	25	Z	41	p	57	5
10	K	26	a	42	q	58	6
11	L	27	b	43	r	59	7
12	M	28	c	44	s	60	8
13	N	29	d	45	t	61	9
14	O	30	e	46	u	62	+
15	P	31	f	47	v	63	/

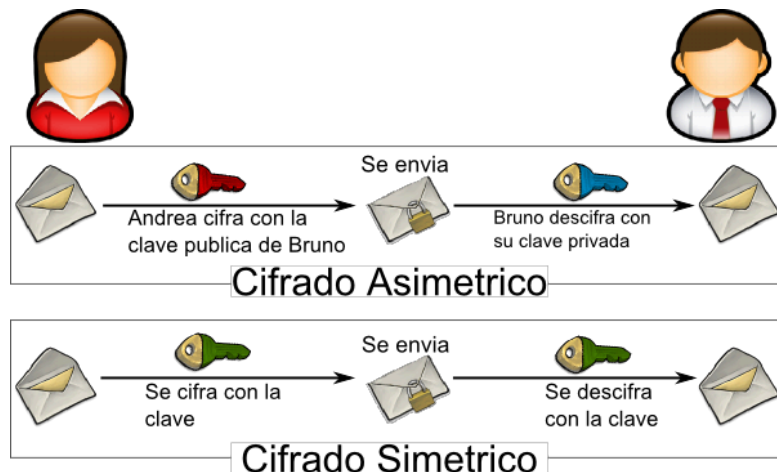
Encriptacion

No se puede imaginar Internet sin cifrado. Internet habría sido un lugar mucho menos seguro sin él. Los datos en Internet se mantienen confidenciales y seguros mediante cifrado. El cifrado hace que los datos sean ilegibles y difíciles de decodificar para un atacante y evita que sean robados.

El cifrado utiliza “claves criptográficas”. Con la clave, los datos se cifran en el extremo del remitente. Y, con la misma clave o una clave diferente, los datos se descifran en el extremo del receptor. Según el tipo de clave que se utiliza para cifrar/descifrar la información, el cifrado se clasifica en dos categorías:

- cifrado de clave simétrica
- cifrado de clave asimétrica

Analicemos estos dos tipos de técnicas de cifrado:



Clave simétrica

El enfoque de la criptografía de clave simétrica es sencillo. Antes de enviar los datos al receptor, el remitente utiliza una clave privada para cifrar los datos. Esta clave privada es conocida solo por el remitente y el receptor.

Entonces, una vez que el receptor obtiene los datos cifrados, él o ella usa la misma clave privada del remitente para descifrarlos. Dado que el remitente y el receptor utilizan la misma clave, esto se conoce como criptografía de “clave simétrica”.

Sí, hay muchos beneficios de usar esta técnica para el cifrado. Por ejemplo, la simplicidad de este sistema proporciona una ventaja logística ya que requiere menos potencia de cálculo.

Además, el simple hecho de aumentar la longitud de la clave puede ayudar fácilmente a escalar el nivel de seguridad del sistema. Pero el problema de utilizar esta técnica ya es visible para nosotros. ¿Cómo va a saber el receptor cuál es la clave secreta con la que se cifraron los datos en primer lugar? Para que él conozca la clave, el remitente también debe enviar la clave.

El remitente también tiene que transmitir la clave al receptor, y si se hace a través de una conexión no segura. Es decir, existe una alta probabilidad de que la clave pueda ser interceptada por cualquier atacante.

Clave asimétrica

Para contrarrestar los problemas de utilizar la técnica de criptografía de clave simétrica, se utiliza la técnica de criptografía asimétrica. Es una técnica relativamente nueva en comparación con la anterior.

Como puedes adivinar por el propio nombre, esta técnica implica dos claves diferentes. Se utiliza una clave para cifrar los datos, que se conoce como clave pública, y es conocida prácticamente por todos en Internet. La otra clave se utiliza para descifrar los datos, que se conoce como clave privada, y solo la conoce el receptor y debe mantenerse discreta. Por lo tanto, el uso de dos claves distintas hace que el sistema sea más seguro y se vuelve demasiado difícil para un atacante descifrarlo.

Las claves públicas y privadas son “llaves” reales para una cerradura. Estos son básicamente dos números primos muy grandes que están relacionados matemáticamente entre sí. Es decir, todo lo que esté cifrado por la clave pública, solo puede ser descifrado por la clave privada relacionada.

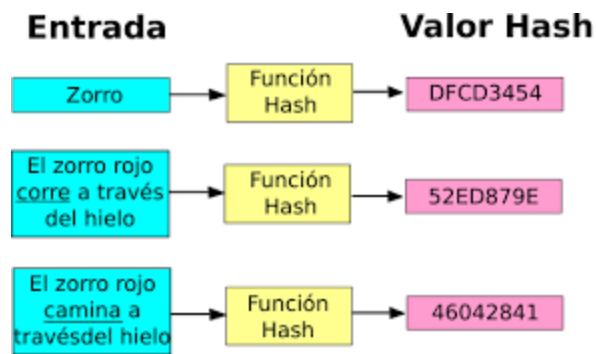
No se puede simplemente adivinar la clave privada basándose en conocer la clave pública.

Algunos algoritmos conocidos para la criptografía de clave asimétrica son: RSA, DSS (estándar de firma digital), criptografía de curva elíptica, etc.

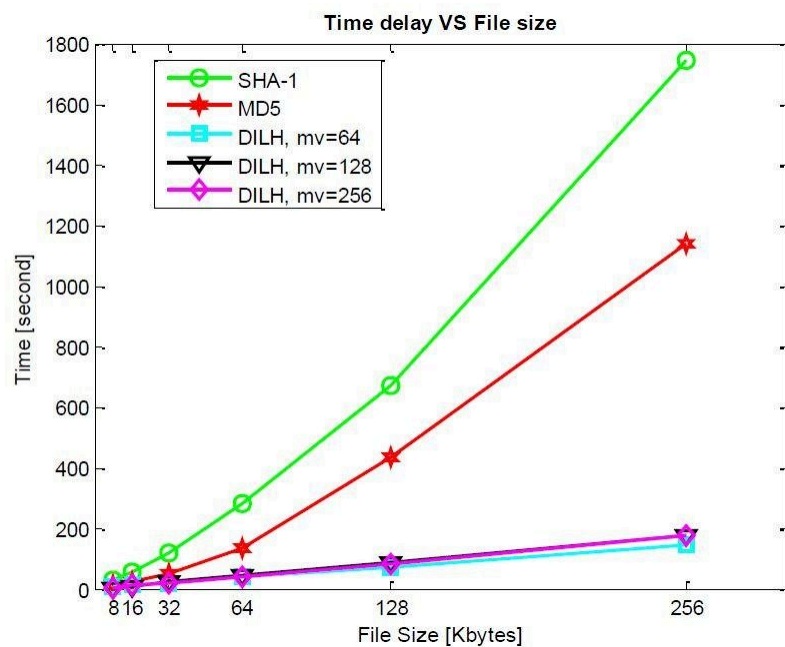
Hash

Un Hash es básicamente una cadena que se genera a partir de la cadena de entrada pasándola a través de un algoritmo Hash. Esta cadena hash es siempre de una longitud fija sin importar el tamaño de la cadena de entrada. El hash también se puede considerar como “cifrado unidireccional”, es decir, los datos una vez procesados no se pueden revertir a su forma original.

Con eso, significa que se asegura de que incluso si se cambia una sola cosa, tu puedes saber que se ha cambiado. El hash protege tus datos contra posibles alteraciones para que tus datos no se modifiquen ni un poco.

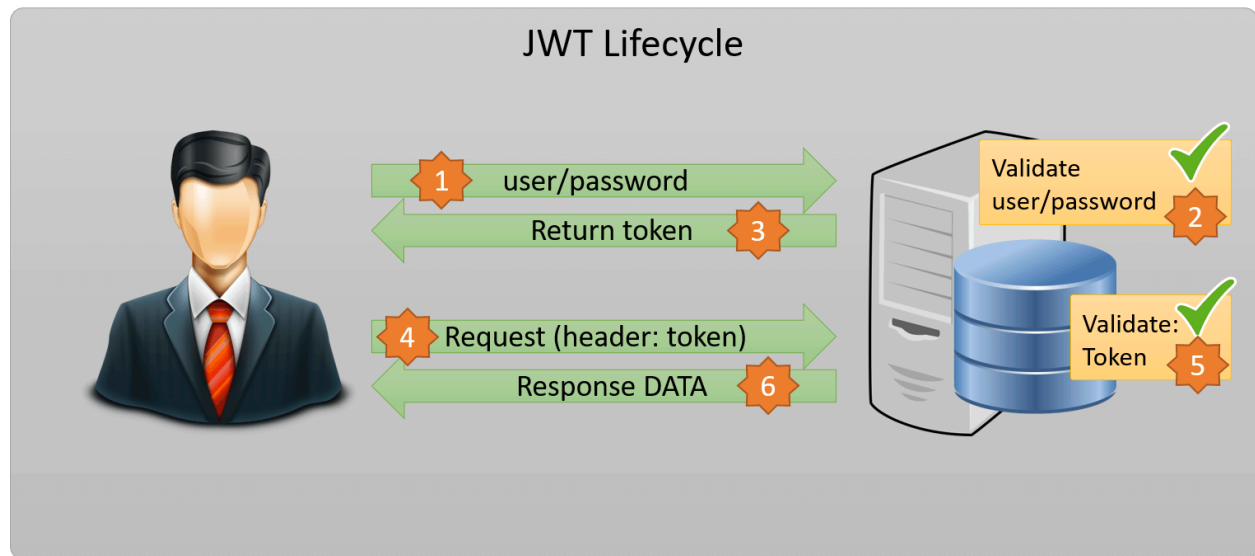


Existe una amplia variedad de algoritmos de hashing, a continuación un gráfico comparativo de algoritmos y el tiempo de hash archivos en KB.



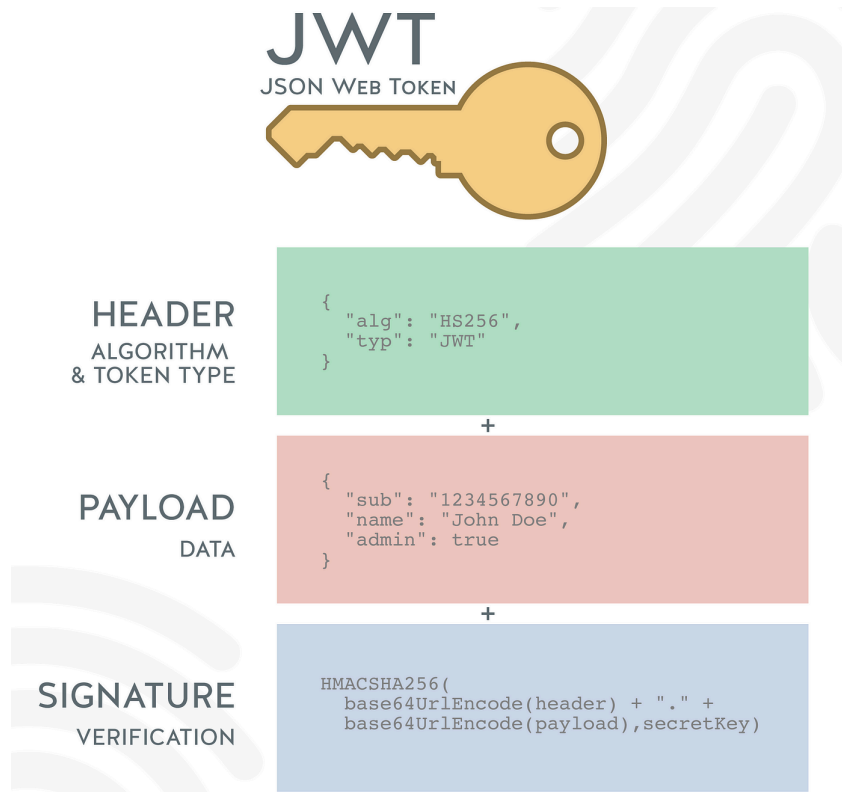
Autenticación con JWT

Proceso de autenticación



1. El cliente solicita mediante una request a un endpoint de autenticación, el token con el cual va a poder acceder a los servicios de la api. Para esto manda en dicha request las credenciales de autenticación que, en muchos casos, es el usuario y la contraseña.
2. La API verifica que el usuario exista en la base de datos y además verifica si realmente es quien dice ser a través de la confirmación de que la contraseña es la correspondiente para el usuario.
3. La API retorna, como respuesta de la request del paso 1, el token de autenticación que el cliente va a usar a futuro si corresponde. En caso de no reconocer el usuario o de que la validación de las credenciales haya fallado entonces retorna una response con status code 401.
4. Desde ese momento, cada vez que el cliente quiera usar un endpoint de la api va a mandar en la request en un header el token que obtuvo en el paso 3 de la API. Generalmente anteponiendo "Bearer " Ej: Bearer tokenquerecibidelaapienelpaso3
5. La api verifica que el token lo haya generado ella y que nadie lo haya alterado. Este proceso de verificación lo veremos mejor en el último título de esta sección.
6. La api procede a procesar la request del cliente normalmente y a responderle lo que corresponda

Estructura del JWT



Un JSON web token es un estándar de código abierto que se utiliza para transmitir información de manera segura entre dos partes en formato JSON. Está diseñado para ser un mecanismo de autenticación y autorización en aplicaciones web y servicios API. Los JWT son especialmente populares en el contexto de aplicaciones basadas en RESTful API y aplicaciones de una sola página (SPA).

Un JWT consta de tres partes: la cabecera (header), el payload y la firma. La cabecera contiene información sobre el algoritmo de cifrado utilizado y el tipo de token. El payload es el contenido propiamente dicho del token y contiene la información que se quiere transmitir.

Un JWT es una forma compacta de intercambiar información en la web. Tiene tres partes bien definidas dentro de la estructura de un JSON web token: la cabecera, el payload y la firma del token.

La cabecera del JWT

La cabecera dentro de la estructura de un JSON web token es la que indica qué formato criptográfico está utilizando el token. Aunque a nivel de usuario no nos interesa mucho conocer los detalles, es importante saber que aquí se especifica el algoritmo que se está usando para la firma. Así, se puede tener una mayor seguridad y confidencialidad.

El payload del JWT

En el payload, como parte de la estructura de un JSON web token, es donde se encuentra la información propiamente dicha. Esta parte está codificada en Base64, lo que permite que cualquiera pueda ver su contenido. Esto no es un problema, porque aunque pueda verse, no se puede modificar.

En término de estructura el payload es un diccionario en donde cada par clave valor lo denominamos una "claim". Como es un diccionario no puede haber más de una clave con el mismo identificador. Existen claims obligatorias y también el servidor puede agregar las que quiera a la hora de generar el JWT. Algunas de las obligatorias más comunes son: exp (fecha de expiración) iat (fecha de creación)

Si copias y pegas el contenido de un token en algún decodificador Base64, verás que se revelan los datos que contiene. Es como mirar a través de una urna de cristal: todo está a la vista, pero no puedes alterarlo. Ya más adelante veremos porque esto es cierto

La firma del token

La firma (signature) es la parte de la estructura de un JSON web token que garantiza la autenticidad del JWT. Aquí es donde entra en juego la clave privada. La firma se genera usando el contenido de la cabecera y el payload, además de una clave secreta que solo conoce el servidor. De esta manera, al recibir el JWT, el servidor puede verificar si ha sido alterado en el camino.

Signature = **Hash**(**Base64**(**Header**)+**Base64**(**Payload**)+**Secret**)

Codificación para transferencia en la web

La estructura de un JSON web token es bastante sencilla. Se compone de tres partes que van separadas por un punto. Cada una de estas partes está codificada en Base64. Por tanto, un JWT tendría esta apariencia:

//Estructura de un JSON web token

[Base64(header)].[Base64(payload)].[firma]

Lo importante de esto es que CUALQUIERA puede ver el contenido de cada parte utilizando alguna herramienta de de-codificación Base64.

Verificación de la autenticidad del JWT

Basado en lo que venimos explicando existe un riesgo de que alguien pueda, a partir de un token válido, modificar el mismo para que el user al que pertenezca sea de otro distinto al del propietario original y de esta manera autenticarse como otra persona. También existe el riesgo

de que alguien genere token falsos, idénticos en estructura a los válidos, para usarlos a la hora de autenticarse. ¿Cómo hace el servidor para evitar este riesgo?

Empecemos por clarificar lo que la API tiene que asegurarse cuando recibe una request con el header con el jwt adentro:

1. Que dicho JWT fue generado por la API
2. Que nadie alteró la misma.
3. Que el token no esté expirado.

El punto 3 es sencillo: el jwt tiene dentro del payload una claim que indica la fecha de expiración del token o ,lo que es lo mismo, el plazo máximo hasta donde un token es válido

El paso 1 y 2 se verifican de la siguiente manera: la API (en el servidor) al recibir una request, una de las primeras cosas que hace es extraer el jwt de un header de la request (generalmente se llama Authorization) a partir de dicho jwt extrae el header y el payload del jwt ignorando la tercera parte que es la signature (por ahora), A partir de esas dos partes procede a armar la signature que le corresponde a la jwt por su cuenta siguiendo la formula expresada con anterioridad:

JWT extraido de la request = HeaderA + PayloadA + SignatureA

SignatureB = Hash(Base64(HeaderA)+Base64(PayloadA)+Secret)

La API compara la signature que acaba de generar (SignatureB) con la que vino originalmente SignatureA

SignatureB == SignatureA ?

Si son iguales entonces el 1 y 2 están verificados y el JWT es válido. El proceso de autenticación termina con éxito. Si no son iguales entonces o bien el jwt no fue generado por la API o bien alguien modifico un jwt generado por la API, en cualquier caso el proceso de autenticación falla y la API genera una response con status code 401

Como el secreto es algo que el servidor solo conoce y a partir de la signature nadie puede deducir el Secret porque los hash son irreversibles entonces no hay forma de que alguien puede generar una Signature válida si se crean o modifican un payload y/o un header porque les falta un parte para poder generarla, el Secret.

Implementación en una API en .net

En esta sección vamos a dar los pasos mínimos e indispensables para implementar JWT, hay muchas variaciones del sistema de autenticación tanto como en el uso como en la implementación así como también se pueden tomar decisiones de diseño y arquitectura para implementar separación de responsabilidades que difieran un poco de lo que se les presentara en este documento

El propósito de esta sección es enfocarse exclusivamente en la autenticación para no extenderla en complejidad y facilitar la lectura por lo que no va a haber una descripción clara de la arquitectura de la API en donde mostraremos la implementación, solo habrá lineamientos generales. Cualquier duda puntual sobre este tema acudir a los apuntes de clase y/o el proyecto de referencia

Endpoint de autenticación

La responsabilidad del endpoint de autenticación es la especificada en los puntos 1,2 y 3 del título “Proceso de autenticación” de este documento.

Se debe encargarse de recibir las credenciales (normalmente puede ser usuario y contraseña pero no necesariamente) lo ideal es hacerlo a través de un dto y no de parámetros sueltos. Este dto se recibe por el body.

El endpoint es un POST porque se encarga de acreditar y registrar un acceso así que aunque no haga no necesariamente registre nada en la base de datos, desde el punto de vista teórico cumple con las características para ser un POST. Por otro lado un GET porque no se puede enviar nada por body en las request con ese verbo http

También se debe encargarse que las credenciales del usuario sean válidas, es decir que el usuario exista y que su contraseña le corresponda. Por último si, si las credenciales son válidas, debe generar un JWT apropiado para el usuario y devolverlo en una response. Debido a que el endpoint tiene que hacer tantas cosas es importante que lo haga a través de un servicio. Como es el caso del “Consulta Alumnos” que es el repositorio de referencia para su trabajo.

```
8 [Route("api/authentication")]
9 [ApiController]
10 public class AuthenticationController : ControllerBase
11 {
12     private readonly ICustomAuthenticationService _customAuthenticationService;
13
14     public AuthenticationController(IConfiguration config, ICustomAuthenticationService autenticacionService)
15     {
16         _customAuthenticationService = autenticacionService;
17     }
18
19     [HttpPost("authenticate")]
20     public ActionResult<string> Autenticar(AuthenticationRequest authenticationRequest)
21     {
22         string token = _customAuthenticationService.Autenticar(authenticationRequest);
23
24         return Ok(token);
25     }
26 }
27
```

Para simplificar la explicación de la generación del token vamos a presentar dicho proceso dentro del código del endpoint. Esto tiene solo un propósito pedagógico y no algo que tiene que usarse de referencia. Sin embargo la metodología y el resultado es muy similar al proyecto de referencia por lo que será una guía para entenderlo más.

```
25 [HttpPost]
26 0 references
27 public IActionResult Authenticate([FromBody] CredentialsDtoRequest credentials)
28 {
29     User? user = _userService.Authenticate(credentials.Username,
30         credentials.Password);
31
32     if (user is not null)
33     {
34         var securityPassword = new SymmetricSecurityKey(
35             Encoding.ASCII.GetBytes(_config["Authentication:SecretForKey"]));
36
37         var signature = new SigningCredentials(securityPassword, SecurityAlgorithms.HmacSha256);
38
39         var claimsForToken = new List<Claim>();
40         claimsForToken.Add(new Claim("sub", user.Id.ToString()));
41         claimsForToken.Add(new Claim("role", user.Role));
42
43         var jwtSecurityToken = new JwtSecurityToken(
44             _config["Authentication:Issuer"],
45             _config["Authentication:Audience"],
46             claimsForToken,
47             DateTime.UtcNow,
48             DateTime.UtcNow.AddHours(1),
49             signature);
50
51         var tokenToReturn = new JwtSecurityTokenHandler()
52             .WriteToken(jwtSecurityToken);
53
54         return Ok(tokenToReturn);
55     }
56     return Unauthorized();
57 }
```

Como podrán ver desde la línea 28 a la línea 30 se encuentra el proceso de verificación de las credenciales del usuario, si las mismas son válidas entonces el método del servicio devuelve un objeto Usuario el cual vamos a usar para armar el payload del token.

A partir de la línea 33 hasta la línea 51 (en donde retornamos el token) se encuentra el proceso de generación del JWT.

En la línea 33 y 34 encriptamos el secret que pusimos del appsetting.json, la variable `_config` es una variable privada parte de la inyección de la interfaz de .Net la cual nos permite acceder a él appsetting.json a través de una navegabilidad tipo diccionario.

```
appsettings.json* X
Schema: https://json.schemastore.org/appsettings.json
1  {
2    "Logging": {
3      "LogLevel": {
4        "Default": "Information",
5        "Microsoft.AspNetCore": "Warning"
6      }
7    },
8    "AllowedHosts": "*",
9    "Authentication": {
10     "SecretForKey": "thisisthesecretforgeneratingakey(mustbeatleast32bitlong)",
11     "Issuer": "https://localhost:7169",
12     "Audience": "consultaalumnosapi"
13   }
14 }
```

En la línea 36 dejamos casi lista la signature con el secreto y el algoritmo de hashing indicado




Desde la línea 38 a la línea 40 agregamos claims a una lista la cual va a ser añadida al payload a las claims por defecto.

Desde la línea 42 a la 48 instanciamos un objeto que representa nuestro JWT para lo cual usamos un constructor que recibe una serie de parámetros:

- Issuer: es quien creó el token, el identificado de la api que generó el JWT. Se coloca como claim en el payload
- Audience: a quién va dirigido dicho token. Se coloca como claim en el payload
- Claims personalizadas: una lista de claims que se van a agregar al payload del JWT.
- Fecha de creación: en UTC que es GMT0 para poder hablar un lenguaje común
- Fecha de expiración: un periodo a partir de la fecha de creación en donde el JWT va a ser válido. Este valor es una claim del payload
- Signature: es la firma del JWT que pre generamos con anterioridad que ahora él constructor la va usar para armar la firma definitiva del JWT usando el payload entero del JWT con todos los valores que le pasamos

Configuración de paquetes

Para poder tener disponibles todas las clases usadas a la hora de crear el JWT y garantizar su verificación posteriormente hacen falta una serie de paquetes a instalar los cuales son los siguientes (la versión que se muestra instalada es la última compatible con el runtime de net instalada en mi notebook el cual es Net 6, ustedes instalarlos en las últimas versiones disponibles que tengan en base a el runtime de Net que tengan instalado)

 Microsoft.AspNetCore.Authentication.JwtBearer by Microsoft	6.0.31
ASP.NET Core middleware that enables an application to receive an OpenID Connect bearer token.	8.0.6
 Microsoft.IdentityModel.Tokens by Microsoft	6.35.0
Includes types that provide support for SecurityTokens, Cryptographic operations: Signing, Verifying Signatures, Encryption.	7.6.1
 System.IdentityModel.Tokens.Jwt by Microsoft	6.35.0
Includes types that provide support for creating, serializing and validating JSON Web Tokens.	7.6.1

Para poder usar la autenticación en Swagger va a hacer falta la siguiente configuración en el program.cs reemplazando a el `builder.Services.AddSwaggerGen()` que tengan por defecto.

Este cambio genera un botón verde en la parte superior de la UI de swagger en donde pueden colocar el token que recuperan del endpoint de autenticación. Lo que hace swagger es añadir un header de nombre `Authorize` con el siguiente valor `Bearer` `<elstringquehayanpuestoenelbotonauthorizedeswagger>`. Esto permite que puedan usar el swagger para seguir probando los endpoints de la api si es que implementan autenticación en ellos.

Es una herramienta sencilla (lo único que hace es agregar un header) pero necesaria si usan swagger, si usan postman pueden agregar el header manualmente o indicar el mecanismos de autenticación de la colección y pasarle el token. Para eso no necesitan ninguna configuración en .Net. En swagger en cambio es necesario.

```
14 builder.Services.AddSwaggerGen(setupAction =>
15 {
16     setupAction.AddSecurityDefinition("ApiBearerAuth", new OpenApiSecurityScheme()
17     {
18         Type = SecuritySchemeType.Http,
19         Scheme = "Bearer",
20         Description = "Acá pegar el token generado al loguearse."
21     });
22
23     setupAction.AddSecurityRequirement(new OpenApiSecurityRequirement
24     {
25         {
26             new OpenApiSecurityScheme
27             {
28                 Reference = new OpenApiReference
29                 {
30                     Type = ReferenceType.SecurityScheme,
31                     Id = "ApiBearerAuth" }
32             }, new List<string>() }
33     });
34 }
```

Verificación de autenticación

La verificación de que la request viene con un JWT válido es un proceso que ya fue explicado en la sección "Verificación de la autenticidad del JWT". Aquí nos centraremos en explicar cómo se implementa eso en .Net .

Es necesario especificarle a .Net que es lo que tiene que verificar. Esto lo haremos a través del Program.cs con la siguiente instrucción. De la línea 41 a la 43 explicamos qué chequeos tiene que hacer al JWT proveniente de la request, de la línea 44 a la 46 especificamos , de esos

chequeos enunciados, contra que tenemos que comparar para que el chequeo se valide, es decir contra que valor tenemos que contrastar lo que nos venga del JWT

```
36 builder.Services.AddAuthentication("Bearer")
37     .AddJwtBearer(options =>
38     {
39         options.TokenValidationParameters = new()
40         {
41             ValidateIssuer = true,
42             ValidateAudience = true,
43             ValidateIssuerSigningKey = true,
44             ValidIssuer = builder.Configuration["Authentication:Issuer"],
45             ValidAudience = builder.Configuration["Authentication:Audience"],
46             IssuerSigningKey = new SymmetricSecurityKey(Encoding.ASCII.GetBytes(builder.Configuration["Authentication:SecretForKey"]))
47         };
48     });
49
50
```

Otro aspecto importante que es necesario para que la autenticación ande es que tenemos que agregarla en la pipeline de la API, la pipeline es el conjunto de llamadas a diferentes procesos a los cuales todas las request, que arriben a nuestra API, se van a someter. Típicamente se encuentran en las últimas líneas del Program.cs justo antes de la línea `app.Run()` que es donde se lanza la API. Tenemos que agregar el proceso de validación del JWT a nuestra pipeline de tal manera que todas las request (que así lo requieran) pasen por este proceso.

```
63 app.UseHttpsRedirection();
64
65 app.UseAuthentication();
66
67 app.UseAuthorization();
68
69 app.MapControllers();
70
71 app.Run();
```

Para que el chequeo de autenticación se haga efectivo tenemos que explicitar en los lugares que así lo deseamos para que se aplique con el decorador `[Authorize]`, este decorador puede ser aplicado a todo el controlador poniéndolo justo de la definición de la clase controller o bien a cada endpoint en particular.

La presencia de este decorador implica que si la request no tiene un JWT válido entonces la API va a devolver un 401 sin siquiera ejecutar la primera línea de código del endpoint. Ningún chequeo es necesario para realizar la autenticación en el endpoint

```
[Route("api/[controller]")]
[ApiController]
[Authorize]
// reference
public class UserController : ControllerBase
{
    // ...
}
```

Uso del JWT

El JWT y sus datos, especialmente el payload se encuentra a disposición dentro del endpoint, nosotros podemos extraer cualquier claim que nos pueda ser de utilidad.

Hay muchas maneras de hacerlo y difiere para diferentes claims. La que proponemos acá no es la mas sencilla o decorosa pero tiene fines pedagógicos, como verán a través del objeto User (que es un objeto que nos provee ASP NET) podemos acceder a las claims y a través de LINQ podemos acceder a aquellas claims que nos interesa y extraer el valor.

Hay algunas claims comunes que tiene identificadores constantes que pueden ser utilizadas lo cual permite encontrarlas más rápidamente como es el caso del identificador de usuario (de clave “sub”) o del rol (de clave “role”). En el ejemplo (extraído del repositorio de referencia de la cátedra) pueden observar cómo extraemos los valores de dichas claims

```
[HttpGet("pendingquestions")]
0 references | efalabrini, 145 days ago | 1 author, 4 changes
public ActionResult<ICollection<QuestionDto>> GetPendingQuestions(bool withResponses = false)
{
    int userId = int.Parse(User.Claims.FirstOrDefault(c => c.Type == ClaimTypes.NameIdentifier)?.Value ?? "");
    var userRole = User.Claims.FirstOrDefault(c => c.Type == ClaimTypes.Role)?.Value;

    if (userRole != typeof(Professor).Name)
        return Forbid();

    return _professorService.GetPendingQuestions(userId, withResponses).ToList();
}
```