

UD7 Servicios web.

7.4 Seguridad en servicios web.

En la parte final de la unidad hablaremos acerca de la seguridad en los servicios web y completaremos nuestro ejemplo añadiendo autenticación a nuestra API de lista de tareas.

Ahora que somos capaces de crear una API y hacerla pública para ser utilizada, es momento de preocuparnos sobre las cuestiones relativas a la seguridad. Muchos de los servicios disponibles en internet son ofrecidos únicamente a usuarios registrados, e incluso algunos de ellos ofrecen diferentes niveles de servicio en función de si se trata de un usuario normal o uno de pago, por ejemplo.

Para asegurar nuestros servicios web utilizaremos los mismos conceptos y técnicas empleados en nuestra aplicación web, ya que de hecho un servicio web es básicamente una aplicación web sin presentación. Para refrescar nuestros conocimientos recordaremos que la seguridad de nuestra aplicación descansará en dos conceptos fundamentales: autenticación y autorización.

La autenticación es el acto por el que podemos confirmar que el usuario es quien dice ser, verificando su identidad mediante el conocimiento de un secreto, que usualmente será su usuario y contraseña. Autorización hace referencia al hecho de permitir a los usuarios autenticados solamente a las acciones a las que tiene derecho según sus privilegios.

Para la autenticación de nuestros servicios, utilizaremos una técnica conocida como autorización basada en tokens, que es la técnica que se usa habitualmente en los servicios web. Consiste en la generación de un token de seguridad que será devuelto al cliente cuando este se loguea correctamente. Consecuentemente, todas las peticiones del cliente deberán enviar dicho token junto con la petición, lo que permitirá identificarse y tener acceso a los servicios autorizados. Se trata de una técnica segura y fácil de utilizar.

Ejemplo: Añadir soporte para autenticación y autorización.

Antes de comenzar vamos a llevar a cabo un paso previo que, aunque en realidad no es necesario, si será muy conveniente. Hasta ahora, nuestro ejemplo utilizaba un almacén de datos en memoria por simplicidad. El problema es que dicho almacén, al ser en memoria, se reinicia cada vez que lanzamos los servicios. Si vamos a registrar usuarios para podernos loguear y usar los servicios, estos usuarios desaparecerían cada vez que hagamos un cambio en el código del proyecto, obligándonos a registrarnos de nuevo cada vez que probemos los servicios. Por esta razón, vamos a cambiar primero nuestro almacén de datos a una base de datos SQL Server LocalDB, de modo que los datos persistan.

Comenzamos añadiendo la cadena de conexión a nuestro fichero *appsettings.json*:

```
{
  "Logging": {
    "LogLevel": {
      "Default": "Warning"
    }
  },
  "AllowedHosts": "*",
  "ConnectionStrings": {
```

```
"TodoContext":
{
  "Server=(localdb)\\mssqllocaldb;Database=TodoAPI;Trusted_Connection=True;MultipleActiveResultSets=true"
}
```

A continuación, en el método *ConfigureServices()* de la clase *Startup*, cambiamos el tipo de almacén de datos por una base de datos SQL:

```
services.AddDbContext<TodoContext>(options =>
    options.UseSqlServer(Configuration.GetConnectionString("TodoContext")));
```

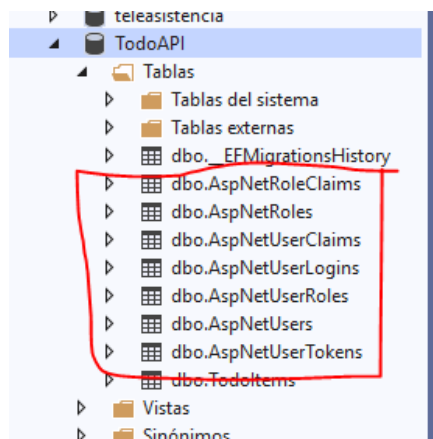
Hora que tenemos nuestra cadena de conexión y hemos definido el almacén de datos, crearemos una migración inicial y la base de datos a partir de esa migración:

```
add-migration InitialCreate
update-database
```

1. Una vez que nuestro nuevo almacén de datos está creado vamos a añadir soporte para autenticación y autorización mediante Identity. Para ello cambiamos nuestro contexto de datos para que herede de un contexto Identity.:

```
public class TodoContext : IdentityDbContext
```

Una vez hecho esto, creamos una nueva migración y actualizamos la base de datos de nuevo. Podemos ver que se han añadido las tablas que utiliza Identity para la gestión de usuarios. En nuestro caso no necesitamos ninguna característica especial en cuanto al usuario, por lo que podemos utilizar el usuario por defecto de Identity.



2. A continuación, habilitamos el uso de Identity para nuestros servicios, a continuación del código en el que definíamos en contexto de datos en *ConfigureServices()*:

```
services.AddIdentity<IdentityUser, IdentityRole>()
    .AddEntityFrameworkStores<TodoContext>()
    .AddDefaultTokenProviders();}
```

3. Indicamos a nuestra aplicación que haremos uso de la autenticación, añadiendo la siguiente línea en *Configure()*:

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

Autenticación mediante tokens JWT (JSON Web Token).

En este momento, nuestra aplicación tiene soporte completo para el uso de autenticación, con un almacén de datos capaz de almacenar nuestros usuarios, pero todavía nos queda por definir la técnica de autenticación que vamos a utilizar.

Uno de los mecanismos de autenticación más utilizados consiste en utilizar un sistema basado en tokens. Con esta técnica, cada vez que el usuario se loguea, un token es generado y se envía de vuelta como respuesta. Este token se utilizará consecuentemente por el servidor para validar la identidad del usuario. El cliente, una vez recibido el token, deberá enviarlo en la cabecera de cada petición, de modo que el servidor podrá obtener los detalles del usuario a partir del token. Algunos de los beneficios de este sistema son los siguientes:

- **Seguridad:** dado que el token está firmado y dependiendo de la configuración también cifrado, el servidor puede saber que los datos son válidos.
- **Stateless:** el servidor no debe almacenar el estado de la sesión, como en los mecanismos utilizados en las aplicaciones web tradicionales. Por lo tanto, este sistema se adapta mejor a los servicios web RESTful.
- **Cross-origin:** puesto que el token es una simple cadena de texto, puede ser enviado y utilizado por cualquier tipo de cliente, incluso localizado en un dominio diferente.
- **Rendimiento:** al contener el token la información del usuario, el servidor puede extraer la información del mismo y evitar accesos a la base de datos.

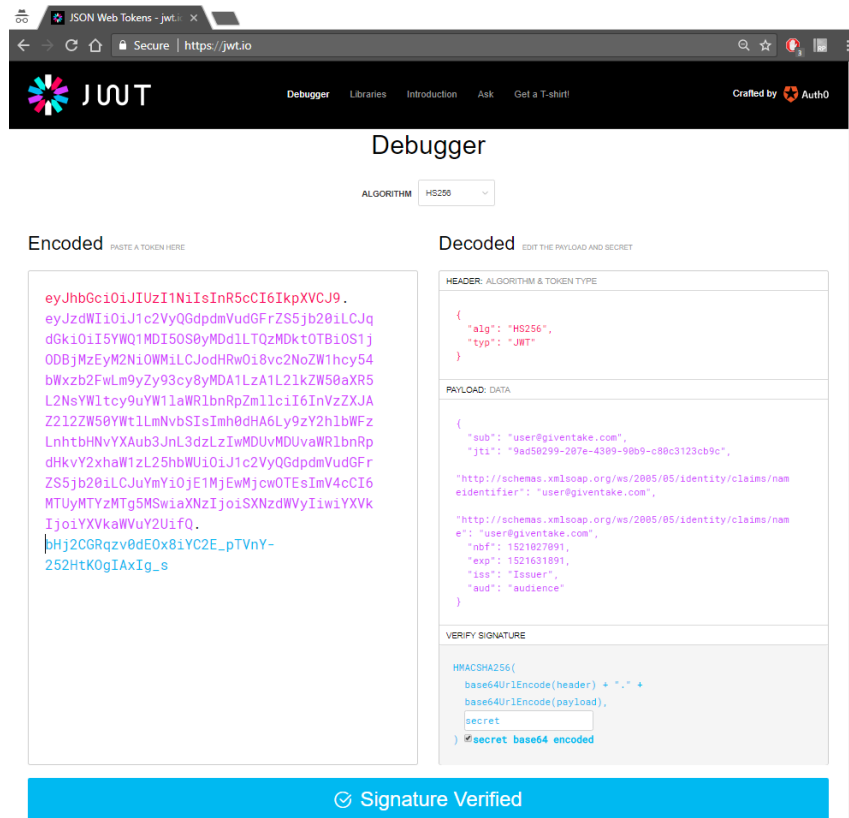
El estándar más empleado para la autenticación basada en tokens es JWT: <https://jwt.io>

Los tokens JWT están compuestos de tres partes, codificadas en Base64 y separadas por un punto.

```
{Header}.{Payload}.{Signature}
```

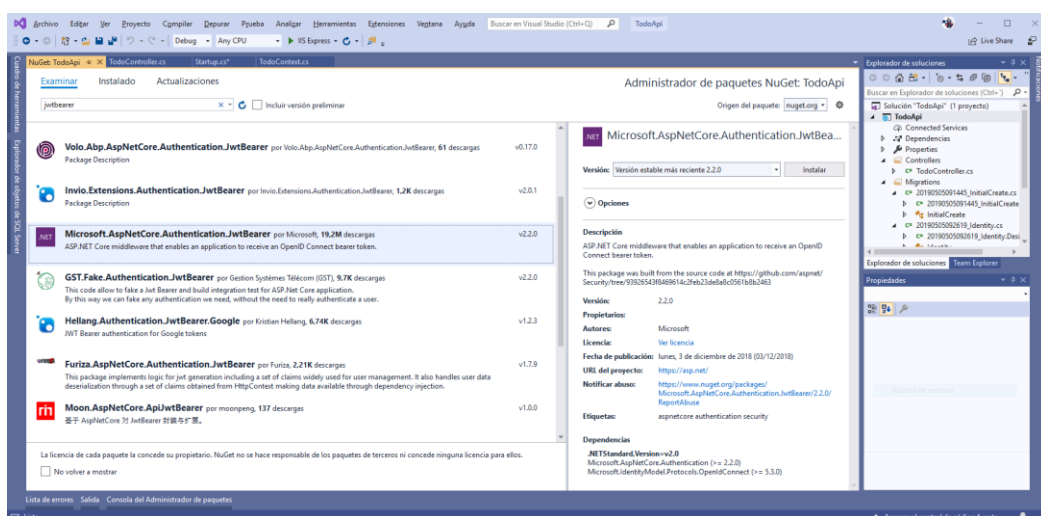
- **Header:** almacena el tipo de token y el algoritmo de hash
- **Payload:** contiene la información encapsulada del usuario, el tiempo de expiración del token, etc..
- **Signature:** un valor criptográfico calculado a partir del header y el payload que se obtiene creando un hash con el algoritmo especificado en el header, y un secreto llamado *signing-key*.

Al estar los JWTs codificados, no son legibles por un humano, así que podemos utilizar herramientas, como el JWT Debugger, disponible en <https://jwt.io> para ver la información almacenada en un token.



Ejemplo: Añadir soporte para autenticación mediante JWT.

1. En primer lugar debemos añadir el paquete Microsoft.AspNetCore.Authentication.JwtBearer a nuestro proyecto, usando el gestor de paquetes Nuget.



2. En el método *ConfigureServices()*, configuramos la autenticación mediante tokens JWT, a continuación de la parte en la que añadimos el soporte para Identity:

```
services.AddAuthentication(option =>
{
    option.DefaultAuthenticateScheme = JwtBearerDefaults.AuthenticationScheme;
    option.DefaultChallengeScheme = JwtBearerDefaults.AuthenticationScheme;
})
.AddJwtBearer(jwtOptions =>
{
    jwtOptions.RequireHttpsMetadata = false;
    jwtOptions.SaveToken = true;
    jwtOptions.TokenValidationParameters = new TokenValidationParameters()
    {
        ValidateActor = true,
        ValidateAudience = true,
        ValidateLifetime = true,

        ValidIssuer = Configuration["JWTConfiguration:Issuer"],
        ValidAudience = Configuration["JWTConfiguration:Audience"],
        IssuerSigningKey = new SymmetricSecurityKey(
            Encoding.UTF8.GetBytes(Configuration["JWTConfiguration:SigningKey"]))
    };
});
```

En este caso hemos especificado el mecanismo de autenticación para utilizar tokens JWT, y configurado el servicio, indicando que ciertos valores, como la clave de signing-key, se almacenarán en el fichero de configuración.

3. Añadimos los valores de configuración de JWT en el fichero *appsettings.json*.

```
"JWTConfiguration": {
  "Issuer": "http://localhost:44386/",
  "Audience": "http://localhost:44386/",
  "SigningKey": "la clave de firmado debe ser suficientemente larga para ser segura",
  "TokenExpirationDays": 7
}
```

Para los primeros dos valores utilizaremos la URL de nuestro servicio. La clave de firmado puede ser cadena de texto que queramos, y será utilizada para el algoritmo hash. Por último especificamos el tiempo de duración de los tokens generados.

Gestión de usuarios del servicio.

Nuestro servicio contiene ya toda la infraestructura necesaria para utilizar el mecanismo de autenticación basado en tokens JWT, pero aún nos queda mucho trabajo por delante. Si queremos que los servicios sean consumidos sólo por usuarios autorizados, debemos ofrecer un mecanismo para que estos puedan registrarse y loguearse, y especificar qué métodos de nuestra API deben ser utilizados por usuarios autorizados.

El flujo de utilización de nuestros servicios será el siguiente:

1. **Registro:** el usuario se registra con un nombre de usuario y una contraseña. El servicio almacenará la información del usuario en la bd correspondiente.

2. **Login:** el usuario ingresa en el sistema utilizando su nombre de usuario y contraseña, el servicio comprueba que los datos son correctos y si es así devuelve un JWT que contiene la información del usuario.
3. **Autorización de peticiones:** cada petición que recibe nuestro servicio debe ir acompañada por el JWT en la cabecera de la petición. El sistema decodifica y valida el token. A continuación comprueba si el usuario correspondiente al token tiene permiso para utilizar el método de la API solicitado y en caso correcto responde a la petición. En caso de no estar autorizado, devolverá un error de autorización.

Registro de usuarios.

Para nuestro ejemplo, permitiremos que los usuarios puedan registrarse en nuestro servicio sin más. Para ello, nuestra API debe ofrecer un método que permita realizar dicha tarea y almacene a continuación los datos del usuario en la bd.

1. Creamos un nuevo controlador al que llamaremos *AccountController*, que se encargará de la API de registro y login de usuarios.

```
using Microsoft.AspNetCore.Identity;
using Microsoft.AspNetCore.Mvc;
using Microsoft.Extensions.Configuration;

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

        public AccountController(UserManager<IdentityUser> userManager,
            SignInManager<IdentityUser> signInManager, IConfiguration configuration)
        {
            _userManager = userManager;
            _signInManager = signInManager;
            _configuration = configuration;
        }
    }
}
```

Nuestro constructor utilizará diversos objetos para realizar su trabajo, por lo que estos son pasados en el constructor. Concretamente necesitará el *UserManager* para registrar los usuarios de la aplicación, un *SignInManager* será utilizado para el login. Estos objetos son proporcionados por *Identity* como parte de la infraestructura de autenticación.

Como será necesario obtener determinados elementos del fichero de configuración, necesitaremos un objeto *Configuration* para realizar dichas tareas.

2. Al igual que ocurría con las aplicaciones web MVC con las que hemos trabajado hasta ahora, no siempre existe una clase del modelo de datos que se adapta exactamente a la información que queremos utilizar en una vista, o en este caso en un método de la API. Nuestro método para registrar usuarios necesitara que el servicio reciba la información de usuario y contraseña, por lo

que lo que haremos, del mismo modo que hemos hecho en nuestras aplicaciones web, es crear un ViewModel que defina los datos que necesitamos que sean pasados a nuestro método.

En este caso, haremos que el usuario se registre mediante el uso de un email y una password, por lo tanto crearemos una carpeta llamada *ViewModels* y en ella la clase *RegisterUserViewModel*.

```
using System.ComponentModel.DataAnnotations;

namespace TodoApi.ViewModels
{
    public class RegisterUserViewModel
    {
        [Required]
        [EmailAddress]
        public string Email { get; set; }

        [Required]
        [StringLength(100, MinimumLength = 6)]
        [DataType(DataType.Password)]
        public string Password { get; set; }

        [DataType(DataType.Password)]
        [Compare("Password", ErrorMessage = "The password and confirmation password do not match.")]
        public string ConfirmPassword { get; set; }
    }
}
```

Decoramos las propiedades del ViewModel del mismo modo que para una aplicación web, porque al ser la misma infraestructura de una aplicación web MVC, disfrutaremos de las mismas características y herramientas, en este caso la validación automática del modelo.

3. A continuación añadimos un nuevo método para registrar usuarios a nuestra API, en este caso añadimos el método al controlador *AccountController*.

```
[AllowAnonymous]
[HttpPost("register")]
public async Task<IActionResult> Register(
    [FromBody] RegisterUserViewModel registration)
{
    if (!ModelState.IsValid) { return BadRequest(ModelState); }

    IdentityUser newUser = new IdentityUser
    {
        Email = registration.Email,
        UserName = registration.Email,
        Id = registration.Email,
    };

    IdentityResult result = await _userManager.CreateAsync(newUser,
        registration.Password);

    if (!result.Succeeded)
    {
        foreach (var error in result.Errors)
        {
            ModelState.AddModelError(error.Code, error.Description);
        }
        return BadRequest(ModelState);
    }
}
```

```
}
    return Ok();
}
```

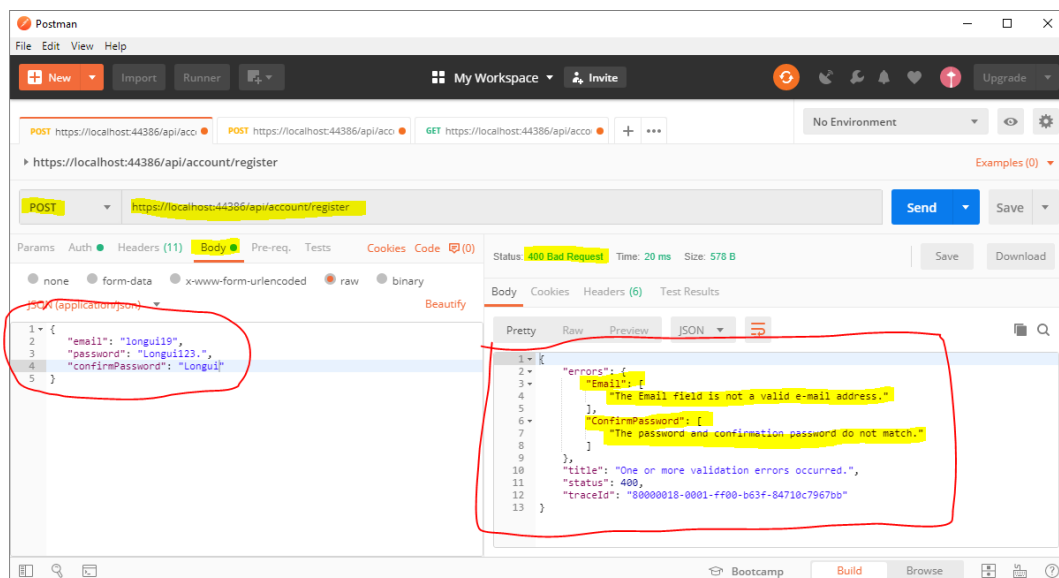
El funcionamiento del método es sencillo. En primer lugar comprobamos si los datos recibidos se ajustan a los requerimientos del modelo, es decir, el correo tiene el formato correcto y la contraseña tiene la longitud requerida y coincide con la confirmación. En caso contrario el método devolvería un error indicando el problema.

A continuación creamos un IdentityUser, que es el usuario por defecto de Identity. En este caso utilizamos el email tanto para el nombre de usuario, como para el propio email, así como para el id del usuario. Esto lo hacemos por simplicidad, aunque obviamente esto no tendría por que ser de este modo.

A continuación registramos el usuario utilizando el manager correspondiente. En caso de que haya algún error devolvemos dichos errores junto con una respuesta de petición incorrecta.

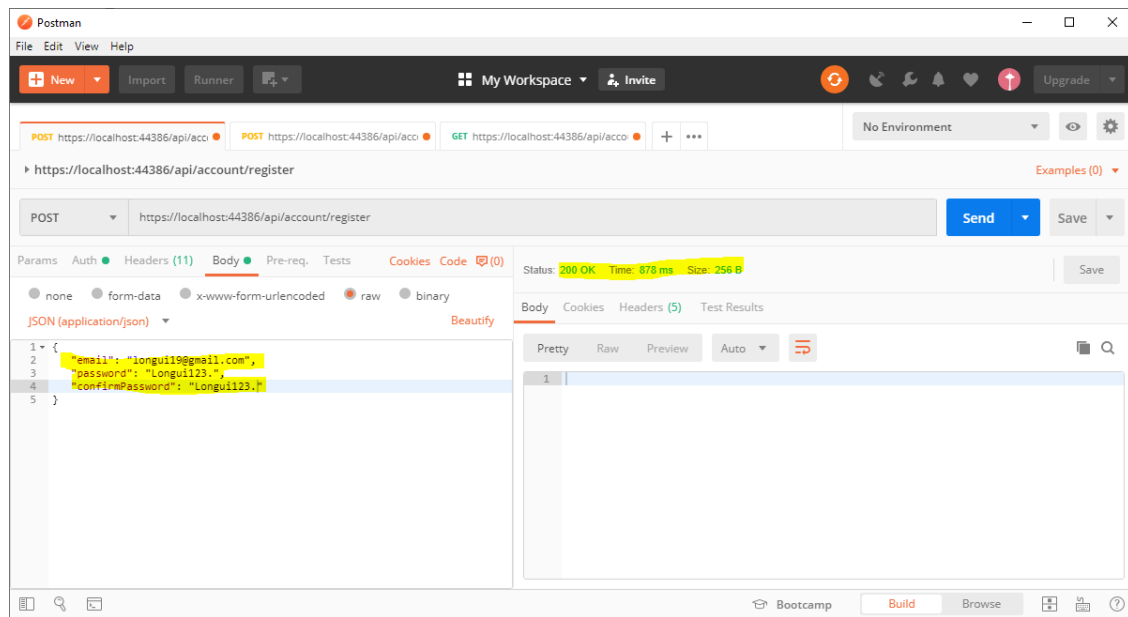
Si todo es correcto y conseguimos registrar al usuario se devuelve un mensaje de operación correcta.

4. Vamos enviar una petición de registro con Postman. Para ello arrancamos nuestro servicio e introducimos la url para llamar al método /account/register de la API. En el cuerpo de la petición POST debemos pasar los datos del usuario conforme al ViewModel, como se muestra en la imagen.

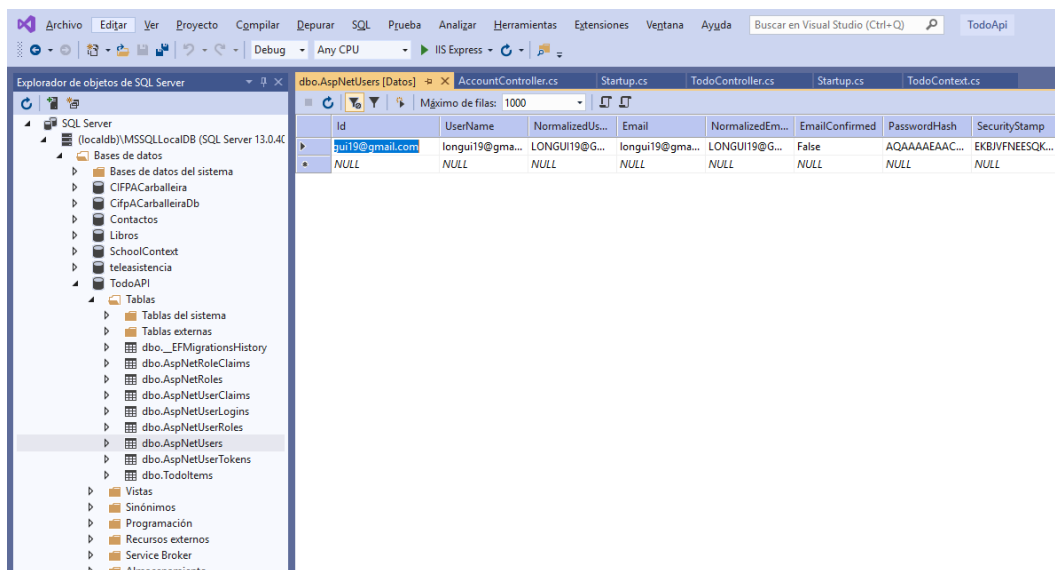


En este caso se ha introducido un correo incorrecto, y las dos contraseñas no coinciden, por lo que podemos ver los mensajes de error indicando dichas circunstancias.

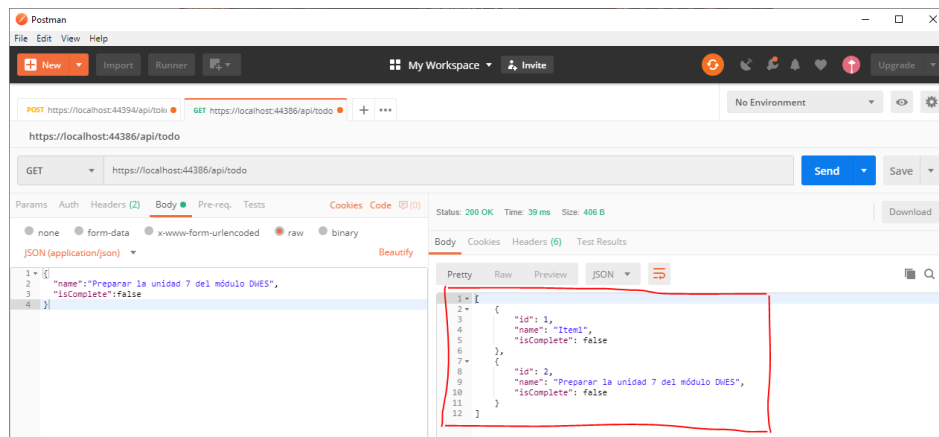
5. Enviamos ahora una nueva petición, esta vez con los datos correctos. Podemos comprobar que en este caso se devuelve un código 200, que indica que la petición ha sido procesada con éxito.



Si abrimos la tabla en la bd comprobamos que el usuario ha sido guardado correctamente, como era de esperar.



- Si cambiamos de nuevo nuestra petición para enviar una solicitud GET veremos como en este caso la lista de tareas incluye la nueva tarea que acaba de ser añadida.



Generación del JWT y login.

Tal y como hemos explicado, el flujo de autenticación basado en tokens implica que cuando un usuario se loguea en el sistema, el servicio genera un token con la información del usuario, que será devuelto al cliente para utilizarlo en las siguientes peticiones.

Debemos, por lo tanto, crear un método en nuestra API que permite a un usuario hacer login. Para su funcionamiento hará uso de otro método auxiliar que será el encargado de generar un token JWT con la información del usuario.

1. Empezamos por implementar el método que genera el token para un usuario:

```
private async Task<JwtSecurityToken> GenerateTokenAsync(IdentityUser user)
{
    var claims = new List<Claim>()
    {
        new Claim(JwtRegisteredClaimNames.Sub, user.UserName),
        new Claim(JwtRegisteredClaimNames.Jti, Guid.NewGuid().ToString()),
        new Claim(ClaimTypes.NameIdentifier, user.Id),
        new Claim(ClaimTypes.Name, user.UserName),
    };

    // Loading the user Claims
    var expirationDays =
        _configuration.GetValue<int>("JWTConfiguration:TokenExpirationDays");
    var signingKey = Encoding.UTF8.GetBytes(
        _configuration.GetValue<string>("JWTConfiguration:SigningKey"));
    var token = new JwtSecurityToken(
        issuer: _configuration.GetValue<string>("JWTConfiguration:Issuer"),
        audience: _configuration.GetValue<string>("JWTConfiguration:Audience"),
        claims: claims,
        expires: DateTime.UtcNow.Add(TimeSpan.FromDays(expirationDays)),
        notBefore: DateTime.UtcNow,
        signingCredentials: new SigningCredentials(
            new SymmetricSecurityKey(signingKey), SecurityAlgorithms.HmacSha256)
    );

    return token;
}
```

Aunque el código puede parecer complejo a simple vista, después de analizarlo veremos que en realidad es sencillo. Lo primero que hacemos es definir una lista de claims, que son objetos que

definen información sobre el usuario, que serán almacenados como información en el token que vamos a generar. En nuestro caso almacenamos el id y el nombre del usuario.

A continuación, generamos el token creando un objeto `JwtSecurityToken`, pasando la información requerida para la creación del mismo. Parte de dicha información la obtenemos del fichero de configuración, como por ejemplo la `signing-key` o el tiempo de expiración. También indicamos el algoritmo que se utilizará para la generación del hash, además de la lista de claims que se incluirán en el token.

- Ahora que podemos generar nuestro token, nos falta añadir un método para hacer login en nuestra API, pero de nuevo, antes de eso, necesitamos definir un `ViewModel` que contenga la información que será pasada al método. En nuestro caso para hacer login necesitaremos solamente el email y la contraseña, así que creamos la clase `LoginUserViewModel`:

```
namespace TodoApi.ViewModels
{
    public class LoginUserViewModel
    {
        public string Email { get; set; }
        public string Password { get; set; }
    }
}
```

- En este caso concreto, además, nuestro método de la API devolverá información al cliente. Esta información debe ser definida como un `ViewModel` del mismo modo. Es importante tener en cuenta que todos los objetos que se pasen en un sentido u otro deben estar perfectamente definidos en la interfaz, para que el cliente conozca la información que va a ser pasada o recibida. Añadiremos a la carpeta de view models la clase `LoginResultViewModel`, que define una única propiedad devuelta, que contiene el token:

```
namespace TodoApi.ViewModels
{
    public class LoginResultViewModel
    {
        public string Token { get; set; }
    }
}
```

- Ahora ya podemos crear el método de la API que permitirá loguearse, obteniendo así el token de autenticación correspondiente:

```
[AllowAnonymous]
[HttpPost("login")]
public async Task<ActionResult<LoginResultViewModel>> Login(
    [FromBody] LoginUser login)
{
    SignInResult result = await _signInManager.PasswordSignInAsync(login.Email,
        login.Password, isPersistent: false, lockoutOnFailure: false);

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

    IdentityUser user = await _userManager.FindByEmailAsync(login.Email);
    JwtSecurityToken token = await GenerateTokenAsync(user);
}
```

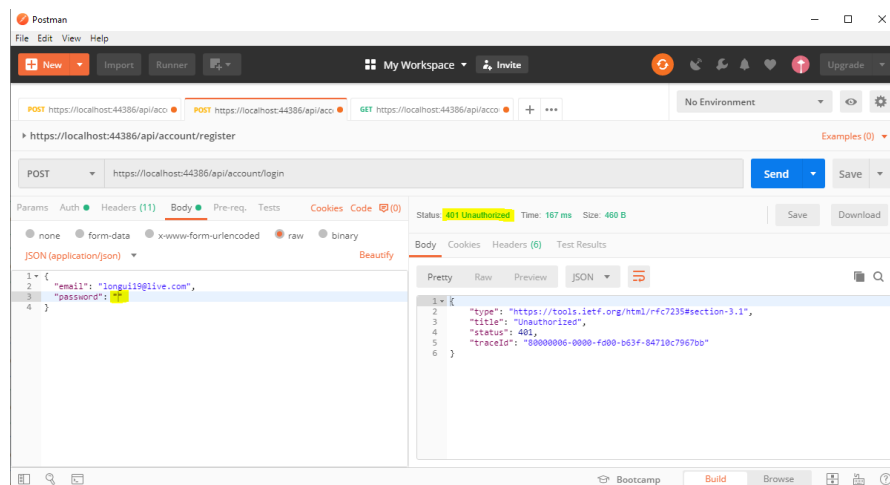
```
string serializedToken = new JwtSecurityTokenHandler().WriteToken(token);

return Ok(new LoginResultViewModel() { Token = serializedToken});
}
```

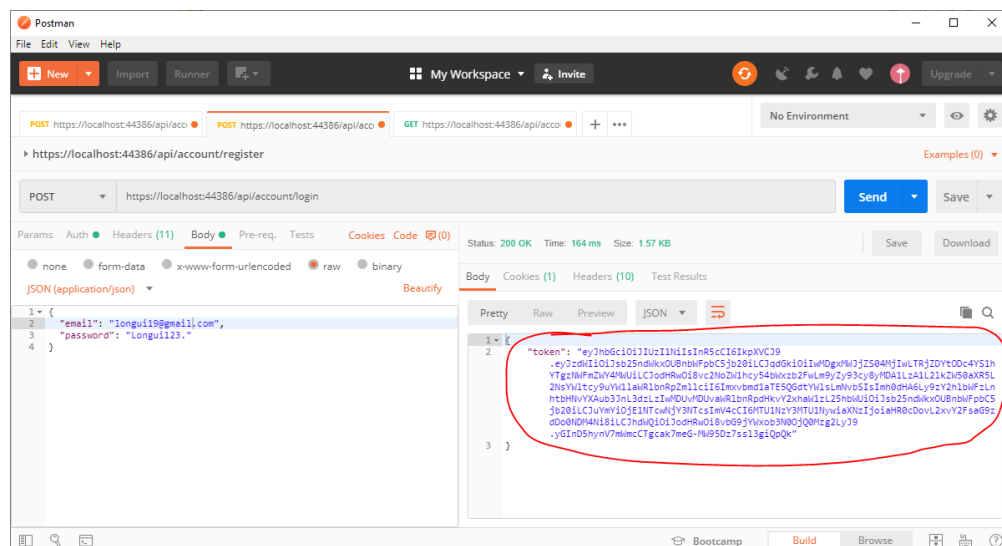
En primer lugar, nuestro código utiliza el SignInManager para hacer el login. Si los datos no son correctos, el método devolverá un error de autorización.

En caso de haber introducido los datos de un usuario registrado, obtenemos los datos de dicho usuario y generamos el token a partir de dichos datos, utilizando el método auxiliar que explicamos con anterioridad. Por último, serializamos el token y lo devolvemos al cliente.

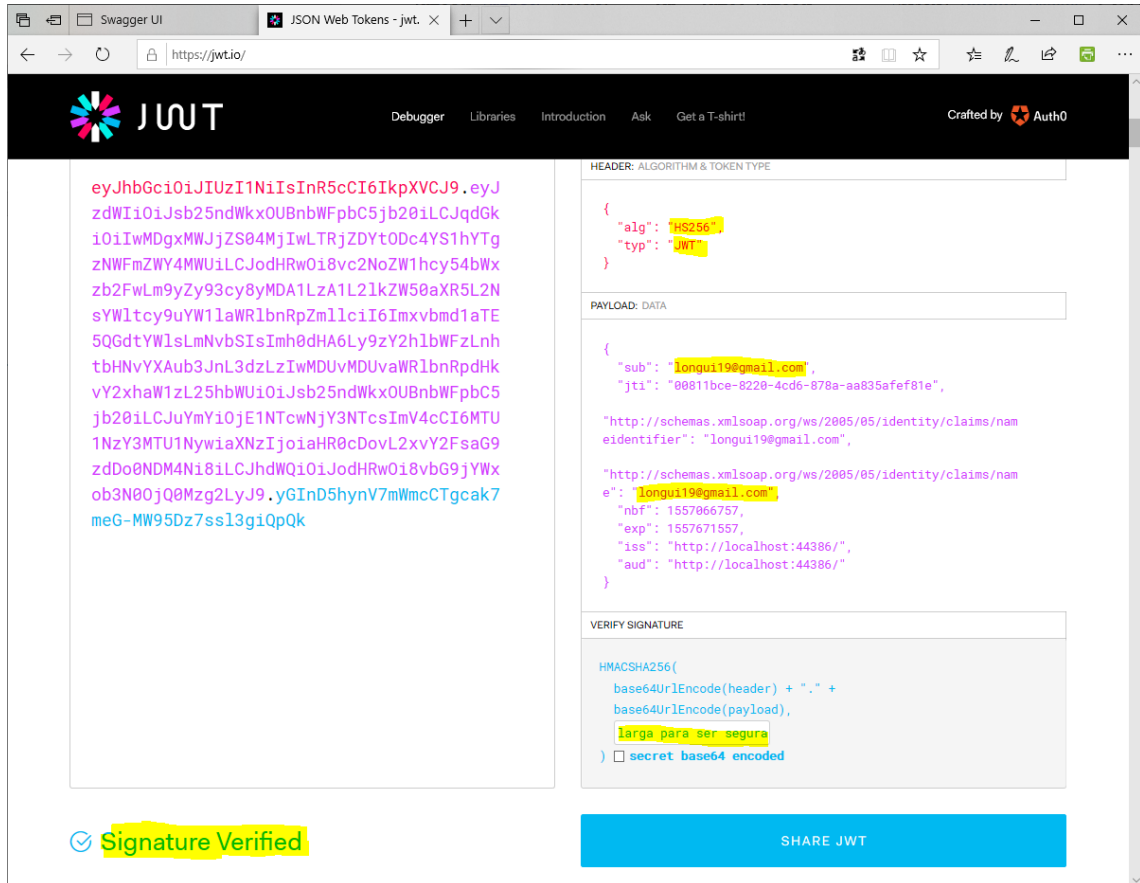
- Una vez hechos estos cambios, arrancaremos de nuevo nuestro servicio y probaremos la nueva funcionalidad con Postman. La petición será un POST a <https://localhost:xxxxxxx/api/account/login>. Probamos primero introduciendo datos de un usuario incorrecto, viendo como un código de error de autorización es devuelto.



Ahora introduciremos los datos de un usuario registrado, obteniendo en este caso un código 200 que indica que la operación ha sido correcta, además de lo más importante, nuestro token de autorización.



Como podemos observar en la imagen, la respuesta incluye la cadena de texto que representa el token JWT. Vamos a comprobar que se trata efectivamente de un JWT correcto, utilizando el depurador de JWT. Para ello copiamos el texto del token de Postman y lo pegamos en el depurador. Debemos introducir también la clave de firmado en el cuadro correspondiente para validar la firma.



The screenshot shows the JWT.io web application interface. The 'Token' field contains a long JWT string. The 'Header' section shows the algorithm as 'HS256' and the token type as 'JWT'. The 'Payload' section shows the claims, including 'sub' (longui19@gmail.com) and 'jti' (00811bce-8220-4cd6-878a-aa835afe81e). The 'Verify Signature' section shows the HMACSHA256 algorithm and the token structure. The 'Signature Verified' status is displayed at the bottom.

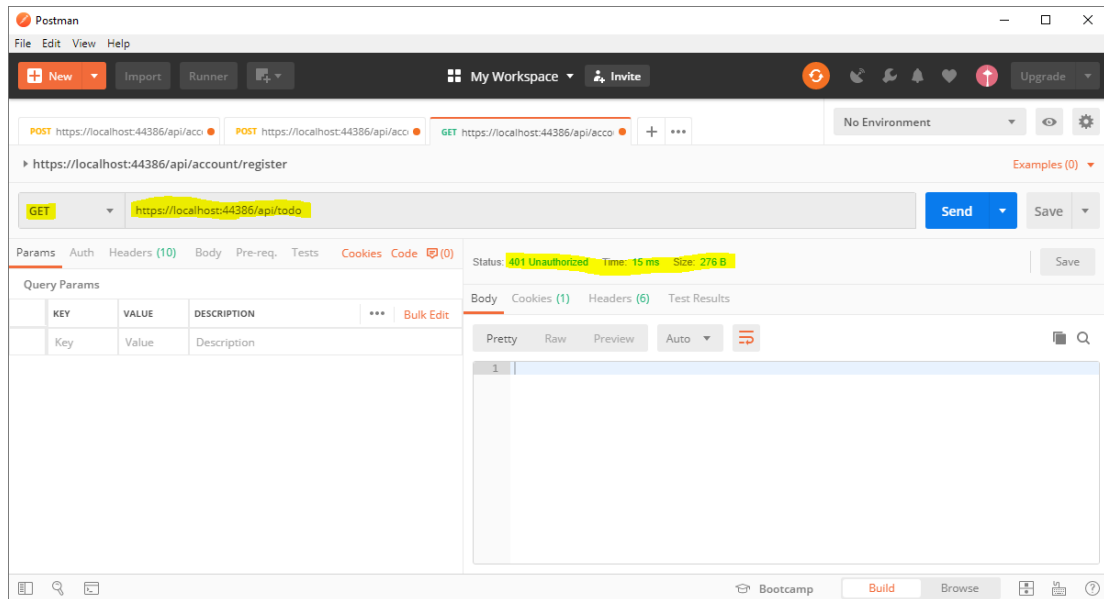
Implementación de la autorización en los métodos de la API.

A pesar de todo nuestro esfuerzo, si intentamos hacer cualquier llamada a la API podremos comprobar como los datos son devueltos sin problema. El último paso es tan necesario como sencillo. Debemos indicar qué métodos de nuestra API requieren que el usuario esté autenticado, indicando esto mediante el atributo `[Authorize]`. Podemos utilizar el atributo para cada método que queramos proteger, o a nivel de controlador. En este caso, ya que queremos proteger todos los métodos del controlador escribiremos el atributo una única vez antes de la definición de la clase.

1. Añadimos el siguiente código a nuestro controlador:

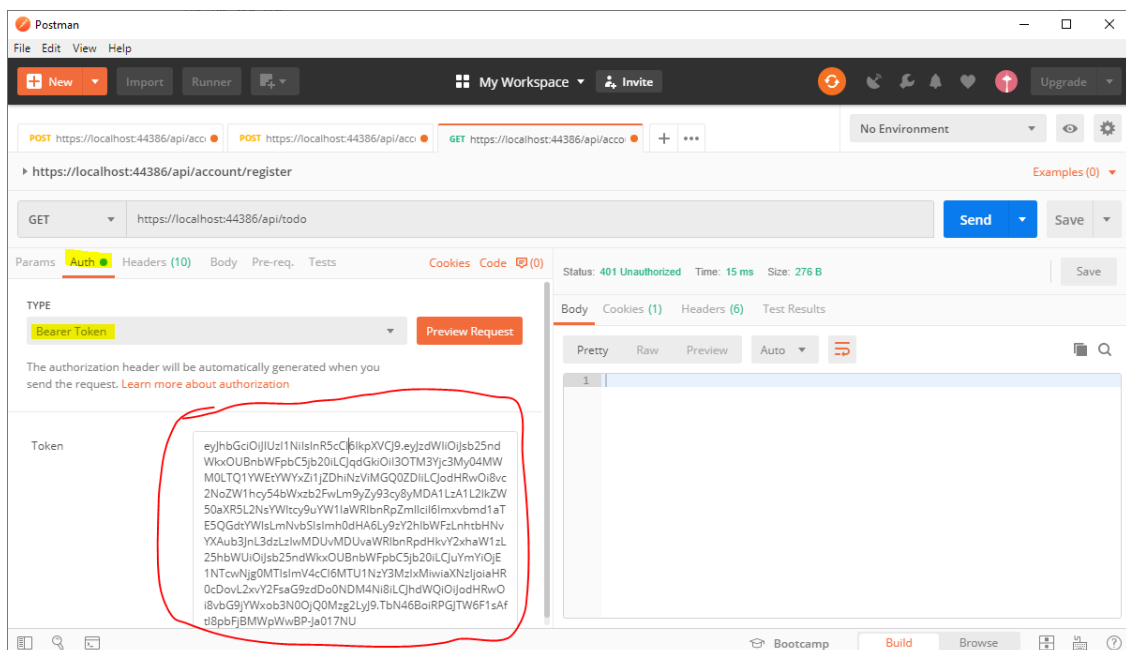
```
[Route("api/[controller]")]
[ApiController]
[Authorize]
public class TodoController : ControllerBase
```

- Ahora que todo está en orden, comprobaremos el funcionamiento del mismo. Lanzamos una vez más nuestro servicio y hacemos una petición GET a la URL `https://localhost:xxxxxxx/api/todo`.

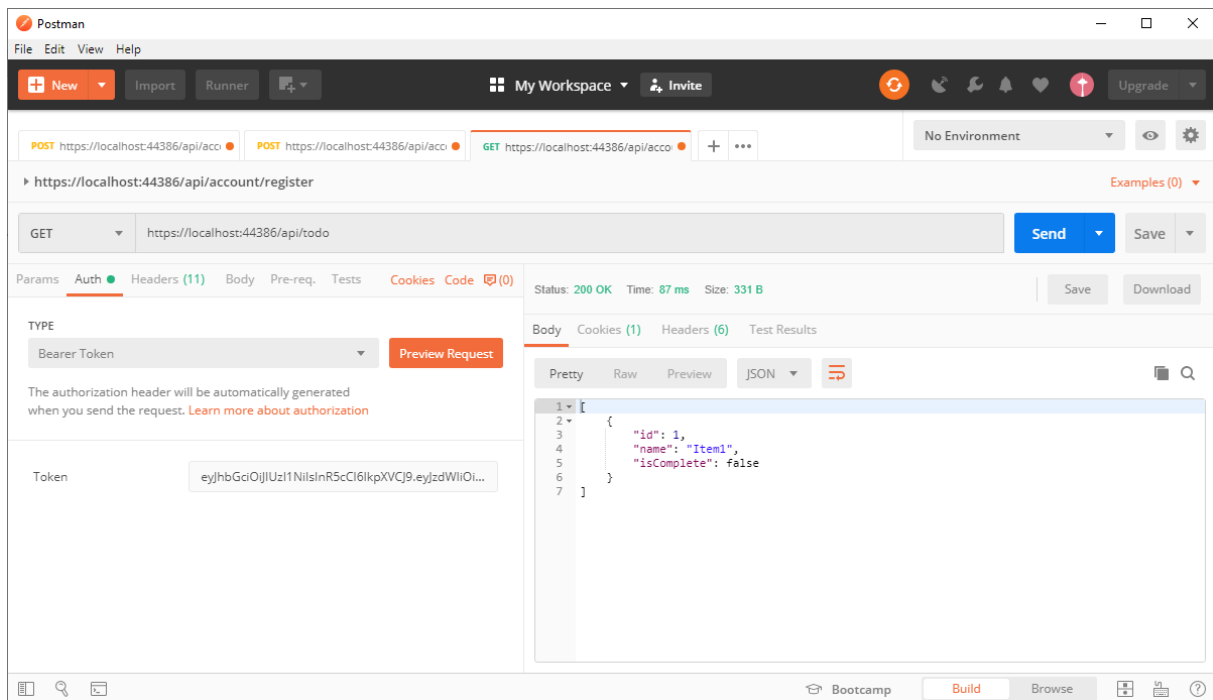


La respuesta obtenida por el servicio es un error de autorización 401, debido a que todos los métodos de nuestra API están ahora protegidos contra el acceso no autorizado. Para poder utilizar nuestro servicio, el token debe ser incluido en la cabecera de la petición.

En Postman disponemos de una pestaña llamada Auth, que facilita la tarea de incluir un token en nuestra petición. Para ello en el desplegable seleccionamos Bearer Token como tipo de dato y pegamos el token en el cuadro Token. Si no lo hemos copiado con anterioridad, debemos llamar de nuevo al método login para obtener el mismo.



Cuando pulsemos enviar de nuevo, esta vez si obtendremos la respuesta esperada, obteniendo la lista de tareas almacenada en la bd.



Para poder utilizar cualquier método de nuestra API, debemos repetir esta operación, copiando el JWT de autorización en la cabecera.

Si implementamos un cliente para acceder a nuestros servicios, será responsabilidad del cliente almacenar el token para conservarlo para futuras peticiones, manteniendo nuestro servicio stateless, siguiendo de este modo las especificaciones RESTful.

Bibliografía:

- Hands-On Full-Stack Web Development with ASP.NET Core

Tamir Dresher, Amir Zuker and Shay Friedman

Copyright © 2018 Packt Publishing