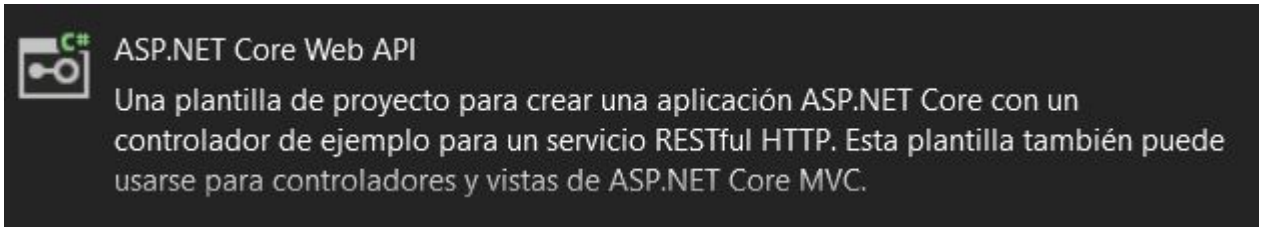
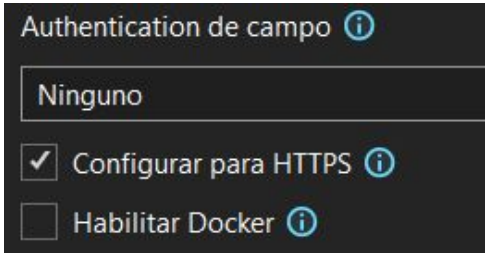
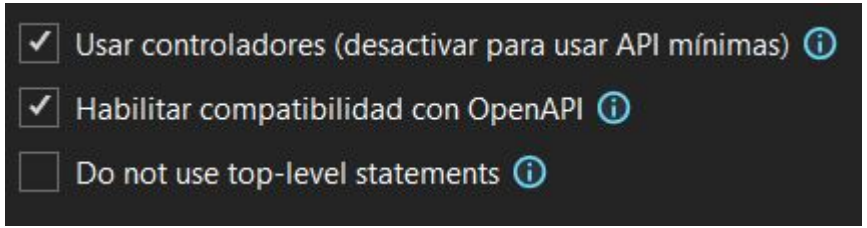
ASP.NET Core Web API: Estructura

Para crear un proyecto de este tipo, seleccionaremos la siguiente plantilla:



Y dejaremos las opciones que vienen por defecto:





ASP.NET Core Web API: Estructura

Controladores

En una API los *Controllers* no devuelven vistas como en MVC, sino JSON, por lo que el retorno de sus métodos serán distintos.

La cabecera del controlador tiene las siguientes particularidades

```
[ApiController]
[Route("[controller]")]
3 referencias
public class WeatherForecastController : ControllerBase
```

- [ApiController] → Indica que es un controlador para una API
- [Route(“[controller]”] → Indica si la ruta de acceso tuviera algún prefijo
- ControllerBase → Todos los controladores de una API implementan esta interfaz, que determina que es un controlador pero no utiliza vistas.

ASP.NET Core Web API: Estructura

Controladores → Acciones

Si existiera ambigüedad a la hora de determinar qué acción del controlador es la que se ha de ejecutar para un *endpoint* determinado, se puede establecer un nombre concreto para una acción, el cual pasará a formar parte del *endpoint* de esa acción.

Esta ambigüedad se produce cuando existen al menos dos acciones del mismo método HTTP que acepten los mismos parámetros.

Para romper esta ambigüedad usamos estos atributos en la acción:

```
[Route("AddStudent")]
[HttpPost]
public ActionResult PostStudent(Student student)
```

→ `https://localhost:7042/Students/AddStudent`

```
[HttpPost("[action]")]
public ActionResult PostStudent(Student student)
```

→ `https://localhost:7042/Students/PostStudent`

ASP.NET Core Web API: Controladores

Atributo **ProducesResponseType**

Este atributo de las acciones de nuestra Web API nos permite especificar los tipos de retorno que tienen estas acciones. Se corresponden con los métodos retornados en las acciones que devuelven IActionResult y ActionResult<T>

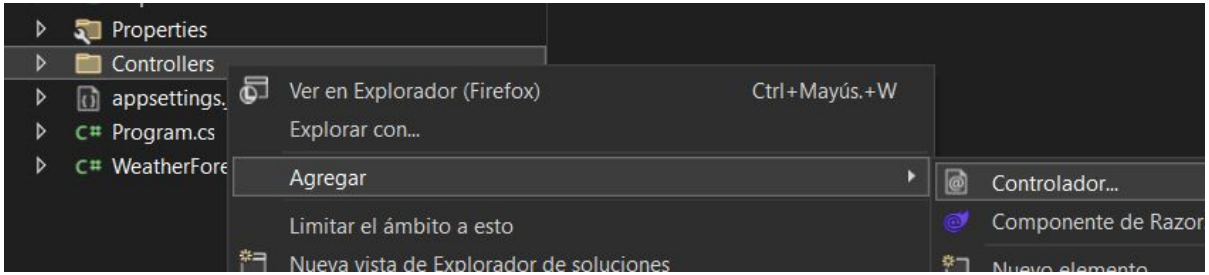
Especificar esto es una buena práctica que ayuda a la mejor documentación de la API desarrollada.

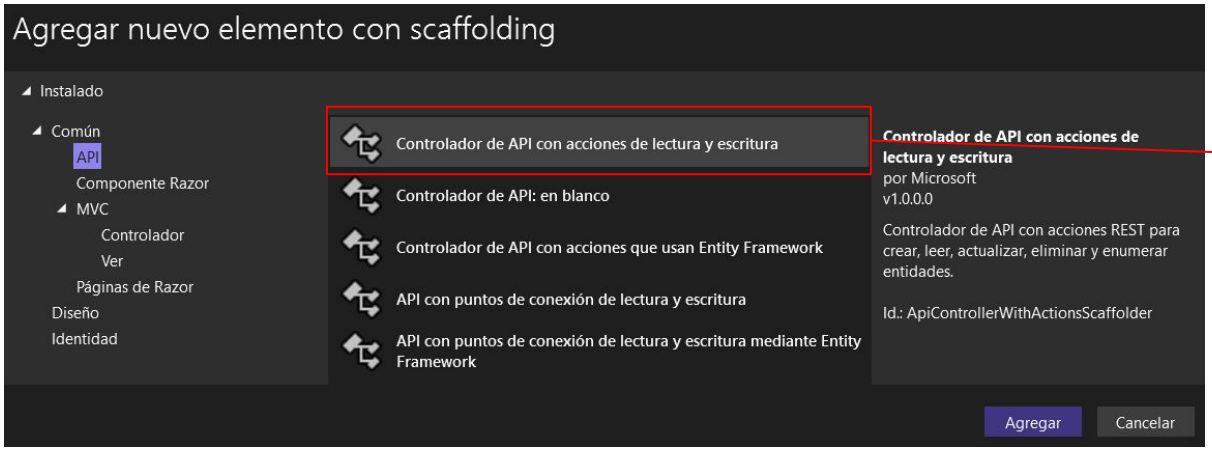
Definen el *Status Code* de la respuesta, y un ejemplo sería el siguiente:

```
// DELETE api/<GamesController>/5
[HttpDelete("{id}")]
[ProducesResponseType(StatusCodes.Status200OK)]
[ProducesResponseType(StatusCodes.Status404NotFound)]
0 referencias
public IActionResult Delete(int id)
```

ASP.NET Core Web API: Crear Controladores

A la hora de incluir un nuevo controlador a la aplicación, veremos que hay varios tipos de los que pertenecen a la categoría API





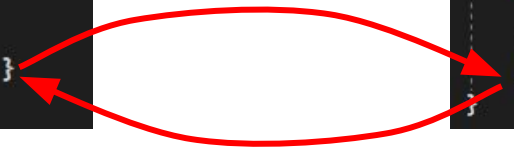
Generamos un nuevo controlador de este tipo, que no está ligado a un modelo en concreto.
De momento no utilizamos persistencia de datos.

Persistencia de Modelos

Referencias Circulares:

```
public class Company
{
    11 referencias
    public int Id { get; set; }
    15 referencias
    public string? Name { get; set; }
    15 referencias
    public string? Description { get; set; }

    11 referencias
    public int ShipperID { get; set; }
    9 referencias
    public Shipper? Shipper { get; set; }
}
```



```
public partial class Shipper
{
    [Key]
    11 referencias
    public int ShipperID { get; set; }
    [Required]
    [StringLength(40)]
    15 referencias
    public string CompanyName { get; set; }
    [StringLength(24)]
    12 referencias
    public string Phone { get; set; }

    0 referencias
    public List<Company> Company { get; set; }
}
```

Persistencia de Modelos

Referencias Circulares: Soluciones

- Ignorar Ciclos

Se añade una opción al servicio de controladores, en Program.cs, para que se ignoren los ciclos redundantes.

```
builder.Services.AddControllers().AddJsonOptions(x =>
                x.JsonSerializerOptions.ReferenceHandler = ReferenceHandler.IgnoreCycles);
```

Persistencia de Modelos

Referencias Circulares: Soluciones

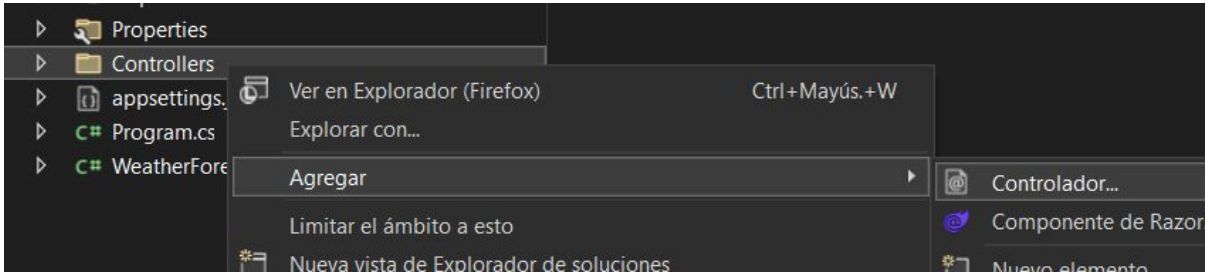
- Seleccionar los campos que contendrá el JSON final

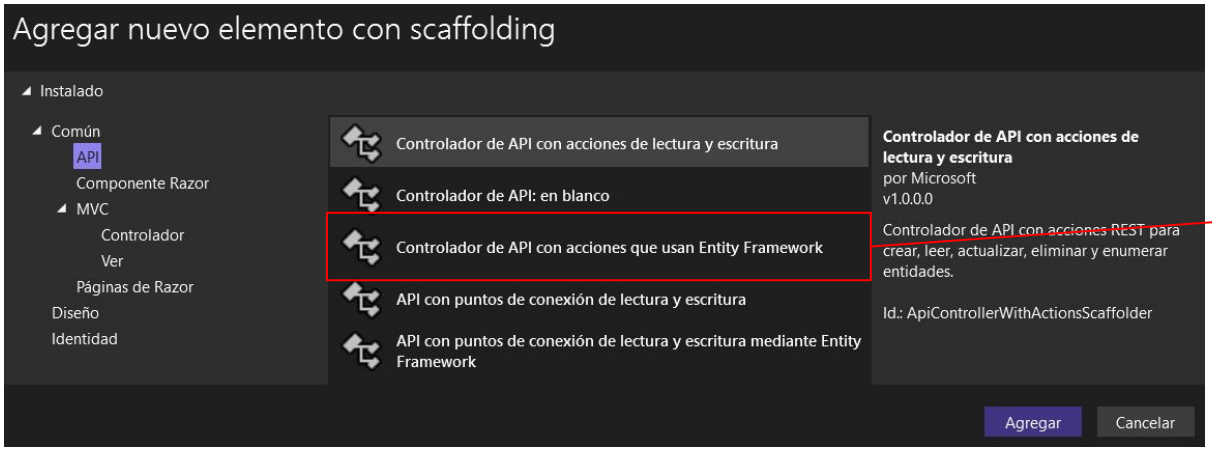
```
return await _context.Game
.Select( g => new
{
    Id = g.Id,
    Name = g.Name,
    Genre = new
    {
        Id = g.Genre.Id,
        Name = g.Genre.Name
    }
})
.ToListAsync();
```

```
return await _context.Genre
    .Select(g => new
    {
        Id = g.Id,
        Name = g.Name,
        Games = g.Games.Select(g => new { g.Name })
    })
    .ToListAsync();
```

ASP.NET Core Web API: Crear Controladores

En base a un modelo se puede generar mediante *scaffold* un controlador de este modelo.





Generamos un nuevo controlador de este tipo que esté ligado a un modelo existente.
Esto genera el controlador del modelo y la clase de contexto para acceso a la BD en caso de que no existiera uno aún.

Autenticación y Autorización → Implementación

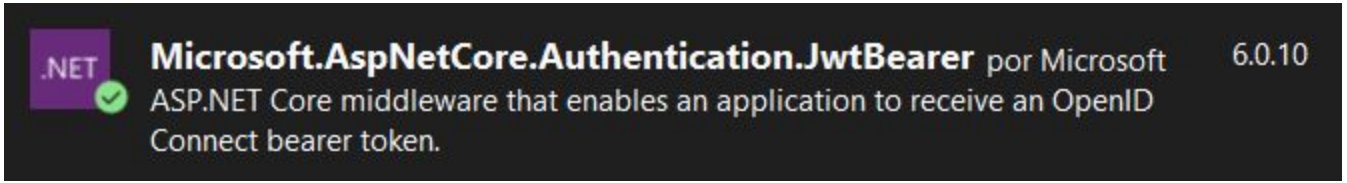
Para poder definir las distintas acciones que hay que realizar para poder utilizar técnicas de autenticación y autorización en nuestra API, mediante Tokens JWT emitidos por nuestro propio servidor (que hará las funciones de servidor de autorización), será necesario partir de un proyecto que cumpla las siguientes condiciones:

- Proyecto ASP.NET Core Web Api con controladores
- Al menos un modelo emparejado con una tabla en una base de datos (ya sea mediante *Code First* o mediante *Database First*) utilizando EF Core.
- Inclusión de los siguientes paquetes NuGet de Identity:
  - Microsoft.AspNetCore.Identity
  - Microsoft.AspNetCore.Identity.EntityFramework
- Incluir servicio Identity en el constructor de la aplicación (program.cs). En esta ocasión se usará la llamada “AddIdentityCore”.
- Haber generado en la BD las tablas que utiliza Identity mediante Migraciones y actualizaciones de la BD.
- Haber sembrado las tablas con usuarios, roles y sus relaciones, como vimos en MVC.

Autenticación y Autorización → Implementación

INSTALAR PAQUETES NUGET NECESARIOS

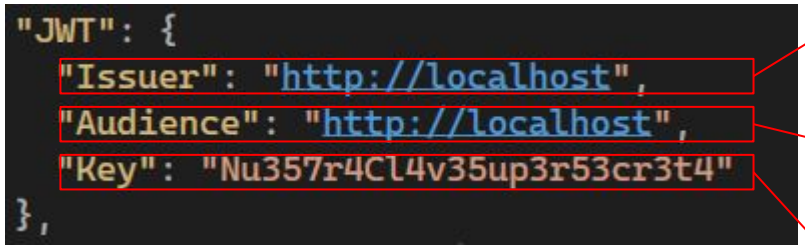
El único paquete necesario, para utilizar tokens JWT en nuestro proyecto, es el siguiente:



Autenticación y Autorización → Implementación

MODIFICAR ARCHIVO DE CONFIGURACIÓN

En nuestro archivo de configuración (appsettings.json) se tendrán que añadir lo siguiente:



El campo Issuer identifica el proveedor de identidad que emitió el Token

Identifica la audiencia o receptores para lo que el JWT fue emitido, normalmente el/los servidor/es de recursos (e.g. la API protegida)

Clave privada que se usará para firmar el token generado o recibido

Autenticación y Autorización → Implementación

MODIFICAR PROGRAM.CS → Añadir Servicio de Autenticación

Dado que usaremos un token JWT de tipo Bearer como esquema de autenticación, tendremos que incluir este servicio al constructor de la siguiente forma:

```
builder.Services.AddAuthentication(JwtBearerDefaults.AuthenticationScheme)
    .AddJwtBearer(options => ...);
```

Vemos que se ha especificado un esquema “JWT Bearer” para la autenticación, y se invocado al método “AddJwtBearer” al que se le pasan una serie de opciones de configuración del funcionamiento de este esquema de autenticación.

Las opciones utilizadas podrían ser como las siguientes:

Autenticación y Autorización → Implementación

MODIFICAR PROGRAM.CS → Añadir Servicio de Autenticación

En entorno de desarrollo no se requiere https en las peticiones

Se almacena el token

```
options.RequireHttpsMetadata = false;
options.SaveToken = true;
options.TokenValidationParameters = new TokenValidationParameters
{
    ValidateIssuer = true,
    ValidateAudience = true,
    ValidateLifetime = true,
    ValidateIssuerSigningKey = true,
    ValidIssuer = builder.Configuration["Jwt:Issuer"],
    ValidAudience = builder.Configuration["Jwt:Audience"],
    IssuerSigningKey = new SymmetricSecurityKey(Encoding.UTF8.GetBytes(builder.Configuration["Jwt:Key"]))
};
```

Campos del token que serán validados en su recepción

Se indican algunas propiedades que se usarán para comprobar si el token es válido.

Autenticación y Autorización → Implementación

MODIFICAR PROGRAM.CS → Habilitar Autenticación

Al igual que en MVC, tendremos que habilitar las opciones de autenticación a la app construida.

NOTA: Es importante que se habilite la autenticación antes de la autorización, al igual que hicimos en MVC.

```
app.UseAuthentication();
```

Autenticación y Autorización → Implementación

CREAR CONTROLADOR DE AUTENTICACIÓN - Login

```
var claims = new List<Claim> {
    new Claim(JwtRegisteredClaimNames.Jti, Guid.NewGuid().ToString()),
    new Claim(JwtRegisteredClaimNames.Iat, DateTime.UtcNow.ToString()),
    new Claim("UserId", user.Id),
    new Claim("UserName", user.UserName),
    new Claim("Email", user.Email)
};

var key = new SymmetricSecurityKey(Encoding.UTF8.GetBytes(_configuration["Jwt:Key"]));
var credentials = new SigningCredentials(key, SecurityAlgorithms.HmacSha256);
var token = new JwtSecurityToken(
    _configuration["Jwt:Issuer"],
    _configuration["Jwt:Audience"],
    claims,
    expires: DateTime.UtcNow.AddMinutes(10),
    signingCredentials: credentials);

return Ok(new JwtSecurityTokenHandler().WriteToken(token));
```