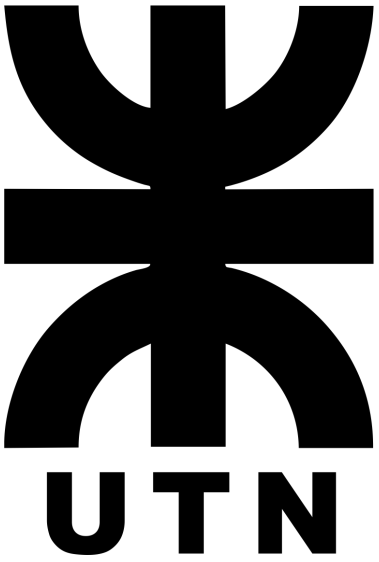
UNIVERSIDAD TECNOLÓGICA NACIONAL



REGIONAL CÓRDOBA

SECRETARIA DE ASUNTOS ESTUDIANTILES
2025

TALLER DE DESARROLLO C#
UNIDAD Nº 2

PROFESOR:

- **Genaro Rafael Bergesio**

COORDINADOR:

- **Octavio Felix Cavalleris Malanca**

SECRETARIO SAE:

- **Exequiel Carranza**

Índice

Introducción

API REST

Una API REST (Representational State Transfer) es una interfaz que permite que distintos sistemas se comuniquen utilizando el protocolo HTTP. Se trata de uno de los pilares fundamentales en el desarrollo moderno de software, especialmente en aplicaciones web y móviles. Debemos verlo como un “traductor” entre los diferentes componentes que tenemos en nuestro sistema, puede hacernos de puente entre la base de datos a JSON, de XML a JSON o ajustarse a la necesidad que tengamos.

Al ser de carácter REST significa que vamos a tener una petición desde el cliente que va a activar a nuestro servidor para que realice una acción. Esto es diferente a por ejemplo colas de mensajería ya que nuestra API espera que le lleguen las solicitudes a los endpoints configurados. Aclaró que un endpoint (o punto final) no es más que una URL válida que mediante el protocolo y parámetros correctos ejecuta una acción en nuestra aplicación.

Algunas Características:

- Statelessness
  - Sin estado, la API no guarda solicitudes anteriores y cada una debe ser resuelta antes de finalizar la conexión.
- Cliente-Servidor
  - Como mencionamos, un cliente realiza solicitudes el servidor responde
- Cache
  - Podemos guardar algunas respuestas para mejorar la velocidad.
- Generalmente HTTP
  - Muy comúnmente utilice los códigos y métodos de HTTP
- Interoperabilidad
  - Es posible ejecutarlos en diversos sistemas operativos.

Métodos HTTP

Las solicitudes HTTP cuentan con varios componentes que permiten la comunicación entre el cliente y el servidor. A destacar:

- **URL**
  - Es la dirección adonde estamos apuntando, sería nuestra ruta de acceso al endpoint.
  - Está compuesta de igual manera que las URL que utilizan los navegadores.
  - Puede usar parámetros que se identifican en la misma URL.
- **Encabezados**

    - **Content-Type:** Tipo de contenido envíado, En nuestro caso usaremos ‘application/json’.
    - **Authorization:** Es un token de autenticación, si bien en el curso no lo vamos a ver es importante que cuando realicen aplicaciones le brinden seguridad y un sistema de reconocimiento de usuarios. Un esquema muy común para lo que vamos a desarrollar es Bearer.
    - **Accept:** Formato que el cliente espera como respuesta.
- **Body**
    - Es el contenido que queremos mandar que por su tamaño no pueden ir en los parámetros
    - Generalmente cuando hacemos un GET obtenemos en el Body la respuesta
    - No todos los métodos nos permiten mandar un body.
- **Códigos**
    - Este protocolo maneja una serie de códigos para sus respuestas que permite identificar al usuario cuál es el estado de su solicitud. Les muestro un par:
    - **200** (OK)
        - Mandamos la solicitud y se procesó correctamente generalmente los GET devuelve esto
    - **201** (Created)
        - Se creó el objeto que deseamos, muy común cuando usamos un método POST
    - **400** (Bad Request)
        - Se utiliza para identificar un error en la solicitud del cliente. Se suele usar a menudo para cuando ocurre una excepción en el código
    - **403** (Forbidden)
        - Este error es que estamos autenticados como usuarios válidos pero estamos solicitando algo fuera de nuestras credenciales. Por ejemplo si nuestro rol es de usuario base y queremos ejecutar un endpoint de administrador
    - **404** (Not Found)
        - Muy común de ver en las páginas web es que ese recurso no existe o no tenemos forma de visualizarlo.
    - **500** (Server error)
        - Generalmente si la aplicación sufre una excepción no controlada.

Los métodos que vamos a utilizar en este curso son:

- GET Obtenemos datos
    - Se puede usar sin parámetros: api/controlador/metodo
    - Con parámetros: api/controlador/metodo/{id}
- POST Enviamos datos
    - También hay consultas que pueden ser hechas por este método porque uno manda en el body el objeto que está consultando.
- PUT o PATCH Actualizamos o parchamos

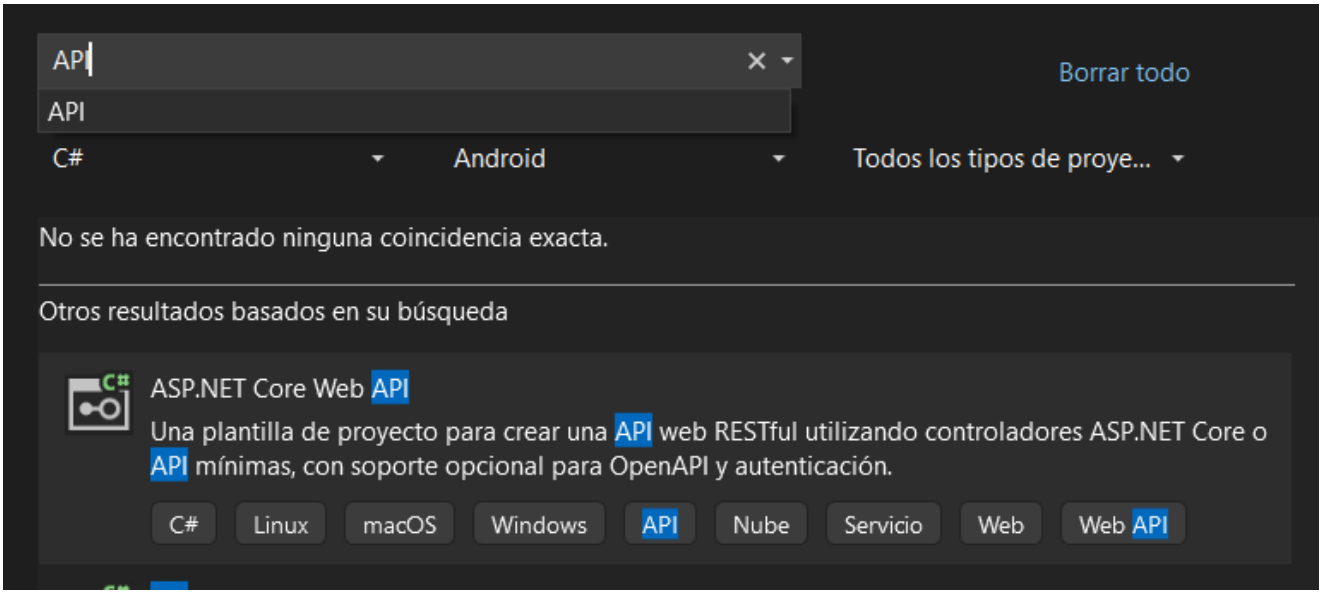
    - PUT se usa con por ejemplo un parámetros id y en el body mandamos a toda la información actualizada
    - PATCH por otro lado puede usarse para modificar una parte de los parámetros
- DELETE Elimina datos
    - Sirve para eliminar algún registro, generalmente se utiliza el ID

Iniciar una API

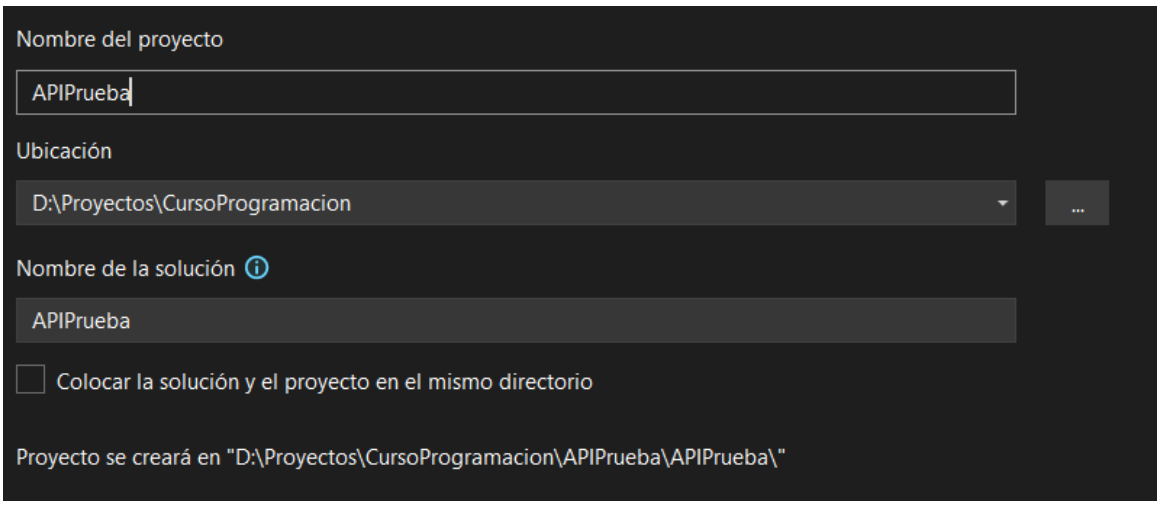
Visual Studio 2022

En la parte derecha del inicio apretamos la opción “Crear un Proyecto”

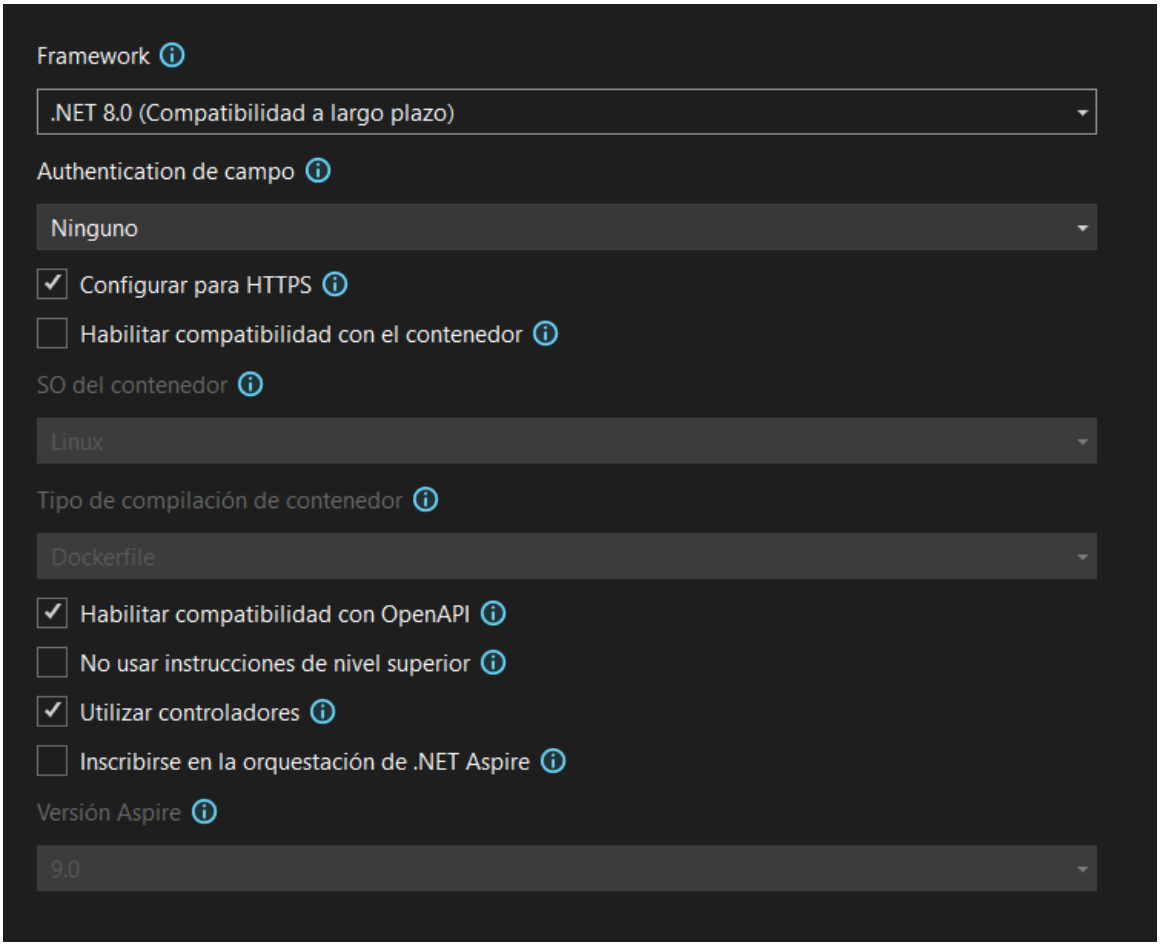
Nos aparecen los templates que usamos y disponibles, en el buscador debemos encontrar la opción que se llama “ASP.NET Core Web API”.



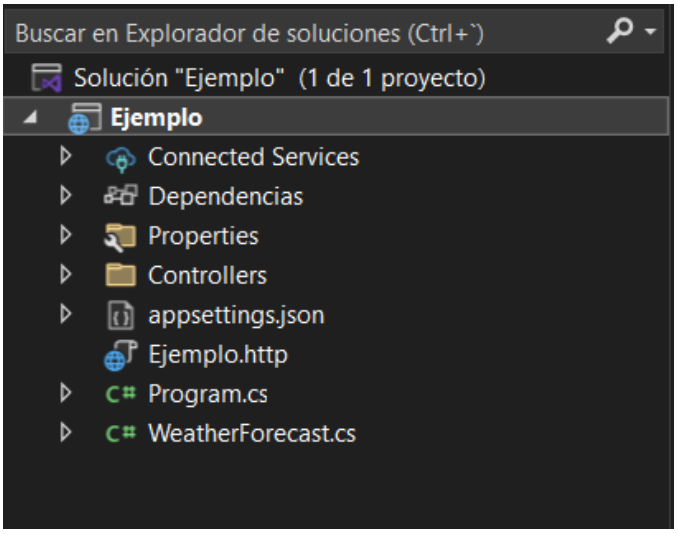
Completamos los datos:



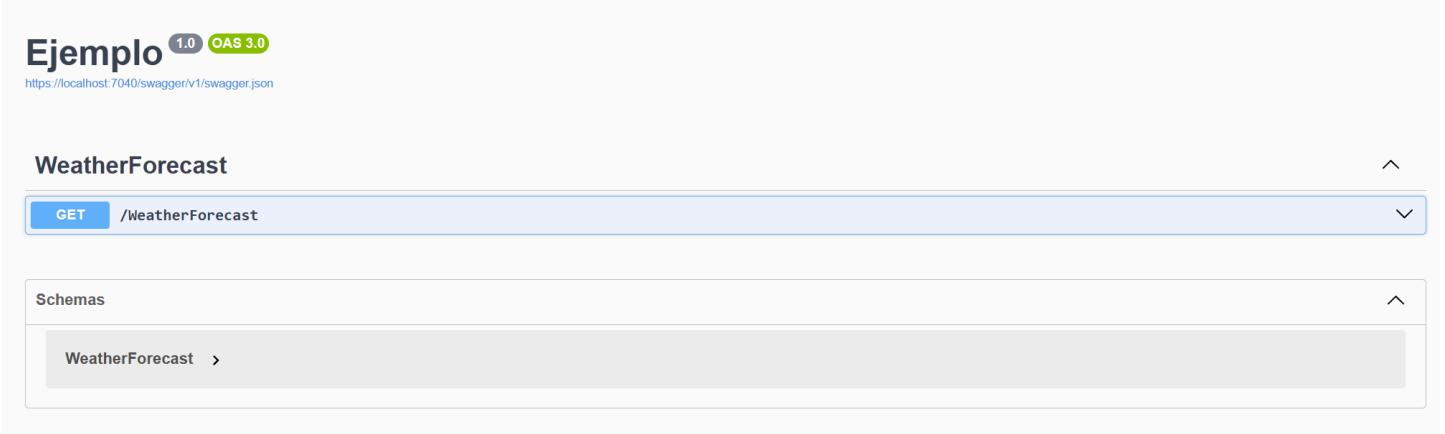
Yo uso la siguiente configuración predeterminada:



Debería crearnos el siguiente conjunto de archivos:



Cuando los ejecutamos cabe la posibilidad que nos solicite una verificación de certificado SSL que le damos que “Si”. En caso de funcionar todo correctamente deberíamos ver esto en nuestro navegador:



En Program.cs vamos a agregar contenido para que a futuro no tengamos problemas:

Arriba de todo agregan los siguientes using (puede que se los agregue automáticamente):

```
using Microsoft.OpenApi.Models;
using System.Reflection;
```

Después del builder vamos a agregar un control de las reglas CORS, esto lo hacemos porque a veces cuando consumimos desde otras aplicaciones podemos llegar a tener problemas:

```
#region CorsRules

var CorsRules = "CorsRules";
builder.Services.AddCors(options =>
{
    options.AddPolicy(name: CorsRules,
        builder =>
        {
            builder.AllowAnyOrigin()
            .AllowAnyHeader()
            .AllowAnyMethod();
        });

});
#endregion
```

Básicamente este conjunto de reglas las vamos a usar más adelante en el código si tenemos algún inconveniente.

Antes del var app sino lo tiene vamos a agregar lo siguiente:

```
builder.Services.AddSwaggerGen(o =>
{
    o.SwaggerDoc("v1",
        new OpenApiInfo
        {
            Title = "Prueba API",
```

```
            Description = "Una aplicacion simple para mostrar el funcionamiento de las
APIs",
            Version = "v1",
            TermsOfService = null,
            Contact = new OpenApiContact
            {
                // Check for optional parameters
            },
            License = new OpenApiLicense
            {
                // Optional Example
                // Name = "Proprietary",
                // Url = new Uri("https://someURLToLicenseInfo.com")
            }
        });
    o.IncludeXmlComments(Path.Combine(AppContext.BaseDirectory,
     $"{Assembly.GetExecutingAssembly().GetName().Name}.xml"));
});

var app = builder.Build();
```

Lo más importante es agregar “o.IncludeXmlComments” porque va a leer los comentarios que hagamos en código sobre las funciones y respuestas (Los comentarios con “///”).

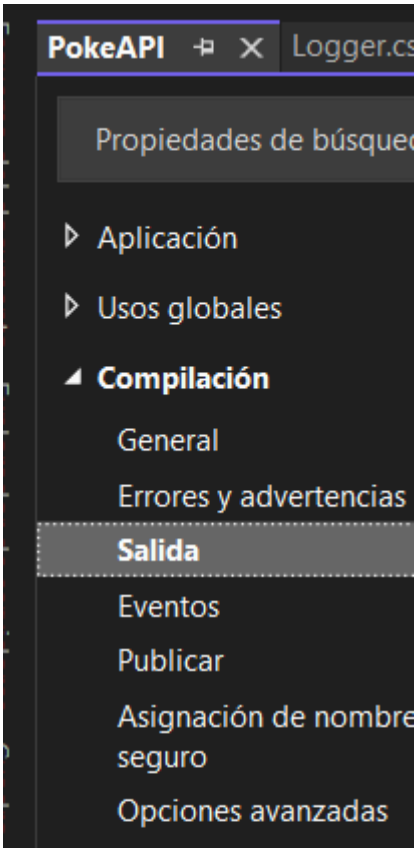
Debemos agregar ademas:

```
// Configure the HTTP request pipeline.
if (app.Environment.IsDevelopment())
{
    app.UseSwagger();
    app.UseSwaggerUI(c =>
    {
        c.SwaggerEndpoint("/swagger/v1/swagger.json", "Service Manager API V1");
    });
}

app.UseCors();
```

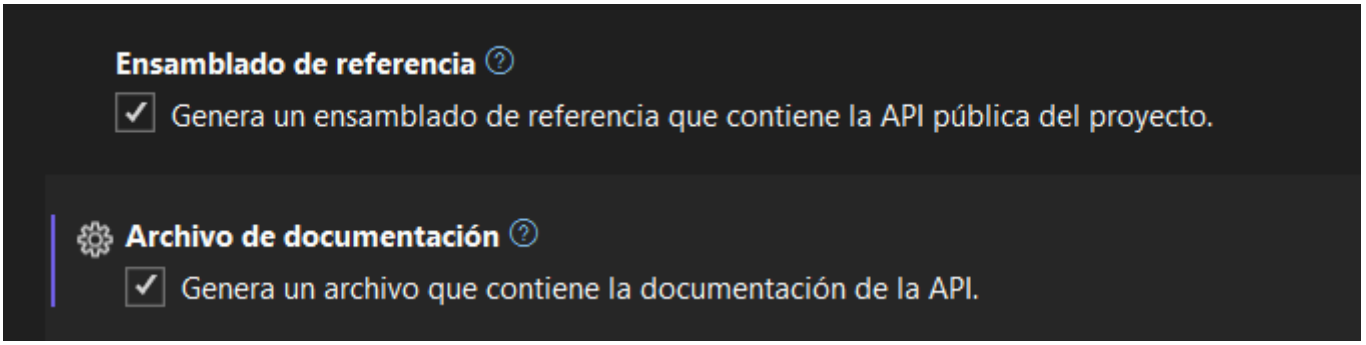
Finalmente lo último que voy a configurar es sobre el proyecto que tenemos hacemos:

Click derecho ➔ Propiedades

Nos aparece:



Y en salida activamos lo siguiente:



Es para que también realice documentación en base a XML.

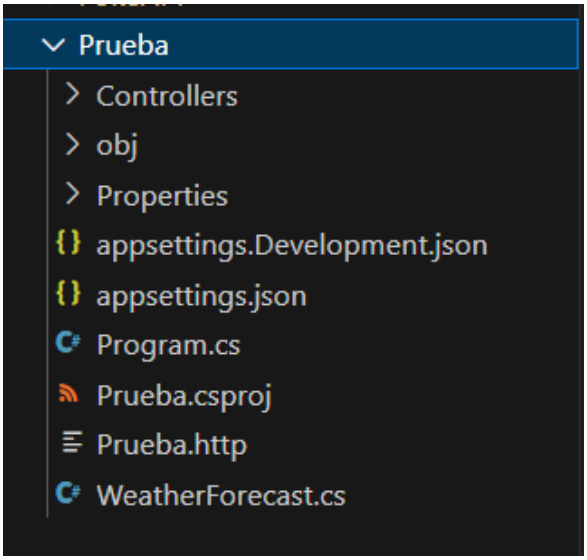
Visual Code

Iniciamos Code, abrimos una nueva consola, utilizamos el comando “cd” para acceder a la carpeta que deseamos crear el proyecto. Una vez situados ahí ejecutamos el siguiente código:

dotnet new webapi --use-controllers -o [Nombre Proyecto]

Obviamente saquen los “[ ]” y pongan el nombre que ustedes deseen al proyecto.

Deberían ver lo siguiente:



En su archivo csproj deben agregar las siguientes referencias

```
<PackageReference Include="Swashbuckle.AspNetCore" Version="8.1.1" />
<PackageReference Include="Swashbuckle.AspNetCore.SwaggerUI" Version="8.1.1" />
```

Estas son las versiones más recientes de estos paquetes. Después su archivo Program.cs les debe quedar de la siguiente manera:

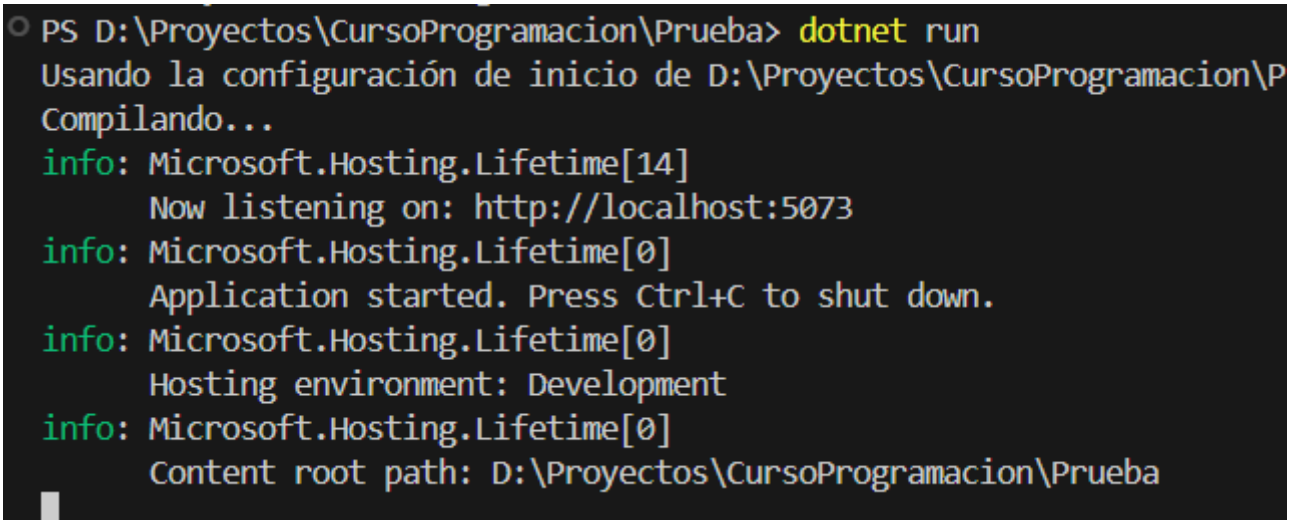
```
using Microsoft.OpenApi.Models;
using System.Reflection;

var builder = WebApplication.CreateBuilder(args);

// Add services to the container.

builder.Services.AddControllers();
// Learn more about configuring OpenAPI at https://aka.ms/aspnet/openapi
builder.Services.AddOpenApi();
builder.Services.AddEndpointsApiExplorer();
builder.Services.AddSwaggerGen(o =>
{
    o.SwaggerDoc("v1",
        new OpenApiInfo
        {
            Title = "API Prueba",
            Description = "Una aplicacion simple para mostrar el funcionamiento de
las APIs",
            Version = "v1",
            TermsOfService = null,
            Contact = new OpenApiContact
            {
```

```
                // Check for optional parameters
            },
            License = new OpenApiLicense
            {
                // Optional Example
                // Name = "Proprietary",
                // Url = new Uri("https://someURLToLicenseInfo.com")
            }
        });
});

var app = builder.Build();

// Configure the HTTP request pipeline.
if (app.Environment.IsDevelopment())
{
    app.MapOpenApi();
    app.UseSwagger();
    app.UseSwaggerUI();

}

app.UseHttpsRedirection();
app.UseAuthorization();
app.MapControllers();
app.Run();
```

Van hasta el proyecto que crearon y lo ejecutan:

```
cd [Nombre Proyecto]
dotnet run
```

Les debería crear la carpeta “bin” que es su código compilado y decirles lo siguiente en consola:



Deben ingresar a ese localhost, pueden hacerlo si le hacen CTRL+Click o lo escriben en el navegador y agregan “/swagger/index.html” .

Una aclaración que hago es que no encontré la manera de hacer funcionar la línea: *“o.IncludeXmlComments(Path.Combine(AppContext.BaseDirectory,$"{Assembly.GetExecutingAssembly().GetName().Name}.xml"));”*

Por lo que ya no se verán los comentarios XML que realicemos en nuestros endpoints.

Consideraciones

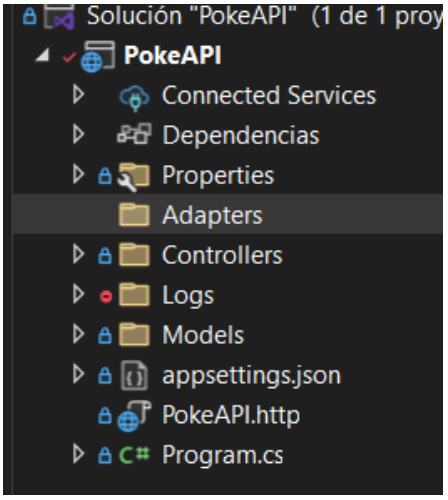
Estructura

En esta parte de la unidad voy a estar trabajando con lo que va a ser la base del Trabajo Práctico que va a ser la materia. No es necesario copiar todo lo que van a ver ya que deberían tener acceso al repositorio en GitHub (Sino lo solicitan vía mail) donde en la rama “Inicio API” podrán clonar para desarrollar ustedes. Recomiendo desde el principio crear su propia rama mediante:

```
git clone [repositorio]
git pull InicioAPI
git branch [legajo]
git checkout branch [legajo]
```

Si no poseen legajo utilizan alguna combinación entre sus iniciales y fecha de nacimiento que los identifique del resto. (EJ: GRBL99).

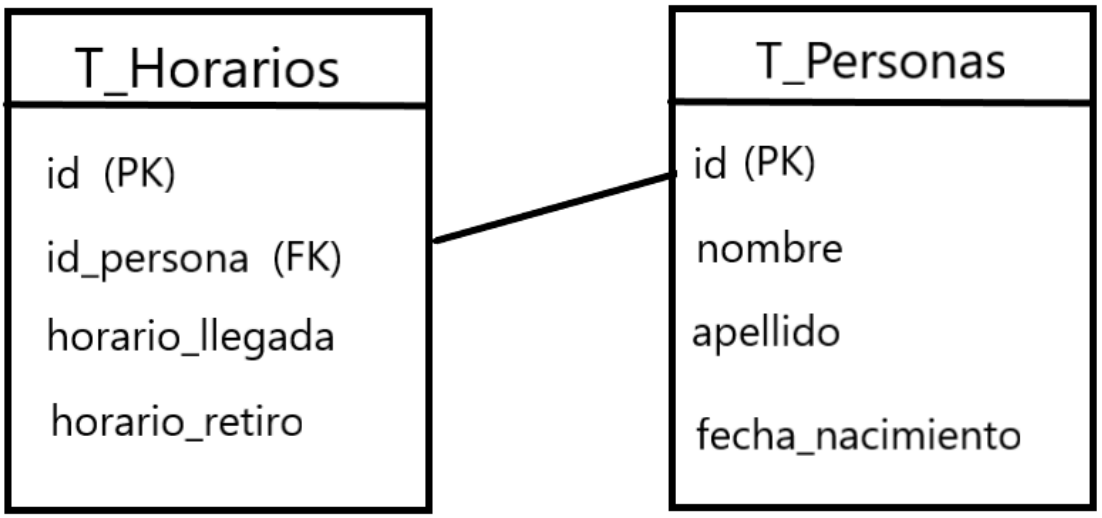
Para empezar lo que voy a hacer es darle la siguiente estructura:



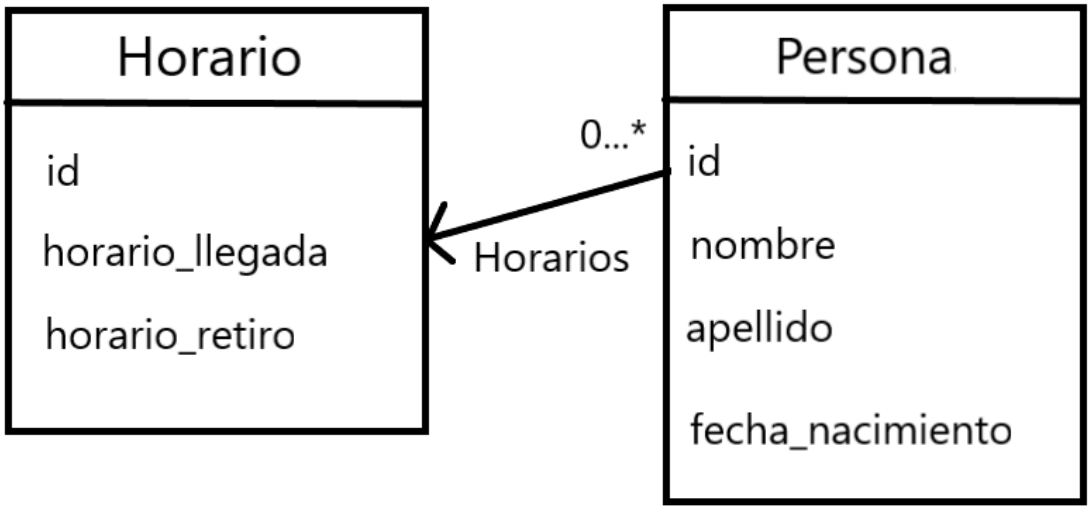
En Adapters se va a encontrar nuestros “Adaptadores” para recuperar información, en este caso vamos a tener uno para SQL que nos permita recuperar de una base de datos. La carpeta Controllers es donde nosotros vamos a desarrollar los endpoints y se van a bases en los Models que vamos a utilizar. Tenemos también una carpeta Logs que dentro va a contener una clase Logger.cs que voy a explicar a continuación. Finalmente tenemos lo que se llama Models que van a ser nuestras clases que vamos a devolver al usuario como también recibir.

Desde este punto surge el inconveniente que muchas veces el diagrama de clases es diferente a como lo guardamos en la base de datos. Les pongo un ejemplo:

En la base de datos nos encontramos con la siguiente estructura para almacenar los horarios de una persona.



Pero en nuestro diagrama de clases nos encontramos lo siguiente:



Como verán la direccionalidad con respecto a las tabla se “invierte” mientras que este puntero que teníamos desde horarios empieza a ser un listado dentro de la clase “Persona”. A nivel código el reflejo de la base de datos lo llamamos model y quedaria asi:

```
public class Horario
{
    public int id { get; set; }
    public int id_persona { get; set; }
    public DateTime horario_ingreso { get; set; }
    public DateTime horario_retiro { get; set; }
}

public class Persona
{
    public int id {  get; set; }
    public string nombre { get; set; }
    public string apellido { get; set; }
    public DateTime fecha_nacimiento { get; set; }
}
```

Mientras que el reflejo del diagrama de clases se denomina entidad y se veria asi:

```
public class Horario
{
    public int id { get; set; }
    public DateTime horario_ingreso { get; set; }
    public DateTime horario_retiro { get; set; }
}

public class Persona
```

```
{
    public int id {  get; set; }
    public string nombre { get; set; }
    public string apellido { get; set; }
    public DateTime fecha_nacimiento { get; set; }
    public List<Horario> horarios { get; set; }
}
```

En este sentido para pasarlo a objeto tenemos las siguientes opciones:

1. Cuando se realice una consulta nosotros en la API vamos a consultar las diferentes tablas de manera separada mediante Models y después armar las Entidades correspondientes.
2. En la API vamos a consultar mediante una vista o procedimiento que ya nos de la información de manera compacta para que nosotros generemos la entidad de manera directa.
3. Vamos a devolver el Models directamente y que sea trabajo del BackEnd consultar los endpoints para armar las entidades.

Por ahora vamos a trabajar con la tercera opción y cuando lleguemos a WPF les mostraré cómo se realiza el pase de modelos relacionales a modelos de objetos.

Log de Errores

Algo muy necesario en todas las aplicaciones es el registro de los problemas, alerta o cualquier evento que sea relevante para realizar un seguimiento de la aplicación. En este caso para la API que vamos a desarrollar solo vamos a realizar un par de registros en caso de excepciones y queda en ustedes si quieren expandir sobre esto.

En cuanto a cómo lo vamos a hacer ocurre que .NET en el pasado tuve muchos inconvenientes para activar el log siguiendo instrucciones de internet. Así que tanto para este tema como para el que viene lo que hice fue crear clases que realicen esto y expandirlo de acuerdo a las necesidades del proyecto. Desde el principio aclaro que voy a darles una versión reducida de esto para no complicarnos, como también decirles que funciona únicamente para Windows ya que en las rutas son con “\” y en servidores como Ubuntu Server es con “/”. Otro detalle que también se encuentran en el ambiente laboral es que estos datos se busca ocultarlos o dejarlos fuera del alcance del usuario final y de la mirada de los curiosos por motivos de seguridad.

Les dejo el código:

```
namespace PokeAPI.Logs
{
    /// <summary>
    /// Esta clase esta creada con el proposito de almacenar mensajes en
```

```
formato .txt dentro de la carpeta Logs
    /// </summary>
    public class Logger
    {
        /// <summary>
        /// Es el directorio donde vamos a almacenar la informacion. Recupera
desde donde se esta ejecutando actualmente
        /// </summary>
        private static string DirectoryPath = Environment.CurrentDirectory +
"\\Logs";
        /// <summary>
        /// Constructor vacio.
        /// </summary>
        public Logger(){}
        /// <summary>
        /// Registra un error en la carpeta "Registros"
        /// </summary>
        /// <param name="ex">La excepcion que provoco el error</param>
        /// <param name="error">El nombre del error</param>
        public static void RegistrarERROR(Exception ex, string error)
        {
            //Prueba en Windows
            DateTime actual = DateTime.Now;
            string fileName = DirectoryPath + "\\Registros\\" +
actual.ToString("yyyy-MM-dd") + ".txt";
            //Sino existe crea la carpeta para registrar errores
            if (!Directory.Exists(DirectoryPath + "\\Registros"))
Directory.CreateDirectory(DirectoryPath + "\\Registros");

            using (var sw = new StreamWriter(fileName, true))
            {
                sw.WriteLine(actual + ", ERROR: " + error + ", MESSAGE
EXCEPTION: " + ex.Message);
            }
        }
        /// <summary>
        /// Registra una Alerta ante situaciones inesperadas pero que no
provocan el fallo de la aplicacion
        /// </summary>
        /// <param name="origen">Desde que parte del programa ocurrio</param>
        /// <param name="comportamiento">Cual fue el comportamiento
detectado</param>
        public static void RegistrarAnomalia(string origen, string
```

```
comportamiento)
        {
            DateTime actual = DateTime.Now;
            string fileName = DirectoryPath + "\\Alertas\\" +
actual.ToString("yyyy-MM-dd") + ".txt";

            using (var sw = new StreamWriter(fileName, true))
            {
                sw.WriteLine(actual + ", ANOMALIA: " + comportamiento + ",
DESDE EL HOST: " + origen);
            }
        }
    }
}
```

Ambiente

Es importante considerar que no siempre vamos a estar corriendo el programa en el mismo dispositivo y que inevitablemente si queremos levantarlo en diversos servidores debemos tener un control sobre los mismos. Lo que más nos importa en esta situación es la cadena de conexión, porque cuando desarrollamos (Ambiente DESA) tenemos una base local que capaz está en el mismo dispositivo. La mayoría de las empresas poseen al menos un ambiente (pueden tener más) **antes** de lo que se denomina producción (Ambiente TEST o PrePROD) por lo que también debemos considerar que va a tener otra cadena de conexión y otra ruta para registrar los errores. Finalmente en la vida real tendrá otro servidor, con otros datos que son más sensibles y que queremos mantener su integridad (Ambiente PROD). Si bien en este curso lo único que vamos a cambiar entre los ambientes va a ser la cadena de conexión, les voy a explicar la forma de almacenarlo y que lo puedan cambiar para hacer un “despliegue”.

En este tipo de APIs se suele utilizar lo que se llama **appsettings.json** el cual tiene una estructura genérica con algunos campos que se suelen pedir como “AllowHosts : ‘*’ ”. Dentro de este archivo JSON podemos agregar las variables que nosotros estemos interesados en cambiar, por ejemplo si permitimos el log de errores, si ocultamos los datos de los registros .txt, si permitimos conexiones desde ciertos equipos o básicamente cualquier cosa que pueda cambiar entre TEST y PROD. Este archivo en proyectos de bajo riesgo suele estar visible como cualquier otro ya que también queremos poder cambiarlos desde nuestro sistema de archivos por si ocurre, por ejemplo, un cambio de ubicación de la base de datos.

Desde VS2022 se ofrecen muchas opciones que sinceramente antes funcionaban pero con los cambios que a veces realizan en el framework no siempre se mantienen a lo largo del tiempo. Es por eso que adopte la idea de hacer una clase lectora que busque este appsettings y lo lea dejandonos llevar a las variables desde el código. Esta clase inevitablemente va a tener que cambiar de acuerdo a

las necesidades de nuestro proyecto, por lo que les dejo esta versión inicial para que vean como funciona.

```
{
  "AllowedHosts": "*",
  "Env": "TEST",
  "ConnectionStrings": {
    "TEST": "Data Source",
    "DESA": "",
    "PROD": ""
  }
}
```

Más allá de este archivo personalizable también existe el launchappsettings.json que controla variables de entorno más generales y que pueden ser tomadas desde fuera de la aplicación. Son esenciales a la hora de utilizar despliegues automatizados con tecnologías como Docker.

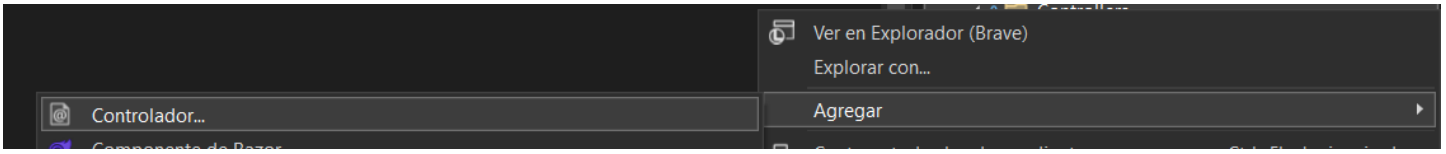
```
namespace PokeAPI.Adapters
{
    /// <summary>
    /// La clase lectora de appsettings.json
    /// </summary>
    public class SettingsReader
    {
        /// <summary>
        /// Que Host permite
        /// </summary>
        public string AllowedHosts = string.Empty;
        /// <summary>
        /// Que entonrno tenemos
        /// </summary>
        public string Env = string.Empty;
        /// <summary>
        /// Las cadenas de conexion disponibles
        /// </summary>
        public Dictionary<string, string> ConnectionStrings = new();
        /// <summary>
        /// Constructor Vacio
        /// </summary>
        public SettingsReader()
        {
        }
```

```
        /// <summary>
        /// Metodo estatico que busca el AppSettings
        /// </summary>
        /// <returns>La clase con los datos del appsettings.json</returns>
        public static SettingsReader GetAppSettings()
        {
            try
            {
                //Va buscar desde donde lo estamos ejecutando
                string file = Directory.GetCurrentDirectory() +
"\\appsettings.json";
                //Va a leer el archivo y convierte en una cadena
                using StreamReader reader = new(file);
                //Lo lee y convierte en String
                var json = reader.ReadToEnd();
                //Prueba convirtiendolo a la clase que creamos
                return JsonConvert.DeserializeObject<SettingsReader>(json) ??
new();
            }
            catch (Exception ex)
            {
                 //Registro el error
                Logger.RegistrarERROR(ex,"Error al recuperar el
appsettings.json");
                //Si falla lo crea de manera generica
                return new();
            }

        }
    }
}
```

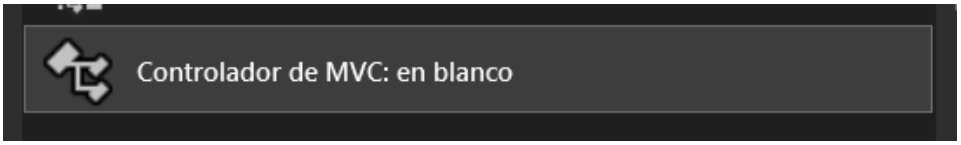
Controladores

Creación

Lo primero que tenemos que hacer es crear nuestro controlador:



Elegimos la siguiente opción:




```
namespace PokeAPI.Controllers
{
    0 referencias
    public class HomeController : Controller
    {
        0 referencias
        public IActionResult Index()
        {
            return View();
        }
    }
}
```

Les muestro de ejemplo el PokemonController:

```
/// <summary>
/// Este controlador esta en su version inicial y es para que tengan referencia
de como funcionan los endpoints
/// </summary>
[EnableCors("CorsRules")]
[Route("api/[controller]/[action]")]
[ApiController]
public class PokemonController : Controller
{

        [HttpGet]
        [ActionName("ObtenerPokemones")]
        public async Task<ActionResult<IEnumerable<Pokemon>>>ObtenerPokemones()
        {
            return Ok();
        }

}
```

Lo primero que agregamos es para no tener problema con el CORS, después es como queremos que organice los endpoints nuestro controlador. En lo personal me gusta organizarlo de esta forma, lo que encuentra entre "[ ]" son variables reconocidas por el compilador. En el caso del endpoint ObtenerPokemones se vería lo siguiente:

- **URL/api/Pokemon/ObtenerPokemones**

La segunda parte "APIController" es para que reconozca a este controlador como parte de una API y le de ese trato a la hora de ensamblarlo. Esto se agrega porque en ocasiones puede fallar algunas funciones del programa.

Endpoints

Códigos de Respuesta

A la hora de desarrollar endpoints tenemos que entender que códigos pueden devolver, es decir, cuales son las respuestas posibles que tenemos desde nuestra aplicación. Esto nos permite armar un flujo de información y como se ve en el siguiente punto una documentación que pueda ser consultada por otros usuarios. Desde el framework que utilizamos se nos ofrece un conjunto de códigos creados para facilitar la respuesta por parte del desarrollador. Aclaro que no tiene la totalidad de los mismos como respuestas así que en caso que deseemos usar uno no contemplado debemos realizarlo de otra manera.

Código 200 (OK)

Este código lo utilizó para indicar que la consulta o procedimiento fue realizado con éxito. Puedo llegar a devolver un mensaje, objeto o listado de objetos.

Se puede usar con estas opciones:

```
return Ok();
return Ok("Mensaje por nosotros");
return Ok(new List<object>());
```

Código 201 (Created)

Este código es para informar al usuario que el objeto que envió en su solicitud fue creado exitosamente. Personalmente yo devuelvo el objeto porque así pueden ver el ID que le asignamos.

```
return Created();
return Created("URI",new object());
return CreatedAtAction("Accion",new object())
```

Código 202 (Accepted)

Significa que la solicitud que nos enviaron la recibimos correctamente pero el resultado puede demorar. Con este mensaje le informamos que se está procesando pero no se garantiza que se complete y que tampoco se informe.

```
return Accepted();
return Accepted(new object());
return Accepted("",new());
```

Código 204 (NoContent)

Significa que la solicitud se solicitó exitosamente pero no existe ningún resultado que cumpla las condiciones. Este es diferente a 404 el cual no encontramos el objeto ya que por ejemplo puede ser que nos solicitan por objetos que cumplan “x” condición. Si bien la tabla y los registros existen, la condición no se cumple. También lo podemos usar cuando ocurre una acción de DELETE porque estamos avisando que el registro no existe más.

```
return NoContent();
```

Código 400 (Bad Request)

Tuvimos un error a la hora de realizar nuestra solicitud, podemos devolverlo con un mensaje de error o un objeto que deseemos.

```
return BadRequest();
return BadRequest("Error");
```

Código 401 (Unauthorized)

Quiere decir que el usuario no presentó las credenciales o son incorrectas por ende no puede acceder a ningún tipo de información. Este codigo tambien se manda si nosotros configuramos la seguridad y ponemos arriba de nuestro endpoint la etiqueta “[Authorize]”

```
return Unauthorized();
return Unauthorized("Motivos");
```

Codigo 403 (Forbidden)

Significa que nuestro usuario está bien logueado pero no tiene permisos sobre los recursos los cuales está solicitando manipulación.

```
return Forbid();
```

Código 404 (Not Found)

El recurso que estamos solicitando no existe, puede ser que el endpoint no exista o el recurso que llamamos del mismo no existe más.

```
return NotFound();
return NotFound("");
```

Código 409 (Conflict)

Ocurre cuando la petición genera algún tipo de conflicto en nuestro servidor o base de datos.

```
return Conflict();
return Conflict("Mensaje");
```

Código 500 (Internal Server Error)

Este código no podemos enviarlo nosotros pero ocurre cada vez que tengamos una excepcion no controlada. Por ejemplo en el caso de CORS cuando me notifica el error utiliza este código aunque yo no lo haya implementado en mi endpoint.

Documentación y visualización en swagger

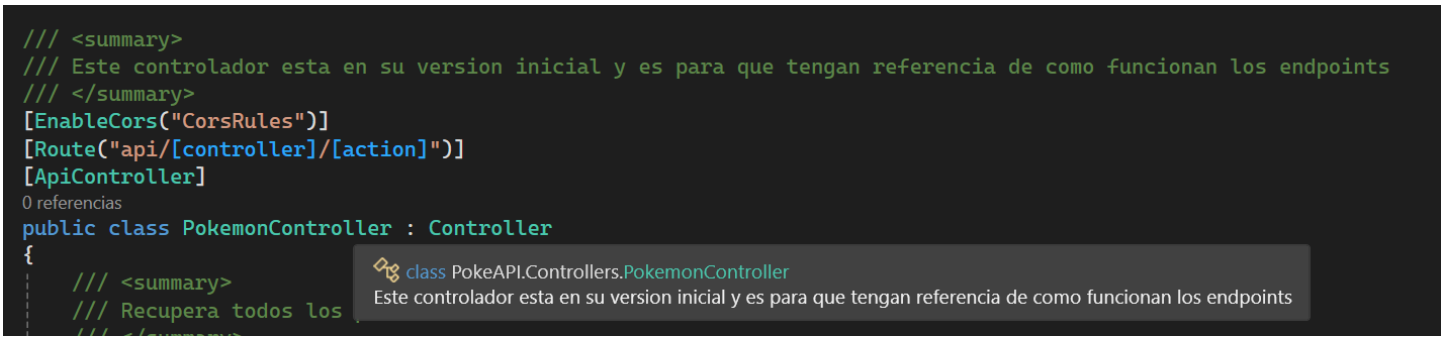
Introducción

La documentación es una parte esencial del desarrollo no porque nos vaya ayudar en el momento sino lo que significa después cuando el código necesita una modificación o ocurre un error. Suele pasar más de lo que me gustaría decir que cuando estamos trabajando nos toca revisar código

sin documentación que es desastroso. Esto hace la tarea complicada que podría haber sido solucionada si se escribiera algún tipo de referencia.

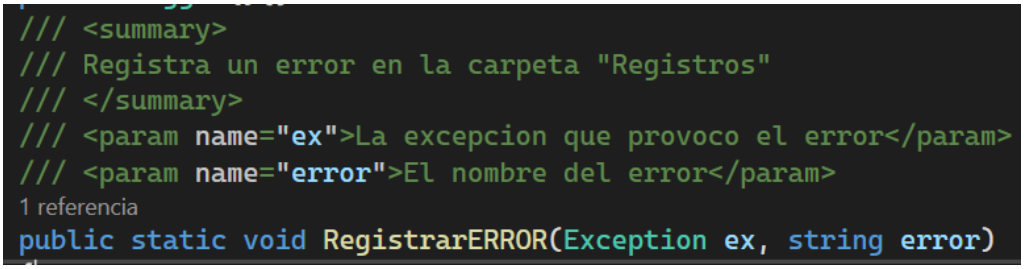
Además de los comentarios (“//”) también tenemos lo que son los textos XML (“///”) que son reconocidos por el compilador y se vuelven parte de las funciones y controladores que estamos creando.

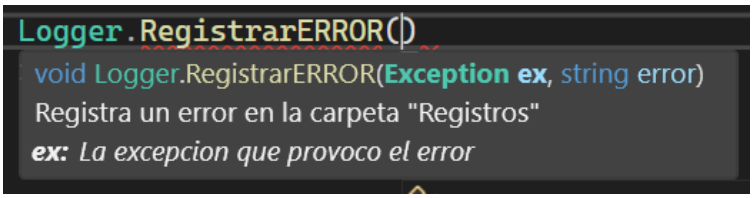
Por mostrar un ejemplo:



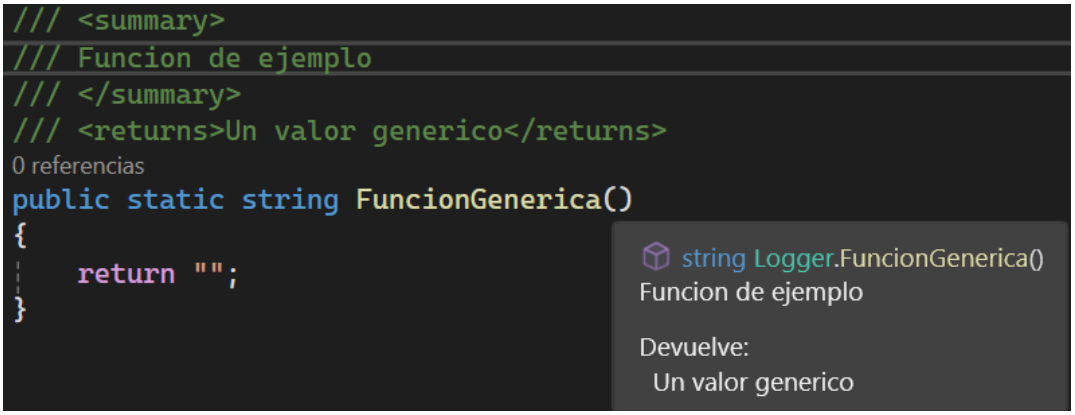
Vemos lo que está escrito en el código y vemos cuando pasamos el cursor por arriba de la clase “PokemonController” que la misma me muestra el mensaje personalizado que cree.

En las funciones además nos da la posibilidad de describir los parámetros y lo que queremos devolver:





Con un return:



Documentar en Swagger

Swagger es un conjunto de herramientas que cuando usamos VS2022 el mismo viene integrado lo que nos permite ver nuestra API mediante una interfaz de usuario desde nuestro navegador. Capaz un desarrollador pueda abrir una consola y utilizar el comando “curl” para realizar las pruebas de los endpoints pero en muchas empresas este trabajo es realizado por un equipo de QA que tienen que verificar que los valores y parámetros sean correctos. Es por eso que Swagger nos permite darle una interfaz agradable para estos usuarios como también diseñar documentación en base a esto.

Si recuerdan en la primera unidad les hice agregar código en el archivo Program.cs que involucraba los XML contents. Ese fragmento permite al compilador reconocer estos comentarios y dejarlos que los muestre una vez se abra la aplicación en el navegador.

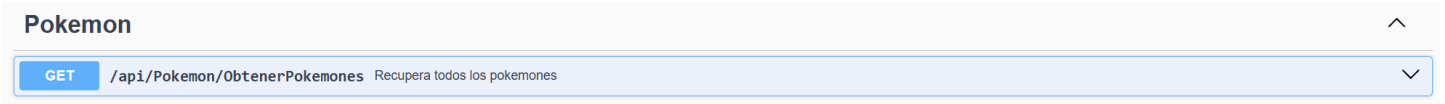
Para utilizar swagger van a ver que en las capturas de pantallas existe un botón que se llama **“Try It Out”**, de esta manera y completando los campos que nos hace falta podremos probar si está funcionando correctamente nuestro endpoint.

Les voy a dejar un ejemplo de como he documentado en el pasado y que consideraciones hago. Está demás aclarar que no es perfecto como lo realizó y mucho menos completo.

```
/// <summary>
/// Recupera todos los pokemones
/// </summary>
/// <returns>Todos los pokemones como listado</returns>
/// <remarks>
/// Ejemplo de uso:
///
///     GET /api/Pokemon/ObtenerPokemones/
///
///     RESPONSE:
///     [
///         {
///             "id": 0,
///             "name": "string"
///         }
///     ]
///
/// </remarks>
/// <response code="200" >Devuelve todos los pokemones </response>
/// <response code="204" >No se encontro ningun pokemon </response>
/// <response code="400" >Ocurre un error en la consulta </response>
/// <response code="409" >Ocurre un error en el procedimiento/vista de la base de datos </response>
```

```
/// <response code="500" >Ocurre un error en la API o en el Servidor no documentada </response>
[HttpGet]
[ActionName("ObtenerPokemones")]
[ProducesResponseType(typeof(IEnumerable<Pokemon>), StatusCodes.Status200OK)]
[ProducesResponseType(StatusCodes.Status204NoContent)]
[ProducesResponseType(StatusCodes.Status400BadRequest)]
[ProducesResponseType(StatusCodes.Status409Conflict)]
[ProducesResponseType(StatusCodes.Status500InternalServerError)]
public async Task<ActionResult<IEnumerable<Pokemon>>> ObtenerPokemones()
{
    return Ok();
}
```

Sin expandir:



Dentro del endpoint veremos lo siguiente:

A continuación cargue algunos datos a mano dentro del código para que vean cómo sería la respuesta pero con información:

Les muestro un ejemplo con un POST para que también vean los parámetros y lo que podemos solicitar como BODY.

```
/// <summary>
/// Cargar un Pokemon
/// </summary>
/// <remarks>
/// Ejemplo de uso:
///
///     POST /api/Pokemon/CargarPokemon/{id_pokemon}
///
```

```
///     BODY:
///     {
///         "id": 0,
///         "name": "string"
///     }
///
///     RESPONSE:
///     {
///         "id": 0,
///         "name": "string"
///     }
/// </remarks>
/// <param name="id_pokemon"></param>
/// <param name="poke"></param>
/// <returns></returns>
[HttpPost("{id_pokemon}")]
[ActionName("CargarPokemon")]
[ProducesResponseType(typeof(Pokemon),StatusCodes.Status201Created)]
public ActionResult<Pokemon> CargarPokemon([AsParameters] int id_pokemon,
[FromBody] Pokemon poke)
{
    return CreatedAtAction("Creado con éxito ", poke);
}
```

(Lo realizo asi para que también vean que los endpoints pueden ser síncronos)

En los parámetros de la función verán que agregue dos formas de ingreso de información diferentes, de esta forma evito los conflictos o excepciones no controladas.

Previo al endpoint

Principal en todos los endpoints

Lo primero que tenemos que definir es el método HTTP que vamos a utilizar para llamarlo, solo acepta un solo tipo.
Deben estar arriba de la función:

```
[HttpGet]
[HttpPut]
[HttpPost]
[HttpDelete]
```

Si queremos que reciba parametros en la URL debemos aclararlo en el método

```
[HttpGet("{id_pokemon}/{generacion}")]//Solicita 2 parametros
[HttpPut("{id_pokemon}")]
[HttpPost("{id_pokemon}")]
[HttpDelete("{id_pokemon}")]
```

A continuación debemos poner de la acción que se va a exponer para que lo utilicemos, personalmente intento que la función y el ActionName tengan el mismo nombre

```
[ActionName("ObtenerPokemones")]
```

Después tenemos los tipo de respuestas que puede dar, esto sirve para que lo contemple en la documentación. El más importante es el primero ya que notifica al usuario cual es la salida de nuestro endpoint en caso de éxito. “IEnumerable” es una interfaz que abarca a todas las colecciones nativas de C# como pueden ser listas, arrays, etc. La parte de “typeof” denota como es el objeto y con esto nos muestra en el swagger el formato JSON de la respuesta.

```
[ProducesResponseType(typeof(IEnumerable<Pokemon>), StatusCodes.Status200OK)]
[ProducesResponseType(StatusCodes.Status204NoContent)]
[ProducesResponseType(StatusCodes.Status400BadRequest)]
[ProducesResponseType(StatusCodes.Status409Conflict)]
[ProducesResponseType(StatusCodes.Status500InternalServerError)]
```

Finalmente tenemos la definición de la función del método en el controlador:

```
public async Task<ActionResult<IEnumerable<Pokemon>>> ObtenerPokemones()
```

Partamos de la base, el método debe ser **público** (public) si queremos que lo puedan usar desde fuera del controlador. Puede ser además asincrónico (async) porque es un requerimiento para los métodos que solicite el operador await. En nuestro caso podemos no usarlo por lo que también debemos remover del retorno el “Task” porque solo funciona en modo asincrónico.

```
public ActionResult<IEnumerable<Pokemon>> ObtenerPokemones()
```

El ActionResult de esta forma obliga que mandemos un código HTTP como respuesta o una colección de la clase “Pokemon”. También puede tratarse de un solo objeto de la clase o de un valor nativo de C#.

```
public ActionResult<Pokemon> ObtenerPokemones()
public ActionResult<int> ObtenerPokemones()
```

Además si sacamos el “ActionResult” nos va a dar error si intentamos mandar “Ok()” como respuesta válida.

Conexión a la base de datos

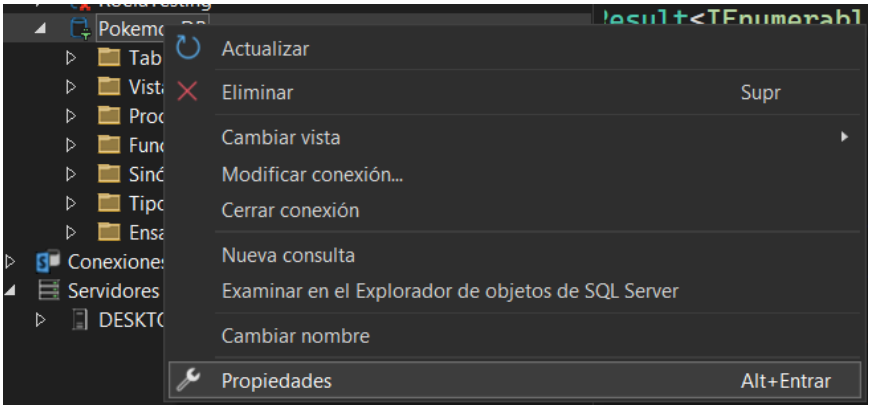
La conexión a la base de datos tiene muchos enfoques, en primera instancia y lo que he visto que se dicta para empezar es que nosotros desde el mismo endpoint abramos y cerremos la conexión a la base de datos con el siguiente código:

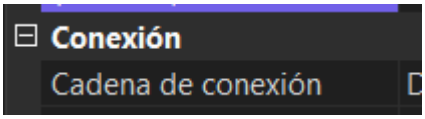
```
//Extracto del endpoint

DataTable respuesta = new();
string consulta = "SELECT * FROM Pokemones";//Consulta SQL

using SqlConnection conexionBase = new SqlConnection("DataSource");
using var comando = new SqlCommand(consulta, conexionBase);//Creamos el comando
en la base con la conexion
comando.CommandType = CommandType.Text;//Notificamos a la base que vamos a
enviar
SqlDataAdapter Adaptador = new(comando);
conexionBase.Open(); //Abre la conexion (Puede fallar)
Adaptador.Fill(respuesta); //Contacta la base y ejecuta el comando

//Linea super importante para que no quede la conexion abierta
conexionBase.Close();
//Resto del endpoint
```

Probablemente las clases SqlConnection, SqlCommand y SqlDataAdapter no les aparezcan como válidas. Haciendo doble click sobre las mismas les va a dar la sugerencia de instalar el paquete NuGet correspondiente.

Sino el NuGet que estoy utilizando en esta versión es:

La cadena de DataSource lo obtienen desde el explorador de servidores, click derecho en su base, propiedades y el campo es Cadena de Conexión.





Volviendo al endpoint, lo que estamos haciendo es generar una instancia de la conexión SQL, le asigna un comando (la consulta) y finalmente le decimos que nos devuelve un DataTable. Si bien este método funciona tiene varios problemas, el **primero** es que lo vamos a tener que pegar todas las veces que hagamos un endpoint y si queremos hacer una variación lo tenemos que cambiar en todos lados. El **segundo** problema es que realizamos una consulta (o cruda) a la base de datos lo cual si trabajamos en alguna empresa no va a suceder. **Tercer** problema es que si queremos cambiar la cadena de conexión vamos a tener que recompilar la aplicación, que puede ser un gran problema si estamos desplegando en un servidor porque tenemos que volver a pegar el ejecutable o DLL. **Finalmente** el último problema es que no existe ningún tipo de control de errores ni de registro ante cualquier inconveniente.

Para resolver todos estos inconvenientes realizó lo siguiente, primero en la carpeta adapters creó un archivo que voy a llamar “GeneralAdapterSQL” el cual va a unificar todas las interacciones en un solo archivo. Para evitar las consultas crudas les voy a mostrar un ejemplo con la consulta de una vista. Además voy agregar un bloque try-catch que sirve para evitar que se cierre el programa ante una excepción como también usar la clase Logger para registrar este error.

La vista en SQL:

```
CREATE VIEW MODULO_POKEMON_OBTENER_POKEMONES_COMPLETO
AS
--Le pongo completo porque podriamos hacer una vista por generación
     SELECT *
          FROM Pokemones
```

(Hay que ejecutarlo en una consulta para que lo cree en nuestra base de datos)

Codigo en C# para consultar una vista:

```
public DataTable EjecutarVista(string vista)
{
    //Es necesario ponerlo por fuera para poder usar el bloque finally
    using SqlConnection conexionBase = new SqlConnection("DataSource");
    DataTable respuesta = new();
    try
    {
        //Con esto recupera la informacion de la vista
        string consulta = "SELECT * FROM " + vista;
        using var comando = new SqlCommand(consulta, conexionBase);//Creamos el
comando en la base con la conexion
        comando.CommandType = CommandType.Text;//Notificamos a la base que
vamos a enviar
```

```
        SqlDataAdapter Adaptador = new(comando);
        conexionBase.Open(); //Abre la conexion (Puede fallar)
        Adaptador.Fill(respuesta); //Contacta la base y ejecuta el comando
    }
    catch (Exception ex)
    {
        //Registramos el error en nuestra carpeta
        Logger.RegistrarERROR(ex, "Error al consultar la vista: " + vista);
        //Esta parte es para devolver un codigo de error al endpoint
        respuesta.Columns.Add("RESULTADO");
        respuesta.Rows.Add("ERROR");
    }
    //Lo va a ejecutar no importa que parte del codigo realice
    finally
    {
        //Limpia cualquier cadena de conexion que tengamos
        SqlConnection.ClearAllPools();
        //Cierra la base de datos siempre
        conexionBase.Close();
    }
    return respuesta;
}
```

Para el tema del ambiente lo veremos más adelante así ya podemos pasar a la parte entretenida que es armar nuestro endpoint.

Nuestro Primer Endpoint

Lamento si parece que no llegábamos más a este punto pero me interesaba que tuvieran una buena estructura antes de realizar el primer endpoint para que en el futuro sea más simple. Vamos a comenzar con el endpoint más fácil de todos que es aquel que realiza una consulta a una vista y convierte los datos de una tabla en Models.

Voy a usar lo que tengo de antes pero para legibilidad va estar sin los comentarios XML.

```
[HttpGet]
[ActionName("ObtenerPokemones")]
public async Task<ActionResult<IEnumerable<Pokemon>>> ObtenerPokemones()
{
    return Ok();
}
```

Lo primero que vamos a hacer es crear la vista-procedimiento SQL que nos va a devolver la información, en mi caso ya se encuentra creado del paso anterior como:

```
"MODULO_POKEMON_OBTENER_POKEMONES_COMPLETO"
```

Vamos a llamar a nuestro “EjecutarVista” pasándole este nombre y la tabla que nos devuelva vamos a revisar que no haya generado ningún error.

```
GeneralAdapterSQL consultor = new();
DataTable respuesta =
consultor.EjecutarVista("MODULO_POKEMON_OBTENER_POKEMONES_COMPLETO");

if (respuesta.Rows.Count > 0)
{
    //Rows[0] es primera fila y Rows[0][0] es primera columna de la primera
fila.
    if (respuesta.Rows[0][0].ToString()?.Trim() == "ERROR")return Conflict();
    //Uso el codigo 409 para notificar que ocurrio un error en la consulta del
servidor
    else
    {
        //Creo el listado de pokemones:

    }
}
else return NoContent();//No ocurrio un error simplemente que no existen
pokemones cargados
```

Finalmente tenemos el siguiente gran paso que es convertir la información de tabla a objetos, desde mi perspectiva lo más fácil es generar un constructor que acepte las filas (DataRow) como valor válido para iniciar la clase.

```
/// <summary>
/// Esta clase representa un animal ficticio
/// </summary>
public class Pokemon
{
    /// <summary>
    /// Su numero en la pokedex
    /// </summary>
    public int id { get; set; }
```

```
    /// <summary>
    /// El nombre del pokemon
    /// </summary>
    public string nombre { get; set; }
    /// <summary>
    /// La altura del pokemon en metros
    /// </summary>
    public float altura { get; set; }
    /// <summary>
    /// El peso del pokemon en kilogramos
    /// </summary>
    public float peso { get; set; }
    /// <summary>
    /// La generacion a la cual pertenece
    /// </summary>
    public int generacion { get; set; }
    /// <summary>
    /// Su tipo primario (obligatorio)
    /// </summary>
    public int id_tipo_primario { get; set; }
    /// <summary>
    /// Su tipo secundario (opcional)
    /// </summary>
    public int? id_tipo_secundario { get; set; }
    /// <summary>
    /// Una fila perteneciente a una tabla
    /// </summary>
    /// <param name="fila">Un DataRow proveniente de un DataTable</param>
    public Pokemon(DataRow fila)
    {
//Los "??" significan que en caso de ser null o no existir deberia adoptar el
valor que le digamos
        this.id = int.Parse(fila["id"].ToString() ?? "0");
        this.nombre = fila["nombre"].ToString() ?? "ERROR";
        this.altura = float.Parse(fila["altura"].ToString() ?? "0");
        this.peso = float.Parse(fila["peso"].ToString() ?? "0");
        this.generacion = int.Parse(fila["generacion"].ToString() ?? "1");
        this.id_tipo_primario = int.Parse(fila["id_tipo_primario"].ToString()?? "0");

        string? secundario = fila["id_tipo_secundario"].ToString();
//Hago un control de nulidad diferente ya que no queremos que muestre un valor
predeterminado
        if (secundario != null && secundario.Trim() != "")
```

```
            this.id_tipo_secundario = int.Parse(secundario);
//Agrego el diferente a "" porque puede ocurrir que lo convierta a esta cadena
        else this.id_tipo_secundario = null;

    }
}
```

En nuestro endpoint lo deberíamos llamar de la siguiente forma:

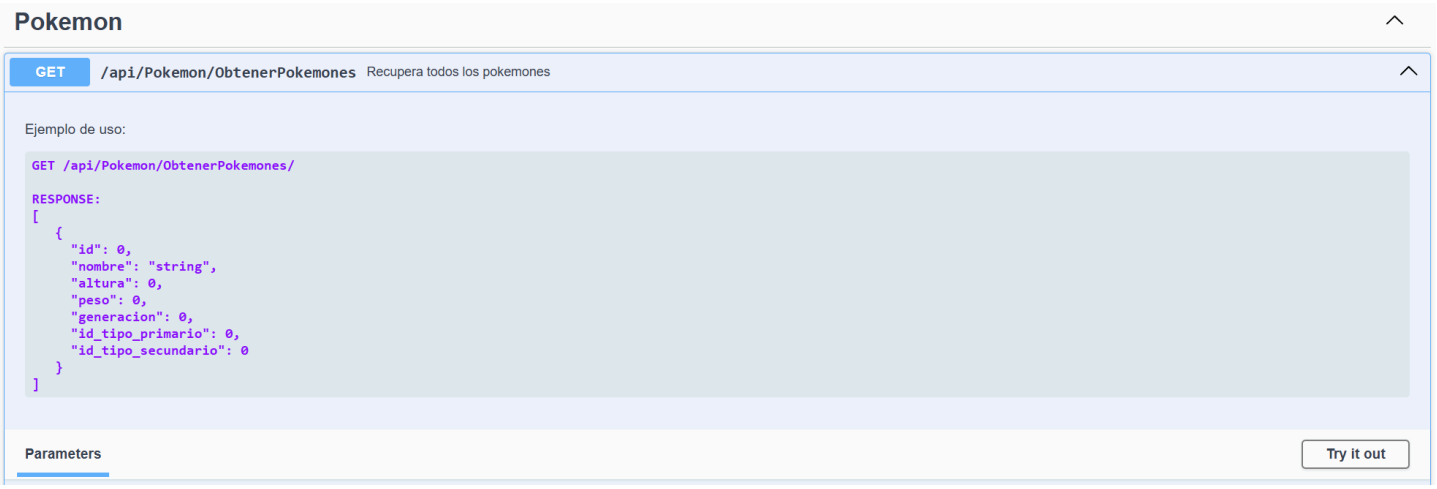
```
//Creo una nueva lista
List<Pokemon> listadoCompleto = new();
//Creo el listado de pokemones:
try
{
    //Recorro los registros de la fila
    foreach (DataRow registro in respuesta.Rows)
    {
        //Agrego todo al listado
        listadoCompleto.Add(new(registro));
    }
    //Devuelvo el listado completo con el codigo 200
    return Ok(listadoCompleto);
}
catch (Exception ex)
{
    Logger.RegistrarERROR(ex, "Imposible crear objeto, inconsistencia en los
datos");
    return Conflict("Ocurrio un error en la creacion de los datos");
}
```

Utilice un bloque try-catch porque puede pasar que realizamos mal la consulta, que después de una actualización en la base de datos borraron una columna (sería considerado pecado capital) o le cambiaron el nombre a la columna (Delito Federal).

Una vez ejecutada la API podremos probarlo usando curl:

```
curl -X 'GET' \
  'https://localhost:1984/api/Pokemon/ObtenerPokemones' \
  -H 'accept: text/plain'
```

O desde la interfaz swagger:



Respuesta esperada si tenemos la base inicial:


```
[
  {
    "id": 1,
    "nombre": "Bulbasaur",
    "altura": 0.7,
    "peso": 6.9,
    "generacion": 1,
    "id_tipo_primario": 13,
    "id_tipo_secundario": 15
  },
  {
    "id": 2,
    "nombre": "Ivysaur",
    "altura": 1,
    "peso": 13,
    "generacion": 1,
    "id_tipo_primario": 13,
    "id_tipo_secundario": 15
  },
  {
    "id": 3,
    "nombre": "Venusaur",
    "altura": 2,
    "peso": 100,
    "generacion": 1,
    "id_tipo_primario": 13,
    "id_tipo_secundario": 15
  },
  (...),
  {
    "id": 151,
    "nombre": "Mew",
    "altura": 0.4,
    "peso": 4,
    "generacion": 1,
    "id_tipo_primario": 7,
    "id_tipo_secundario": null
  }
]
```

ABMC

A configurar

Este término se refiere a **A**lta, **B**aja, **M**odificación, **C**onsulta y es el conjunto de operaciones fundamentales que se hacen con los datos. En general cuando usamos este término nos referimos a que realizar todas las operaciones sobre una clase para de esta forma tener una nocion de como replicarlo en toda la aplicacion. Como vamos a ver una vez completamos un ABMC su réplica suele ser lo más fácil de realizar en el resto de la aplicación.

En el caso de las API involucran las operaciones antes mencionadas que una vez que completamos las vamos a reutilizar en el resto de la aplicación. Para esta parte vamos a necesitar desarrollar procedimientos almacenados que tomen parámetros.

Primero una funcion de conversion de tipos de variable:

```
private static SqlDbType GetDBType(object variable)
{
        //A medida que fui programando fui modificando esta función por lo que
en algunas ramas no va a estar igual
    switch (variable.GetType().Name)
    {
        case "Int32":
            return SqlDbType.Int;
        case "String":
            return SqlDbType.VarChar;
        case "TimeOnly":
            return SqlDbType.Time;
        case "DateTime":
            return SqlDbType.Date;
        case "Single":
            return SqlDbType.Decimal;
        case "Decimal":
            return SqlDbType.Decimal;
        case "Float":
            return SqlDbType.Float;
        case "Bool":
            return SqlDbType.Bit;
        //Es muy probable que la base de datos pueda convertir de string a
cualquier tipo de variable no reconocida
```

```
        default:
            return SqlDbType.VarChar;
    }
}
```

Segundo como ejecutar el procedimiento:

```
public DataTable EjecutarProcedimiento(string procedimiento, Dictionary<string,
object> parametros)
{
    //Es necesario ponerlo por fuera para poder usar el bloque finally
    using SqlConnection conexionBase = new("Data Source=");
    DataTable respuesta = new();
    try
    {
        using var comando = new SqlCommand(procedimiento,
conexionBase);//Creamos el comando en la base con la conexion
                                                                    //Este
tipo de comando es para ejecutar un proceso almacenado por ende no necesita el
"SELECT"
        comando.CommandType = CommandType.StoredProcedure;

        //Nuestro adaptador con el procedimiento
        SqlDataAdapter Adaptador = new(comando);

        //Por cada parametro vamos a realizar el siguiente agregado:
        foreach (var item in parametros)
        {
            //Si es null o mandamos la palabra "NULL" le informa a la base de
datos de esto
            if (item.Value == null || item.Value.ToString()?.Trim() == "NULL")
            {
                comando.Parameters.AddWithValue(item.Key, DBNull.Value);
            }
            //El parametro se le asigna un tipo de valor y despues su valor
            else
            {
                comando.Parameters.Add(item.Key, GetDBType(item.Value));
                comando.Parameters[item.Key].Value = item.Value;
            }
        }
        conexionBase.Open();
```

```
        Adaptador.Fill(respuesta);
    }
    catch (Exception ex)
    {
        //Registramos el error en nuestra carpeta
        Logger.RegistrarERROR(ex, "Error al consultar el procedimiento: " +
procedimiento);
        //Esta parte es para devolver un codigo de error al endpoint
        respuesta.Columns.Add("RESULTADO");
        respuesta.Rows.Add("ERROR");
    }
    //Lo va a ejecutar no importa que parte del codigo realice
    finally
    {
        //Limpia cualquier cadena de conexion que tengamos
        SqlConnection.ClearAllPools();
        //Cierra la base de datos siempre
        conexionBase.Close();
    }
    return respuesta;
}
```

Además van a necesitar una función que cree para resolver la conversión de C# a sqlDbType. Generalmente no hace falta ya que usando el comando “AddWithValue()” y formateando correctamente la información no tenemos problemas

```
comando.Parameters.AddWithValue(Key,Value)
```

A veces utilizando este método tenemos problemas con la conversión de fechas o decimales. Antes de empezar a los métodos EjecutarVista y EjecutarProcedimiento los voy a modificar para poder cambiar la cadena de conexión dependiendo el tipo de equipo en el cual este:

appsettings.json:

```
{
  "AllowedHosts": "*",
  "Env": "TESTNOTE",
  "ConnectionStrings": {
    "TESTCASA": "Cadena1",
    "TESTNOTE": "Cadena2",
    "DESA": "Cadena3",
    "PROD": "Cadena4"
```

```
  }
}
```

No pongo todo el data source porque ocupa mucho lugar. En nuestro general adapter agregó los atributos y métodos:

```
/// <summary>
/// La cadena de conexion que esta usando actualmente nuestra API
/// </summary>
private static string? CadenaConexion = null;
/// <summary>
/// La configuracion que tenemos en nuestra API
/// </summary>
private static SettingsReader? Configuracion = null;
/// <summary>
/// Metodo que recupera la configuracion de nuestro appsettings.json
/// </summary>
private static SettingsReader ObtenerConfiguracion() =>
SettingsReader.GetAppSettings();
/// <summary>
/// Recupera la cadena de conexion desde nuestra configuracion
/// </summary>
private static void ObtenerCadenaConexion()
{
    if (CadenaConexion != null && CadenaConexion.Trim() != "") return;
    //Este if no tiene el else porque una vez que recupera la conexion deberia
continuar normalmente
    if (Configuracion == null) Configuracion = ObtenerConfiguracion();

    //En esta situacion siempre esta asignado algun valor a Configuracion

    //Comprueba que exista un valor de env y una clave para este entorno
    if (Configuracion.Env.Trim() != "" &&
Configuracion.ConnectionStrings.ContainsKey(Configuracion.Env))
        CadenaConexion = Configuracion.ConnectionStrings[Configuracion.Env];
    else CadenaConexion = null;
}
```

Finalmente en el comienzo de los métodos de vistas y procedimientos:

```
ObtenerCadenaConexion();
 //Es necesario ponerlo por fuera para poder usar el bloque finally
 using SqlConnection conexionBase = new(CadenaConexion);
```

La forma en que lo desarrolló es que todo el generalAdapterSQL va a tener un solo atributo estático con la cadena de conexión. Lo realizo de esta forma porque tenemos una sola fuente de información pero puede haber situaciones en que necesitemos más de una cadena SQL. En esos casos uno debería no guardarlo como atributo estático y en todo caso guardar las conexiones posibles en el appsettings recuperando el que corresponda utilizando alguna palabra clave.

Get

En vez de ver un listado como vimos en el primer endpoint vamos a realizar una consulta pero solicitando un parámetro.

Empezamos con el procedimiento SQL que solicita el parámetro “@id_pokemon”

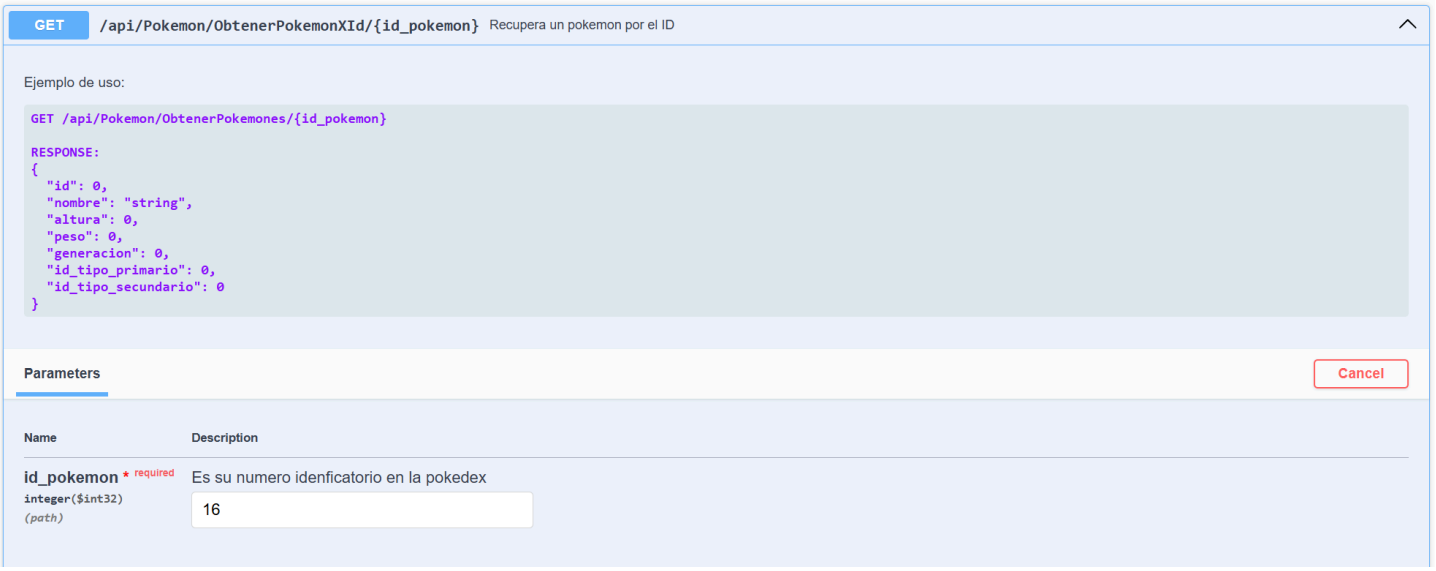
```
CREATE PROCEDURE MODULO_POKEMON_OBTENER_POKEMON_ID (@id_pokemon int)
AS
BEGIN
     SELECT *
          FROM Pokemones
               WHERE id = @id_pokemon
END
```

Para que nos reconozca que necesitamos mandar un parámetro tenemos que agregar en el HttpGet y en la función:

```
/// <param name="id_pokemon">Es su numero idenficatorio en la pokedex</param>
[HttpGet("{id_pokemon}")]
[ActionName("ObtenerPokemonXId")]
public ActionResult<Pokemon> ObtenerPokemonXId(int id_pokemon)
{
    //Creo la instancia para ejecutar el metodo
    GeneralAdapterSQL consultor = new();

    //Ejecuto la vista
    DataTable respuesta =
consultor.EjecutarProcedimiento("MODULO_POKEMON_OBTENER_POKEMON_ID",
        new() { { "@id_pokemon", id_pokemon } });
    //Verifico que tenga al menos un registro
    if (respuesta.Rows.Count > 0)
    {
        //Rows[0] es primera fila y Rows[0][0] es primera columna de la primera
```

```
fila.
        if (respuesta.Rows[0][0].ToString()?.Trim() == "ERROR") return
Conflict();
        //Uso el codigo 409 para notificar que ocurrio un error en la consulta
del servidor
        else
        {
            try
            {
                //Pokemon encontrado
                Pokemon busqueda = new(respuesta.Rows[0]);
                return Ok(busqueda);
            }
            catch (Exception ex)
            {
                Logger.RegistrarERROR(ex, "Imposible crear objeto,
inconsistencia en los datos");
                return Conflict("Ocurrio un error en la creacion de los
datos");
            }
        }
    }
    else return NoContent();
}
```

Desde Swagger



Resultado:

200

Response body


```
{
  "id": 16,
  "nombre": "Pidgey",
  "altura": 0.3,
  "peso": 1.8,
  "generacion": 1,
  "id_tipo_primario": 6,
  "id_tipo_secundario": 12
}
```


Put

Como mencioné anteriormente esta acción se refiere a la actualización de registros en la base de datos. Para empezar, algo que me sucedió en esta versión de .NET que no me había pasado, es necesario crear un constructor vacío para que pueda tomar el valor desde el body.

```
/// <summary>
/// Debemos crear un constructor sin parametros para cuando se manda desde el body la clase
/// </summary>
public Pokemon()
{
    this.id = -1;
    this.nombre = "";
    this.altura = 0;
    this.peso = 0;
    this.generacion = 0;
    this.id_tipo_primario = 0;
    this.id_tipo_secundario = null;
}
```

Para evitar cualquier tipo de inconveniente cree el constructor para que tenga datos en sus atributos y no sean nulos. El id = -1 es porque con eso evitó que ningún registro pueda ser actualizado o utilizado en la base ya que no acepta índices negativos.

En el procedimiento en la base de datos hacemos:

```
CREATE PROCEDURE MODULO_POKEMON_ACTUALIZAR_POKEMON(@id_pokemon int,
@nombre varchar(255),@altura decimal(18,2),@peso decimal(18,2),
@generacion int,@id_tipo_primario int, @id_tipo_secundario int NULL)
AS
BEGIN

     UPDATE Pokemones
```

```
            SET nombre = @nombre,
                  altura = @altura,
                  peso = @peso,
                  generacion = @generacion,
                  id_tipo_primario = @id_tipo_primario,
                  id_tipo_secundario = @id_tipo_secundario
                  WHERE id = @id_pokemon

      SELECT *
            FROM Pokemones
                  WHERE id = @id_pokemon

END
```

Devuelvo el registro para de esta forma controlar que se realizó un cambio, en este sentido si uno lo desea puede también devolver una tabla con la columna “RESPUESTA” y poner “OK” significando una modificación exitosa.

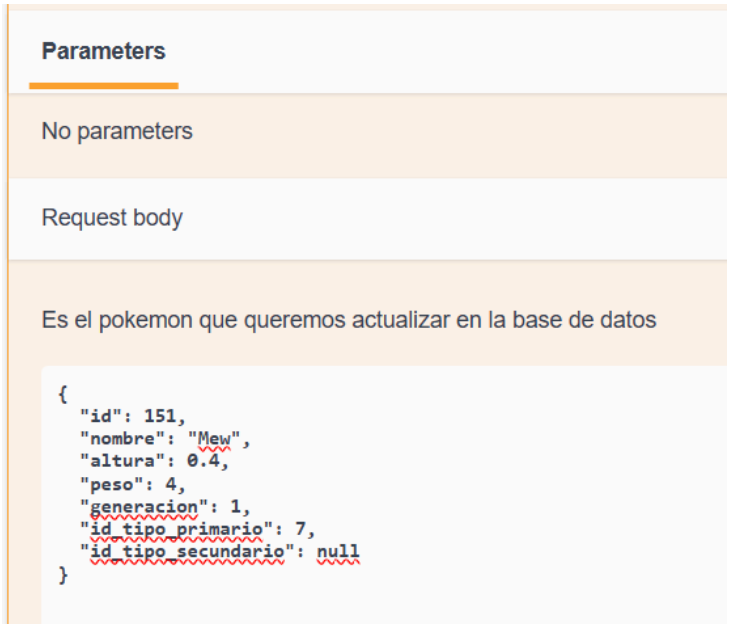
El endpoint tenemos que hacer lo siguiente a lo que venimos trabajando:

```
/// <param name="pokemonActualizado"> Es el pokemon que queremos actualizar en
la base de datos</param>
[HttpPut]
[ActionName("ActualizarPokemon")]
public ActionResult<Pokemon> ActualizarPokemon([FromBody] Pokemon
pokemonActualizado)
{
    //Creo la instancia para ejecutar el metodo
    GeneralAdapterSQL consultor = new();

    //Ejecuto el procedimiento de actualizar
    DataTable respuesta = consultor.EjecutarProcedimiento
        ("MODULO_POKEMON_ACTUALIZAR_POKEMON",
        new() { { "@id_pokemon", pokemonActualizado.id },
                  {"@nombre",pokemonActualizado.nombre },
                  {"@altura",pokemonActualizado.altura },
                  {"@peso",pokemonActualizado.peso },
                  {"@generacion",pokemonActualizado.generacion },
                  {"@id_tipo_primario",pokemonActualizado.id_tipo_primario },
                 {"@id_tipo_secundario",pokemonActualizado.id_tipo_secundario
},
        });
    //En la base por decision devuelto el registro modificado
```

```
    //Verifico que tenga al menos un registro
    if (respuesta.Rows.Count > 0)
    {
        //Rows[0] es primera fila y Rows[0][0] es primera columna de la primera
fila.
        if (respuesta.Rows[0][0].ToString()?.Trim() == "ERROR") return
Conflict();
        //Uso el codigo 409 para notificar que ocurrio un error en la consulta
del servidor
        else
        {
            try
            {
                //Pokemon actualizado
                Pokemon actualizado = new(respuesta.Rows[0]);
                //Devuelvo el listado completo con el codigo 200
                return Ok(actualizado);
            }
            catch (Exception ex)
            {
                Logger.RegistrarERROR(ex, "Error modificando datos");
                return Conflict("Ocurrio un error en la creacion de los
datos");
            }
        }
    }
    else return NoContent();//No ocurrio un error simplemente que no existen
pokemones cargados
}
```

Recomiendo que este tipo de endpoints prueben modificando todos los campos por separado, la combinación de los mismos, con campos vacíos, con campos con datos negativos y sobre todo que manden valores null desde el JSON, con esto pueden ver la efectividad. En general ser precavido y pensar en muchas posibilidades hace que tengamos menos problemas a futuro cuando inevitablemente se de una de estas situaciones.

Como notaran no hice muchos cambios al código que ya teníamos, esto es porque al definir previamente la mayoría de las respuestas y tener en claro cómo funciona podemos reutilizar la mayoría del endpoint.



Finalmente aclaró que un método parecido es el PATCH, este sirve para hacer modificaciones parciales a elementos de la base de datos. No voy a realizar el endpoint completo para mostrárselo pero lo podríamos usar en situaciones donde queremos modificar un atributo como puede ser el nombre. En este caso tendríamos que solicitar un campo identificatorio de este pokemon y luego realizar un procedimiento que solo actualice el nombre.

Post

La inserción tenemos que cambiar un par de cosas pero no difiere mucho del método que hicimos. En la base tenemos que crear el siguiente SP:

```
CREATE PROCEDURE MODULO_POKEMON_INSERTAR_POKEMON(
@nombre varchar(255),@altura decimal(18,2),@peso decimal(18,2),
@generacion int,@id_tipo_primario int, @id_tipo_secundario int NULL)
AS
BEGIN
```

```
      INSERT INTO Pokemones
            VALUES(@nombre,@altura,@peso,
            @generacion,@id_tipo_primario,@id_tipo_secundario)

      SELECT TOP 1 *
            FROM Pokemones
                  ORDER BY id DESC
END
```

Igual que en el UPDATE podemos usar un código de respuesta para decir que fue creado, personalmente devuelvo lo creado porque muestro el ID que le asignamos. Como verán en SQL uso la palabra clave “TOP 1” que hace que recupere un solo registro de arriba de la lista mientras que “ORDER BY id DESC” va a poner el último registro creado como el primero de la selección. Además cuando hago el INSERT no agrego a los valores el ID ya que va a ser asignado de manera automática si el registro es válido.

```
[HttpPost]
[ActionName("CrearPokemon")]
[ProducesResponseType(typeof(Pokemon), StatusCodes.Status201Created)]
[ProducesResponseType(StatusCodes.Status400BadRequest)]
[ProducesResponseType(StatusCodes.Status409Conflict)]
[ProducesResponseType(StatusCodes.Status418ImATeapot)]
[ProducesResponseType(StatusCodes.Status500InternalServerError)]
public ActionResult<Pokemon> CrearPokemon([FromBody] Pokemon
pokemonActualizado)
{
    //Creo la instancia para ejecutar el metodo
    GeneralAdapterSQL consultor = new();

    //Ejecuto el procedimiento de actualizar
    DataTable respuesta = consultor.ExecuteStoredProcedure
        ("MODULO_POKEMON_INSERTAR_POKEMON",
        new() {
                    {"@nombre",pokemonActualizado.nombre },
                    {"@altura",pokemonActualizado.altura },
                    {"@peso",pokemonActualizado.peso },
                    {"@generacion",pokemonActualizado.generacion },
                    {"@id_tipo_primario",pokemonActualizado.id_tipo_primario },

{"@id_tipo_secundario",pokemonActualizado.id_tipo_secundario },
        });
    //En la base por decision devuelto el registro modificado
```

```
    //Verifico que tenga al menos un registro
    if (respuesta.Rows.Count > 0)
    {
        //Rows[0] es primera fila y Rows[0][0] es primera columna de la primera
fila.
        if (respuesta.Rows[0][0].ToString()?.Trim() == "ERROR") return
Conflict();
        //Uso el codigo 409 para notificar que ocurrio un error en la consulta
del servidor
        else
        {
            try
            {
                //Pokemon actualizado
                Pokemon pokemonCreado = new(respuesta.Rows[0]);
                //Devuelve el pokemon creado 201
                return Created("Pokemon Creado: ", pokemonCreado);
            }
            catch (Exception ex)
            {
                Logger.RegistrarERROR(ex, "Error creando el pokemon");
                return Conflict("Ocurrio un error en la creacion de los
datos");
            }
        }
    }
    else
    {
        ObjectResult result = new("ERROR: No es la pokedex correcta")
        {
            StatusCode = 418
        };
        return result;
    }
}
```

Cambie algunos códigos y también muestro como podemos mandar cualquier object result con el código que nosotros deseemos. Utilice el status code 418 “Soy una tetera” porque quería mostrarles esa curiosidad para que lo busquen (si es que les interesa).

Podemos escribir en el swagger los siguientes datos:

```
{
  "id": 0,
  "nombre": "Cyndaquil",
  "altura":0.5,
  "peso": 7.9,
  "generacion": 2,
  "id_tipo_primario": 3,
  "id_tipo_secundario": null
}
```


o usar Curl

Curl

```
curl -X 'POST' \
  'https://localhost:44398/api/Pokemon/CrearPokemon' \
  -H 'accept: text/plain' \
  -H 'Content-Type: application/json' \
  -d '{
  "id": 0,
  "nombre": "Cyndaquil",
  "altura":0.5,
  "peso": 7.9,
  "generacion": 2,
  "id_tipo_primario": 3,
  "id_tipo_secundario": null
}'
```

Como verán el ID que mande es 0 porque hay que mandar algo aunque no lo usemos y también mandó el segundo tipo como “null”. La respuesta que recibimos es:

Code	Details
201	Response body


```
{
  "id": 155,
  "nombre": "Cyndaquil",
  "altura": 0.5,
  "peso": 7.9,
  "generacion": 2,
  "id_tipo_primario": 3,
  "id_tipo_secundario": null
}
```

Delete

Finalmente el DELETE, esta operación es delicada en todos los aspectos porque si esta mal el procedimiento vamos a borrar todos los registros, nos queda un id vacío y podemos tener problemas si eliminamos un valor que está referenciado en otro registro por su FK.

Si bien les voy a mostrar con esta tabla para completar el ABMC en la práctica si quisiera dar de baja un registro de este tipo lo más probable es que usara una baja lógica mediante la creación de una columna llamada “activo” del tipo BIT. En general los únicos registros que se suelen eliminar son aquellos que son tablas de Muchos a Muchos porque es común que las combinaciones cambien.

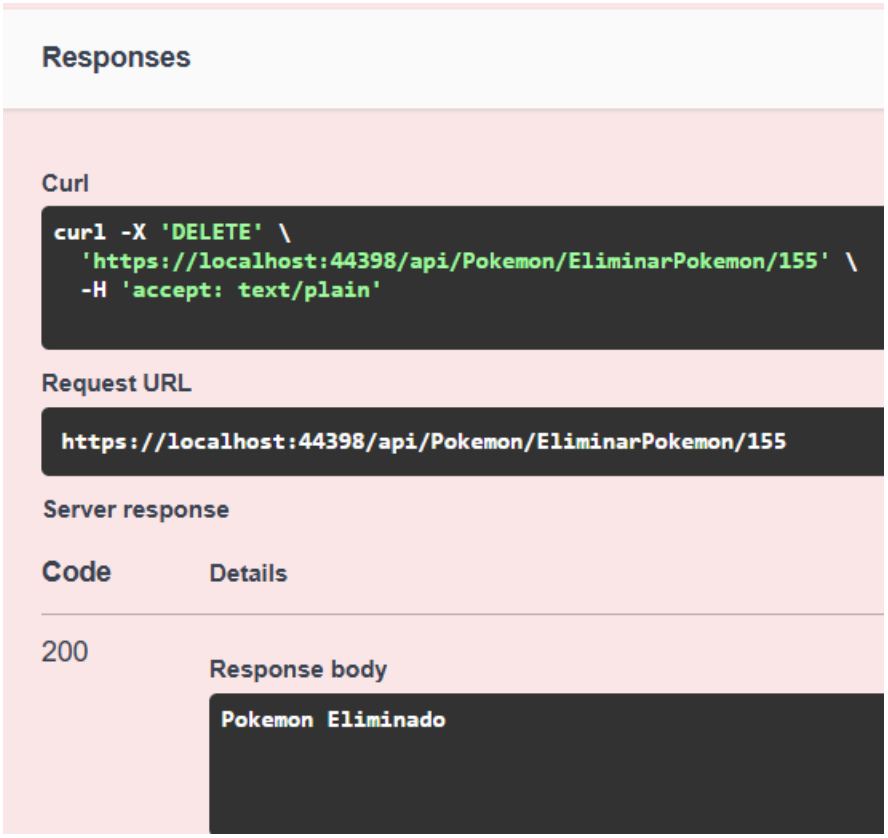
```
CREATE PROCEDURE MODULO_POKEMON_ELIMINAR_POKEMON(@id_pokemon int)
AS
BEGIN
     DELETE FROM Pokemones
          WHERE id = @id_pokemon -- La linea mas importante de todas

     SELECT *  --No deberia devolver registros
          FROM Pokemones
               WHERE id = @id_pokemon
END
```

Este endpoint es bastante diferente a los otros en cuanto a salidas y flujo.

```
[HttpDelete("{id_pokemon}")]
[ActionName("EliminarPokemon")]
[ProducesResponseType(StatusCodes.Status200OK)]
[ProducesResponseType(StatusCodes.Status400BadRequest)]
[ProducesResponseType(StatusCodes.Status409Conflict)]
[ProducesResponseType(StatusCodes.Status500InternalServerError)]
public ActionResult<Pokemon> EliminarPokemon(int id_pokemon)
{
    //Creo la instancia para ejecutar el metodo
    GeneralAdapterSQL consultor = new();

    //Ejecuto la vista
    DataTable respuesta =
consultor.ExecuteStoredProcedure("MODULO_POKEMON_ELIMINAR_POKEMON",
        new() { { "@id_pokemon", id_pokemon } });
    //Si no tiene registro es que no existe mas el pokemon
    if (respuesta.Rows.Count > 0)
    {
        //Rows[0] es primera fila y Rows[0][0] es primera columna de la primera
```

```
fila.
        if (respuesta.Rows[0][0].ToString()?.Trim() == "ERROR") return
Conflict();
        else return BadRequest();
    }
    else return Ok("Pokemon Eliminado");
}
```

Lo pruebo con el ULTIMO pokemon (les recomiendo lo mismo):



Ahora el próximo índice de inserción es el 156 pero como sabrán en la pokédex cyndaquil es el 155, por lo que eliminarlo y volver a cargarlo va a generar una discrepancia en los ID. Lo que se puede hacer es cambiar el último ID de la tabla para que cuando volvamos a agregar el pokemon tenga el número correcto.

```
DBCC CHECKIDENT('Pokemones', RESEED, 154);
```

Hay que poner el número anterior porque cuando lo cree lo que hace es sumar 1 al último índice.

Transacciones

Importancia

Las transacciones son el conjunto de acciones que realizamos en la base de datos que involucran más de una tabla. Se deben completar en “bloque”, es decir, se deben completar todas las acciones requeridas de una sola vez. Si fallara un evento se debe dar marcha atrás a lo que veníamos haciendo. En SQL existe una instrucción para esto que se llama “BEGIN TRANSACTION” que informa que las consultas que hagamos se tienen que realizar todas juntas, la base va a crear una copia en base a las modificaciones que hagamos antes de ejecutarse. Si todo sale bien las va a impactar en la realidad (commit) y si falla va a realizar una marcha atrás a como estaba la información antes de empezar (rollback).

En el curso puede ser que no tenga tanta importancia si nos equivocamos y lo arreglamos mediante base de datos, pero en muchas organizaciones nuestro ingreso a la base de datos va a estar limitado por burocracia, permisos, solicitudes, etc. Cometer un error en estos casos suele ser tedioso y en cierta forma también nos hace perder credibilidad en nuestro ámbito. Además puede acontecer que nuestro error afecte a un cliente o peor aún, a las finanzas de un cliente. Es por eso que las acciones se ejecutan por bloque no solo queremos que se realice por ejemplo un pago sino que su registro quede registrado y que si uno de los dos falla no registre nada.

Más tablas

En cuanto a lo que venimos trabajando en base de datos voy a agregar un par de tablas para representar los combates, los equipos y el resultado.

Deberían tener la tabla Entrenadores sino aca se las dejo:

```
CREATE TABLE [dbo].[Entrenadores] (
    [id]                INT           IDENTITY (0, 1) NOT NULL,
    [nombres]           VARCHAR (255) NOT NULL,
    [apellidos]         VARCHAR (255) NOT NULL,
    [fecha_nacimiento]  DATE          NOT NULL,
    [entrenador_activo] BIT           NOT NULL,
    [medallas]          INT           NOT NULL,
    CONSTRAINT [PK_Entrenadores] PRIMARY KEY CLUSTERED ([id] ASC)
);
```

Cree una tabla de soporte para que estén las diferentes posibilidades de un combate. Me pareció útil ya que con esto controlar que siempre haya algún resultado aunque no se haya empezado.

```
CREATE TABLE [dbo].[ResultadosPelea]
(
      [id] INT NOT NULL PRIMARY KEY IDENTITY(0,1),
      [nombre_resultado] varchar(255) NOT NULL,
      [descripcion] varchar(255) NOT NULL,
      [sigla] varchar(255) NOT NULL
)
```

Les dejo mis registros para que estemos en la misma página.

```
SET IDENTITY_INSERT [dbo].ResultadosPelea ON

INSERT [dbo].ResultadosPelea ([id], [nombre_resultado],[descripcion],[sigla])
VALUES (0, N'Ganador Local',     N'Ocurre cuando gana el entrenador local',N'L')
INSERT [dbo].ResultadosPelea ([id], [nombre_resultado],[descripcion],[sigla])
VALUES (1, N'Ganador Visitante', N'Ocurre cuando gana el entrenador
visitante',N'V')
INSERT [dbo].ResultadosPelea ([id], [nombre_resultado],[descripcion],[sigla])
VALUES (2, N'Empate',N'Ocurre cuando ambos entrenadores se quedan sin pokemones
al mismo tiempo',N'E')
INSERT [dbo].ResultadosPelea ([id], [nombre_resultado],[descripcion],[sigla])
VALUES (4, N'Cancelado',    N'El combate se cancelo por acuerdo de los
entrenadores',N'C')
INSERT [dbo].ResultadosPelea ([id], [nombre_resultado],[descripcion],[sigla])
VALUES (5, N'Programado',   N'El combate esta programado y aun no comenzo',N'P')
INSERT [dbo].ResultadosPelea ([id], [nombre_resultado],[descripcion],[sigla])
VALUES (6, N'Equipo Rocket',     N'El equipo Rocket (Jessie-James-Meowth)
nuevamente irrumpio el combate por lo que queda inconcluso',N'R')

SET IDENTITY_INSERT [dbo].ResultadosPelea OFF
```

Lo que viene a continuación **puede ser debatible** ya que cree una estructura bastante rápida para este ejemplo e intente considerar algunas cosas. Sin embargo no creo que sea perfecto pero tiene su lógica detrás. Primero está la tabla combates la cual tiene un entrenador local y visitante porque quería hacerlo con esas condiciones. Después cree una tabla equipos que seria algo asi como, combateXpokemonXentrenador, la logica detras de esto es porque considere que los entrenadores no van a tener un equipo fijo sino que lo van a ir cambiando entre los diferentes combates. Con esto en mente también volví a utilizar la FK que apunte a entrenadores para distinguir que pokemon es de cada cual.

```
CREATE TABLE [dbo].[Combates] (
    [id]                  INT         NOT NULL IDENTITY (0, 1),
    [id_entrenador_local]  INT         NOT NULL,
    [id_entrenador_visita] INT         NOT NULL,
    [fecha_combate]        DATE        NOT NULL,
    [inicio_combate]       TIME (0)    NOT NULL,
    [final_combate]        TIME (0)    NULL,
    [resultado_combate]    INT         NOT NULL,
    PRIMARY KEY CLUSTERED ([id] ASC),
    CONSTRAINT [FK_EntrenadorLocal] FOREIGN KEY ([id_entrenador_local])
REFERENCES [dbo].[Entrenadores] ([id]),
    CONSTRAINT [FK_EntrenadorVisita] FOREIGN KEY ([id_entrenador_visita])
REFERENCES [dbo].[Entrenadores] ([id]),
     CONSTRAINT [FK_CombateResultado] FOREIGN KEY (resultado_combate)
REFERENCES [dbo].ResultadosPelea ([id])
);
```

La siguiente tabla ya existe si ejecutaron mi script por lo que si hay un registro deben borrarlo o permitir que sea “NULL” el ID combate hasta que le asignen un valor a todos los registros.

```
CREATE TABLE [dbo].[Equipos] (
    [id]            INT IDENTITY (0, 1) NOT NULL,
    [id_pokemon]    INT NOT NULL,
    [id_entrenador] INT NOT NULL,
     [id_combate]    INT NOT NULL,
    CONSTRAINT [PK_Equipos] PRIMARY KEY CLUSTERED ([id] ASC),
    CONSTRAINT [FK_Equipos_Entrenadores] FOREIGN KEY ([id_entrenador])
REFERENCES [dbo].[Entrenadores] ([id]),
    CONSTRAINT [FK_Equipos_Pokemones] FOREIGN KEY ([id_pokemon]) REFERENCES
[dbo].[Pokemones] ([id]),
     CONSTRAINT [FK_Equipos_Combate] FOREIGN KEY ([id_combate]) REFERENCES
[dbo].Combates ([id])
);
```

Elegí crearlo así pero tuve las algunas ideas que capaz les ocurran a ustedes:

- Crear una tabla equipo con 6 FK que puedan ser Null excepto la primera. Crear una tabla Detalles Combate que referencie al entrenador y a su equipo.
    - Esto es porque todos los equipos no pueden ser más de 6 y tienen que tener al menos un pokémon. Además considere no tener dos veces la FK entrenador desde 2 tablas diferentes.
    - Aunque lo considere, no me gusta la idea de tener tantas FK que pueden estar vacías o sin referencia.

    - La tabla detalle también podría funcionar para combates dobles o de “n” entrenadores pero me parece más fácil crear otra tabla que se llame combates dobles/triples/etc.
- En combates no hacer que haya entrenador local o visitante sino que directamente en la tabla equipos ponga si era local o visitante.
    - No me gustó esta idea porque iba a estar poniendo una columna repetida entre todos los registros del equipo.
    - También pensé en que no haya local y visita pero quería darle ese enfoque de alguna forma.

Realización

Para el tema de las transacciones podemos ejecutar un solo procedimiento en la base de datos que contenga el bloque BEGIN TRANSACTION o abrir la transacción desde el backend de la API y hacer un commit cuando terminemos. La primera parte suele ser más seguro y requiere menos programacion en C# mientras que la segunda permite modularizar las acciones para después ejecutarlas en orden.

Supongamos ahora que queremos dar de alta un nuevo combate que contenga ambos equipos pokémon y los datos del combate que está programado. En la base lo que corresponde hacer es registrar el combate y luego registrar los pokemones que va a usar cada uno para el mismo.

Si quisiéramos hacerlo en SQL en un solo procedimiento tendríamos los siguientes inconvenientes:

- Es difícil enviar listas como parámetro, un método poco ortodoxo que he visto sería si nosotros enviamos los id de los pokemones como string así: ‘1,12,3’ después hacemos un split y los cargamos por separado. Entiendo que se pueden mandar tablas pero me parece más complicado.
- Deberíamos pedir un monton de parametros para un solo procedimiento y separarlos en local y visitante
- En codigo seria algo asi:

```
DECLARE @nombre_transaccion VARCHAR(20);
SELECT @nombre_transaccion = 'Alta Combate';

BEGIN TRANSACTION @nombre_transaccion;
USE PokeBase;

--Operaciones

COMMIT TRANSACTION @TranName;
GO
```

Por otro lado en C# tenemos el siguiente problema:

- Si lo queremos hacer usando nuestro “EjecutarProcedimiento” tenemos el problema que para cargar el combate en la tabla Equipos no vamos a conocer el último id de combate.
- La forma de iniciar el procedimiento no es tan segura como la de SQL y a veces cuando hacemos un rollback en una inserción no restaura correctamente el último ID.

Actualmente lo que yo hago es generar una clase que controle la transacción de manera exitosa y además en los procedimientos creo 2 versiones, una en la cual utilice el último id (id_combate) y otra en la que utilice un id que nosotros le enviamos. En este caso no es tan obvio pero hay veces que las acciones dentro de la transacción las utilizamos de manera separada en otros procedimientos, por lo que modularizar y luego llamarlo desde la API nos da reutilización. El último detalle a tener en cuenta, para cualquiera de los 2 métodos, es que queremos ejecutarlo todo en un solo endpoint porque sino se vuelve demasiado complicado desde el backend para manejar.

Empecemos por el controlador, endpoint y las clases que vamos a usar:

```
/// <summary>
/// Este controlador permite manipular las funcionalidades relacionadas a
combate
/// </summary>
[EnableCors("CorsRules")]
[Route("api/[controller]/[action]")]
[ApiController]
public class CombateController : Controller
{
        [HttpPost]
        [ActionName("CrearCombate")]
        [ProducesResponseType(typeof(Combate), StatusCodes.Status201Created)]
        [ProducesResponseType(StatusCodes.Status400BadRequest)]
        [ProducesResponseType(StatusCodes.Status409Conflict)]
        [ProducesResponseType(StatusCodes.Status418ImATeapot)]
        [ProducesResponseType(StatusCodes.Status500InternalServerError)]
        public ActionResult<Combate> CrearCombate()
        {
            // Este Ya lo vamos a completar
        }
}
```

Creó el modelo de salida de combate sin los comentarios XML:

```
public class Combate
{
    public int id { get; set; }
    public int id_entrenador_local { get; set; }
```

```
    public int id_entrenador_visita { get; set; }
    public DateTime fecha_combate { get; set; }
    public TimeOnly inicio_combate { get; set; }
    public TimeOnly? final_combate { get; set; }
    public int resultado_combate { get; set; }

    public Combate(DataRow fila)
    {
        this.id = int.Parse(fila["id"].ToString() ?? "0");
        this.id_entrenador_local =
int.Parse(fila["id_entrenador_local"].ToString() ?? "0");
        this.id_entrenador_visita =
int.Parse(fila["id_entrenador_visita"].ToString() ?? "0");
        this.fecha_combate = DateTime.Parse(fila["fecha_combate"].ToString() ??
"2022-12-18");
        this.inicio_combate = TimeOnly.Parse(fila["inicio_combate"].ToString()
?? "00:00:00");

        string? final = fila["final_combate"].ToString();
        //Hago un control de nulidad diferente ya que no queremos que muestre
un valor predeterminado
        if (final != null && final.Trim() != "") this.final_combate =
TimeOnly.Parse(final);
        //Agrego el diferente a "" porque puede ocurrir que lo convierta a esta
cadena
        else this.final_combate = null;

        this.resultado_combate = int.Parse(fila["resultado_combate"].ToString()
?? "0");
    }

    public Combate()
    {
        this.id = -1;
        this.id_entrenador_local = 0;
        this.id_entrenador_visita = 0;
        this.fecha_combate = DateTime.Now;
        this.inicio_combate = TimeOnly.MinValue;
        this.final_combate = TimeOnly.MinValue;
        this.resultado_combate = 0;
    }
}
```

Para finalizar les muestro que clase creo yo para generar la transacción:

```
public class CombateTransaccion
{
    public int id_entrenador_local { get; set; }
    public int id_entrenador_visita { get; set; }
    public DateTime fecha_combate { get; set; }
    public TimeOnly inicio_combate { get; set; }
    List<int> pokemones_local { get; set; }
    List<int> pokemones_visita { get; set; }
    public CombateTransaccion()
    {
        this.id_entrenador_local = 0;
        this.id_entrenador_visita = 0;
        this.fecha_combate = DateTime.Now;
        this.inicio_combate = TimeOnly.MaxValue;
        this.pokemones_local = new();
        this.pokemones_visita = new();
    }
}
```

Notaran que hay campos que no están ya que estos los vamos asignar desde la base de datos o tienen un valor predeterminado. Al endpoint le agrego esta clase como parámetro:

```
CrearCombate([FromBody] CombateTransaccion nuevoCombate)
```

Les muestro finalmente los SP que voy a utilizar:

```
CREATE PROCEDURE MODULO_COMBATE_INSERTAR_COMBATE(
     @id_entrenador_local int,
     @id_entrenador_visita int,
     @fecha_combate date,
     @inicio_combate time(0)
)
AS
BEGIN

     -- Los ultimos valores siempre son los mismos cuando se empieza
     -- El final del combate es NULL y el id de resultado es programado
     INSERT INTO Combates

VALUES(@id_entrenador_local,@id_entrenador_visita,@fecha_combate,@inicio_combat
e,NULL,5)
```

```
      SELECT TOP 1 *
            FROM Combates
                  ORDER BY id DESC
END
```

```
CREATE PROCEDURE MODULO_COMBATE_INSERTAR_EQUIPO_TRANSAC(@id_pokemon
int,@id_entrenador int)
AS
BEGIN

--Este metodo lo que hace es asumir que queremos cargar los equipos del ultimo
combate
--Con esto evitamos la manipulacion desde C# de variables
      DECLARE @id_combate INT

      SELECT TOP 1 @id_combate = id
            FROM Combates
                  ORDER BY id DESC

      INSERT INTO Equipos
            VALUES(@id_pokemon,@id_entrenador, @id_combate)

      SELECT 'OK' AS 'Codigo'

END
GO
```

Como verán en esta versión recuperó el último combate para asignarle el ID ya que sabemos que va a estar en un bloque de transacción. Cuando finalice voy a recuperar el combate para mostrarlo en el mensaje creado.

```
CREATE PROCEDURE MODULO_COMBATE_RECUPERAR_ULTIMO_COMBATE
AS
BEGIN
      SELECT TOP 1 *
            FROM Combates
                  ORDER BY id DESC
END
```

Finalmente necesitamos controlar el bloque de transacción de C#. Les dejo como se hace con las clases de SQL para que lo prueben por su cuenta:

```
public static DataTable ExecuteTransaction()
{
    DataTable respuesta = new();

    using (var conexion = new SqlConnection("Conexion"))
    {
        conexion.Open();
        using (var transaccion = conexion.BeginTransaction())
        {
            try
            {
                using var comando = new SqlCommand("procedimiento1",
"conexionBase");
                using var comando2 = new SqlCommand("procedimiento2",
"conexionBase");
                //La idea es que ejecutemos varios comandos durante la
transaccion
                comando.CommandType = CommandType.StoredProcedure;

                SqlDataAdapter adaptador = new(comando);
                adaptador.Fill(respuesta);

                if (respuesta.Rows.Count == 0)
                {
                    transaccion.Rollback();
                    return new();
                }

                SqlDataAdapter adaptador = new(comando2 );
                adaptador.Fill(respuesta);

                if (respuesta.Rows.Count == 0)
                {
                    transaccion.Rollback();
                    return new();
                }
                //Si todo funciona hacemos commit
                transaccion.Commit();

            }
            catch (Exception ex)
            {
                transaccion.Rollback();
```

```
            }
            finally
            {
                SqlConnection.ClearAllPools();
                conexion.Close();
            }
        }

    };

    return respuesta;
}
```

Ahora les brindó como yo lo solucionaría, como siempre les digo, no significa que sea la resolución perfecta pero si la que he utilizado.

Creo una clase para almacenar todos los datos:

```
/// <summary>
/// Clase que nos permite manipular una transaccion
/// </summary>
public class TransaccionSQL
{
    public string nombre_transaccion { get; set; }

    public List<string> procedimientos { get; set; }

    public List<Dictionary<string, object>> parametros { get; set; }

    public TransaccionSQL(string nombreTransaccion)
    {
        this.nombre_transaccion = nombreTransaccion;
        this.procedimientos = new();
        this.parametros = new();
    }
}
```

Creó el siguiente procedimiento en nuestro GeneralAdapterSQL

```
public bool EjecutarTransaccion(TransaccionSQL transaccionPropia)
{
    ObtenerCadenaConexion();

    DataTable respuesta = new();
```

```
    bool resultado = true;

    using (var conexionBase = new SqlConnection(CadenaConexion))
    {
        conexionBase.Open();

        //Iniciamos la transaccion
        using var transaccionSQL =
conexionBase.BeginTransaction(transaccionPropia.nombre_transaccion);
        try
        {
            //Recorro todos los procedimientos en orden
            for (int i = 0; i < transaccionPropia.procedimientos.Count; i++)
            {
                string procedimiento = transaccionPropia.procedimientos[i];
                Dictionary<string, object> parametros =
transaccionPropia.parametros[i];

                respuesta = new();
                using var comando = new SqlCommand(procedimiento, conexionBase)
                {
                    CommandType = CommandType.StoredProcedure,
                    Transaction = transaccionSQL
                };

                SqlDataAdapter adapter = new(comando);

                foreach (var item in parametros)
                {
                    if (item.Value == null || item.Value.ToString()?.Trim() ==
"NULL")
                    {
                        comando.Parameters.AddWithValue(item.Key,
DBNull.Value);
                    }
                    else
                    {
                        comando.Parameters.Add(item.Key,
GetDBType(item.Value));
                        comando.Parameters[item.Key].Value = item.Value;
                    }
                }
```

```
                adapter.Fill(respuesta);

                //En todas las respuestas debemos devolver algo
                //Podemos elegir que sea un codigo o que sea la tabla
                if (respuesta.Rows.Count == 0)
                {
                    Logger.RegistrarERROR(new(), "ERROR EJECUTANDO EL
PROCEDIMIENTO: " + procedimiento);
                    respuesta = new();
                    //Esto es para que la API pueda devolver un conflict()
(409)
                    respuesta.Columns.Add("MENSAJE");
                    respuesta.Rows.Add("ERROR");
                    //Si falla hacemos un rollback y rompemos le ciclo con
return
                    transaccionSQL.Rollback();
                    resultado = false;
                    break;
                }
            }
            if (resultado) transaccionSQL.Commit();

        }
        catch (Exception ex)
        {
            respuesta = new();
            //Esto es para que la API pueda devolver un conflict() (409)
            respuesta.Columns.Add("MENSAJE");
            respuesta.Rows.Add("ERROR");
            //Registramos en el Log el error
            Logger.RegistrarERROR(ex, "ERROR EJECUTANDO LA TRANSACCION: " +
transaccionPropia.nombre_transaccion);
            transaccionSQL.Rollback();
            resultado = false;
        }
        finally
        {
            SqlConnection.ClearAllPools();
            conexionBase.Close();
        }

    };
```

```
    return resultado;
}
```

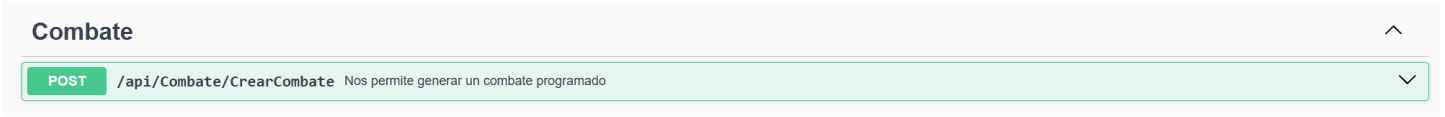
Por lo que nuestro endpoint queda de la siguiente forma:

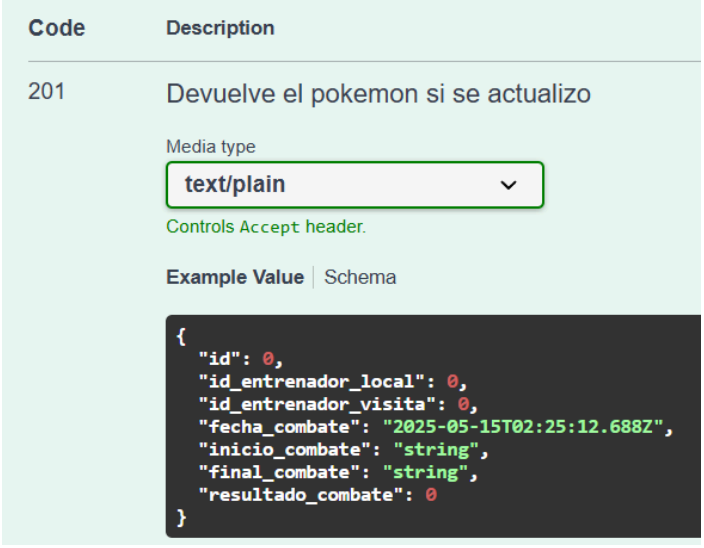
```
public ActionResult<Combate> CrearCombate([FromBody] CombateTransaccion
nuevoCombate)
{
    //Ejemplos de que puede ser un bad request en esta operacion
    if (nuevoCombate.id_entrenador_local == nuevoCombate.id_entrenador_visita)
        return BadRequest("El id_entrenador_local y id_entrenador_visita deben
ser diferentes");
    if (nuevoCombate.pokemones_local.Count ||
        nuevoCombate.pokemones_visita.Count == 0)
        return BadRequest("Ambos equipos deben contener al menos un pokemon");

    //Creo el objeto transaccion con el nombre correspondiente
    TransaccionSQL transac = new("TransaccionCombate");

    //Agrego el procedimiento a la lista
    transac.procedimientos.Add("MODULO_COMBATE_INSERTAR_COMBATE");
    //Agrego sus parametros en la misma posicion
    transac.parametros.Add(new()
            {
                {"@id_entrenador_local",nuevoCombate.id_entrenador_local },
                {"@id_entrenador_visita",nuevoCombate.id_entrenador_visita },
                {"@fecha_combate",nuevoCombate.fecha_combate },
                {"@inicio_combate",nuevoCombate.inicio_combate },
            });

    //Cargamos el procedimiento con los pokemones locales
    foreach (int id_pokemon_local in nuevoCombate.pokemones_local)
    {
        //Agrego el procedimiento a la lista
        transac.procedimientos.Add("MODULO_COMBATE_INSERTAR_EQUIPO");
        //Agrego sus parametros en la misma posicion
        transac.parametros.Add(new()
                {
                    {"@id_pokemon",id_pokemon_local },
                    {"@id_entrenador",nuevoCombate.id_entrenador_local }
                });
    }
```

```
    //Agregamos los pokemones visitante
    foreach (int id_pokemon_visita in nuevoCombate.pokemones_visita)
    {
        //Agrego el procedimiento a la lista
        transac.procedimientos.Add("MODULO_COMBATE_INSERTAR_EQUIPO");
        //Agrego sus parametros en la misma posicion
        transac.parametros.Add(new()
                {
                    {"@id_pokemon", id_pokemon_visita},
                    {"@id_entrenador",nuevoCombate.id_entrenador_visita }
                });
    }

    try
    {
        GeneralAdapterSQL controlador = new();
        if (controlador.EjecutarTransaccion(transac))
        {
            DataTable respuesta =
controlador.EjecutarVista("MODULO_COMBATE_RECUPERAR_ULTIMO_COMBATE");

            //Puede ocurrir que la transaccion se completo pero fallo cuando
buscamos el combate
            if (respuesta.Rows.Count > 0 &&
respuesta.Rows[0][0].ToString()?.Trim() == "ERROR")
                return Conflict("ERROR: Se completo la transaccion pero no se
pudo recuperar el combate");
            else
            {
//Creamos el combate y lo enviamos. Podria haber otro try catch por si la
creacion falla
            return Created("Combate creado: ", new Combate(respuesta.Rows[0]));
            }
        }
        //No se completo exitosamente la transaccion levanto una excepcion
        else throw new Exception("TRANSACCION INVALIDA");
    }
    catch (Exception ex)
    {
        Logger.RegistrarERROR(ex, "ERROR: No se pudo completar el procedimiento
CrearCombate ");
        return Conflict(ex.Message + " Imposible completar la accion");
    }
```

```
}
```

Con esto completamos una transacción, para corroborar que funciona exitosamente podemos utilizar datos correctos para crear el combate pero pasarle por ejemplo el id_pokemon -1 que hace que salte una excepción en SQL. Podemos omitir variables, modificar el código para que devuelva mal un valor etc. Si todo es correcto cuando nosotros inspeccionamos la tabla “Combates” no debería haber creado ningún registro al igual que Equipos.





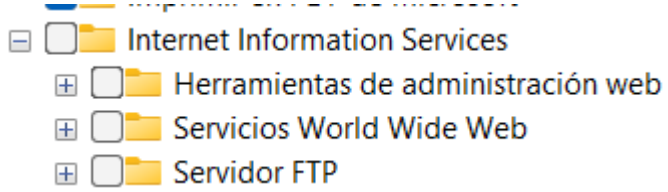
Despliegue en IIS

Previo

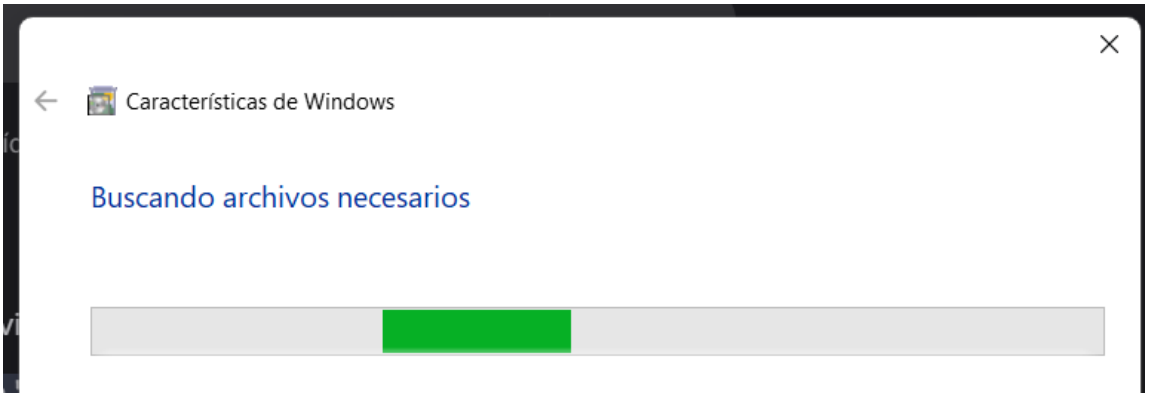
El Internet Information Service es un conjunto de herramientas que nos permite desplegar aplicaciones web en los dispositivos Windows. Para el propósito de este curso podremos desplegarlo en nuestra computadora de manera local y puede ser accedida desde otros dispositivos en la misma LAN.

Debemos instalar una característica desde la siguiente opción en búsqueda de nuestro sistema operativo.

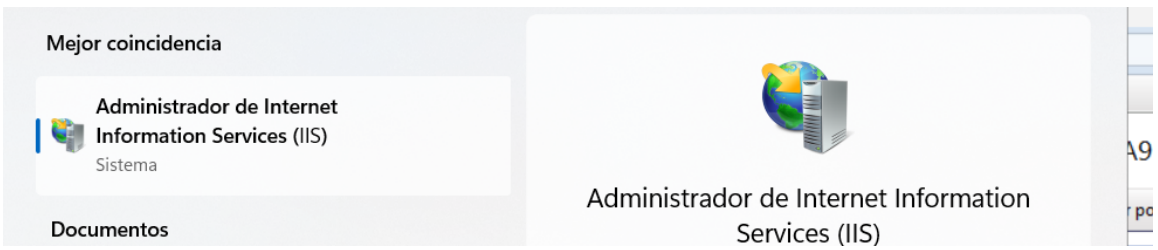
Dentro de la opción debemos buscar:

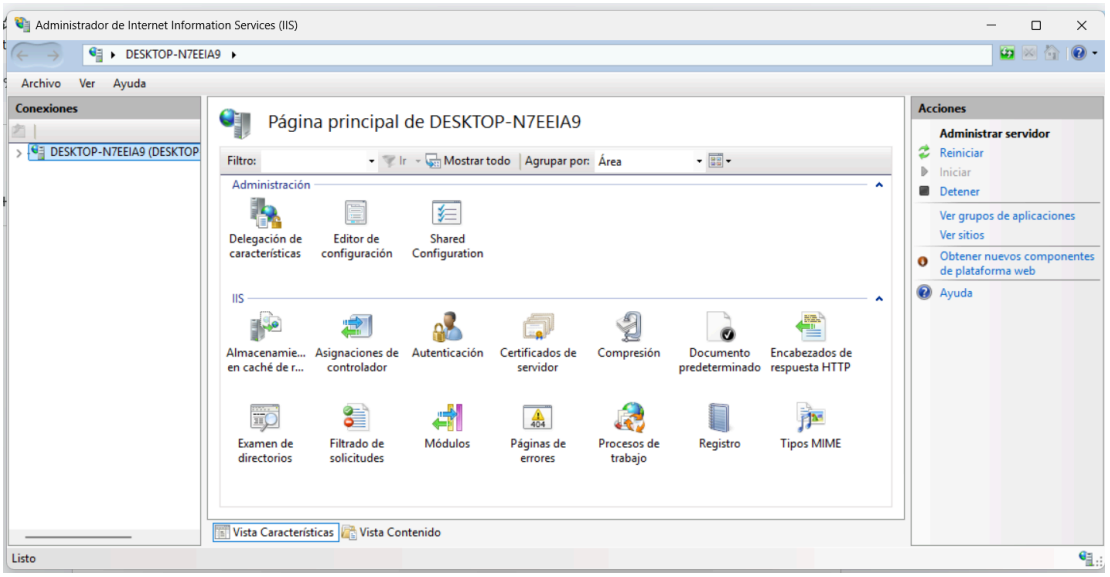


Marcamos esta opción y continuamos



Una vez completados debería aparecer la siguiente opción en búsqueda:





Una vez configurado tenemos que descargar el “Hosting Bundle” que son un conjunto de librerías que permiten desplegar aplicaciones web.

Run apps - Runtime

ASP.NET Core Runtime 8.0.16

The ASP.NET Core Runtime enables you to run existing web/server applications. **On Windows, we recommend installing the Hosting Bundle, which includes the .NET Runtime and IIS support.**

IIS runtime support (ASP.NET Core Module v2)
18.0.25107.16

OS	Installers	Binaries
Linux	Package manager instructions	Arm32 | Arm32 Alpine | Arm64 | Arm64 Alpine | x64 | x64 Alpine
macOS		Arm64 | x64
Windows	x64 | x86 | Arm64 | Hosting Bundle | winget instructions	x64 | x86 | Arm64

https://dotnet.microsoft.com/en-us/download/dotnet/thank-you/runtime-aspnetcore-8.0.16-windows-hosting-bundle-installer

Ahora tengo que mencionar algunos detalles que van cambiando de acuerdo a las versiones de C# y en todos los despliegues podemos tener problemas. Lo primero es en Program.cs, debemos agregar un “OR” para que también muestre swagger en producción.

```
if (app.Environment.IsDevelopment() || app.Environment.IsProduction())
```

Después tuve que comentar el app.UseAuthorization() para que me muestre el swagger ya que estaba teniendo conflictos con esto.

```
//app.UseAuthorization();
```

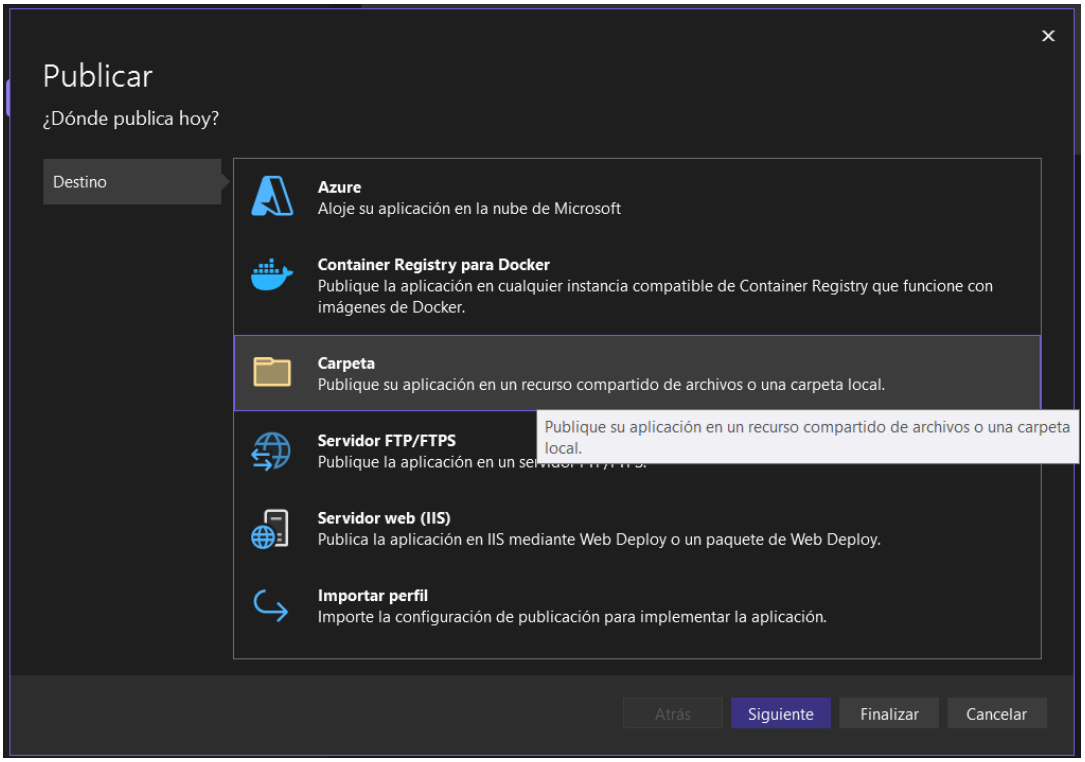
Otro detalle es que las APIs no pueden usar el Windows Auth que usamos normalmente para conectarnos a la base de datos. Es por eso que debemos crear un login y un usuario para nuestra base de datos dándole permisos únicamente de seleccionar y ejecutar nuestras vistas y procedimientos. Cada vez que creamos un objeto en la base debemos hacer esto

```
--Creamos las credenciales
CREATE LOGIN [IIS APPPOOL\PokeAPI] FROM WINDOWS WITH
DEFAULT_DATABASE=[PokeBase], DEFAULT_LANGUAGE=[Español]
GO
CREATE USER PokeUser FOR LOGIN [IIS APPPOOL\PokeAPI];
GO

-- Dar permisos sobre las vistas
GRANT SELECT ON MODULO_COMBATE_RECUPERAR_ULTIMO_COMBATE TO PokeUser;
GO
GRANT SELECT ON MODULO_POKEMON_OBTENER_POKEMONES_COMPLETO TO PokeUser;
GO
-- Dar permisos sobre los procedimientos
GRANT EXECUTE ON MODULO_COMBATE_INSERTAR_COMBATE TO PokeUser;
GO
GRANT EXECUTE ON MODULO_POKEMON_ACTUALIZAR_POKEMON TO PokeUser;
GO
GRANT EXECUTE ON MODULO_POKEMON_ELIMINAR_POKEMON TO PokeUser;
GO
GRANT EXECUTE ON MODULO_POKEMON_INSERTAR_POKEMON TO PokeUser;
GO
GRANT EXECUTE ON MODULO_POKEMON_OBTENER_POKEMON_ID TO PokeUser;
GO
```

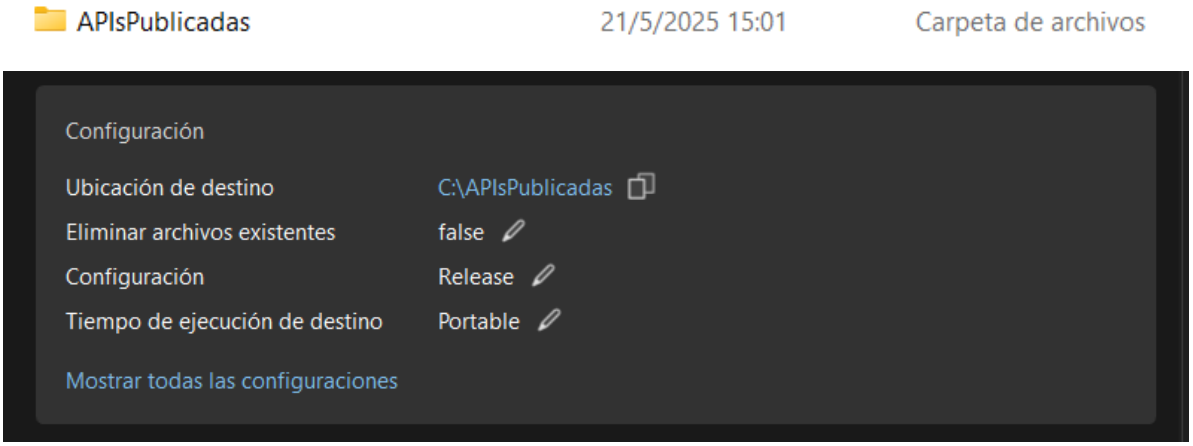
El IIS APPPOOL es un grupo de aplicaciones predeterminado que le asigna nuestro IIS cuando creamos el sitio web.

Despliegue

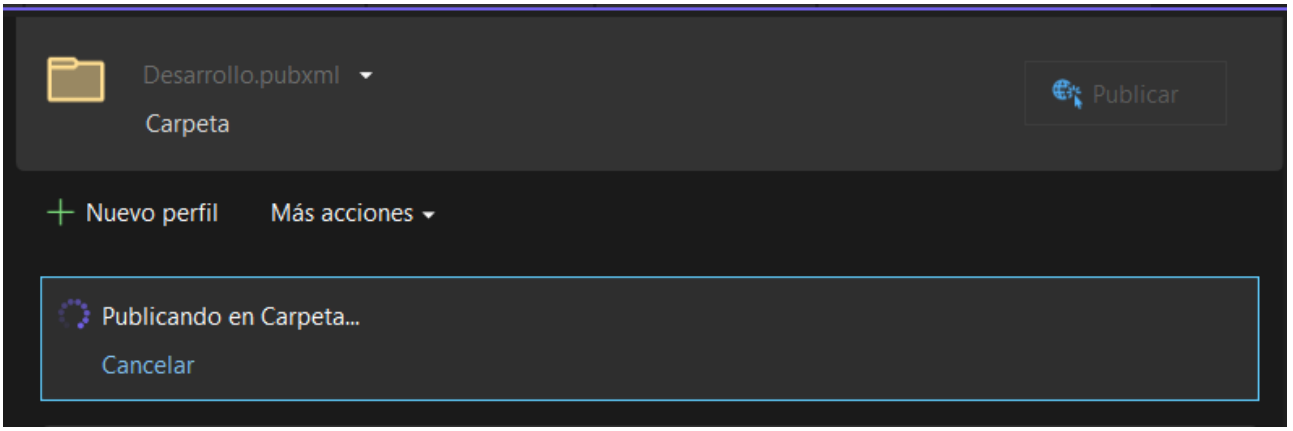
Para desplegar debemos hacer click derecho en el proyecto de la API y buscar la opción de “Publicar” la cual nos redirige a sus opciones.

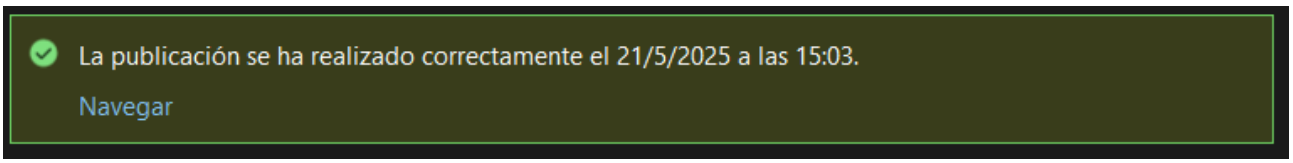


En general usar el método de carpeta es lo mismo que ir a la carpeta bin/release de nuestro proyecto y copiarlo en otro lado. Si nosotros lo hacemos de esta forma nos va a permitir publicarlo de manera más cómoda en vez de estar pisando nuestros archivos. Deberían crear la carpeta donde se está realizando las publicaciones fuera de usuarios así no tienen conflicto con los permisos.

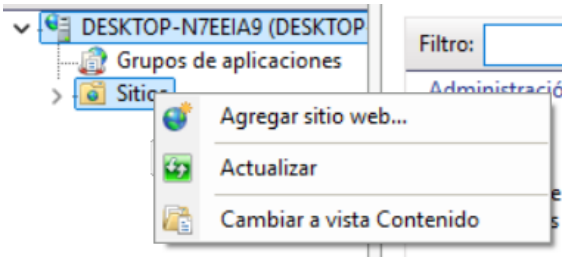


Como veran los cree en C:\APIsPublicadas, le damos a publicar:

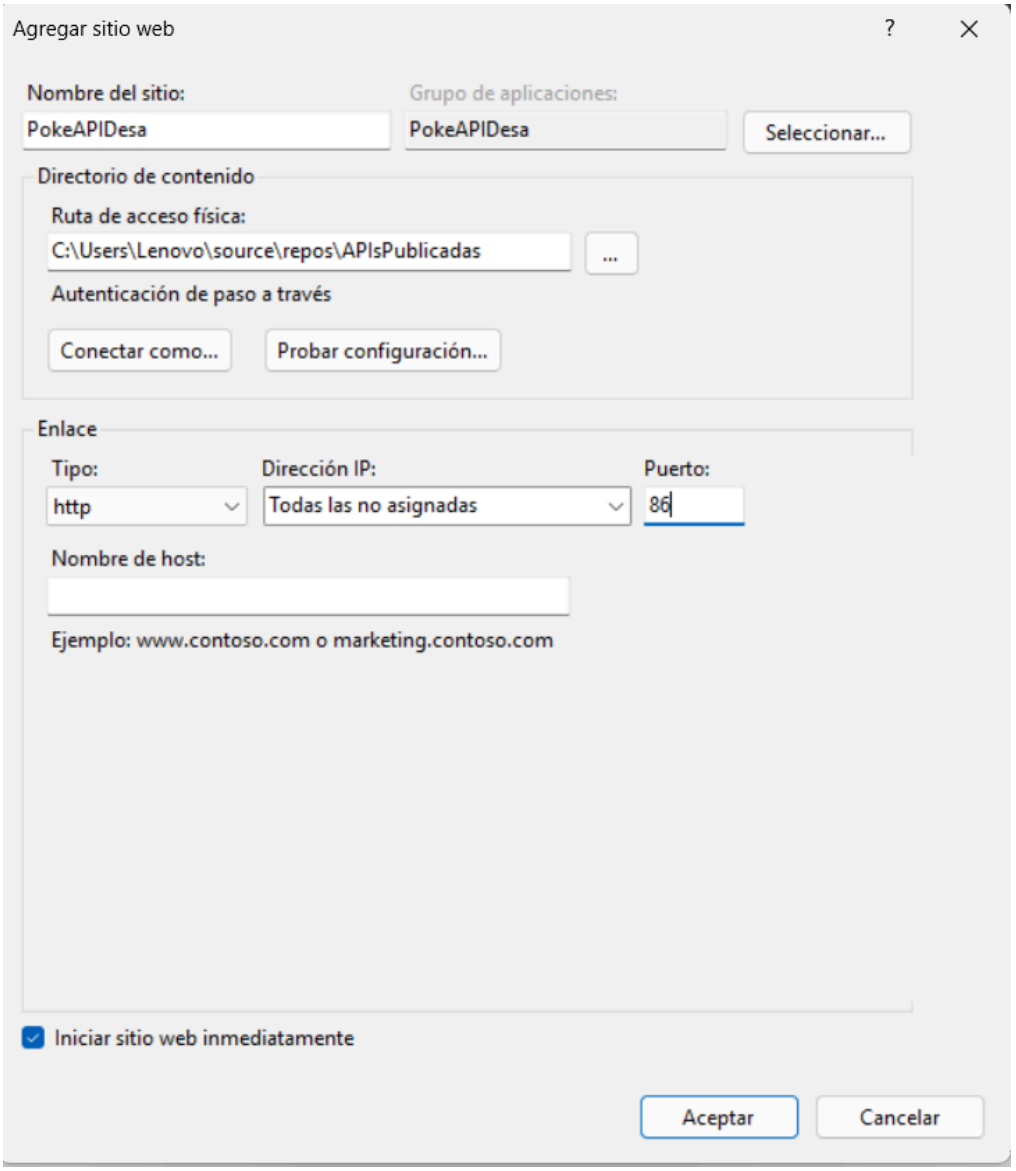




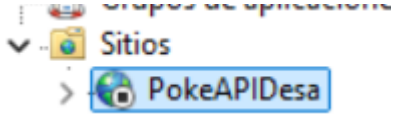
Finalmente vamos al IIS y hacemos click derecho en la parte de sitios para agregar un nuevo sitio web.



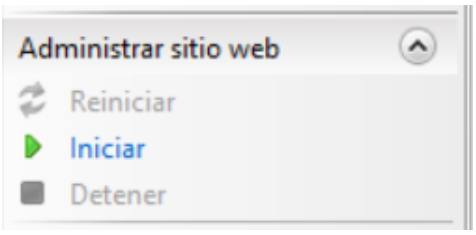
Debemos completar la siguiente información, generalmente para las pruebas utilizo http para no tener problemas con SSL o los certificados.



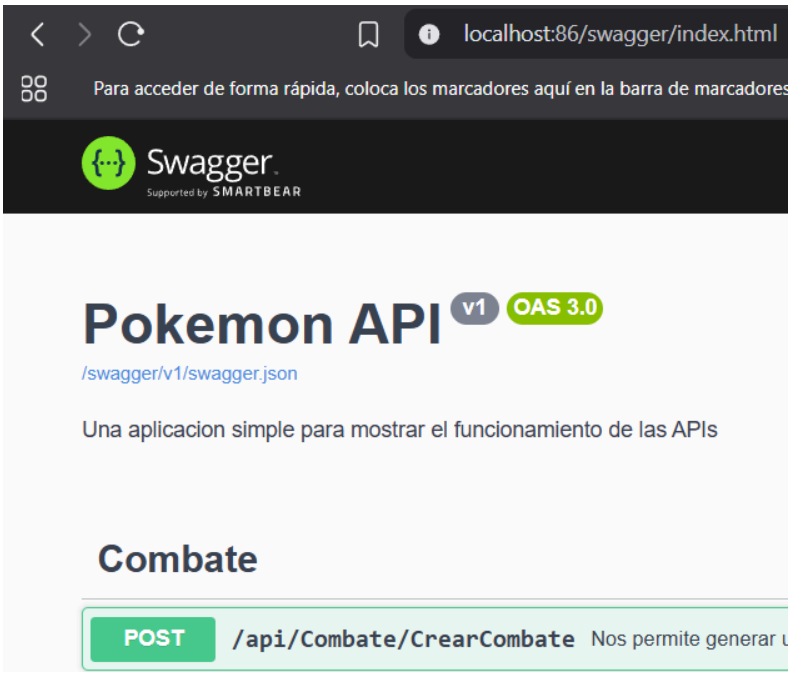
Le damos aceptar y nos queda:



A la derecha le debemos dar a iniciar:



Y podemos revisar si está funcionando entrando a nuestro localhost:86/swagger/index.html



Con esto finaliza la parte del despliegue en una localhost, esto es replicable en servidores que contengan este componente.