# Qué es una web API?

Es un conjunto de interfaces definidas las cuales permiten a un conjunto de aplicaciones externas consumir el software de la aplicación web.

**Permite que dos apps completamente distintas se puedan comunicar entre sí.**

**Un enchufe es una interfaz que permite conectar aparatos eléctricos al tomacorriente, permitiendo que cualquier aparato eléctrico pueda usar el enchufe.**

**enchufe API → aparatos son los clientes del api**

resumen:

un web api es un conjunto de funciones de nuestra app web que pueden ser consumidas por otras app a distancia.

**diferencia entre web service y web api**

* La diferencia principal es que el **servicio Web** *facilita que dos máquinas interactúen a través de una red*, mientras que una **API** *es una interfaz. ... Y a través de ella dos aplicaciones establecen comunicación.*

estilos de hacer web apis

**Rest (representational state transfer)**

Es un estilo para construcción de servicios web los cuales se adhieren a un conjunto de principios establecidos.

cuando consumimos un web api, es por que queremos acceder a sus recursos.

entendiéndose por recursos:

1. listados de productos .
2. listados de vendedores.

se utilizan métodos http sobre una url para ejecutar distintas funciones

por ejemplo: <https://miwebapi.com/api/user> y sobre esta se ejecuta un método get.

* REST no es un http crud
* no todos los web apis que se crean deben ser RESFUL

**6 condiciones que se deben cumplir para que sea RESTFUL**

1. Arquitectura cliente servidor
   * La arquitectura cliente servidor nos habla de la separación entre un cliente y un proveedor o servidor
   * Esto permite la evolución independiente de nuestra Web API sin afectar a los clientes existentes.
2. Interfaz uniforme
   * la idea de la interfaz uniforme es tener una forma estandarizada de transmisión de información.
     1. sub condiciones
        1. identificación de recursos.

[https://miwebapi.com/api/](https://miwebapi.com/api/user)libros.

* + - 1. manipulación de recursos usando representaciones
      2. mensajes autodescriptivos.(peticiones http al servidor, usando media types por ejemplo)
      3. HATEOS (incluir links que permita seguir explorando los demás recursos del api)

1. Protocolo sin estado
   * Cada una de las peticiones realizadas al web api tienen toda la información necesaria para que la petición sea resuelta de manera satisfactoria.
2. cache
   * las respuestas del web api deben indicar cuándo se deben guardar en caché, no todo se debe guardar en cache ya que esto provocaría que brindamos al cliente información desactualizada.
3. sistema de capas
   * el servicio del servidor debe tener una arquitectura de capas, donde su evolución sea completamente transparente para el cliente. low balancer
4. código en demanda (opcional)
   * el servicio web tiene la opción de enviar código fuente el cual se va a ejecutar en el cliente. típicamente este codigo es **JavaScript**.

# anatomía de una petición HTTP

cuando trabajamos la comunicación entre el cliente y el api se hace a través de peticiones http.

* una petición http es un mensaje que una computadora envía a otra utilizando el protocolo HTTP
* contestamos con una respuesta http.

estructura de una petición http,compuesta por 3 partes:

1. una línea de petición
2. conjunto de campos cabecera
3. un cuerpo, el cual es opcional

**línea de petición**

Se coloca el método http a utilizar la URI, de la petición y el protocolo http a utilizar.

estructura: **MÉTODO**-http URI PROTOCOLO-http

**MÉTODO: indica que tipo de acción se va a realizar**

**URI: ubicación del recurso.**

**protocolo: que protocolo http se va a utilizar HTTP/ 1.1, http 2.0**

**ejemplo: get/api/autores http/1.1**

post/ test.html http/1.1 (indica que se quiere enviar algo al servidor ese algo viene en el cuerpo de la petición)

**cabecera**

la cabecera de la petición es donde se encuentran los metadatos de la petición que se utiliza para enviar información de la petición.

* la cabecera de la petición es donde se encuentran las cabeceras de la petición.
* las cabeceras son metadatos que se envían a la petición para brindar información sobre la petición.

ejemplo:

* host: en.wikipedia.org
* Cache-control: no-cache

get /api/autores HTTP/1.1

host: en.wikipedia.org

Cache-Control: no-cache.

**Cuerpo de la petición**

* El cuerpo de una petición es donde se coloca la información adicional que se desea enviar al servidor. (en get no se usa el cuerpo por que no se envían datos complejos, para el caso del post si)

ejemplo:

HOla

ejemplo 2

{

"nombre": "David Martinez Osor",

"Edad":20,

"creditCard": "5521410059378570",

"url":"https://localhost:44304/api/autores"

}

post/api/autores HTTP/1.1

Host: miWebApi.com

Content-Type: application/json

Cache-Control: no cache

(aquí un espacio en blanco divide el cuerpo de la cabecera)

{

"nombre": "David Martinez Osor",

"Edad":20,

"creditCard": "5521410059378570",

"url":"https://localhost:44304/api/autores"

}

**Respuesta http**

cuando el cliente nos envía una petición HTTP, nuestro servidor debe de responder con una respuesta HTTP.

esta también tiene su propia estructura

* línea de estatus
* cabecera
* cuerpo, el cual es opcional

**Anatomía de una respuesta HTTP**

* en la línea de estatus, se indica el estado de la petición, es decir, si fue exitosa si hubo error, o si se requiere que se tome algún tipo de acción.
* la cabecera es un conjunto de cabeceras, igual que la cabecera de la petición (el servidor puede enviar en respuesta varias cabeceras) .
* el cuerpo es la data que el servidor quiere transmitir. (a través de esta se recibe el html de la página que se quiere visualizar)

ejemplo :

HTTP/1.1 200 OK

Date: thu, 03 jan 2019

server:gws

Accept-Ranges: bytes

Content-Lenght: 68894

Content-Type: text/html;charset=UTF-8

<!doctype html><html>

# Métodos HTTP / Verbos HTTP

* Queremos hacer manipulaciones de recursos a través de una URL
* Los métodos HTTP, son un mecanismo del protocolo HTTP los cuales nos permiten expresar la acción la cual queremos ejercer sobre un recurso.

ejemplo:

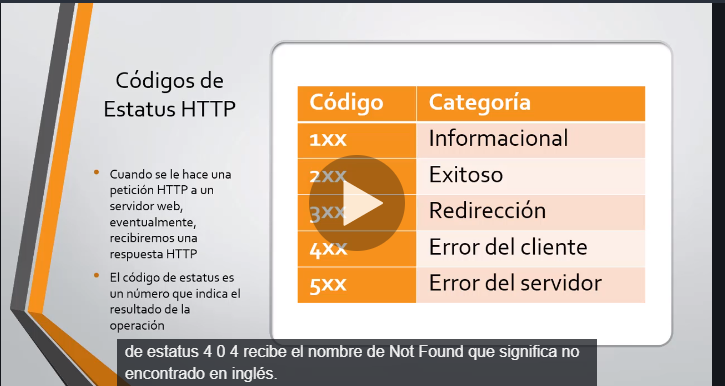
**https//miApi.com/usuarios**

los métodos http nos permiten expresar las acciones que queremos realizar sobre un recurso.

* **HTTP GET**: se utiliza para pedir información a un servidor.
* **HTTP HEAD**: hace lo mismo que el método get, sin embargo no nos trae el cuerpo de la respuesta, sino solamente la cabecera. la cabecera tiene información del mensaje pero no el mensaje como tal.
* **POST**: sirve para indicar que queremos enviar información al servidor, típicamente a través del cuerpo de la petición http.
* **HTTP PUT**: se utiliza para realizar una actualización completa de un recurso.
* todos los anteriores representan un HTTP-Crud
* **HTTP DELETE:** Sirve para expresar que se desea borrar un recurso determinado.
* **HTTP PATCH:** Este se utiliza para realizar actualizaciones parciales a un recurso.

es más rápida que el PUT, pero su implementación es un poco más trabajosa.

# CÓDIGOS DE ESTATUS DE LAS RESPUESTAS HTTP

****

los códigos de estatus también tienen nombre, por ejemplo el nombre del 404 es not found.

1XX-Respuesta informativa

* esta es una respuesta provisional y no final
* 100 continue
* 101 Switching protocols (ejemplo web socket)

2XX-códigos exitoso

* 200 ok (al actualizar un recurso con put la respuesta 200 ok indica que se ha actualizado el recurso)
* 201 Created, la acción se ha completado cuando se crea un recurso
* 202 accepted, la petición ha sido aceptada pero el procesamiento no ha sido terminado.
* 204 no content, la petición ha sido realizada con éxito y no se va a retornar un cuerpo de respuesta, solamente se envía la línea de respuesta y la cabecera de la respuesta).

3xx-codigos de redireccion

* 300 multiple choices (un video con muchos formatos)
* 301 moved Permanently
* 302 found, el registro se encuentra en una nueva uri
* 303 see other, indica que la respuesta al recurso puede ser encontrada en otro uri.
* 307 Temporary redirect.
* 308 Permanent redirect

4xx-códigos de error del cliente.

* 400 bad request
* 401 unauthorized
* 403 forbidden
* 404 not found
* 405 Method not Alloweb.
* 408 resquest timeOut

5xx- errores del servidor

* 500 internal server error (causa casi por cualquier cosa)
* 501 not implemented (cuando el servidor no conoce y no soporta el método del cliente)
* 503 service unavailable

servicios

Servicio se le conoce a la clase que puede ser inyectada a otras clases por el framework

public void ConfigureServices(IServiceCollection services)

{

//cuando se indique en cualquier parte del proyecto la depependencia, con IClase b será suplida con la claseB

services.AddTransient<IClaseB, ClaseB>();

services.AddDbContext<ApplicationDbContexts>(options => options.UseSqlServer(Configuration.GetConnectionString("DefaultConnnection")));

services.AddControllers().

AddNewtonsoftJson(options =>

options.SerializerSettings.ReferenceLoopHandling = Newtonsoft.Json.ReferenceLoopHandling.Ignore);

}

Tiempo de vida

Los servicios pueden tener distintos tiempos de vida, esto es el tiempo que se nos sirve una clase o una instancia de una clase distinta.

Transient : cada vez que un servicio sea solicitado se va a servir una nueva instancia de una clase.

services.AddTransient<IClaseB, ClaseB>();

Scoped

Estos son creados uno por petición http, si diferentes clases solicitan el mismo servicio durante una petición http se les va a entregar la misma instancia del servicio.

addScope es usada cuando se usa una clase que depende de un contexto de datos se debe servir la clase usando addscope

services.AddScoped<IClaseB, ClaseB>();

Singleton

Siempre se nos va a dar la misma instancia, solamente habrá variación cuando el servidor sea apagado y encendido.

services.AddSingleton<IClaseB, ClaseB>();

# Loggers

cuando se habla de logger se refiere a que desde la aplicación se emiten mensajes desde la api para permitir visualizar dichos mensajes escritos desde la app, son muy útiles para depurar y conocer qué es lo que está sucediendo en una ejecución.

**ilogger service**

private readonly ILogger<AutoresController> logger;

public AutoresController(ApplicationDbContexts context, IClaseB claseB, ILogger<AutoresController>logger)

{

this.Context = context;

this.claseB = claseB;

this.logger = logger;

}

[HttpGet("/listado")]

[HttpGet("listado")]

public ActionResult<IEnumerable<Autor>> Get()

{

logger.LogInformation("Obteniendo listado de autores");

claseB.HacerAlgo();

return Context.Autores.Include(x => x.Libros).ToList();

}

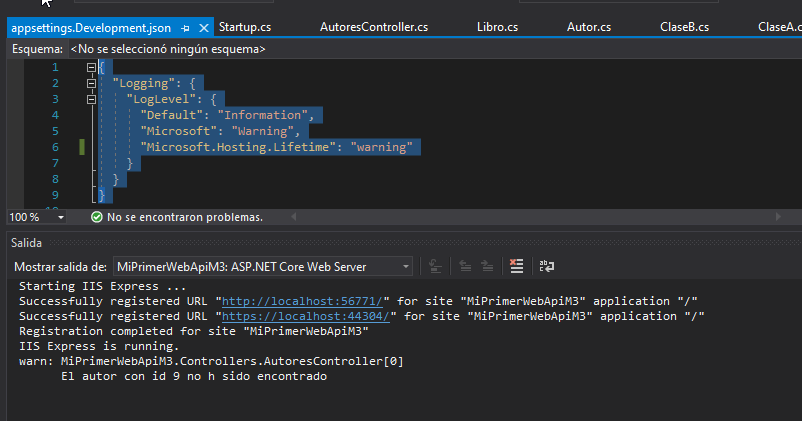
**Niveles de mensajes**

1. Critical (nivel más alto)
2. Error
3. Warning
4. Information
5. Debug
6. Trace (nivel más bajo)

para ver estos mensajes en la consola se puede desde la ventana output o salida en español.

estos errores se pueden configurar por niveles en

appsettings.Development



para aplicar un nivel a todo el proyecto se puede hacer desde el “Default”, por ejemplo si se aplica Warning en este nivel todos los mensajes que se generen iniciaran desde

1. Critical (nivel más alto)
2. Error
3. Warning (más bajo)

pero también se puede especificar un controlador en esta configuración, creando una categoría específica para ese controlador ejemplo:

{

"Logging": {

"LogLevel": {

"Default": "Information",

"Microsoft": "Warning",

"MiPrimerWebApiM3.Controllers.AutoresController": "Warning",

"Microsoft.Hosting.Lifetime": "warning"

}

}

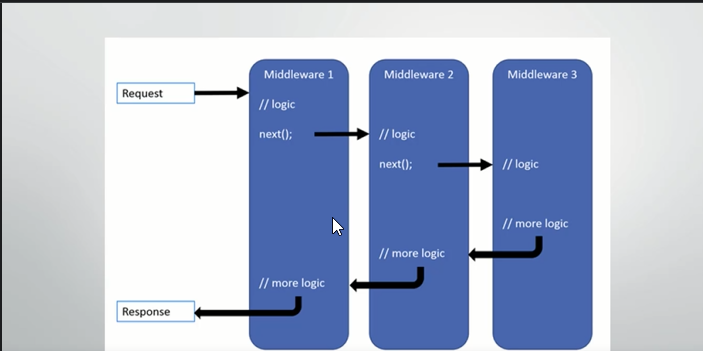
}

# Middleware

* una tubería es una cadena de procesos conectados de tal forma que la salida de cada elemento de la cadena es la entrada del próximo.
* a cada uno de los procesos le llamamos middleware.
* MVC es un middleware de los tantos que hay.
* El orden de los middleware es importante

un middleware puede detener un proceso y dar una respuesta al usuario.

los middleware se trabajan en la clase startup en el método configure



# Filtros

* Los filtros nos ayudan a correr código en determinados momentos del ciclo de vida del procesamiento de una petición HTTP.
* los filtros son útiles cuando tenemos la necesidad de ejecutar una lógica en varias acciones de varios controladores y queremos evitar tener que repetir código.
* Un filtro más utilizado es el de autorizacion.

# Tipos de filtros

* Filtros de autorización (determina si un usuario tiene acceso)
* Filtros de Recursos (validaciones generales o capa de cache)
* filtros de acción (se ejecutan antes y después de una acción, se pueden usar para manipular la información enviada a una acción o la información retornada por los mismos)
* filtros de excepción (cuando se presenta una excepción no atrapada en un try cathc )
* Filtros de resultado (antes y después de un actioresult)

los filtros se pueden aplicar a nivel de un controlador de tal modo que aplica a todas las acciones de un controlador.

también se pueden aplicar a nivel del webapi.

ejemplo de filtro es cache response.

ejemplo de filtro de cache.

en la clase startup se debe agregar el services.AddResponseCaching();

public void ConfigureServices(IServiceCollection services)

{

//cuando se indique en cualquier parte del proyecto la depependencia, con IClase b sera suplida con la claseB

//services.AddTransient<ClaseB2>();

services.AddResponseCaching();//habilitando un conjunto de servicio para la funcionalidad de guardar info en cache.

services.AddTransient<IClaseB, ClaseB>();

services.AddDbContext<ApplicationDbContexts>(options => options.UseSqlServer(Configuration.GetConnectionString("DefaultConnnection")));

services.AddControllers().

AddNewtonsoftJson(options =>

options.SerializerSettings.ReferenceLoopHandling = Newtonsoft.Json.ReferenceLoopHandling.Ignore);

}

// This method gets called by the runtime. Use this method to configure the HTTP request pipeline.

public void Configure(IApplicationBuilder app, IWebHostEnvironment env)

{

if (env.IsDevelopment())

{

app.UseDeveloperExceptionPage();

}

app.UseHttpsRedirection();

app.UseResponseCaching();

app.UseRouting();

app.UseAuthorization();

app.UseEndpoints(endpoints =>

{

endpoints.MapControllers();

});

}

Filtros Personalizados

* podemos crear nuestros propios filtros
* crearemos un filtro de acción
* para crear un filtro de accion se usan IActionFilter o IAsyncActionFilter

filtro de accion

para crear el filtro de acción se agrega la carpeta Helpers en el proyecto, dentro de ella se agrega la clase MiFiltroDeAccion.cs, que contiene:

using Microsoft.AspNetCore.Mvc.Filters;

//using MiPrimerWebApiM3.Services;

using System;

using System.Collections.Generic;

using System.Linq;

using System.Threading.Tasks;

using Microsoft.Extensions.Logging;

namespace MiPrimerWebApiM3.Helpers

{

public class MiFiltroDeAccion : IActionFilter

{

private readonly ILogger<MiFiltroDeAccion> logger;

public MiFiltroDeAccion(ILogger<MiFiltroDeAccion> logger)

{

this.logger = logger;

}

public void OnActionExecuted(ActionExecutedContext context)

{

logger.LogError("OnActionExecuted");

}

public void OnActionExecuting(ActionExecutingContext context)

{

logger.LogError("OnActionExecuting");

}

}

}

**luego se agrega en el configure services de la clase startup**

public void ConfigureServices(IServiceCollection services)

{

//cuando se indique en cualquier parte del proyecto la depependencia, con IClase b sera suplida con la claseB

//services.AddTransient<ClaseB2>();

services.AddScoped<MiFiltroDeAccion>();

services.AddAuthentication(JwtBearerDefaults.AuthenticationScheme).AddJwtBearer();

services.AddResponseCaching();//habilitando un conjunto de servicio para la funcionalidad de guardar info en cache.

services.AddTransient<IClaseB, ClaseB>();

services.AddDbContext<ApplicationDbContexts>(options => options.UseSqlServer(Configuration.GetConnectionString("DefaultConnnection")));

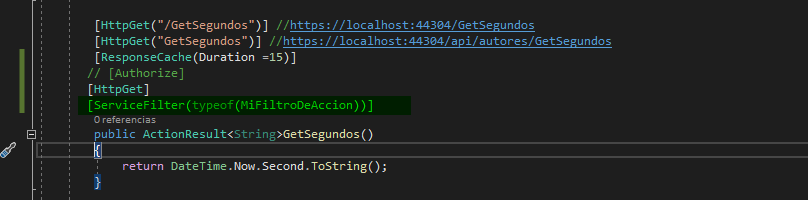
services.AddControllers().

AddNewtonsoftJson(options =>

options.SerializerSettings.ReferenceLoopHandling = Newtonsoft.Json.ReferenceLoopHandling.Ignore);

}

para llamarlo desde mi controlador se hace de la siguiente manera, se agrega [ServiceFilter(typeof(MiFiltroDeAccion))], por que se está usando inyección de dependencias en la clase MiFiltroDeAccion



se pueden crear filtros globales

filtro de excepciones, con este filtro se puede personalizar las excepciones

Clase para excepcion

using Microsoft.AspNetCore.Mvc.Filters;

using System;

using System.Collections.Generic;

using System.Linq;

using System.Threading.Tasks;

namespace MiPrimerWebApiM3.Helpers

{

public class MiFiltroDeExcepcion : ExceptionFilterAttribute

{

public override void OnException(ExceptionContext context)

{

}

}

}

en el start up se agrega en configure services

public void ConfigureServices(IServiceCollection services)

{

//cuando se indique en cualquier parte del proyecto la depependencia, con IClase b sera suplida con la claseB

//services.AddTransient<ClaseB2>();

services.AddScoped<MiFiltroDeAccion>();

services.AddAuthentication(JwtBearerDefaults.AuthenticationScheme).AddJwtBearer();

services.AddResponseCaching();//habilitando un conjunto de servicio para la funcionalidad de guardar info en cache.

services.AddTransient<IClaseB, ClaseB>();

services.AddDbContext<ApplicationDbContexts>(options => options.UseSqlServer(Configuration.GetConnectionString("DefaultConnnection")));

services.AddControllers().

AddNewtonsoftJson(options =>

options.SerializerSettings.ReferenceLoopHandling = Newtonsoft.Json.ReferenceLoopHandling.Ignore);

services.AddMvc(options =>

{

options.Filters.Add(new MiFiltroDeExcepcion());

}).SetCompatibilityVersion(CompatibilityVersion.Version\_3\_0);

}

# IHostedService

en ocasiones se necesitará ejecutar una tarea recurrente, como para consumir un web service y mantener actualizada nuestra base de datos, por ejemplo si cambia la tasa del dólar se puede obtener con esta interfaz.

* es una interfaz que nos permite crear un servicio el cual va a ejecutar una función cuando levantemos el servidor el web host y cuando lo detengamos.

ejemplos:

* uso básico de IHostedService
* Hacer Tareas recurrentes con IhostedService
* Múltiples IhostedServices corriendo simultáneamente

# IHostedService y Entity Framework

* El dbcontext de ser servido en modo scoped
* el hosted service es transient
* IserviceProvider para crear un contexto

Manipulación de recursos

DTOs o viewModels

objeto de transferencia de datos, se utilizan para representar una entidad de nuestro proyecto, no es buena idea retornar entidades desde nuestras acciones

es mejor retornar objetos de transferencia de datos (DTO’s)

los DTOs, nos ayudan a que distintas partes puedan evolucionar, de manera independiente sin afectar a dichas partes, esto se conoce como principio de responsabilidad única.

si algun dia se crea un nuevo campo en la clase autor eso no afecta mi autorDTO.

automaper: sirve para

Packagemanager console

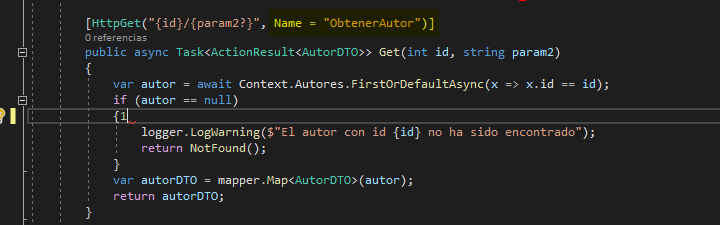
Install-Package AutoMapper.Extensions.Microsoft.DependencyInjection

funciona con clases y con lista de elementos

para poder usar automaper tambien se debe crear una carpeta utilidades en donde se va a realizar el mapero de la clase DTO a la clase de nuestra entidad

Creando recursos con POST

* cuando creamos un recurso, lo correcto es que nuestro web API retorne un código estatus 201 created.
* una clase llamada CreatedAtRouteResult nos ayuda con esto.
* debemos ponerle un nombre a nuestras reglas de ruteo.



actulizaciones parciales HTTP PATCH

se utiliza el método HTTP PATCH, para actualizar parcialmente un recurso

¿cómo funciona?

el cliente nos va a enviar un cuerpo en la petición HTTP, en donde nos indicará qué campos son los que quiere actualizar.

se utiliza JASON PATCH (RFC 6902)

JASON PATCH

[

{"op": "replace","path":"/nombre","Value":"FelipeGavilan"},

{ "op": "replace","path":"/fechaNacimiento","Value":"01-10-1990"}

]

con asp.net.netcore mayor a 3.1 se debe instalar en el aminstrador de paquetes

install-package Microsoft.AspnetCore.JsonPatch

**ejemplo sin DTO**

[HttpPatch("{id}")]

public async Task<ActionResult> Patch(int id, [FromBody] JsonPatchDocument<Autor> jsonPatchDocument)

{

if (jsonPatchDocument == null)

{

return BadRequest();

}

var autorDB = await Context.Autores.FirstOrDefaultAsync(x => x.id == id);

if(autorDB== null)

{

return NotFound();

}

jsonPatchDocument.ApplyTo(autorDB, ModelState);

var isValid = TryValidateModel(autorDB);

if (!isValid)

{

return BadRequest(ModelState);

}

await Context.SaveChangesAsync();

return NoContent();

}

ejemplo con DTO

se agrega en el start up

Configuration.CreateMap<AutorCreacionDTO, Autor>().ReverseMap();

[HttpPatch("{id}")]

public async Task<ActionResult> Patch(int id, [FromBody] JsonPatchDocument<AutorCreacionDTO> jsonPatchDocument)

{

if (jsonPatchDocument == null)

{

return BadRequest();

}

var autorDB = await Context.Autores.FirstOrDefaultAsync(x => x.id == id);

if (autorDB == null)

{

return NotFound();

}

var autorDTO = mapper.Map<AutorCreacionDTO>(autorDB);

jsonPatchDocument.ApplyTo(autorDTO, ModelState);

var isValid = TryValidateModel(autorDB);

mapper.Map(autorDTO, autorDB);

if (!isValid)

{

return BadRequest(ModelState);

}

await Context.SaveChangesAsync();

return NoContent();

}

Instalando entity framework core

se busca en el nuget

entityframeworkCore.sqlserver

entityframeworkCore.tool (para obtener las herramientas en visual studio)