### Perfiles en Visual Studio .Net

El los proyectos .Net en la carpeta Properties, en el archivo launchSettings.json, está la sección profiles.

Se puede llegar en Visual Studio, botón derecho en el proyecto => propiedades del proyecto. Se abre la sección Debug => General y se presiona el link “Open debug launch profiles UI”

Captura de pantalla de un celular

Descripción generada automáticamente

Lo que da paso a la pantalla de configuración de cada perfil

Texto

Descripción generada automáticamente

En el perfil para https, se puede configurar que en vez de caer en la raíz de la URL de la aplicación después de iniciar el servicio, se caiga directamente en el path designado. En el ejemplo todoitems

Captura de pantalla de un celular

Descripción generada automáticamente

### Usar Dockers para el desarrollo

En el proyecto, click derecho, Add -> Docker Support: Linux

Tabla

Descripción generada automáticamente

Esto crea el Dockerfile completo para Build y publicación:

Texto

Descripción generada automáticamente

En la ventana de output, si se elije Container Tools, se ven los comando para

Texto

Descripción generada automáticamente

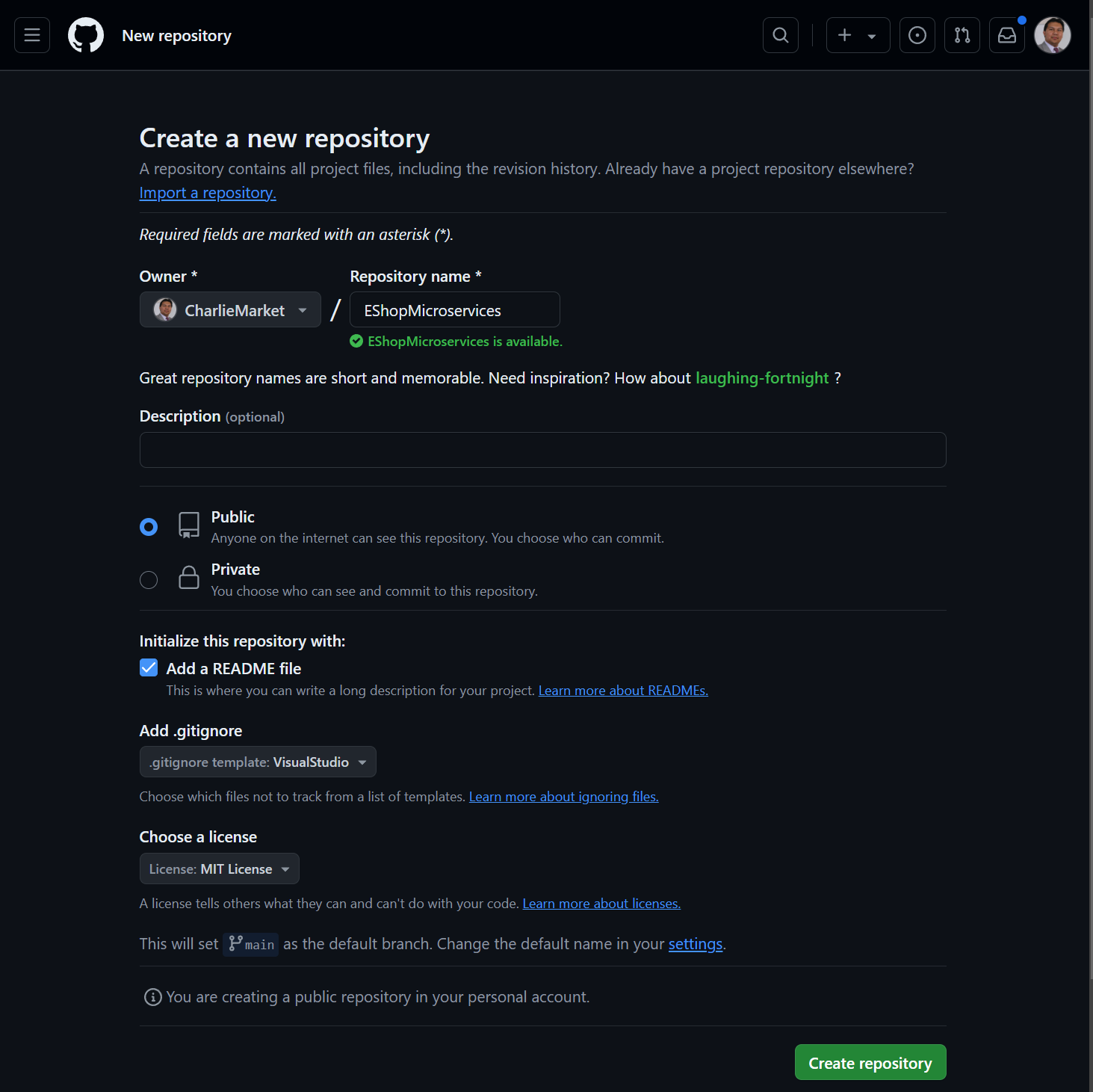
Finalmente ejecutamos la API y podemos USAR EL DEBUGGER !!

### Web API con Postgresql y arquitectura vertical

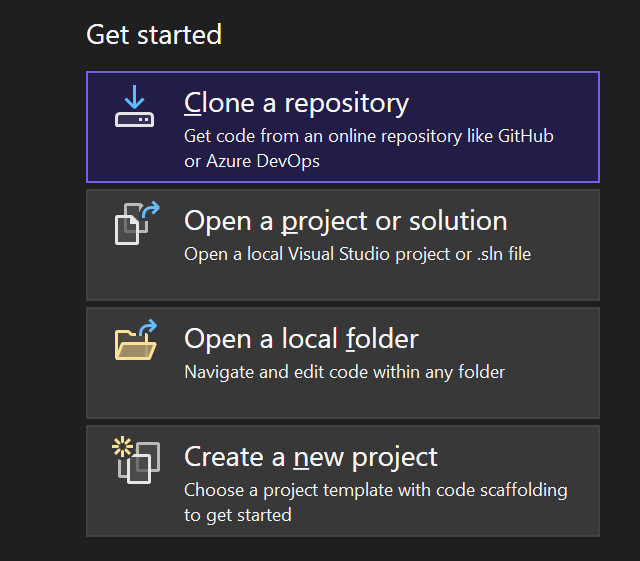
Para usar postgresql como base documental usamos la librería Martendb (<https://martendb.io/>)

#### Iniciar desde github

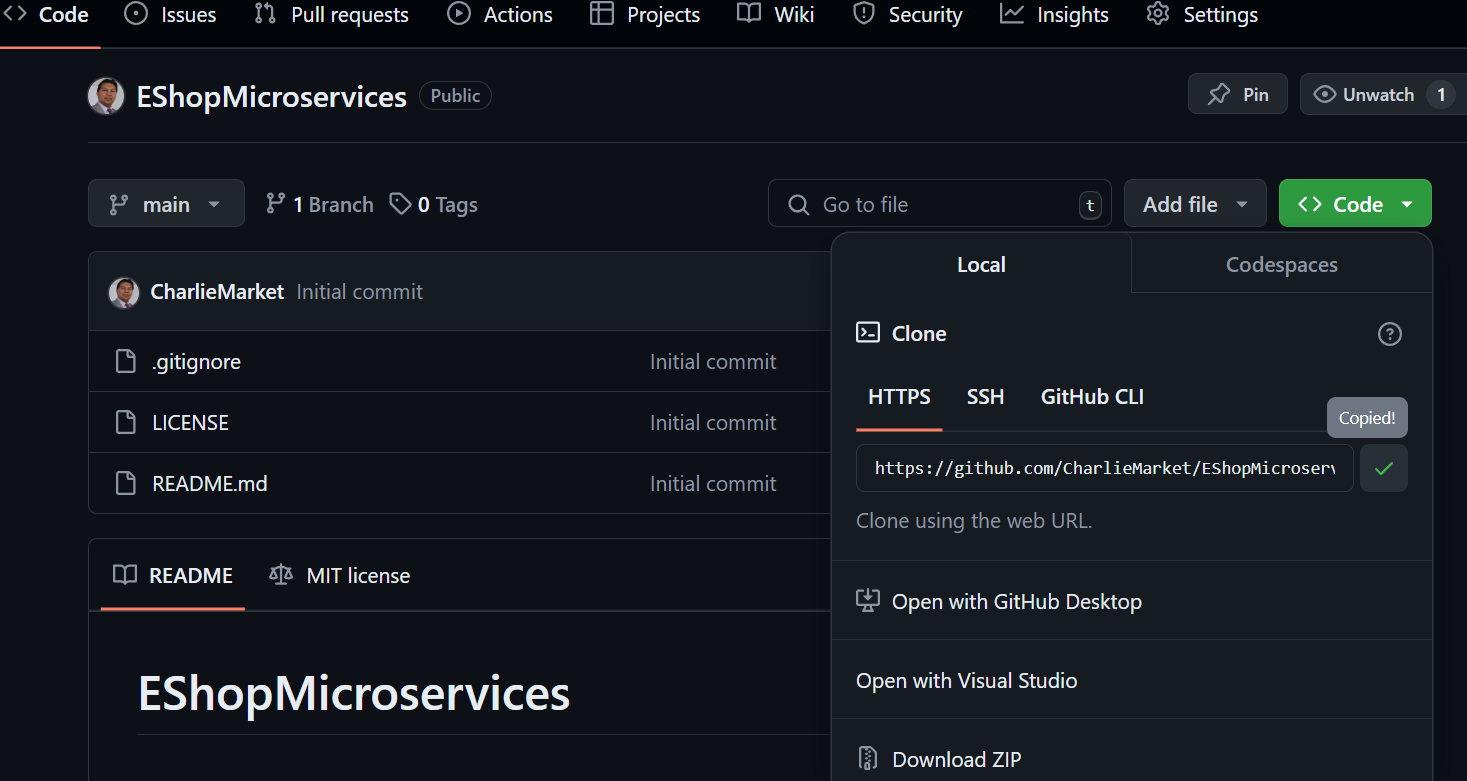
Iniciamos creando un repositorio en github:



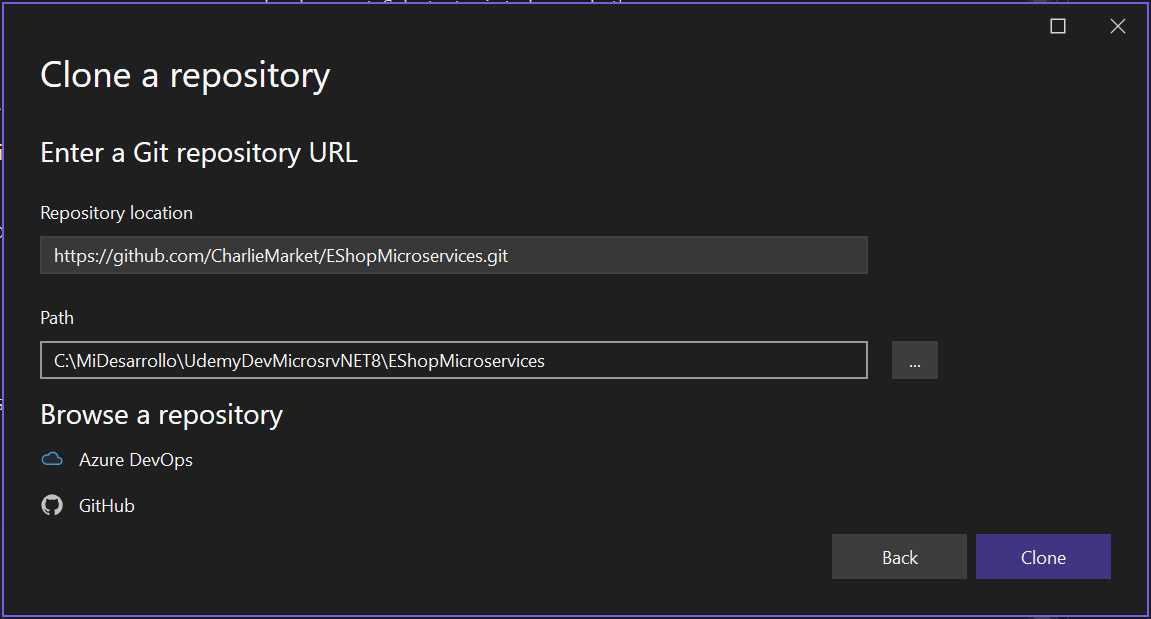
En Visual Studio 2022 clonamos el repositorio



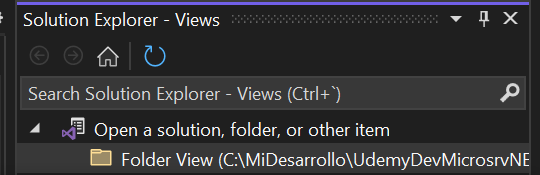
Copiamos la url desde github

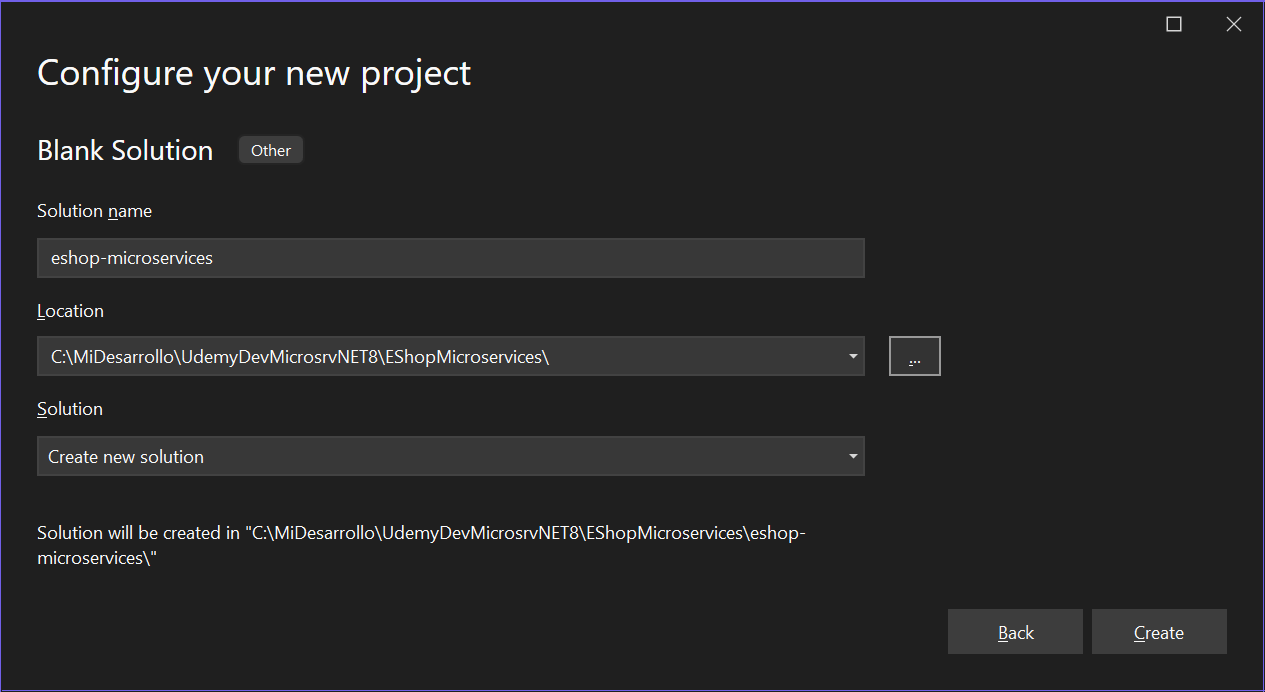


Y lo clonamos



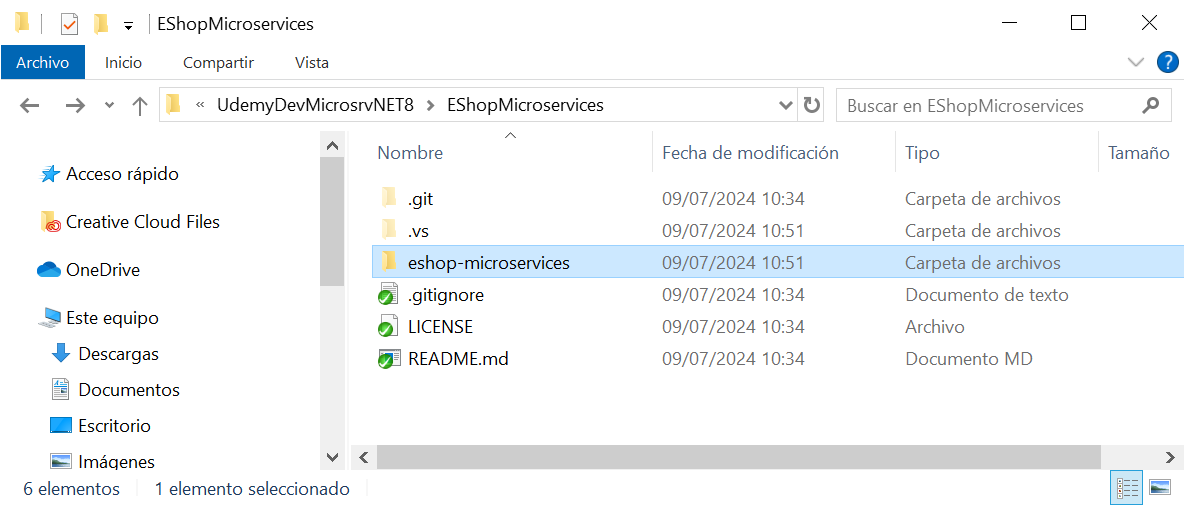
Nos posicionamos en el Solution Explorer en Folder View y en el menú creamos un proyecto nuevo a partir del template *Blank Solution*





Es MUY importante que se cree en la misma carpeta que clonamos y que tiene la carpeta .git

Una vez terminado el proceso, la solución creó la carpeta *eshop-microservices* dentro de la carpeta controlada por Git *EShopMicroservices*. Para que sea ordenado cambiaremos el nombre de la carpeta de la solución por ***src*** .



### Web API con CQRS con MediatR y abastracción de ICommand, IQuery, ICommandHandler e IQueryHandler

### Web API uso de Carter para extender minimal API’s en módulos

### Web API con Martens para Postgresql NoSQL

### Web API con MediatR Pipeline Behavior cross cutting concerns: Logging Behavior

### Web API cross cutting concerns: Validate con fluentvalidation usando Minimal API filters en la entrada del endpoint:

Jono Williams: <https://www.youtube.com/watch?v=1qJTVcR1VN8>  
<https://github.com/jonowilliams26/youtube-videos/tree/main/RequestValidationInMinimalAPIs>

Agregar al proyecto BuildingBlocks la referencia al framework ***Microsoft.AspNetCore.App***

<ItemGroup>

<FrameworkReference Include="Microsoft.AspNetCore.App" />

En las estructuras de request de los endpoints hay que declarar todas las propiedades que sean structs como nuleables. Ej. Guid? , decimal?, int?, etc. Para que si no vienen se puedan chequear contra null.

### Web API para POST, PUT, DELETE y PATCH se debe usar un filtro de con Postgresql y arquitectura vertical

### Web API con Postgresql y arquitectura vertical

* Llenado de base de datos automático. A medida que se avanza en el proyecto se debe ir llenando el script de creación.
* Para detener los errores de reglas del mensaje de entrada (validación semántica) se usa FluentValidation el cual confía en el método de extensión ValidationExtensions.WithRequestValidation que se agrega a cada endpoint (MapGet, MapPost, MapPut, MapDelete)
* Para no usar excepciones cuando el endpoint recibe el mensaje de entrada, ya validado semánticamente, se usa la librería OneOf, la que se aplica a la definición del commandResult o queryResult que se enviará a MediatR (en el archivo del handler), OneOf<resultado, error1, error2>, como regla el primero siempre es el resultado exitoso.
* Usar nullable reference en todo el proyecto para tener absolutamente presente cuando algo es nulo o no lo es. https://dotnetteach.com/blog/csharp-nullable-project-setting

En la configuración del proyecto ver:

<Project Sdk="Microsoft.NET.Sdk.Web">

<PropertyGroup>

<TargetFramework>net8.0</TargetFramework>

<Nullable>enable</Nullable>

* sdfsd

### Los resultados de las API que vienen del los enrutadores (MapGet, MapXXX) deben ser descriptivas. (<https://learn.microsoft.com/es-mx/aspnet/core/fundamentals/openapi/aspnetcore-openapi?view=aspnetcore-8.0&tabs=visual-studio>)

**Describir los tipos de respuesta**

OpenAPI permite proporcionar una descripción de las respuestas que devuelve una API. Las API mínimas admiten tres estrategias para establecer el tipo de respuesta de un punto de conexión:

* Mediante el método de extensión [Produces](https://learn.microsoft.com/es-es/dotnet/api/microsoft.aspnetcore.http.openapiroutehandlerbuilderextensions.produces) en el punto de conexión
* Mediante el atributo [ProducesResponseType](https://learn.microsoft.com/es-es/dotnet/api/microsoft.aspnetcore.mvc.producesresponsetypeattribute) en el controlador de ruta
* Mediante la devolución de [TypedResults](https://learn.microsoft.com/es-es/dotnet/api/microsoft.aspnetcore.http.typedresults) desde el controlador de ruta

El método de extensión Produces se puede usar para agregar metadatos de Produces a un punto de conexión. Cuando no se proporciona ningún parámetro, el método de extensión rellena los metadatos del tipo de destino bajo un código de estado 200 y un tipo de contenido application/json.

C#Copiar

app

.MapGet("/todos", async (TodoDb db) => await db.Todos.ToListAsync())

.Produces<IList<Todo>>();

Al usar [TypedResults](https://learn.microsoft.com/es-es/dotnet/api/microsoft.aspnetcore.http.typedresults) en la implementación del controlador de ruta de un punto de conexión, se incluyen automáticamente los metadatos del tipo de respuesta para dicho punto. Por ejemplo, el código siguiente anota automáticamente una respuesta en el punto de conexión bajo el código de estado 200 con un tipo de contenido application/json.

C#Copiar

app.MapGet("/todos", async (TodoDb db) =>

{

var todos = await db.Todos.ToListAsync());

return TypedResults.Ok(todos);

});

**Establecimiento de respuestas para ProblemDetails**

Al establecer el tipo de respuesta para los puntos de conexión que pueden devolver una respuesta ProblemDetails, se puede usar el método de extensión [ProducesProblem](https://learn.microsoft.com/es-es/dotnet/api/microsoft.aspnetcore.http.openapiroutehandlerbuilderextensions.producesproblem), [ProducesValidationProblem](https://learn.microsoft.com/es-es/dotnet/api/microsoft.aspnetcore.http.openapiroutehandlerbuilderextensions.producesvalidationproblem) o [TypedResults.Problem](https://learn.microsoft.com/es-es/dotnet/api/microsoft.aspnetcore.http.typedresults.problem?) para agregar la anotación adecuada a los metadatos del punto de conexión. Ten en cuenta que los métodos de extensión ProducesProblem y ProducesValidationProblem no se pueden usar con los [grupos de rutas](https://learn.microsoft.com/es-mx/aspnet/core/fundamentals/minimal-apis/route-handlers?view=aspnetcore-8.0#route-groups) en .NET 8 y versiones anteriores.

Cuando las estrategias anteriores no proporcionan anotaciones explícitas, el marco intenta determinar un tipo de respuesta predeterminado mediante el examen de la signatura de la respuesta. Esta respuesta predeterminada se rellena bajo el código de estado 200 en la definición de OpenAPI.

**Tipos de respuestas múltiples**

Si un punto de conexión puede devolver diferentes tipos de respuesta en escenarios distintos, puedes proporcionar metadatos de las siguientes maneras:

* Llama al método de extensión [Produces](https://learn.microsoft.com/es-es/dotnet/api/microsoft.aspnetcore.http.openapiroutehandlerbuilderextensions.produces) varias veces, como se muestra en el ejemplo siguiente:

C#Copiar

app.MapGet("/api/todoitems/{id}", async (int id, TodoDb db) =>

await db.Todos.FindAsync(id)

is Todo todo

? Results.Ok(todo)

: Results.NotFound())

.Produces<Todo>(StatusCodes.Status200OK)

.Produces(StatusCodes.Status404NotFound);

* Usa [Results<TResult1,TResult2,TResultN>](https://learn.microsoft.com/es-es/dotnet/api/microsoft.aspnetcore.http.httpresults.results-6) en la firma y [TypedResults](https://learn.microsoft.com/es-es/dotnet/api/microsoft.aspnetcore.http.typedresults) en el cuerpo del controlador, como se muestra en el ejemplo siguiente:

C#Copiar

app.MapGet("/book/{id}", Results<Ok<Book>, NotFound> (int id, List<Book> bookList) =>

{

return bookList.FirstOrDefault((i) => i.Id == id) is Book book

? TypedResults.Ok(book)

: TypedResults.NotFound();

});

Los [tipos de unión](https://en.wikipedia.org/wiki/Union_type) Results<TResult1,TResult2,TResultN> declaran que un controlador de ruta devuelve varios tipos de elementos IResult que implementan tipos concretos, y cualquiera de esos tipos que implementa IEndpointMetadataProvider contribuirá a los metadatos del punto de conexión.

### Web API Health Checks

Agregar los Health Checks en la configuración

builder.Services.AddHealthChecks();

var app = builder.Build();

app.UseHealthChecks("/health");

Para chekear servicios de varios tipos existe la librería open source:

AspNetCore.Diagnostics.HealthChecks

<https://github.com/Xabaril/AspNetCore.Diagnostics.HealthChecks>

Ej: Para Postgres usaremos el Nuget AspNetCore.HealthChecks.NpgSql

Y cambiamos la configuración del servicio

builder.Services.AddHealthChecks()

.AddNpgSql(builder.Configuration.GetConnectionString("ReadWriteConnection")!);

Para ver los detalles del check se puede usar HealthCheckUI, de la misma librería

<https://github.com/Xabaril/AspNetCore.Diagnostics.HealthChecks?tab=readme-ov-file#HealthCheckUI>

Se agrega el Nuget **AspNetCore.HealthChecks.UI.Client**

Y se cambia la configuración:

// Configuramos Health Checks, con la ruta correspondiente

app.UseHealthChecks("/health", new HealthCheckOptions

{

ResponseWriter = UIResponseWriter.WriteHealthCheckUIResponse

});

### Pasar la el API Project a Contenedores Dockers

Se debe agregar Docker Support al proyecto de API

Captura de pantalla de computadora

Descripción generada automáticamente

Como ya estaba creado el Dockerfile, le pedimos que lo cree de nuevo

Interfaz de usuario gráfica, Texto, Aplicación

Descripción generada automáticamente

Para que todo quede orquestado en el proyecto Docker compose, debemos agregar soporte de orquestación al proyecto Catalog.API

Captura de pantalla de computadora

Descripción generada automáticamente

Y al hacerlo, el Docker-compose cambia de:

Captura de pantalla de computadora

Descripción generada automáticamente

A este nuevo:

Captura de pantalla de computadora

Descripción generada automáticamente

Y en el proyecto docker-compose, se cambia el docker-compose.override.yml a:

### Texto Descripción generada automáticamente

En donde en el servicio catalogdb agregamos container\_name con el mismo nombre, además agregamos la variable ambiente ConnectionStrings\_\_Database que debe tener \_\_ para representar que hace override de la configuración original de Catalog.API en appsettings.json

Interfaz de usuario gráfica, Texto

Descripción generada automáticamente

También agregamos la sección depends\_on: con – catalogdb que es el primer servicio declarado y en connection string cambiamos el servidor por el nombre del contenedor dentro de la red interna que se forma en el compose.

Finalmente mapeamos las puertas internas con las externas 6000 y 6060.

Para ejecutar el compose se usa el comando normal:

docker-compose -f docker-compose.yaml -f docker-compose.override.yaml up -d

Pero si queremos poder “debuguear” en los contenedores entonces debemos ejecutar desde el visual studio.

### Unit testing with TUnit para plataforma moderna Microsoft desde .net 8

### Unit testing with TUnit para plataforma moderna Microsoft desde .net 8

### <https://github.com/thomhurst/TUnit>

### <https://thomhurst.github.io/TUnit/docs/intro/>

### <https://www.youtube.com/watch?v=dtdgm8lKJZU>