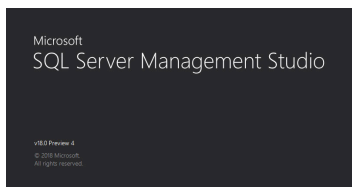


Creación de un API con arquitectura Hexagonal



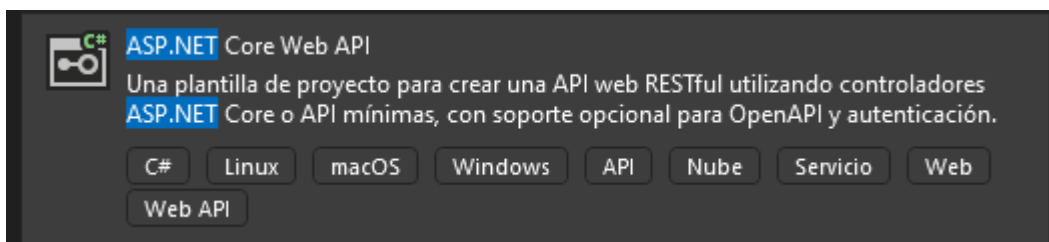
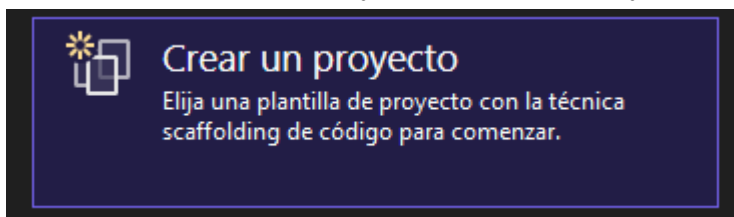
<https://visualstudio.microsoft.com/es/>



<https://learn.microsoft.com/es-es/sql/ssms/download-sql-server-management-studio-ssms?view=sql-server-ver16>

Como crear una API

Vamos a abrir visual studio y crear un nuevo proyecto “ASP.NET core Web API”



Configure su nuevo proyecto

ASP.NET Core Web API

C#

Linux

macOS

Windows

API

Nube

Servicio

Web

Web API

Nombre del proyecto

aspDotNetBlankProject

Ubicación

C:\Users\docto\source\repos

Nombre de la solución ⓘ

aspDotNetBlankProject

☐ Colocar la solución y el proyecto en el mismo directorio

Proyecto se creará en "C:\Users\docto\source\repos\aspDotNetBlankProject\aspDotNetBlankProject\"

Información adicional

ASP.NET Core Web API

C#

Linux

macOS

Windows

API

Nube

Servicio

Web

Web API

Framework ⓘ

.NET 8.0 (Compatibilidad a largo plazo)

Authentication de campo ⓘ

Ninguno

☒ Configurar para HTTPS ⓘ

☐ Habilitar compatibilidad con el contenedor ⓘ

SO del contenedor ⓘ

Linux

Tipo de compilación de contenedor ⓘ


Dockerfile

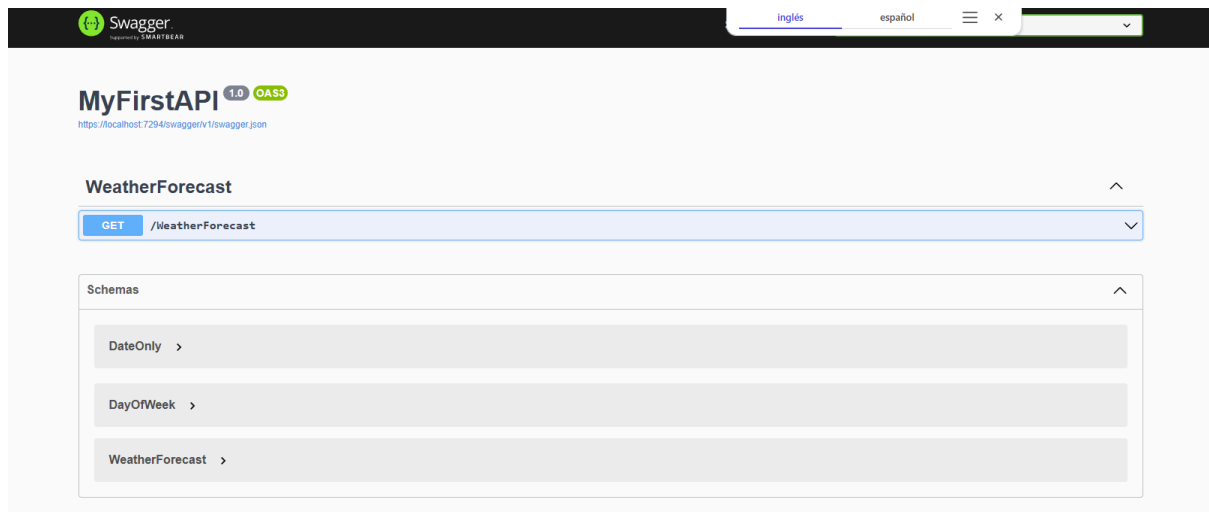
☒ Habilitar compatibilidad con OpenAPI ⓘ

☐ No usar instrucciones de nivel superior ⓘ

☒ Utilizar controladores ⓘ

☐ Inscribirse en la orquestación de .NET Aspire ⓘ

Una vez creado lo podemos ejecutar al darle play  [https](https://) y nos enviará a un swagger con el código de ejemplo del clima, este código lo vamos a borrar.

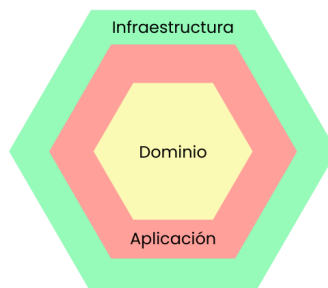


Borrar los siguientes archivos:

aspDotNetBlankProject\aspDotNetBlankProject\WeatherForecast.cs

aspDotNetBlankProject\aspDotNetBlankProject\Controllers\WeatherForecastController.cs

La arquitectura hexagonal



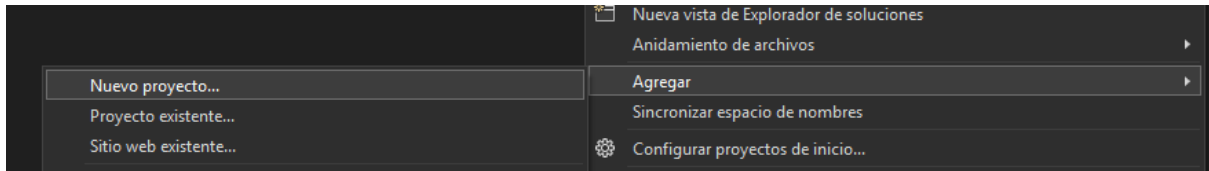
Esta arquitectura consta de 3 capas:

- Dominio: Resolver la lógica de negocio.
- Aplicación: Intermediar entre el dominio y la capa de infraestructura.
- Infraestructura: Conectarse a base de datos.

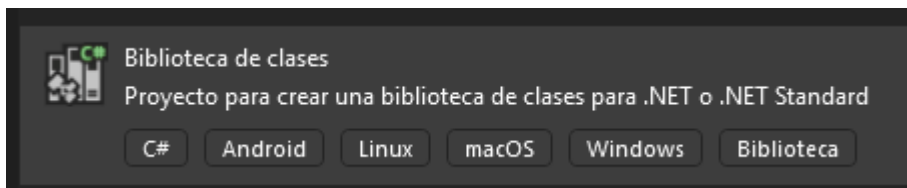
Como crear una capa

Una capa es un proyecto que tiene una única responsabilidad, vamos a crear nuestras 3 capas

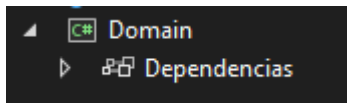
Basta solo con darle click derecho a la solución y agregar un nuevo proyecto:



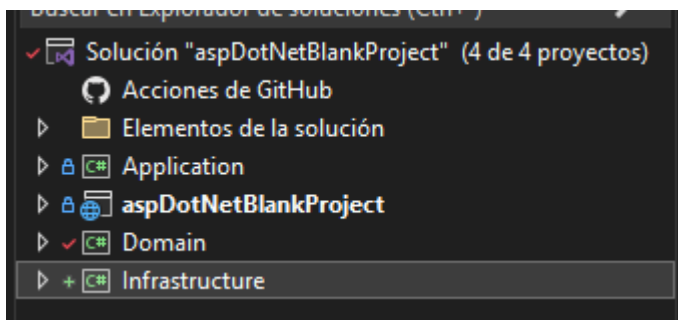
Y Agregamos una biblioteca de clases:



Por ejemplo creamos la capa de dominio el cual vendrá con un archivo de ejemplo el cual procederemos a borrar.



Y luego creamos nuestras capas de Infrastructure y Application



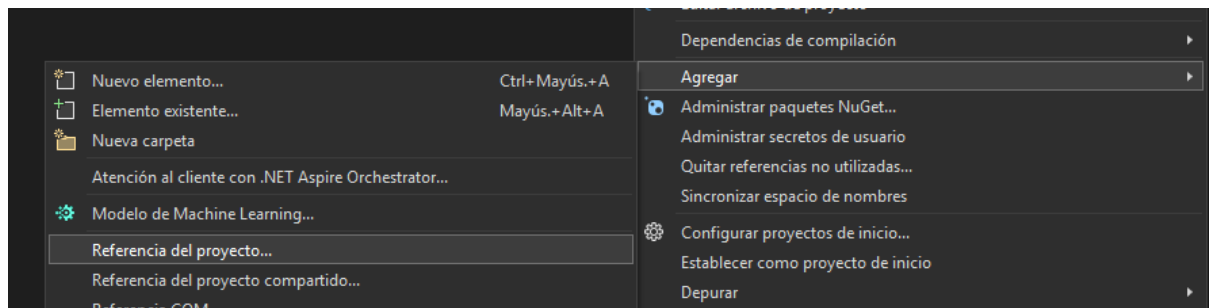
Ahora vamos a definir las capas internas y su responsabilidad:

Nr	Tipo	Ubicación	Descripción y responsabilidad.
01	Entidad de Dominio	Domain	<p>Es la encargada de contener todas las entidades del dominio de negocio como por ejemplo: Usuarios, Clientes, ventas, Productos.</p> <p>Su responsabilidad es modelar las entidades de bases de datos.</p>
02	Puertos	Domain	<p>Serán los encargados de conectar las capas e inyectar dependencias.</p> <p>Su responsabilidad es asegurar que los repositorios tengan una consistencia de métodos.</p>
03	Servicio	Domain	<p>Será el encargado de usar los repositorios para manipular la información.</p> <p>Su responsabilidad es ejecutar la lógica de los casos de uso.</p>
04	Entidades de aplicación	Application	<p>Se utilizará el patrón CQRS para modelar los récords y los handler. Cada 01 tendrá su propia carpeta que a su vez va a contener 2 carpetas: "Commands" y "Querys".</p> <p>Su responsabilidad será inyectar servicios y resolver lógica de operaciones con entidades.</p>
05	Adaptador	Infrastructure	<p>Son los repositorios e implementan el contrato pactado en 02.</p> <p>Su responsabilidad será hacer valer el contrato y ser inyectados.</p>
06	Conector a base de datos	Infrastructure	<p>Se encargará de usar la librería MEntityFrameworkCore para:</p> <ul style="list-style-type: none"> • Conectarse a la base de datos. • Registrar por cada entidad su respectiva tabla en base de datos. • <p>Su responsabilidad será conectar y modelar la base de datos.</p>
07	Controlador de API	aspDotNetBlankProject	<p>Se encargará de ser el punto de conexión entre los usuarios y los url endpoint.</p> <p>Su responsabilidad es responder al usuario las solicitudes.</p>

Cómo conectar las capas

Las capas no trabajan independientemente, cada capa necesita conectarse a otra para traer por ejemplo: un modelo, servicio o inyectar.

Para conectar una capa con otra basta con darle click derecho a nuestra capa y luego seleccionar agregar.

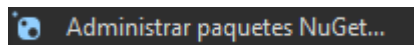


Las conexiones que realizaremos son:

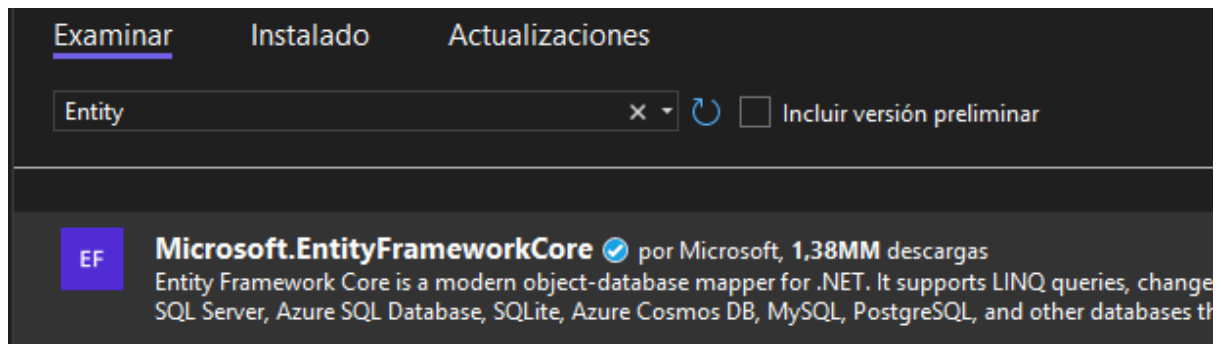
- aspDotNetBliankProject: infraestructura, Aplicación, Dominio
- Aplicación: dominio.
- Infraestructura: dominio.

Instalación de paquetes NuGet

A cada capa vamos a darle click derecho y luego administración de paquetes “NuGet”.



Y se nos abrirá la ventana de instalación de paquetes, vamos a darle click a examinar:



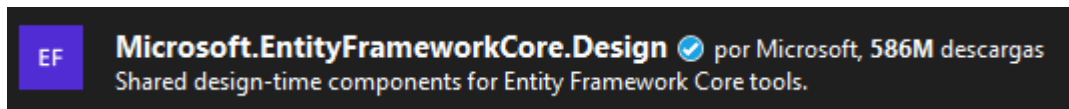
Y vamos a instalar los siguientes paquetes NuGet en las capas correspondientes:



Advertencia: en caso de fallar en el número de versión de instalación de NuGet basta con desinstalar y volver a instalar.

- **aspDotNetBlankProject:**

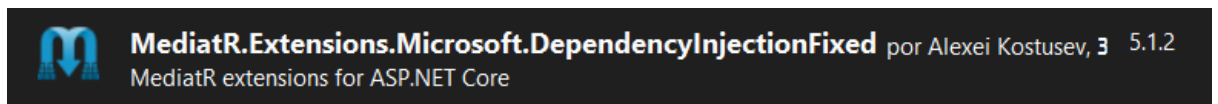
Microsoft.EntityFrameworkCore.Design
8.0.0



MediatR
12.4.1



MediatR.Extensions.Microsoft.DependencyInjectionFixed
5.1.2




- **Application:**

MediatR
12.4.1




MediatR.Extensions.Microsoft.DependencyInjectionFixed
5.1.2


**MediatR.Extensions.Microsoft.DependencyInjectionFixed** por Alexei Kostusev, 3 5.1.2
MediatR extensions for ASP.NET Core

- **Infrastructure:**

Microsoft.EntityFrameworkCore
8.0.0

**Microsoft.EntityFrameworkCore** por Microsoft, 1,38MM descargas
Entity Framework Core is a modern object-database mapper for .NET. It supports LINQ queries, change tracking, updates, and schema migrations. EF Core works with SQL Server, Azure SQL Database, SQLite, Azure Cosmos DB, MySQL, PostgreSQL, and other databases through a provider plugin API.


Microsoft.EntityFrameworkCore.SqlServer
8.0.0

**Microsoft.EntityFrameworkCore.SqlServer** por Microsoft, 605M descargas
Microsoft SQL Server database provider for Entity Framework Core.

MediatR
12.4.1

**MediatR** por Jimmy Bogard, 271M descargas 12.4.1
Simple, unambitious mediator implementation in .NET

Dapper
2.1.66

**Dapper** por Sam Saffron, Marc Gravell, Nick Craver, 436M descargas 2.1.66
A high performance Micro-ORM supporting SQL Server, MySQL, Sqlite, SqICE, Firebird etc. Major Sponsor: Dapper Plus from ZZZ Projects.

Creación de nuestra ENTIDAD de Ejemplo

Vamos a crear una entidad de ejemplo:

Example(Id, Title, Description, Information, IsDelete)

La cual nos servirá para practicar de ahora en adelante como realizar la escritura en base de datos usando solo los modelos.

aspDotNetBlankProject\Domain\Entities\Example.cs

```
namespace Domain.Entities
{
    2 referencias
    public class Example
    {
        0 referencias
        public int Id { get; set; }
        0 referencias
        public string Title { get; set; } = string.Empty;
        0 referencias
        public string Description { get; set; } = string.Empty;
        0 referencias
        public string Information { get; set; } = string.Empty;
        0 referencias
        public bool IsDeleted { get; set; }
    }
}
```

Creación del contexto “mapeador a Base de Datos”

En la actualidad existe un paquete NuGet llamado: Microsoft.EntityFrameworkCore el cual se encarga de:

- Dados los modelos convertirlos a tablas de bases de datos sin escribir código SQL.
- Conectarse a la base de datos haciendo la configuración e inyección en la main del proyecto.

aspDotNetBlankProject\Infraestructure\Context\DbContext.cs

```

using Microsoft.EntityFrameworkCore;
using Domain.Entities;

namespace MyFirstAPI.Infraestructure
{
    2 referencias
    public class PersitenceContext : DbContext
    {
        0 referencias
        public PersitenceContext(DbContextOptions<PersitenceContext> options) : base(options)
        {
        }

        //Mapers
        0 referencias
        public DbSet<Example> examples { get; set; }

        //Create entity in DB
        0 referencias
        protected override void OnModelCreating(ModelBuilder builder)
        {
            builder.Entity<Example>().ToTable("examples");
        }
    }
}

```

El código que observamos se encarga de inyectar un conector de base de datos y mapear nuestras entidades a tablas de base de datos.

Registrar e inyectar nuestro contexto de base de datos

Nuestro archivo main es aspDotNetBlankProject\aspDotNetBlankProject\Program.cs allí nosotros tenemos que gestionar la inyección de dependencias, tenemos que hacerlo justo antes de la línea de código en donde se construye la APP

```

var app = builder.Build();

```

Nosotros procederemos a realizar la siguiente operación para poder dejar nuestro contexto inyectable.

Paso 0 para registrar la conexión a base de datos importar:

```

using Microsoft.EntityFrameworkCore;
using MyFirstAPI.Infraestructure;

```

Paso 1

declarar la conexión de base de datos SQLserver, esto se hace con el string que nos proporcionó la instalación de SQL server.

Ejemplo:

```
Server=localhost\\MSSQLSERVER01;Database=master;Trusted_Connection=True;TrustServerCertificate=True;
```

Esa cadena nosotros tenemos que ponerla en el archivo:

aspDotNetBlankProject\aspDotNetBlankProject\appsettings.json

```
{
  "Logging": {
    "LogLevel": {
      "Default": "Information",
      "Microsoft.AspNetCore": "Warning"
    }
  },
  "ConnectionStrings": {
    "DefaultConnection": "Server=localhost\\MSSQLSERVER01;Database=MyFirstAPI;Trusted_Connection=True;TrustServerCertificate=True;"
  },
  "AllowedHosts": "*"
}
```

Luego este string tenemos que usarlo en el
aspDotNetBlankProject\aspDotNetBlankProject\Program.cs
Para declarar la conexión a base de datos:

```
builder.Services.AddSwaggerGen();
var connectionString = builder.Configuration.GetConnectionString("DefaultConnection");

builder.Services.AddDbContext<PersistenceContext>(options =>
    options.UseSqlServer(connectionString));

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

Ejecutar una “MIGRACIÓN” Convertir entidades en tablas de bases de datos e insertar

Lo que vamos a hacer a continuación es a través de la consola ejecutar comandos que harán que nuestras entidades se conviertan en tablas de bases de datos y luego esas tablas procederemos a meterlas dentro de nuestra base de datos.

Paso 1 instalar las herramientas de ejecución de Entity Framework:

```
dotnet tool install --global dotnet-ef
```

```
+ PowerShell para desarrolladores
PS C:\Users\docto\source\repos\aspDotNetBlankProject> dotnet tool install --global dotnet-ef
La herramienta "dotnet-ef" se actualizó correctamente de la versión "9.0.1" a la versión "9.0.2".
PS C:\Users\docto\source\repos\aspDotNetBlankProject>
```

paso 2 restaurar dependencias:

```
dotnet restore
```

```
C:\Users\docto\source\repos\aspDotNetBlankProject> dotnet restore
Determining projects to restore...
Se ha restaurado C:\Users\docto\source\repos\aspDotNetBlankProject\aspDotNetBlankProject\aspDotNetBlankProject.csproj (en 390 ms).
3 de 4 proyectos están actualizados para la restauración.
```

Paso 4 listar los proyectos:

dotnet sln list

```
PS C:\Users\docto\source\repos\aspDotNetBlankProject> dotnet sln list
Proyectos
-----
Application\Application.csproj
aspDotNetBlankProject\aspDotNetBlankProject.csproj
Domain\Domain.csproj
Infraestructure\Infrastructure.csproj
PS C:\Users\docto\source\repos\aspDotNetBlankProject>
```

Paso 5: limpiar y construir el proyecto:

dotnet clean

dotnet restore

dotnet build

Paso 3 ejecutar la migración (leer modelos construir tablas sql):

dotnet ef migrations add InitialCreate --project Infraestructure/Infrastructure.csproj
--startup-project aspDotNetBlankProject/aspDotNetBlankProject.csproj

```
C:\Users\docto\source\repos\aspDotNetBlankProject> dotnet ef migrations add InitialCreate --project Infraestructure/Infrastructure.csproj --startup-project aspDotNetBlankProject/aspDotNetBlankProject.csproj
Id started...
Id succeeded.
e. To undo this action, use 'ef migrations remove'
C:\Users\docto\source\repos\aspDotNetBlankProject>
```

paso 4 con las tablas construidas en la migración actualizar los valores de la base de datos:

dotnet ef database update --project Infraestructure/Infrastructure.csproj --startup-project
aspDotNetBlankProject/aspDotNetBlankProject.csproj

```
C:\Users\docto\source\repos\aspDotNetBlankProject> dotnet ef database update --project Infraestructure/Infrastructure.csproj --startup-project aspDotNetBlankProject/aspDotNetBlankProject.csproj
Id started...
Id succeeded.
: Microsoft.EntityFrameworkCore.Database.Command[20101]
  Executed DbCommand (11ms) [Parameters=[], CommandType='Text', CommandTimeout='30']
  SELECT 1
: Microsoft.EntityFrameworkCore.Database.Command[20101]
  Executed DbCommand (15ms) [Parameters=[], CommandType='Text', CommandTimeout='30']
```

Y por consola vamos a ver la creación de nuestra entidad de ejemplo:

```

o: Microsoft.EntityFrameworkCore.Database.Command[20101]
Executed DbCommand (7ms) [Parameters=[], CommandType='Text', CommandTimeout='30']
CREATE TABLE [examples] (
  [Id] int NOT NULL IDENTITY,
  [Title] nvarchar(max) NOT NULL,
  [Description] nvarchar(max) NOT NULL,
  [Information] nvarchar(max) NOT NULL,
  [IsDeleted] bit NOT NULL,
  CONSTRAINT [PK_examples] PRIMARY KEY ([Id])
);
Microsoft.EntityFrameworkCore.Database.Command[20101]

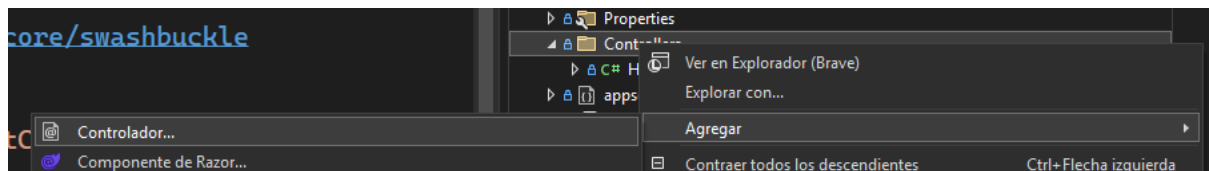
```

Creación de nuestro controlador

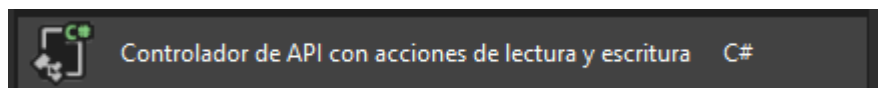
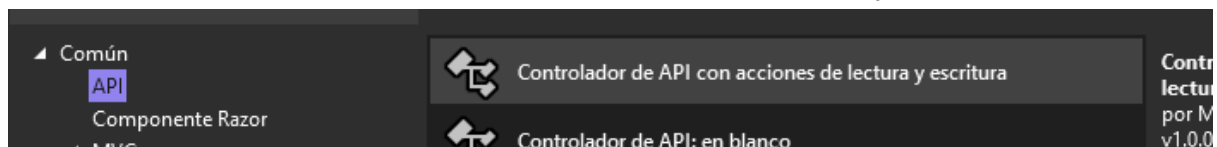
Los controladores son los encargados de exponer las URL donde los usuarios se conectarán para retornar solicitudes. Algunos ejemplos de solicitudes son:

- Login.
- Obtener todos los productos.
- Obtener un ejemplo-
- Borrar un ejemplo.
- ..

Para crear un controlador basta con darle click derecho a nuestra carpeta de controladores y luego agregar controlador.



Y vamos a seleccionar un controlador API con acciones de lectura y escritura.



El nombre de archivo para cada controlador es <Entidad>Controller.cs para nuestro caso será:

```

+ C# ExampleController.cs

```

Creación de nuestro puerto

Un puerto no es más que la celebración de un contrato por cada entidad y el contrato se describe de la siguiente manera:

“Es una interfaz que contiene el nombre y argumentos de los métodos posibles para cada entidad”

Para nuestro caso vamos a crear un repositorio genérico y se llama genérico por que va a contener todos los métodos básicos que necesita cada entidad: Listar, Obtener por id, Crear, Editar y Eliminar.

aspDotNetBlankProject\Domain\Ports\IGenericRepository.cs

```
namespace Domain.Ports
{
    0 referencias
    public interface IGenericRepository<T> where T : class
    {
        0 referencias
        public Task<List<T>> Get();
        0 referencias
        public Task<T> Get(int id);
        0 referencias
        public Task<T> Create(T data);
        0 referencias
        public Task<T> Edit(T data);
        0 referencias
        public Task Delete(int id);
    }
}
```

El puerto sirve para ser inyectado

Creación de nuestro repositorio ADAPTADOR

Se observó en la creación del puerto que los objetos comparten 5 métodos nosotros vamos a implementar ese contrato.

Paso 0: crear la carpeta BASE la cual nos sirve para indicar a los compañeros programadores que ahí se encuentra el padre de todos los repositorios.

aspDotNetBlankProject\Infraestructure\Adapters\BASE\

aspDotNetBlankProject\Infraestructure\Adapters\BASE\GenericRepository.cs

```
private readonly PersitenceContext _context;  
private readonly IDbConnection _dapperSource;
```

Vamos a inyectar el contexto y dapper. Luego procederemos a escribir los métodos del puerto para cumplir con el contrato.

Inyección de dependencias

Una inyección se refiere a cuándo servicios y repositorios que se necesitan en múltiples partes los registramos en la main y los dejamos listos para trabajar. Para nuestro caso estas son las inyecciones:

aspDotNetBlankProject\aspDotNetBlankProject\Program.cs

1 -> Obtener el string de conexión a base de datos:

```
var connectionString = builder.Configuration.GetConnectionString("DefaultConnection");
```

Inyectar conector a SQLserver:

```
builder.Services.AddDbContext<PersitenceContext>(options =>  
    options.UseSqlServer(connectionString));
```

Inyectar conector dapper:

```
builder.Services.AddTransient<IDbConnection>(  
    (sp) => new SqlConnection(connectionString)  
);
```

Inyectar mediatr por cada comando:

```
builder.Services.AddMediatr(config => config.RegisterServicesFromAssemblies(typeof(Application.Example.Commands.CommandCreateExampleHandler).Assembly));
```

Es la inyección en la main (Un puerto se asocia a un adaptador)

aspDotNetBlankProject\aspDotNetBlankProject\Program.cs

```
builder.Services.AddScoped<IGenericRepository<Example>, GenericRepository<Example>>();
```

Las inyecciones se hacen antes de construir la APP.

Creación del servicio

El servicio no es más que una capa intermedia entre el controlador y los repositorios.

aspDotNetBlankProject\Domain\Services\ExampleService.cs

```
private readonly IGenericRepository<Example> _repository;  
0 referencias  
public ExampleService(IGenericRepository<Example> repository)  
{  
    _repository = repository;  
}
```

Lo que se hace es inyectar el puerto y llamar a todos los métodos que se define en ese método.

Recuerda inyectar el servicio:

```
builder.Services.AddScoped<ExampleService>();
```

Implementación del CQRS

Cada entidad necesita conectarse a la base de datos para poder leer/escribir información y la forma de hacer eso es crear por cada entidad sus métodos básicos hacia la base de datos:

- Listar todos.
- Obtener por identificador.
- Crear.
- Modificar.
- Eliminar.
- ...

CQRS es un patrón de diseño el cual se encarga de separar las responsabilidades de lectura (Query) y escritura (Command) de la base de datos. Esto significa que cada modelo va a tener una carpeta asociada en la capa de infraestructura. Se implementa para mejorar el mantenimiento de código.

Si usted tiene una mínima experiencia en el desarrollo de software ya descubrió que tenemos que escribir exactamente los mismos métodos para cada entidad. por eso necesitamos un estándar de lectura y escritura.

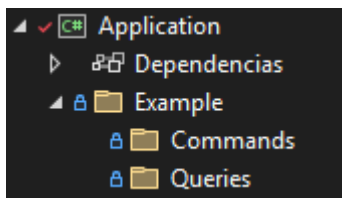
Para cumplir con un estándar nosotros vamos a implementar el patrón CQRS que en realidad es simple:

“Toda acción hacia la base de datos (ENTIDAD) se clasifica en 2 carpetas una carpeta de escritura llamada Commands y una de lectura llamada Queries”

Independientemente de si es Commands/Queries se compone de 2 archivos el record y el handler. El record son los argumentos de entrada y el handler son las llamadas a los servicios.

Paso 0:

Crear las carpetas correspondientes a CQRS:



Paso 1:

Crear el Record Comando para crear un ejemplo:

aspDotNetBlankProject\Application\Example\Commands\CommandCreateExample.cs

```
using MediatR;
using System.ComponentModel.DataAnnotations;

namespace Application.Example.Commands
{
    0 referencias
    public record CommandCreateExample
    (
        [Required] int Id,
        [Required] string Title,
        [Required] string Description,
        [Required] string Information,
        bool? IsDeleted
    ) : IRequest<int>;
}
```

Paso 2:

Crear el handler del comando:

```
using MediatR;
using Domain.Services;

namespace Application.Example.Commands
{
    1 referencia
    public class CommandCreateExampleHandler : IRequestHandler<CommandCreateExample, int>
    {
        private readonly ExampleService _service;
        0 referencias
        public CommandCreateExampleHandler(ExampleService service)
        {
            _service = service;
        }

        0 referencias
        public async Task<int> Handle(CommandCreateExample request, CancellationToken cancellationToken)
        {
            var entity = new Domain.Entities.Example
            {
                Id = request.Id,
                Title = request.Title,
                Description = request.Description,
                Information = request.Information,
                IsDeleted = false,
            };

            Domain.Entities.Example example = await _service.Create(entity);

            return example.Id;
        }
    }
}
```