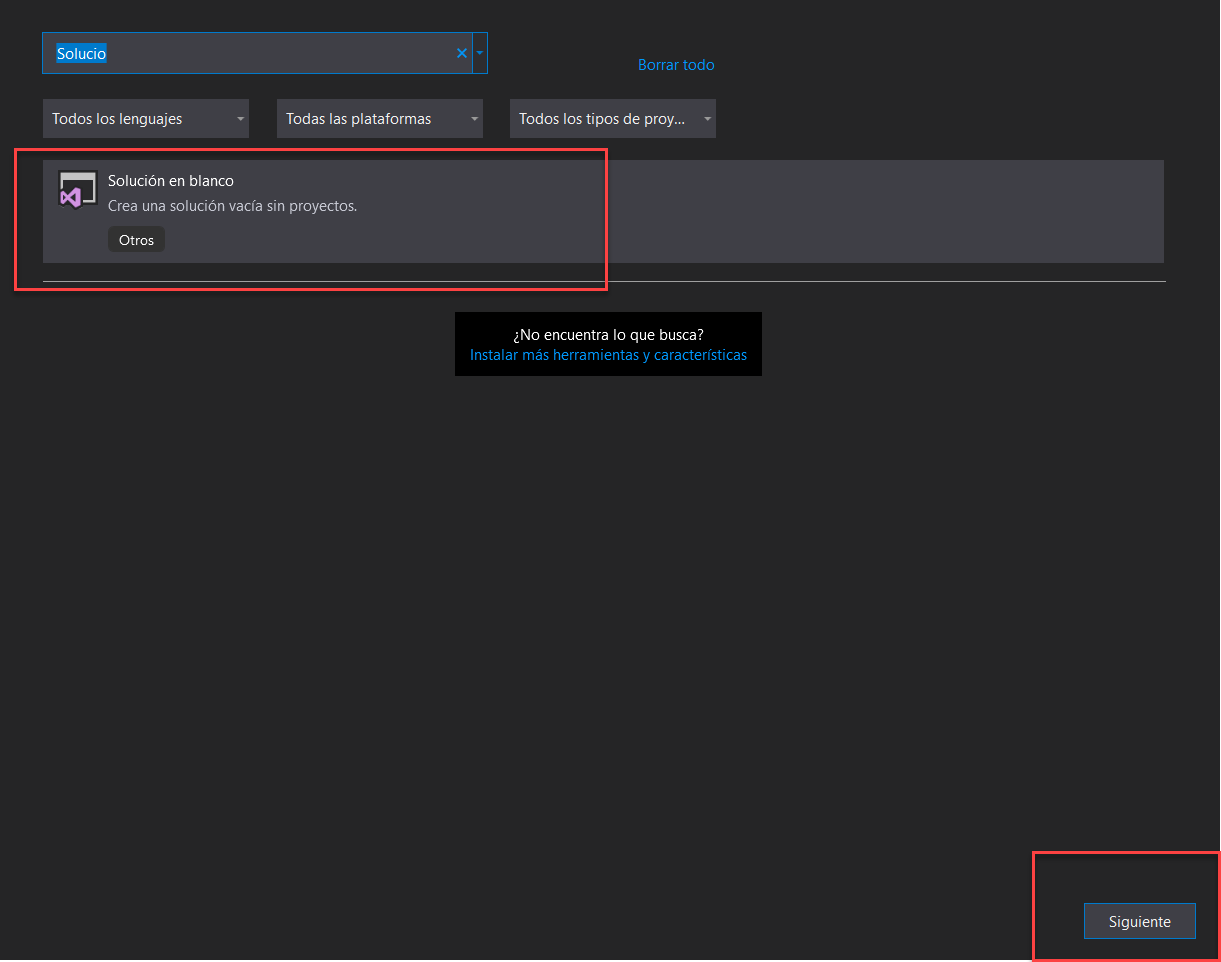
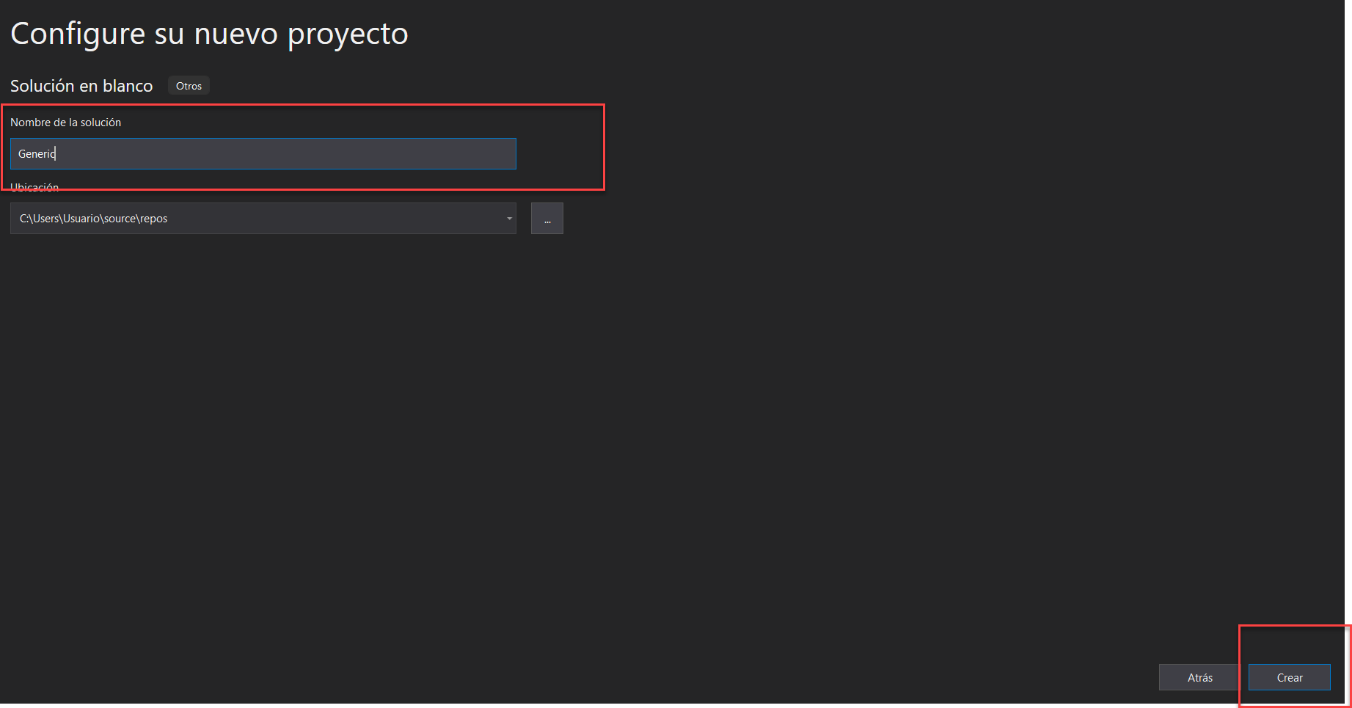
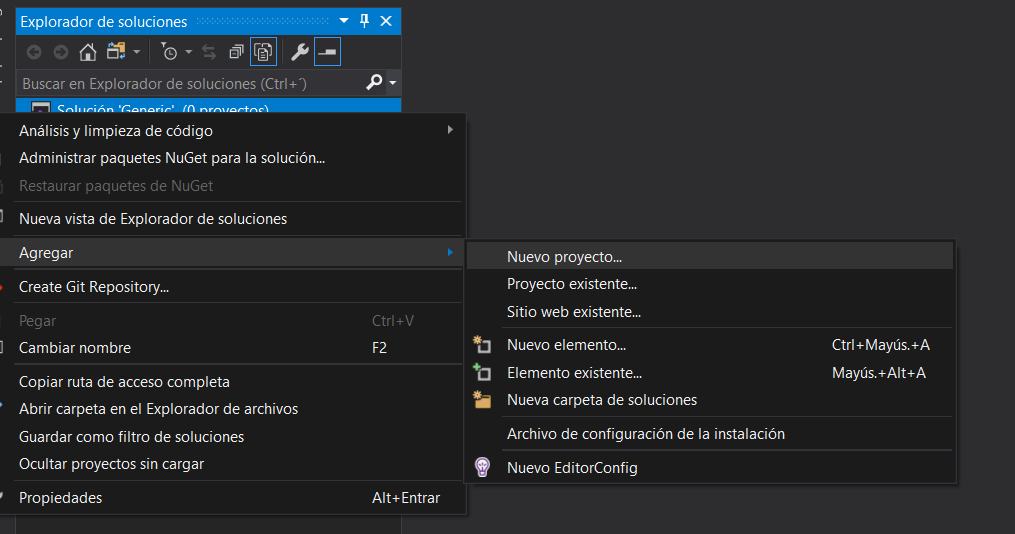
1. Crear una solución en blanco

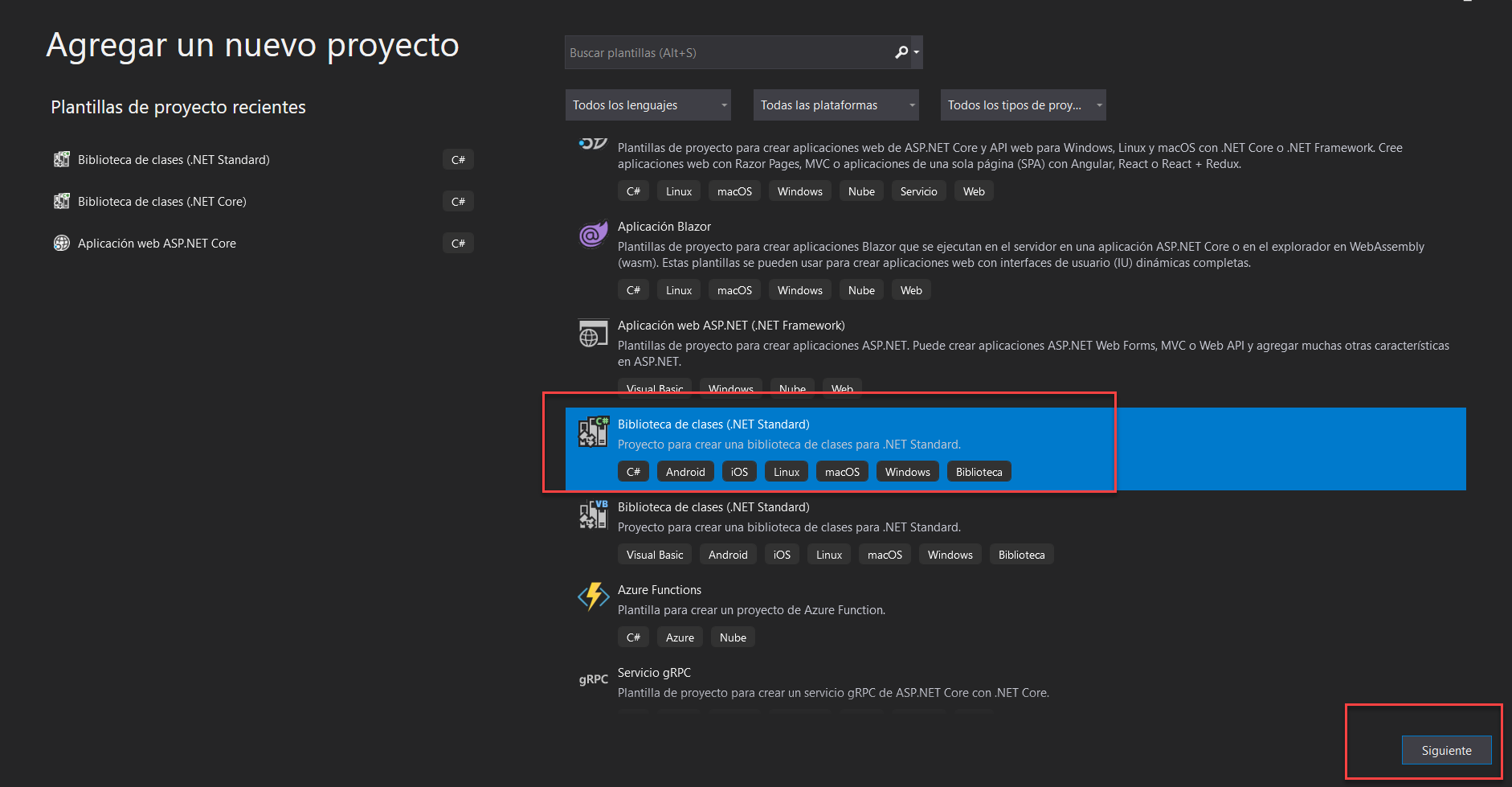


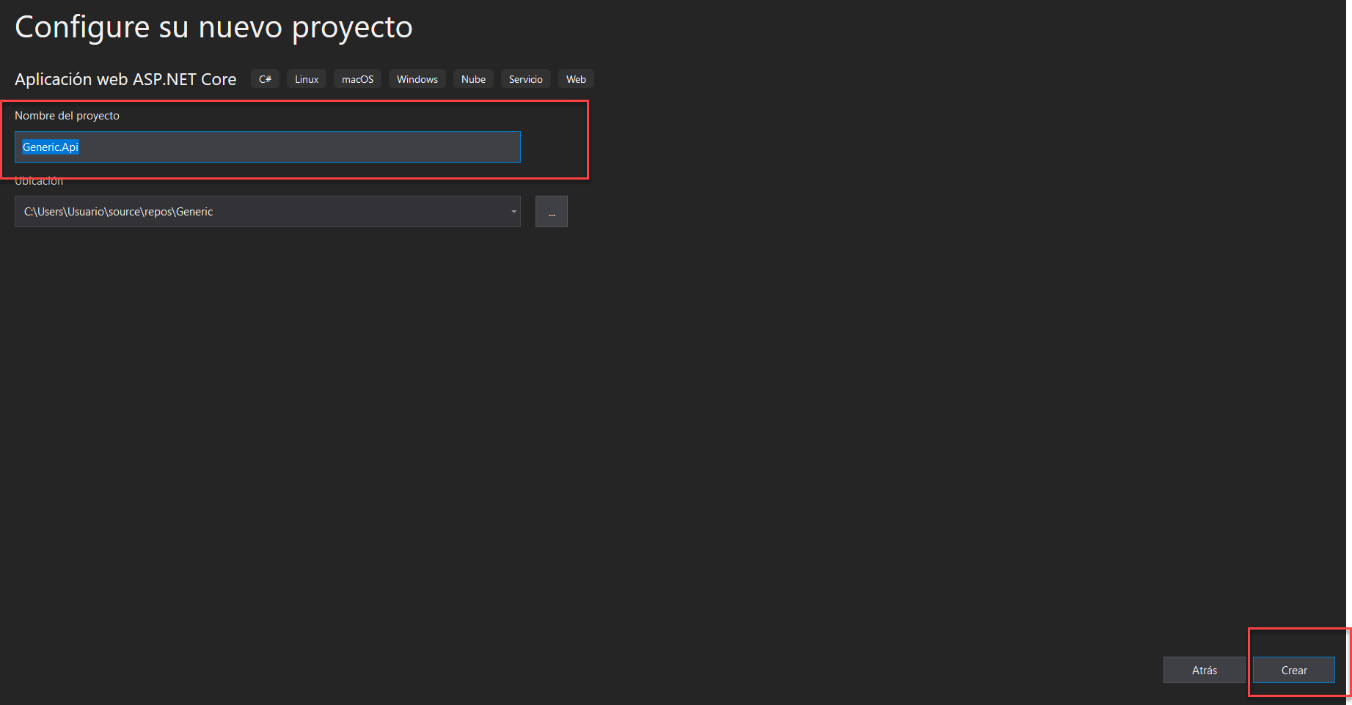


CLI si estás usando visual Code   
dotnet new solution

1. Agregar 3 proyectos de Bibliotecas Net Standard Core, Service y Data, sigue las imágenes



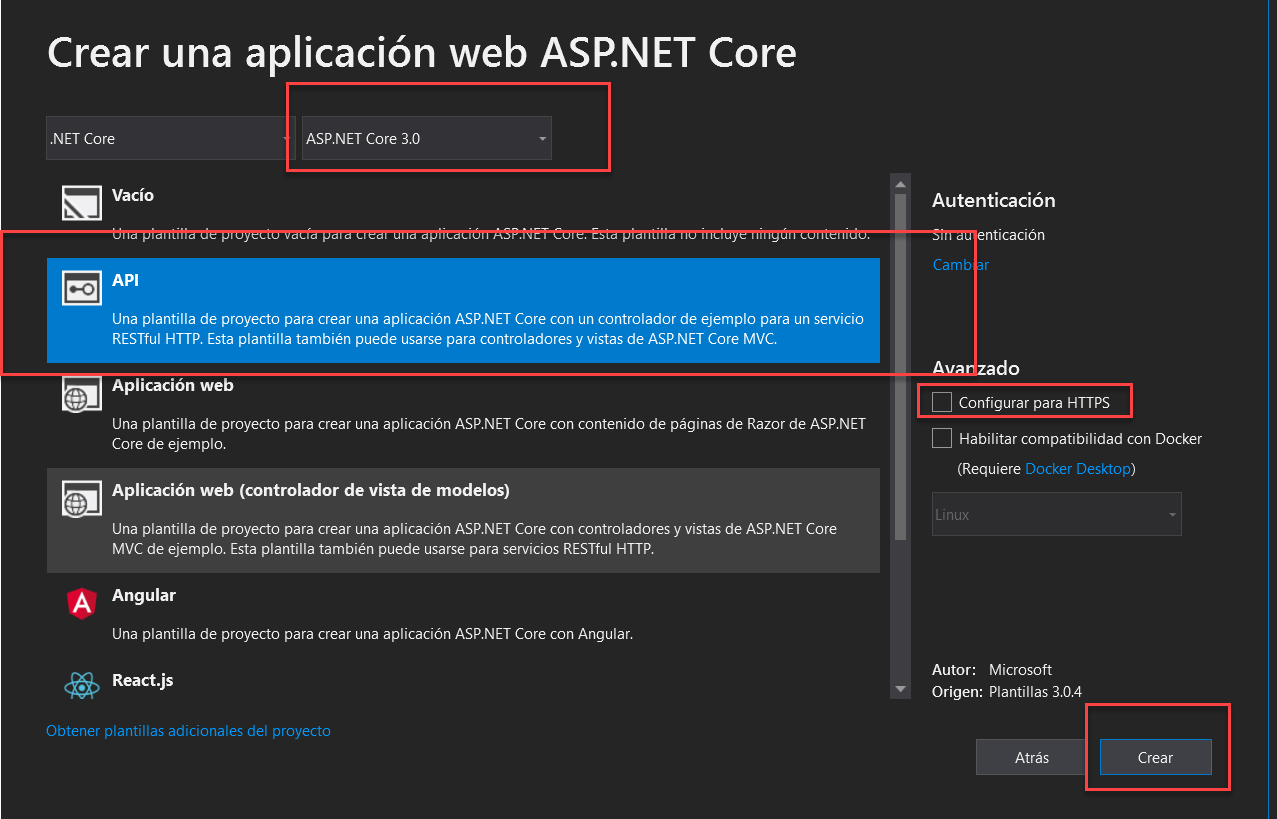


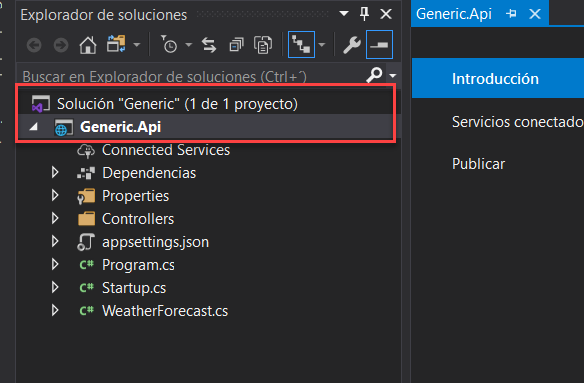


CLI si estás usando visual Code   
dotnet new classlib -o Generic.Core  
dotnet new classlib -o Generic.Services

dotnet new classlib -o MyMusic.Data

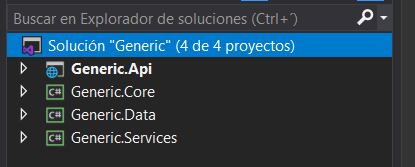
1. Crear un proyecto API en ASP.NET Core 3.0, sin HTTPS



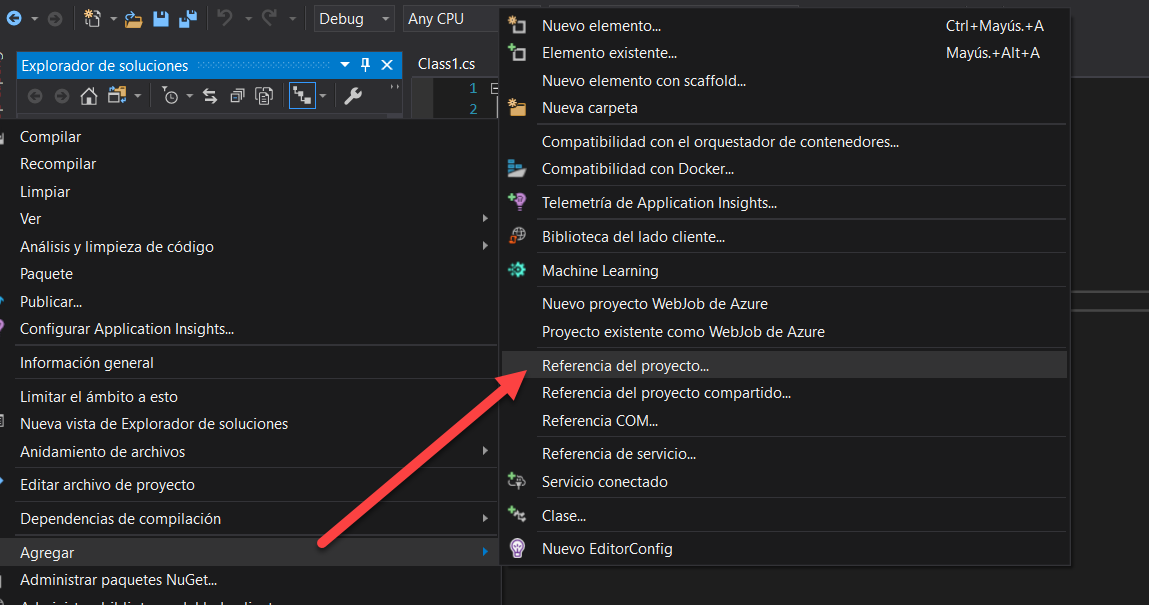


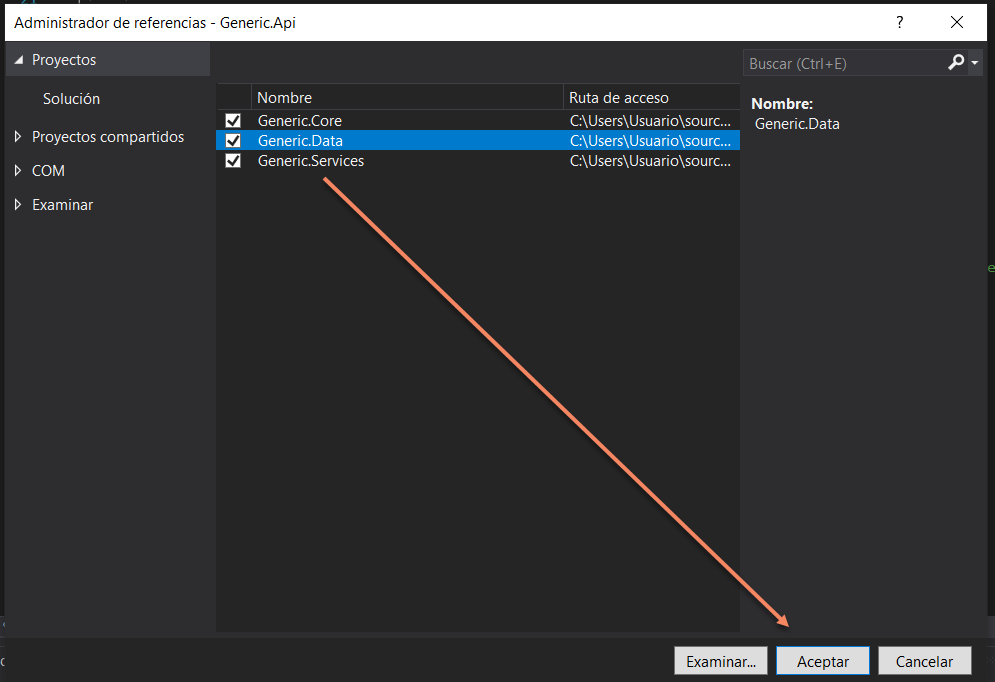
CLI si estás usando visual Code   
dotnet new webapi -o Generic.Api

Tu solución se debería ver así;

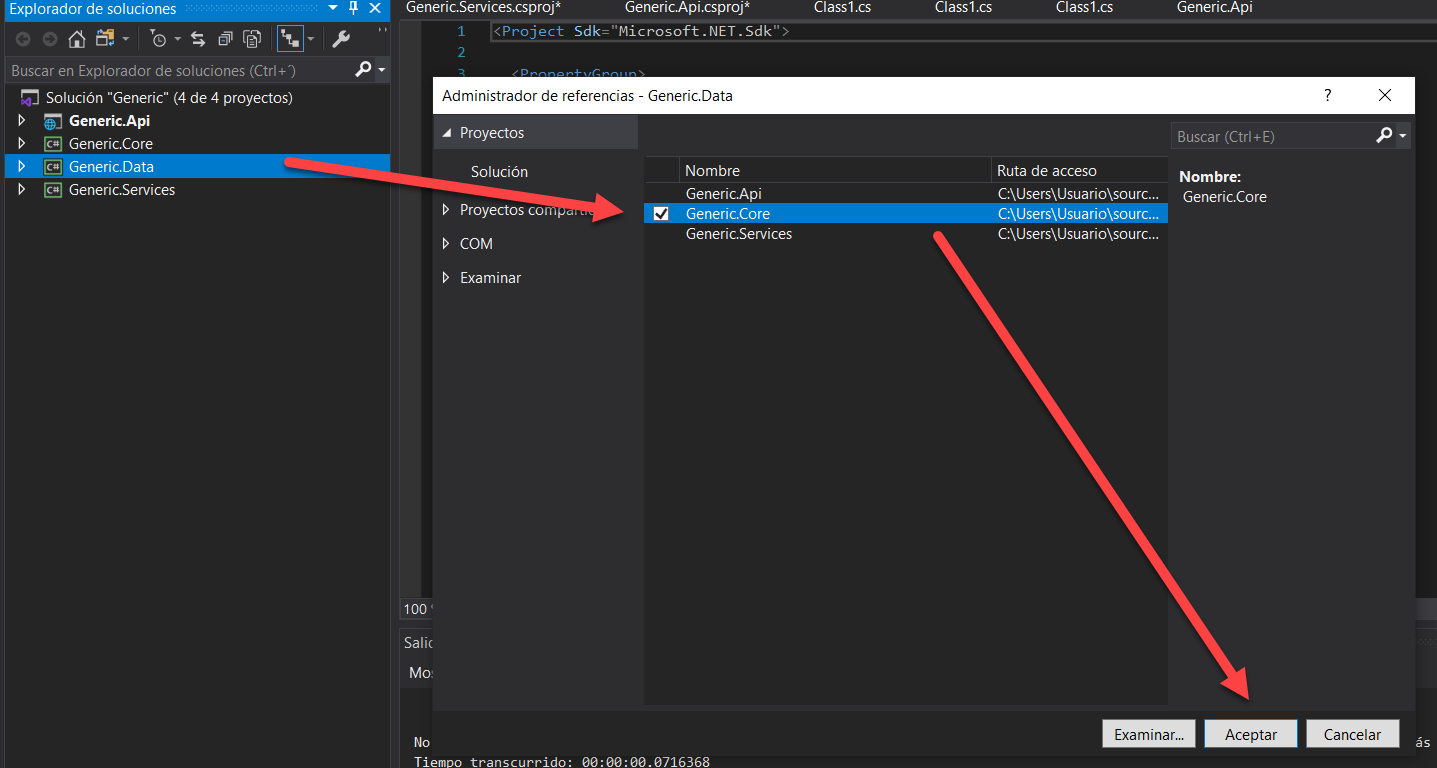


4.- Agrega las referencias entre proyectos como sigue;   
  
Para Generic.Api

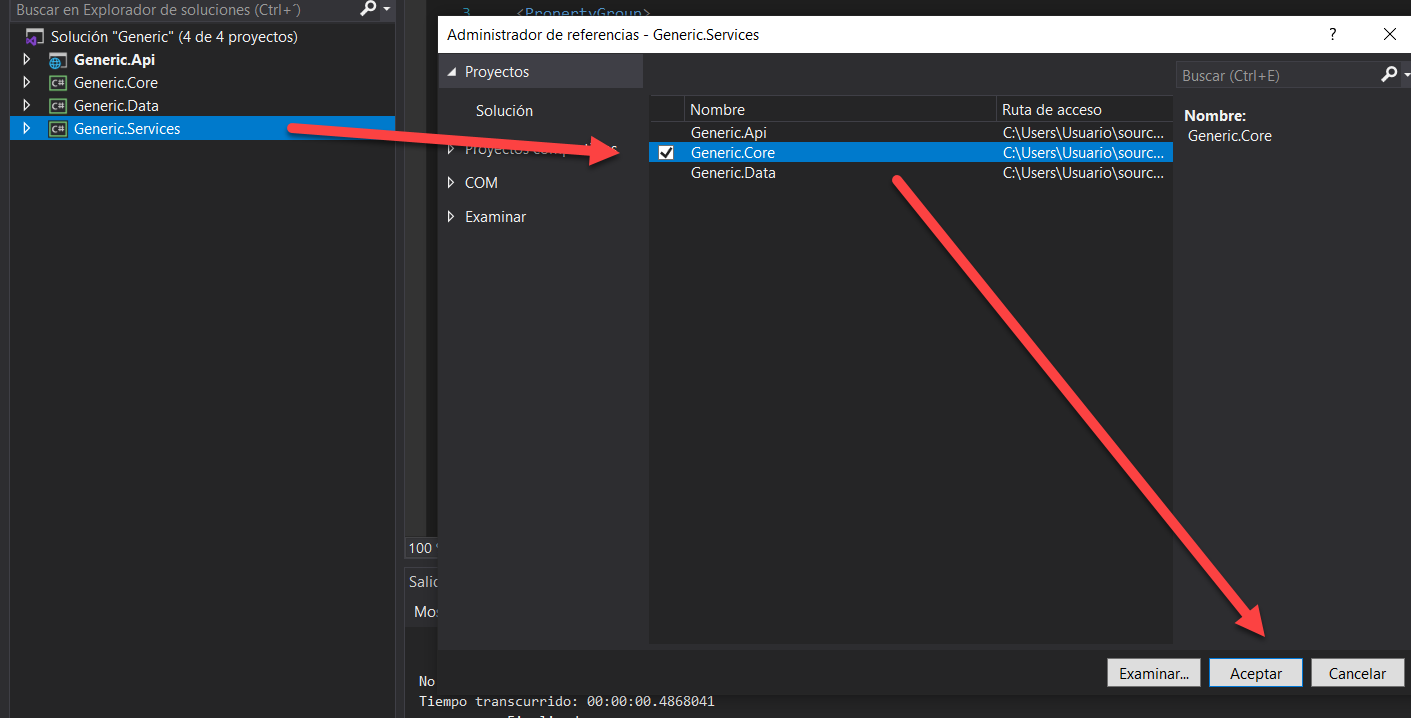




Para Generic.Data



Para Generic.Service

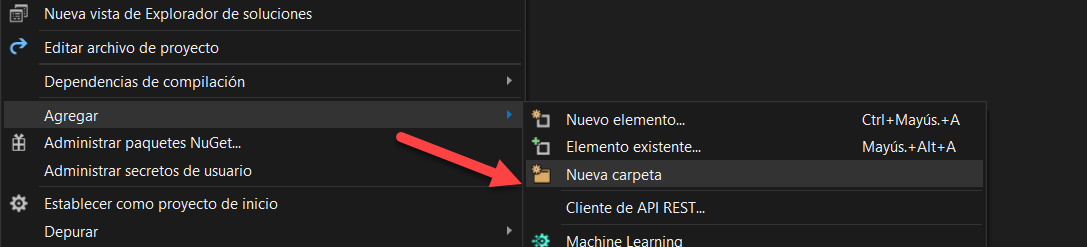


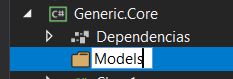
CLI si estás usando visual Code

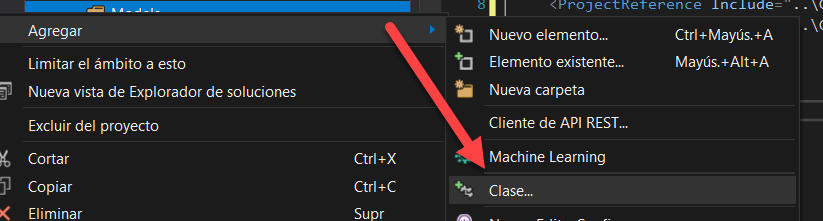
dotnet add Generic.Api/Generic.Api.csproj reference Generic.Core/Generic.Core.csproj Generic.Services/Generic.Services.csprojdotnet add Generic.Data/Generic.Data.csproj reference Generic.Core/Generic.Core.csprojdotnet add Generic.Services/Generic.Services.csproj reference Generic.Core/Generic.Core.csprojdotnet add Generic.Api/Generic.Api.csproj reference Generic.Services/Generic.Services.csproj Generic.Core/Generic.Core.csproj Generic.Data/Generic.Data.csproj

# Core — La capa con la responsabilidad de abstraer

5.- Agrega la carpeta Models en el proyecto Generic.Core para alojar la clase Clients.cs;

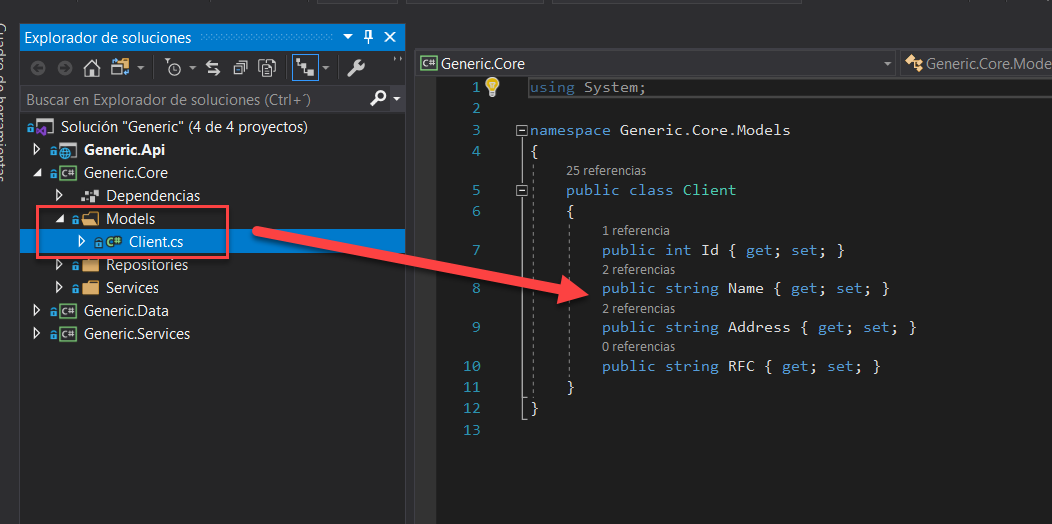




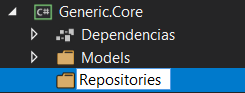


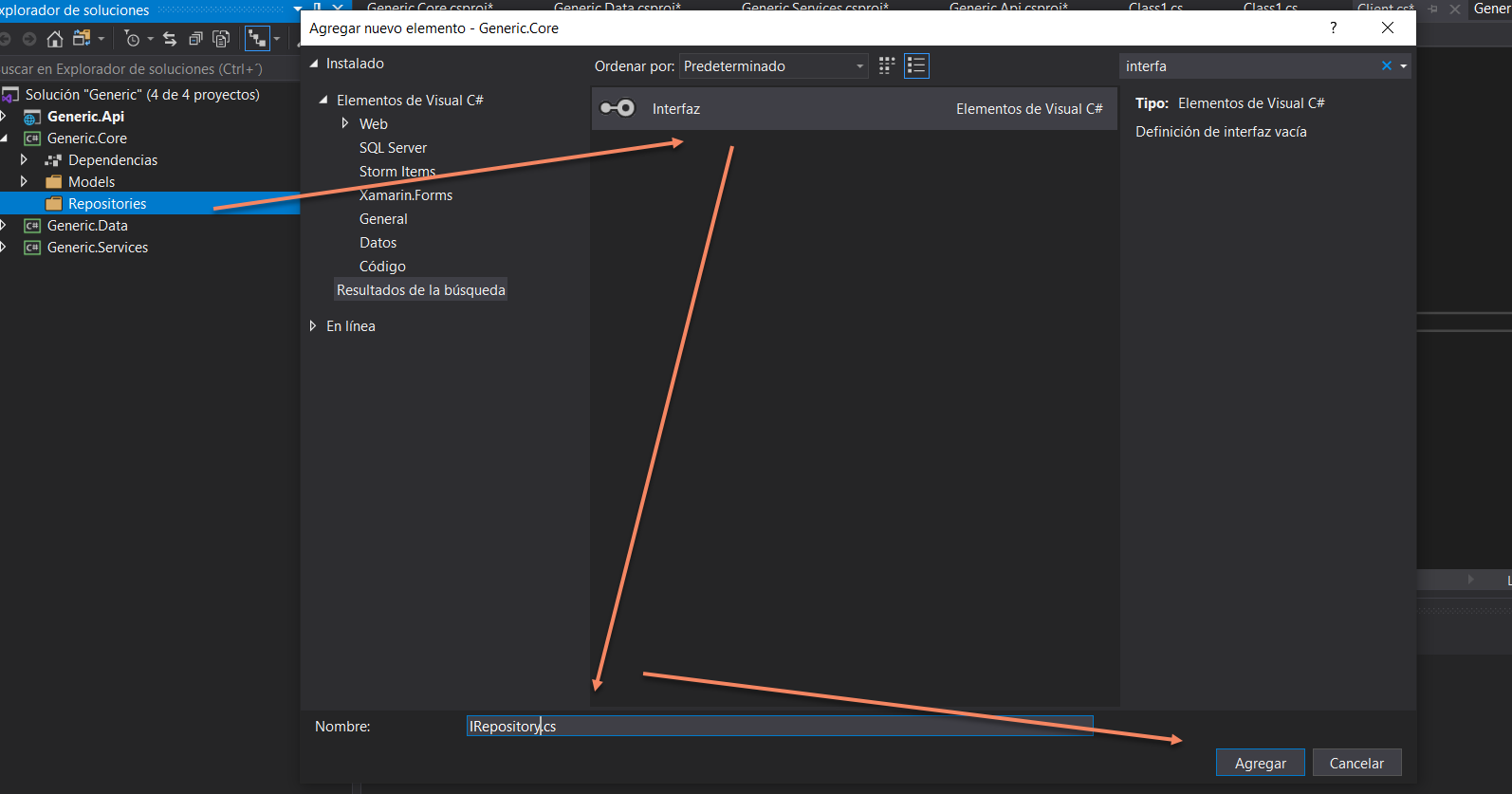
**Cient**

|  |
| --- |
| using System; |
|  |  |
|  | namespace Generic.Core.Models |
|  | { |
|  | public class Client |
|  | { |
|  | public int Id { get; set; } |
|  | public string Name { get; set; } |
|  | public string Address { get; set; } |
|  | public string RFC { get; set; } |
|  | } |
|  | } |

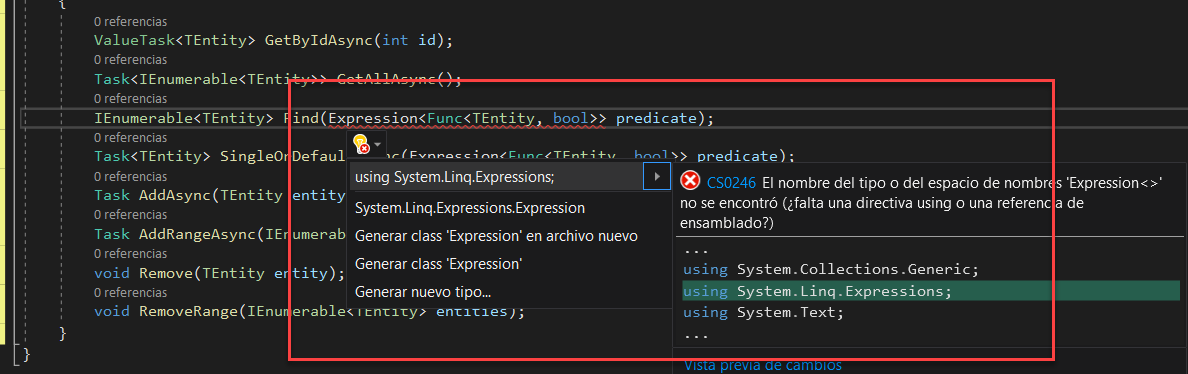


5.- Agrega la carpeta Repositories en el proyecto Generic.Core para alojar la interfaz IRepository.cs y IClientRepository.cs y IUnitOfWork .cs que van a compartir en común las instancias que lo hereden;





NOTA: Si empiezan a tener tema de dependencias pueden ayudarse del corrector de problemas para que les agregue los Using necesarios



**IClientRepository**

using Generic.Core.Models;

using System;

using System.Collections.Generic;

using System.Text;

namespace Generic.Core.Repositories

{

public interface IClientRepository : IRepository<Client>

{

}

}

**IRepository**

using System;

using System.Collections.Generic;

using System.Linq.Expressions;

using System.Text;

using System.Threading.Tasks;

namespace Generic.Core.Repositories

{

public interface IRepository<TEntity> where TEntity : class

{

ValueTask<TEntity> GetByIdAsync(int id);

Task<IEnumerable<TEntity>> GetAllAsync();

IEnumerable<TEntity> Find(Expression<Func<TEntity, bool>> predicate);

Task<TEntity> SingleOrDefaultAsync(Expression<Func<TEntity, bool>> predicate);

Task AddAsync(TEntity entity);

Task AddRangeAsync(IEnumerable<TEntity> entities);

void Remove(TEntity entity);

void RemoveRange(IEnumerable<TEntity> entities);

}

}

**IUnitOfWork**

using System;

using System.Collections.Generic;

using System.Text;

using System.Threading.Tasks;

namespace Generic.Core.Repositories

{

public interface IUnitOfWork : IDisposable

{

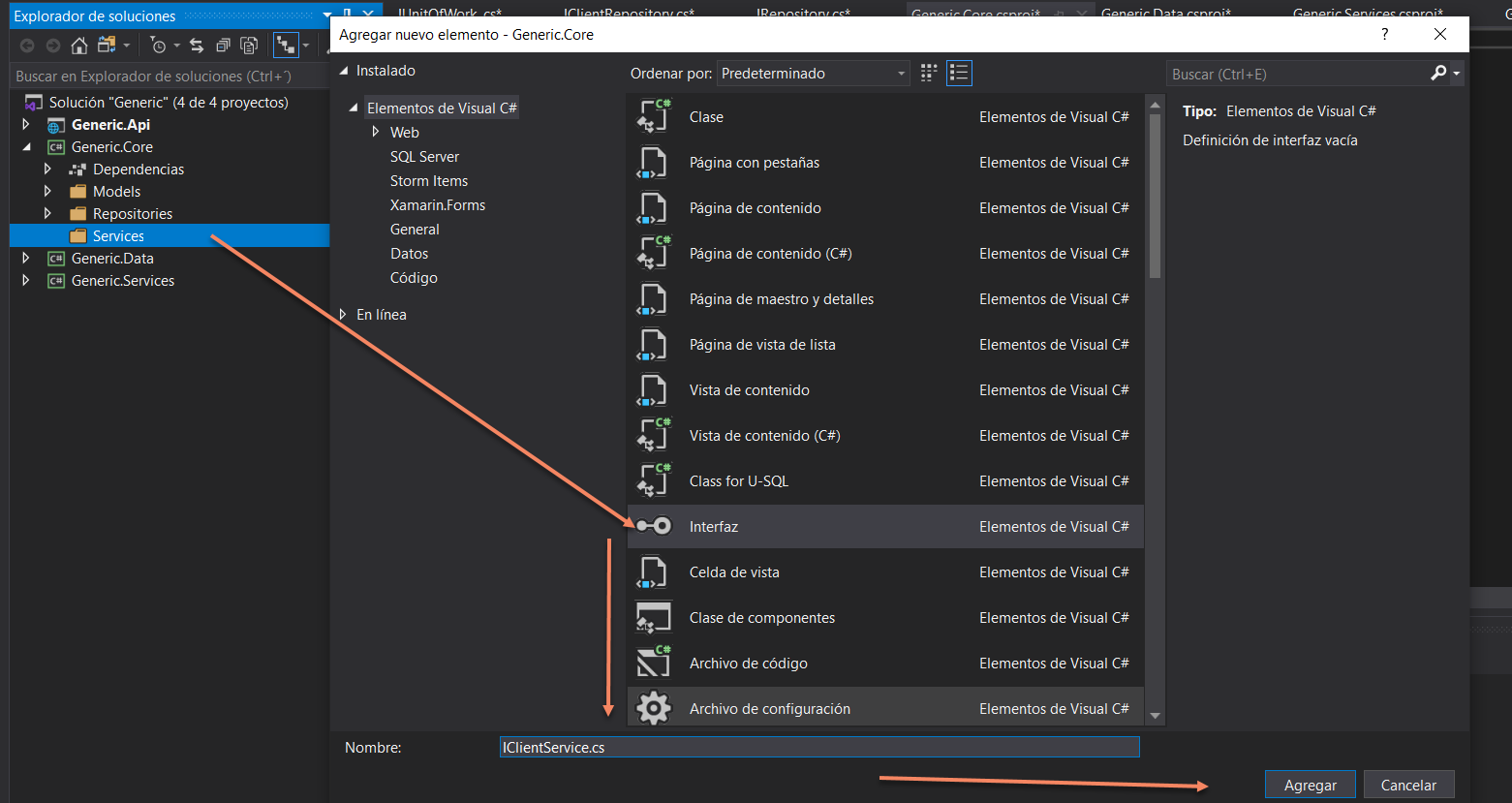
IClientRepository Clients { get; }

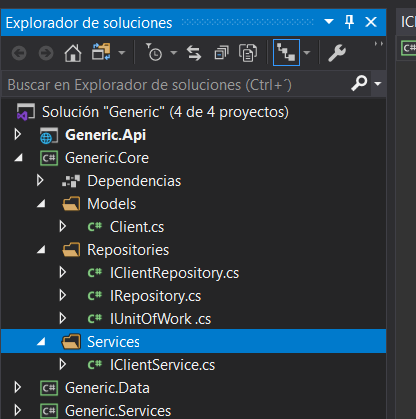
Task<int> CommitAsync();

}

}

6.- Agrega la carpeta Services en el proyecto Generic.Core para alojar la interfaz IClientService.cs





**IClientService**

using Generic.Core.Models;

using System;

using System.Collections.Generic;

using System.Text;

using System.Threading.Tasks;

namespace Generic.Core.Services

{

public interface IClientService

{

Task<IEnumerable<Client>> GetAllClients();

Task<Client> GetClientById(int id);

Task<Client> CreateClient(Client newClient);

Task UpdateClient(Client ClientToBeUpdated, Client Client);

Task DeleteClient(Client Client);

}

}

Data Access — Persistencia

7.- Verificar si tienes instalado el soporte para Entity Framework Core

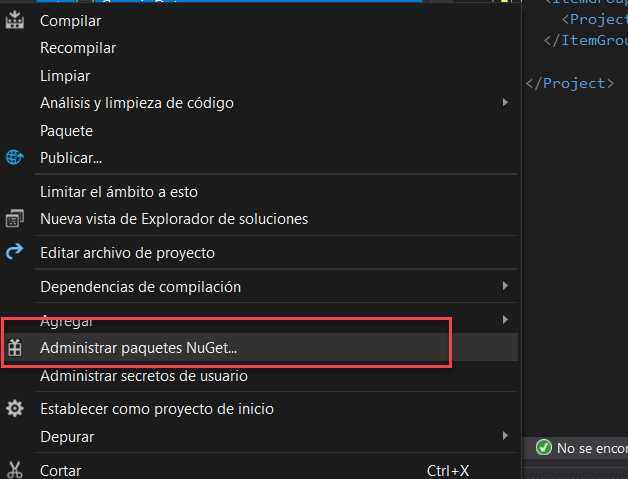
Desde tu consola con el comando;

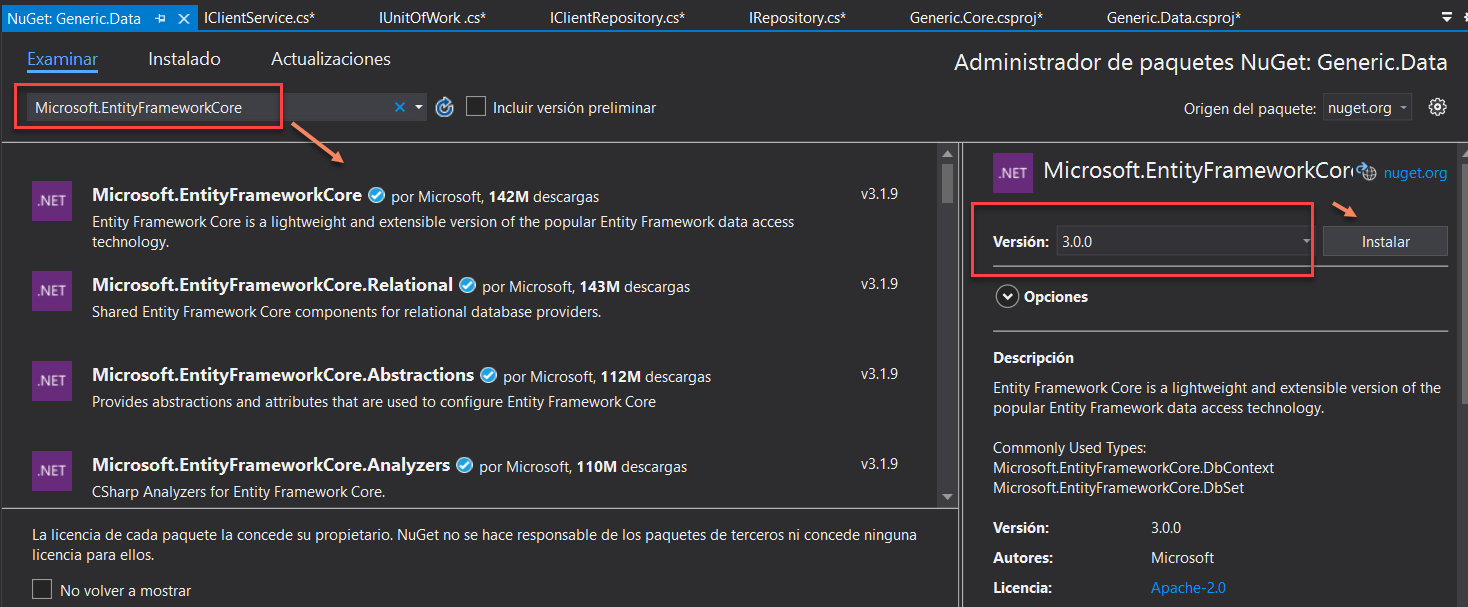
dotnet ef --version

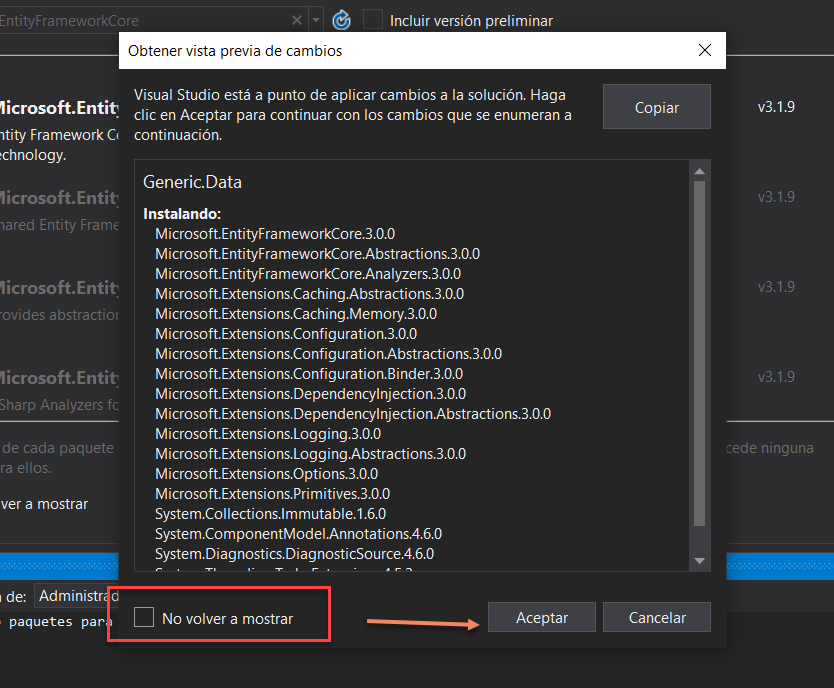
7 a) Si no lo tienes instalado, desde tu consola;

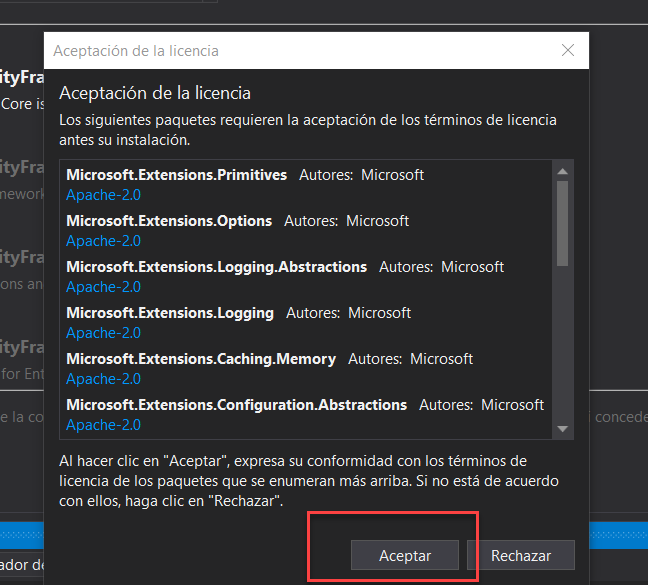
dotnet tool install --global dotnet-ef

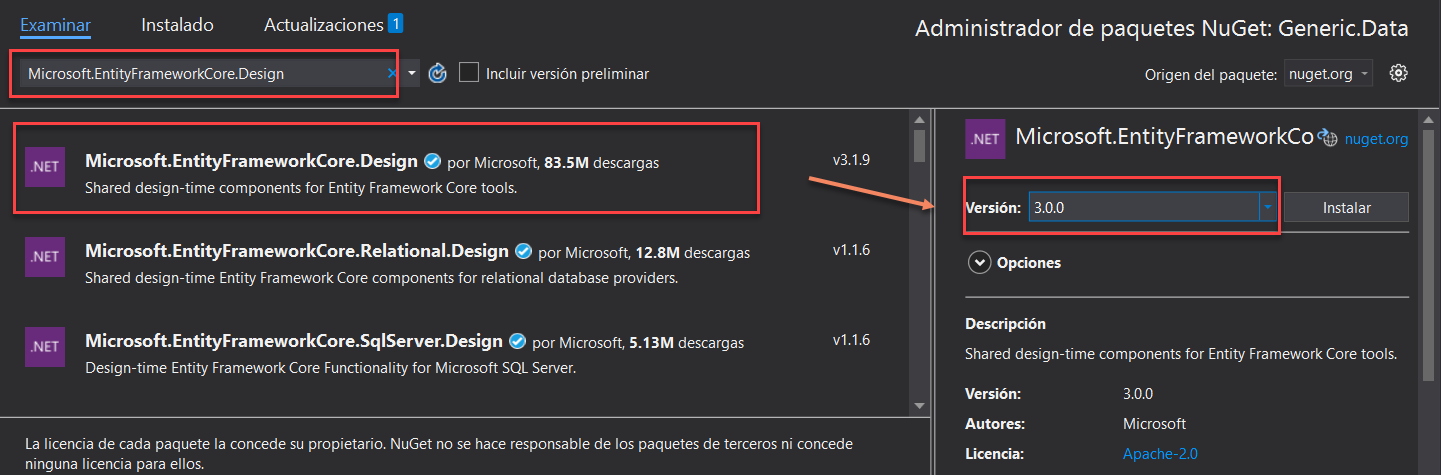
8.- Agrega el paquete Microsot.Entity.Framework.Core 3.0.0 al GenericData

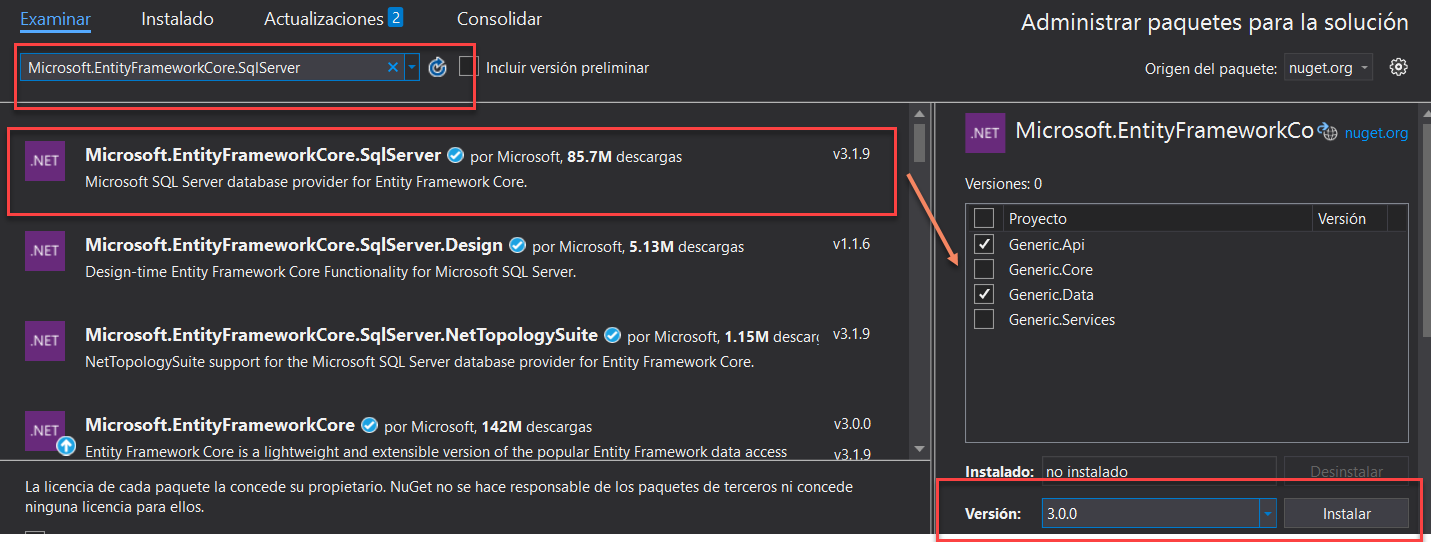












Comando CLI como opcional

dotnet add MyMusic.Data/MyMusic.Data.csproj package Microsoft.EntityFrameworkCoredotnet add MyMusic.Data/MyMusic.Data.csproj package Microsoft.EntityFrameworkCore.Designdotnet add MyMusic.Data/MyMusic.Data.csproj package Microsoft.EntityFrameworkCore.SqlServer

9.- Agrega la clase DataContext quien será la responsable de administrar los DbSet en este caso que vamos por Code First

**DataContext**

using Generic.Core;

using Generic.Core.Models;

using Microsoft.EntityFrameworkCore;

using System;

namespace Generic.Data

{

public class DataContext : DbContext

{

public DbSet<Client> Client { get; set; }

public DataContext(DbContextOptions<DataContext> options) : base(options) { }

}

}

10.- Creamos una carpeta de Repositories ahora en la capa de Data con la finalidad de implementar las interfaces que tenemos expuestas en Core y agregamos la clase Repository y ClientRepository

**Repository**

using Generic.Core.Repositories;

using Microsoft.EntityFrameworkCore;

using System;

using System.Collections.Generic;

using System.Linq;

using System.Linq.Expressions;

using System.Text;

using System.Threading.Tasks;

namespace Generic.Data.Repositories

{

public class Repository<TEntity> : IRepository<TEntity> where TEntity : class

{

protected readonly DbContext Context;

public Repository(DbContext context)

{

this.Context = context;

}

public async Task AddAsync(TEntity entity)

{

await Context.Set<TEntity>().AddAsync(entity);

}

public async Task AddRangeAsync(IEnumerable<TEntity> entities)

{

await Context.Set<TEntity>().AddRangeAsync(entities);

}

public IEnumerable<TEntity> Find(Expression<Func<TEntity, bool>> predicate)

{

return Context.Set<TEntity>().Where(predicate);

}

public async Task<IEnumerable<TEntity>> GetAllAsync()

{

return await Context.Set<TEntity>().ToListAsync();

}

public ValueTask<TEntity> GetByIdAsync(int id)

{

return Context.Set<TEntity>().FindAsync(id);

}

public void Remove(TEntity entity)

{

Context.Set<TEntity>().Remove(entity);

}

public void RemoveRange(IEnumerable<TEntity> entities)

{

Context.Set<TEntity>().RemoveRange(entities);

}

public Task<TEntity> SingleOrDefaultAsync(Expression<Func<TEntity, bool>> predicate)

{

return Context.Set<TEntity>().SingleOrDefaultAsync(predicate);

}

}

}

**ClientRepository**

using Generic.Core;

using System;

using System.Collections.Generic;

using System.Text;

using Generic.Core.Repositories;

using Generic.Core.Models;

namespace Generic.Data.Repositories

{

class ClientRepository : Repository<Client>, IClientRepository

{

public ClientRepository(DataContext context)

: base(context)

{ }

}

}

11.- Creamos una clase UnitOfWork para implementar la Interfaz de la capa Core que nos encapsula los Repositories que vayamos agregando

**UnitOfWork**

using Generic.Core.Repositories;

using Generic.Data.Repositories;

using System;

using System.Collections.Generic;

using System.Text;

using System.Threading.Tasks;

namespace Generic.Data

{

public class UnitOfWork : IUnitOfWork

{

private readonly DataContext \_context;

private ClientRepository \_clientRepository;

public UnitOfWork(DataContext context)

{

this.\_context = context;

}

public IClientRepository Clients => \_clientRepository = \_clientRepository ?? new ClientRepository(\_context);

public async Task<int> CommitAsync()

{

return await \_context.SaveChangesAsync();

}

public void Dispose()

{

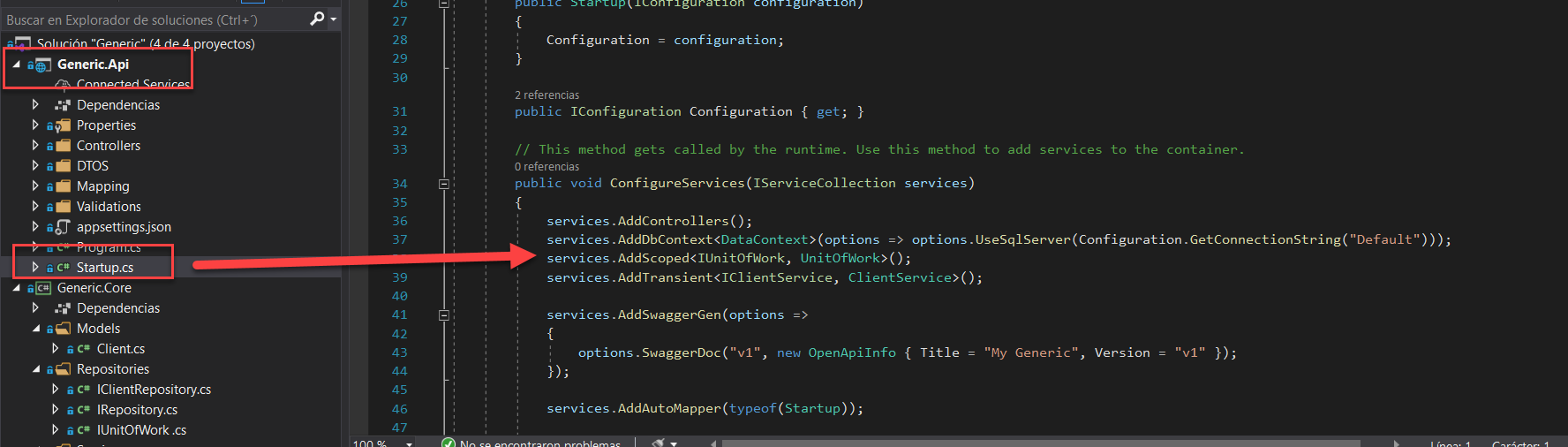
\_context.Dispose();

}

}

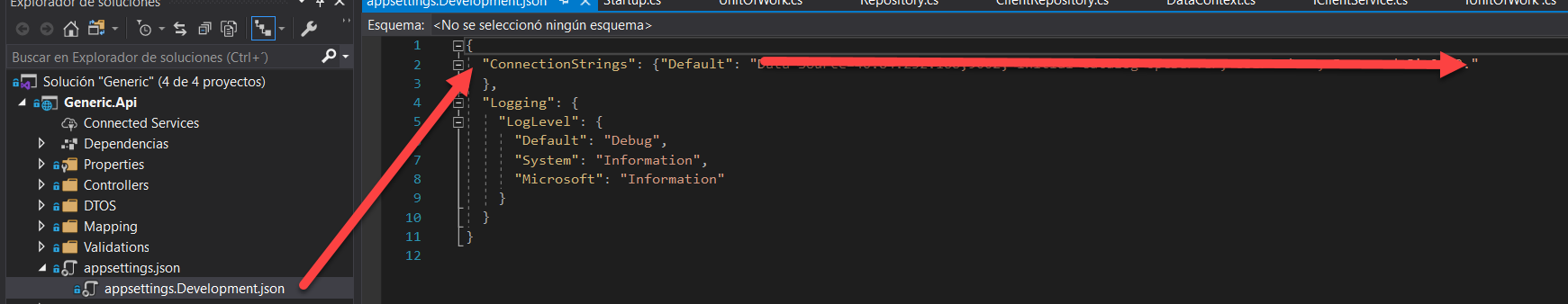
}

11.- En la ConfigureService del StartUp que está contenido en el proyecto Generic.Api agrega la declaración del UnitOfWork y su implementación de modo que queden inyectables a las demás capas



services.AddScoped<IUnitOfWork, UnitOfWork>();

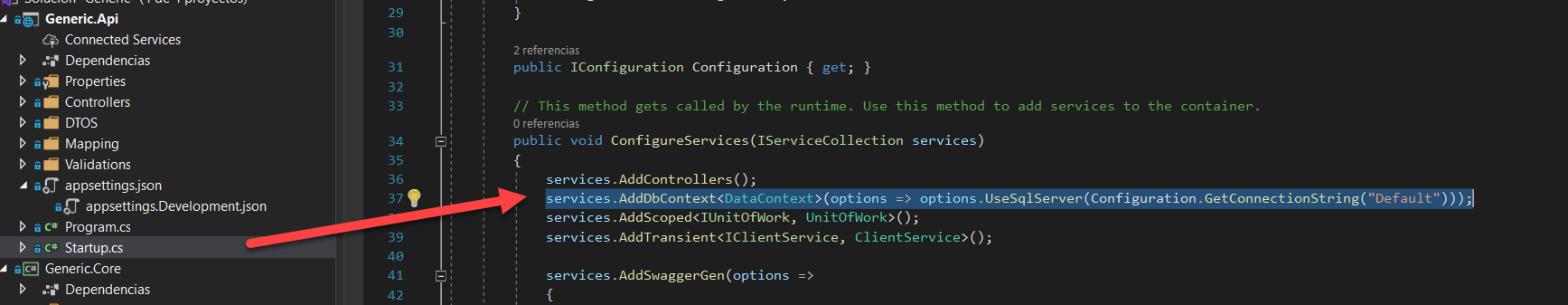
12.- Procedemos a configurar atributos que nos servirán para el EntityFramework, para ello busca el archivo appsettings.Development.json que está en Generic.Api y agrega la cadena de conexión misma que debe tener los atributos correspondientes a tu instancia de base de datos utilizada



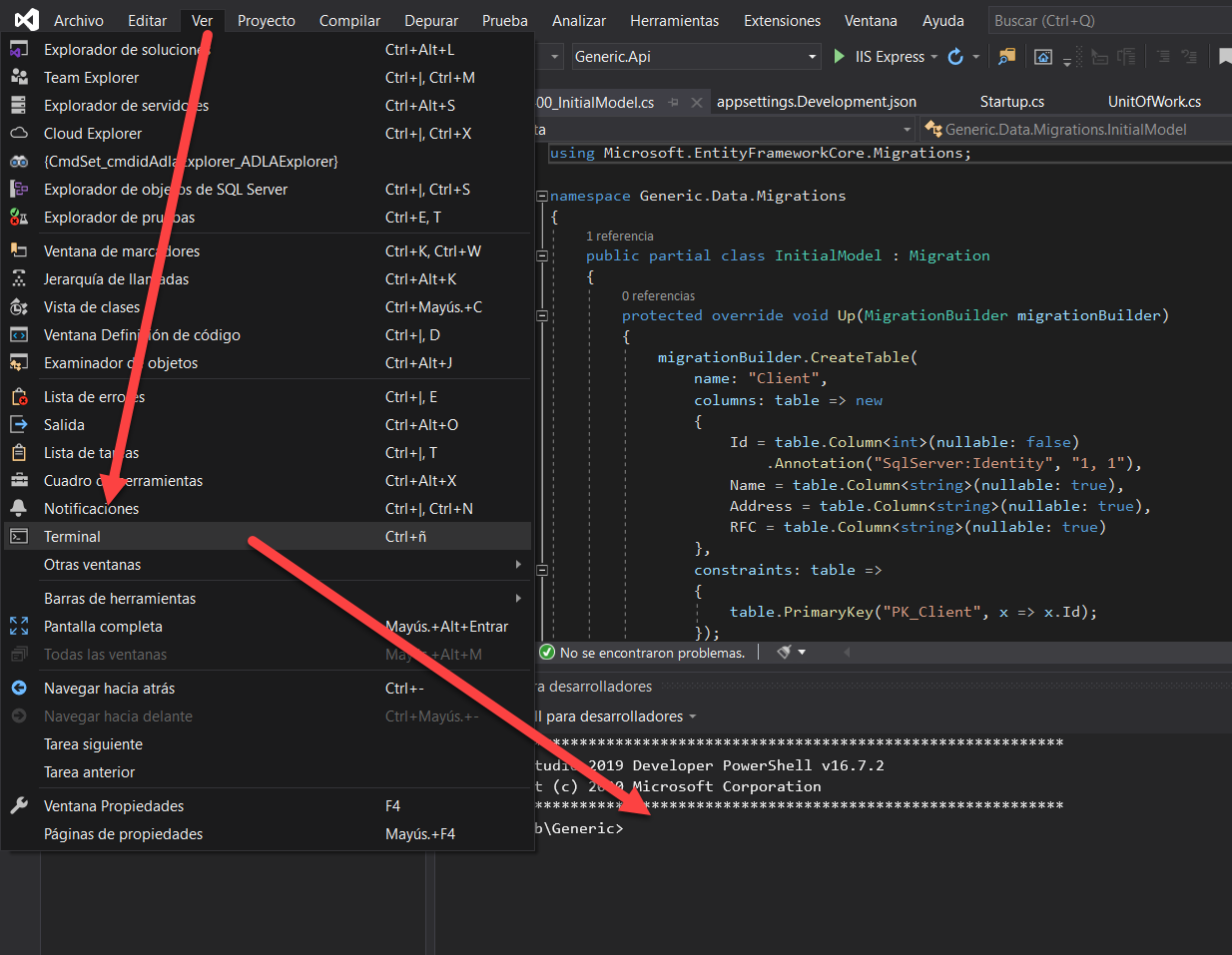
|  |
| --- |
| "ConnectionStrings": { |
|  | "Default": "<Nombre de tu instancia>; database="<Nombre de tu base>; user id="<usuario>; password="<password> |
|  | }, |

13.- En la ConfigureService del StartUp que está contenido en el proyecto Generic.Api agrega la declaración del Contexto de la base de datos

services.AddDbContext<DataContext>(options => options.UseSqlServer(Configuration.GetConnectionString("Default")));



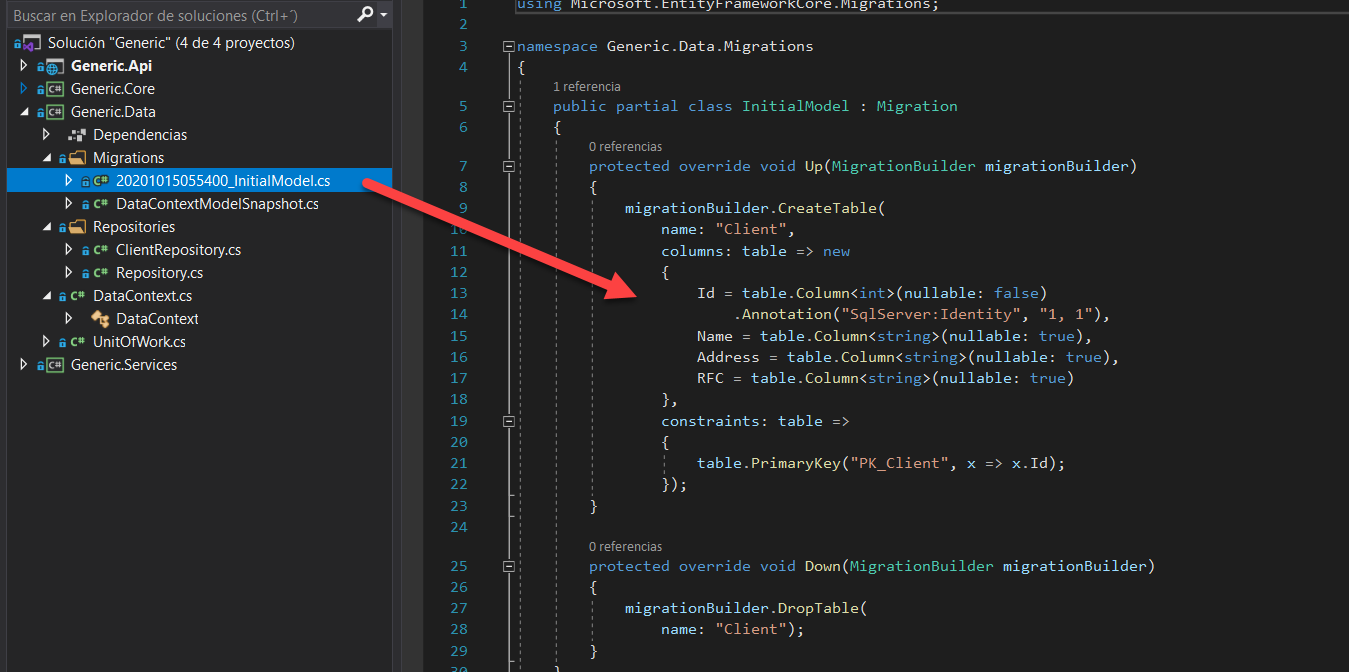
14.- Toca el turno de crear el primer Migration corriendo el siguiendo comando en este caso desde nuestra consola de Visual Studio



dotnet ef --startup-project Generic.Api/Generic.Api.csproj migrations add InitialModel -p Generic.Data/Generic.Data.csproj

NOTA: Si no estás usando Visual puedes correr el mismo comando desde tu consola siempre y cuando estes en el path de tu solución

NOTA 2: lo que se está creando es el código que se va a ejecutar en la instancia de base de datos que especificamos



15.- Impactar en base de datos la estructura del Migration con el siguiente comando;

dotnet ef --startup-project MyMusic.Api/MyMusic.Api.csproj database update

# Services — Capa de Negocio

16.- Agregaremos en el proyecto Generic.Service las clases que van a implementar las Interfaces que se declararon en Generic.Core

**ClientService**

using System.Collections.Generic;

using System.Threading.Tasks;

using Generic.Core.Models;

using Generic.Core.Repositories;

using Generic.Core.Services;

namespace Generic.Services

{

public class ClientService : IClientService

{

private readonly IUnitOfWork \_unitOfWork;

public ClientService(IUnitOfWork unitOfWork)

{

this.\_unitOfWork = unitOfWork;

}

public async Task<Client> CreateClient(Client newClient)

{

await \_unitOfWork.Clients.AddAsync(newClient);

await \_unitOfWork.CommitAsync();

return newClient;

}

public async Task DeleteClient(Client Client)

{

\_unitOfWork.Clients.Remove(Client);

await \_unitOfWork.CommitAsync();

}

public async Task<IEnumerable<Client>> GetAllClients()

{

return await \_unitOfWork.Clients

.GetAllAsync();

}

public async Task<Client> GetClientById(int id)

{

return await \_unitOfWork.Clients

.GetByIdAsync(id);

}

public async Task UpdateClient(Client ClientToBeUpdated, Client Client)

{

ClientToBeUpdated.Name = Client.Name;

ClientToBeUpdated.Address = Client.Address;

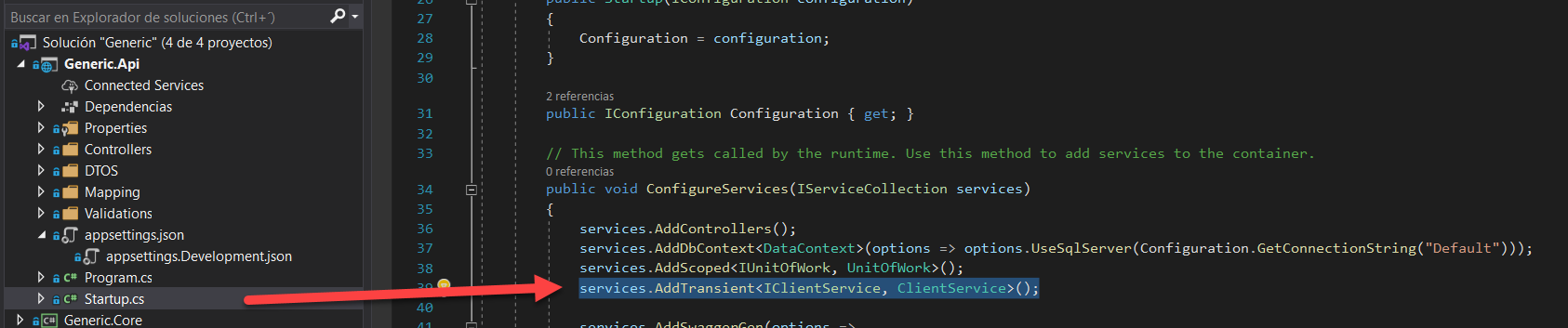
await \_unitOfWork.CommitAsync();

}

}

}

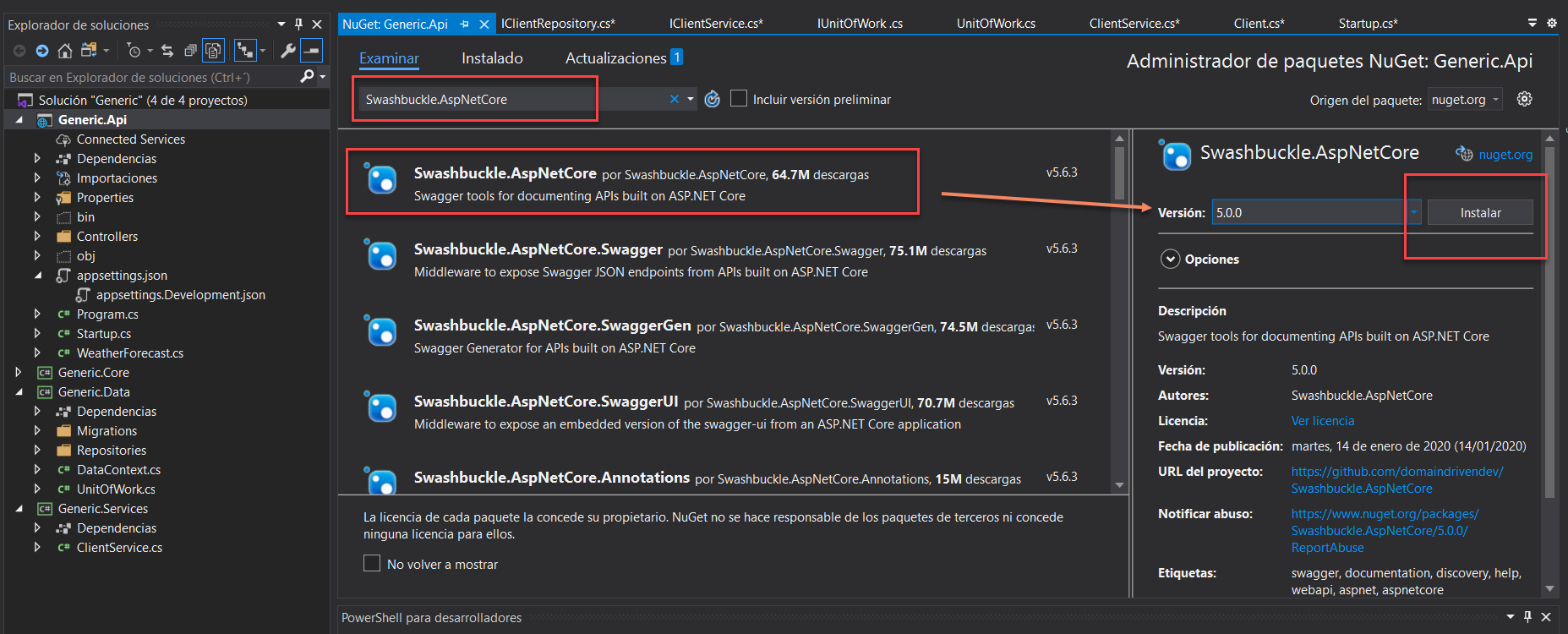
17.- Una vez que agregamos la clase entonces es momento de agregarla al StartUp en el método ConfigureService de Generic.Api para poder entonces inyectarla cuando la necesitemos



services.AddTransient<IClientService, ClientService>();

# API — Presenting our app

18.- Empezaremos por agregar Swagger a nuestro proyecto para eso en el administrador de paquetes del proyecto Generic.Api buscamos SwashBuckle e instalamos la versión como se muestra en la imagen;



Si estás en modo Consola el comando equivalente en CLI es

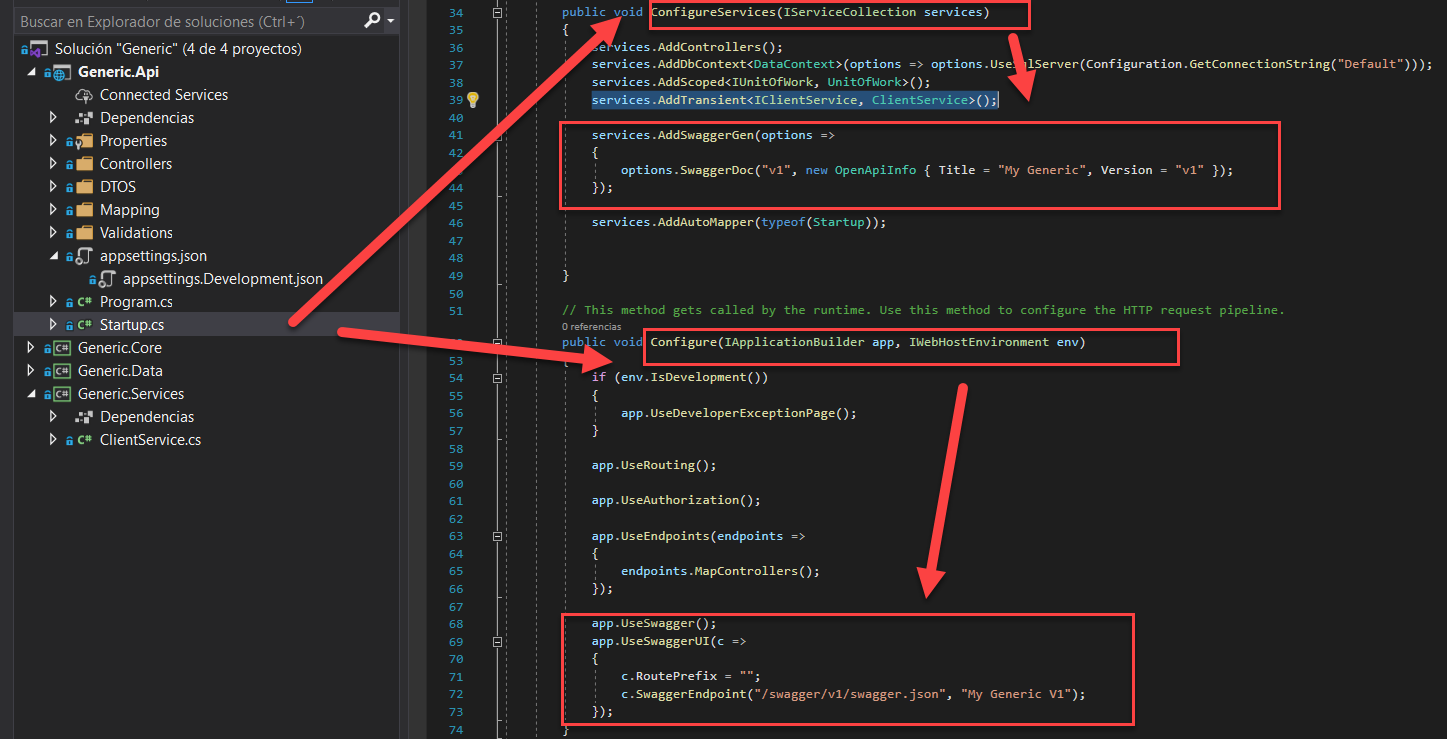
dotnet add Generic.Api/ Generic.Api.csproj package Swashbuckle.AspNetCore --version 5.0.0-rc3

19.- Paso seguido vamos a configurar en el archivo Startup del siguiente modo;   
En el método de Configureservice agrega las líneas

services.AddSwaggerGen(options =>  
{  
options.SwaggerDoc("v1", new OpenApiInfo { Title = "My Generic", Version = "v1" });  
});

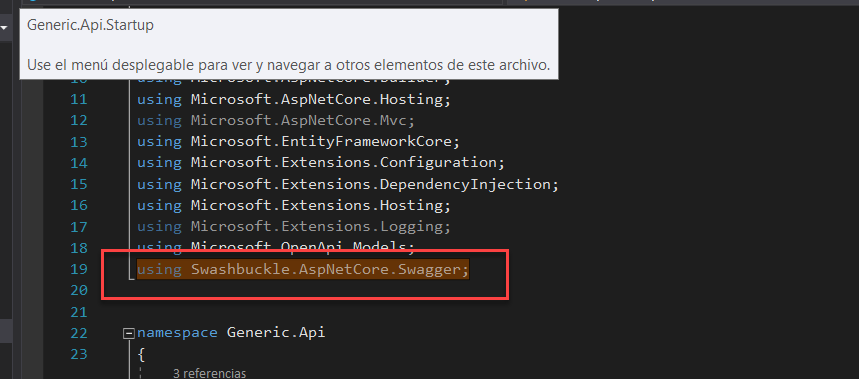
En seguida en Configure

app.UseSwagger();app.UseSwaggerUI(c =>  
{  
 c.RoutePrefix = "";  
 c.SwaggerEndpoint("/swagger/v1/swagger.json", "My Generic V1");  
});



Finalmente agrega este using de modo que te quede así;

using Swashbuckle.AspNetCore.Swagger;



20.- Para hacer utilización de DTOS de entrada y salida ahora vamos a crear una carpeta llamada DTOS dentro del mismo proyecto Generic.Api aquí mismo de pronto vamos agregar un par de clases ClientCreate y ClientReturn, que serán el punto de entrada y salida en nuestro Controller

**ClientCreate**

using System;

using System.Collections.Generic;

using System.Linq;

using System.Threading.Tasks;

namespace Generic.Api.DTOS

{

public class ClientCreate

{

public string Name { get; set; }

public string Address { get; set; }

public string RFC { get; set; }

}

}

**ClientReturn**

using System;

using System.Collections.Generic;

using System.Linq;

using System.Threading.Tasks;

namespace Generic.Api.DTOS

{

public class ClientReturn

{

public int Id { get; set; }

public string Name { get; set; }

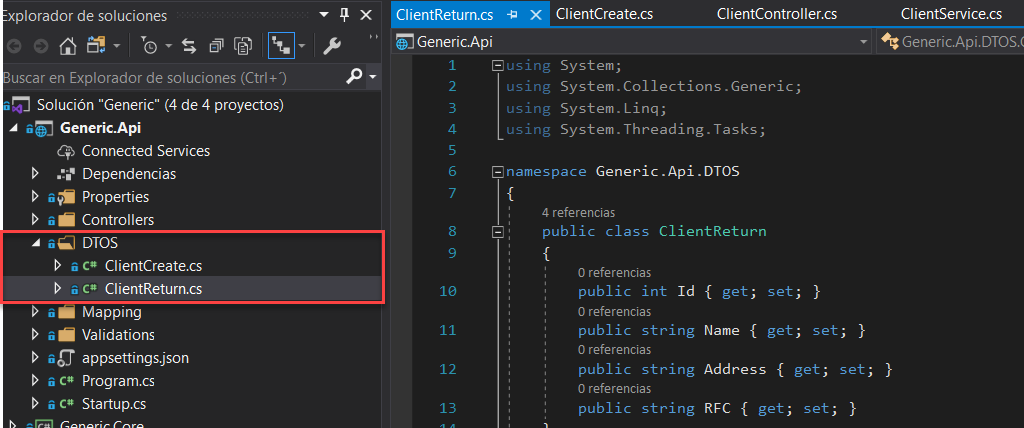
public string Address { get; set; }

public string RFC { get; set; }

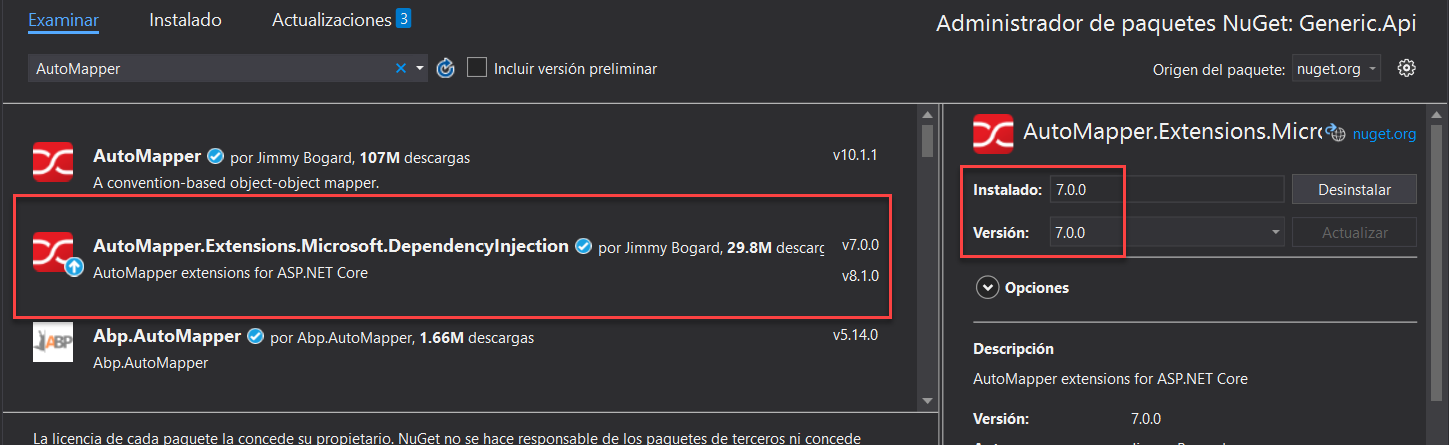
}

}

21.- Ahora necesitamos convertir de DTO a nuestra clase Client contenida en la carpeta de Model



23.- Para hacer la conversión mencionada en el punto anterior haremos uso del paquete AutoMapper.Extensions.Microsoft.DependecyInjection al proyecto de Generic.Api



Si estás en modo Consola el comando equivalente en CLI es

dotnet add Generic.Api/ Generic.Api.csproj package AutoMapper.Extensions.Microsoft.DependencyInjection –version 7.0.0

22.- Crearemos el perfil de mappeo donde vamos a ir agregando las conversiones que vayamos necesitando conforme crezca la necesidad de implementar, para ello agregamos una Capeta que llamaremos Mapping y dentro una clase MappingProfile

**MappingProfile**

using AutoMapper;

using Generic.Api.DTOS;

using Generic.Core.Models;

namespace MyMusic.Api.Mapping

{

public class MappingProfile : Profile

{

public MappingProfile()

{

// Domain to Resource

CreateMap<Client, ClientReturn>();

// Resource to Domain

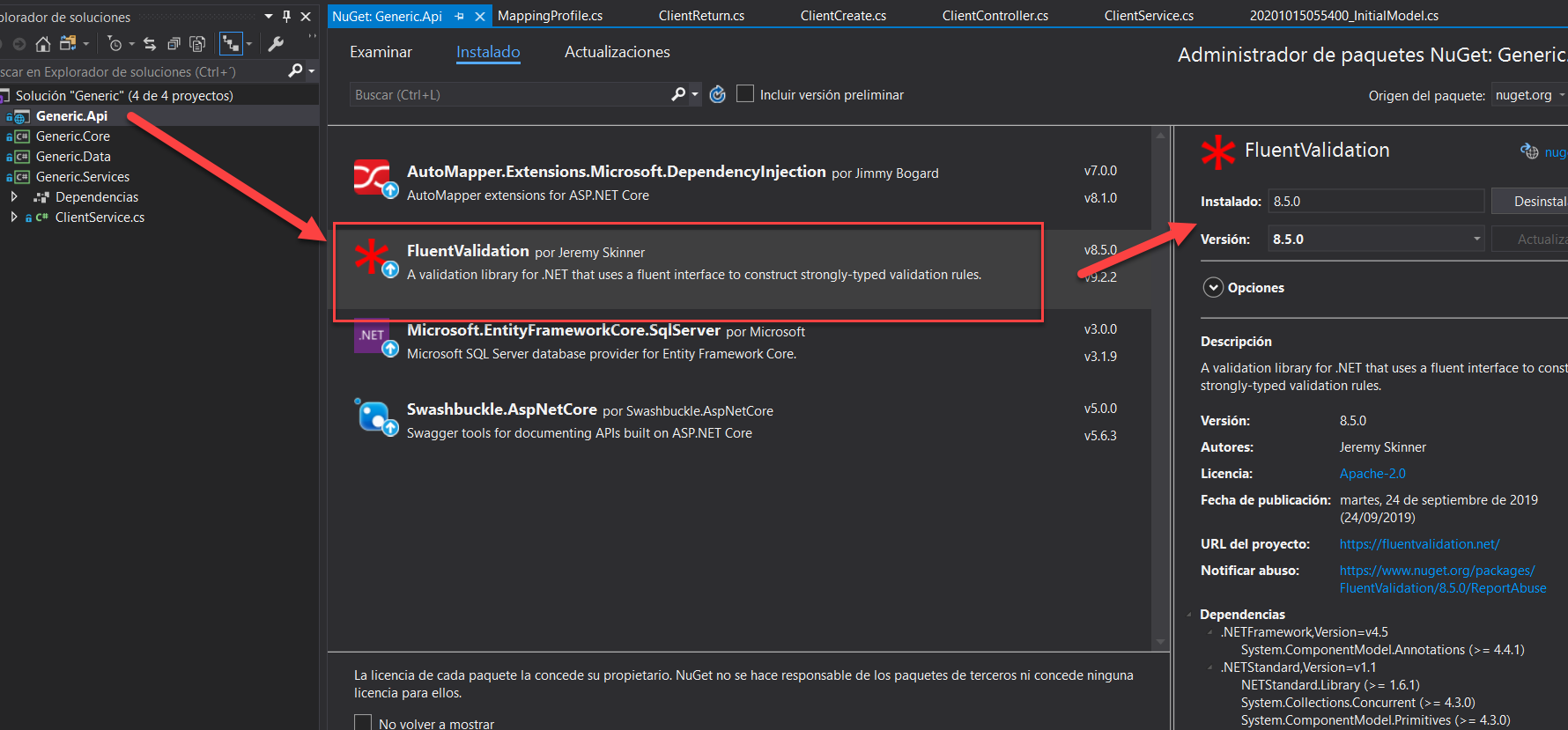
CreateMap<ClientCreate, Client>();

}

}

}

23.- Las validaciones a nuestros DTOS van a ir por un paquete más llamado FluentValidation, para lo cual agregaremos un nuevo paquete a Generic.Api

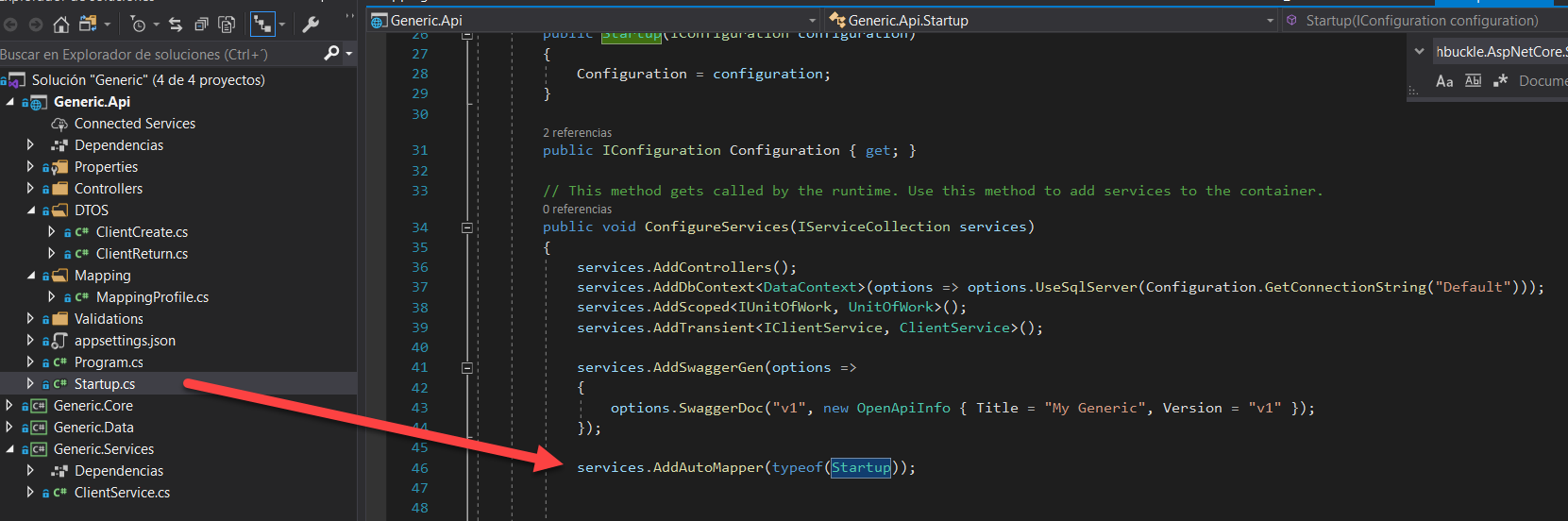


Si estás en modo Consola el comando equivalente en CLI es

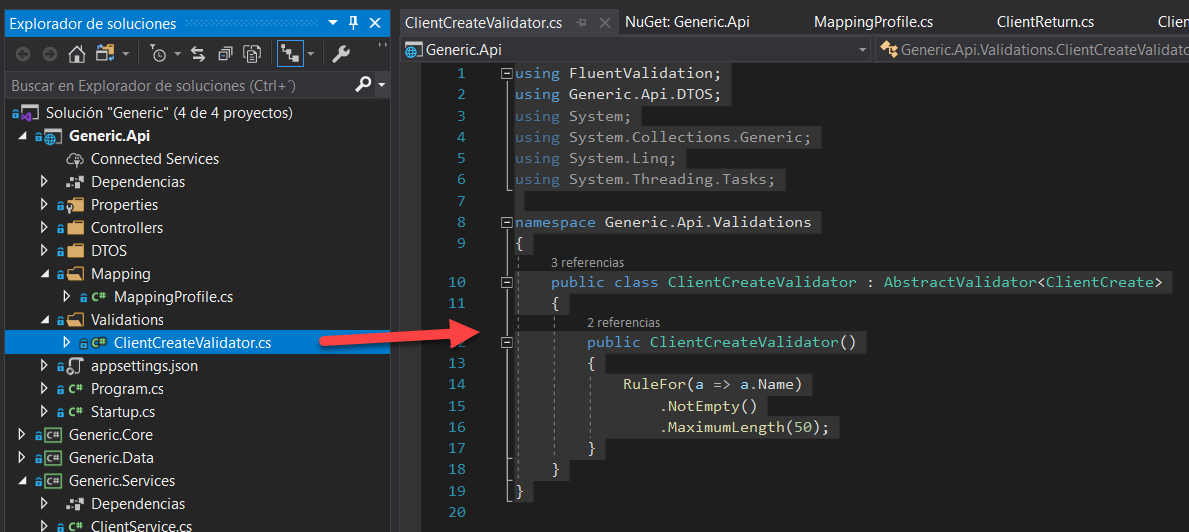
dotnet add MyMusic.Api/Generic.Api.csproj package FluentValidation –version 8.5.0

24.- Configuramos el FluentValidator a nuestro StartUp en el método ConfigureService

services.AddAutoMapper(typeof(Startup));



25.- Creamos para configurar las validaciones una nueva Carpeta en Generic.Api llamada Validation y dentro una nueva clase ClientCreateValidator donde indicamos cuales son las reglas que necesitamos agregar a nuestro DTO



**MappingProfile**

using FluentValidation;

using Generic.Api.DTOS;

using System;

using System.Collections.Generic;

using System.Linq;

using System.Threading.Tasks;

namespace Generic.Api.Validations

{

public class ClientCreateValidator : AbstractValidator<ClientCreate>

{

public ClientCreateValidator()

{

RuleFor(a => a.Name)

.NotEmpty()

.MaximumLength(50);

}

}

}

26.- Vamos ahora a agregar la carpeta de Controller en Generic.Api para tener nuestos puntos de entrada para las operaciones que deseamos hacer y dentro de esa carpeta una clase que llamaremos ClientsController

**ClientController**

using System.Collections.Generic;

using System.Threading.Tasks;

using Generic.Core.Models;

using Generic.Core.Repositories;

using Generic.Core.Services;

namespace Generic.Services

{

public class ClientService : IClientService

{

private readonly IUnitOfWork \_unitOfWork;

public ClientService(IUnitOfWork unitOfWork)

{

this.\_unitOfWork = unitOfWork;

}

public async Task<Client> CreateClient(Client newClient)

{

await \_unitOfWork.Clients.AddAsync(newClient);

await \_unitOfWork.CommitAsync();

return newClient;

}

public async Task DeleteClient(Client Client)

{

\_unitOfWork.Clients.Remove(Client);

await \_unitOfWork.CommitAsync();

}

public async Task<IEnumerable<Client>> GetAllClients()

{

return await \_unitOfWork.Clients

.GetAllAsync();

}

public async Task<Client> GetClientById(int id)

{

return await \_unitOfWork.Clients

.GetByIdAsync(id);

}

public async Task UpdateClient(Client ClientToBeUpdated, Client Client)

{

ClientToBeUpdated.Name = Client.Name;

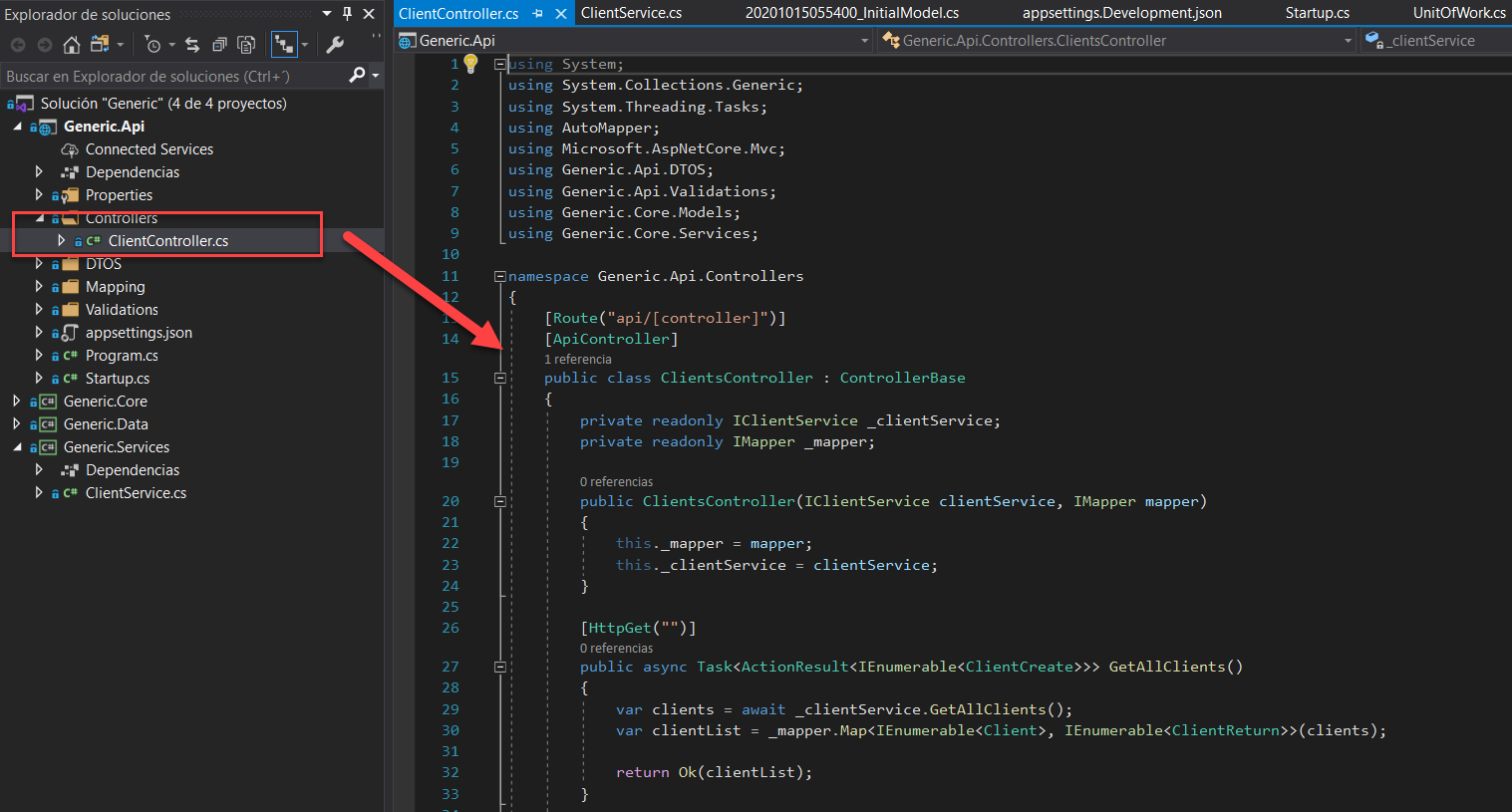
ClientToBeUpdated.Address = Client.Address;

await \_unitOfWork.CommitAsync();

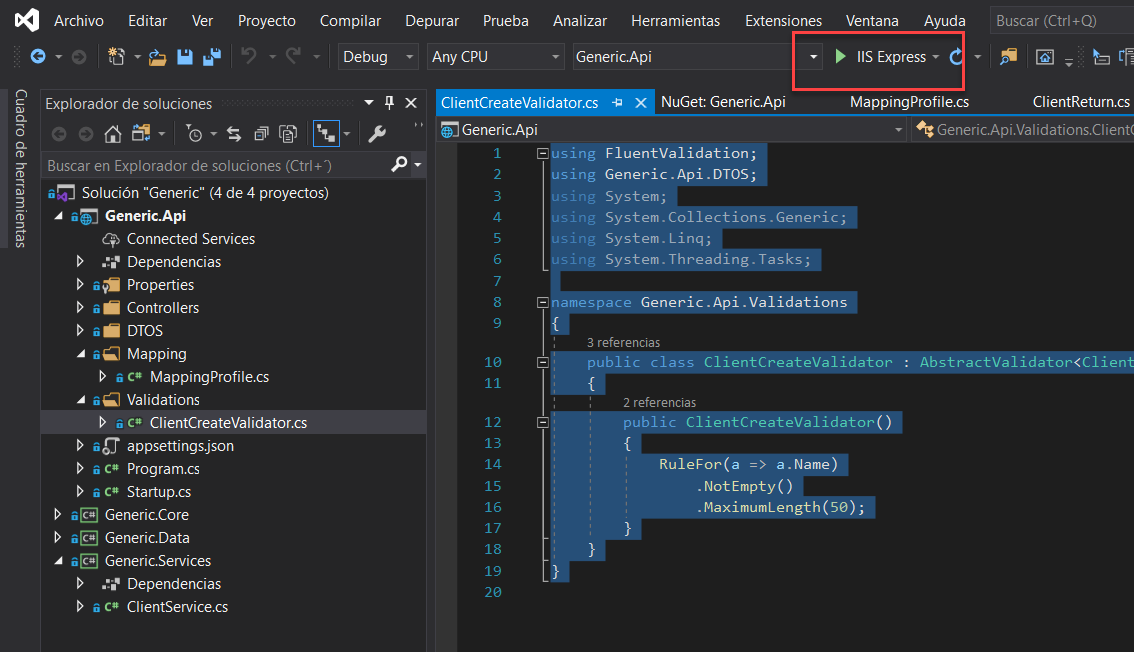
}

}

}



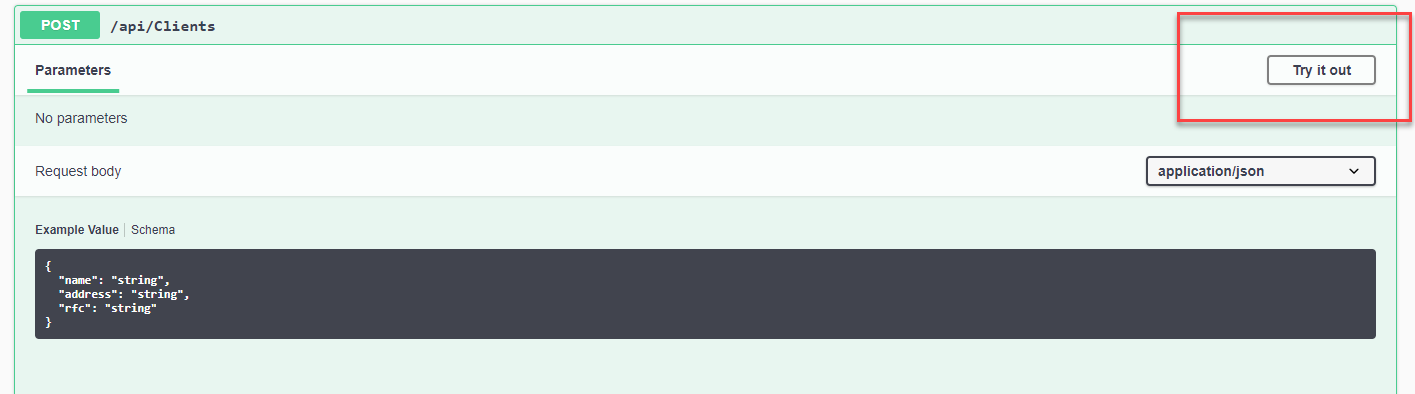
27.- Ejecutamos nuestra API



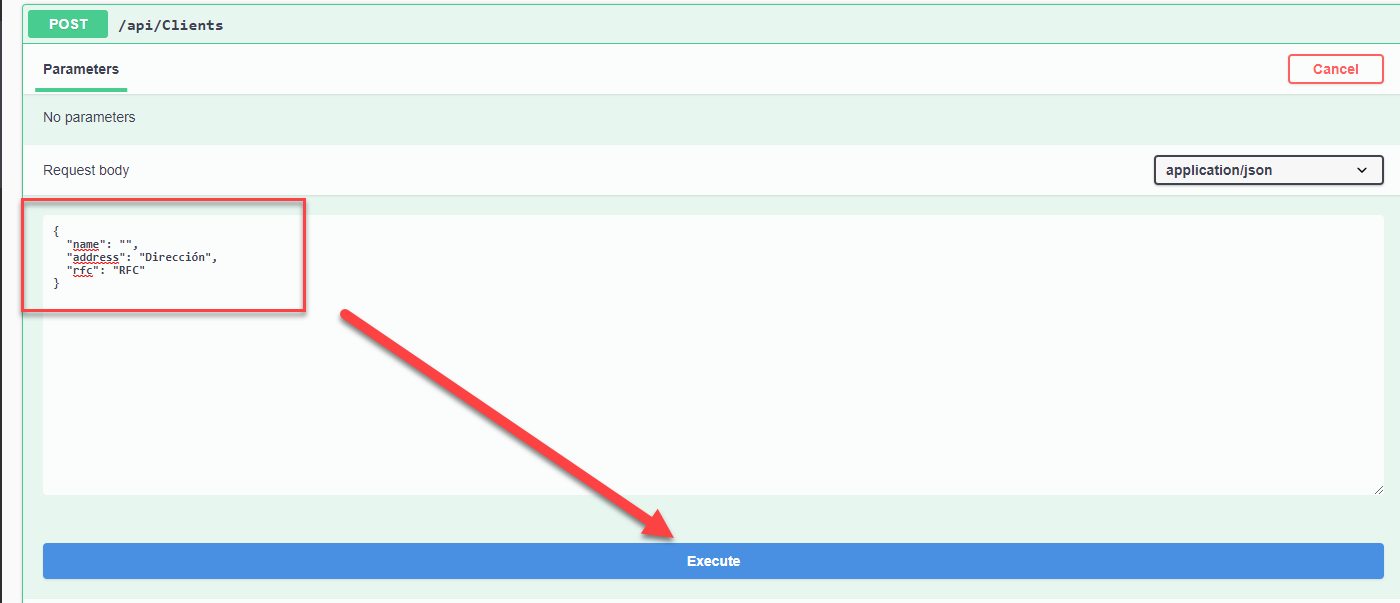
Si estás en modo Consola el comando equivalente en CLI es

dotnet run -p Generic.Api/Generic.Api.csproj

dirigiendo en nuestro navegador hacía la ruta <http://localhost:56864/index.html>, debemos ver el contrato que nos genera Swagger donde ya podemos ver cómo podemos invocar a cada uno de los métodos que hemos codeado en nuestro Controller

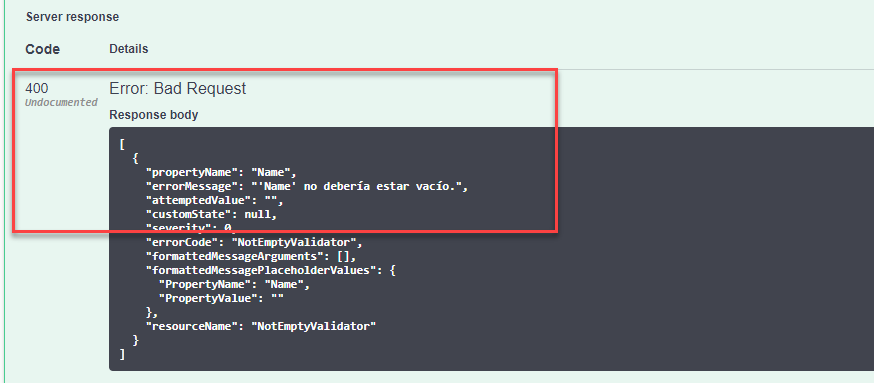


Si damos click a Try it Out entonces podemos empezar a tirar request



NOTA: Observa que ya me da los campos que el método POST requiere de entrada

NOTA 2 : Observa que si no le agrego un name y das click a Execute entonces obtendremos un response 400 con la leyenda de campo requerido esto es por qué así lo decidimos en la regla del punto 25



Por lo que podemos observar que nuestro validador funciona

NOTA 3: Observa que si cumplo con la regla de no vacío en name y das click a Execute entonces obtendremos un response 200 y revisando en nuestra base de datos el registro está correcto

