.Net Clean Architecture

Contenido

[Creando Proyecto 1](#_Toc189738002)

[Definir el dominio 4](#_Toc189738003)

[Objetos de Valor (Object Value) 5](#_Toc189738004)

[Aggreate Root 6](#_Toc189738005)

[Configuración Aplicación 8](#_Toc189738006)

[Patrón CQRS 10](#_Toc189738007)

[Capa Infractructura 13](#_Toc189738008)

[Configuración de entidades (Mapeo) 16](#_Toc189738009)

[Migraciones 17](#_Toc189738010)

[Comandos Terminal 20](#_Toc189738011)

[Volver hacer una migración (borrar anterior migración, generar nueva migración) 21](#_Toc189738012)

[Problem Details 22](#_Toc189738013)

[Instalación de errorOr (en Nuget gallery) 22](#_Toc189738014)

[Implementación de Problem Details 22](#_Toc189738015)

[Prueba de la API con Swagger 29](#_Toc189738016)

[Validación Behavior 31](#_Toc189738017)

[Error Middleware 34](#_Toc189738018)

[Casos por implementar CRUD 36](#_Toc189738019)

[DELETE 38](#_Toc189738020)

[GetAll 40](#_Toc189738021)

[GetByID 41](#_Toc189738022)

[Update 43](#_Toc189738023)

# Creando Proyecto

dotnet new sln -o Proyecto // crea el proyecto solución

Creación de las tres capas básicas para una arquitectura limpia

NOTA: elimina en un principio la clase que aparece por defecto y la carpeta obj

dotnet new classlib -o Domain -f net7.0

dotnet new classlib -o Application -f net7.0

dotnet new classlib -o Infrastructure-f net7.0

Vista:

Captura de pantalla de computadora

El contenido generado por IA puede ser incorrecto.

Crea la API (no borres nada de ella)

dotnet new webapi -o Web.API -f net7.0

NOTA: comprobar que todo está bien hasta el momento 🡪 dotnet build

Captura de pantalla de computadora

El contenido generado por IA puede ser incorrecto.

Establecer relaciones (como, por ejemplo):

dotnet add Application/Application.csproj reference .\Domain\Domain.csproj

Captura de pantalla de computadora

El contenido generado por IA puede ser incorrecto.

Captura de pantalla de computadora

El contenido generado por IA puede ser incorrecto.

Comprobamos que todo está bien

dotnet build

Arrancar la API

dotnet run -p .\Web.API\

Vista:

Captura de pantalla de computadora

El contenido generado por IA puede ser incorrecto.

**NOTA**: abre el navegador en el localhost, es verdad que no aparecerá nada, debes de añadir a la url el swagger:

<http://localhost:5229/swagger/index.html>

Interfaz de usuario gráfica, Texto, Aplicación

El contenido generado por IA puede ser incorrecto.

# Definir el dominio

Añadir primitivos

* Creas una carpeta primitivos en dominio
* Creas una clase AggregateRoot.cs
* Creas una clase DomainEvents.cs
  + Instala Nuget Gallery – MediatR- instalar en Domain y en Application
* Creas una Interfaz UnitOfWork.cs

Captura de pantalla de computadora

El contenido generado por IA puede ser incorrecto.

**AggregateRoot.cs**

namespace Domain.Primitivos;

public abstract class AggregateRoot

{

    // lista de eventos de dominio

    private readonly List<DomainEvent> \_domainEvents = new();

    // propiedad de solo lectura para acceder a la lista de eventos

    public ICollection<DomainEvent> DomainEvents => \_domainEvents;

    // método para levantar eventos de dominio

    protected void Raise(DomainEvent domainEvent) => \_domainEvents.Add(domainEvent);

}

**DomainEvent.cs**

using MediatR;

namespace Domain.Primitivos;

public record DomainEvent(Guid Id) : INotification;

**IUnitOfWork.cs**

namespace Domain.Primitivos;

public interface IUnitOfWork

{

    // método para guardar los cambios en la base de datos

    Task<bool> SaveChangesAsync(CancellationToken cancellationToken = default);

}

# Objetos de Valor (Object Value)

Valores sin identidad, pero hay que implementarlos iguales -> deben ser inmutables

Enlace: <https://learn.microsoft.com/es-es/dotnet/architecture/microservices/microservice-ddd-cqrs-patterns/implement-value-objects>

Creamos una carpeta para los objetos valor en Domain

Creamos una clase Objeto Valor llamada PhoneNumber.cs

**PhoneNumber.cs**

using System.Reflection.Metadata;

using System.Text.RegularExpressions;

namespace Domain.ObjetosValor;

public partial record PhoneNumber

{

    private const int DefaultLength = 9; // 9 digitos

    private const string Pattern = @"^\d{9}$"; // 9 digitos

    private PhoneNumber(string value) => Value = value;

    public string Value { get; init; }

    /// <summary>

    /// Crea un objeto PhoneNumber si el valor es valido

    /// </summary>

    /// <param name="value"></param>

    /// <returns></returns>

    public static PhoneNumber? Create(string value)

    {

        if (string.IsNullOrEmpty(value) || !PhoneNumberRegex().IsMatch(value) || value.Length != DefaultLength)

        {

            return null;

        }

        return new PhoneNumber(value);

    }

    /// <summary>

    /// Expresion regular para validar el valor

    /// </summary>

    /// <returns></returns>

    [GeneratedRegex(Pattern)]

    private static partial Regex PhoneNumberRegex();

}

Vista:

Captura de pantalla de computadora

El contenido generado por IA puede ser incorrecto.

# Aggreate Root

Creamos una nueva carpeta

Creamos una Clase Customer también

Creamos una identidad Customer

Creamos una interfaz CustomerRepository

**Customer.cs**

using Domain.ObjetosValor;

using Domain.Primitivos;

namespace Domain.Customer;

// Customer es una entidad, por lo que hereda de AggregateRoot

public sealed class Customer : AggregateRoot

{

    public Customer(CustomerId id, string name, string lastName, string email, PhoneNumber phoneNumber, Address address)

    {

        Id = id;

        Name = name;

        LastName = lastName;

        Email = email;

        PhoneNumber = phoneNumber;

        Address = address;

    }

    public Customer()

    {

    }

    public CustomerId Id { get; private set; } // Value Object

    public string Name { get; private set; } = string.Empty; // Propiedad

    public string LastName { get; set; } = string.Empty; // Propiedad

    public string FullName => $"{Name} {LastName}"; // Propiedad de solo lectura

    public string Email { get; private set; } = string.Empty; // Propiedad

    public PhoneNumber PhoneNumber { get; private set; }; // Value Object

    public Address Address { get; private set; } // Value Object

    public bool IsActive { get;  set; } // Propiedad

}

**CustomerId.cs**

namespace Domain.Customer;

/// <summary>

/// Identificador de cliente

/// </summary>

/// <param name="value"></param>

public record class CustomerId(Guid value);// Value Object

**ICustomerRepository.cs**

namespace Domain.Customer;

public interface ICustomerRepository

{

    Task<Customer?> GetByIdAsync(CustomerId id); // Metodo

    Task Add(Customer customer); // Metodo

}

Captura de pantalla de computadora

El contenido generado por IA puede ser incorrecto.

# Configuración Aplicación

Creamos:

using Microsoft.Extensions.DependencyInjection;

using FluentValidation.AspNetCore;

using FluentValidation;

namespace Application;

// Clase que contiene los métodos de extensión para la inyección de dependencias.

public static class DependencyInjection

{

    // Método de extensión que añade los servicios de la aplicación.

    public static IServiceCollection AddApplication(this IServiceCollection services)

    {

        // añadiendo los servicios de MediatR

        services.AddMediatR(config =>

        {

            config.RegisterServicesFromAssemblyContaining<ApplicationAssemblyReference>();

        });

        // añadiendo los servicios de FluentValidation

        services.AddValidatorsFromAssemblyContaining<ApplicationAssemblyReference>();

        return services;

    }

}

**ApplicationAssemblyReference.cs**

using System.Reflection;

namespace Application;

/// <summary>

/// Esta clase se utiliza para obtener la referencia a la asamblea de la aplicación.

/// </summary>

public class ApplicationAssemblyReference

{

    // Esta propiedad estática se utiliza para obtener la referencia a la asamblea de la aplicación.

    internal static readonly Assembly Assembly = typeof(ApplicationAssemblyReference).Assembly;

}

Creamos una carpeta Data y dentro de ella una interfaz

**IApplicationDbContext.cs**

namespace Application;

using Domain.Customer;

using Microsoft.EntityFrameworkCore;

/// <summary>

/// Interfaz que define el contexto de la aplicación.

/// </summary>

public interface IApplicationDbContext

{

    public DbSet<Customer> Customers { get; set; } // Propiedad que representa la tabla de clientes en la base de datos.

    // Método que guarda los cambios en la base de datos.

    public Task<int> SaveChangesAsync(CancellationToken cancellationToken = default);

}

**Vista final:**

Captura de pantalla de computadora

El contenido generado por IA puede ser incorrecto.

**NOTA:** Atiende a las versiones de los paquetes instalados porque pueden darte problemas según la versión de .net con la que trabajes;

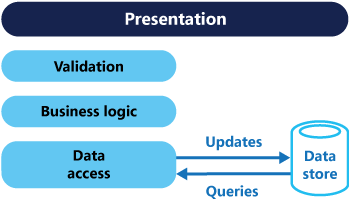
# Patrón CQRS

Arquitectura limpia

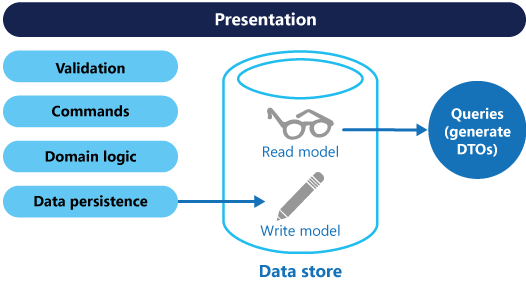
SRP, escalabilidad, extensión, etc.

Enlace: <https://learn.microsoft.com/es-es/azure/architecture/patterns/cqrs>

Inicio (CRUD):



**Solución (CQRS):**



Creación CRUD carpetas. Primero creamos una carpeta Customer en Application y según las responsabilidades (créate, update, etc.) creamos sus correspondientes carpetas.

**Vista:**

Captura de pantalla de computadora

El contenido generado por IA puede ser incorrecto.

Creamos las clases:

**CreateCustomerCommand.cs**

using MediatR;

namespace Application.Customers.Create

{

    // Es una clase sellada, es decir, no puede ser heredada

    public record CreateCustomerCommand(

        string Name,

        string LastName,

        string Email,

        string PhoneNumber,

        string Country,

        string State,

        string City,

        string Street,

        string ZipCode

    ) : IRequest<Unit>;

}

**CreateCustomerCommandHandler.cs**

using Domain.Customer;

using Domain.ObjetosValor;

using Domain.Primitivos;

using MediatR;

namespace Application.Customers.Create;

// Clase sellada que implementa la interfaz IRequestHandler

internal sealed class CreateCustomerCommandHAndler : IRequestHandler<CreateCustomerCommand, Unit>

{

    private readonly ICustomerRepository \_customerRepository;

    private readonly IUnitOfWork \_unitOfWork;

    public CreateCustomerCommandHAndler(ICustomerRepository customerRepository, IUnitOfWork unitOfWork)

    {

        \_customerRepository = customerRepository ?? throw new ArgumentNullException(nameof(customerRepository));

        \_unitOfWork = unitOfWork ?? throw new ArgumentNullException(nameof(unitOfWork));

    }

    // Método que se encarga de manejar la solicitud

    public async Task<Unit> Handle(CreateCustomerCommand request, CancellationToken cancellationToken)

    {

        // Se valida que el nombre no sea nulo o vacío

        if (PhoneNumber.Create(request.PhoneNumber) is not PhoneNumber phoneNumber)

        {

            throw new Exception("Phone number is required. " + nameof(PhoneNumber));

        }

        var address = Address.Create(request.Street, request.City, request.State, request.Country, request.ZipCode);

        // Se valida que la dirección no sea nula

        if (address is null)

        {

            throw new Exception("Address is required. " + nameof(Address));

        }

        var customer = new Customer(new CustomerId(Guid.NewGuid()), request.Name, request.LastName, request.Email, phoneNumber, address);

        if (customer is null)

        {

            throw new Exception("Customer is required. " + nameof(Customer));

        }

        await \_customerRepository.Add(customer); // Se agrega el cliente

        await \_unitOfWork.SaveChangesAsync(cancellationToken); // Se guardan los cambios en la base de datos

        return Unit.Value;

    }

}

# Capa Infractructura

Crear carpeta en infraestructura llamada Persistencia

**ApplicationDbContext.cs**

using Application;

using Domain.Customer;

using Domain.Primitivos;

using MediatR;

using Microsoft.EntityFrameworkCore;

namespace Infrastructure.Persistence

{

    // clase que implementa la interfaz IApplicationDbContext y la interfaz IUnitOfWork

    public class ApplicationDbContext : DbContext, IApplicationDbContext, IUnitOfWork

    {

        private readonly IPublisher \_publisher; // propiedad de solo lectura para acceder al publicador

        public ApplicationDbContext(DbContextOptions options, IPublisher publisher) : base(options)

        {

            // asignar el publicador a la propiedad

            \_publisher = publisher ?? throw new ArgumentNullException(nameof(publisher));

        }

        public DbSet<Customer> Customers { get; set; } // propiedad para acceder a la tabla de clientes

        // método para guardar los cambios en la base de datos

        public override async Task<int> SaveChangesAsync(CancellationToken cancellationToken = new CancellationToken())

        {

            // obtener los eventos de dominio de las entidades que implementan AggregateRoot

            var domainEvents = ChangeTracker.Entries<AggregateRoot>()

                .Select(e => e.Entity)

                .Where(e => e.GetDomainEvents().Any())

                .SelectMany(e => e.GetDomainEvents());

            // guardar los cambios en la base de datos

            var result = await base.SaveChangesAsync(cancellationToken);

            foreach (var domainEvent in domainEvents)

            {

                await \_publisher.Publish(domainEvent, cancellationToken); // publicar los eventos de dominio

            }

            return result;

        }

    }

}

Crea otra carpeta dentro de Persistencia llamada Repositories.

Crea una clase **CustomerRepository.cs**

using Domain.Customer;

using Microsoft.EntityFrameworkCore;

namespace Infrastructure.Persistence.Repositories

{

    // interfaz para el repositorio de clientes

    public class CustomerRepository : ICustomerRepository

    {

        private readonly ApplicationDbContext \_context; // propiedad de solo lectura para acceder al contexto de la aplicación

        public CustomerRepository(ApplicationDbContext context)

        {

            \_context = context ?? throw new ArgumentNullException(nameof(context)); // asignar el contexto a la propiedad

        }

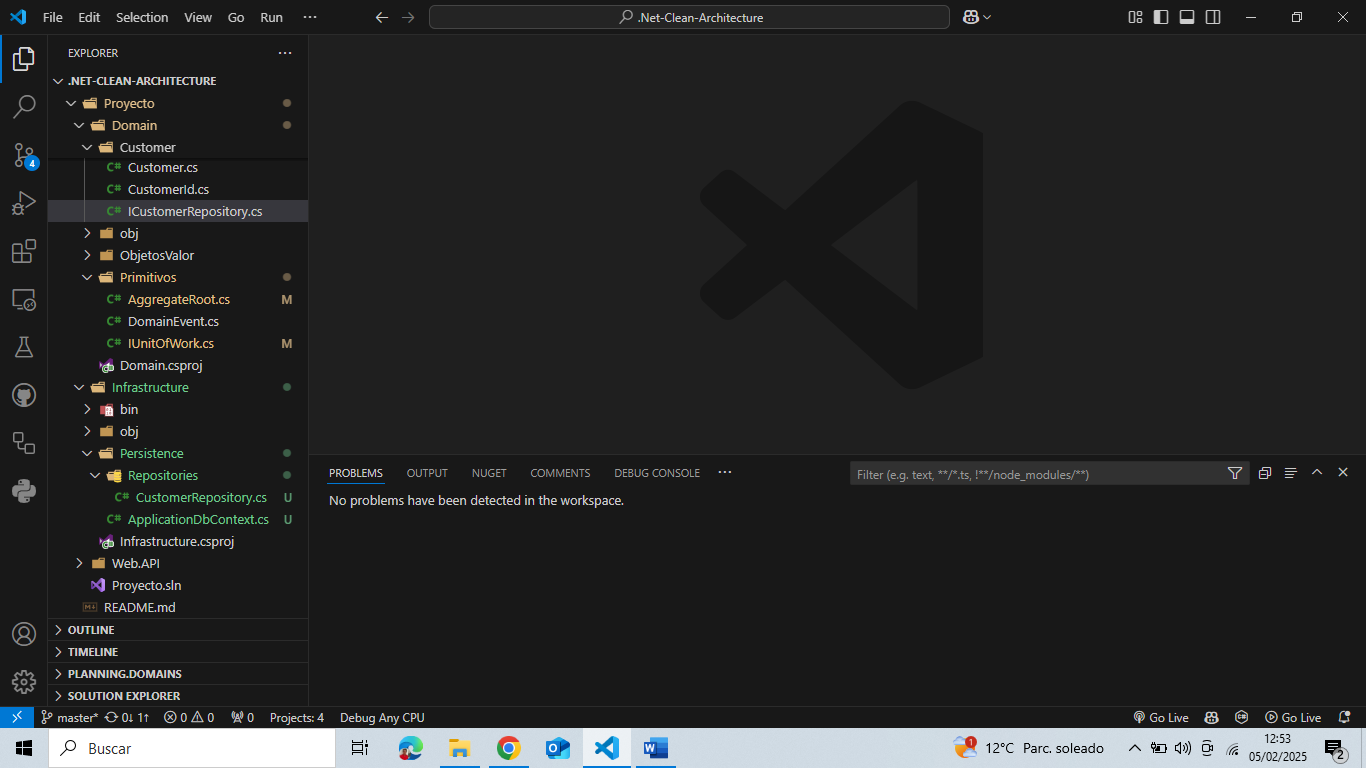
        public async Task<Customer?> GetByIdAsync(CustomerId id) => await \_context.Customers.SingleOrDefaultAsync(c => c.Id == id); // obtener un cliente por su identificador

        public async Task Add(Customer customer) => await \_context.Customers.AddAsync(customer); // agregar un cliente al contexto

    }

}

Vista:



# Configuración de entidades (Mapeo)

Se crea una carpeta en Persistence llamada Configuration y en ella se crea la siguiente clase para el mapeo

**ConfigurationCustomer**.cs

using Domain.Customer;

using Domain.ObjetosValor;

using Microsoft.EntityFrameworkCore;

using Microsoft.EntityFrameworkCore.Metadata.Builders;

namespace Infrastructure.Persistence.Configuration

{

    // Clase para configurar la entidad Customer

    public class CustomerConfiguration : IEntityTypeConfiguration<Customer>

    {

        // Configuración de la entidad Customer

        public void Configure(EntityTypeBuilder<Customer> builder)

        {

            // builder.ToTable("Customers"); // Configuración de la tabla Customers

            builder.HasKey(c => c.Id); // Primary Key

            // Configuración de la propiedad Id

            builder.Property(c => c.Id).HasConversion(

                id => id.Value,

                value => new CustomerId(value) // Value Object

            );

            builder.Property(c => c.Name).IsRequired().HasMaxLength(50); // Configuración de la propiedad Name

            builder.Property(c => c.LastName).HasMaxLength(50); // Configuración de la propiedad LastName

            builder.Ignore(c => c.FullName); // Ignorar propiedad FullName

            builder.Property(c => c.Email).HasMaxLength(255); // Configuración de la propiedad Email

            builder.HasIndex(c => c.Email).IsUnique(); // Configuración de índice para la propiedad Email

            builder.Property(c => c.PhoneNumber).HasConversion(

                phone => phone.Value,

                value => PhoneNumber.Create(value)! // Value Object

            ).HasMaxLength(9); // Configuración de la propiedad Phone

            builder.OwnsOne(c => c.Address, a =>

            {

                a.Property(a => a.Street).HasMaxLength(100); // Configuración de la propiedad Street

                a.Property(a => a.City).HasMaxLength(50); // Configuración de la propiedad City

                a.Property(a => a.State).HasMaxLength(50); // Configuración de la propiedad State

                a.Property(a => a.ZipCode).HasMaxLength(10).IsRequired(); // Configuración de la propiedad ZipCode

            });

        }

    }

}

# Migraciones

Se crea la clase siguiente en la carpeta Infrastructura

**DependencyInjection.cs**

using Application;

using Domain.Customer;

using Domain.Primitivos;

using Infrastructure.Persistence;

using Infrastructure.Persistence.Repositories;

using Microsoft.EntityFrameworkCore;

using Microsoft.Extensions.Configuration;

using Microsoft.Extensions.DependencyInjection;

namespace Infrastructure;

//(1) Se crea una clase estática llamada DependencyInjection

public static class DependencyInjection

{

    //(2) Se crea un método de extensión llamado AddInfrastructure

    public static IServiceCollection AddInfrastructure(this IServiceCollection services, IConfiguration configuration)

    {

        services.AddPersistence(configuration); //Se llama al método AddPersistence

        return services;

    }

    //(3) Se crea un método de extensión llamado AddPersistence

    private static IServiceCollection AddPersistence(this IServiceCollection services, IConfiguration configuration)

    {

        //Se agrega el contexto de la base de datos

        services.AddDbContext<ApplicationDbContext>(options => options.UseSqlServer(configuration.GetConnectionString("Database")));

        //Se agregan los servicios necesarios para la inyección de dependencias

        services.AddScoped<IApplicationDbContext>(sp =>

                sp.GetRequiredService<ApplicationDbContext>());

        // Se agrega el UnitOfWork como servicio

        services.AddScoped<IUnitOfWork>(sp =>

                sp.GetRequiredService<ApplicationDbContext>());

        //Se agregan los repositorios necesarios para la inyección de dependencias

        services.AddScoped<ICustomerRepository, CustomerRepository>();

        return services;

    }

}

**NOTA:** Como denomines en el archivo de configuración siguiente a las credenciales de la BBDD: **“**services.AddDbContext<ApplicationDbContext>(options => options.UseSqlServer(configuration.GetConnectionString("Database")));**”**

Busca el archivo ‘appsettings.Development.json’ en la carpeta Web.API

{

  "ConnectionStrings": {

    "Database": "Data Source=DESKTOP-3829VRG;Initial Catalog=tutorial;Integrated Security=True"

  },

  "Logging": {

    "LogLevel": {

      "Default": "Information",

      "Microsoft.AspNetCore": "Warning"

    }

  }

}

**NOTA:** en este caso se ha usado la configuración de SQLserver.

En la carpeta Web.API debemos de crear una carpeta llamada Extensions. Dentro de ella tenemos que crear una clase:

**MigrationsExtensions.cs**

using Infrastructure.Persistence;

using Microsoft.EntityFrameworkCore;

namespace Web.API.Extensions;

// Clase de extensión para aplicar las migraciones a la base de datos

public static class MigrationExtensions

{

    // Método para aplicar las migraciones a la base de datos

    public static void ApplyMigrations(this WebApplication app){

        // Crear un alcance para acceder a los servicios

        using var scope = app.Services.CreateScope();

        // Obtener el contexto de la aplicación

        var dbContext = scope.ServiceProvider.GetRequiredService<ApplicationDbContext>();

        // Aplicar las migraciones a la base de datos

        dbContext.Database.Migrate();

    }

}

**Debemos de crear otra clase en esta carpeta, Web.API:**

**DependencyInjections.cs**

using Infrastructure.Persistence;

using Microsoft.EntityFrameworkCore;

namespace Web.API.Extensions;

// Clase de extensión para aplicar las migraciones a la base de datos

public static class MigrationExtensions

{

    // Método para aplicar las migraciones a la base de datos

    public static void ApplyMigrations(this WebApplication app){

        // Crear un alcance para acceder a los servicios

        using var scope = app.Services.CreateScope();

        // Obtener el contexto de la aplicación

        var dbContext = scope.ServiceProvider.GetRequiredService<ApplicationDbContext>();

        // Aplicar las migraciones a la base de datos

        dbContext.Database.Migrate();

    }

}

**NOTA:** mucho del código de esta clase ha sido refactorizado de la clase program.cs de esta misma carpeta

Nuestra clase program.cs debe de quedar de la siguiente manera:

**Program.cs**

using Application;

using Infrastructure;

using Web.API;

using Web.API.Extensions;

var builder = WebApplication.CreateBuilder(args);

// Add services to the container.

builder.Services.AddPresentation() // Añadimos la inyección de dependencias de presentación

                .AddInfrastructure(builder.Configuration) // Añadimos la inyección de dependencias de infraestructura

                .AddApplication(); // Añadimos la inyección de dependencias de aplicación

var app = builder.Build();

// Configure the HTTP request pipeline.

if (app.Environment.IsDevelopment())

{

    app.UseSwagger();

    app.UseSwaggerUI();

    app.ApplyMigrations(); // Añadimos la migración

}

app.UseHttpsRedirection();

app.UseAuthorization();

app.MapControllers();

app.Run();

## Comandos Terminal

**NOTA**: la secuencia de instrucciones siguientes no se asegura estar en el orden real, puede que sí o puede que no

**NOTA**: hubo muchas complicaciones respecto a las versiones de .net y los paquetes a instalar

* Montar las migraciones 🡪

dotnet ef migrations add InitialMigration -p .\Infrastructure\ -s .\Web.API\ -o .\Infrastructure\Persistence\Migrations\

**NOTA**: Se crea en la carpeta Infrastructure\Persistence una carpeta Migrations con tres clases mapeadas con nuestras entidades para la BBDD

Otra forma muy similar (sin autocompletar):

dotnet ef migrations add InitialMigration -p Infrastructure -s Web.API -o Persistence/Migrations

* Lanzar la BBDD 🡪

dotnet ef database update -p Infrastructure -s Web.API

* Limpiar proyecto 🡪

dotnet clean

* Restaurar proyecto 🡪

dotnet restore

Resultado BBDD en HeidiSql:

Una captura de pantalla de una computadora

El contenido generado por IA puede ser incorrecto.

## Volver hacer una migración (borrar anterior migración, generar nueva migración)

Borra manualmente los archivos de la carpeta Migrations (esta incluida). También los obj (carpeta) de las distintas carpetas (Application, Domain, etc)

Haz una limpieza del proyecto 🡪 dotnet clean

Vuelve a construir el proyecto 🡪 dotnet build

Finalmente, vuelve hacer 🡪 dotnet ef migrations add InitialMigration -p Infrastructure -s Web.API -o Persistence/Migrations

Y lanzala 🡪 dotnet ef database update -p Infrastructure -s Web.API

# Problem Details

Formato de respuesta usado en las APIs HTTP

Proporciona información acerca de los errores

Permite consistencia

Ejemplo 🡪 Error 400.

## Instalación de errorOr (en Nuget gallery)

Vamos a la galería buscamos errorOr e instalamos en todas las carpetas menos en Infrastructure

## Implementación de Problem Details

1. Vamos a Application y En nuestras operaciones CRUD, en sus commands los iremos tipando con ErrorOr

**CreateCustomerCommand.cs**

using ErrorOr;

using MediatR;

namespace Application.Customers.Create

{

    // Es una clase sellada, es decir, no puede ser heredada

    public record CreateCustomerCommand(

        string Name,

        string LastName,

        string Email,

        string PhoneNumber,

        string Country,

        string State,

        string City,

        string Street,

        string ZipCode

    ) : IRequest<ErrorOr<Unit>>;// Implementa la interfaz IRequest y se le pasa el tipo de retorno ErrorOr<Unit>

}

**CreateCustomerCommandHandler.cs**

using Domain.Customer;

using Domain.ObjetosValor;

using Domain.Primitivos;

using ErrorOr;

using MediatR;

namespace Application.Customers.Create;

// Clase sellada que implementa la interfaz IRequestHandler

internal sealed class CreateCustomerCommandHAndler : IRequestHandler<CreateCustomerCommand, ErrorOr<Unit>>

{

    private readonly ICustomerRepository \_customerRepository; // Se declara una variable de solo lectura de tipo ICustomerRepository

    private readonly IUnitOfWork \_unitOfWork; // Se declara una variable de solo lectura de tipo IUnitOfWork

    public CreateCustomerCommandHAndler(ICustomerRepository customerRepository, IUnitOfWork unitOfWork)

    {

        // Se inicializan las variables

        \_customerRepository = customerRepository ?? throw new ArgumentNullException(nameof(customerRepository));

        \_unitOfWork = unitOfWork ?? throw new ArgumentNullException(nameof(unitOfWork));

    }

    // Método que se encarga de manejar la solicitud

    public async Task<ErrorOr<Unit>> Handle(CreateCustomerCommand request, CancellationToken cancellationToken)

    {

        try

        {

            // Se valida que el nombre no sea nulo o vacío

            if (PhoneNumber.Create(request.PhoneNumber) is not PhoneNumber phoneNumber)

            {

                // throw new Exception("Phone number is required. " + nameof(PhoneNumber))

                return Error.Validation("Customer.Phone " + "Customer phone number is required."); // Se retorna un error de validación

            }

            var address = Address.Create(request.Street, request.City, request.State, request.Country, request.ZipCode);

            // Se valida que la dirección no sea nula

            if (address is null)

            {

                // throw new Exception("Address is required. " + nameof(Address));

                return Error.Validation("Customer.Address: " + "Customer address is required."); // Se retorna un error de validación

            }

            var customer = new Customer(new CustomerId(Guid.NewGuid()), request.Name, request.LastName, request.Email, phoneNumber, address);

            if (customer is null)

            {

                // throw new Exception("Customer.Customer: " + "Customer is null");

                 return Error.Validation("Customer.Customer: " + "Customer is null"); // Se retorna un error de validación

            }

            await \_customerRepository.Add(customer); // Se agrega el cliente

            await \_unitOfWork.SaveChangesAsync(cancellationToken); // Se guardan los cambios en la base de datos

            return Unit.Value;

        }

        catch (Exception ex)

        {

            //

            return Error.Failure("CreateCustomer.Failure" + ex.Message); // Se retorna un error de fallo

        }

    }

}

1. Luego se pasa a la carpeta de WebAPI, donde creamos otra llamada Common y dentro de esta, otras dos carpetas llamadas Errors y Http

Vista:

Captura de pantalla de computadora

El contenido generado por IA puede ser incorrecto.

**ProyectioProblemDetailsFactory.cs**

using System.Diagnostics;

using ErrorOr;

using Microsoft.AspNetCore.Mvc;

using Microsoft.AspNetCore.Mvc.Infrastructure;

using Microsoft.AspNetCore.Mvc.ModelBinding;

using Web.API.Common.Http;

namespace Web.API.Common.Errors;

// Clase que implementa la interfaz IProblemDetailsFactory para la creación de problemas

public class ProyectoProblemDetailsFactory : ProblemDetailsFactory

{

    private readonly ApiBehaviorOptions \_options; // Opciones de comportamiento de la API

    public ProyectoProblemDetailsFactory(ApiBehaviorOptions options)

    {

        // Inicializamos las opciones

        this.\_options = options ?? throw new ArgumentNullException(nameof(options));

    }

    // Método para crear detalles de problemas de errores genéricos

    public override ProblemDetails CreateProblemDetails(

        HttpContext httpContext, int? statusCode = null,

         string? title = null, string? type = null, string? detail = null, string? instance = null)

    {

        statusCode ??= 500; // Si el código de estado es nulo, lo establecemos a 500

        var problemDetails = new ProblemDetails

        {

            Status = statusCode,

            Title = title,

            Type = type,

            Detail = detail,

            Instance = instance

        };

        ApplyProblemDetailsDefaults(httpContext, problemDetails, statusCode.Value); // Aplicamos los valores por defecto

        return problemDetails;

    }

    // Método para crear detalles de problemas de validación de errores

    public override ValidationProblemDetails CreateValidationProblemDetails(HttpContext httpContext,

     ModelStateDictionary modelStateDictionary, int? statusCode = null, string? title = null,

      string? type = null, string? detail = null, string? instance = null)

    {

        if (modelStateDictionary == null)

        {

            throw new ArgumentNullException(nameof(modelStateDictionary));

        }

        statusCode ??= 400; // Si el código de estado es nulo, lo establecemos a 400

        var problemDetails = new ValidationProblemDetails(modelStateDictionary)

        {

            Status = statusCode,

            Title = title,

            Type = type,

            Detail = detail,

            Instance = instance

        };

        if (title == null)

        {

            problemDetails.Title = title; // Establecemos el título

        }

        // Aplicamos los valores por defecto

        ApplyProblemDetailsDefaults(httpContext, problemDetails, statusCode.Value);

        return problemDetails;

    }

    // Método para aplicar los valores por defecto de los detalles del problema

    private void ApplyProblemDetailsDefaults(HttpContext httpContext, ProblemDetails problemDetails, int statusCode)

    {

        // Si el código de estado se encuentra en el mapeo de errores del cliente

        if (\_options.ClientErrorMapping.TryGetValue(statusCode, out var clientErrorData))

        {

            problemDetails.Title ??= clientErrorData.Title; // Establecemos el título

            problemDetails.Type ??= clientErrorData.Link; // Establecemos el tipo

        }

        // Si el código de estado es 500

        var traceId = Activity.Current?.Id ?? httpContext.TraceIdentifier;

        if (traceId != null)

        {

            problemDetails.Extensions["traceId"] = traceId; // Establecemos el identificador de traza

        }

        // Si el contexto HTTP contiene errores

        var errors = httpContext.Items[HttpContextItemKeys.Errors] as List<Error>;

        if (errors != null)

        {

            problemDetails.Extensions.Add("errorCodes", errors.Select(e => e.Code)); // Añadimos los códigos de error

        }

    }

}

**HttpContextItemsKEys.cs**

namespace Web.API.Common.Http

{

    // Clase que contiene las claves de los elementos del contexto HTTP

    public static class HttpContextItemKeys

    {

        public const string Errors = "errors"; // Clave para los errores

    }

}

**Program.cs**

app.UseExceptionHandler("/error"); // Añadimos el manejador de excepciones

1. Se crea también en Web.API otra clase pero esta vez en una carpeta distinta que teníamos de antes, llamada Controller. En ella se crea:

**ApiController.cs**

using ErrorOr;

using Microsoft.AspNetCore.Mvc;

using Microsoft.AspNetCore.Mvc.ModelBinding;

using Web.API.Common.Http;

namespace Web.Api.Controller;

// Clase base para los controladores de la API

public class ApiController : ControllerBase

{

    // Método para devolver un error

    protected IActionResult Problem(List<Error> errors)

    {

        // Si no hay errores en la lista

        if (errors is null || errors.Count == 0)

        {

            return Problem(); // 500

        }

        // Si todos los errores son de validación

        if (errors.All(error => error.Type == ErrorType.Validation))

        {

            return ValidationProblem(errors); // 400

        }

        // Si hay errores de validación y otros tipos de errores mezclados en la lista de errores

        HttpContext.Items[HttpContextItemKeys.Errors] = errors;

        return Problem(errors[0]);

    }

    // Método para devolver un error

    private IActionResult Problem(Error error)

    {

        // Dependiendo del tipo de error, se devuelve un código de estado HTTP diferente

        var statusCode = error.Type switch

        {

            ErrorType.Conflict => StatusCodes.Status409Conflict,

            ErrorType.Validation => StatusCodes.Status400BadRequest,

            ErrorType.NotFound => StatusCodes.Status404NotFound,

            ErrorType.Failure => StatusCodes.Status500InternalServerError,

            \_ => StatusCodes.Status500InternalServerError

        };

        return Problem(statusCode: statusCode, title: error.Description);

    }

    // Método para devolver un error de validación

    private IActionResult ValidationProblem(List<Error> errors)

    {

        var modelState = new ModelStateDictionary(); // Diccionario de errores de validación

        foreach (var error in errors)

        {

            // Se añade un error de validación al diccionario de errores

            modelState.AddModelError(error.Code, error.Description);

        }

        return ValidationProblem(modelState);

    }

}

# Prueba de la API con Swagger

Creamos en la carpeta API.Web/Controllers las siguientes clases:

**ErrorsController.cs**

using Microsoft.AspNetCore.Diagnostics;

using Microsoft.AspNetCore.Mvc;

namespace Web.API.Controllers;

// Controlador para manejar los errores de la API

public class ErrorsController : ControllerBase

{

    [ApiExplorerSettings(IgnoreApi = true)] // Ignorar en la documentación de Swagger

    [Route("/error")] // Ruta para manejar errores

    public IActionResult Error()

    {

        // Obtener el error de la petición

        Exception? exception = HttpContext.Features.Get<IExceptionHandlerFeature>()?.Error;

        return Problem();

    }

}

**CustomController.cs**

using Application.Customers.Create;

using MediatR;

using Microsoft.AspNetCore.Mvc;

using Web.Api.Controller;

using Web.API.Controllers;

namespace Api.Web.Controllers;

// Controlador para manejar las peticiones relacionadas con los clientes

[Route("customers")] // Ruta base para las peticiones relacionadas con los clientes

public class CustomersController : ApiController

{

    private readonly ISender \_mediator; // Mediator para enviar comandos y consultas

    public CustomersController(ISender mediator)

    {

        \_mediator = mediator ?? throw new ArgumentNullException(nameof(mediator));

    }

    // Crear un cliente con los datos recibidos en el cuerpo de la petición

    [HttpPost] // POST /customers

    public async Task<IActionResult> Create([FromBody] CreateCustomerCommand command)

    {

        var createResult = await \_mediator.Send(command); // Enviar el comando para crear un cliente

        return createResult.Match(

            customerId => Ok(customerId), // Si se creó el cliente, devolver el ID

            errors => Problem(errors) // Si hubo errores, devolverlos

        );

    }

}

En la clase APIController de la carpeta Controllers de la API.WEB debemos de añadir una ruta:

**ApiController.cs**

[ApiController] // Decorador para indicar que es un controlador de API

public class ApiController : ControllerBase

{

Finalmente, probamos y ejecutamos nuestro proyecto (program.cs RUN) y se nos abrirá con swagger la interfaz

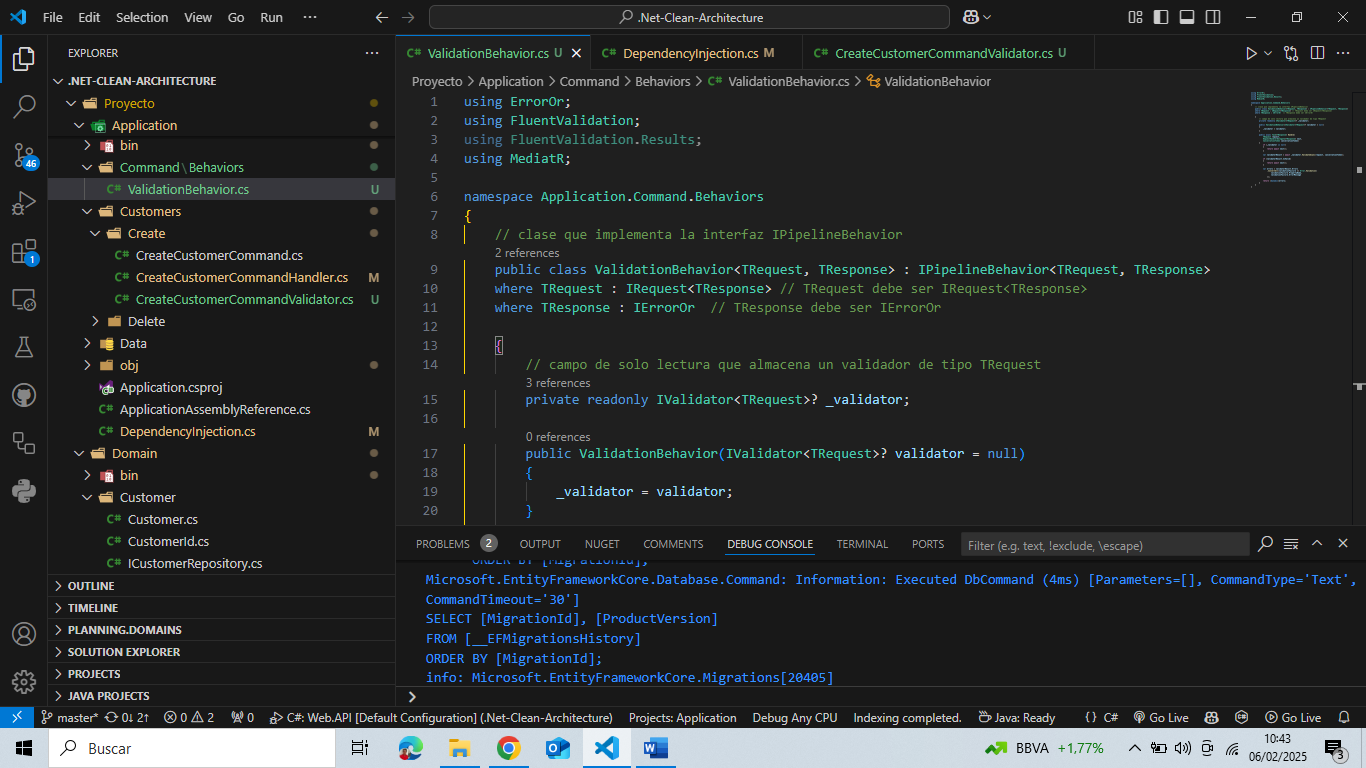
Interfaz de usuario gráfica, Aplicación, Word

El contenido generado por IA puede ser incorrecto.

# Validación Behavior

Creamos en la carpeta Application una carpeta Command y en ella otra mas llamada Behaviors

Vista:



En la carpeta Behaviors creamos una clase llamada:

**ValidationBehavior.cs**

using ErrorOr;

using FluentValidation;

using FluentValidation.Results;

using MediatR;

namespace Application.Command.Behaviors

{

    // clase que implementa la interfaz IPipelineBehavior

    public class ValidationBehavior<TRequest, TResponse> : IPipelineBehavior<TRequest, TResponse>

    where TRequest : IRequest<TResponse> // TRequest debe ser IRequest<TResponse>

    where TResponse : IErrorOr  // TResponse debe ser IErrorOr

    {

        // campo de solo lectura que almacena un validador de tipo TRequest

        private readonly IValidator<TRequest>? \_validator;

        public ValidationBehavior(IValidator<TRequest>? validator = null)

        {

            \_validator = validator;

        }

        // método que maneja la solicitud

        public async Task<TResponse> Handle(

            TRequest request,

            RequestHandlerDelegate<TResponse> next,

            CancellationToken cancellationToken)

        {

            if (\_validator is null)

            {

                return await next(); // si el validador es nulo, se llama al siguiente manejador

            }

            // se valida la solicitud

            var validatorResult = await \_validator.ValidateAsync(request, cancellationToken);

            if (validatorResult.IsValid)

            {

                return await next(); // si la solicitud es válida, se llama al siguiente manejador

            }

            // si la solicitud no es válida, se crea una lista de errores

            var errors = validatorResult.Errors

                .ConvertAll(validationFailure => Error.Validation(

                    validationFailure.PropertyName, // se obtiene el nombre de la propiedad

                    validationFailure.ErrorMessage // se obtiene el mensaje de error

                ));

            return (dynamic)errors; // se retorna la lista de errores como un objeto dinámico

        }

    }

}

En nuestra carpeta Application/Customers debemos de añadir una nueva clase para validar llamada:

**CreateCustomerCommandValidator.cs**

using FluentValidation;

namespace Application.Customers.Create;

//  Clase que contiene las reglas de validación para el comando de creación de un cliente.

public class CreateCustomerCommandValidator : AbstractValidator<CreateCustomerCommand>

{

    // Constructor de la clase.

    public CreateCustomerCommandValidator()

    {

        // Reglas de validación para el comando de creación de un cliente.

        RuleFor(x => x.Name)

            .NotEmpty() // El campo no puede estar vacío.

            .MaximumLength(50) // Longitud máxima del campo.

            .WithName("Name: "); // Nombre del campo en el mensaje de error.

        RuleFor(x => x.LastName)

            .NotEmpty()

            .MaximumLength(50)

            .WithName("Last Name: ");

        RuleFor(x => x.Email)

            .NotEmpty()

            .EmailAddress() // El campo debe ser una dirección de correo electrónico válida.

            .MaximumLength(255);

        RuleFor(x => x.PhoneNumber)

            .NotEmpty()

            .MaximumLength(9)

            .WithName("Phone Number: ");

        RuleFor(x => x.Country)

            .NotEmpty()

            .MaximumLength(50);

        RuleFor(x => x.State)

            .NotEmpty()

            .MaximumLength(50);

        RuleFor(x => x.City)

            .NotEmpty()

            .MaximumLength(50);

        RuleFor(x => x.Street)

            .NotEmpty()

            .MaximumLength(50);

        RuleFor(x => x.ZipCode)

            .NotEmpty()

            .MaximumLength(10)

            .WithName("Zip Code: ");

    }

}

Por último, en la carpeta Application debemos añadir a nuestra clase DependencyInjection.cs un nuevo servicio:

**DependencyInjection.cs**

        // añadiendo los servicios de FluentValidation

        services.AddScoped(

            typeof(IPipelineBehavior<,>), // tipo de servicio a añadir al contenedor de servicios

            typeof(ValidationBehavior<,>) // tipo de implementación del servicio

        );

# Error Middleware

**NOTA**: Definición: Manejador de Errores

Crear en Web.API una carpeta denominda Middleware, en ella creamos una clase:

**GlobalExceptionHandlingMiddleware.cs**

using System.Net;

using System.Text.Json;

using Microsoft.AspNetCore.Mvc;

namespace Web.Api.Middlewares;

// Clase para el manejo de excepciones globales

public class GlobalExceptionHandlingMiddleware : IMiddleware

{

    private readonly ILogger<GlobalExceptionHandlingMiddleware> \_logger; // Logger para el middleware

    // Constructor de la clase

    public GlobalExceptionHandlingMiddleware(ILogger<GlobalExceptionHandlingMiddleware> logger) => \_logger = logger;

    // Método para invocar el middleware

    public async Task InvokeAsync(HttpContext context, RequestDelegate next)

    {

        try

        {

            await next(context); // Invocar el siguiente middleware

        }

        catch (Exception ex)

        {

            \_logger.LogError(ex, ex.Message); // Loggear la excepción

            context.Response.StatusCode = (int)HttpStatusCode.InternalServerError; // Establecer el código de estado

            ProblemDetails problem = new()

            {

                Title = "An error occurred. Server error",

                Detail = "An internal server has ocurred: " + ex.Message,

                Status = (int)HttpStatusCode.InternalServerError,

                Type = "Server error"

            };

            string json = JsonSerializer.Serialize(problem); // Serializar el objeto a JSON

            context.Response.ContentType = "application/json"; // Establecer el tipo de contenido

            await context.Response.WriteAsync(json); // Escribir la respuesta

        }

    }

}

En esta carpeta WEB.API, añadiremos en la clase DependencyInjection.cs el siguiente servicio:

**DependencyInjection.cs**

        // Agregar la inyección de dependencias de FluentValidation

        services.AddTransient<GlobalExceptionHandlingMiddleware>();

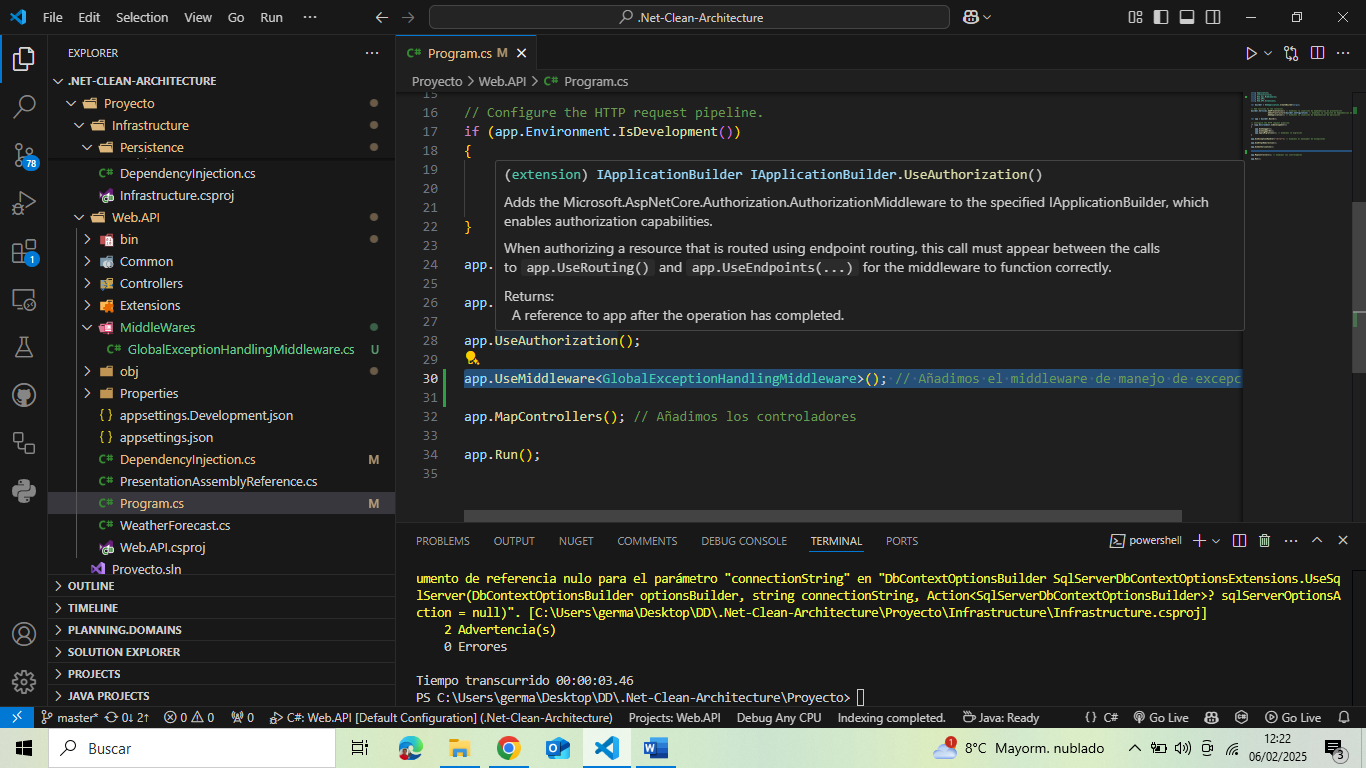
Por último, en program.cs se añadirá:

**Program.cs**

app.UseMiddleware<GlobalExceptionHandlingMiddleware>(); // Añadimos el middleware de manejo de excepciones

**NOTA:** gracias a añadir el manejador global Middleware, podremos eliminar entonces todos aquello ty-catch de nuestros casos CRUD (Application/Customer/Create por ejemplo, etc.)

Vista final:



## Casos por implementar CRUD

Al añadir mas casos, debemos de implementar código adicional en Infrastructure/Persistence/Repositories/CustomerRepository.cs

**CustomerRepository.cs**

using Domain.Customer;

using Microsoft.EntityFrameworkCore;

namespace Infrastructure.Persistence.Repositories

{

    // interfaz para el repositorio de clientes

    public class CustomerRepository : ICustomerRepository

    {

        private readonly ApplicationDbContext \_context; // propiedad de solo lectura para acceder al contexto de la aplicación

        public CustomerRepository(ApplicationDbContext context)

        {

            \_context = context ?? throw new ArgumentNullException(nameof(context)); // asignar el contexto a la propiedad

        }

        public async Task<Customer?> GetByIdAsync(CustomerId id) => await \_context.Customers.SingleOrDefaultAsync(c => c.Id == id); // obtener un cliente por su identificador

        public async Task Add(Customer customer) => await \_context.Customers.AddAsync(customer); // agregar un cliente al contexto

        public void Delete(Customer customer) => \_context.Customers.Remove(customer); // eliminar un cliente del contexto

        public void Update(Customer customer) => \_context.Customers.Update(customer); // actualizar un cliente en el contexto

        public async Task<List<Customer>> GetAll() => await \_context.Customers.ToListAsync(); // obtener todos los clientes

    }

}

En nuestra carpeta Domain, también debemos de hacer ciertas modificaciones:

En la carpeta Customer de Domain, la clase ICustomerRepository.cs sufrirá cambios añadidos:

**ICustomerRepository.cs**

namespace Domain.Customer;

public interface ICustomerRepository

{

    Task<Customer?> GetByIdAsync(CustomerId id); // Metodo

    Task Add(Customer customer); // Metodo

    void Delete(Customer customer); // Metodo

    void Update(Customer customer); // Metodo

    Task<List<Customer>> GetAll(); // Metodo

}

También en la clase Customer. Es un método que se necesitara en el caso Update de CRUD para que nos retorne un nuevo Customer con una id determinado

**Customer.cs**

// Metodo de clase que retorna una nueva instancia de Customer

    public static Customer Update(CustomerId customerId, string name, string lastName, string email, PhoneNumber phoneNumber, Address address)

    {

        return new Customer(customerId, name, lastName, email, phoneNumber, address);

    }

### DELETE

**DeleteCustomerCommand.cs**

namespace Application.Customers.Delete;

using ErrorOr;

using MediatR;

public record DeleteCustomerCommand(Guid Id) : IRequest<ErrorOr<Unit>>{}

**DeleteCustomerCommandValidator.cs**

using FluentValidation;

namespace Application.Customers.Delete;

// Clase sellada que hereda de AbstractValidator y recibe un DeleteCustomerCommand

public class DeleteCustomerCommandValidator : AbstractValidator<DeleteCustomerCommand>

{

    // Constructor que inicializa la regla de validación

    public DeleteCustomerCommandValidator()

    {

        RuleFor(x => x.Id).NotEmpty(); // Se valida que el Id no sea nulo

    }

}

**DeleteCustomerCommandHandler.cs**

namespace Application.Customers.Delete

{

    using Domain.Customer;

    using Domain.Primitivos;

    using ErrorOr;

    using MediatR;

    using System;

    // Clase sellada que implementa la interfaz IRequestHandler

    internal sealed class DeleteCustomerCommandHandler : IRequestHandler<DeleteCustomerCommand, ErrorOr<Unit>>

    {

        private readonly ICustomerRepository \_customerRepository; // Se declara una variable de solo lectura de tipo ICustomerRepository

        private readonly IUnitOfWork \_unitOfWork; // Se declara una variable de solo lectura de tipo IUnitOfWork

        // Constructor que recibe un ICustomerRepository y un IUnitOfWork

        public DeleteCustomerCommandHandler(ICustomerRepository customerRepository, IUnitOfWork unitOfWork)

        {

            //  Se valida que el customerRepository no sea nulo

            \_customerRepository = customerRepository ?? throw new ArgumentNullException(nameof(customerRepository));

            \_unitOfWork = unitOfWork ?? throw new ArgumentNullException(nameof(unitOfWork));

        }

        // Método que se encarga de manejar la solicitud

        public async Task<ErrorOr<Unit>> Handle(DeleteCustomerCommand command, CancellationToken cancellationToken)

        {

            // Se valida que el id no sea nulo

            if (await \_customerRepository.GetByIdAsync(new CustomerId(command.Id)) is not Customer customer)

            {

                // throw new Exception("Customer not found.");

                return Error.NotFound("Customer.NotFound", "The customer with the provide Id was not found.");

            }

            \_customerRepository.Delete(customer); // Se elimina el cliente

            await \_unitOfWork.SaveChangesAsync(cancellationToken); // Se guardan los cambios en la base de datos

            return Unit.Value; // Se retorna un valor Unit

        }

    }

}

### GetAll

**GetAllCustomerQuery.cs**

using Customers.Common;

using ErrorOr;

using MediatR;

namespace Application.Customers.GetAll;

// Clase sellada que implementa la interfaz IRequest

public record GetAllCustomersQuery() : IRequest<ErrorOr<IReadOnlyList<CustomerResponse>>>;

**GetAllCustomerQueryHandler.cs**

using System.Linq;

using Customers.Common;

using Domain.Customer;

using ErrorOr;

using MediatR;

namespace Application.Customers.GetAll;

// Clase sellada que implementa la interfaz IRequestHandler

internal sealed class GetAllCustomersQueryHandler : IRequestHandler<GetAllCustomersQuery, ErrorOr<IReadOnlyList<CustomerResponse>>>

{

    private readonly ICustomerRepository \_customerRepository;   // Se declara una variable de solo lectura de tipo ICustomerRepository

    public GetAllCustomersQueryHandler(ICustomerRepository customerRepository)

    {

        \_customerRepository = customerRepository ?? throw new ArgumentNullException(nameof(customerRepository));

    }

    // Método que se encarga de manejar la solicitud

    public async Task<ErrorOr<IReadOnlyList<CustomerResponse>>> Handle(GetAllCustomersQuery request, CancellationToken cancellationToken)

    {

        // Se obtienen todos los clientes y se convierten a una lista de CustomerResponse

        IReadOnlyList<Customer> customers = (IReadOnlyList<Customer>)await \_customerRepository.GetAll();

        // Se retorna la lista de CustomerResponse

        return customers.Select(customer => new CustomerResponse(

                customer.Id.Value,

                customer.FullName,

                customer.Email,

                customer.PhoneNumber.Value,

                new AddressResponse(

                    customer.Address.Street,

                    customer.Address.City,

                    customer.Address.State,

                    customer.Address.Country,

                    customer.Address.ZipCode)

            )).ToList();

    }

}

**NOTA:** Se debe de crear una carpeta en application denominada Common, en la que tendremos una clase con dos clases Record (dtos):

**CustomerResponse.cs**

namespace Customers.Common;

//  Clase sellada que recibe un Guid, un string, un string, un string y un AddressResponse

public record CustomerResponse(Guid Id, string Name, string Email, string Phone, AddressResponse Address);

// Clase sellada que recibe un string, un string, un string, un string y un string

public record AddressResponse(string Street, string City, string State, string Country, string ZipCode);

### GetByID

**GetCustomerByIDQuery.cs**

using Customers.Common;

using ErrorOr;

using MediatR;

namespace Application.Customers.GetById;

// Clase sellada que implementa la interfaz IRequest y recibe un Guid

public record GetCustomerByIdQuery(Guid Id) : IRequest<ErrorOr<CustomerResponse>>;

**GetCustomerByIdQueryHandler.cs**

using Customers.Common;

using Domain.Customer;

using ErrorOr;

using MediatR;

namespace Application.Customers.GetById;

// Clase sellada que implementa la interfaz IRequest y recibe un Guid

internal sealed class GetCustomerByIdQueryHandler : IRequestHandler<GetCustomerByIdQuery, ErrorOr<CustomerResponse>>

{

    private readonly ICustomerRepository \_customerRepository; // Se declara una variable de solo lectura de tipo ICustomerRepository

    public GetCustomerByIdQueryHandler(ICustomerRepository customerRepository)

    {

        \_customerRepository = customerRepository ?? throw new ArgumentNullException(nameof(customerRepository));

    }

    // Método que se encarga de manejar la solicitud

    public async Task<ErrorOr<CustomerResponse>> Handle(GetCustomerByIdQuery request, CancellationToken cancellationToken)

    {

        // Se obtiene el cliente por el id proporcionado en la solicitud y se almacena en la variable customer de tipo Customer

        Customer? customer = await \_customerRepository.GetByIdAsync(new CustomerId(request.Id));

        // Se retorna un error si el cliente es nulo o un CustomerResponse si no lo es

        return customer is null

            ? Error.NotFound("Customer.NotFound", "The customer with the provide Id was not found.")

            : new CustomerResponse(

                customer.Id.Value,

                customer.FullName,

                customer.Email,

                customer.PhoneNumber.Value,

                new AddressResponse(

                    customer.Address.Street,

                    customer.Address.City,

                    customer.Address.State,

                    customer.Address.Country,

                    customer.Address.ZipCode)

            );

    }

}

### Update

**UpdateCustomerCommand.cs**

using ErrorOr;

using MediatR;

namespace Application.Customers.Update;

public record UpdateCustomerCommand(Guid Id,

    string Name,

    string LastName,

    string Email,

    string PhoneNumber,

    string Country,

    string State,

    string City,

    string Street,

    string ZipCode) : IRequest<ErrorOr<Unit>>;

**UpdateCustomerCommandHandler.cs**

using Domain.Customer;

using Domain.ObjetosValor;

using Domain.Primitivos;

using ErrorOr;

using MediatR;

namespace Application.Customers.Update;

// Clase sellada que implementa la interfaz IRequest y recibe un Guid

internal sealed class UpdateCustomerCommandHandler : IRequestHandler<UpdateCustomerCommand, ErrorOr<Unit>>

{

    private readonly ICustomerRepository \_customerRepository; // Se declara una variable de solo lectura de tipo ICustomerRepository

    private readonly IUnitOfWork \_unitOfWork; // Se declara una variable de solo lectura de tipo IUnitOfWork

    public UpdateCustomerCommandHandler(ICustomerRepository customerRepository, IUnitOfWork unitOfWork)

    {

        \_customerRepository = customerRepository ?? throw new ArgumentNullException(nameof(customerRepository));

        \_unitOfWork = unitOfWork ?? throw new ArgumentNullException(nameof(unitOfWork));

    }

    // Método que se encarga de manejar la solicitud

    public async Task<ErrorOr<Unit>> Handle(UpdateCustomerCommand command, CancellationToken cancellationToken)

    {

        // Se valida que el id no sea nulo

        if (await \_customerRepository.GetByIdAsync(new CustomerId(command.Id)) is not Customer customer)

        {

            return Error.NotFound("Customer.NotFound", "The customer with the provide Id was not found.");

        }

        // Se actualiza el cliente  con los datos proporcionados en la solicitud y se almacena en la variable x de tipo Customer

        Customer x = Customer.Update(

              new CustomerId(command.Id),

              command.Name,

              command.LastName,

              command.Email,

              PhoneNumber.Create(command.PhoneNumber) ?? throw new Exception("Phone number is required. " + nameof(PhoneNumber)),

              Address.Create(command.Street, command.City, command.State, command.Country, command.ZipCode) ?? throw new Exception("Address is required. " + nameof(Address)

              )

          );

        \_customerRepository.Update(x); // Se actualiza el cliente en la base de datos

        await \_unitOfWork.SaveChangesAsync(cancellationToken); // Se guardan los cambios en la base de datos

        return Unit.Value; // Se retorna un valor Unit

    }

}

**UpdateCustomerCommandValidator.cs**

using FluentValidation;

namespace Application.Customers.Update;

// Clase que contiene las reglas de validación para el comando de actualización de un cliente.

public class UpdateCustomerCommandValidator : AbstractValidator<UpdateCustomerCommand>

{

    public UpdateCustomerCommandValidator()

    {

        RuleFor(x => x.Id)

            .NotEmpty().WithMessage("Id is required.") // El campo no puede estar vacío.

            .NotEqual(Guid.Empty).WithMessage("Id must not be empty."); // El campo no puede ser un Guid vacío.

        // Reglas de validación para el comando de creación de un cliente.

        RuleFor(x => x.Name)

            .NotEmpty() // El campo no puede estar vacío.

            .MaximumLength(50) // Longitud máxima del campo.

            .WithName("Name: "); // Nombre del campo en el mensaje de error.

        RuleFor(x => x.LastName)

            .NotEmpty()

            .MaximumLength(50)

            .WithName("Last Name: ");

        RuleFor(x => x.Email)

            .NotEmpty()

            .EmailAddress() // El campo debe ser una dirección de correo electrónico válida.

            .MaximumLength(255);

        RuleFor(x => x.PhoneNumber)

            .NotEmpty()

            .MaximumLength(9)

            .WithName("Phone Number: ");

        RuleFor(x => x.Country)

            .NotEmpty()

            .MaximumLength(50);

        RuleFor(x => x.State)

            .NotEmpty()

            .MaximumLength(50);

        RuleFor(x => x.City)

            .NotEmpty()

            .MaximumLength(50);

        RuleFor(x => x.Street)

            .NotEmpty()

            .MaximumLength(50);

        RuleFor(x => x.ZipCode)

            .NotEmpty()

            .MaximumLength(10)

            .WithName("Zip Code: ");

    }

}