Arquitectura Microservicios NET 8

Introducción

En esta sección se presentará una visión general del proyecto, explicando brevemente el propósito del microservicio y la justificación de la elección de la arquitectura utilizada.

Vertical Slice Architecture

La Arquitectura de Slice Vertical es una alternativa a la arquitectura tradicional en capas y la monolítica. En lugar de dividir la aplicación por capas (como presentación, lógica de negocio, acceso a datos), se divide en slices verticales(Features) que contienen todas las capas necesarias para una funcionalidad específica.

Ventajas respecto a Arquitecturas Tradicionales

- **Cohesión Alta y Acoplamiento Bajo**: Cada slice es independiente y encapsula toda la lógica relacionada con una funcionalidad específica.
- **Facilidad de Mantenimiento y Escalabilidad**: Permite trabajar de manera más modular, facilitando la actualización y escalabilidad de partes individuales del sistema.
- **Mejor Alineación con el Dominio del Negocio**: Cada slice puede ser modelada de acuerdo con las necesidades del negocio, mejorando la claridad y la coherencia del código.

Arquitectura Tradicional : Te obliga a seguir muchas reglas y muchas abstracciones por poco beneficio. Uno de los objetivos del *Clean Architecture* es eliminar la dependencia del Core de cualquier framework y persistencia.

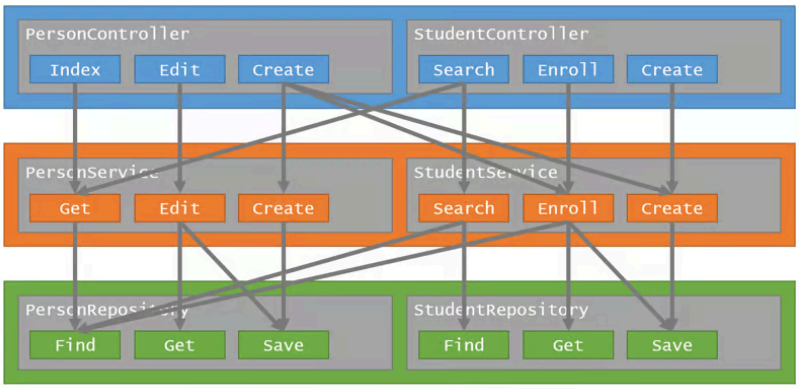
En temas de mantenibilidad, si queremos cambiar algo en la funcionalidad (o agregar nueva) tenemos que editar/agregar archivos, abstracciones(interfaces) las implementaciones de estas, y todo el ciclo de dependencias ya qué el grado de cohesión qué existe entre los miembros de la clase y los componentes de software es alta.

```
public interface IPersonRepository : IRepository<Person>
{
    Person GetByPersonId(string personId);
    Person GetByPersonIdForFoodProcessing(string personId);
    Person GetByPersonIdForCoffeeProcessing(string personId);
    Person GetByPersonIdForSpendMoreGetMoreProcessing(string personId);
    Person GetByPersonIdWithNoEagerFetch(string personId);
    Person GetByPersonIdForBirthdayRewardProcessing(string personId);

    Person[] GetByPersonIdOrEmail(string personId, string email);

    IEnumerable<PersonWithThermbarUpdates> GetUpdatedPeople();

    void RemoveUpdatedFlagFromAllPeople();
    Person FindByEmail(string email);
    Person[] GetPeopleWithPersonIdInATempTable(string tempTableName);
    Person GetByIdFetchSpendEntries(Guid id);
    Person GetByIdFetchBenefits(Guid id);
    PersonWithExpiredFoodEntry[] GetPeopleWithExpiredFoodEntriesIn(DateTime date);
    PersonWithExpiredCoffeeEntry[] GetPeopleWithExpiredCoffeeEntriesIn(DateTime date);
    PersonWithExpiredSpendMoreGetMoreEntry[] GetPeopleWithExpiredSpendMoreGetMoreEntriesIn(DateTime date);
    Boolean DoesPersonExist(string personId);
    Person[] FindAllMatchingEmail(string email);
    IEnumerable<Person> GetByEmailOfPersonId(string personId);
```

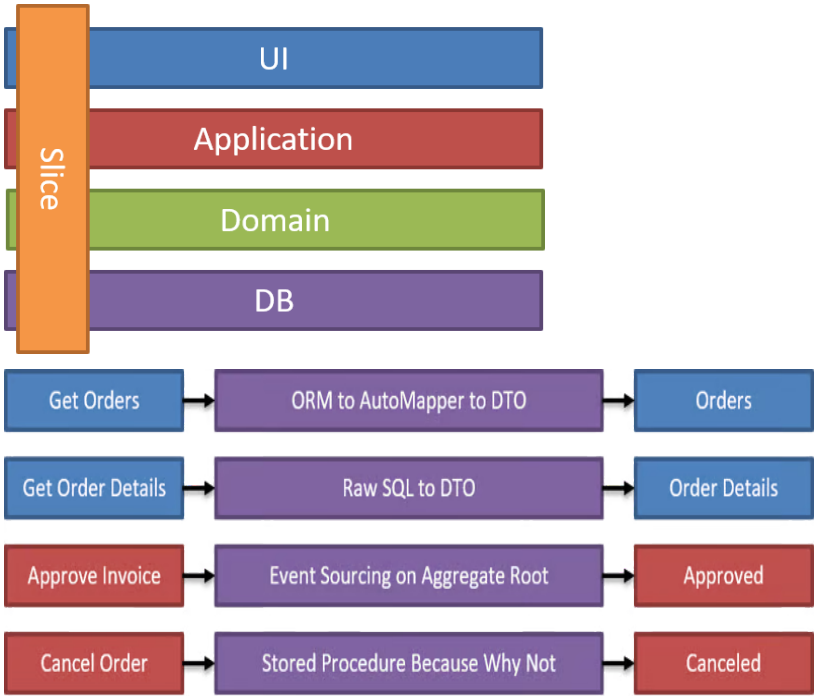


Vertical Slice Architecture:
La arquitectura vertical slice propone dividir todo por cuestiones técnicas, dividir todo por funcionalidad.

Dividir un sistema por funcionalidad tiene el beneficio de que cada Request tiene un caso de uso distinto, el cual tiene sus propias necesidades y se abordan como se necesite.

En esta arquitectura, el código es implementado alrededor de distintos requests (operaciones que cambian o no el estado del dominio), encapsulando y agrupando todos los "concerns" desde el UI/Presentation hasta el back. Es decir, tomas un modelo "n-tier" o clean architecture y eliminas las "puertas" entre esas capas y las acoplas según la funcionalidad.

En lugar de acoplar las capas horizontalmente, es mejor acoplar de forma vertical (según un slice/feature) y evitar acoplar entre slices y más bien que un slice esté fuertemente acoplado consigo mismo.



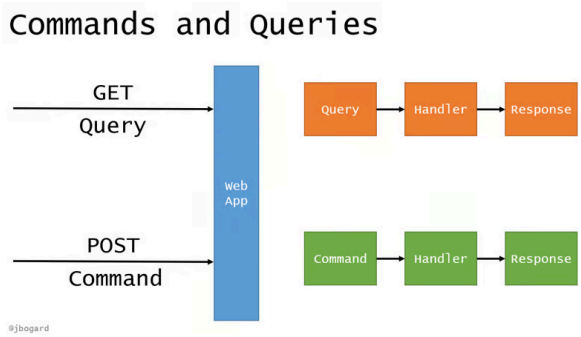
Con este enfoque, la mayoría de las abstracciones desaparecen y no necesitamos ningún tipo de abstracción de capas "compartidas" como los repositorios, servicios y controladores. A veces, nuestras herramientas aún lo requieren, pero mantenemos nuestro intercambio de lógica vertical al mínimo.

Cada "vertical slice" puede decidir por sí misma como cumple con las necesidades del Request.

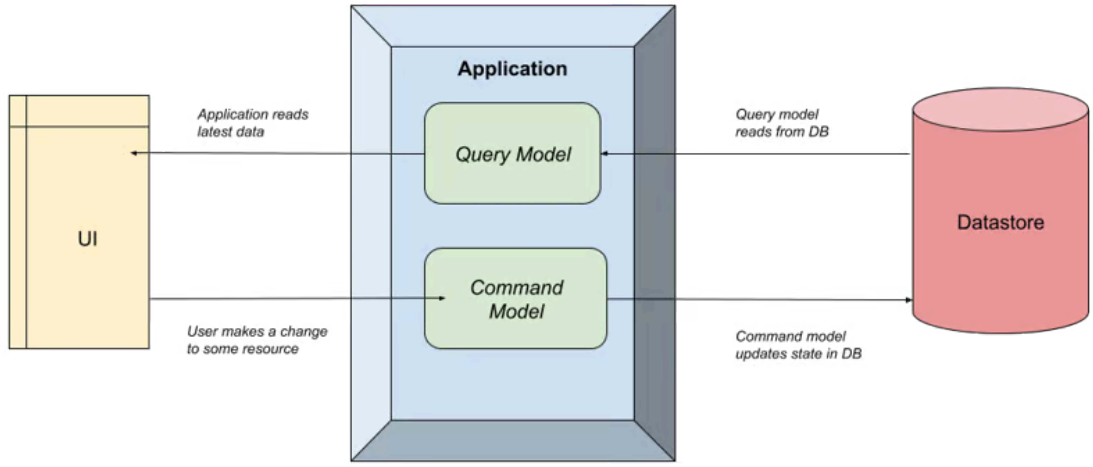
CQRS (Command Query Responsibility Segregation)

CQRS es un patrón que separa las operaciones de lectura (Query) y escritura (Command) en una aplicación. Esta separación permite optimizar cada tipo de operación de manera independiente, mejorando la eficiencia y escalabilidad del sistema.

- Comandos (Commands): Estos son responsables de modificar el estado del dominio.
- Consultas (Queries): Se encargan de obtener información del estado del dominio



La siguiente imagen ilustra cómo funciona esta separación de responsabilidades:



Problemas qué intenta resolver 👍

Problemas del diseño en "n-capas":

1. **Mantenimiento complejo**: Los grandes repositorios y servicios que manejan todas las operaciones de una entidad se vuelven difíciles de mantener y modificar.
2. **Curva de aprendizaje empinada**: Nuevos desarrolladores tienen dificultades para realizar cambios sin temor a romper algo debido al fuerte acoplamiento del código.
3. **Dificultad en pruebas**: Las dependencias y mocks necesarios para probar funciones específicas complican las pruebas.

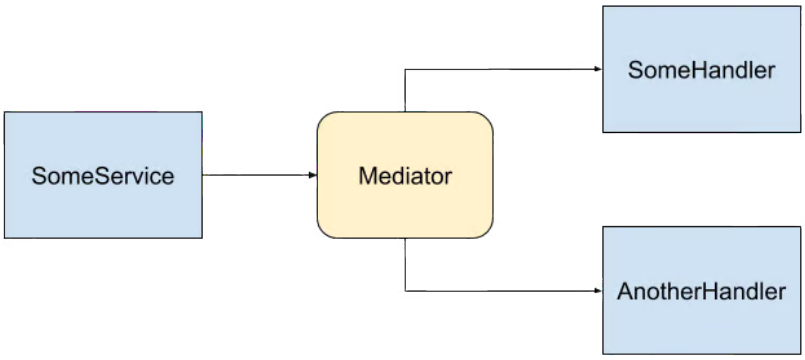
Propuesta de solución:

1. **Segregación de responsabilidades**: Dividir el código en Queries (consultas) y Commands (comandos) para mantenerlo organizado y modular.
2. **Encapsulamiento**: Cada Command se encapsula con sus propias dependencias, facilitando su modificación sin afectar otros componentes.

Cómo podemos darnos cuenta Vertical Slice nos obliga a implementar CQRS

Patrón Mediador

Define un objeto que gestiona la interacción entre otros objetos, desacoplando directamente a estos últimos.Simplifica el desarrollo, mantenimiento y pruebas al reducir las dependencias directas entre componentes.



Nos permite desacoplar por completo componentes, pero aún así permite que interactúen entre sí. Cuanto menos tenga que preocuparse un componente para funcionar, más sencillo será desarrollarlo, mantenerlo y probarlo.

MediatR:

Implementa el patrón Mediador dentro del mismo proceso de la aplicación, facilitando la creación de sistemas basados en CQRS (Command Query Responsibility Segregation).

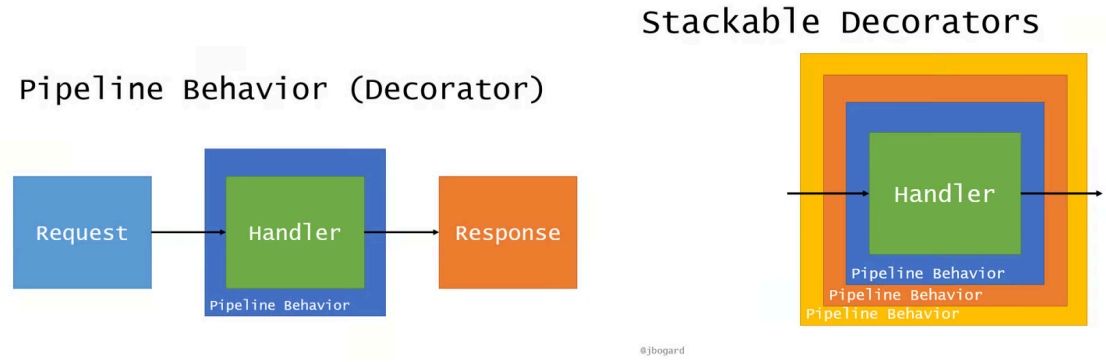
Adecuado para manejar toda la comunicación interna de la aplicación

¿Por qué hacer esto? Hay varias razones, pero una de las más importantes es asegurarse de que todo el procesamiento de las solicitudes en la API no dependa directamente de los controladores. En su lugar, delegar esta responsabilidad a una capa en la "Aplicación Central" (siguiendo los principios de Clean Architecture o Vertical Slices).

Patron decorador(Pipeline Behaviour)

Los Behaviors de MediatR funcionan de manera similar a cómo lo hace un Middleware en ASP.NET. Ejecutan alguna acción y luego delegan la ejecución al siguiente elemento en la cadena (esperando una respuesta).

Podemos utilizar decoradores para una variedad de tareas, y MediatR nos permite configurar los que necesitemos sin problemas.



Capas del Proyecto

Application Core

- **Descripción**: Agrupa el dominio, núcleo, persistencia e infraestructura.
- **Componentes Principales**:
    - **Behaviours**: Decoradores con MediatR para reglas de negocio.
    - **Exceptions**: Excepciones personalizadas.
    - **Helpers (Utils)**: Utilidades como Hash IDs.
    - **Dominio**: Entidades de dominio, objetos de valor, enums, servicios de dominio.
    - **Features**: Segmentos funcionales.
    - **Infrastructure**: Adaptadores y servicios externos.
    - **Persistence**: Acceso a datos : Entity Framework Core, Dapper, ADO.NET

WebApi

- **Descripción**: Esta capa expone las funcionalidades del microservicio a través de una API RESTful.
- **Componentes Principales**:
    - **Controladores**: Gestionan las solicitudes HTTP y las dirigen a los servicios correspondientes en la capa Application Core.
    - **Filtros**: Se aplican a los controladores, por lo tanto, están estrechamente relacionados con la presentación.

Implementación de CQRS y MediatR

CQRS (Command Query Responsibility Segregation)

- **Concepto**: Separa las operaciones de lectura (Query) de las de escritura (Command) para mejorar la claridad y escalabilidad.
- **Ventajas**:
    - Permite optimizar y escalar de manera independiente las operaciones de lectura y escritura.
    - Mejora la organización y separación de responsabilidades en el código.

MediatR

- **Concepto**: Un patrón mediador que facilita la comunicación entre diferentes componentes de la aplicación sin necesidad de referencias directas.
- **Ventajas**:
    - Reduce el acoplamiento entre componentes.
    - Facilita la implementación de CQRS al manejar comandos y consultas de manera centralizada.

Validación con FluentValidation

- **Descripción**: Herramienta utilizada para implementar validaciones de manera fluida y sencilla.
- **Ventajas**:
    - Mejora la legibilidad y mantenibilidad del código de validación.
    - Permite definir reglas de validación de manera declarativa.

Mapeo de Objetos con AutoMapper

- **Descripción**: Biblioteca utilizada para mapear objetos de un tipo a otro, facilitando la conversión entre entidades de dominio y DTOs.
- **Ventajas**:
    - Reduce el código boilerplate de mapeo manual.
    - Asegura que las transformaciones de datos sean consistentes y fáciles de mantener.

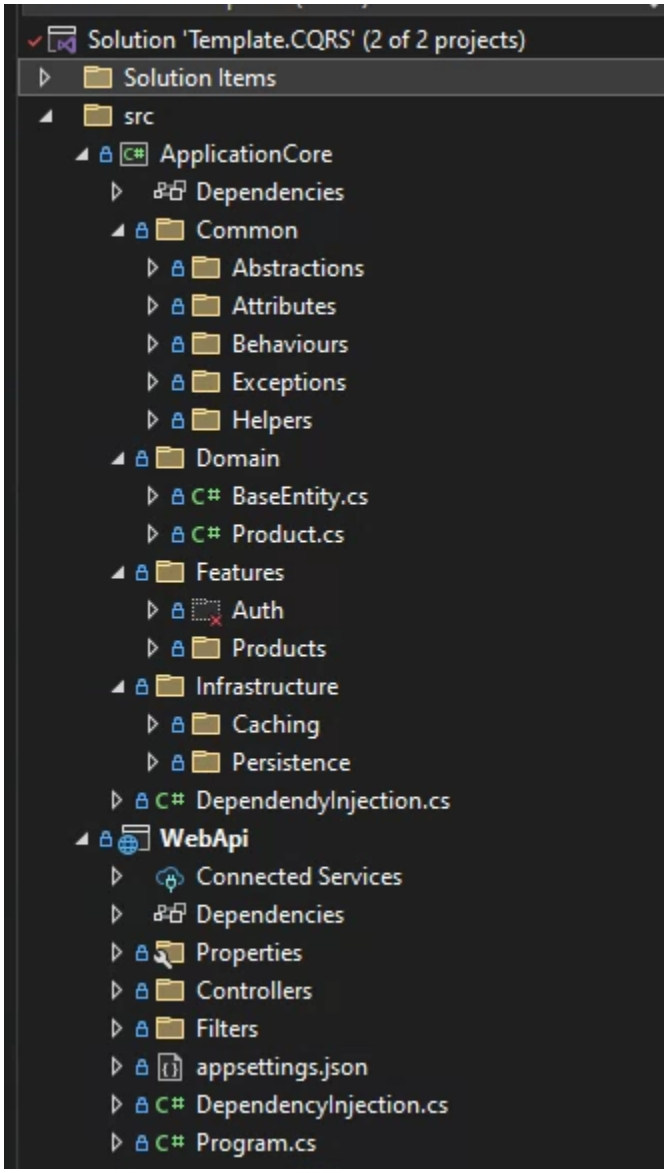
Seguridad con ASP.NET Identity y JWT

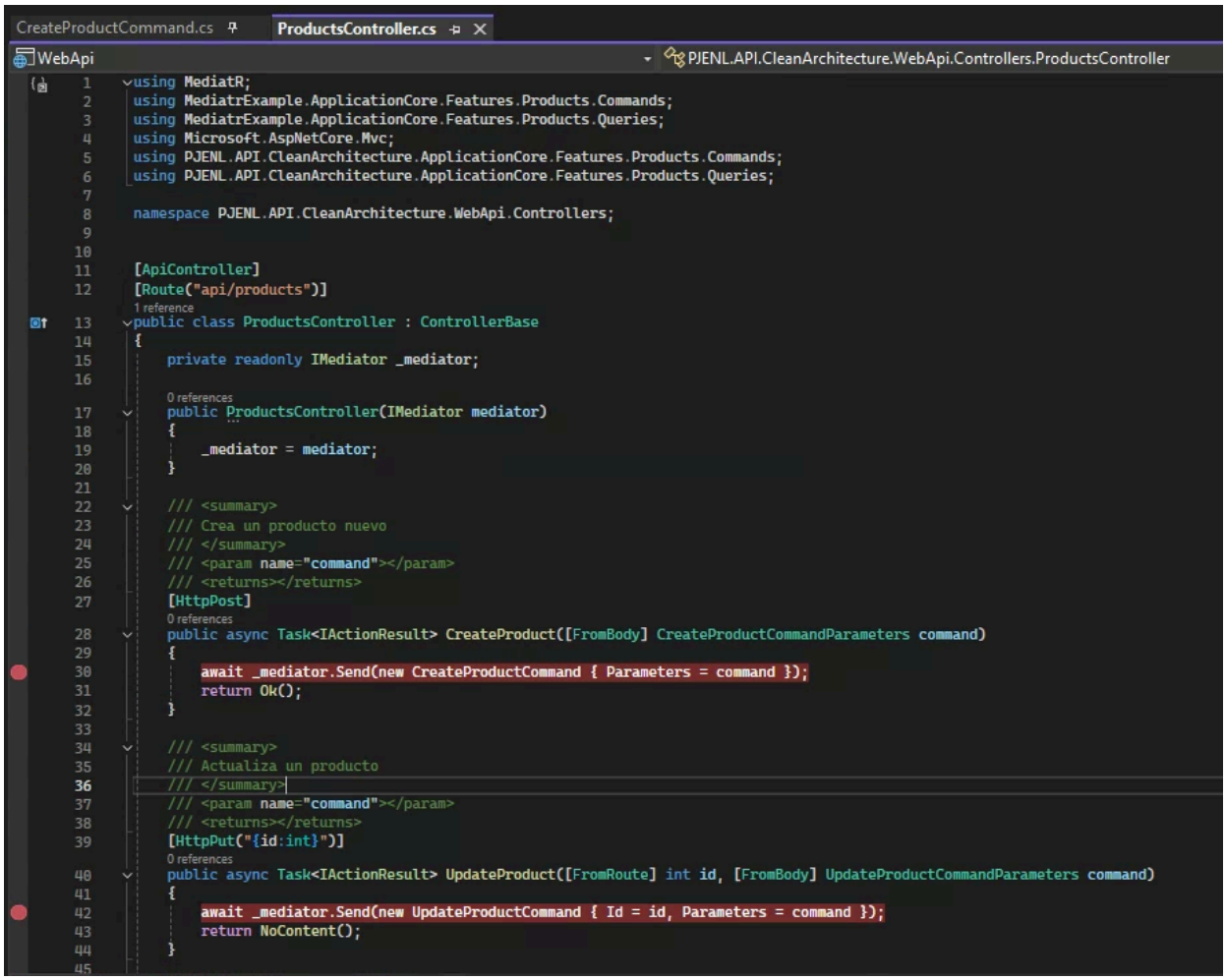
- **Descripción**: Implementación de autenticación y autorización utilizando ASP.NET Identity para gestionar usuarios y JWT (JSON Web Tokens) para la autenticación basada en tokens.
- **Ventajas**:
    - Mejora la seguridad de la aplicación.
    - Permite la autenticación sin estado (stateless), ideal para microservicios.

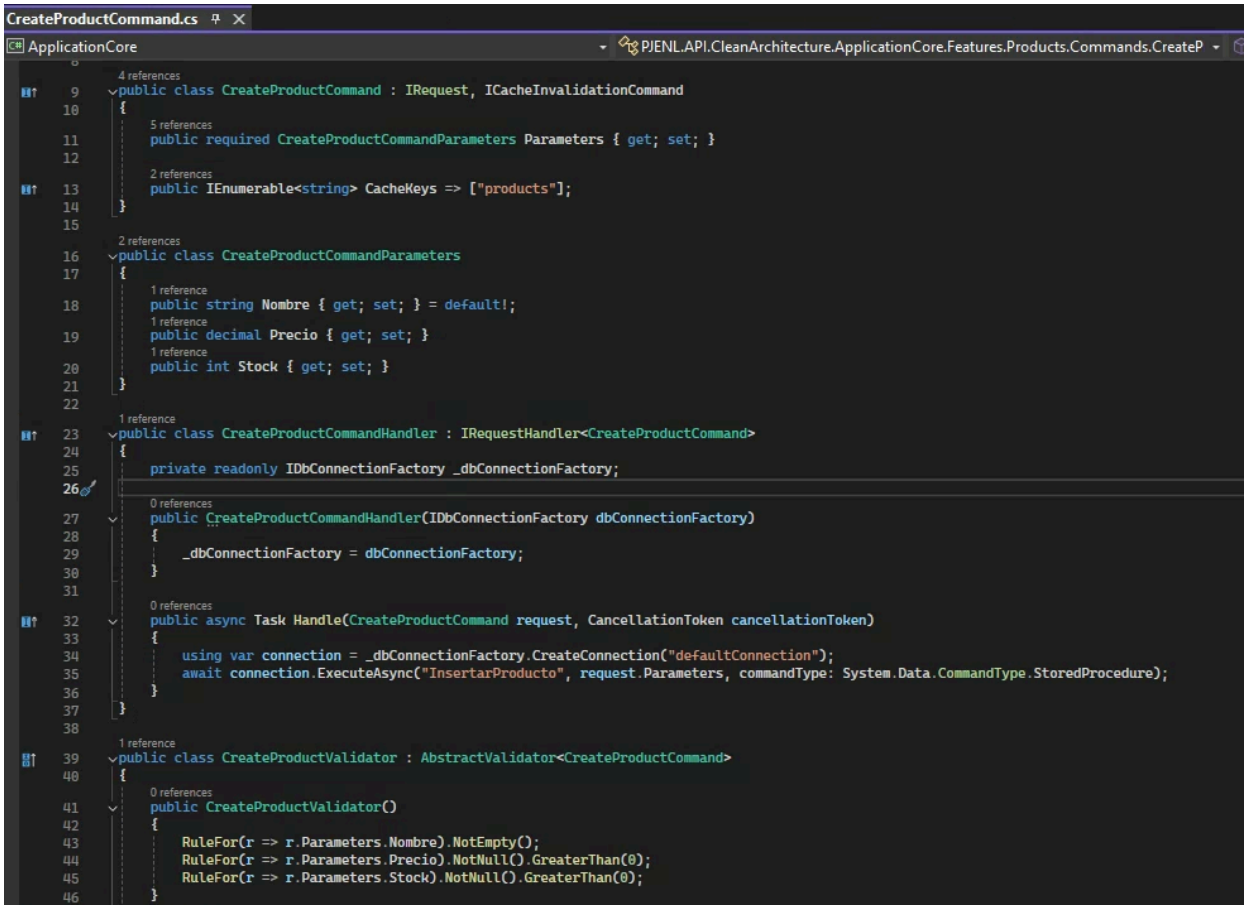
URLs Seguros con Hashids

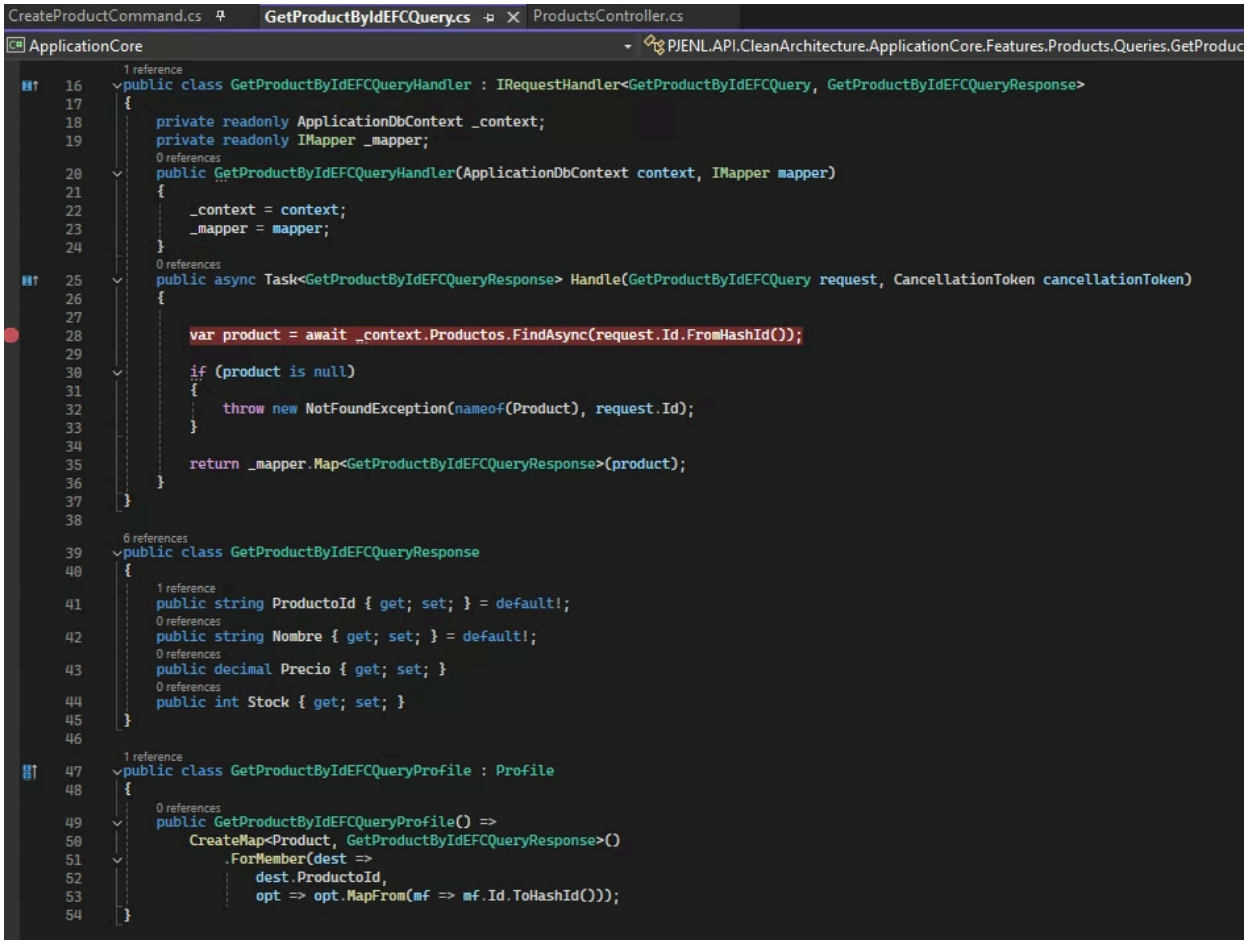
- **Descripción**: Uso de Hashids para generar identificadores únicos y seguros en las URLs.
- **Ventajas**:
    - Aumenta la seguridad al evitar la exposición de identificadores predecibles.
    - Mejora la estética y la usabilidad de las URLs.

Solución


Solution 'Template.CQRS' (2 of 2 projects)
Solution Items
src
ApplicationCore
Dependencies
Common
Abstractions
Attributes
Behaviours
Exceptions
Helpers
Domain
BaseEntity.cs
Product.cs
Features
Auth
Products
Infrastructure
Caching
Persistence
DependendyInjection.cs
WebApi
Connected Services
Dependencies
Properties
Controllers
Filters
appsettings.json
DependencyInjection.cs
Program.cs


CreateProductCommand.cs
ProductsController.cs
WebApi
PJENL.API.CleanArchitecture.WebApi.Controllers.ProductsController
using MediatR;
using MediatrExample.ApplicationCore.Features.Products.Commands;
using MediatrExample.ApplicationCore.Features.Products.Queries;
using Microsoft.AspNetCore.Mvc;
using PJENL.API.CleanArchitecture.ApplicationCore.Features.Products.Commands;
using PJENL.API.CleanArchitecture.ApplicationCore.Features.Products.Queries;
namespace PJENL.API.CleanArchitecture.WebApi.Controllers;
[ApiController]
[Route("api/products")]
1 reference
public class ProductsController : ControllerBase
{
private readonly IMediator _mediator;
0 references
public ProductsController(IMediator mediator)
{
_mediator = mediator;
}
/// <summary>
/// Crea un producto nuevo
/// </summary>
/// <param name="command"></param>
/// <returns></returns>
[HttpPost]
0 references
public async Task<IActionResult> CreateProduct([FromBody] CreateProductCommandParameters command)
{
await _mediator.Send(new CreateProductCommand { Parameters = command });
return Ok();
}
/// <summary>
/// Actualiza un producto
/// </summary>
/// <param name="command"></param>
/// <returns></returns>
[HttpPut("{id:int}")]
0 references
public async Task<IActionResult> UpdateProduct([FromRoute] int id, [FromBody] UpdateProductCommandParameters command)
{
await _mediator.Send(new UpdateProductCommand { Id = id, Parameters = command });
return NoContent();
}


CreateProductCommand.cs
ApplicationCore
PJENL.API.CleanArchitecture.ApplicationCore.Features.Products.Commands.CreateP
4 references
public class CreateProductCommand : IRequest, ICacheInvalidationCommand
{
5 references
public required CreateProductCommandParameters Parameters { get; set; }
2 references
public IEnumerable<string> CacheKeys => ["products"];
}
2 references
public class CreateProductCommandParameters
{
1 reference
public string Nombre { get; set; } = default!;
1 reference
public decimal Precio { get; set; }
1 reference
public int Stock { get; set; }
}
1 reference
public class CreateProductCommandHandler : IRequestHandler<CreateProductCommand>
{
private readonly IDbConnectionFactory _dbConnectionFactory;
0 references
public CreateProductCommandHandler(IDbConnectionFactory dbConnectionFactory)
{
_dbConnectionFactory = dbConnectionFactory;
}
0 references
public async Task Handle(CreateProductCommand request, CancellationToken cancellationToken)
{
using var connection = _dbConnectionFactory.CreateConnection("defaultConnection");
await connection.ExecuteAsync("InsertarProducto", request.Parameters, commandType: System.Data.CommandType.StoredProcedure);
}
}
1 reference
public class CreateProductValidator : AbstractValidator<CreateProductCommand>
{
0 references
public CreateProductValidator()
{
RuleFor(r => r.Parameters.Nombre).NotEmpty();
RuleFor(r => r.Parameters.Precio).NotNull().GreaterThan(0);
RuleFor(r => r.Parameters.Stock).NotNull().GreaterThan(0);
}


CreateProductCommand.cs
GetProductByIdEFCQuery.cs
ProductsController.cs
ApplicationCore
PJENL.API.CleanArchitecture.ApplicationCore.Features.Products.Queries.GetProduc
1 reference
public class GetProductByIdEFCQueryHandler : IRequestHandler<GetProductByIdEFCQuery, GetProductByIdEFCQueryResponse>
{
private readonly ApplicationDbContext _context;
private readonly IMapper _mapper;
0 references
public GetProductByIdEFCQueryHandler(ApplicationDbContext context, IMapper mapper)
{
_context = context;
_mapper = mapper;
}
0 references
public async Task<GetProductByIdEFCQueryResponse> Handle(GetProductByIdEFCQuery request, CancellationToken cancellationToken)
{
var product = await _context.Productos.FindAsync(request.Id.FromHashId());
if (product is null)
{
throw new NotFoundException(nameof(Product), request.Id);
}
return _mapper.Map<GetProductByIdEFCQueryResponse>(product);
}
}
6 references
public class GetProductByIdEFCQueryResponse
{
1 reference
public string ProductoId { get; set; } = default!;
0 references
public string Nombre { get; set; } = default!;
0 references
public decimal Precio { get; set; }
0 references
public int Stock { get; set; }
}
1 reference
public class GetProductByIdEFCQueryProfile : Profile
{
0 references
public GetProductByIdEFCQueryProfile() =>
CreateMap<Product, GetProductByIdEFCQueryResponse>()
.ForMember(dest =>
dest.ProductoId,
opt => opt.MapFrom(mf => mf.Id.ToHashId()));
}