**Introducción**

En este tutorial, primero entendimos cómo funcionan CQRS y MediatR y luego creamos una operación CRUD completa en la aplicación ASP.NET Core. También creamos Notificaciones (Notifications) y comportamientos (Behaviors) de MediatR.

**Objetivo**

Vamos a usar dos patrones de diseño: CQRS y el patrón Mediator.

**¿Qué es el patrón CQRS?**

CQRS son las siglas de Command Query Responsibility Segregation, este patrón establece que se separan las operaciones de lectura y escritura para una fuente de datos. Cada método debe ser un comando o una consulta, pero no ambos. En CQRS los "Comandos" son conocidos por guardar en la base de datos y las "Consultas" por leer de la base de datos.

La idea detrás de CQRS es tener diferentes modelos para diferentes operaciones. Entonces para una operación CRUD tendremos 4 modelos que son:

1. Un modelo para crear registros.
2. Un modelo para leer registros.
3. Un modelo de actualización de registros.
4. Un modelo para borrar registros.

Vea la imagen de abajo donde he ilustrado la arquitectura CQRS.

Diagrama

Descripción generada automáticamente

**¿Qué es el patrón de mediador?**

El patrón Mediator define un objeto llamado Mediator, este Mediator encapsula cómo un conjunto de objetos interactúa entre sí. Mediator promueve el acoplamiento flexible al evitar que los objetos se refieran entre sí de forma explícita y le permite variar su interacción de forma independiente.

El patrón Mediator garantiza que los componentes no se llamen entre sí de forma explícita, sino que lo hagan mediante llamadas a un mediador.

La siguiente imagen ilustra el patrón de Mediador:

Diagrama

Descripción generada automáticamente

**Ventajas del patrón CQRS y mediador**

Algunas ventajas importantes del uso de patrones CQRS y Mediator son:

**1. Diseño optimizado y escalable**

No tenemos modelos complejos, sino modelos separados por operación de datos. Esto proporciona diseños altamente optimizados. También podemos escalar nuestra aplicación fácilmente sin tener que preocuparnos por lidiar con las complejidades.

**2. Rendimiento más rápido**

Las operaciones de la base de datos se vuelven más rápidas mediante el uso de CQRS y Mediator. También proporciona mejores operaciones en paralelo ya que hay "Comandos" y "Consultas".

**3. Arquitectura débilmente acoplada**

El patrón Mediador proporciona una arquitectura débilmente acoplada a nuestra aplicación, es ligera, con una sola responsabilidad, sin muchas dependencias, lo que permite a los equipos trabajar de forma independiente, implementar de forma independiente, fallar y escalar de forma independiente y aumentar la capacidad de respuesta del negocio.

**Usando la biblioteca MediatR (implementación)**

MediatR es una biblioteca .NET que ayuda a crear aplicaciones basadas en patrones CQRS y Mediator. Toda la comunicación entre la interfaz de usuario y el almacén de datos se realiza a través de MediatR. La biblioteca MediatR está disponible en NuGet y en breve la instalaremos en nuestra aplicación.

**Configuración del proyecto de API web ASP.NET Core con el paquete MediatR**

Lo primero que debe hacer es crear una nueva aplicación ASP.NET Core Web API en Visual Studio.

Interfaz de usuario gráfica, Texto, Aplicación, Correo electrónico

Descripción generada automáticamente

Nombramos, por ejemplo, la aplicación como CQRSMediator.

A continuación, seleccione Herramientas ➤ Administrador de paquetes NuGet ➤ Administrar paquetes NuGet para la solución en su Visual Studio e instale los siguientes 2 paquetes:

* MediatR
* MediatR.Extensions.Microsoft.DependencyInjection

Interfaz de usuario gráfica, Texto, Aplicación, Correo electrónico

Descripción generada automáticamente

Ahora abrimos la clase Startup.cs y añadimos el espacio de nombres MediaR.

using MediatR;

|  |  |
| --- | --- |
| 1  2  3  4  5 | public void ConfigureServices(IServiceCollection services)  {      services.AddMediatR(typeof(Startup));      services.AddControllers();  } |

A continuación, modifiquemos el método ConfigureServices para configurar MediatR:

Ahora MediatR está configurado y listo para funcionar.

**Configuraciones principales de Entity Framework**

Ahora instalaremos y configuraremos Entity Framework Core en nuestra aplicación. Entity Framework Core trabajará con CQRS y Mediator para realizar operaciones de base de datos. Así que primero instale los siguientes 2 paquetes en su aplicación.

* Microsoft.EntityFrameworkCore.SqlServer
* Microsoft.EntityFrameworkCore.Tools

Interfaz de usuario gráfica, Texto, Aplicación, Correo electrónico

Descripción generada automáticamente

A continuación, vaya al archivo appsettings.json y agregue la cadena de conexión para la base de datos. Estoy usando la conexión LocalDb.

{

"ConnectionStrings": {

"DefaultConnection": "Server=(localdb)\\mssqllocaldb;Database=ProductDB;Trusted\_Connection=True;MultipleActiveResultSets=true"

}

}

A continuación, agregue la clase Product.cs dentro de la carpeta Modelos. Esta clase sirve como modelo. Realizaré operaciones CRUD en registros de productos. El código de la clase Product.cs se proporciona a continuación:

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14 | using System;  using System.Collections.Generic;  using System.Linq;  using System.Threading.Tasks;    namespace CQRSMediator.Models  {      public class Product      {          public int Id { get; set; }          public string Name { get; set; }          public decimal Price { get; set; }      }  } |

A continuación, debemos agregar el archivo DbContext para Entity Framework Core. Entonces, agregue una nueva clase llamada ProductContext.cs dentro de la misma carpeta "Modelos" con el siguiente código.

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17 | using Microsoft.EntityFrameworkCore;  using System;  using System.Collections.Generic;  using System.Linq;  using System.Threading.Tasks;    namespace CQRSMediator.Models  {      public class ProductContext: DbContext      {          public ProductContext(DbContextOptions<ProductContext> options) : base(options)          {          }            public DbSet<Product> Product { get; set; }      }  } |

A continuación, debemos agregar Entity Framework Core a IServiceCollection. Esto se hace dentro del método ConfigureServices () de la clase Startup.cs.

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7 | public void ConfigureServices(IServiceCollection services)  {      services.AddMediatR(typeof(Startup));      services.AddDbContext<ProductContext>(options =>          options.UseSqlServer(Configuration.GetConnectionString("DefaultConnection")));      services.AddControllers();  } |

El código anterior obtiene la Cadena de conexión que se denomina DefaultConnection. Recuerde que hemos definido esta conexión en el archivo appsettings.json.

Lo último es realizar migraciones de Entity Framework Core. Esto se hace ejecutando los siguientes 2 comandos que se dan a continuación en la ventana Herramientas ➤ NuGet Package Manager ➤ Packages Manager Console de Visual Studio.

dotnet ef migrations add Migration1

dotnet ef database update

Una vez que se complete la migración, abra la ventana Ver ➤ Explorador de objetos de SQL Server. Luego, haga clic derecho en (localdb) \ MSSQLLocalDB y seleccione "Actualizar". Ahora verá que se crea la base de datos ProductDB.

Interfaz de usuario gráfica, Texto, Aplicación, Correo electrónico

Descripción generada automáticamente

Esto completa la configuración de Entity Framework Core. Ahora estamos listos para comenzar a construir nuestras operaciones CRUD para los Productos en el Controlador.

**Creación de un producto con CQRS**

Ahora crearé el comando CQRS que creará un nuevo producto. Entonces, cree la carpeta CQRS ➤ Commands en la carpeta raíz de la aplicación. Luego agregue una nueva clase llamada CreateProductCommand.cs a la carpeta Commands. Vea la imagen de abajo donde he mostrado esta cosa.

Interfaz de usuario gráfica, Texto, Aplicación, Correo electrónico

Descripción generada automáticamente

Ya discutimos los comandos y consultas de CQRS. Para separar estos dos, necesitamos crear 2 carpetas separadas, la primera carpeta " Commands " ya está creada dentro de la carpeta CQRS. La carpeta Commands contendrá clases que se ocuparán de los guardados de la base de datos. Ya agregamos una de esas clases CreateProductCommand.cs.

Ahora también agregue la carpeta "Queries" dentro de la carpeta "CQRS". Esta carpeta contendrá clases que se ocuparán de las lecturas de la base de datos. Agregaremos estas clases durante el transcurso de este tutorial.

Ahora agregue el siguiente código a la clase CreateProductCommand.cs.

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17  18  19  20  21  22  23  24  25  26  27  28  29  30  31  32  33  34  35 | using CQRSMediator.Models;  using MediatR;  using System;  using System.Collections.Generic;  using System.Text;  using System.Threading;  using System.Threading.Tasks;    namespace CQRSMediator.CQRS.Commands  {      public class CreateProductCommand : IRequest<int>      {          public string Name { get; set; }          public decimal Price { get; set; }            public class CreateProductCommandHandler : IRequestHandler<CreateProductCommand, int>          {              private ProductContext context;              public CreateProductCommandHandler(ProductContext context)              {                  this.context = context;              }              public async Task<int> Handle(CreateProductCommand command, CancellationToken cancellationToken)              {                  var product = new Product();                  product.Name = command.Name;                  product.Price = command.Price;                    context.Product.Add(product);                  await context.SaveChangesAsync();                  return product.Id;              }          }      }  } |

Vamos a explicar las cosas:

* Nuestra clase CreateProductCommand.cs está implementando la interfaz IRequest <int> de la biblioteca MediatR. El tipo es int, lo que significa que nuestra clase devolverá un valor de tipo int. Observe el método Handle donde estoy devolviendo el valor product.Id que es de tipo int.
* CreateProductCommand.csclass tiene 2 propiedades llamadas Nombre y Precio. Estos recibirán los valores del producto del controlador y luego se creará un nuevo producto (con estos valores).
* A continuación, creamos una clase interna que será la clase Handler y la nombramos CreateProductCommandHandler. Hereda de la clase IRequestHandler <CreateProductCommand, int>. Esta clase de controlador manejará la solicitud de comando CQRS. El IRequestHandler es una interfaz de la biblioteca MediatR, su primera firma contiene la clase que se ocupa de la operación CQRS, es decir, CreateProductCommand, mientras que la segunda firma contiene el tipo de retorno que obviamente es int.
* Luego implementamos el método Handle cuyo trabajo es crear, es decir, guardar un producto en la base de datos.

Ahora cree el controlador desde donde se realiza la solicitud http para crear el producto. Entonces, dentro de la carpeta "Controladores", cree un nuevo controlador con el nombre de ProductController. Para el constructor de este controlador, haga una dependencia para el tipo de IMediator.

Luego agregamos un nuevo método Http Post Type llamado "Create" que hará una llamada a la clase de comando CQRS CreateProductCommand que acabamos de hacer.

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17  18  19  20  21  22  23  24  25  26  27 | using CQRSMediator.CQRS.Commands;  using MediatR;  using Microsoft.AspNetCore.Mvc;  using System;  using System.Collections.Generic;  using System.Linq;  using System.Threading.Tasks;    namespace CQRSMediator.Controllers  {      [Route("api/[controller]")]      [ApiController]      public class ProductController : ControllerBase      {          private IMediator mediator;          public ProductController(IMediator mediator)          {              this.mediator = mediator;          }            [HttpPost]          public async Task<IActionResult> Create(CreateProductCommand command)          {              return Ok(await mediator.Send(command));          }      }  } |

Tenga en cuenta que el método Create tiene el parámetro de tipo CreateProductCommand y es responsabilidad del patrón Mediator (biblioteca MediatR) realizar una llamada a esta clase cuando se ejecuta el método mediator.Send ().

**Prueba del comando CQRS**

Probemos ahora si el CQRS está funcionando correctamente. Por lo tanto, agregue un punto de interrupción en el método Handle de la clase CreateProductCommand.cs. Ahora ejecute la aplicación en Visual Studio. Ahora abra Postman para realizar una solicitud de tipo HTTP POST a la aplicación. También puedes usar Swagger en lugar de Postman.

En el cartero, seleccione:

1. Opción "POST" del menú desplegable e ingrese URL - https: // localhost: 44378 / api / Product El puerto en su caso será diferente.
2. Seleccione la opción "Cuerpo" y luego seleccione la opción "sin procesar".
3. En el menú desplegable del lado derecho, seleccione "JSON" como opción.
4. En el cuadro de texto grande, ingrese el json del producto:

{

"name": "Shirts",

"price": 49

}

Haga clic en el botón enviar para realizar la solicitud. Vea la imagen de abajo donde he mostrado esto. Su punto de interrupción llegará, verifique el valor del parámetro "comando", verá los valores que completó en el cartero para el producto que ha recibido allí. Verifique la siguiente imagen de punto de interrupción:

Interfaz de usuario gráfica, Texto, Aplicación, Correo electrónico

Descripción generada automáticamente

El método devuelve la identificación del producto que se crea en la base de datos. Esta identificación será 1 y, a medida que continúe creando nuevos productos, la identificación se convertirá en 2, 3,…. etcétera.

Postman mostrará esta identificación dentro del campo Cuerpo de respuesta.

Felicidades, acabamos de crear nuestro primer comando con patrones CQRS y Mediator. A continuación, crearemos las operaciones restantes: Leer, Actualizar y Eliminar.

**Leer operaciones de productos con CQRS**

Habrá 2 operaciones de lectura: una para leer todos los productos al mismo tiempo y otra para leer los productos por id.

Comencemos leyendo todos los productos

Cree la carpeta Consultas dentro de la carpeta "CQRS". Las clases que leerán la base de datos se mantendrán dentro de esta carpeta de Consultas.

Entonces, cree una nueva clase llamada GetAllProductQuery.cs en la carpeta Consultas. El código completo de esta clase se proporciona a continuación:

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17  18  19  20  21  22  23  24  25  26  27  28 | using CQRSMediator.Models;  using MediatR;  using Microsoft.EntityFrameworkCore;  using System;  using System.Collections.Generic;  using System.Linq;  using System.Threading;  using System.Threading.Tasks;    namespace CQRSMediator.CQRS.Queries  {      public class GetAllProductQuery : IRequest<IEnumerable<Product>>      {          public class GetAllProductQueryHandler : IRequestHandler<GetAllProductQuery, IEnumerable<Product>>          {              private ProductContext context;              public GetAllProductQueryHandler(ProductContext context)              {                  this.context = context;              }              public async Task<IEnumerable<Product>> Handle(GetAllProductQuery query, CancellationToken cancellationToken)              {                  var productList = await context.Product.ToListAsync();                  return productList;              }          }      }  } |

Expliquemos las cosas:

* La clase GetAllProductQuery hereda de IRequest <IEnumerable <Product>>. Tenga en cuenta que el tipo de IRequest es IEnumerable <Product>, lo que significa que esta clase devolverá una lista de productos. Consulte el método Handle que devuelve esta lista de productos.
* He definido una clase interna llamada GetAllProductQueryHandler que es la clase de controlador. Contiene el método Handle, que en realidad devolverá todos los productos de la base de datos.

A continuación, necesitamos agregar un método al controlador que manejará una solicitud GET de tipo HTTP y llamará a la clase Query. Entonces, agregue el método GetAll() al ProductController.cs y su código se proporciona a continuación.

[HttpGet]

public async Task<IActionResult> GetAll()

{

return Ok(await mediator.Send(new GetAllProductQuery()));

}

**Prueba de consulta CQRS**

Ahora abra Postman y realice una solicitud HTTP GET a la URL: https: // localhost: 44378 / api / Product. El puerto será diferente en tu caso.

El cartero recibirá todos los productos en json y se mostrarán en la sección del cuerpo de respuesta. Vea la imagen de abajo donde he mostrado esto.

Interfaz de usuario gráfica, Texto, Aplicación, Correo electrónico

Descripción generada automáticamente

Si tiene más de un producto en la base de datos, entonces el json contendrá sus detalles para, ejemplo, como se muestra a continuación:

{

"id": 1,

"name": "Shirts",

"price": 49.00

},

{

"id": 2,

"name": "Pants",

"price": 79.00

}

Leer un producto por su ID

Creemos otra clase de consulta cuyo trabajo será buscar un producto por su identificación. Así que cree una nueva clase llamada GetProductByIdQuery.cs dentro de la carpeta Consultas y agregue el siguiente código.

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17  18  19  20  21  22  23  24  25  26  27  28  29 | using CQRSMediator.Models;  using MediatR;  using Microsoft.EntityFrameworkCore;  using System;  using System.Collections.Generic;  using System.Linq;  using System.Threading;  using System.Threading.Tasks;    namespace CQRSMediator.CQRS.Queries  {      public class GetProductByIdQuery : IRequest<Product>      {          public int Id { get; set; }          public class GetProductByIdQueryHandler : IRequestHandler<GetProductByIdQuery, Product>          {              private ProductContext context;              public GetProductByIdQueryHandler(ProductContext context)              {                  this.context = context;              }              public async Task<Product> Handle(GetProductByIdQuery query, CancellationToken cancellationToken)              {                  var product = await context.Product.Where(a => a.Id == query.Id).FirstOrDefaultAsync();                  return product;              }          }      }  } |

El código de esta clase es muy similar al de la clase anterior. La clase GetProductByIdQuery hereda de IRequest <Product>. El tipo de IRequest es Producto ya que la clase solo devolverá un producto.

Ahora vea el método de la clase Hander GetProductByIdQueryHandler Handle (). Emplea la cláusula where de LINQ para buscar un producto cuya identificación se proporciona a través del controlador.

var product = await context.Product.Where(a => a.Id == query.Id).FirstOrDefaultAsync();

A continuación, vaya a ProductController.cs y agregue GetById () de tipo HTTP GET se llamará cuando se realice el tipo de solicitud HTTP GET a la URL: https: // localhost: 44378 / api / Product / {id}. Aquí reemplace {id} con la identificación del producto como:

https://localhost:44378/api/Product/1

https://localhost:44378/api/Product/2

https://localhost:44378/api/Product/3

El codigo del método es:

[HttpGet("{id}")]

public async Task<IActionResult> GetById(int id)

{

return Ok(await mediator.Send(new GetProductByIdQuery { Id = id }));

}

Prueba con Postman

Ahora abra Postman y envíe la solicitud HTTP GET a la URL: https: // localhost: 44378 / api / Product / 1. El cartero le mostrará el JSON del producto que tiene la identificación 1. He mostrado esto en la siguiente imagen:

Interfaz de usuario gráfica, Texto, Aplicación, Correo electrónico

Descripción generada automáticamente

**Actualización de un producto con CQRS**

Creemos ahora un comando CQRS para actualizar un producto. Dentro de la carpeta CQRS ➤ Commands, agregue una nueva clase llamada UpdateProductCommand.cs. Su código se proporciona a continuación:

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17  18  19  20  21  22  23  24  25  26  27  28  29  30  31  32  33  34  35  36  37  38  39  40  41  42 | using CQRSMediator.Models;  using MediatR;  using System;  using System.Collections.Generic;  using System.Linq;  using System.Threading;  using System.Threading.Tasks;    namespace CQRSMediator.CQRS.Commands  {      public class UpdateProductCommand : IRequest<int>      {          public int Id { get; set; }          public string Name { get; set; }          public decimal Price { get; set; }            public class UpdateProductCommandHandler : IRequestHandler<UpdateProductCommand, int>          {              private ProductContext context;              public UpdateProductCommandHandler(ProductContext context)              {                  this.context = context;              }              public async Task<int> Handle(UpdateProductCommand command, CancellationToken cancellationToken)              {                  var product = context.Product.Where(a => a.Id == command.Id).FirstOrDefault();                    if (product == null)                  {                      return default;                  }                  else                  {                      product.Name = command.Name;                      product.Price = command.Price;                      await context.SaveChangesAsync();                      return product.Id;                  }              }          }      }  } |

Observe que la clase UpdateProductCommand implementa IRequest <int> igual que CreateProductCommand.cs que estaba creando un nuevo producto en la base de datos.

Esta clase se explica por sí misma y puede comprender muy bien que está actualizando un producto. El producto se actualiza dentro del método Handle (), y esta clase devuelve el ID de producto del producto que se actualiza.

A continuación, agregue el método Update () de tipo Http Put a la clase ProductController. El código se proporciona a continuación:

[HttpPut("{id}")]

public async Task<IActionResult> Update(int id, UpdateProductCommand command)

{

command.Id = id;

return Ok(await mediator.Send(command));

}

Hagamos ahora una llamada a este método con Postman y actualicemos nuestro producto.

Entonces, en Postman, seleccione PUT para el verbo http y la URL como https: // localhost: 44378 / api / Product / 1. Tenga en cuenta que 1 al final de la URL pasa la identificación del producto.

Seleccione "Body", "raw" y "JSON" para las opciones, como cuando creamos un producto. Para el niño de texto, ingrese los nuevos valores para el producto en json. Voy a cambiar el nombre del producto a "Camisa de hombre" y el precio a 89 agregando el siguiente json al cuadro de texto.

{

"name": "Men's Shirt",

"price": 89

}

Finalmente, haga clic en el botón Enviar y su producto se actualizará. Vea la siguiente imagen:

Interfaz de usuario gráfica, Aplicación

Descripción generada automáticamente

Ahora puede confirmar que el producto está actualizado haciendo una solicitud GET con Postman a la URL: https: // localhost: 44378 / api / Product. Vea la captura de pantalla a continuación.

Interfaz de usuario gráfica, Texto, Aplicación

Descripción generada automáticamente

**Eliminar un producto con CQRS**

La operación CRUD final es eliminar un producto con el comando CQRS. Entonces, cree una nueva clase llamada DeleteProductByIdCommand dentro de la carpeta CQRS ➤ Commands. Agregue el siguiente código a esta clase:

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17  18  19  20  21  22  23  24  25  26  27  28  29  30  31 | using CQRSMediator.Models;  using MediatR;  using Microsoft.EntityFrameworkCore;  using System;  using System.Collections.Generic;  using System.Linq;  using System.Threading;  using System.Threading.Tasks;    namespace CQRSMediator.CQRS.Commands  {      public class DeleteProductByIdCommand : IRequest<int>      {          public int Id { get; set; }          public class DeleteProductByIdCommandHandler : IRequestHandler<DeleteProductByIdCommand, int>          {              private ProductContext context;              public DeleteProductByIdCommandHandler(ProductContext context)              {                  this.context = context;              }              public async Task<int> Handle(DeleteProductByIdCommand command, CancellationToken cancellationToken)              {                  var product = await context.Product.Where(a => a.Id == command.Id).FirstOrDefaultAsync();                  context.Product.Remove(product);                  await context.SaveChangesAsync();                  return product.Id;              }          }      }  } |

La clase implementa IRequest <int>, int ya que el tipo significa que devolverá un valor int. Confirme esto ya que el método Handle devuelve la identificación del producto al final.

return product.Id;

El método handle realiza la eliminación del producto a quién lo envía el controlador. Entity Framework Core se encarga de eliminar el producto.

A continuación, agregue el método de eliminación al controlador del producto.

[HttpDelete("{id}")]

public async Task<IActionResult> Delete(int id)

{

return Ok(await mediator.Send(new DeleteProductByIdCommand { Id = id }));

}

Entonces, es hora de probar la funcionalidad de eliminación con Postman. Entonces, en Postman, haga el tipo de solicitud Delete a la URL: https: // localhost: 44378 / api / Product / 1. Esto eliminará el producto con id 1 de la base de datos. La captura de pantalla de Postman se muestra a continuación:

Interfaz de usuario gráfica, Texto, Aplicación, Correo electrónico

Descripción generada automáticamente

Ahora, como de costumbre, haga una solicitud GET para confirmar que se eliminó el producto. Esta vez obtendrá un json vacío que le indica que el producto se ha eliminado.

Interfaz de usuario gráfica, Texto, Aplicación

Descripción generada automáticamente

Felicitaciones, creamos con éxito operaciones CRUD con CQRS y patrones Mediator en ASP.NET Core usando Entity Framework Core.

Así que ahora tenemos 3 clases dentro de la carpeta Comandos y 2 clases dentro de la carpeta Consultas. Verifique la siguiente imagen:

Interfaz de usuario gráfica, Aplicación

Descripción generada automáticamente

Veamos ahora otras características importantes de la biblioteca MediatR.

**Notificaciones de MediatR**

Diagrama

Descripción generada automáticamente

Hasta ahora, hemos visto un solo request manejado por una sola handler. Ejemplo: solicitud de eliminación de un producto manejada por el método Handle de la clase DeleteProductByIdCommandHandler.

Ahora veremos cómo una sola solicitud será manejada por múltiples manejadores. Esto se hace mediante Notificaciones MediatR.

Por ejemplo, siempre que se elimine un producto, haremos 2 cosas:

1. Enviar un correo electrónico a los clientes que el Producto ya no está disponible.
2. Enviar SMS al equipo de ventas para que se elimine el producto, por lo que probablemente deberían crear un nuevo producto para su lugar.

Creemos notificaciones MediatR para ello.

Primero cree una nueva carpeta llamada Notificaciones en la raíz de la aplicación. Luego, dentro de él, cree una nueva clase llamada DeleteProductNotification.cs con el código que se indica a continuación:

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17  18  19  20  21  22  23  24  25  26  27  28  29  30  31  32  33  34 | using MediatR;  using System;  using System.Collections.Generic;  using System.Linq;  using System.Threading;  using System.Threading.Tasks;    namespace CQRSMediator.Notifications  {      public class DeleteProductNotification : INotification      {          public int ProductId { get; set; }      }        public class EmailHandler : INotificationHandler<DeleteProductNotification>      {          public Task Handle(DeleteProductNotification notification, CancellationToken cancellationToken)          {              int id = notification.ProductId;              // send email to customers              return Task.CompletedTask;          }      }        public class SMSHandler : INotificationHandler<DeleteProductNotification>      {          public Task Handle(DeleteProductNotification notification, CancellationToken cancellationToken)          {              int id = notification.ProductId;              //send sms to sales team              return Task.CompletedTask;          }      }  } |

Esta es la ubicación de la clase en el directorio de la aplicación.

Interfaz de usuario gráfica, Texto, Aplicación

Descripción generada automáticamente

Entendamos lo que estamos haciendo en esta clase:

1. Creamos una clase llamada DeleteProductNotification que implementa la interfaz INotification del espacio de nombres MediatR. La clase contiene una única propiedad denominada ProductId.
2. A continuación, creamos 2 clases de manejadores llamadas EmailHandler y SMSHandler. Ambos implementan INotificationHandler con un tipo de DeleteProductNotification. Esto significa que manejarán eventos de tipo DeleteProductNotification.
3. Los manipuladores hacen el trabajo que se les asigna, como enviar correo electrónico o SMS. También vea que el método Handle de estos 2 manejadores tiene la clase DeleteProductNotification como primer parámetro y a través del cual podemos obtener los valores de propiedad. Ejemplo Obtengo el valor ProductId como se muestra a continuación.

int id = notification.ProductId;

Activar la notificación de MediatR.

Activemos la notificación que se realiza a través del método Publish. Así que vaya a ProductController.cs y agregue el método Publish al método Delete como se muestra en el código resaltado a continuación:

|  |  |
| --- | --- |
| 1  2  3  4  5  6 | [HttpDelete("{id}")]  public async Task<IActionResult> Delete(int id)  {      await mediator.Publish(new Notifications.DeleteProductNotification { ProductId = id });      return Ok(await mediator.Send(new DeleteProductByIdCommand { Id = id }));  } |

Entonces, ahora, cuando elimine un producto, se publicará la Notificación de MediatR y se ejecutarán 2 clases de controlador. Las clases de controlador están acopladas libremente con el método de publicación. Esto es una gran cosa, ya que podemos extender más clases de controladores sin necesidad de modificar el método de publicación en el método Delete de la clase Product Controller.

**MediatR Behaviors**

Los comportamientos de MediatR son muy similares a los middlewares en ASP.NET Core. Aceptan una solicitud, realizan alguna acción y luego (opcionalmente) pasan la solicitud. Lo bueno de los comportamientos es que podemos poner una lógica en ellos en lugar de repetirla una y otra vez en nuestra aplicación.

Creemos un comportamiento de MediatR que hará el loggeo por nosotros. Así que cree una nueva carpeta llamada Behaviors en la raíz de la aplicación y agregue una nueva clase llamada LoggingBehavior.cs.

Interfaz de usuario gráfica, Texto, Aplicación

Descripción generada automáticamente

El código completo de esta clase se proporciona a continuación:

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17  18  19  20  21  22  23  24  25  26 | using MediatR;  using Microsoft.Extensions.Logging;  using System;  using System.Collections.Generic;  using System.Linq;  using System.Threading;  using System.Threading.Tasks;    namespace CQRSMediator.Behaviors  {      public class LoggingBehavior<TRequest, TResponse> : IPipelineBehavior<TRequest, TResponse>      {          private readonly ILogger<LoggingBehavior<TRequest, TResponse>> logger;          public LoggingBehavior(ILogger<LoggingBehavior<TRequest, TResponse>> logger)          {              this.logger = logger;          }          public async Task<TResponse> Handle(TRequest request, CancellationToken cancellationToken, RequestHandlerDelegate<TResponse> next)          {              logger.LogInformation($"Befor {typeof(TRequest).Name}");              var response = await next();              logger.LogInformation($"After {typeof(TResponse).Name}");              return response;          }      }  } |

Cosas importantes:

1. La clase MediatR Behavior debe tener dos parámetros de tipo TRequest y TResponse, y debe implementar la interfaz IPipelineBehavior <TRequest, TResponse>.
2. En el método Handle, registramos la información antes y después de cualquier solicitud. Este controlador de registro se puede aplicar a cualquier solicitud y registrará la salida antes y después de que se maneje.
3. El next() pasa a la siguiente acción en la canalización de middleware.

Para probar el comportamiento, haga una solicitud de producto de lectura con Postman. Como antes, haga una solicitud HTTP GET a la URL: https: // localhost: 44378 / api / Product. El puerto será diferente en tu caso.

Interfaz de usuario gráfica, Texto, Aplicación, Correo electrónico

Descripción generada automáticamente

A continuación, en Visual Studio, abra la ventana de salida cuya ruta es el menú Ver y luego Salida. Verá 2 registros:

Information: Befor GetAllProductQuery

Information: After IEnumerable`1

He mostrado estos registros en la siguiente imagen:

Texto

Descripción generada automáticamente

¡Excelente! Esta es la salida de registro antes y después de que se invocó nuestro controlador de consultas GetAllProductQuery. Debido a la arquitectura poco acoplada de CQRS, enganchamos el comportamiento sin tocar ninguna otra característica de la aplicación.