Implementing CQRS with MediatR in ASP.NET Core – Ultimate Guide

By Mukesh Murugan Updated on October 24, 2021

In this article let's talk about CQRS in ASP.NET Core 3.1 and it's implementation along with MediatR and Entity Framework Core – Code First Approach. I will implement this pattern on a WebApi Project. The source code of this sample is linked at the end of the post. Of the several design patterns available, CQRS is one of the most commonly used patterns that helps architect the Solution to accommodate the Onion Architecture. I will be writing a post on Implementation of Onion Architecture too later, the cleanest way to structure a .NET Solution. So, Let's get started!

CQRS with MediatR in ASP.NET Core 3.1 – Coverage Topics

What Is CQRS?

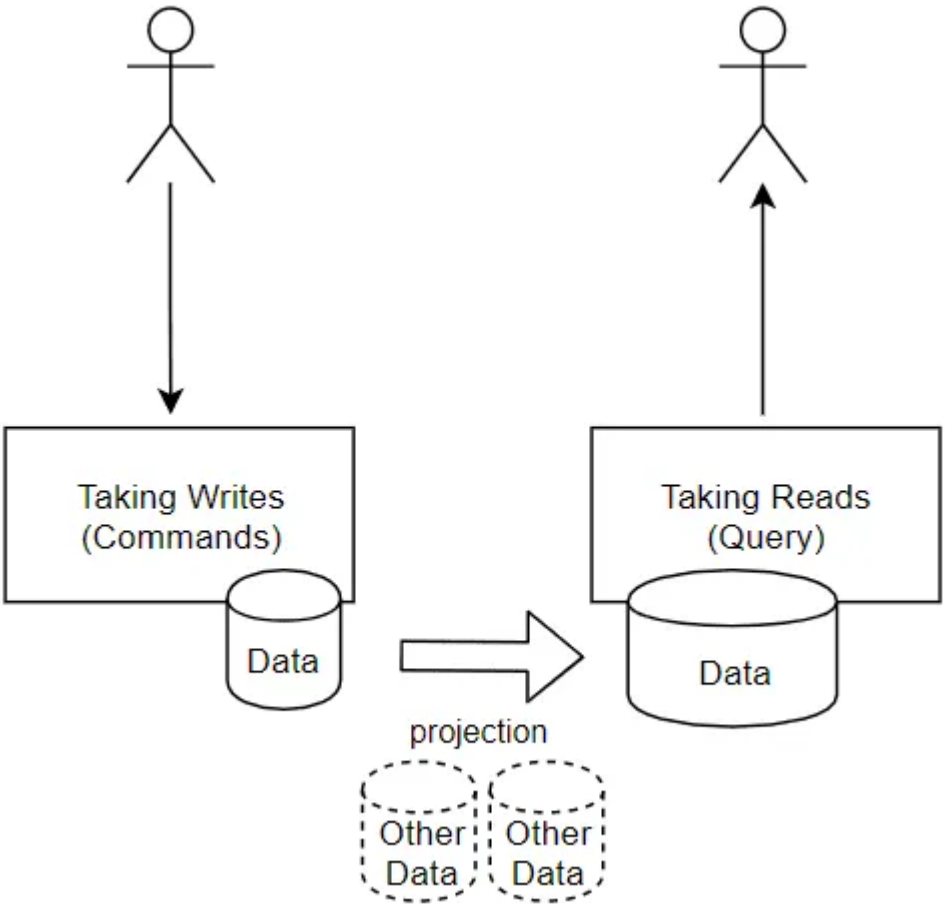
CQRS, Command Query Responsibility Segregation is a design pattern that separates the read and write operations of a data source. Here Command refers to a Database Command, which can be either an Insert / Update or Delete Operation, whereas Query stands for Querying data from a source. It essentially separates the concerns in terms of reading and writing, which makes quite a lot of sense. This pattern was originated from the Command and Query Separation Principle devised by Bertrand Meyer. It is defined on Wikipedia as follows.

> *It states that every method should either be a command that performs an action, or a query that returns data to the caller, but not both. In other words, asking a question should not change the answer. More formally, methods should return a value only if they are referentially transparent and hence possess no side effects.*
>
> Wikipedia

The problem with traditional architectural patterns is that the same data model or DTO is used to query as well as update a data source. This can be the go-to approach when your application is related to just CRUD operations and nothing more. But when your requirements suddenly start getting complex, this basic approach can prove to be a disaster.

In practical applications, there is always a mismatch between the read and write forms of data, like the extra properties you may require to update. Parallel operations may even lead to data loss in the worst cases. That means, you will be stuck with just one Data Transfer Object for the entire lifetime of the application unless you choose to introduce yet another DTO, which in-turn may break your application architecture.

The idea with CQRS is to allow an application to work with different models. Long story short, you have one model that has data needed to update a record, another model to insert a record, yet another to query a record. This gives you flexibility with varying and complex scenarios. You don't have to rely on just one DTO for the entire CRUD Operations by implementing CQRS.



Pros Of CQRS

There are quite of lot of advantages on using the CQRS Pattern for your application. Few of them are as follows.

Optimised Data Transfer Objects

Thanks to the segregated approach of this pattern, we will no longer need those complex model classes within our application. Rather we have one model per data operation that gives us all the flexibility in the world.

Highly Scalable

Having control over the models in accordance with the type of data operations makes your application highly scalable in the long run.

Improved Performance

Practically speaking there are always 10 times more Read Operations as compared to the Write Operation. With this pattern you could speed up the performance on your read operations by introducing a cache or NOSQL Db like Redis or Mongo. CQRS pattern will support this usage out of the box, you would not have to break your head trying to implement such a cache mechanism.

Secure Parallel Operations

Since we have dedicated models per oprtation, there is no possibility of data loss while doing parellel operations.

Cons Of CQRS

Added Complexity and More Code

The one thing that may concern a few programmers is that this is a code demanding pattern. In other words, you will end up with at least 3 or 4 times more code-lines than you usually would. But everything comes for a price. This according to me is a small price to pay while getting the awesome features and possibilities with the pattern.

Implementing CQRS Pattern In ASP.NET Core 3.1 WebAp

Let' s build an ASP.NET Core 3.1 WebApi to showcase the implementation and better understand the CQRS Pattern. I will push the implemented solution over to Github, You can find the link to my repository at the end of this post. Let us build an API endpoint that does CRUD operations for a Product Entity, ie, Create / Delete / Update / Delete product record from the Database. Here, I use Entity Framework Core as the ORM to access data from my local DataBase.

PS – We will not be using any advanced architectural patterns, but let's try to keep the code clean. The IDE I use is Visual Studio 2019 Community. If you do not have it, I totally recommend getting it. It's completely free. Read the Installation Guide here.

Setting up the Project

Open up Visual Studio and Create a new ASP.NET Core Web Application with the WebApi Template.

Installing the required Packages

Install these following packages to your API project via the Package Manager Console. Just Copy Paste the below lines over to your Package Manager Console. All the required packages get installed. We will explore these packages as we progress.

```
1. Install-Package Microsoft.EntityFrameworkCore
2. Install-Package Microsoft.EntityFrameworkCore.Relational
3. Install-Package Microsoft.EntityFrameworkCore.SqlServer
4. Install-Package MediatR
5. Install-Package MediatR.Extensions.Microsoft.DependencyInjection
6. Install-Package Swashbuckle.AspNetCore
7. Install-Package Swashbuckle.AspNetCore.Swagger
```

Adding the Product Model

Since we are following a code first Approach, let's design our data models. Add **Models** Folder and create a new class named Product with the following properties.

```
public class Product
    {
        public int Id { get; set; }
        public string Name { get; set; }
        public string Barcode { get; set; }
        public bool IsActive { get; set; } = true;
        public string Description { get; set; }
        public decimal Rate { get; set; }
        public decimal BuyingPrice { get; set; }
        public string ConfidentialData { get; set; }
    }
```

Adding the Context Class and Interface

Make a new Folder called Context and add a class named Application Context. This particular class will help us to access the data using Entity Framework Core ORM.

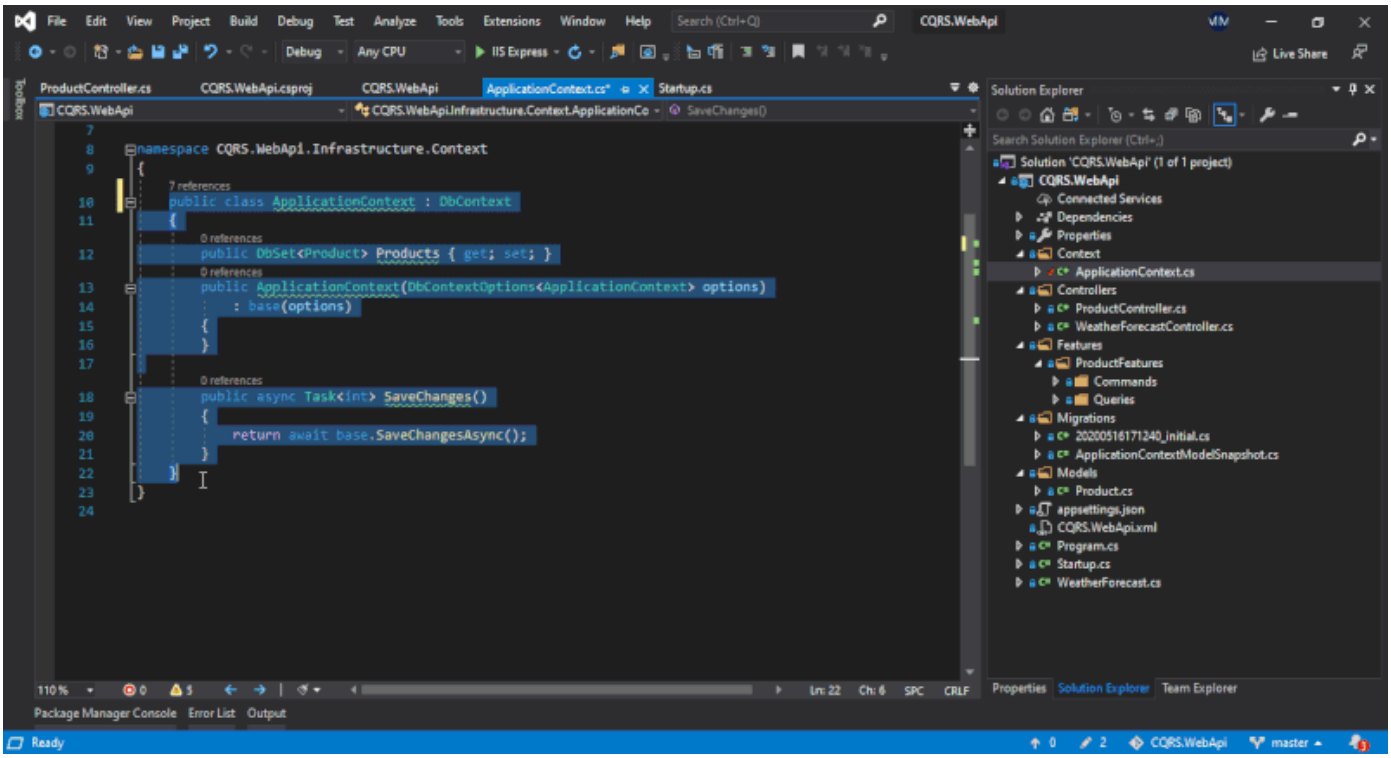
```
public class ApplicationContext : DbContext
    {
        public DbSet<Product> Products { get; set; }
        public ApplicationContext(DbContextOptions<ApplicationContext> options)
            : base(options)
        { }
    public async Task<int> SaveChanges()
        {
            return await base.SaveChangesAsync();
        }
    }
```

PRO TIP – How to Extract an Interface from a Class?

Now that we have completed the class, let me show you an easy way to generate an Interface for any given class. Visual Studio is much powerful than what we think it is. So here is how it goes.

1. Go to the class that we need an Interface for. In our case, go to the ApplicationContext.cs class.
2. Select the entire class.
3. Once selected, go to Edit -> Refactor -> Extract Interface.

4. VS will ask for confirmation. Verify the name of the Interface to be generated and click Ok.
5. Boom, we have our Interface ready. Imagine how helpful this feature will be when we have pretty long classes.



Configuring the API Services to support Entity Framework Core

I have written an article on Entity Framework Core in ASP.NET Core 3.1. Do give it a look.

Navigate to your API Project's Startup class. This is the class where the Application knows about various services and registrations required. Let's add the support for EntityFrameworkCore. Just add these lines to your Startup Class's ConfigureServices method. This will register the EF Core with the application.

```
services.AddDbContext<ApplicationContext>(options =>
                    options.UseSqlServer(
                        Configuration.GetConnectionString("DefaultConnection"),
                        b => b.MigrationsAssembly(typeof(ApplicationContext).Ass
```

> Line 3 Says about the Connection String that is named as DefaultConnection. But we haven't defined any such connection, have we?

Defining the Connection String in appsettings.json

We will need to connect a data source to the API. For this, we have to define a connection string in the appsettings.json found within the API Project. For demo purposes, I am using LocalDb Connection. You could scale it up to support multiple Database type. Various connection string formats can be found here.

Here is what you would add to your appsettings.json.

```
1.  "ConnectionStrings": {
2.      "DefaultConnection": "Server=(localdb)\\mssqllocaldb;Database=developmentDb;Tru
3.    },
```

Generating the Database

Now we have our models and the connection string ready, all we have to do is to generate a database from the defined models. For this, we have to use the Package Manager Console of Visual Studio. You can open this by going to Tools -> Nuget Package Manager -> Package Manager Console.

> Before continuing, let's check if there are any Build issues with the Solution. Build the Application once to ensure that there is no error, because it is highly possible for the next steps to not show any proper warning and fail if any errors exist.

Once the Build has succceeded, Let's start by adding the migrations.

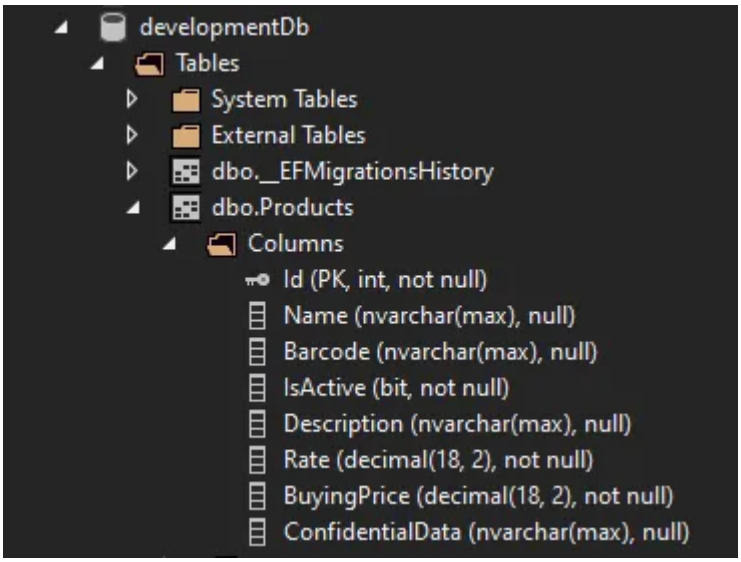
```
1.  add-migration "initial"
```

After that, move on to update the database

```
1.  update-database
```

You will get a Done Message once these operations are completed. As simple as that, we have got our database running in no time. This is exactly why including me, many developers prefer Entity Framework, It's too easy and does not make any compromises in terms of performance or features. I am also planning to make an in-depth guide on EFCore later, which I will link up here once ready.

Let's verify that the database has been created properly. Go to View -> SQL Server Object Explorer



The Database it properly created. Now, we will wire up this database to our API to perform CRUD Operations.

The Mediator Pattern

While building applications, especially ASP.NET Core Applications, it's highly important to keep in mind that we always have to keep the code inside controllers as minimal as possible. Theoretically, Controllers are just routing mechanisms that take in a request and sends it internally to other specific services/libraries ar

return the data. It wouldn't really make sense to put all your validations and logics within the Controllers.

Mediator pattern is yet another design pattern that dramatically reduces the coupling between various components of an application by making them communicate indirectly, usually via a special mediator object. We will dive deep into Mediator in another post. Essentially, the Mediator pattern is well suited for CQRS implementation.

MediatR Library

MediatR is a library that helps implements Mediator Pattern in .NET. The first thing we need to do, is to install the following packages.

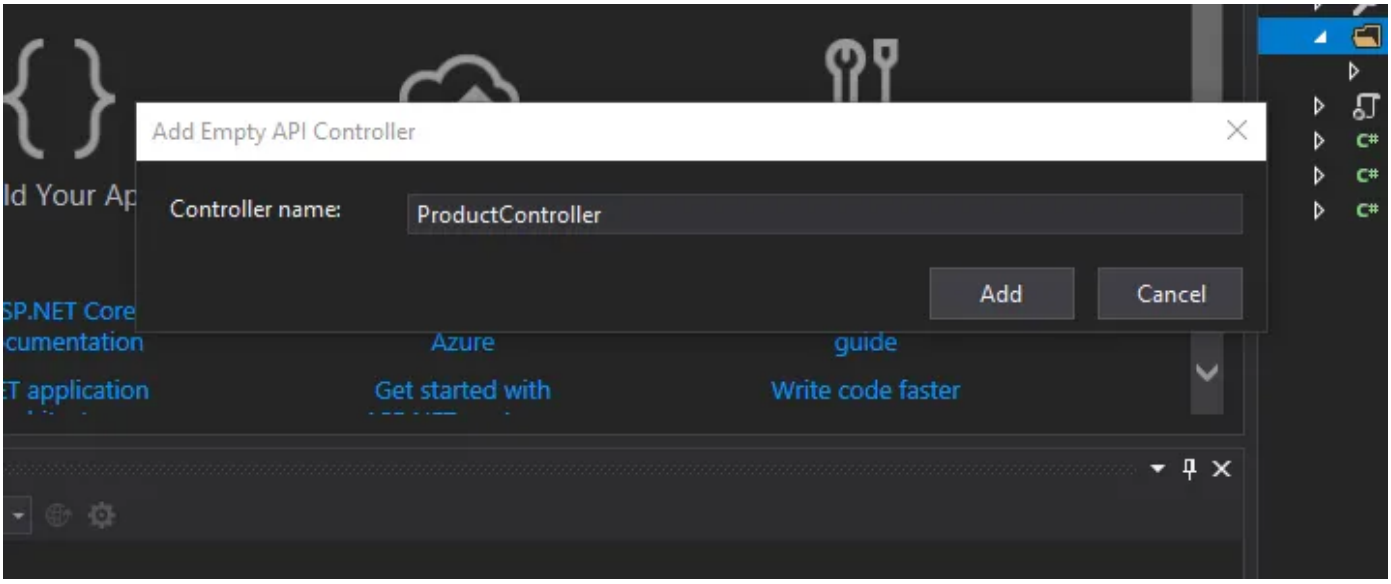
Configuring MediatR

We have already installed the required package to our application. To register the library, add this line to the end of our API startup class in the ConfigureService Method.

```
1.  services.AddMediatR(Assembly.GetExecutingAssembly());
```

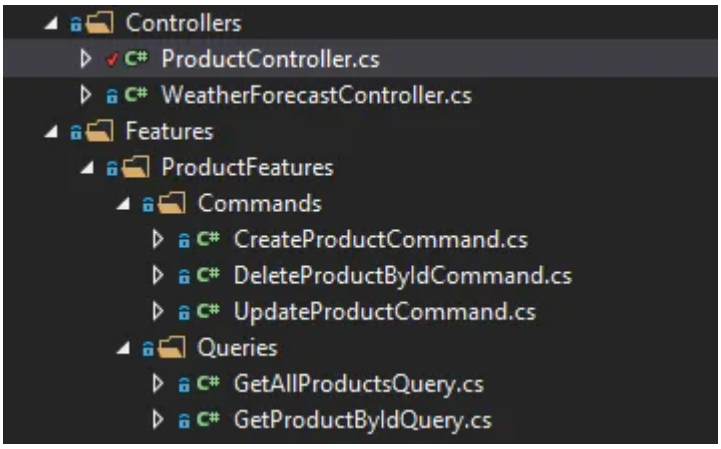
Creating the Product Controller

To the **Controllers** Folder, Add a new Empty API Controller and name it ProductController.



Implementing the CRUD Operations

CRUD essentially stands for Create, Read, Update, and Delete. These are the Core components of RESTFul APIs. Let's see how we can implement them using our CQRS Approach. Create a Folder named Features in the root directory of the Project and subfolders for the Queries and Command.



Queries

Here is where we will wire up the queries, ie, GetAllProducts and GetProductById. Make 2 Classes under the ProductFeatures / Queries Folder and name them GetAllProductsQuery and GetProductByIdQuery

Query to Get All Products

```
1.  public class GetAllProductsQuery : IRequest<IEnumerable<Product>>
2.      {
3.          public class GetAllProductsQueryHandler : IRequestHandler<GetAllProductsQue
4.          {
5.              private readonly IApplicationContext _context;
6.              public GetAllProductsQueryHandler(IApplicationContext context)
7.              {
8.                  _context = context;
9.              }
10.             public async Task<IEnumerable<Product>> Handle(GetAllProductsQuery quer
11.             {
12.                 var productList = await _context.Products.ToListAsync();
13.                 if (productList == null)
14.                 {
15.                     return null;
16.                 }
17.                 return productList.AsReadOnly();
18.             }
19.         }
20.     }
```

Line 1 suggests that we intend to return an IEnumerable list of Products from this Class implementing the IRequest interface of MediatR. Every request must have a request handler. Here it is mentioned in Line 3. At Line 10, we define how the request is being handled. This is almost the same for all types of requests and commands.

Query to Get Product By Id

```
1.   public class GetProductByIdQuery : IRequest<Product>
2.      {
3.          public int Id { get; set; }
4.          public class GetProductByIdQueryHandler : IRequestHandler<GetProductByIdQue
5.          {
6.              private readonly IApplicationContext _context;
7.              public GetProductByIdQueryHandler(IApplicationContext context)
8.              {
9.                  _context = context;
10.             }
11.             public async Task<Product> Handle(GetProductByIdQuery query, Cancellati
12.             {
13.                 var product =  _context.Products.Where(a => a.Id == query.Id
14.                 if (product == null) return null;
15.                 return product;
16.             }
```

```
        }
    }
```

Line #3 is the id of the product we need to fetch.
Line #13 , we query the database and fetch the record with Id as our query Id.

Commands

Add the following classes to ProductFeatures / Commands.
1.CreateProductCommand
2.DeleteProductByIdCommand
3.UpdateProductCommand

Command to Create a New Product

```
public class CreateProductCommand : IRequest<int>
    {
        public string Name { get; set; }
        public string Barcode { get; set; }
        public string Description { get; set; }
        public decimal BuyingPrice { get; set; }
        public decimal Rate { get; set; }
        public class CreateProductCommandHandler : IRequestHandler<CreateProductCom
        {
            private readonly IApplicationContext _context;
            public CreateProductCommandHandler(IApplicationContext context)
            {
                _context = context;
            }
            public async Task<int> Handle(CreateProductCommand command, Cancellatio
            {
                var product = new Product();
                product.Barcode = command.Barcode;
                product.Name = command.Name;
                product.BuyingPrice = command.BuyingPrice;
                product.Rate = command.Rate;
                product.Description = command.Description;
                _context.Products.Add(product);
                await _context.SaveChanges();
                return product.Id;
            }
        }
```

```
    }
```

Command to Delete a Product By Id

```
public class DeleteProductByIdCommand : IRequest<int>
    {
        public int Id { get; set; }
        public class DeleteProductByIdCommandHandler : IRequestHandler<DeleteProduc
        {
            private readonly IApplicationContext _context;
            public DeleteProductByIdCommandHandler(IApplicationContext context)
            {
                _context = context;
            }
            public async Task<int> Handle(DeleteProductByIdCommand command, Cancell
            {
                var product = await _context.Products.Where(a => a.Id == command.Id
                if (product == null) return default;
                _context.Products.Remove(product);
                await _context.SaveChanges();
                return product.Id;
            }
        }
    }
```

Command to Update a Product

```
public class UpdateProductCommand : IRequest<int>
    {
        public int Id { get; set; }
        public string Name { get; set; }
        public string Barcode { get; set; }
        public string Description { get; set; }
        public decimal BuyingPrice { get; set; }
        public decimal Rate { get; set; }
        public class UpdateProductCommandHandler : IRequestHandler<UpdateProductCom
        {
            private readonly IApplicationContext _context;
            public UpdateProductCommandHandler(IApplicationContext context)
            {
                _context = context;
            }
            public async Task<int> Handle(UpdateProductCommand command, Cancellatio
```

```
            {
                var product = _context.Products.Where(a => a.Id == command.Id).Firs
                if (product == null)
                {
                    return default;
                }
                else
                {
                    product.Barcode = command.Barcode;
                    product.Name = command.Name;
                    product.BuyingPrice = command.BuyingPrice;
                    product.Rate = command.Rate;
                    product.Description = command.Description;
                    await _context.SaveChanges();
                    return product.Id;
                }
            }
        }
    }
```

Product Controller

```
public class ProductController : ControllerBase
    {
        private IMediator _mediator;

        protected IMediator Mediator => _mediator ??= HttpContext.RequestServices.G

        [HttpPost]
        public async Task<IActionResult> Create(CreateProductCommand command)
        {
            return Ok(await Mediator.Send(command));
        }
        [HttpGet]
        public async Task<IActionResult> GetAll()
        {
            return Ok(await Mediator.Send(new GetAllProductsQuery()));
        }
        [HttpGet("{id}")]
        public async Task<IActionResult> GetById(int id)
        {
            return Ok(await Mediator.Send(new GetProductByIdQuery { Id = id }));
        }
        [HttpDelete("{id}")]
        public async Task<IActionResult> Delete(int id)
        {
```

```
            return Ok(await Mediator.Send(new DeleteProductByIdCommand { Id = id })
        }
        [HttpPut("{id}")]
        public async Task<IActionResult> Update(int id, UpdateProductCommand comman
        {
            if (id != command.Id)
            {
                return BadRequest();
            }
            return Ok(await Mediator.Send(command));
        }
    }
```

Testing

Since we are done with the implementation, let's check the result. For this we use Swagger UI. We have already installed the package. Let's configure it.

Configure Swagger

Add this piece of code to Startup.cs / ConfigureServices

```
#region Swagger
            services.AddSwaggerGen(c =>
            {
                c.IncludeXmlComments(string.Format(@"{0}\CQRS.WebApi.xml", System.A
                c.SwaggerDoc("v1", new OpenApiInfo
                {
                    Version = "v1",
                    Title = "CQRS.WebApi",
                });

            });
            #endregion
```

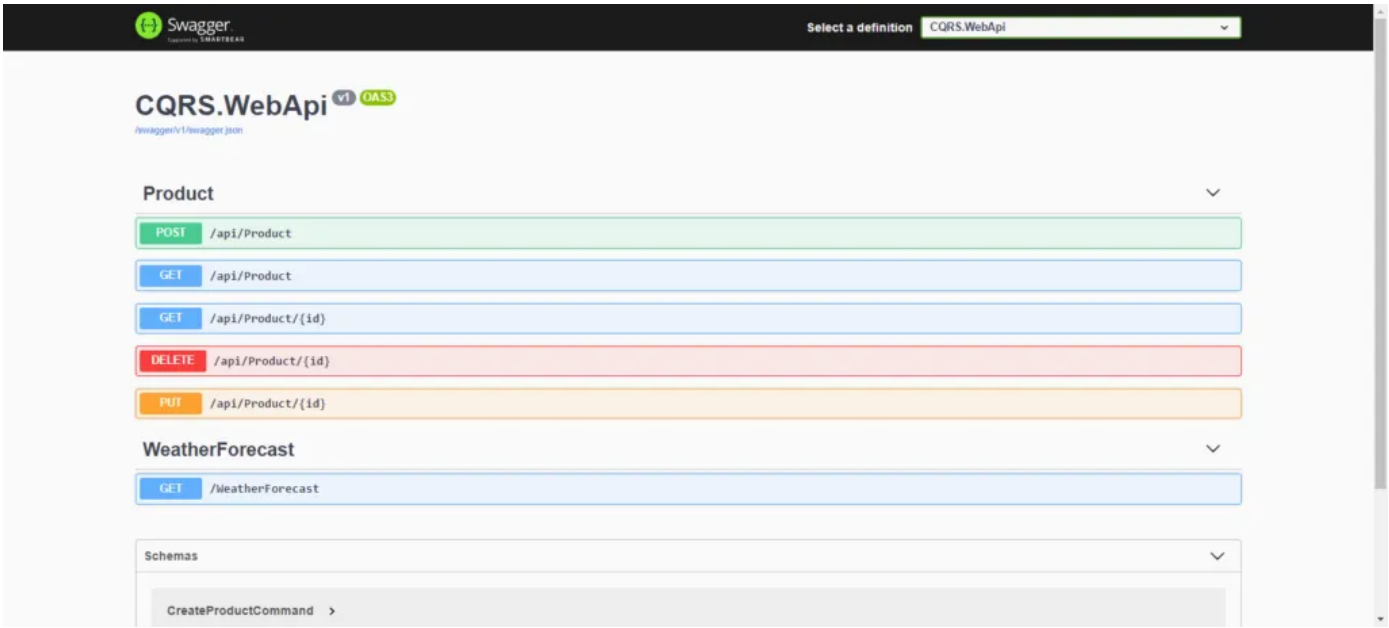
Then, Add this to Configure method of Startup.cs

```
#region Swagger
```

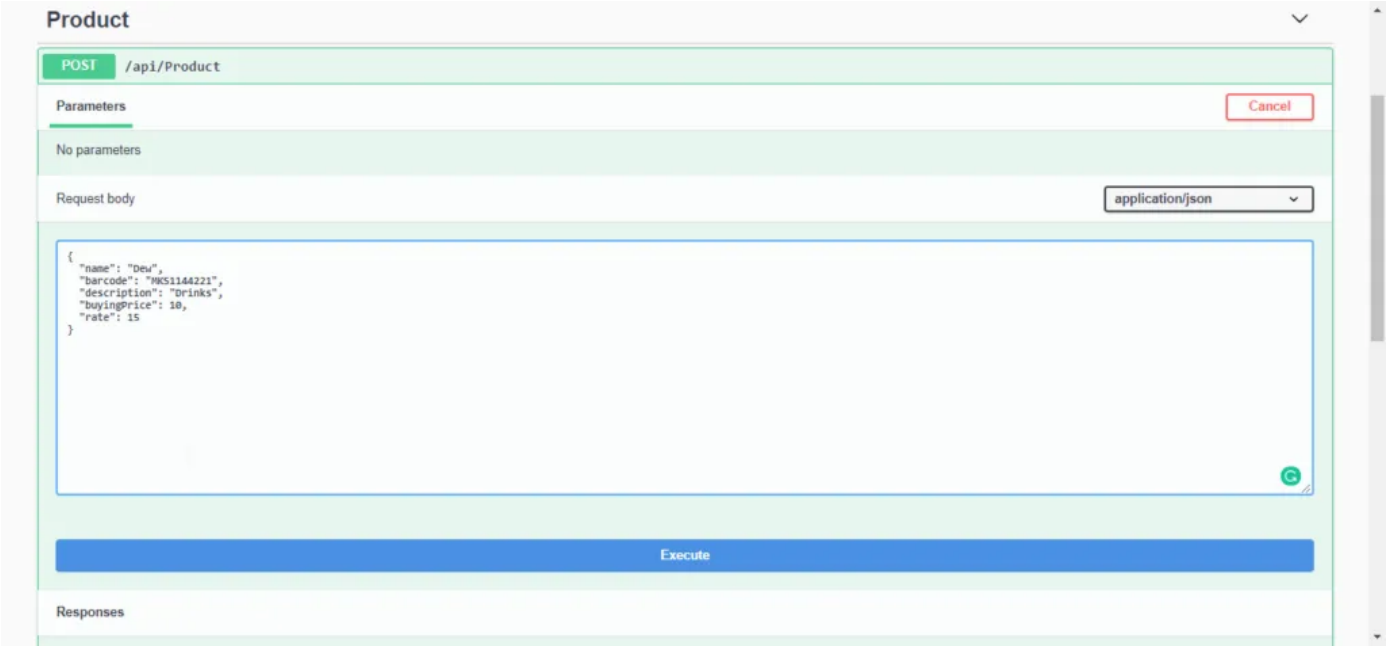
```
            app.UseSwagger();
            app.UseSwaggerUI(c =>
            {
                c.SwaggerEndpoint("/swagger/v1/swagger.json", "CQRS.WebApi");
            });
#endregion
```

Testing with Swagger

Now build the application and run it. Navigate to https://localhost:44311/swagger/. PS , your localhost port may vary. Navigate accordingly.



This is Swagger UI. It lets you document and test your APIs with great ease. To add a new product, click on the POST dropdown and add the details of the new product like below.



Once done, click on execute. This will try to add the specified record to the database. Now let's see the list of products. Navigate to the GET method and click on execute. This is how easy things get with swagger.

Summary

We have covered CQRS implementation and definition, Mediator pattern and MediatR Library, Entity Framework Core Implementation, Swagger Integration, and much more. If I missed out on something or was not clear with the guide, let me know in the comments section below. Is CQRS your go-to approach for Complicated Systems? Let us know.

Source Code at Github

The finished source code of this implementation is available at Github.
Do follow me at GitHub as well

Take me to GitHub

Frequently Asked Questions

What is the full form of CQRS?

CQRS stands for Command Query Responsibility Segregation. It's a design pattern that seperated the read and write operations, hence decoupling the solution to a great extend.

Does CQRS Slow down your application?

No. CQRS may require a lot of code. But it does not hurt the performance at level. With well written code, CQRS actually performs the data source calls in better way.

Is CQRS a scalable design pattern?

Yes, CQRS is devised with scalability in mind.

#.NET Core #API #CQRS #Mediator

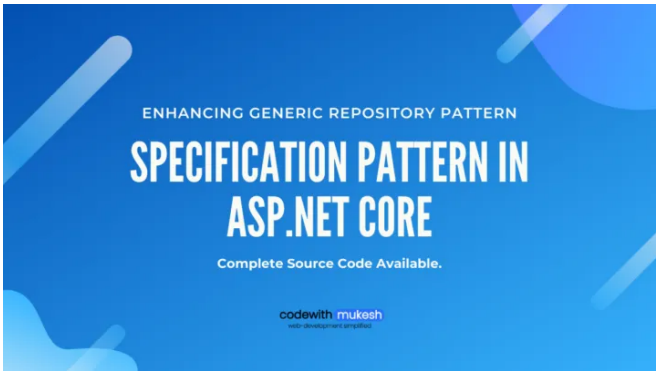
← PREVIOUS

How to Install Visual Studio 2019 Community – The Best IDE for C#

NEXT →

15 BEST Libraries For ASP.NET Core Developers

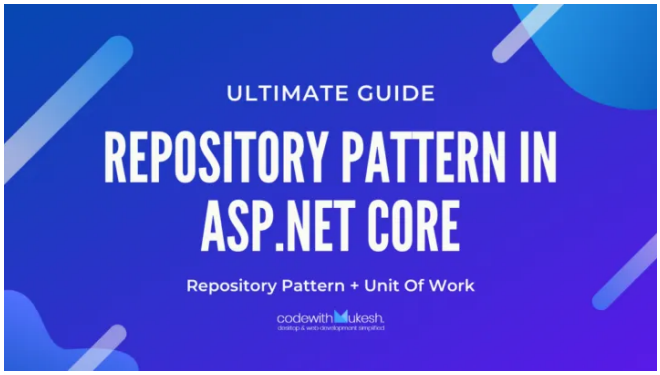
Similar Posts



Specification Pattern in ASP.NET Core – Enhancing Generic Repository Pattern

By Mukesh Murugan

April 24, 2021



Repository Pattern in ASP.NET Core – Ultimate Guide

By Mukesh Murugan

June 28, 2020

Leave a Reply

Your email address will not be published. Required fields are marked *

Comment *

Name *

Email *

Website

Save my name, email, and website in this browser for the next time I comment.

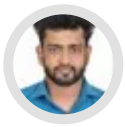
Post Comment

37 Comments

Florin Asavei says:
May 20, 2020 at 12:40 pm

You should also run “Install-Package Microsoft.EntityFrameworkCore.Tools” in the Nuget package Manager Console in order to be able to generate the migrations.

Reply



Mukesh Murugan says:
May 20, 2020 at 1:44 pm

Yes, I believe this package is included by default while installing the other EF Core packages on ASP.NET Core Applications. You can refer to https://docs.microsoft.com/en-us/ef/core/get-started/install/#get-the-package-manager-console-tools . Thanks!

Reply

Denis says:
June 28, 2020 at 4:36 pm

Is it necessary to check products for null in GetAllProductsQueryHandler.Handle(...) method?

Based on the source code of the ToListAsync(...) method, an empty collection will be returned instead of null

https://bit.ly/2YE5sOA

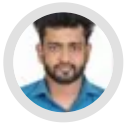
Reply

khalifa says:

June 29, 2020 at 11:04 pm

love it keep posting new articles made my day

Reply



Mukesh Murugan says:
June 30, 2020 at 2:33 pm

I hope that some of the articles helped you. Thanks and Regards

Reply

Zanid Haytam says:
July 12, 2020 at 7:19 am

Hi, good article!
Although I'm not sure that using reflection in the logging part is a good idea, especially in a high load scenario.

Reply

Mukesh Murugan says:
July 12, 2020 at 7:57 am

Hi, Yes I agree. It's not apt to use Reflection. But since this is just a demonstration of pipeline and it has really nothing to do with logging, I trie keep things simple. I had also added a small note to not use this in produc Thanks for the feedback.

Regards.

Reply

Dmitriy says:
August 3, 2020 at 2:13 pm

Hello! Can you provide also solution for Events on CQRS ?
Thank you very much for your work!

Reply

Bujar Ademi says:
August 23, 2020 at 6:01 am

Hi, great article, I liked the way how you simplify things so everyone can understand. One thing to mention in your code basically the class Product is not model but it should be an entity

Reply

Alireza says:
August 25, 2020 at 8:02 pm

Great Article, Thank you.

Reply

Alireza says:
August 25, 2020 at 8:43 pm

great article, thank you

Reply

---

jose says:
September 2, 2020 at 1:47 pm

Hi,
The code below show a alert that it hides inherited member. Its really necessary ?

```
public async Task SaveChanges()
{
return await base.SaveChangesAsync();
}
```

Reply

Robert says:
October 9, 2020 at 3:50 pm

@Jose I have this and it’s working fine, don’t forget

```
public async Task SaveChanges()
{
return await base.SaveChangesAsync();
}
```

BTW love the article, learning loads 🙂

Reply

Robert says:
October 9, 2020 at 3:56 pm

@Jose

Add the generic int to the method signature. My last post stripped it out.

Reply

Emanuel Adrian Lucaci says:
November 6, 2020 at 6:14 pm

Wow. This article is article is pure gold. Thank you!

Reply

Oleg says:
November 23, 2020 at 12:01 pm

Hi! Thank you for your post! It's absolutely clear and useful!
But one thing seems strange to me – duplication of ID in body and in the route of Update request. In my opinion, there is no reason for the Service user to specify ID twice.
Can you suggest solution to avoid it?

Reply

Mick says:
December 27, 2020 at 11:12 pm

Thank you ! I really like the way you explain us the concepts …

Reply

Nguyễn Tuấn Anh says:
January 29, 2021 at 5:21 pm

You forget to install package EntityFrameworkCore..tool sir.

Reply

Kaushik Roy Chowdhury says:
April 14, 2021 at 6:14 am

Yes, that’s a must to include the tools package.

Reply

Dan says:
February 9, 2021 at 8:35 am

I have to say I like this approach way more than standard service – repo appr

Reply

Mukesh Murugan says:
February 9, 2021 at 3:32 pm

Absolutely, this gives more control on the segregating the request and response and ultimately much more cleaner.

Reply

Noureddine says:
March 10, 2021 at 11:38 am

you must add this services.AddScoped();

Reply

Kaushik Roy Chowdhury says:
April 3, 2021 at 5:00 am

Hi Mukesh,
This is a well-written article. I am implementing the code in ASP.NET 5. Everything is working fine except there is 1 error in ProductController at the line:
protected IMediator Mediator => _mediator ??=
HttpContext.RequestServices.GetService();
Error CS0308 The non-generic method 'IServiceProvider.GetService(Type)' cannot be used with type arguments
Do you have any idea how to resolve this issue so that the project builds and

Reply

Kaushik Roy Chowdhury says:
April 3, 2021 at 6:39 am

Subsequent to my question earlier, all you need to do to remove the error is to include the using statement:
using Microsoft.Extensions.DependencyInjection;

Reply

Kaushik Roy Chowdhury says:
April 3, 2021 at 8:01 am

Final changes to get the project running fine is:
Include in the ProductController class:
using Microsoft.Extensions.DependencyInjection;
Then in ProductController class you need to replace:
protected IMediator Mediator => _mediator ??=
HttpContext.RequestServices.GetService();
with:
protected IMediator Mediator => _mediator ??=
HttpContext.RequestServices.GetService(typeof(IMediator)) as IMediator;

Then in Startup class: (Register IApplicationContext) : services.AddScoped();
If anybody requires the full source code I am happy to share it on GitHub.

Reply

Kaushik Roy Chowdhury says:
April 14, 2021 at 7:14 am

The correct code for ASP.NET 5 implementation for Handle Method will be :
You shall need to include, using Microsoft.EntityFrameworkCore;

```
public class GetProductByIdQuery : IRequest
{
public async Task Handle(GetProductByIdQuery query, CancellationToken
cancellationToken)
{
var product = _context.Products.Where(a => a.Id ==
query.Id).FirstOrDefaultAsync();
if (product == null) return null;
return product;
}
}
}
```

Reply

---

duke_meister says:
April 28, 2021 at 5:43 am

How does this relate to your BlazorHero implementation? It seems they are quite different. Thanks.

Reply

Mukesh Murugan says:
April 28, 2021 at 9:40 am

This is a base implementation of CQRS features which is also implemented in the BlazorHero Project. You need to understand this with MediatR pattern to get the complete knowledge of the CQRS Pattern applied in the Application layer.

Regards

Reply

Ghanshyam Singh says:
May 6, 2021 at 3:37 am

Hi,
I am getting this error (Asp.net Core 3.1):-

Error while validating the service descriptor 'ServiceType: MediatR.IRequestHandler`2[CQRS.Api.Features.Queries.GetAllProductsQuery,System.Collections.Generic.IEnumerable`1[CQRS.Api.Models.Product]] Lifetime: Transient ImplementationType: CQRS.Api.Features.Queries.GetAllProductsQuery+GetAllProductsQueryHandler': Unable to resolve service for type 'CQRS.Api.Context.IApplicationContext' while attempting to activate 'CQRS.Api.Features.Queries.GetAllProductsQuery+GetAllProductsQueryHandler'.

However I have configured EF properly in statup class as mentioned in tutorial

services.AddDbContext(options =>
options.UseSqlServer(
Configuration.GetConnectionString("DefaultConnection"),
b => b.MigrationsAssembly(typeof(ApplicationContext).Assembly.FullName)));

Reply

Mukesh Murugan says:
May 6, 2021 at 4:28 am

could you check if you have added in the IApplicationContext interface to the service container?

Reply

Franklin Ezeji says:
May 21, 2021 at 8:37 am

Thank you Mukesh for this great article.

Here's an implementation of this article in ASP.NET Core 5.0 using MS SQL Server for write operations and Redis for read operations:
https://github.com/Ezeji/CQRS_Mediator.WebApi

Let me know your thoughts, I'd love to learn from you.

Reply

Fazrin says:
June 30, 2021 at 7:28 am

I am facing some issues while implementing authentication using JW Token, have you got any documentation for that, it will be very useful, moreover I would like to contribute your Fluent POS OS project, but getting started link is not working, if you can share your idea, I can work on the user interface part which supposed to be in angular

Reply

Mukesh Murugan says:
June 30, 2021 at 7:51 am

Hi, JWT Authentication is implemented in fluentpos as well. you can refer to it's source code.
About getting-started, there is no separate document for it. But you can see the readme – https://github.com/fluentpos/fluentpos#readme. All the details are mentioned here.
If you are interested in contributing to fluentpos UI, could be reach me out on Discord. The links are on the readme as well.

Regards

Reply

Dewa says:
July 20, 2021 at 8:36 am

I believe you need to install-package Microsoft.EntityFrameworkCore.Design to run add migration , thanks.

Reply

---

patrick says:
October 22, 2021 at 1:20 pm

why Iam getting this warnings? new in this

1>C:\Users\001733\source\projects\CQRSwithMediatR\CQRSwithMediatR\Context\ApplicationContext.cs(16,32,16,43): warning CS0114: 'ApplicationContext.SaveChanges()' hides inherited member 'DbContext.SaveChanges()'. To make the current member override that

implementation, add the override keyword. Otherwise add the new keyword.
1>C:\Users\001733\source\projects\CQRSwithMediatR\CQRSwithMediatR\Features\Queries\GetProductByIdQuery.cs(22,40,22,46): warning CS1998: This async method lacks 'await' operators and will run synchronously. Consider using the 'await' operator to await non-blocking API calls, or 'await Task.Run(...)' to do CPU-bound work on a background thread.
1>CQRSwithMediatR -> C:\Users\001733\source\projects\CQRSwithMediatR\CQRSwithMediatR\bin\Debug\net5.0\CQRSwithMediatR.dll

Reply

Osun Temidayo Olusegun says:
January 20, 2022 at 2:24 pm

Addition of Microsoft.EntityFrameworkCore.Tools as additional nuget package to add to the solution. This is to alllow Migrations. Without it Migration command like "add-migration" and "Update-database" will not be recognized

Reply

---

kamzheng says:
March 11, 2022 at 7:43 am

I am new for CQRS. I do not understand why MediatR can implement CQRS, should this be done by separated database?

Reply

SUPPORT ME :)

Search for Articles....