

Hosting

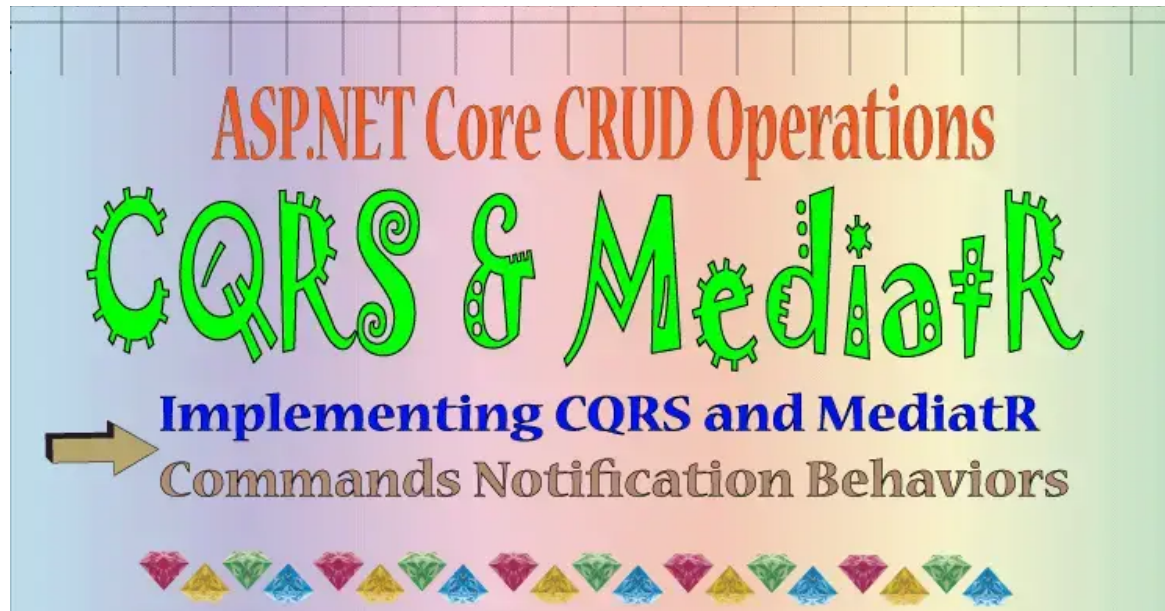


...

Implementing ASP.NET Core CRUD Operation with CQRS and MediatR Patterns

...

Updated on: December 17, 2023



In computer programming we generally use a design pattern which will help us write codes to solve problems efficiently. There are many design patterns and one of the most commonly used once are **CQRS and Mediator Patterns**. The CQRS and Mediator patterns will work together to create a loosely coupled object that will not call one another explicitly.

■ ■ ■



Download the complete source code of this tutorial from my [GitHub Repository](#).

Page Contents



- What is CQRS Pattern
- What is Mediator Pattern
- Pros of CQRS and Mediator Patterns
- Using the MediatR library
- Setting up ASP.NET Core app with MediatR package
- Entity Framework Core Configurations
- Creating a Product with CQRS
- Read Products operations with CQRS
- Updating a Product with CQRS
- Delete a Product with CQRS
- MediatR Notifications
- MediatR Behaviors

In this tutorial we will be creating an **ASP.NET Core CRUD operations** using

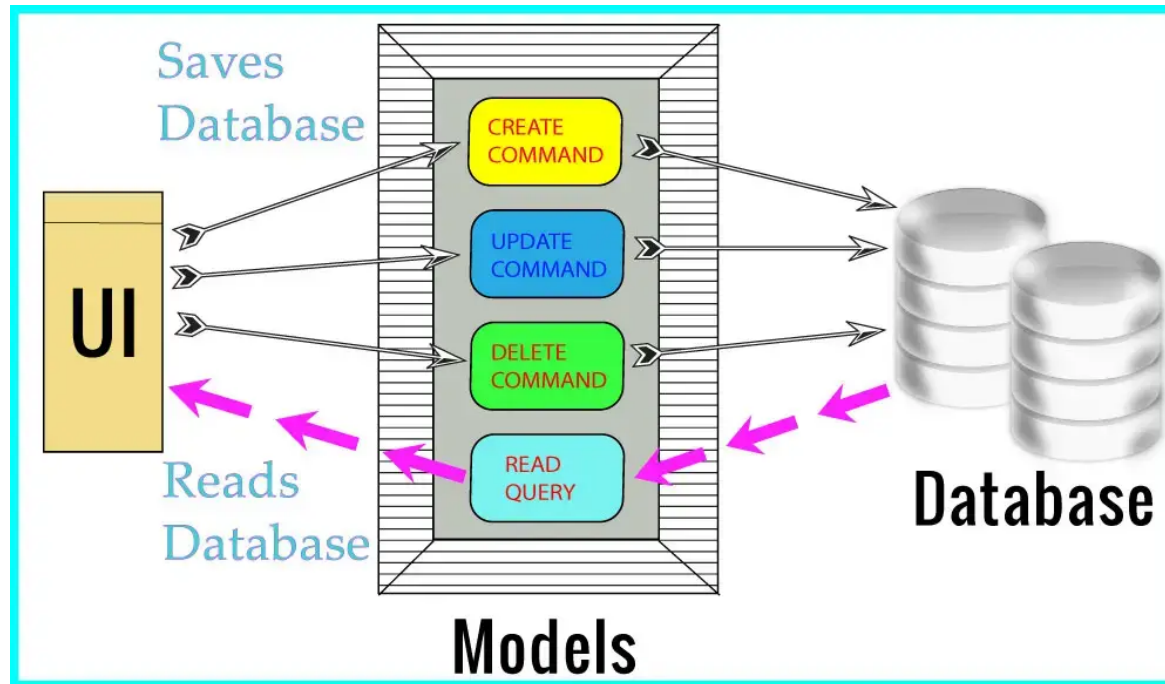
What is CQRS Pattern

CQRS stands for **Command Query Responsibility Segregation**, this pattern separates read and write operations for a data source. Each method should either be a Command or a Query but not both. In CQRS “Commands” is known for database saves and “Queries” is known for reading from the database.

The idea behind CQRS is to have different models for different operations. So for a CRUD operation we will have 4 models which are:

1. A model for creating records.
2. A model for reading records.
3. A model for updating records.
4. A model for deleting records.

See the below image where I have illustrated the CQRS architecture.



What is Mediator Pattern

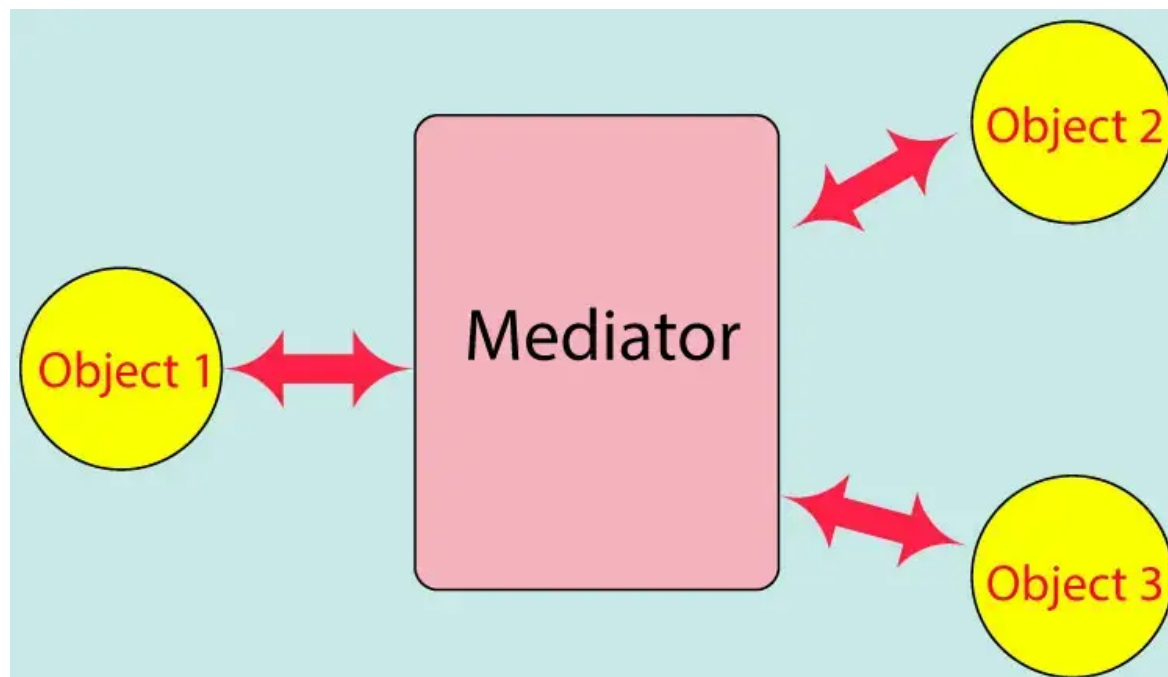
The **Mediator** pattern defines an object called Mediator, this Mediator encapsulates how a set of objects interact with one another. Mediator promotes loose coupling by keeping objects from referring to each other explicitly, and it lets you vary their interaction independently.

■ ■ ■

The Mediator pattern ensures that components don't call each other explicitly, but instead do so through calls to a mediator.

★ Their is also Repository Pattern, I implemented Generic Repository Pattern in ASP.NET Core. The tutorial is [ASP.NET Core Razor Pages : CRUD Operations with Repository Pattern and Entity Framework Core.](#)

Check the below image which illustrates the Mediator pattern:



Pros of CQRS and Mediator Patterns

Some important pros of using CQRS and Mediator patterns are not limited to:

Optimized and Scalable design

We do not have complex models but instead there are separate models per data operation. This provides highly optimized designs. We can also scale our apps easily without having to worry about dealing with the complexities.



The database operations become faster by using CQRS and Mediator. It also provides better parallel operations since there are “Commands” and “Queries”.

Loosely Coupled Architecture

Mediator pattern provided Loosely coupled architecture to our apps, it is lean, with a single responsibility, without many dependencies, allowing teams to work independently, deploy & scale independently, and increases business responsiveness.

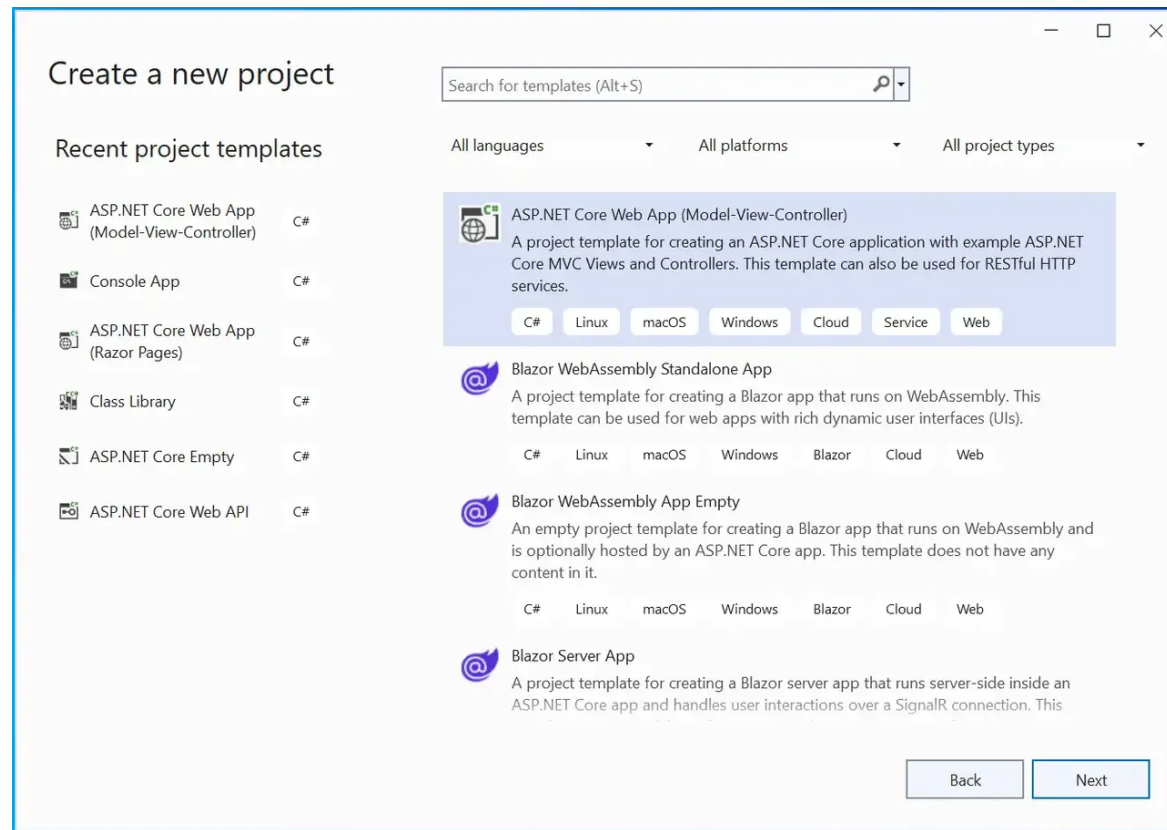
Using the MediatR library

MediatR is a .NET library which helps in building apps based on CQRS and Mediator patterns. All communication between the user interface and the data store happens via MediatR. The MediatR library is available in NuGet and we will shortly install it to our app.

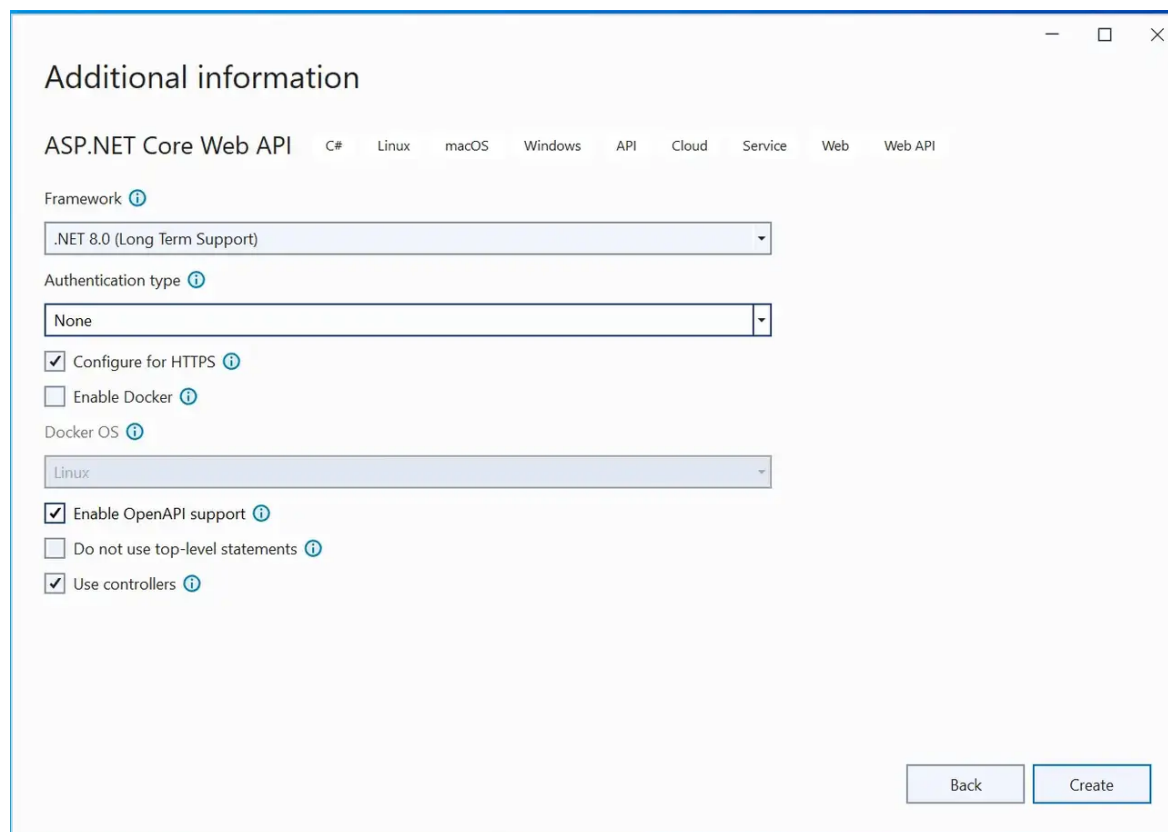


Setting up ASP.NET Core app with MediatR package

First thing you have to do is create a new ASP.NET Core Web APP (Model-View-Controller) in Visual Studio.



Name the app as CQRSMediator. Then select the latest version of .NET which is 8.0 currently.



The screenshot shows the 'Additional information' dialog box in Visual Studio. The title bar indicates it's for an 'ASP.NET Core Web API' project. The 'Framework' dropdown is set to '.NET 8.0 (Long Term Support)'. The 'Authentication type' is set to 'None'. The 'Configure for HTTPS' checkbox is checked. The 'Enable Docker' checkbox is unchecked. The 'Docker OS' dropdown is set to 'Linux'. The 'Enable OpenAPI support' checkbox is checked. The 'Do not use top-level statements' checkbox is unchecked. The 'Use controllers' checkbox is checked. At the bottom right, there are 'Back' and 'Create' buttons.

Next, select Tools ► NuGet Package Manager ► Manage NuGet Packages for Solution in your Visual Studio, and install the **MediatR** package.

Program class

Now open up the Program.cs and configured MediatR by adding the following code line:

```
builder.Services.AddMediatR(a =>  
a.RegisterServicesFromAssembly(typeof(Program).Assembly));
```

Now MediatR is configured and ready to go.



Entity Framework Core Configurations

We will now install and configure Entity Framework Core in our app. Entity Framework Core will work with CQRS and Mediator to perform database operations. So first install following packages to your app.

Microsoft.EntityFrameworkCore.SqlServer

Microsoft.EntityFrameworkCore.Design

■ ■ ■

Microsoft.EntityFrameworkCore.Tools



Next, go to appsettings.json file and add the connection string for the database. I am using LocalDb connection.

```
{
  "ConnectionStrings": {
    "DefaultConnection": "Server=
(localdb)\\mssqllocaldb;Database=ProductDB;Trusted_Connection=True;
MultipleActiveResultSets=true"
  }
}
```

Next, add the Product.cs class inside the Models folder. This class serves as the Model. I will be doing CRUD operations on the Product records. The code of the Product.cs class is given below:

```
1 namespace CQRSMediator.Models
2 {
3     public class Product
4     {
5         public int Id { get; set; }
6         public string Name { get; set; }
7         public decimal Price { get; set; }
8     }
9 }
```

Next, we need to add Db Context file for Entity Framework Core. Go to add a

```
using Microsoft.EntityFrameworkCore;

namespace CQRSMediator.Models
{
    public class ProductContext : DbContext
    {
        public ProductContext(DbContextOptions<ProductContext>
options) : base(options)
        {
        }

        public DbSet<Product> Product { get; set; }
    }
}
```

Next, we need to add Entity Framework Core to the IServiceCollection. This is done on the `Program.cs` .

```
builder.Services.AddDbContext<ProductContext>(options =>
options.UseSqlServer(builder.Configuration.GetConnectionString("Def
aultConnection")));
```

The above code says to get the Connection String that is named as `DefaultConnection`. Recall we have defined this in the `appsettings.json` file.

The final thing is to perform Entity Framework Core migrations. This is done

Tools ► NuGet Package Manager ► Packages Manager Console window of Visual studio.

```
add-migration Migration1
```

```
Update-Database
```

Once the migration completes open the View ► SQL Server Object Explorer window. Then click on (localdb)\MSSQLLocalDB and select “Refresh” you will now see ProductDB database is created.

This completes the configuration of Entity Framework Core. We are now ready to start building our CRUD operations for the Products in the Controller.

I strongly recommend you to know Onion Architecture also. Link – [Onion](#)

I will now create **CQRS Command** that will Create a New Product. So, create CQRS ► Commands folder on the root folder of the app. Then add a new class called CreateProductCommand.cs to the Commands folder. See the below image where I have shown this thing.

folder “Commands” is already created inside the CQRS folder. The Commands folder will contain classes that will deal with database saves. We already added one such class CreateProductCommand.cs.

Now also add “Queries” folder inside the “CQRS” folder. This folder will contain classes that will be dealing with database reads. We will be adding these classes during the course of the tutorial.

Now add the following code to the CreateProductCommand.cs class.

```
1  using CQRSMediator.Models;
2  using MediatR;
3
4  namespace CQRSMediator.CQRS.Commands
5  {
6      public class CreateProductCommand : IRequest<int>
7      {
8          public string Name { get; set; }
9          public decimal Price { get; set; }
10
11         public class CreateProductCommandHandler : IRequest
12         {
13             private ProductContext context;
14             public CreateProductCommandHandler(ProductCo
15             {
16                 this.context = context;
17             }
18             public async Task<int> Handle(CreateProductC
19             {
20                 var product = new Product();
```

```
24 |         context.Product.Add(product);  
25 |         await context.SaveChangesAsync();  
26 |         return product.Id;  
27 |     }  
28 | }  
29 | }  
30 | }
```

Let's explain things:

Our class CreateProductCommand.cs is implementing `IRequest<int>` interface of MediatR library. The type is `int` which means our class will be returning an `int` type value. Notice the `Handle` method where I am returning `product.Id` value which is of type `int`.

The `CreateProductCommand.cs` class has 2 properties called `Name` and `Price`. These will receive the product values from the controller and then a new product (having these values) will be created.

Next, we create an inner class which will be the Handler class and named it as `CreateProductCommandHandler`. It inherits from the `IRequestHandler<CreateProductCommand, int>` class. This handler class will handle the CQRS Command request. The `IRequestHandler` is an interface of the MediatR library, its first signature contains the class dealing with the CQRS operation i.e. `CreateProductCommand` while the second signature contains the return type which is obviously `int`.

We then implemented the `Handle` method whose work is to create i.e. save a product to the database.

Now create the controller from where the http request to create the product is made. So, inside the “Controllers” folder create a new controller by the name of ProductController.cs. To this controller’s constructor make a dependency for IMediator type.

Then add a new Http Post Type method called “Create” which will be making a call to the CQRS command class CreateProductCommand.cs which we just made.

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

Note that the Create method has the parameter of CreateProductCommand type and it is the responsibility of Mediator patter (MediatR library) to make a call to this class when `mediator.Send()` method is executed.

Testing CQRS Command

■ ■ ■

■ ■ ■

■ ■ ■

Let us now test if the CQRS is working properly. So, add a breakpoint on the Handle method of the CreateProductCommand.cs class. Now run the app on visual studio. Now open Postman to make HTTP POST type request to the app. You can also use Swagger instead of Postman.

In the Postman select:

1. "POST" option from the dropdown and enter URL – <https://localhost:44378/api/Product> The port in your case will be different.

3. On the right side drop down select “JSON” as the option.
4. On the big text box enter the product json –

```
{  
  "name": "Shirts",  
  "price": 49  
}
```

Click the send Button to make the request. See the below image where I have shown this.

Your breakpoint will hit, check the value of the “command” parameter, you will see the values which you filled in the postman for the product has

The method returns the id of the product which is created on the database.
This id will be 1 and as you keep on creating new products the id will become 2, 3, and so on.

Postman will show this id inside the response Body field.

Congrats, we just created our first Command with **CQRS and Mediator patterns**. Next, we are going to create the remaining operations – Read, Update & Delete.

★ [Do you want to learn Microservices? Then start with the first tutorial on Microservices in ASP.NET Core – First ASP.NET Core Microservice with Web API CRUD Operations on a MongoDB database \[Clean Architecture\]](#)

Read Products operations with CQRS

There will be 2 Read Operations – One for reading all the products at the same time while other for reading products by id.

Let's start with Reading all the Products

Create the Queries folder inside the “CQRS” folder. The classes that will be reading the database will be kept inside this Queries folder.

So, create a new class called GetAllProductQuery.cs to the Queries folder. The full code of this class is given below:

```
1 | using CQRSMediator.Models;  
2 | using MediatR;  
3 | using Microsoft.EntityFrameworkCore;
```

```
8      {
9      public class GetAllProductQueryHandler : IRequestHandler<GetAllProductQuery, IEnumerable<Product>>
10     {
11         private ProductContext context;
12         public GetAllProductQueryHandler(ProductContext context)
13         {
14             this.context = context;
15         }
16         public async Task<IEnumerable<Product>> Handle(GetAllProductQuery request, CancellationToken cancellationToken)
17         {
18             var productList = await context.Product.ToListAsync(cancellationToken);
19             return productList;
20         }
21     }
22 }
23 }
```

...

The GetAllProductQuery class inherits from

`IRequest<IEnumerable<Product>>` . Note that the type of `IRequest` is `IEnumerable<Product>` which means this class will be returning a list of products. See the `Handle` method which is returning this list of products.

I have defined an inner class called `GetAllProductQueryHandler` which is the handler class. It contains “Handle” method which will be returning all the products in the database.

Next, we need to add a method to the controller which will handle an HTTP Type GET request and will be calling the Query class. So, add `GetAll()` method to the `ProductController.cs` and it's code is given below.

```
[HttpGet]
public async Task<IActionResult> GetAll()
{
    return Ok(await mediator.Send(new GetAllProductQuery()));
}
```

Testing CQRS Query

Now open Postman and make HTTP GET request to the url – <https://localhost:44378/api/Product>. The port will be different in your case.



The postman will receive all the products in json and they will be displayed on the response body section. See the below image where I have shown this.

■ ■ ■



If you have more than one product in the database then the json will contain their details to, example as shown below:

```
{
  "id": 1,
  "name": "Shirts",
  "price": 49.00
},
{
  "id": 2,
  "name": "Pants",
  "price": 79.00
}
```

Reading a Product by it's Id

■ ■ ■

Let's create another query class whose work will be to fetch a product by its id. So create a new class called `GetProductByIdQuery.cs` inside the Queries folder and add the following code to it.

```
using CQRSMediator.Models;
using MediatR;
using Microsoft.EntityFrameworkCore;

namespace CQRSMediator.CQRS.Queries
```

```
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;
    }
}
}
```

The code of this class is very similar to the previous class. The `GetProductByIdQuery` class inherits from `IRequest<Product>`. The type of `IRequest` is `Product` since the class is going to return a single product only.

Now see the handler class `GetProductByIdQueryHandler's Handle()`


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

Next, go to the ProductController.cs and add the `GetById()` of type HTTP GET will be called when HTTP GET type of request is made to the URL – <https://localhost:44378/api/Product/{id}>. Here replace {id} with the id of the product like:

```
https://localhost:44378/api/Product/1  
https://localhost:44378/api/Product/2  
https://localhost:44378/api/Product/3
```

This method's code is given below:

```
[HttpGet("{id}")]  
public async Task<IActionResult> GetById(int id)  
{  
    return Ok(await mediator.Send(new GetProductByIdQuery { Id = id  
}));  
}
```

Testing with Postman

Now open Postman and send HTTP GET request to the URL – <https://localhost:44378/api/Product/1>. The Postman will show you the JSON of the product having id 1. I have shown this in the below image:

Let us now create a CQRS command to update a product. Inside the CQRS

► Commands folder, add a new class called UpdateProductCommand.cs.

It's code is given below:

```
1  using CQRSMediator.Models;
2  using MediatR;
3
4  namespace CQRSMediator.CQRS.Commands
5  {
6      public class UpdateProductCommand : IRequest<int>
7      {
8          public int Id { get; set; }
9          public string Name { get; set; }
10         public decimal Price { get; set; }
11
12         public class UpdateProductCommandHandler : IRequ
13         {
14             private ProductContext context;
15             public UpdateProductCommandHandler(ProductCo
16             {
17                 this.context = context;
18             }
19             public async Task<int> Handle(UpdateProductC
20             {
21                 var product = context.Product.Where(a =>
22
23                 if (product == null)
24                 {
25                     return default;
26                 }
27                 else
28                 {
29                     product.Name = command.Name;
30                     product.Price = command.Price;
```

```
35 |  
36 |  
37 | }
```

Notice that the `UpdateProductCommand` class implements `IRequest<int>` same as the `CreateProductCommand.cs` which was creating a new product on the database.

This class is self-explanatory and you can very well understand that it is updating a product. The product is updated inside the `Handle()` method, and this class returns the product id of the product which is updated.

■ ■ ■

Next, add `update()` method of type `Http Put` to the `ProductController`



```
[HttpPut("{id}")]  
public async Task<IActionResult> Update(int id,  
UpdateProductCommand command)  
{  
    command.Id = id;  
    return Ok(await mediator.Send(command));  
}
```

Let us now make a call to this method with Postman and update our product.

So, in Postman select PUT for the http verb and URL as <https://localhost:44378/api/Product/1>. Note that 1 at the end of the url is passing the id of the product.

Select “Body”, “raw” and “JSON” for the options just like when we created a product. For the text box enter the new values for the product in json. I am changing the name of the product to “Men’s Shirt” and price to 89 by adding the below json to the text box.

```
{  
    "name": "Men's Shirt",  
    "price": 89  
}
```

Finally click Send button and your product will get updated. See below

You can now confirm the product is updated by making GET request with Postman to the url – <https://localhost:44378/api/Product>. See the below screenshot

Postman

File Edit View Help

Home Workspaces Reports Explore

Search Postman

GET https://localhost:44378/api/Product

Save

GET https://localhost:44378/api/Product Send

Params Auth Headers (8) Body Pre-req. Tests Settings

Query Params

KEY	VALUE	DESCRIPTION
Key	Value	Description

Body

200 OK 703 ms 227 B Save Response

Pretty Raw Preview Visualize JSON

```
1 {  
2   {  
3     "id": 1,  
4     "name": "Men's Shirt",  
5     "price": 89.00  
6   }  
7 }
```

Product is updated

Find and Replace Console

Delete a Product with CQRS

■ ■ ■

The final CRUD operation is to delete a product with CQRS command. So, create a new class called DeleteProductByIdCommand.cs inside the CQRS

► Commands folder. Add the following code to this class:

```
1 using CQRSMediator.Models;
2 using MediatR;
3 using Microsoft.EntityFrameworkCore;
4
5 namespace CQRSMediator.CQRS.Commands
6 {
7     public class DeleteProductByIdCommand : IRequest<int>
8     {
9         public int Id { get; set; }
10        public class DeleteProductByIdCommandHandler : I
11    }
```



```
15         this.context = context;
16     }
17     public async Task<int> Handle(DeleteProductB
18     {
19         var product = await context.Product.Where
20         context.Product.Remove(product);
21         await context.SaveChangesAsync();
22         return product.Id;
23     }
24 }
25 }
26 }
```

The class implements `IRequest<int>`, `int` as the type signifies that it will return an `int` value. Confirm this thing as `Handle` method is returning the product id at the very end.

```
return product.Id;
```

The handle method does the product deletion whose id the controller sends it to. The deletion of the product is done by Entity Framework Core.





Next, add the delete method to the ProductController.cs.

```
[HttpDelete("{id}")]
public async Task<IActionResult> Delete(int id)
{
    return Ok(await mediator.Send(new DeleteProductByIdCommand { Id
= id }));
}
```

So, it's time to test the delete functionality with Postman. So in Postman make Delete type of request to the URL –

<https://localhost:44378/api/Product/1>. This will delete the product with id 1 from the database. The screenshot of Postman is shown below:

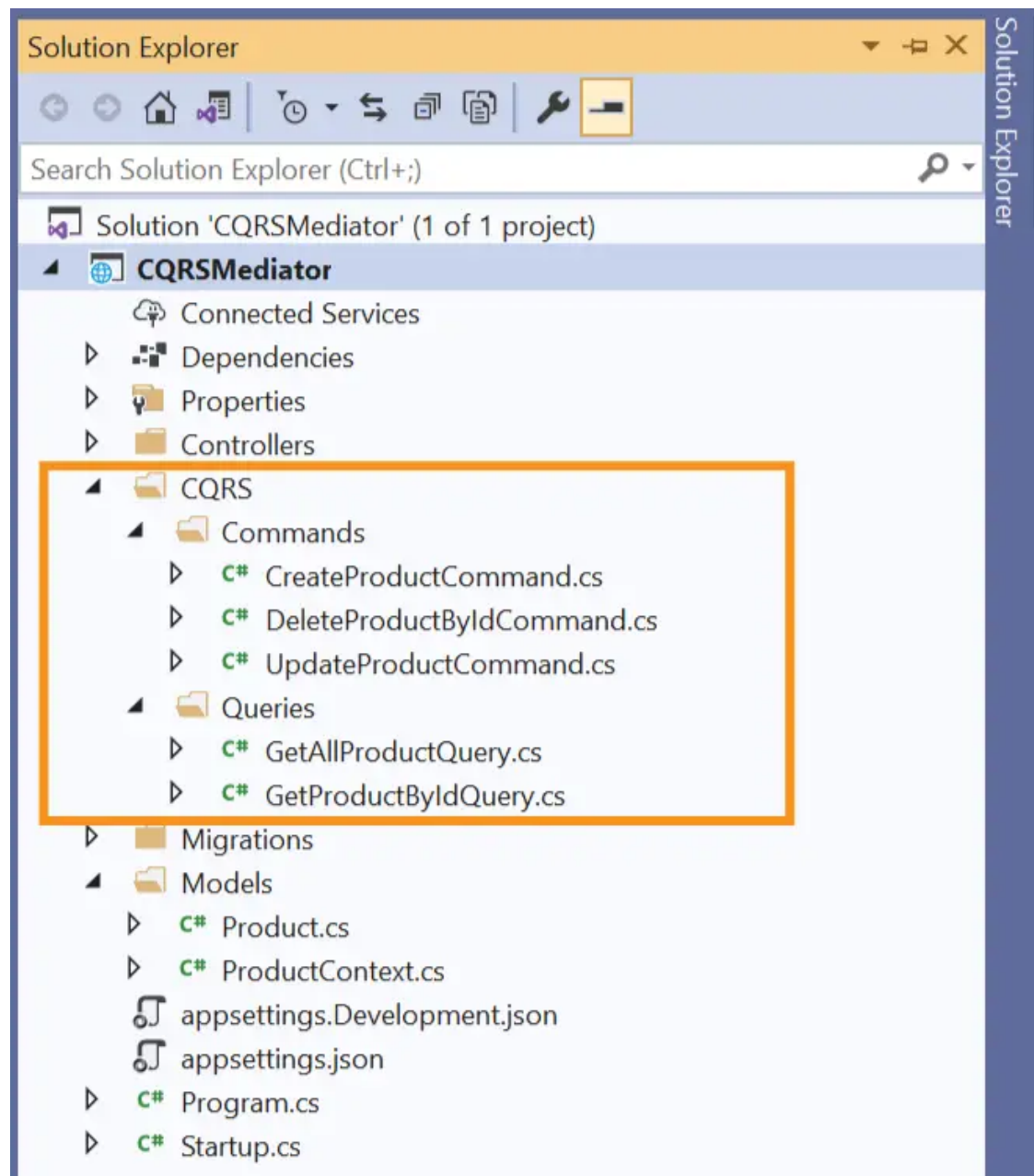
■ ■ ■



Now as usual make GET request to confirm the product is deleted. This time you will get empty json which tells the product is indeed deleted.

Congratulations, we successfully created **CRUD operations with CQRS and Mediatry patterns in ASP.NET Core using Entity Framework Core.**

So now we have 3 classes inside the Commands folder and 2 classes inside the Queries folder. Check the below image:



Till now we have seen a single request being handled by a single handler.
Example – Delete request of a Product is handled by the Handle method of DeleteProductByIdCommandHandler.cs class.

Now we will see how a single request will be handled not by one but by multiple handlers. This is done by **MediatR Notifications**.

For example – whenever a product is deleted, we will do 2 things:

2. Sending SMS to Sales team that the Product is deleted, so they should probable create a new product in it's place.

Let's us create MediatR Notifications for it.

First create a new folder called Notifications on the root of the app. Then inside it, create a new class called DeleteProductNotification.cs with the code as given below:

```
1  using MediatR;
2
3  namespace CQRSMediator.Notifications
4  {
5      public class DeleteProductNotification : INotificati
6      {
7          public int ProductId { get; set; }
8      }
9
10     public class EmailHandler : INotificationHandler<Del
11     {
12         public Task Handle(DeleteProductNotification not
13         {
14             int id = notification.ProductId;
15             // send email to customers
16             return Task.CompletedTask;
17         }
18     }
19
20     public class SMSHandler : INotificationHandler<Delet
21     {
22         public Task Handle(DeleteProductNotification not
23         {
```



```
28 |  
29 }
```

This is the location of the class on the app directory.



Let's understand what we are doing in this class:

contains a single property called `ProductId`.

2. Next, we created 2 handler classes called 'EmailHandler' and 'SMSHandler'. Both of them implements `INotificationHandler` with a type of `DeleteProductNotification`. This signifies that they will handle the event of `DeleteProductNotification` type.

3. The handlers does the work they are assigned to like sending email or SMS. Also see the 2 `Handle` methods, they have `DeleteProductNotification` as the first parameter and through which they get the property values. Example: getting the `ProductId` value as shown below.

```
int id = notification.ProductId;
```

Trigger the MediatR Notification

Let's trigger the notification which is done through the `Publish` method. So go to the `ProductController.cs` and add the `Publish` method to the `Delete` method as shown in highlighted code below:

```
1 | [HttpDelete("{id}")]
2 | public async Task<IActionResult> Delete(int id)
3 | {
4 |     await mediator.Publish(new Notifications.DeleteProduc
5 |     return Ok(await mediator.Send(new DeleteProductByIdCo
6 | }
```

loosely coupled with the publish method. This is a great thing as we can extend more handler classes without any need to modify the publish method on the Delete method of the Product Controller class.

MediatR Behaviors

MediatR Behaviors are very similar to middlewares in ASP.NET Core. They accept a request, perform some action, then (optionally) pass along the request. The good thing about Behaviors is that we can put a logic in them instead of repeating it again and again in our app.

Let us create a MediatR behavior which will do logging for us. So create a new folder called Behaviors on the root of the app and add a new class called LoggingBehavior.cs to it.

The full code of this class is given below:

```
1 | using MediatR;  
2 |  
3 | namespace CQRSMediator.Behaviors  
4 | {
```

```

9      {
10         this.logger = logger;
11     }
12
13     public async Task<TResponse> Handle(TRequest req
14     {
15         logger.LogInformation($"Before {typeof(TRequest)} request received");
16         var response = await next();
17         logger.LogInformation($"After {typeof(TResponse)} response received");
18         return response;
19     }
20 }
21 }

```

Important things:

1. The MediatR Behavior class must have two type parameters TRequest and TResponse, and it should implement the `IPipelineBehavior<TRequest, TResponse>` interface.
2. In the Handle method we logged the information before and after any request. This logging handler can then be applied to any request, and will log output before and after it is handled.
3. The `next()` moves to the next action in the middleware pipeline.

Registering Behavior in the Program class

Go to the Program.cs and register this Behavior as shown below:

```

1 | builder.Services.AddSingleton(typeof(IPipelineBehavior<, >

```

To test the Behavior make a read product request with Postman. Like before make an HTTP GET request to the url – <https://localhost:44378/api/Product>. The port will be different in your case.

Next on Visual Studio open Output window whose path is View menu then Output. You will see 2 logs:

```
Information: Befor GetAllProductQuery
```

```
Information: After IEnumerable`1
```

I have shown these logs in the below given image:

Great! This is the logging output before and after our GetAllProductQuery query handler was invoked. Because of the loosely coupled architecture of CQRS we hooked the behaviour without touching any other feature of the app.


Conclusion

In this tutorial we first understood how **CQRS and MediatR** works and later created a full CRUD operation in ASP.NET Core app. We also created **MediatR Notifications and Behaviors**. This must be a good starting point for you. I hope you like this tutorial so kindly share it on reddit, facebook, twitter



SHARE THIS ARTICLE

ABOUT THE AUTHOR

I am Yogi S. I write DOT NET articles on my sites hosting.work and yogihosting.com. You can connect with me on [Twitter](#). I hope my articles are helping you in some way or the other, if you like my articles consider buying me a coffee -  [Buy Me A Coffee](#)

Leave a Reply

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

*

Comment *

Name *

Email *

Post Comment

Related Posts based on your interest

[Necessary .NET](#)

[Security features for
securing your web
applications](#)

[Automated UI Testing
with Selenium in
ASP.NET Core](#)

[How to perform
Integration Testing in
ASP.NET Core](#)

Hello Everyone,

Welcome to
Hosting.Work - A
Programming

Tutorial Website. It covers
ASP.NET Core topics. I hope
you enjoy reading it.

SUBSCRIBE TO NEWSLETTER

Enter your email address to subscribe to
this blog and receive notifications of new
posts by email

Email Address *

Subscribe



Subscribe to our Newsletter and receive "1 email per week" for the article published on the site. No spamming.

Copyright ©2024