.NET Zero to Hero Series is now LIVE! **JOIN** 🚀

12 min read Updated on May 20, 2024

Validation with MediatR Pipeline and FluentValidation

Mukesh Murugan
@iammukeshm

#dotnet

Validation is super essential for your application to ensure the sanity of the incoming requests. In this article, we will learn a clean way to implement validation in your ASP.NET Core projects using the MediatR and FluentValidation libraries. This is powerful in projects that has already adapted the CQRS / CQS patterns using MediatR.

Join the .NET Series! 😎

This article is part of my ongoing .NET Zero to Hero Series! To get notified whenever I post new content in this series, subscribe to my .NET Newsletter.

The end game of this series is to build scalable .NET applications with Clean Architecture! I am building the content one by one to reach to the finale - *Practical Clean Architecture with ASP.NET Core!* Join Now to BOOST your .NET Skills 🚀

Essentially, this article builds up on the knowledge we already gained from the previous articles of the .NET Series.

> Global Exception Handling in ASP.NET Core - IExceptionHandler in .NET 8 - Read
> How to use FluentValidation in ASP.NET Core - Super Powerful Validations - Read
> CQRS and MediatR in ASP.NET Core - Building Scalable Systems - Read

Thus, it is highly recommended that you go through the above-mentioned articles, so that you can grasp the concepts mentioned in this article much easily.

Now, we will learn about MediatR Pipeline Behavior in ASP.NET Core, the idea of Pipelines, and How to intersect the pipeline to add custom service, and most importantly the validation behavior using FluentValidation.

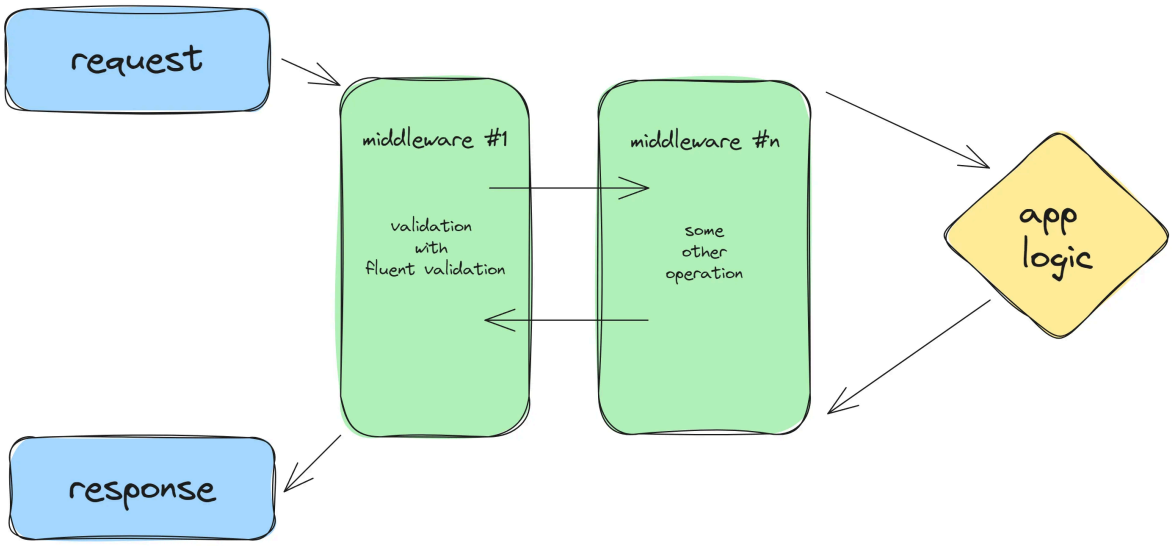
As you know, MediatR is a tool/library which essentially can make your Controllers / API Endpoints thin and decouples the functionalities to a more message-driven approach. This is used to implement the CQRS Pattern in ASP.NET Core Applications, which is Commands and Queries Responsibility Segregation.

Understanding the MediatR Pipeline

What happens internally when you send a request to any application?

In ASP.NET Core, requests and responses travel through pipelines. When a request is made, it passes through a pipeline to the application, where the requested operation is performed. After processing, the application sends the response back through the pipeline to the user. This means the pipelines handle and are aware of both the request and the response. Understanding these pipelines is crucial for learning about middleware in ASP.NET Core.

Here is how requests would flow in the application pipeline.



Let's say you want to validate the incoming requests with certain business rules. How would you do it?

Typically, you might write validation logic to execute after the request has reached the application, at the end of the pipeline. This means the request is validated only after it has been processed by the application. While this approach works, consider this: why wait to validate the request until it reaches the application? Instead, you can validate incoming requests earlier in the pipeline, before they interact with any application logic. This proactive approach can streamline processing and improve efficiency. Makes sense?

A better approach is to integrate your validation logic directly into the pipeline. This way, when a user sends a request, it first goes through the pipeline where the validation logic is applied. If the request is valid, it proceeds to the application logic; if not, a validation exception is thrown. This method enhances efficiency by preventing invalid data from reaching the application.

This approach isn't limited to validations. It can also be applied to other operations like logging, performance tracking, and more. There's a lot of room for creativity in how you utilize the pipeline.

MediatR Pipeline Behavior

Coming to MediatR, it takes a more pipeline kind of approach where your queries, commands and responses flow through a pipeline setup by MediatR.

We know that these MediatR queries/commands are like the first contact within our application, so why not attach some services into the MediatR Pipeline?

By doing this, we will be able to execute services/logics like validations even before the Command or Query Handlers know about it. This way, we will be sending only valid requests to our actual application code. Logging and Validation using this MediatR Pipeline Behavior are some common implementations.

Let's Code

Let's jump straight into some code. For this article, I will be re-using the ASP.NET Core 8 CRUD Web API that we had built in the previous demonstration, where we implemented CQRS Pattern using the MediatR library. Thus, we will have a good starting point with CRUD and CQRS already in place. You can fork the starting code from here.

I assume that you have the concepts clear. Here is what we are going to build. In our CQRS based ASP.NET Core Web API Solution, we are going to add validation using FluentValidation within the MediatR Pipeline. We will also enhance this by adding Exception Handler, so that all the validation exceptions thrown within the pipeline by FluentValidation will be elegantly caught by our application using the `IExceptionHandler` .

Thus, any `Command` or `Query` would be validated even before the request hits the application logic. Also, we will log every request and response that goes through the MediatR Pipeline.

Request Response Logging Behavior with MediatR

First up, to understand MediatR Pipeline Behavior, we will build a simple behavior that can log the request and response of incoming MediatR requests in our API.

Create a new folder named `Behaviors` and add the following class, `RequestResponseLoggingBehavior` .

```
public class RequestResponseLoggingBehavior<TRequest, TResponse>(ILogger<RequestResponseL
    : IPipelineBehavior<TRequest, TResponse>
    where TRequest : class
{
    public async Task<TResponse> Handle(TRequest request, RequestHandlerDelegate<TRespons
```

```
        {
            var correlationId = Guid.NewGuid();

            // Request Logging
            // Serialize the request
            var requestJson = JsonSerializer.Serialize(request);
            // Log the serialized request
            logger.LogInformation("Handling request {CorrelationID}: {Request}", correlationI

            // Response Logging
            var response = await next();
            // Serialize the request
            var responseJson = JsonSerializer.Serialize(response);
            // Log the serialized request
            logger.LogInformation("Response for {Correlation}: {Response}", correlationId, re

            // Return response
            return response;
        }
    }
```

In the above implementation, we inherit from `IPipelineBehavior` and implement the `Handle` method. Note that, using this interface, we get access to both the request and response within the MediatR Pipeline.

Within the `Handle` method,

1. We first set a random GUID as the Correlation ID. This is to ensure both the request and responses can be tracked using this ID.
2. Next, we serialize the incoming request and log the resulting string along with the Correlation ID.
3. `var response = await next();` ensures that the request is processed and response is received. We then log the response as well.

Once that is written, you will have to register this new behavior within the MediatR pipeline. Navigate to the `Program.cs` file and add the following line to your MediatR configuration.

```
builder.Services.AddMediatR(cfg =>
{
    cfg.RegisterServicesFromAssembly(Assembly.GetExecutingAssembly());
```

```
    cfg.AddOpenBehavior(typeof(RequestResponseLoggingBehavior<,>));
});
```

Let's see this behavior in action. Build your application, and hit an Endpoint using Swagger. I am hitting the `/products` GET endpoint which would return me a list of available products.

```
info: MediatRPipelineFluentValidation.Behaviors.RequestResponseLoggingBehavior[0]
      Handling request eda599e1-51d9-4dc7-b55b-3aca34bfe07a: {}
info: MediatRPipelineFluentValidation.Behaviors.RequestResponseLoggingBehavior[0]
      Response for eda599e1-51d9-4dc7-b55b-3aca34bfe07a: [{"Id":"d2330a35-5d84-4368-aa5d-2b57ec8f5649","Name":
"iPhone 15 Pro","Description":"Apple\u0027s latest flagship smartphone with a ProMotion display and improved c
ameras","Price":999.99},{"Id":"041f2e90-9ee5-49e3-8cdc-8c232675a2bc","Name":"Dell XPS 15","Description":"Dell\
u0027s high-performance laptop with a 4K InfinityEdge display","Price":1899.99},{"Id":"fc0a7c1d-8ad0-47d6-b6ff
-84eb4e43c411","Name":"Sony WH-1000XM4","Description":"Sony\u0027s top-of-the-line wireless noise-canceling he
adphones","Price":349.99}]
```

In the above logs, both the requests and responses are logged as expected. Next, we will try validating the incoming requests with FluentValidation.

FluentValidation with MediatR

Previously in another article, we learned about validating requests with FluentValidation. But the problem was that we had to perform this validation in each and every service / command / query handlers, which is not scalable for the long run. Instead, we need an approach where every request object that passes through MediatR can be validated somewhere centrally. This is one of the most common use cases for MediatR pipeline behavior. Basically, you would want to free your command handlers from any of the validation logic.

So, using this approach,

1. Your entities no longer have validation rules embedded within them, as they will be maintained separately using FluentValidation Rules.
2. Your handlers would not need to have code to validate the incoming requests. The requests would be validated even before hitting your actual application logic.

For validating our MediatR requests, we will use the Fluent Validation Library.

> *To learn more about FluentValidation in ASP.NET Core, check out my previous post - How to use FluentValidation in ASP.NET Core - Super Powerful Validations*

Here is how we will implement this. We have an API endpoint that is responsible for creating a product in the database from the request object that includes the product name, description, and

so on. But we would want to validate this request in the pipeline itself.

First, install the required FluentValidation Packages.

```
Install-Package FluentValidation
Install-Package FluentValidation.DependencyInjectionExtensions
```

Since we already have the Feature folders setup, let's add the validators where it belongs. Under the `Features/Products/Commands/Create` folder, add a new class called `CreateProductCommandValidator`.

```
public class CreateProductCommandValidator : AbstractValidator<CreateProductCommand>
{
    public CreateProductCommandValidator()
    {
        RuleFor(p => p.Name).NotEmpty().MinimumLength(4);
        RuleFor(p => p.Price).GreaterThan(0);
    }
}
```

Here, we are going to validate the incoming Name and Price properties. A validation exception would be thrown if the name is empty, or less the 4 characters, or if the product is given an invalid price. You could take this a step further by injecting a `DBContext` to this constructor and checking if the product name already exists.

We have n number of similar validators for each command and query. This helps keep the code well-organized and easy to test. This is the elegance of feature folders. Every piece of code related to a feature will be within the same folder, very easy to navigate.

Before continuing, let's register this validator with the ASP.NET Core DI Container. Navigate to `Program.cs` and add in the following line.

```
builder.Services.AddValidatorsFromAssembly(Assembly.GetExecutingAssembly());
```

This essentially registers all the validators that are available within the current assembly.

Now that we have our validator set up, Let's add the MediatR Pipeline Behavior. Under the `Behaviors` folder, add a new class `ValidationBehavior`

```
public class ValidationBehavior<TRequest, TResponse>(IEnumerable<IValidator<TRequest>> va
    : IPipelineBehavior<TRequest, TResponse>
    where TRequest : class
{
    public async Task<TResponse> Handle(TRequest request, RequestHandlerDelegate<TRespons
    {
        ArgumentNullException.ThrowIfNull(next);

        if (validators.Any())
        {
            var context = new ValidationContext<TRequest>(request);

            var validationResults = await Task.WhenAll(
                validators.Select(v =>
                    v.ValidateAsync(context, cancellationToken))).ConfigureAwait(false);

            var failures = validationResults
                .Where(r => r.Errors.Count > 0)
                .SelectMany(r => r.Errors)
                .ToList();

            if (failures.Count > 0)
                throw new FluentValidation.ValidationException(failures);
        }
        return await next().ConfigureAwait(false);
    }
}
```

Similar to the Logging Behavior, we will implement the Handle method that takes in the incoming request.

1. First we check if there are any validators attached to the incoming request.
2. If available, we create a new validation context using the request, and validate it. This would return the `ValidationResults` array.
3. From this array of results, we filter out the failures (where the error count is greater than 0).

4. And finally, we throw this as `FluentValidation.ValidationException` and pass the errored out messages.

With that done, register this Pipeline Behavior in the ASP.NET Core Service Container. Again, go back to `Program.cs` and add this.

```
builder.Services.AddMediatR(cfg =>
{
    cfg.RegisterServicesFromAssembly(Assembly.GetExecutingAssembly());
    cfg.AddOpenBehavior(typeof(RequestResponseLoggingBehavior<,>));
    cfg.AddOpenBehavior(typeof(ValidationBehavior<,>));
});
```

> *Note that the order by which your register your Behaviors is crucial. According to the current setup, if a validation error occurs,*
>
> 1. *Request will be logged.*
> 2. *Validation exception will be thrown.*
> 3. *Response will not be logged, the flow will break. This is because an error has already been through even before we could log the response.*

Handling Validation Exceptions in Pipeline

We will add Global Exception Handling to the mix too! This way, whenever an exception of type `FluentValidation.ValidationException` is thrown in the pipeline, we can elegantly catch it our Exception Handler, and return a response of type `ProblemDetails`.

> *To read about various ways to implement Exception Handling in ASP.NET Core application, refer to this article : Global Exception Handling in ASP.NET Core. Using `IExceptionHandler` is the recommended way!*

Create a new folder named `Exceptions` and add in the following class.

```
public class GlobalExceptionHandler(ILogger<GlobalExceptionHandler> logger) : IExceptionH
{
    public async ValueTask<bool> TryHandleAsync(HttpContext httpContext, Exception except
    {
        var problemDetails = new ProblemDetails();
        problemDetails.Instance = httpContext.Request.Path;

        if (exception is FluentValidation.ValidationException fluentException)
        {
            problemDetails.Title = "one or more validation errors occurred.";
            problemDetails.Type = "https://tools.ietf.org/html/rfc7231#section-6.5.1";
            httpContext.Response.StatusCode = StatusCodes.Status400BadRequest;
            List<string> validationErrors = new List<string>();
            foreach (var error in fluentException.Errors)
            {
                validationErrors.Add(error.ErrorMessage);
            }
            problemDetails.Extensions.Add("errors", validationErrors);
        }

        else
        {
            problemDetails.Title = exception.Message;
        }

        logger.LogError("{ProblemDetailsTitle}", problemDetails.Title);

        problemDetails.Status = httpContext.Response.StatusCode;
        await httpContext.Response.WriteAsJsonAsync(problemDetails, cancellationToken).Co
        return true;
    }
}
```

This would be pretty straight forward if you have already gone through my Exception Handling article.

Here, we are just creating an object of type `ProblemDetails` and adding the current path as the instance property of the object.

If the current exception is of type `FluentValidation.ValidationException`, we will set the properties of the problem details object, append the list of errors that was returned as part of our validation behavior, set the status code to 400 Bad Request, and return.

To add this handler to our application, let's register it.

```
builder.Services.AddExceptionHandler<GlobalExceptionHandler>();
builder.Services.AddProblemDetails();
.
.
app.UseExceptionHandler();
```

Time to test our implementation! Build and run your application. Since we have added validators to the `CreateProductCommand` only, let's test out the POST endpoint via Swagger.

Here is the request I will be sending to our API.

```
{
  "name": "ab",
  "description": "",
  "price": 0
}
```

If you can remember our validator, the rule that we set was that the name should be greater than 4 character, and the price should be greater than 0. The above request clearly violates these rules, and thus we except errors to be thrown.

Here is the response from API.

```
{
  "type": "https://tools.ietf.org/html/rfc7231#section-6.5.1",
  "title": "one or more validation errors occurred.",
  "status": 400,
  "instance": "/products",
  "errors": [
    "The length of 'Name' must be at least 4 characters. You entered 2 characters.",
    "'Price' must be greater than '0'."
  ]
}
```

As excepted, it has returned a response of type `ProblemDetails` with a list of validation failures. You can see how powerful it is combine CQRS, MediatR Pipeline Behavior, FluentValidation, and IExceptionHandler. This is the pattern I use in almost all the .NET projects, and find it super clean and helpful.

The MediatR Pipeline Behavior allows you to neatly plug in some validation logics within the MediatR pipeline, which can be handled gracefully using the `IExceptionHandler`. Highly recommended approach!

I hope you thoroughly enjoyed this article and found it helpful! For the next article of the series, we will start exploring Caching Techniques in ASP.NET Core.

The end game of this series is to build a scalable Clean Architecture .NET application! I am building the content one by one to reach to the finale - Practical Clean Architecture with ASP.NET Core!

You can find the code implementations of each of article in the series within the .NET Zero to Hero Series Repository over at GitHub. Star this repository to reach to more folks! Thanks.

Source Code ✌️

Grab the source code of the entire implementation by clicking here. Do Follow me on GitHub .

Support ❤️

If you have enjoyed my content and code, do support me by buying a couple of coffees. This will enable me to dedicate more time to research and create new content. Cheers!

Share this Article

Share this article with your network to help others!

What's your Feedback?

Do let me know your thoughts around this article.

0 reactions

5 comments – *powered by giscus*

Oldest Newest

Mukesh's .NET Newsletter

Join **5,000+ Engineers** to Boost your .NET Skills. I have started a .NET Zero to Hero Series that covers everything from the basics to advanced topics to help you with your .NET Journey! You will receive 1 Awesome Email every week.

Subscribe

codewithmukesh

web development simplified!

© 2024 Mukesh Murugan