



codewithmukesh

Global Exception Handling in ASP.NET Core - IExceptionHandler in .NET 8 [Recommended]



Mukesh Murugan
@iammukeshm

#dotnet

Exception Handling is vital for applications of all types and traffic volumes. If exceptions are not handled well within the application, it may break the entire application or even lead to data loss. In ASP.NET Core, there are multiple ways that one can handle exceptions effectively. I have included this article as part of my ongoing [.NET Zero to Hero Series](#) as this is an important aspect for developers learning about building production-ready .NET applications.

In this article, we will focus on handling exceptions globally in an ASP.NET Core application, so that there is a central error-handling mechanism throughout your applications. This makes things quite easy and manageable.

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 🚀

Exceptions in .NET

Exceptions in .NET are objects that inherit from the `System.Exception` base class and can be thrown from the part of your code base wherever the problem has occurred.

Some common exceptions that you often encounter while working with .NET are `NullReferenceException`, `ArgumentNullException`, `IndexOutOfRangeException`, etc.

Getting started with Error Handling in ASP.NET Core

For this demonstration, we will be working on a new ASP.NET Core Web API (.NET 8) project and Visual Studio 2022 as my default IDE.

Here are a few ways how you can handle exceptions /errors in your ASP.NET Core Applications:

Traditional Try Catch blocks

Built-In Exception Handling Middleware

Custom Middleware to Handle Exceptions

All New `IExceptionHandler` [Highly Recommended] - Starting from .NET 8

We will go through each of these mechanisms, but our main focus and recommendation would be to use the `IExceptionHandler` feature that got introduced from .NET 8!

Try Catch Block

The try-catch block is our go-to approach when it comes to quick exception handling. Let's see a code snippet that demonstrates the same.

```
[HttpGet]  
public IActionResult Get()
```

```
{
    try
    {
        var data = GetData(); //Assume you get some data here which is also likely to thr
        return Ok(data);
    }
    catch (Exception ex)
    {
        _logger.LogError(ex.Message);
        return StatusCode(500);
    }
}
```

Here is a basic implementation that we are all used to, yeah? Assume, the method `GetData()` is a service call that is also prone to exceptions due to certain external factors. The thrown exception is caught by the catch block whose responsibility is to log the error to the console and return a status code of 500 Internal Server Error in this scenario.

For Logging, it's recommended to use `Serilog`. As part of the .NET Series, we have already covered in-depth structured logging in ASP.NET Core using Serilog. [Read here](#) for more.

Let's say that there was an exception during the execution of the `Get()` method. The below code is the exception that gets triggered.

```
throw new Exception("An error occurred...");
```

Here is what you would be seeing on Swagger.

Request URL	
https://localhost:5001/WeatherForecast	
Server response	
Code	Details
500 <small>undocumented</small>	Error: Response headers content-length: 0 date: Sun09 May 2021 15:37:43 GMT server: Kestrel

The Console may get you a bit more details on the exception, like the line number and other trace logs.

```

C:\Users\iammu\source\repos\GlobalExceptionHandler\GlobalExceptionHandler.WebApi\bin\Debug\net5.0\GlobalExceptionHandler.WebApi.exe
cation.
System.Exception: An error occurred...
   at GlobalExceptionHandler.WebApi.Controllers.WeatherForecastController.Get() in C:\Users\iammu\source\repos\GlobalExceptionHandler\GlobalExceptionHandler.WebApi\Controllers\WeatherForecastController.cs:line 29
   at lambda_method2(Closure , Object , Object[]) 
   at Microsoft.AspNetCore.Mvc.Infrastructure.ActionMethodExecutor.SyncActionResultExecutor.Execute(IActionResultTypeMapper mapper, ObjectMethodExecutor executor, Object controller, Object[] arguments)
   at Microsoft.AspNetCore.Mvc.Infrastructure.ControllerActionInvoker.InvokeActionMethodAsync()
   at Microsoft.AspNetCore.Mvc.Infrastructure.ControllerActionInvoker.Next(State& next, Scope& scope, Object& state, Boolean& isCompleted)
   at Microsoft.AspNetCore.Mvc.Infrastructure.ControllerActionInvoker.InvokeNextActionFilterAsync()
   --- End of stack trace from previous location ---
   at Microsoft.AspNetCore.Mvc.Infrastructure.ControllerActionInvoker.Rethrow(ActionExecutedContextSealed context)
   at Microsoft.AspNetCore.Mvc.Infrastructure.ControllerActionInvoker.Next(State& next, Scope& scope, Object& state, Boolean& isCompleted)
   at Microsoft.AspNetCore.Mvc.Infrastructure.ControllerActionInvoker.InvokeInnerFilterAsync()
   --- End of stack trace from previous location ---
   at Microsoft.AspNetCore.Mvc.Infrastructure.ResourceInvoker.<InvokeFilterPipelineAsync>g__Awaited|19_0(ResourceInvoker invoker, Task lastTask, State next, Scope scope, Object state, Boolean isCompleted)
   at Microsoft.AspNetCore.Mvc.Infrastructure.ResourceInvoker.<InvokeAsync>g__Awaited|17_0(ResourceInvoker invoker, Task task, IDisposable scope)
   at Microsoft.AspNetCore.Routing.EndpointMiddleware.<Invoke>g__AwaitRequestTask|6_0(Endpoint endpoint, Task requestTask, ILogger logger)
   at Microsoft.AspNetCore.Authorization.AuthorizationMiddleware.Invoke(HttpContext context)
   at Swashbuckle.AspNetCore.SwaggerUI.SwaggerUIMiddleware.Invoke(HttpContext httpContext)
   at Swashbuckle.AspNetCore.Swagger.SwaggerMiddleware.Invoke(HttpContext httpContext, ISwaggerProvider swaggerProvider)
   at Microsoft.AspNetCore.Server.Kestrel.Core.Internal.Http.HttpProtocol.ProcessRequests[TContext](IHttpApplication application)
  
```

Although this is a simple way of handling exceptions in ASP.NET Core applications, this can also increase the lines of code of our application. Yes, you could have this approach for very simple and small POC applications. Imagine having to write the try-catch block in every controller's action and other service methods. Pretty repetitive and not feasible, yeah?

It would be ideal if there was a way to handle all the exceptions centrally in one location, right? In the next sections, we will see 2 such approaches that can drastically improve our exception-handling mechanism by isolating all the handling logic to a single area. This not only gives a better codebase but also a more controlled application with even lesser exception handling concerns.

Default Exception Handling Middleware in .NET - UseExceptionHandler

To make things easier, `UseExceptionHandler` Middleware comes out of the box with ASP.NET Core applications. This when configured in the `Program.cs`, adds a middleware to the pipeline of the application that will catch any exceptions in and out of the application. A very straightforward implementation of middleware.

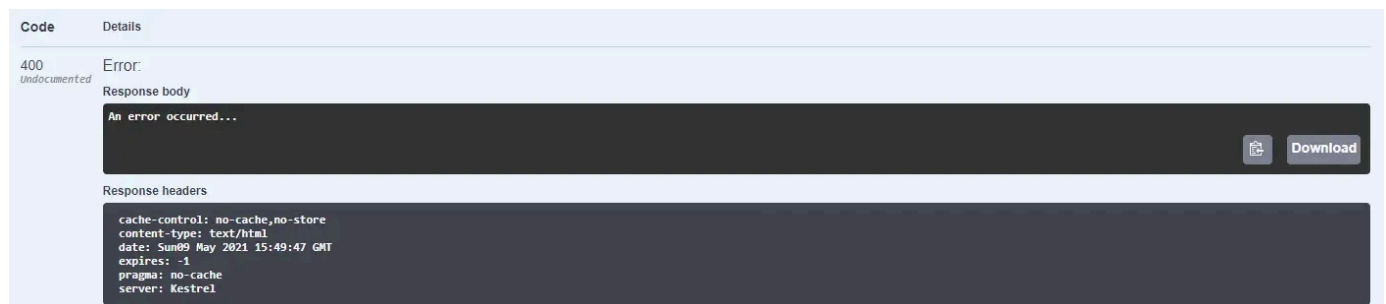
Let's see how `UseExceptionHandler` can be configured. Open up the `Program.cs` class of your ASP.NET Core application and configure the following.

```
app.UseExceptionHandler(options =>
```

```
{
    options.Run(async context =>
    {
        context.Response.StatusCode = (int)HttpStatusCode.BadRequest;
        context.Response.ContentType = "application/json";
        var exception = context.Features.Get<IExceptionHandlerFeature>();
        if (exception != null)
        {
            var message = $"{exception.Error.Message}";
            await context.Response.WriteAsync(message).ConfigureAwait(false);
        }
    });
});
```

This is a very basic setup & usage of `UseExceptionHandler` Middleware in your ASP.NET Core applications. So, whenever there is an exception that is detected within the Pipeline of the application, the control falls back to this middleware, which in return will send a custom response to the request sender.

In this case, a status code of 400 Bad Request is sent along with the Message content of the original exception which in our scenario is 'An error occurred...'. Pretty straightforward, yeah? Here is how the exception is displayed on Swagger.



The screenshot shows the Swagger UI interface for a 400 Bad Request error. The 'Code' tab is selected, displaying the status code '400' and the label 'undocumented'. The 'Details' tab shows the error message 'Error: An error occurred...' in the 'Response body' section. The 'Response headers' section lists the following headers: 'cache-control: no-cache, no-store', 'content-type: text/html', 'date: Sun09 May 2021 15:49:47 GMT', 'expires: -1', 'pragma: no-cache', and 'server: Kestrel'. A 'Download' button is visible next to the response body.

Now, whenever there is an exception thrown in any part of the application, this middleware catches it and throws the required exception back to the client. Much cleaned-up code, yeah? But there are still more ways to make this better, by miles.

Custom Exceptions

It's important and cleaner to segregate your error types. This lets you, at a later point in time to decide how your application should react to specific types of errors. Let's create Custom Exception classes that can essentially make your application throw more sensible exceptions that can be easily understood.

Create a new folder named Exceptions and add a new class named `BaseException`. Make sure that you inherit Exception as the base class. Here is what the custom exception looks like.

```
public class BaseException : Exception
{
    public HttpStatusCode StatusCode { get; }
    public BaseException(string message, HttpStatusCode statusCode = HttpStatusCode.InternalServerError) : base(message)
    {
        StatusCode = statusCode;
    }
}
```

So, this will be the base exception class, inheriting which, other exception classes can be created. This is a far cleaner approach while designing your exception classes. For example, you can create a new class named `ProductNotFoundException` which inherits from this `BaseException` class.

For example,

```
public class ProductNotFoundException : BaseException
{
    public ProductNotFoundException(Guid id) : base($"product with id {id} not found", HttpStatusCode.NotFound)
    {
    }
}
```

So, in any of your product-related service classes, if the product is not found within your database/cache, you can simply throw the `ProductNotFoundException` and pass the ID of the product. And anyone that goes through the error logs would have instant clarity on what the error is, and for which product the error has occurred, instead of going through trace logs. As simple as that.

Here is how you would be using this Custom Exception class that we created now.

```
throw new ProductNotFoundException(product.Id);
```

Get the idea, right? In this way, you can differentiate between exceptions. To get even more clarity related to this scenario, let's say we have other custom exceptions like

`StockExpiredException`, `CustomerInvalidException`, and so on. Just give some meaningful names so that you can easily identify them. Now you can use these exception classes wherever the specific exception arises. This sends the related exception to the middleware, which has logic to handle it.

Custom Middleware - Global Exception Handling In ASP.NET Core [Old Method]

Now that we have our custom exception classes ready, let's create a Custom Global Exception Handling Middleware that gives even more control to the developer and makes the error-handling process much better.

Custom Global Exception Handling Middleware - Firstly, what is it? It's a piece of code that can be configured as a middleware in the ASP.NET Core pipeline which contains our custom error handling logic. There are a variety of exceptions that can be caught by this pipeline.

Now, let's create the Global Exception Handling Middleware. Create a new class and name it

`ErrorHandlerMiddleware`

```
1 public class ErrorHandlerMiddleware(RequestDelegate _next, ILogger<ErrorHandlerMidd
2 {
3     public async Task Invoke(HttpContext context)
4     {
5         try
6         {
7             await _next(context);
8         }
9         catch (Exception error)
10        {
11            var response = context.Response;
12            response.ContentType = "application/json";
13            response.StatusCode = error switch
14            {
```

```
15         BaseException e => (int)e.StatusCode,  
16         _ => StatusCodes.Status500InternalServerError,  
17     };  
18     var problemDetails = new ProblemDetails  
19     {  
20         Status = response.StatusCode,  
21         Title = error.Message,  
22     };  
23     logger.LogError(error.Message);  
24     var result = JsonSerializer.Serialize(problemDetails);  
25     await response.WriteAsync(result);  
26     }  
27 }  
28 }
```

Line #5 has a simple try-catch block over the request delegate. It means that whenever there is an exception of any type in the pipeline for the current request, the control goes to the catch block. In this middleware, the Catch block has the error-handling logic.

Line #9 catches all the Exceptions. Remember, all our custom exceptions are derived from the Exception base class.

Lines #13 to #17 have a neat switch expression that can allow us to set the status code of the returned response based on the exception type. This is where custom exception classes can come in handy.

Line #15 fetches the status code of the custom exception of type `BaseException` and sets it to the status code of the response. For instance, whenever a `ProductNotFound` exception is thrown, the status code we had set earlier in our `ProductNotFound` exception class, which is 404 will be fetched here. This how helpful the custom exception classes are!

In lines #18 to #22, we create a new `ProblemDetails` class where we will fill in error-related information like status code, message, and other custom properties if needed.

Line #25 - Finally, the created problems detail model is serialized and sent as a response.

Before running this implementation, make sure that you don't miss adding this middleware to the application pipeline. Open up the `Program.cs` class and add the following line.

```
app.UseMiddleware<ExceptionHandlerMiddleware>();
```


Make sure that you comment out or delete the `UseExceptionHandler` default middleware as it may cause unwanted clashes. It doesn't make sense to have multiple middlewares doing the same thing, yeah?

Additionally, I have added a Minimal endpoint with the route as `"/"`, which directly throws the `ProductNotFoundException` exception.

```
app.MapGet("/", () => { throw new ProductNotFoundException(Guid.NewGuid()); });
```

With that done, let's run the application and see how the error gets displayed on Swagger.

The screenshot shows the Swagger UI interface. At the top, the 'Request URL' is `https://localhost:7051/`. Below it, the 'Server response' section is expanded, showing a '404' status code and the message 'Error: response status is 404'. The 'Response body' is displayed as a JSON object: `{ "title": "product with id 714522f3-3970-4cdf-81fb-191bd0adbe31 not found", "status": 404 }`. The 'Response headers' section shows `content-type: application/json`, `date: Sat, 27 Apr 2024 06:42:51 GMT`, and `server: Kestrel`. There are 'Download' buttons for the response body and headers.

There you go! You can see how well-built the response is and how easy it is to read what the API has to say to the client. Now, we have a completely custom-built error-handling mechanism, all in one place. And yes, of course as mentioned earlier, you are always free to add more properties to the `ProblemDetails` class that suits your application's needs.

IExceptionHandler in .NET 8 and above [Recommended]

`IExceptionHandler` is an interface that was introduced as part of .NET 8, and is the recommended approach while handling exceptions globally. This interface is internally used by ASP.NET Core applications for their built-in exception-handling mechanism as well. This is an improved approach considering the other mechanisms that we have gone through.

The `IExceptionHandler` interface wants you to implement a single method, `TryHandleAsync` which works with the HTTP context and the actual error object. Another advantage is that you won't have to write an additional middleware for this to work, since it uses the already available `UseExceptionHandler` middleware of .NET, which we have seen earlier.

This way, you will be able to separately define the error handling mechanism for every error, if needed. This helps build a more modular and maintainable code base.

Let's see `IExceptionHandler` in action.

```
1 public class GlobalExceptionHandler(ILogger<GlobalExceptionHandler> logger) : IExce
2 {
3     public async ValueTask<bool> TryHandleAsync(HttpContext httpContext, Exception
4     {
5         var problemDetails = new ProblemDetails();
6         problemDetails.Instance = httpContext.Request.Path;
7         if (exception is BaseException e)
8         {
9             httpContext.Response.StatusCode = (int)e.StatusCode;
10            problemDetails.Title = e.Message;
11        }
12        else
13        {
14            problemDetails.Title = exception.Message;
15        }
16        logger.LogError("{ProblemDetailsTitle}", problemDetails.Title);
17        problemDetails.Status = httpContext.Response.StatusCode;
18        await httpContext.Response.WriteAsJsonAsync(problemDetails, cancellationTok
19        return true;
20    }
21 }
```

The `TryHandleAsync` is where all the exception-handling logic resides. As you see in line #19, this method should always return `true` if the exception is handled as required. Else, if the exception is not handled, or for any of your use cases, it can return false. This is typically applicable when you want to chain multiple such `IExceptionHandler` implementations for multiple errors. We will learn about this in the next section!

Once the `IExceptionHandler` implementation is completed, navigate to `Program.cs` and add in the following.

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

```
app.UseExceptionHandler();
```

The above ensures that your `IExceptionHandler` implementation is registered into the service container of the application along with `ProblemDetails`. Also, as we saw earlier, you will need to add the built-in exception middleware to the pipeline.

Simply run the application, and invoke the minimal API endpoint that we had created earlier.



As you can see, we can see an almost similar response to earlier.

Silencing Microsoft.AspNetCore.Diagnostics.ExceptionHandlerMiddleware Logs

If you see the console output, you will notice additional log messages that come directly from the built-in middleware. I often tend to silence this error message because I want to log the error message directly from my handlers. To silence the message from

`Microsoft.AspNetCore.Diagnostics.ExceptionHandlerMiddleware` source, simply open up the `appsettings.json` and add the following line to the `Logging` section. I have set the log level to `None`. This ensures that we won't be seeing the middleware logs anymore.

```
{
  "Logging": {
    "LogLevel": {
      "Default": "Information",
      "Microsoft.AspNetCore": "Warning",
      "Microsoft.AspNetCore.Diagnostics.ExceptionHandlerMiddleware": "None"
    }
  },
}
```

```
"AllowedHosts": "*"
}
```

Handling Multiple Errors with IExceptionHandler

Earlier we discussed a scenario where we would want to write separate handler classes for each exception. To elaborate on this point, let's say you have 2 custom exception classes that you want to handle separately, `ProductNotFoundException` and `StockExhaustedException`.

In this case, you would want to define 2 handler classes that inherit from `IExceptionHandler`.

```
public class ProductNotFoundExceptionHandler(ILogger<ProductNotFoundExceptionHandler> log
{
    public async ValueTask<bool> TryHandleAsync(HttpContext httpContext, Exception except
    {
        if (exception is not ProductNotFoundException e)
        {
            return false;
        }

        //handle error

        return true;
    }
}
```

Here, as soon as the control falls to `TryHandleAsync`, it first checks if the type of exception is `ProductNotFoundException`. If the exception type does not match, it simply returns a false, which means that the exception is not handled. In this case, the next exception handler chained into the pipeline will come into play.

```
builder.Services.AddExceptionHandler<ProductNotFoundExceptionHandler>();
builder.Services.AddExceptionHandler<StockExhaustedExceptionHandler>();
```

As I said, you can chain your exception handlers this way, where the

`ProductNotFoundExceptionHandler` will be executed first. If the error is not handled, the next chained handler, which is the `StockExhaustedExceptionHandler` will try to handle the error. This way you can build a well decoupled and modular application with ease.

That's it for today. I hope you all had an interesting read!

Summary

In this article, we have looked through various ways to implement Exception handling in our ASP.NET Core applications. The recommended approach for any .NET 8 or above applications would be to use the `IExceptionHandler` interface since it provides more control and readability within your application code. Have any suggestions or questions? Feel free to leave them in the comment section below. Thanks and Happy Coding! 😊

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



4 comments – powered by giscus

Oldest

Newest

Write

Preview

Aa

Sign in to comment



Sign in with GitHub

dkj468 May 3

edited

@iammukeshm , What is the advantage of using this newly introduced feature over previous method which is error handling middleware ? Just trying to understand why DOTNET team has introduced this feature ?

↑ 1



0 replies

zabronm Jun 11

Way you write some of your articles seems is for the experts who would normally already be aware of the subject. If you could explain in your code where to put what procedure explicitly rather than assuming. I got lost too early and abandoned rest of the article. Please do not assume all your audiences know all the concepts that you would normally call basics.

↑ 1



1

0 replies

iago-mota-dev Jun 24

Really good! Keep with the quality content, helped me a lot to understand new ways of error handling in .net.
For those complaining in the comments, i just recommend to keep studying, cause you need.

↑ 1



0 replies

CalebTek Jun 24

This is a very nice article on handling Exception

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!



[Home](#) [Blog](#) [Contact](#) [Sponsorship](#) [About](#)

© 2024 Mukesh Murugan