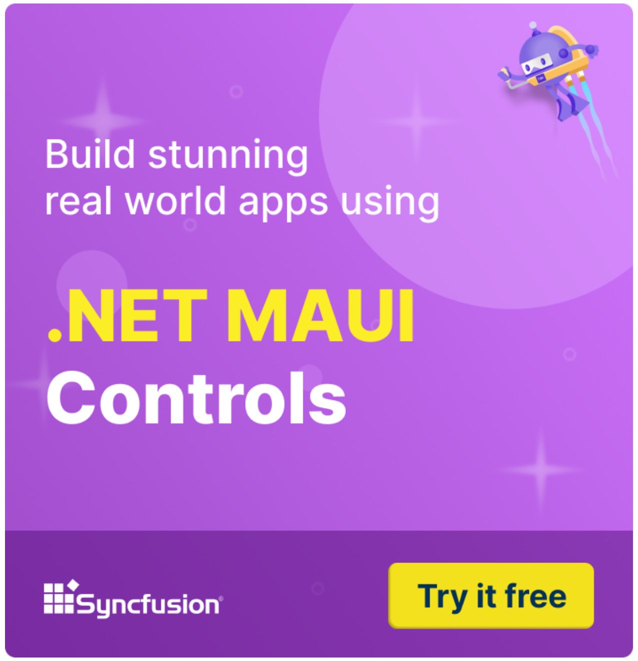
Global Exception Handling in .NET 6

Piumi Liyana Gunawardhana • 6 min read • Apr 24, 2024 • Updated • 6 Comments



> **TL;DR:** *Want to streamline error handling in your ASP.NET 6 apps? Explore two methods: traditional try-catch blocks and custom middleware for global exception handling.*

Exception handling is one of the most critical areas in modern web application development. If exceptions are not handled properly, the whole app can be terminated, causing severe issues for users and developers.

In this article, I will discuss different methods of global exception handling in .NET apps.

Error handling with try-catch blocks

Try-catch blocks are widely used in apps to handle exceptions. They are the most basic way of handling exceptions. To demonstrate this, I will be using an ASP .NET Core web API project based on .NET 6.

Consider the following **CustomerController** class as a code example:

Copy

```
using ExceptionHandling.Services;
using Microsoft.AspNetCore.Mvc;
namespace ExceptionHandling.Controllers;

[ApiController]
[Route("api/[controller]")]
public class CustomerController : Controller
{
    private readonly ICustomerService _customerService;
    private readonly ILogger<CustomerController> _logger;

    public CustomerController(ICustomerService customerService, ILogger<Customer
    {
        _customerService = customerService;
        _logger = logger;
    }

    [HttpGet]
    public IActionResult GetCustomers()
    {
        try
        {
            _logger.LogInformation("Getting customer details");

            var result = _customerService .GetCustomers();
            if (result == null)
                throw new ApplicationException("Getting errors while fetching cu

            return Ok(result);
        }
        catch (Exception e)
```

```
            {
                _logger.LogError(e.Message);
                return BadRequest("Internal server error");
            }
        }
    }
```

The above code example is a typical case where we use try-catch to handle the exception. Any exception thrown by the code enclosed within the try block will be caught and handled by the catch block.

The try-catch method is ideal for novice developers, and it is something that every developer should be aware of. However, this technique has a disadvantage when working with massive projects with complex architectures.

Consider a scenario where you have many controllers and actions in your project, and you need to utilize try-catch for each action in the controllers. In some cases, try-catch must be used in the services as well. In such instances, it will double the lines of code in your project, which is undesirable.

If the solution architecture contains different layers (e.g., data access, business logic, and presentation layer), you will have to map that exception in between layers when your code throws an exception. Or else you may have more severe issues, such as missing the exception where no one knows the error until checking the log/trace.

Global exception handling is a handy approach for eliminating these drawbacks. In the next section of this article, we'll see how you can use custom middleware to handle exceptions globally in .NET 6.

Global exception handling with custom middleware

Global exception handling with custom middleware grants the developer much broader authority and enhances the procedure. It's a block of code that can be added to the ASP.NET Core pipeline as middleware and holds our custom error handling mechanism. This pipeline is capable of catching a wide range of exceptions.

This approach aims to ensure your ASP.NET Core API produces consistent responses regardless of the type of request. It makes things simpler for anybody who uses your API to do their job. It also provides a better development experience.

Create a separate folder named **CustomMiddlewares** and add a class file named **ExceptionHandlingMiddleware.cs** within it.

Refer to the following code example:

Copy

```
using System.Net;
using System.Text.Json;
using ExceptionHandling.Models.Responses;

namespace ExceptionHandling.CustomMiddlewares;

public class ExceptionHandlingMiddleware
{
    private readonly RequestDelegate _next;
    private readonly ILogger<ExceptionHandlingMiddleware> _logger;

    public ExceptionHandlingMiddleware(RequestDelegate next, ILogger<ExceptionHa
    {
        _next = next;
        _logger = logger;
    }

    public async Task InvokeAsync(HttpContext httpContext)
    {
        try
        {
            await _next(httpContext);
        }
```

```
            catch (Exception ex)
            {
                await HandleExceptionAsync(httpContext, ex);
            }
        }

        private async Task HandleExceptionAsync(HttpContext context, Exception excep
        {
            context.Response.ContentType = "application/json";
            var response = context.Response;

            var errorResponse = new ErrorResponse
            {
                Success = false
            };
            switch (exception)
            {
                case ApplicationException ex:
                    if (ex.Message.Contains("Invalid Token"))
                    {
                        response.StatusCode = (int) HttpStatusCode.Forbidden;
                        errorResponse.Message = ex.Message;
                        break;
                    }
                    response.StatusCode = (int) HttpStatusCode.BadRequest;
                    errorResponse.Message = ex.Message;
                    break;
                default:
                    response.StatusCode = (int) HttpStatusCode.InternalServerError;
                    errorResponse.Message = "Internal server error!";
                    break;
            }
            _logger.LogError(exception.Message);
            var result = JsonSerializer.Serialize(errorResponse);
            await context.Response.WriteAsync(result);
        }
    }
```

The above code is the custom middleware that handles exceptions. We must first use dependency injection to register the **ILogger** and **RequestDelegate** services. The **_next** parameter of the **RequestDelegate** type is a function delegate that handles our HTTP requests. In addition, after the middleware receives the request delegate, it either processes or passes it on to the next middleware in the chain.

If the request fails, an exception may occur, and the **HandleExceptionAsync** method will be called to capture the exception as per its type. In such scenarios, use **switch** statements to determine the exception type and then utilize the appropriate status code for the exception.

Also, we need not send the exception messages to the project's client side. Instead, use **ILogger** to log the exception message as an error and pass the custom message. We can then look through the logs and traces to find the exception message.

Next, the custom middleware must be included in the **Program.cs** file. Using the previous versions, you may add the custom middleware to the **Startup** class **Configure** method.

Copy

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

Now, remove the try-catch block from the **Controller.**

Copy

```
using ExceptionHandling.Services;
using Microsoft.AspNetCore.Mvc;

namespace ExceptionHandling.Controllers;

[ApiController]
[Route("api/[controller]")]
public class CustomerController : Controller
{
    private readonly ICustomerService _customerService;
    private readonly ILogger<CustomerController> _logger;

    public CustomerController(ICustomerService customerService, ILogger<Customer
    {
        _customerService = customerService;
        _logger = logger;
    }
    [HttpGet]
    public IActionResult GetCustomers()
    {

        _logger.LogInformation("Getting customer details");

        var result = _customerService.GetCustomers();
        if (result.Count == 0)
            throw new ApplicationException("Invalid Token");

        return Ok(result);

    }
}
```

You just have to throw the relevant exception in the **Controller** instead of using a try-catch block.

Advantage of Global exception handling

Global exception handling allows us to organize all exception handling logic in one place. Thus, we can improve the readability of the action methods and the maintainability of the error handling process. This strategy can effectively help your app throw more logical and understandable exceptions. Having global exception handling also eliminates the need to map each exception and map between architectural layers.

Conclusion

In this article, we went through how to handle errors at the global level in ASP.NET Core web API projects based on .NET 6. When working on massive projects, this strategy is beneficial since we won't have to use try-catch in every controller action. Additionally, it improves code clarity and provides the project with a straightforward and reusable exception handling technique.

I hope you found this helpful. Thank you for reading!

The Syncfusion ASP.NET Core platform contains over 80 high-performance, lightweight, modular, and responsive UI controls in a single package. Use them to build stunning web apps!

If you have questions, you can contact us through our support forum, support portal, or feedback portal. We are always happy to assist you!

Related blogs

- Implementing CPU-Bound Operations in an ASP.NET Core Application
- Authentication Support in Syncfusion ASP.NET Core Project Template: An Overview
- Easily Improve Front-end ASP.NET Core App Development with Gulp
- Firebase Push Notifications for Android and iOS Using Ionic and ASP.NET- A Complete Guide

MEET THE AUTHOR

Piumi Liyana Gunawardhana

Software Engineer | Technical Writer since 2020