

Add Exception Middleware:

public class ExceptionHandlingMiddleware

{

private readonly RequestDelegate \_next;

private readonly ILogger<ExceptionHandlingMiddleware> \_logger;

public ExceptionHandlingMiddleware(

RequestDelegate next,

ILogger<ExceptionHandlingMiddleware> logger)

{

\_next = next;

\_logger = logger;

}

public async Task InvokeAsync(HttpContext context)

{

try

{

await \_next(context);

}

catch (Exception exception)

{

\_logger.LogError(

exception, "Exception occurred: {Message}", exception.Message);

var problemDetails = new ProblemDetails

{

Status = StatusCodes.Status500InternalServerError,

Title = "Server Error"

};

context.Response.StatusCode =

StatusCodes.Status500InternalServerError;

await context.Response.WriteAsJsonAsync(problemDetails);

}

}

}

Add it in Program.cs:  
builder.Services.AddExceptionHandler<GlobalExceptionHandler>();

New way in Asp.net 8:

Built in Exception handler using IExceptionHandler implementation:  
This interface has only one TryHandleAsync method.

internal sealed class GlobalExceptionHandler : IExceptionHandler

{

private readonly ILogger<GlobalExceptionHandler> \_logger;

public GlobalExceptionHandler(ILogger<GlobalExceptionHandler> logger)

{

\_logger = logger;

}

public async Task<bool> TryHandleAsync(

HttpContext httpContext,

Exception exception,

CancellationToken cancellationToken)

{

\_logger.LogError(

exception, "Exception occurred: {Message}", exception.Message);

var problemDetails = new ProblemDetails

{

Status = StatusCodes.Status500InternalServerError,

Title = "Server error"

};

httpContext.Response.StatusCode = problemDetails.Status.Value;

await httpContext.Response

.WriteAsJsonAsync(problemDetails, cancellationToken);

return true;

}

}

You need two things to add an IExceptionHandler implementation to the ASP.NET Core request pipeline:

Register the IExceptionHandler service with dependency injection

Register the ExceptionHandlerMiddleware with the request pipeline

You call the AddExceptionHandler method to register the GlobalExceptionHandler as a service. It's registered with a singleton lifetime. So be careful about injecting services with a different lifetime.

I'm also calling AddProblemDetails to generate a Problem Details response for common exceptions.

In program.cs:

builder.Services.AddExceptionHandler<GlobalExceptionHandler>();

builder.Services.AddProblemDetails();

app.UseExceptionHandler();

You can add multiple IExceptionHandler implementations, and they're called in the order they are registered. A possible use case for this is using exceptions for flow control.

Bad request Exception handler:

internal sealed class BadRequestExceptionHandler : IExceptionHandler

{

private readonly ILogger<BadRequestExceptionHandler> \_logger;

public BadRequestExceptionHandler(ILogger<BadRequestExceptionHandler> logger)

{

\_logger = logger;

}

public async ValueTask<bool> TryHandleAsync(

HttpContext httpContext,

Exception exception,

CancellationToken cancellationToken)

{

if (exception is not BadRequestException badRequestException)

{

return false;

}

\_logger.LogError(

badRequestException,

"Exception occurred: {Message}",

badRequestException.Message);

var problemDetails = new ProblemDetails

{

Status = StatusCodes.Status400BadRequest,

Title = "Bad Request",

Detail = badRequestException.Message

};

httpContext.Response.StatusCode = problemDetails.Status.Value;

await httpContext.Response

.WriteAsJsonAsync(problemDetails, cancellationToken);

return true;

}

}