**Error Handling in ASP.NET Core MVC**

Error handling is a crucial aspect of building robust and user-friendly web applications. In ASP.NET Core MVC, it involves gracefully handling exceptions, providing informative feedback to users, and ensuring the application continues to function smoothly even when unexpected errors occur.

**Exception Handling Middleware**

Exception handling middleware is a type of custom middleware in ASP.NET Core that catches exceptions thrown during the request processing pipeline. This middleware allows you to:

* **Centralize Error Handling:** Implement a single point where you can catch and handle exceptions from different parts of your application.
* **Custom Error Responses:** Generate appropriate error responses (HTML pages, JSON messages) for different types of exceptions.
* **Logging:** Log exceptions and their details for troubleshooting and analysis.

**Custom Exceptions**

In some scenarios, you might want to define your own custom exceptions to represent specific error conditions in your application. This allows you to:

* **Provide Context:** Include additional information in the exception object that helps you understand the root cause of the error.
* **Categorization:** Differentiate between different types of errors based on their exception types.
* **Error Handling Logic:** Implement custom logic in your exception handling middleware to respond to specific custom exceptions differently.

**UseExceptionHandler Middleware**

The UseExceptionHandler middleware is a built-in middleware component in ASP.NET Core that handles unhandled exceptions in your application. It provides a centralized way to control the error response sent to the client.

* **Custom Error Pages:** You can configure UseExceptionHandler to redirect to a specific error page or endpoint (e.g., /Error) when an exception occurs. This allows you to present a user-friendly error message or provide additional information to help users understand what happened.
* **Development vs. Production:** In development environments, you often use UseDeveloperExceptionPage to display a detailed error page with stack traces and other diagnostic information. In production, you should use UseExceptionHandler to hide those sensitive details and provide a more generic error message.

**Code Example**

// ExceptionHandlingMiddleware.cs

public class ExceptionHandlingMiddleware

{

private readonly RequestDelegate \_next;

private readonly ILogger<ExceptionHandlingMiddleware> \_logger;

// Injected logger

private readonly IDiagnosticContext \_diagnosticContext; // For enriching Serilog logs

// Constructor injection

public ExceptionHandlingMiddleware(RequestDelegate next, ILogger<ExceptionHandlingMiddleware> logger, IDiagnosticContext diagnosticContext)

{

\_next = next; // Represents the next middleware in the pipeline

\_logger = logger;

\_diagnosticContext = diagnosticContext;

}

public async Task Invoke(HttpContext httpContext)

{

try

{

await \_next(httpContext);

// Invoke the next middleware

}

catch (Exception ex)

{

// Log the inner exception if present, otherwise log the original exception

if (ex.InnerException != null)

{

\_logger.LogError("{ExceptionType} {ExceptionMessage}", ex.InnerException.GetType().ToString(), ex.InnerException.Message);

}

else

{

\_logger.LogError("{ExceptionType} {ExceptionMessage}", ex.GetType().ToString(), ex.Message);

}

// (Optional) You can customize the error response here

// httpContext.Response.StatusCode = 500;

// await httpContext.Response.WriteAsync("Error occurred");

throw; // Re-throw the exception for further handling (e.g., by UseExceptionHandler)

}

}

}

// Extension method for easy registration

public static class ExceptionHandlingMiddlewareExtensions

{

public static IApplicationBuilder UseExceptionHandlingMiddleware(this IApplicationBuilder builder)

{

return builder.UseMiddleware<ExceptionHandlingMiddleware>();

}

}

* **Purpose:** This custom middleware catches exceptions and logs them using Serilog.
* **Constructor Injection:** It receives the RequestDelegate (\_next), an ILogger, and an IDiagnosticContext (used for adding contextual information to Serilog logs) through constructor injection.
* **Invoke Method:**
  1. await \_next(httpContext);: Invokes the next middleware in the pipeline.
  2. try-catch Block: Catches any exceptions thrown during the execution of subsequent middleware or the action method.
  3. **Logging:** Logs the exception details using Serilog, including the exception type and message. If there's an inner exception, it logs that instead.
  4. **Re-throwing:** The throw; statement re-throws the exception, allowing it to be handled further up the pipeline, potentially by the UseExceptionHandler middleware.

**Program.cs**

// ... (other configuration) ...

var app = builder.Build();

if (app.Environment.IsDevelopment())

{

app.UseDeveloperExceptionPage(); // Detailed error page in development

}

else

{

app.UseExceptionHandler("/Error"); // Redirect to a custom error page in other environments

app.UseExceptionHandlingMiddleware(); // Use the custom exception handling middleware

}

// ... (other middleware and routing) ...

* **UseDeveloperExceptionPage():** This middleware is enabled only in the Development environment to provide detailed error information for debugging.
* **UseExceptionHandler("/Error"):** In non-development environments, this middleware redirects to the "/Error" endpoint (which you'll need to define in your controllers) when an unhandled exception occurs.
* **UseExceptionHandlingMiddleware():** This registers your custom exception handling middleware, which will catch and log exceptions before they reach UseExceptionHandler.

**Notes**

* **Centralized Error Handling:** Use exception handling middleware or UseExceptionHandler to create a single point for managing exceptions.
* **Environment-Specific Behavior:** Provide detailed error information in development, but use generic error pages in production for security.
* **Custom Exceptions:** Consider creating custom exceptions to convey specific error conditions in your application.
* **Logging:** Always log exceptions and their details for troubleshooting and analysis.
* **User-Friendly Error Messages:** Provide clear and informative error messages to users, guiding them on how to resolve the issue.
* **Testing:** Write unit tests for your exception handling middleware and custom exception classes to ensure they work as expected.

**Key Points to Remember**

**Goals**

* **Graceful Recovery:** Handle exceptions and errors smoothly, preventing application crashes.
* **User Experience:** Provide informative and helpful error messages to users.
* **Security:** Avoid exposing sensitive information in error responses.
* **Maintainability:** Centralize error handling logic for easier maintenance.

**Key Techniques**

* **Exception Handling Middleware:**
  + Custom middleware that catches exceptions during the request pipeline.
  + Centralizes error handling logic.
  + Can generate custom error responses or log exceptions.
* **UseExceptionHandler Middleware:**
  + Built-in middleware for handling unhandled exceptions.
  + Redirects to a specific error page or endpoint (e.g., /Error).
  + Useful for providing user-friendly error messages in production.
* **UseDeveloperExceptionPage Middleware:**
  + Displays a detailed error page with stack trace and other diagnostic information.
  + **Only for development environments.**
* **Custom Exceptions:**
  + Create your own exception classes to represent specific error conditions.
  + Add contextual information to the exception object.
  + Can be used to trigger specific error handling logic.

**Best Practices**

* **Centralized Handling:** Use exception handling middleware or UseExceptionHandler to manage exceptions in one place.
* **Environment-Specific Errors:**
  + **Development:** Use UseDeveloperExceptionPage for detailed errors.
  + **Production:** Use UseExceptionHandler for generic error pages, avoid exposing sensitive details.
* **Custom Exceptions:** Create custom exceptions for specific error scenarios.
* **Logging:** Always log exceptions with relevant details for troubleshooting.
* **User-Friendly Messages:** Provide clear and helpful error messages to users.
* **HTTP Status Codes:** Use appropriate status codes to indicate the type of error (e.g., 400 Bad Request, 404 Not Found, 500 Internal Server Error).

**Interview Tips**

* **Explain the Flow:** Articulate how exceptions are handled in ASP.NET Core MVC and the role of middleware.
* **Custom Middleware:** Discuss scenarios where you would create custom exception handling middleware.
* **Custom Exceptions:** Explain when and how to create custom exception classes.
* **Security:** Emphasize the importance of protecting sensitive information in error responses.
* **User Experience:** Highlight the need for user-friendly error messages.