**Handling Errors in ASP .NET Core 3.1**

By Shahed C on February 24, 2020

[7 Replies](https://wakeupandcode.com/handling-errors-in-asp-net-core-3-1/#comments)



This is the eighth of a new [series of posts](https://wakeupandcode.com/aspnetcore/#aspnetcore2020) on ASP .NET Core 3.1 for 2020. In this series, we’ll cover 26 topics over a span of 26 weeks from January through June 2020, titled **ASP .NET Core A-Z!** To differentiate from the [2019 series](https://wakeupandcode.com/aspnetcore/#aspnetcore2019), the 2020 series will mostly focus on a growing single codebase ([NetLearner!](https://wakeupandcode.com/netlearner-on-asp-net-core-3-1/)) instead of new unrelated code snippets week.

Previous post:

* [Generic Host Builder in ASP .NET Core 3.1](https://wakeupandcode.com/generic-host-builder-in-asp-net-core-3-1/)

**NetLearner on GitHub**:

* Repository: <https://github.com/shahedc/NetLearnerApp>
* v0.8-alpha release: <https://github.com/shahedc/NetLearnerApp/releases/tag/v0.8-alpha>

**In this Article:**

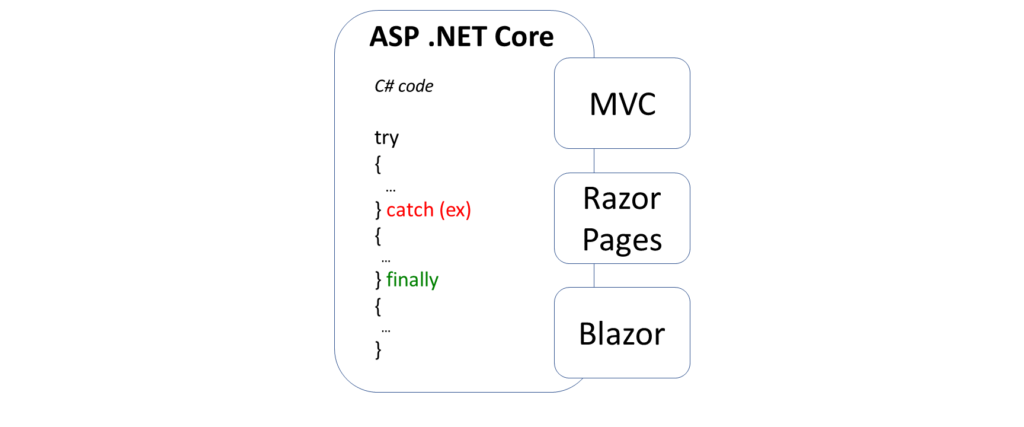
* H is for Handling Errors
* Exceptions with Try-Catch-Finally
* Try-Catch-Finally in NetLearner
* Error Handling for MVC
* Error Handling for Razor Pages
* Error Handling in Blazor
* Logging Errors
* Transient Fault Handling
* References

**H is for Handling Errors**

Unless you’re perfect 100% of the time (who is?), you’ll most likely have errors in your code. If your code doesn’t build due to compilation errors, you can probably correct that by fixing the offending code. But if your application encounters runtime errors while it’s being used, you may not be able to anticipate every possible scenario.

Runtime errors may cause Exceptions, which can be caught and handled in many programming languages. Unit tests will help you write better code, minimize errors and create new features with confidence. In the meantime, there’s the good ol’ try-catch-finally block, which should be familiar to most developers.

**NOTE:** You may skip to the next section below if you don’t need this refresher.

 try-catch-finally in C#

**Exceptions with Try-Catch-Finally**

The simplest form of a try-catch block looks something like this:

try  
{  
 // try something here  
  
} catch (Exception ex)  
{  
 // catch an exception here  
}

You can chain multiple catch blocks, starting with more specific exceptions. This allows you to catch more generic exceptions toward the end of your try-catch code.  In a string of ***catch***() blocks, only the caught exception (if any) will cause that block of code to run.

try  
{  
 // try something here  
  
} catch (IOException ioex)  
{  
 // catch specific exception, e.g. IOException  
  
} catch (Exception ex)  
{  
 // catch generic exception here  
  
}

Finally, you can add the optional ***finally***block. Whether or not an exception has occurred, the finally block will always be executed.

try  
{  
 // try something here  
  
} catch (IOException ioex)  
{  
 // catch specific exception, e.g. IOException  
  
} catch (Exception ex)  
{  
 // catch generic exception here  
  
} finally  
{  
 // always run this code  
  
}

**Try-Catch-Finally in NetLearner**

In the [NetLearner sample app](https://wakeupandcode.com/netlearner-on-asp-net-core-3-1/), the [LearningResourcesController](https://github.com/shahedc/NetLearnerApp/blob/main/src/NetLearner.Mvc/Controllers/LearningResourcesController.cs) uses a [LearningResourceService](https://github.com/shahedc/NetLearnerApp/blob/main/src/NetLearner.SharedLib/Services/LearningResourceService.cs) from a shared .NET Standard Library to handle database updates. In the overloaded Edit() method, it wraps a call to the Update() method from the service class to catch a possible DbUpdateConcurrencyException exception. First, it checks to see whether the ModelState is valid or not.

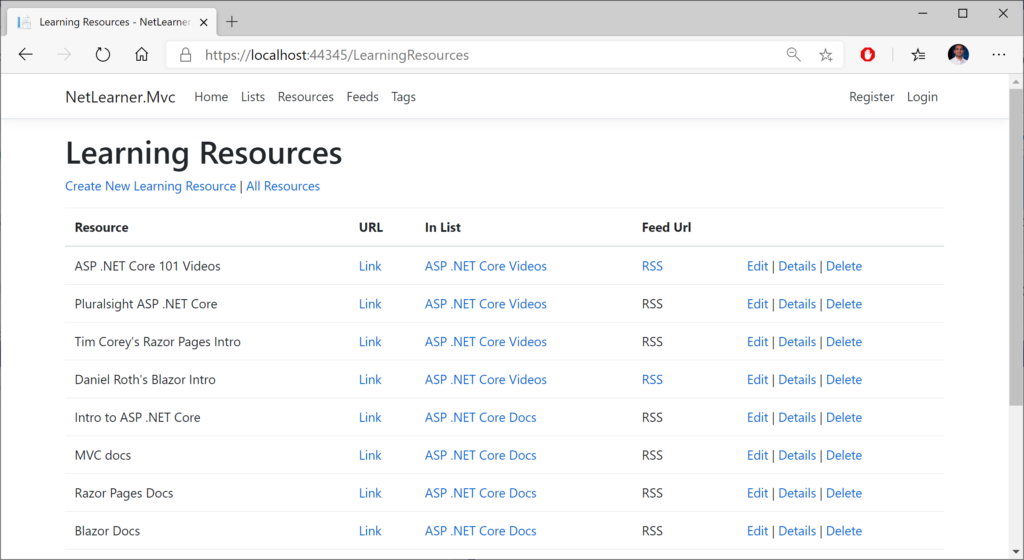
if (ModelState.IsValid)  
{  
 ...  
}

Then, it tries to update user-submitted data by passing a LearningResource object. If a DbUpdateConcurrencyException exception occurs, there is a check to verify whether the current LearningResource even exists or not, so that a 404 (NotFound) can be returned if necessary.

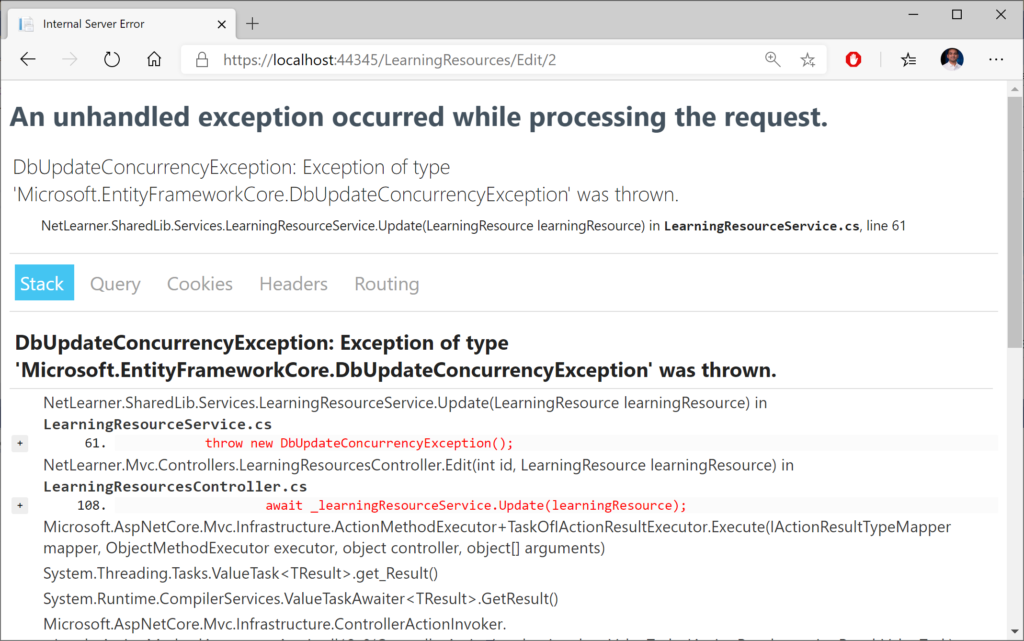
try  
{  
 await \_learningResourceService.Update(learningResource);  
}  
catch (DbUpdateConcurrencyException)  
{  
 if (!LearningResourceExists(learningResource.Id))  
 {  
 return NotFound();  
 }  
 else  
 {  
 throw;  
 }  
}  
return RedirectToAction(nameof(Index));

In the above code, you can also see a **throw** keyword by itself, when the expected exception occurs if the Id is found to exist. In this case, the throw statement (followed immediately by a semicolon) ensures that the exception is *rethrown*, while *preserving* the stack trace.

Run the MVC app and navigate to the Learning Resources page after adding some items. If there are no errors, you should see just the items retrieved from the database.

 Learning Resources UI (no errors)

If an exception occurs in the Update() method, this will cause the expected exception to be thrown. To simulate this exception, you can intentionally throw the exception inside the update method. This should cause the error to be displayed in the web UI when attempting to save a learning resource.

Exception details in Web UI

**Error Handling for MVC**

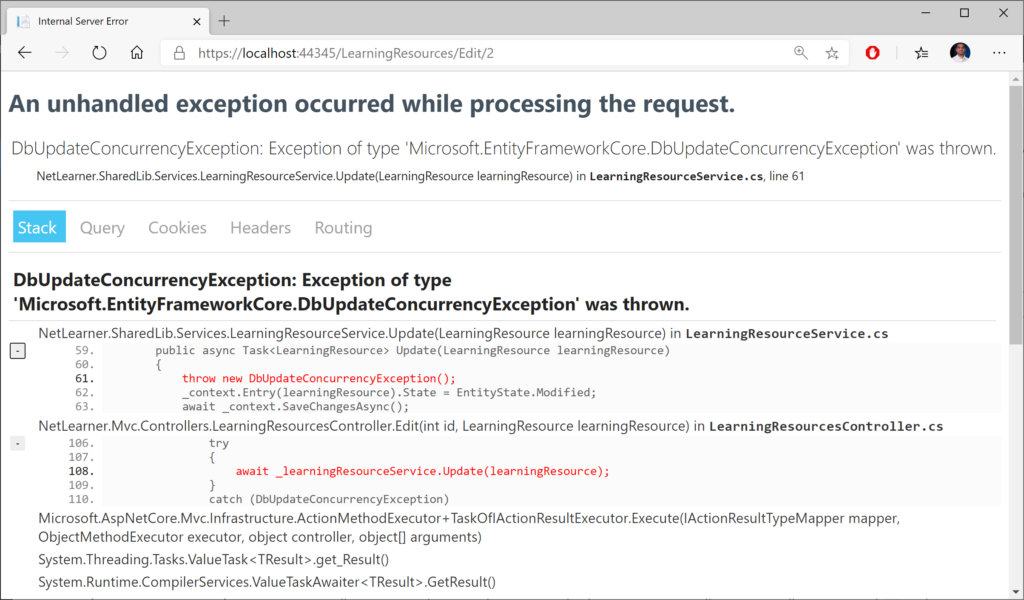
In ASP .NET Core MVC web apps, unhandled exceptions are typically handled in different ways, depending on whatever [environment](https://docs.microsoft.com/en-us/aspnet/core/fundamentals/environments) the app is running in. The default template uses the **DeveloperExceptionPage** middleware in a development environment but redirects to a shared Error view in non-development scenarios. This logic is implemented in the **Configure**() method of the [Startup.cs](https://github.com/shahedc/ErrorHandlingWeb/blob/master/ErrorHandlingMvc/Startup.cs) class.

if (env.IsDevelopment())  
{  
 app.UseDeveloperExceptionPage();  
 app.UseDatabaseErrorPage();   
}  
else  
{  
 app.UseExceptionHandler("/Home/Error");  
 ...  
}

The [DeveloperExceptionPage](https://docs.microsoft.com/en-us/dotnet/api/microsoft.aspnetcore.diagnostics.developerexceptionpagemiddleware) middleware can be further customized with [DeveloperExceptionPageOptions](https://docs.microsoft.com/en-us/dotnet/api/microsoft.aspnetcore.builder.developerexceptionpageoptions) properties, such as FileProvider and SourceCodeLineCount.

var options = new DeveloperExceptionPageOptions  
{  
 SourceCodeLineCount = 2  
};   
app.UseDeveloperExceptionPage(options);

Using the snippet shown above, the error page will show the offending line in red, with a variable number of lines of code above it. The number of lines is determined by the value of **SourceCodeLineCount**, which is set to 2 in this case. In this contrived example, I’m forcing the exception by throwing a new Exception in my code.

Customized lines of code within error page

For non-dev scenarios, the shared **Error** view can be further customized by updating the [Error.cshtml](https://github.com/shahedc/NetLearnerApp/blob/main/src/NetLearner.Mvc/Views/Shared/Error.cshtml) view in the **Shared** subfolder. The [ErrorViewModel](https://github.com/shahedc/NetLearnerApp/blob/main/src/NetLearner.Mvc/Models/ErrorViewModel.cs) has a **ShowRequestId** boolean value that can be set to **true** to see the **RequestId** value.

@model ErrorViewModel  
@{  
 ViewData["Title"] = "Error";  
}  
  
<h1 class="text-danger">Error.</h1>  
<h2 class="text-danger">An error occurred while processing your request.</h2>  
  
@if (Model.ShowRequestId)  
{  
<p>  
<strong>Request ID:</strong> <code>@Model.RequestId</code>  
</p>  
}  
  
<h3>header content</h3>  
<p>text content</p>

In the MVC project’s [Home Controller](https://github.com/shahedc/NetLearnerApp/blob/main/src/NetLearner.Mvc/Controllers/HomeController.cs), the **Error**() action method sets the **RequestId** to the current **Activity.Current.Id** if available or else it uses **HttpContext.TraceIdentifier**. These values can be useful during debugging.

[ResponseCache(Duration = 0, Location = ResponseCacheLocation.None, NoStore = true)]  
public IActionResult Error()  
{  
 return View(new ErrorViewModel {   
 RequestId = Activity.Current?.Id ?? HttpContext.TraceIdentifier   
 });  
}

But wait… what about **Web API** in ASP .NET Core? After posting the 2019 versions of this article in a popular ASP .NET Core group on Facebook, I got some [valuable feedback](https://www.facebook.com/groups/637731783071479/permalink/1198959006948751/?comment_id=1198998203611498) from the admin:

[**Dmitry Pavlov**](http://dmitry-pavlov.com/): “For APIs there is a nice option to handle errors globally with the custom middleware <https://code-maze.com/global-error-handling-aspnetcore> – helps to get rid of try/catch-es in your code. Could be used together with FluentValidation and MediatR – you can configure mapping specific exception types to appropriate status codes (400 bad response, 404 not found, and so on to make it more user friendly and avoid using 500 for everything).”

For more information on the aforementioned items, check out the following resources:

* Global Error Handling in ASP.NET Core Web API: <https://code-maze.com/global-error-handling-aspnetcore/>
* FluentValidation • ASP.NET Integration: <https://fluentvalidation.net/aspnet>
* MediatR Wiki: <https://github.com/jbogard/MediatR/wiki>
* Using MediatR in ASPNET Core Apps: <https://ardalis.com/using-mediatr-in-aspnet-core-apps>

Later on in this series, we’ll cover ASP .NET Core Web API in more detail, when we get to “W is for Web API”. Stay tuned!

**Error Handling for Razor Pages**

Since Razor Pages still use the MVC middleware pipeline, the exception handling is similar to the scenarios described above. For starters, here’s what the **Configure**() method looks like in the [Startup.cs](https://github.com/shahedc/NetLearnerApp/blob/main/src/NetLearner.Pages/Startup.cs) file for the Razor Pages web app sample.

if (env.IsDevelopment())  
{  
 app.UseDeveloperExceptionPage();  
 app.UseDatabaseErrorPage();   
}  
else  
{  
 app.UseExceptionHandler("/Error");  
 ...  
}

In the above code, you can see the that development environment uses the same **DeveloperExceptionPage** middleware. This can be customized using the same techniques outlined in the previous section for MVC pages, so we won’t go over this again.

As for the non-dev scenario, the exception handler is slightly different for Razor Pages. Instead of pointing to the **Home** controller’s **Error**() action method (as the MVC version does), it points to the to the /**Error** page route. This [Error.cshtml](https://github.com/shahedc/NetLearnerApp/blob/main/src/NetLearner.Pages/Pages/Error.cshtml) Razor Page found in the root level of the **Pages** folder.

@page  
@model ErrorModel  
@{  
 ViewData["Title"] = "Error";  
}  
  
<h1 class="text-danger">Error.</h1>  
<h2 class="text-danger">An error occurred while processing your request.</h2>  
  
@if (Model.ShowRequestId)  
{  
 <p>  
 <strong>Request ID:</strong> <code>@Model.RequestId</code>  
 </p>  
}  
  
<h3>custom header text</h3>  
<p>custom body text</p>

The above Error page for looks almost identical to the Error view we saw in the previous section, with some notable differences:

* @**page** directive (required for Razor Pages, no equivalent for MVC view)
* uses **ErrorModel** (associated with Error page) instead of ErrorViewModel (served by Home controller’s action method)

**Error Handling for Blazor**

In Blazor web apps, the UI for error handling is included in the Blazor project templates. Consequently, this UI is available in the [NetLearner Blazor sample](https://github.com/shahedc/NetLearnerApp/tree/main/src/NetLearner.Blazor) as well. The \_Host.cshtml file in the Pages folder holds the following <**environment**> elements:

<div id="blazor-error-ui">  
 <environment include="Staging,Production">  
 An error has occurred. This application may no longer respond until reloaded.  
 </environment>  
 <environment include="Development">  
 An unhandled exception has occurred. See browser dev tools for details.  
 </environment>  
 <a href="" class="reload">Reload</a>  
 <a class="dismiss">🗙</a>  
 </div>

The div identified by the id “blazor-error-ui” ensures that is hidden by default, but only shown when an error has occured. Server-side Blazor maintains a connection to the end-user by preserving state with the use of a so-called circuit.

Each browser/tab instance initiates a new circuit. An unhandled exception may terminate a circuit. In this case, the user will have to reload their browser/tab to establish a new circuit.

According to the [official documentation](https://docs.microsoft.com/en-us/aspnet/core/blazor/handle-errors?view=aspnetcore-3.1), unhandled exceptions may occur in the following areas:

1. **Component instantiation**: when constructor invoked
2. **Lifecycle methods**: (see [Blazor post](https://wakeupandcode.com/blazor-full-stack-web-dev-in-asp-net-core-3-1/#lifecycle) for details)
3. **Upon rendering**: during BuildRenderTree() in .razor component
4. **UI Events**: e.g. onclick events, data binding on UI elements
5. **During disposal**: while component’s .Dispose() method is called
6. **JavaScript Interop**: during calls to IJSRuntime.InvokeAsync<T>
7. **Prerendering**: when using the [Component tag helper](https://docs.microsoft.com/en-us/aspnet/core/release-notes/aspnetcore-3.1?view=aspnetcore-3.1#blazor-component-tag-helper-and-pass-parameters-to-top-level-components)

To avoid unhandled exceptions, use try-catch exception handlers within .razor components to display error messages that are visible to the user. For more details on various scenarios, check out the official documentation at:

* Handle errors in ASP.NET Core Blazor apps: <https://docs.microsoft.com/en-us/aspnet/core/blazor/handle-errors?view=aspnetcore-3.1>

**Logging Errors**

To log errors in ASP .NET Core, you can use the built-in logging features or 3rd-party logging providers. In ASP .NET Core 2.x, the use of **CreateDefaultBuilder**() in [Program.cs](https://github.com/shahedc/ErrorHandlingWeb/blob/master/ErrorHandlingMvc/Program.cs) takes of care default Logging setup and configuration (*behind the scenes*).

public static IWebHostBuilder CreateWebHostBuilder(string[] args) =>  
 WebHost.CreateDefaultBuilder(args)  
 .UseStartup<Startup>();

The Web Host Builder was replaced by the Generic Host Builder in ASP .NET Core 3.0, so it looks slightly different now. For more information on Generic Host Builder, take a look at the previous blog post in this series: [Generic Host Builder in ASP .NET Core](https://wakeupandcode.com/generic-host-builder-in-asp-net-core/).

public static IHostBuilder CreateHostBuilder(string[] args) =>  
 Host.CreateDefaultBuilder(args)  
 .ConfigureWebHostDefaults(webBuilder =>  
 {  
 webBuilder.UseStartup();  
 });

The host can be used to set up [logging configuration](https://docs.microsoft.com/en-us/aspnet/core/fundamentals/logging/?view=aspnetcore-3.1), e.g.:

public static IHostBuilder CreateHostBuilder(string[] args) =>  
 Host.CreateDefaultBuilder(args)  
 .ConfigureLogging((hostingContext, logging) =>  
 {  
 logging.ClearProviders();  
 logging.AddConsole(options => options.IncludeScopes = true);  
 logging.AddDebug();  
 })  
 .ConfigureWebHostDefaults(webBuilder =>  
 {  
 webBuilder.UseStartup();  
 });

To make use of error logging (in addition to other types of logging) in your web app, you may call the necessary methods in your controller’s action methods or equivalent. Here, you can log various levels of information, warnings and errors at various severity levels.

As seen in the snippet below, you have to do the following in your ***MVC Controller*** that you want to add Logging to:

1. Add using statement for Logging namespace
2. Add a private readonly variable for an ILogger object
3. Inject an ILogger<model> object into the constructor
4. Assign the private variable to the injected variable
5. Call various log logger methods as needed.

...  
using Microsoft.Extensions.Logging;  
  
public class MyController: Controller  
{  
 ...  
 private readonly ILogger \_logger;   
  
 public MyController(..., ILogger<MyController> logger)  
 {  
 ...  
 \_logger = logger;  
 }  
   
 public IActionResult MyAction(...)  
 {  
 \_logger.LogTrace("log trace");  
 \_logger.LogDebug("log debug");  
 \_logger.LogInformation("log info");  
 \_logger.LogWarning("log warning");  
 \_logger.LogError("log error");  
 \_logger.LogCritical("log critical");  
 }  
}

In Razor Pages, the logging code will go into the Page’s corresponding Model class. As seen in the snippet below, you have to do the following to the ***Model class that corresponds to a Razor Page***:

1. Add using statement for Logging namespace
2. Add a private readonly variable for an ILogger object
3. Inject an ILogger<model> object into the constructor
4. Assign the private variable to the injected variable
5. Call various log logger methods as needed.

...  
using Microsoft.Extensions.Logging;  
  
public class MyPageModel: PageModel  
{  
 ...  
 private readonly ILogger \_logger;  
  
 public MyPageModel(..., ILogger<MyPageModel> logger)  
 {  
 ...  
 \_logger = logger;   
 }  
 ...  
 public void MyPageMethod()   
 {  
 ...   
 \_logger.LogInformation("log info");   
 \_logger.LogError("log error");  
 ...  
 }  
}

You may have noticed that Steps 1 through 5 are pretty much identical for MVC and Razor Pages. This makes it very easy to quickly add all sorts of logging into your application, including error logging.

**Transient fault handling**

Although it’s beyond the scope of this article, it’s worth mentioning that you can avoid transient faults (e.g. temporary database connection losses) by using some proven patterns, practices and existing libraries. To get some history on transient faults, check out the following article from the classic “patterns & practices”. It describes the so-called “Transient Fault Handling Application Block”.

* Classic Patterns & Practices: <https://docs.microsoft.com/en-us/previous-versions/msp-n-p/dn440719(v=pandp.60)>

More recently, check out the docs on Transient Fault Handling:

* Docs: <https://docs.microsoft.com/en-us/aspnet/aspnet/overview/developing-apps-with-windows-azure/building-real-world-cloud-apps-with-windows-azure/transient-fault-handling>

And now in .NET Core, you can add resilience and transient fault handling to your .NET Core HttpClient with Polly!

* Adding Resilience and Transient Fault handling to your .NET Core HttpClient with Polly: <https://www.hanselman.com/blog/AddingResilienceAndTransientFaultHandlingToYourNETCoreHttpClientWithPolly.aspx>
* Integrating with Polly for transient fault handling: <https://www.stevejgordon.co.uk/httpclientfactory-using-polly-for-transient-fault-handling>
* Using Polly for .NET Resilience with .NET Core: <https://www.telerik.com/blogs/using-polly-for-net-resilience-and-transient-fault-handling-with-net-core>

You can get more information on the Polly project on the official Github page:

* Polly on Github: <https://github.com/App-vNext/Polly>

**References**

* try-catch-finally: <https://docs.microsoft.com/en-us/dotnet/csharp/language-reference/keywords/try-catch-finally>
* Handle errors in ASP.NET Core: <https://docs.microsoft.com/en-us/aspnet/core/fundamentals/error-handling>
* Use multiple environments in ASP.NET Core: <https://docs.microsoft.com/en-us/aspnet/core/fundamentals/environments>
* UseDeveloperExceptionPage: <https://docs.microsoft.com/en-us/dotnet/api/microsoft.aspnetcore.builder.developerexceptionpageextensions.usedeveloperexceptionpage>
* DeveloperExceptionPageOptions: <https://docs.microsoft.com/en-us/dotnet/api/microsoft.aspnetcore.builder.developerexceptionpageoptions>
* Logging: <https://docs.microsoft.com/en-us/aspnet/core/fundamentals/logging>
* Handle errors in ASP.NET Core Blazor apps: <https://docs.microsoft.com/en-us/aspnet/core/blazor/handle-errors?view=aspnetcore-3.1>