**Handling Errors in ASP .NET Core**

By Shahed C on February 25, 2019

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

This is the **eighth**of a [series of posts](https://wakeupandcode.com/aspnetcore/#aspnetcore2019) on ASP .NET Core in 2019. In this series, we’ll cover 26 topics over a span of 26 weeks from January through June 2019, titled **A-Z of ASP .NET Core!**

** A – Z of ASP .NET Core!**

**In this Article:**

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

This article will refer to the following sample code on GitHub:

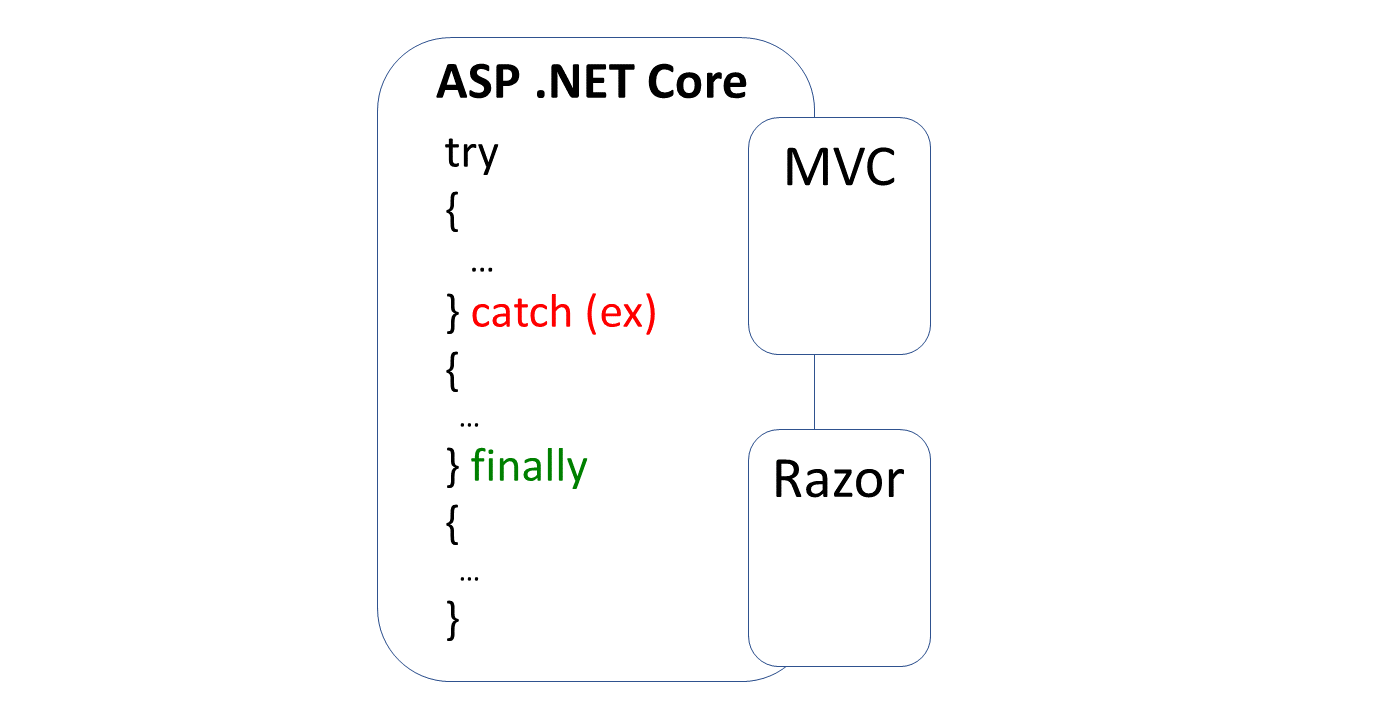
 Error Handling Samples: <https://github.com/shahedc/ErrorHandlingWeb>

**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.

[](https://wakeupandcode.com/wp-content/uploads/2019/02/Blog-Diagram-errors.png)

**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 Sample App**

In the [MVC sample app](https://github.com/shahedc/ErrorHandlingWeb/tree/master/ErrorHandlingMvc), the [Reader Controller](https://github.com/shahedc/ErrorHandlingWeb/blob/master/ErrorHandlingMvc/Controllers/ReaderController.cs) uses a [Data Service](https://github.com/shahedc/ErrorHandlingWeb/blob/master/ErrorHandlingLib/Services/DataService.cs) from a shared .NET Standard Library to read from a data file that may exist in the Web App’s static files. It displays a view with some hard-coded data and tries to replace some data with additional information obtained from the data file.

// hard-code some data  
var data1 = new DataItem  
{  
 Id = 1,  
 SomeData = "data 1 initialized"  
};   
var data2 = new DataItem  
{  
 Id = 2,  
 SomeData = "data 2 initialized"  
};   
var data3 = new DataItem  
{  
 Id = 3,  
 SomeData = "data 3 initialized"  
};

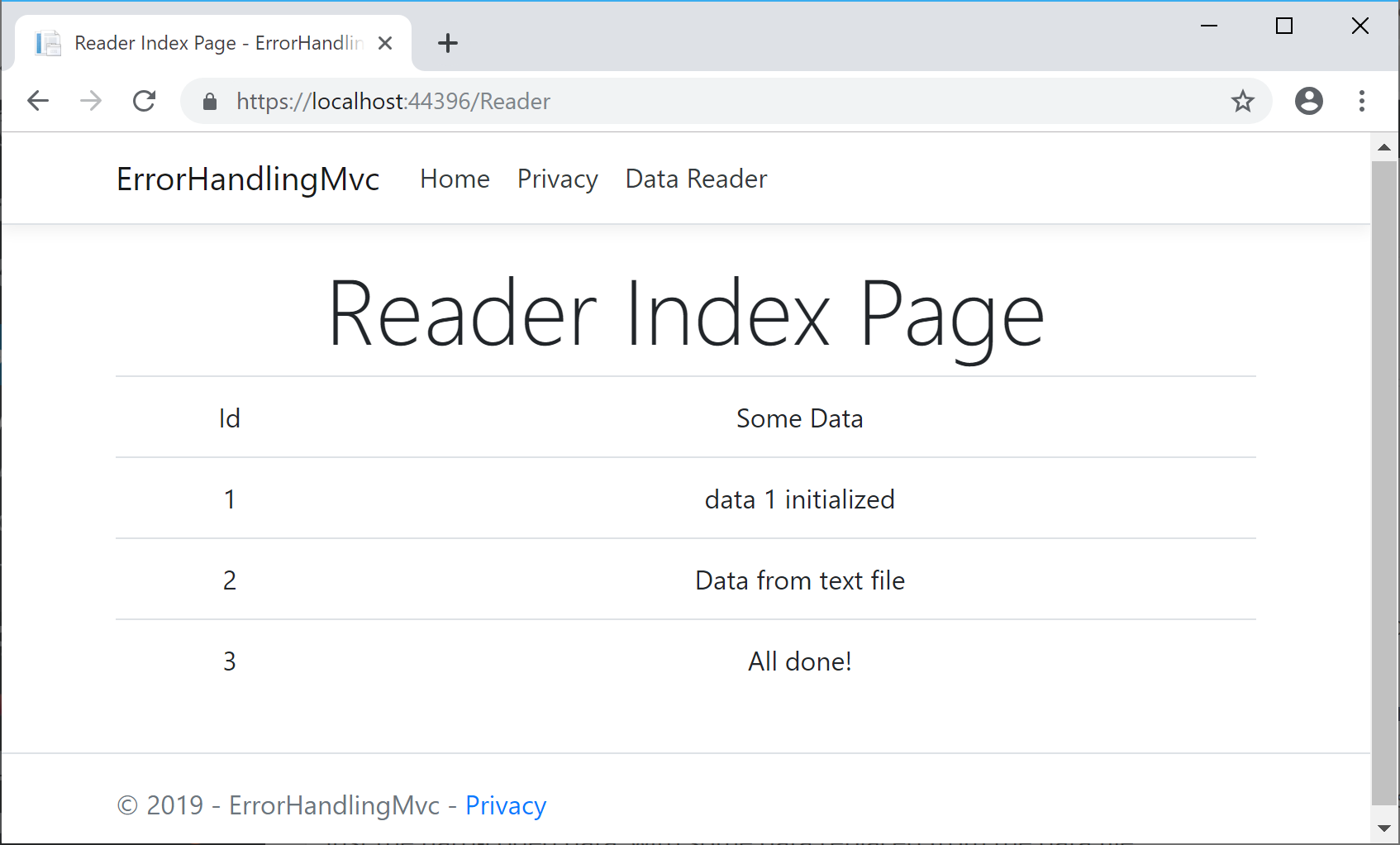
Then, try to read some data from a data file, to replace information in the data model.

// get data from file if possible  
try  
{ // Open the text file using a stream reader.  
 var webRoot = \_env.WebRootPath;  
 var file = System.IO.Path.Combine(webRoot, @"data\datafile.txt");  
  
 using (StreamReader sr = new StreamReader(file))  
 {  
 // Read the stream to a string, overwrite some data  
 data2.SomeData = sr.ReadToEnd();  
 }  
}  
catch (IOException ioException)  
{  
 data2.SomeData = $"IO Error: {ioException.Message}";  
}  
catch (Exception exception)  
{  
 data2.SomeData = $"Generic Error: {exception.Message}";  
}  
finally  
{  
 data3.SomeData = "All done!";  
}

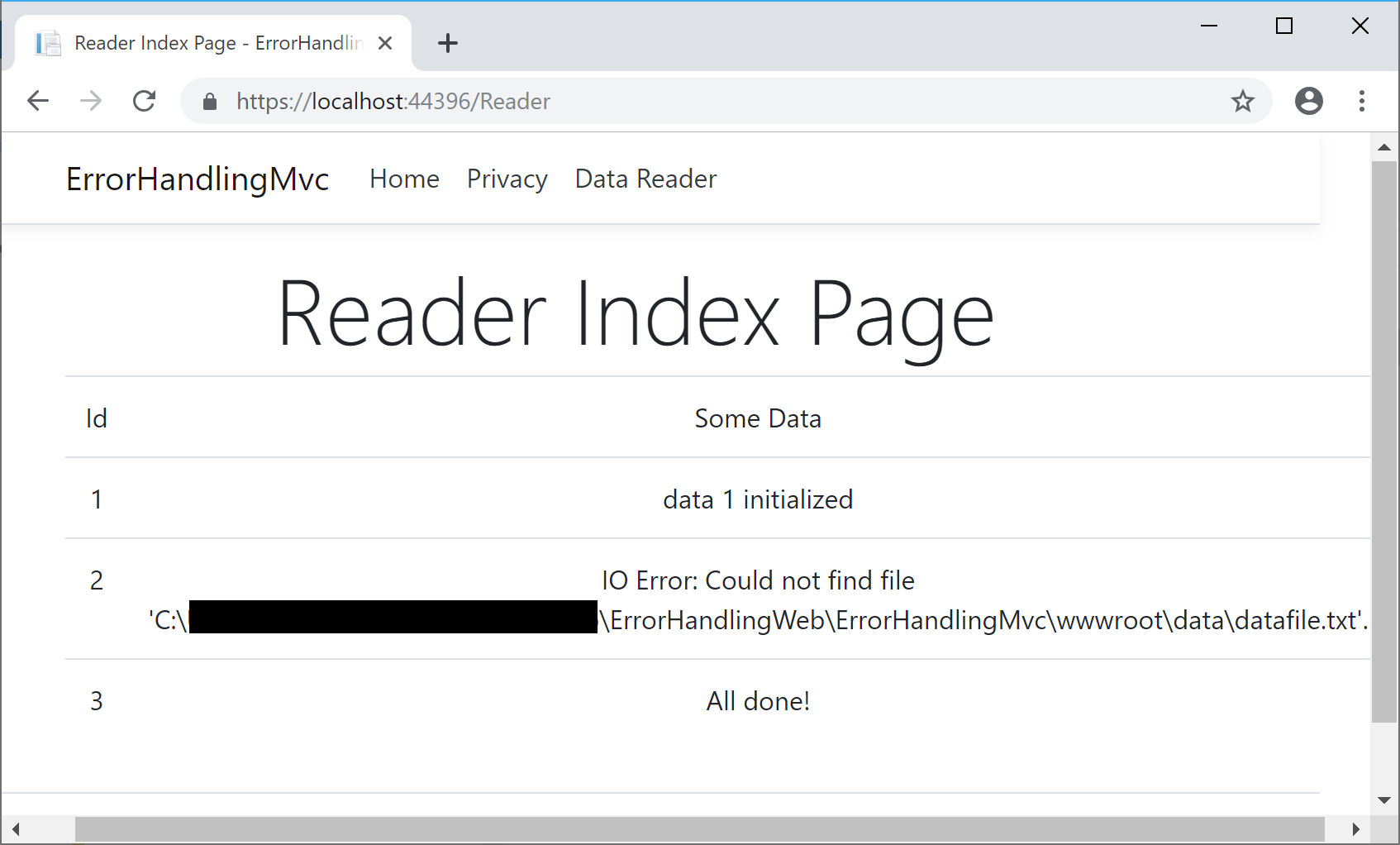
In the above code, you can see a series of try-catch blocks, ending with a finally block:

1. try to read the file and overwrite some data in the 2nd data object.
2. catch an **IOException**, capture the error message
3. catch a generic **Exception** if a different exception has occurred, capture the generic error message
4. ***finally***, overwrite some data in the 3rd data object whether or not any errors have occurred.

Run the MVC app and navigate to the Reader Controller. If there are no errors, you should see just the hard-coded data, with some data replaced from the data file.

[](https://wakeupandcode.com/wp-content/uploads/2019/02/ErrorApp-Success.png)

If you rename/delete the data file, then run the program again, you should see an error message as well. This reflects the IOException that occurred while attempting to read a missing file.

[](https://wakeupandcode.com/wp-content/uploads/2019/02/ErrorApp-IOError.png)

**Error Handling for MVC**

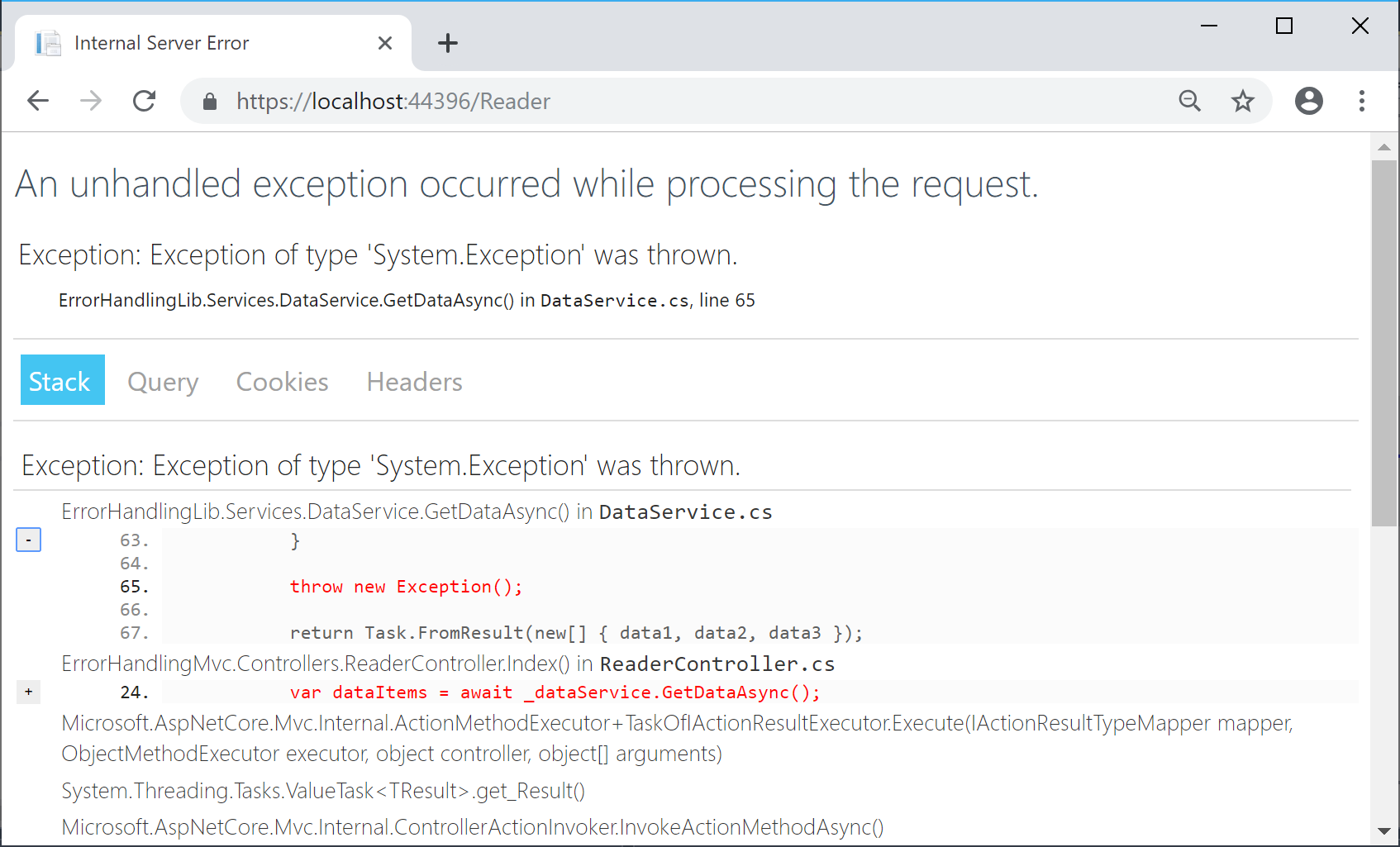
In ASP .NET Core MVC web apps, unhandled exceptions are typically handled in different ways, depending on whatever [environment](https://docs.microsoft.com/en-us/aspnet/core/fundamentals/environments) the app is running in. The default template uses the **DeveloperExceptionPage** middleware in a development 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();  
}  
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.

[](https://wakeupandcode.com/wp-content/uploads/2019/02/ErrorApp-MvcError-Dev-SourceLine.png)

For non-dev scenarios, the shared **Error** view can be further customized by updating the [Error.cshtml](https://github.com/shahedc/ErrorHandlingWeb/blob/master/ErrorHandlingMvc/Views/Shared/Error.cshtml) view in the **Shared** subfolder. The [ErrorViewModel](https://github.com/shahedc/ErrorHandlingWeb/blob/master/ErrorHandlingMvc/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 template’s [Home Controller](https://github.com/shahedc/ErrorHandlingWeb/blob/master/ErrorHandlingMvc/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   
 });  
}

**UPDATE:**

Wait… what about **Web API** in ASP .NET Core? After posting 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/ErrorHandlingWeb/blob/master/ErrorHandlingRazor/Startup.cs) file for the Razor Pages web app sample.

if (env.IsDevelopment())  
{  
 app.UseDeveloperExceptionPage();  
}  
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/ErrorHandlingWeb/blob/master/ErrorHandlingRazor/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)

**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>();

**NOTE**: The Web Host Builder is being replaced by the Generic Host Builder in ASP .NET Core 3.0, but you can expect similar initial behavior. 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/).

The host [sets up the logging configuration](https://github.com/aspnet/MetaPackages/blob/release/2.2/src/Microsoft.AspNetCore/WebHost.cs) for you, e.g.:

.ConfigureLogging((hostingContext, logging) =>  
{  
 logging.AddConfiguration(hostingContext.Configuration.GetSection("Logging"));  
 logging.AddConsole();  
 logging.AddDebug();  
 logging.AddEventSourceLogger();  
})

Since ASP .NET Core is open-source, you can find the above snippet (or something similar) on Github. Here is a link to the 2.2 release of WebHost:

* WebHost.cs at release/2.2: <https://github.com/aspnet/MetaPackages/blob/release/2.2/src/Microsoft.AspNetCore/WebHost.cs>

To make use of error logging (in addition to other types of logging) in your MVC web app, you may call the necessary methods in your controller’s action methods. 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>

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

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

**References**

* Read text from a file: <https://docs.microsoft.com/en-us/dotnet/standard/io/how-to-read-text-from-a-file>
* 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>
* 2.x Sample: <https://github.com/aspnet/Docs/tree/master/aspnetcore/fundamentals/error-handling/samples/2.x/ErrorHandlingSample>
* Razor Sample: <https://github.com/shahedc/_workshops/tree/master/AspNetCoreRazor/AspNetCoreRazor>
* MVC Sample: <https://github.com/shahedc/_workshops/tree/master/AspNetCoreMvc/MvcMovie>
* Logging: <https://docs.microsoft.com/en-us/aspnet/core/fundamentals/logging>