Handling Errors in ASP.NET Core:

Dealing with errors is a common factor we would face in any kind of application development, no matter what language we have used for the development. ASP.NET Core provides fabulous support for Error Handling strategies.

Before diving into the core part of the subject, it is highly recommended to have a basic understanding of concepts like C#, ASP.NET Core programming, OOP's concepts, Exception handling, try, catch, throw, finally, etc.

Now let us discuss a following approaches related to Error handling in ASP.NET Core.

Developer Exception Page - Exception Handling in Developer Environment

The *Developer Exception Page* displays detailed information about unhandled request exceptions. The ASP.NET Core templates generate the following code in Startup.cs file:

Razor Pages:

```
public void Configure(IApplicationBuilder app, IWebHostEnvironment env)
{
    if (env.IsDevelopment())
    {
        app.UseDeveloperExceptionPage();
    }
    else
    {
        app.UseExceptionHandler("/Error");
    }

    app.UseStaticFiles();

    app.UseRouting();

    app.UseAuthorization();

    app.UseEndpoints(endpoints =>
    {
        endpoints.MapRazorPages();
    });
}
```

Core MVC:

```
public void Configure(IApplicationBuilder app, IWebHostEnvironment env)
{
    if (env.IsDevelopment())
    {
        app.UseDeveloperExceptionPage();
    }
    else
    {
        app.UseExceptionHandler("/Home/Error");
    }
    app.UseStaticFiles();

    app.UseRouting();

    app.UseAuthorization();

    app.UseEndpoints(endpoints =>
    {
        endpoints.MapControllerRoute(
            name: "default",
            pattern: "{controller=Home}/{action=Index}/{id?}");
    });
}
```

The UseDeveloperExceptionPage extension method adds middleware into the request pipeline.

It enables the developer exception page **only** when the app is running in the Development environment. Detailed exception information shouldn't be displayed publicly when the app runs in the Production environment.

The templates place UseDeveloperExceptionPage early in the middleware pipeline so that it can catch unhandled exceptions thrown in middleware that follows.

The Developer Exception Page displays developer friendly detailed information about request exceptions. This helps developers in tracing errors that occur during development phase.

As this middleware displays sensitive information, it is advisable to add it only in development environment. The developer environment is a new feature in .NET Core.

The Developer Exception Page can include the following information about the exception and the request:

- Stack trace
- Query string parameters if any
- Cookies if any
- Headers
- Routing

The Developer Exception Page isn't guaranteed to provide any information. Use Logging for complete error information.

Example:

In Core MVC:

Change the code in Home Controller

Replace the Index method in the HomeController with the code below:

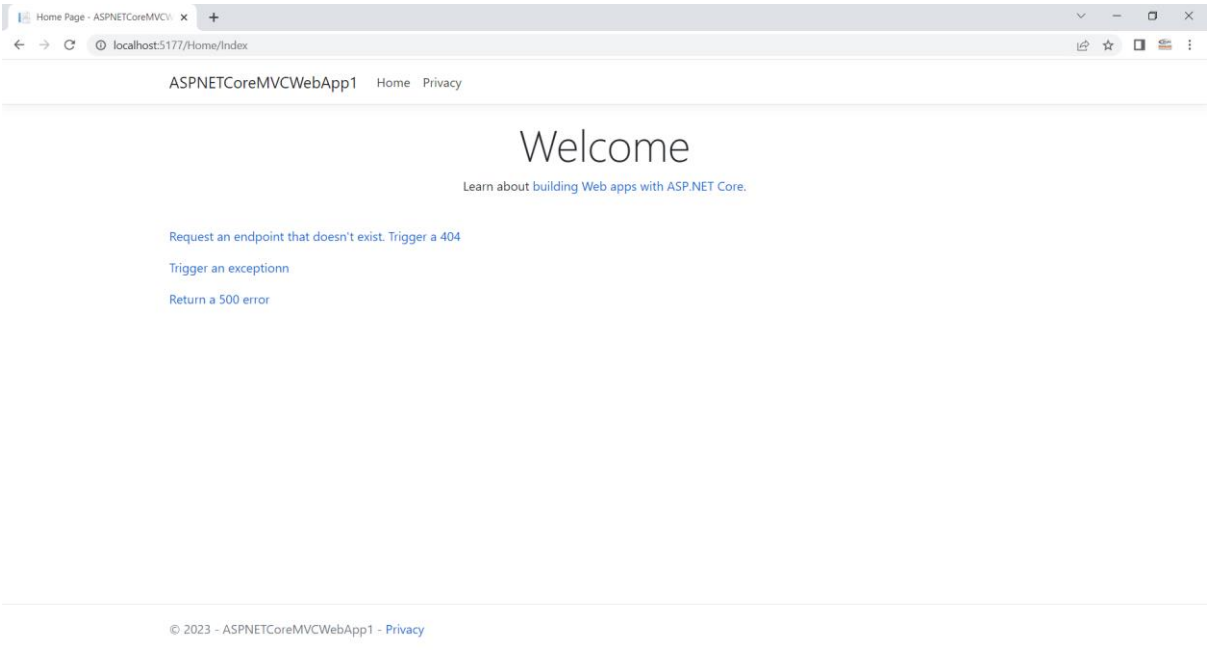
```
public IActionResult Index(int? id = null)
{
    if (id.HasValue)
    {
        if (id == 1)
        {
            throw new FileNotFoundException("File not found exception thrown in index.chtml");
        }
        else if (id == 2)
        {
            return StatusCode(500);
        }
    }
    return View();
}
```

Change code in Index view

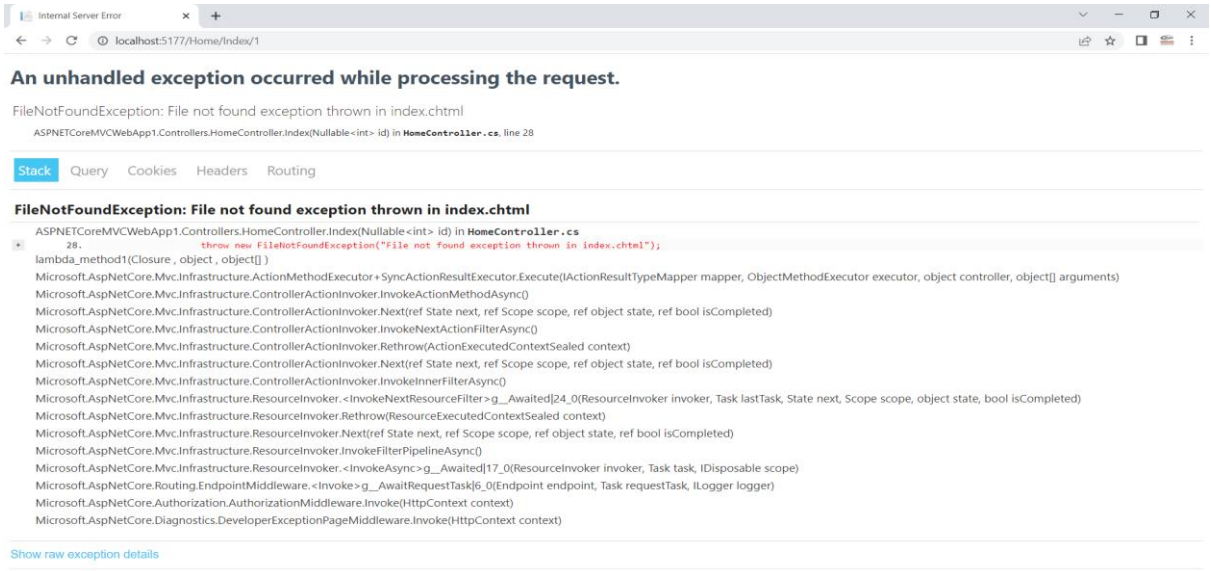
Add the code in the bottom of Home/Index view, i.e., the file Index.cshtml in Views/Home directory.

```
<br />

<div class="text-left">
    <p>
        <a href="/Home/About">
            Request an endpoint that doesn't exist. Trigger a 404
        </a>
    </p>
    <p><a href="/Home/Index/1">Trigger an exceptionn</a></p>
    <p><a href="/Home/Index/2">Return a 500 error</a></p>
</div>
```

Run app and Test



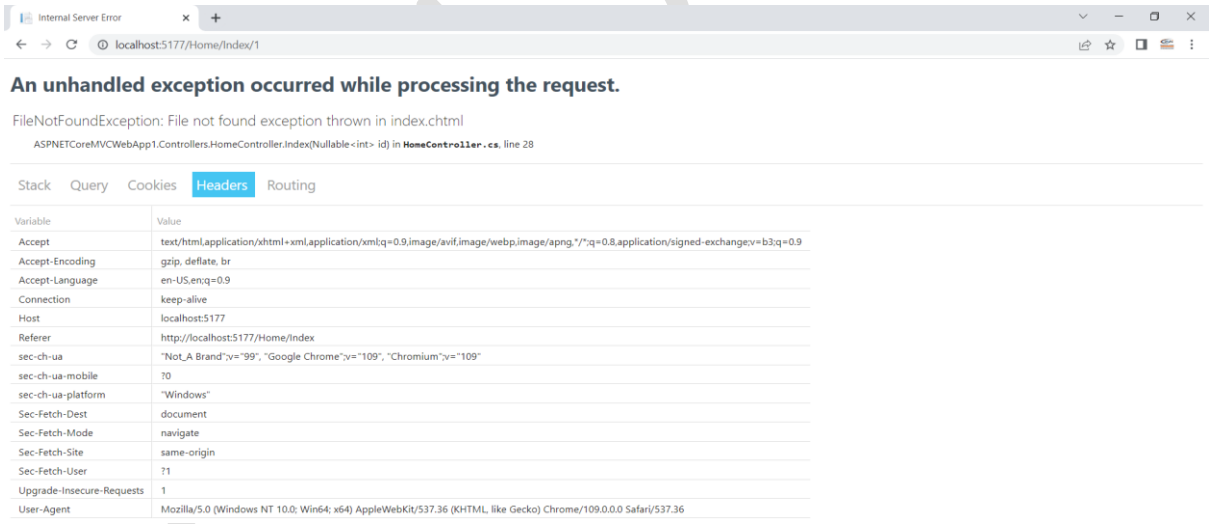
Click "Trigger an exception." you will get

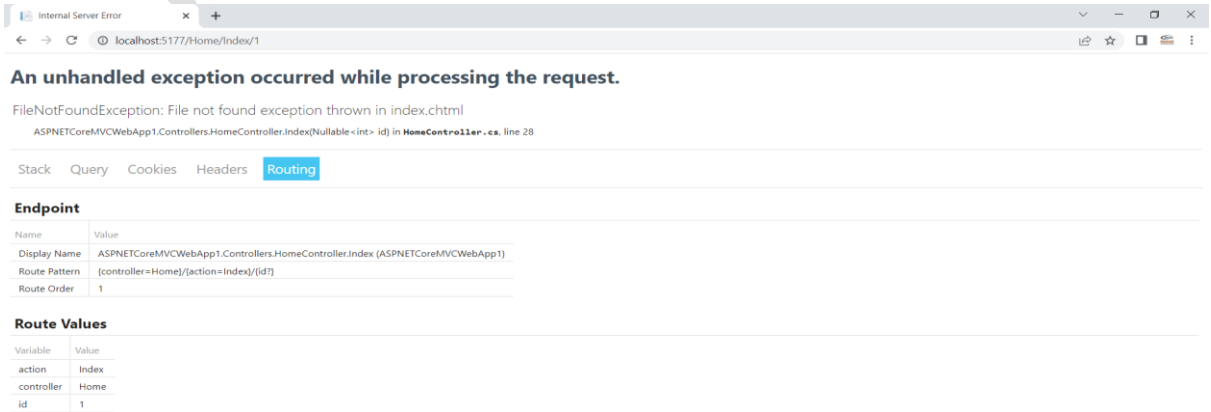


This is the Developer Exception Page that includes the following information about the exception and the request,

- Stack trace
- Query string parameters if any
- Cookies if any
- Headers
- Routing

For examples: Headers, and Routing





Exception Handler Page - Exception Handling in Production Environment

ASP.NET Core configures app behavior based on the runtime environment that is determined in launchSettings.json file:

- **Development:** The launchSettings.json file sets ASPNETCORE_ENVIRONMENT to Development on the local machine.
- **Staging**
- **Production:** The default if DOTNET_ENVIRONMENT and ASPNETCORE_ENVIRONMENT have not been set.

Note

The **launchSettings.json** file:

- It is only used on the local development machine.
- It is not deployed.
- It contains profile settings.

Now, we switch environment from Development to Production:

```
{
  "iisSettings": {
    "windowsAuthentication": false,
    "anonymousAuthentication": true,
    "iisExpress": {
      "applicationUrl": "http://localhost:5177",
      "sslPort": 0
    }
  },
  "profiles": {
    "IIS Express": {
      "commandName": "IISExpress",
      "launchBrowser": true,
      "environmentVariables": {
        //"ASPNETCORE_ENVIRONMENT": "Development",
        "ASPNETCORE_ENVIRONMENT": "Production"
      }
    },
    "ASPNETCoreMVCWebApp1": {
      "commandName": "Project",
      "dotnetRunMessages": true,
      "launchBrowser": true,
      "applicationUrl": "http://localhost:5000",
      "environmentVariables": {
        //"ASPNETCORE_ENVIRONMENT": "Development",
        "ASPNETCORE_ENVIRONMENT": "Production"
      }
    }
  }
}
```

Approach 1: UseExceptionHandler

1: Exception Handler Page

For the Production environment, ASP.NET Core handles exception by calling UseExceptionHandler in Startup.cs file Configure method as follows:

Razor Pages:

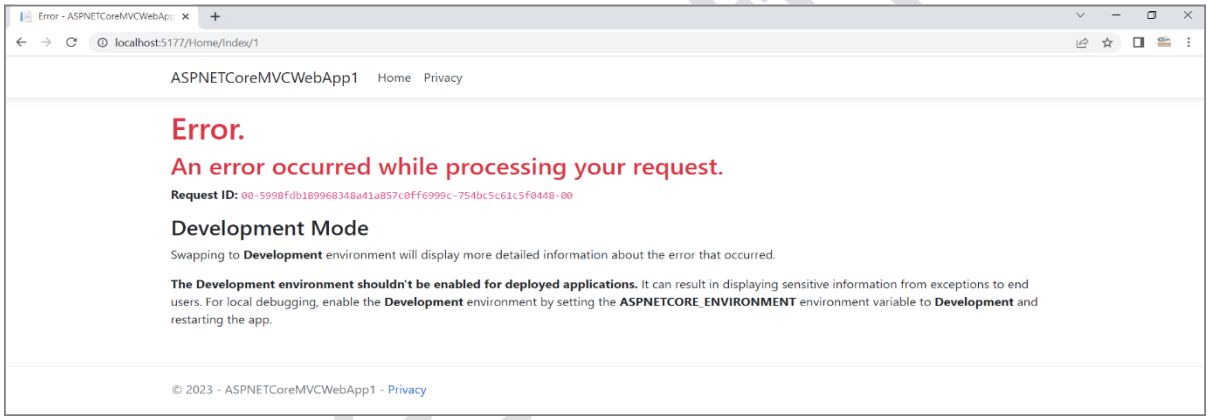
```
public void Configure(IApplicationBuilder app, IWebHostEnvironment env)
{
    if (env.IsDevelopment())
    {
        app.UseDeveloperExceptionPage();
    }
    else
    {
        app.UseExceptionHandler("/Error");
    }

    app.UseStaticFiles();

    app.UseRouting();

    app.UseAuthorization();

    app.UseEndpoints(endpoints =>
    {
        endpoints.MapRazorPages();
    });
}
```

Core MVC:

```
public void Configure(IApplicationBuilder app, IWebHostEnvironment env)
{
    if (env.IsDevelopment())
    {
        app.UseDeveloperExceptionPage();
    }
    else
    {
        app.UseExceptionHandler("/Home/Error");
    }
    app.UseStaticFiles();

    app.UseRouting();

    app.UseAuthorization();

    app.UseEndpoints(endpoints =>
    {
        endpoints.MapControllerRoute(
            name: "default",
            pattern: "{controller=Home}/{action=Index}/{id?}");
    });
}
```

Run the app, and Click **Trigger an exception** link in the Home page, we will get the Exception Handler Page.

The Razor Pages app template provides an Error page (.cshtml) and PageModel class (ErrorModel) in the *Pages* folder. For an MVC app, the project template includes an Error action method and an Error view for the Home controller.



This exception handling middleware:

- Catches and logs unhandled exceptions.
- Re-executes the request in an alternate pipeline using the path indicated. The request isn't re-executed if the response has started. The template generated code re-executes the request using the /Error path in Razor Pages and /Home/Error path in MVC.

Warning

- If the alternate pipeline throws an exception of its own, Exception Handling Middleware re-throws the original exception.

The exception handling middleware re-executes the request using the *original* HTTP method. If an error handler endpoint is restricted to a specific set of HTTP methods, it runs only for those HTTP methods. For example, an MVC controller action that uses the **[HttpGet]** attribute runs only for GET requests. To ensure that *all* requests reach the custom error handling page, don't restrict them to a specific set of HTTP methods.

To handle exceptions differently based on the original HTTP method:

- For Razor Pages, create multiple handler methods. For example, use **OnGet** to handle GET exceptions and use **OnPost** to handle POST exceptions.
- For MVC, apply HTTP verb attributes to multiple actions. For example, use **[HttpGet]** to handle GET exceptions and use **[HttpPost]** to handle POST exceptions.

To allow unauthenticated users to view the custom error handling page, ensure that it supports anonymous access.

Access the exception

Use **IExceptionHandlerPathFeature** to access the exception and the original request path in an **Error** action. The following code adds **ExceptionMessage** to the default controller (Home) and action (Error) i.e. Home/Error generated by the ASP.NET Core MVC templates:

Controllers -> HomeController.cs:

```
using ASPNETCoreMVCWebApp1.Models;
using Microsoft.AspNetCore.Diagnostics;
using Microsoft.AspNetCore.Mvc;
using Microsoft.Extensions.Logging;
using System.Diagnostics;
using System.IO;

namespace ASPNETCoreMVCWebApp1.Controllers
{
  public class HomeController : Controller
  {
        ……….
        ……….
        ……….
        ……….
        ……….

        [ResponseCache(Duration = 0, Location = ResponseCacheLocation.None, NoStore = true)]
        public IActionResult Error()
        {
              string exceptionMessage = string.Empty;

              var exceptionHandlerPathFeature =
                    HttpContext.Features.Get<IExceptionHandlerPathFeature>();
              if (exceptionHandlerPathFeature?.Error is FileNotFoundException)
              {
                    exceptionMessage = "File error thrown";
                    _logger.LogError(exceptionMessage);
              }
              if (exceptionHandlerPathFeature?.Path == "/Home/Index/1")
              {
                    exceptionMessage += " from home page";
              }

              ErrorViewModel errorViewModel = new ErrorViewModel()
              {
                    RequestId = Activity.Current?.Id ?? HttpContext.TraceIdentifier,
                    ExceptionMessage = exceptionMessage
              };

              return View(errorViewModel);
        }
  }
}
```

Models -> ErrorViewModel.cs:

```
namespace ASPNETCoreMVCWebApp1.Models
{
  public class ErrorViewModel
  {
    public string RequestId { get; set; }

    public bool ShowRequestId => !string.IsNullOrEmpty(RequestId);

    public string ExceptionMessage { get; set; }
  }
}
```

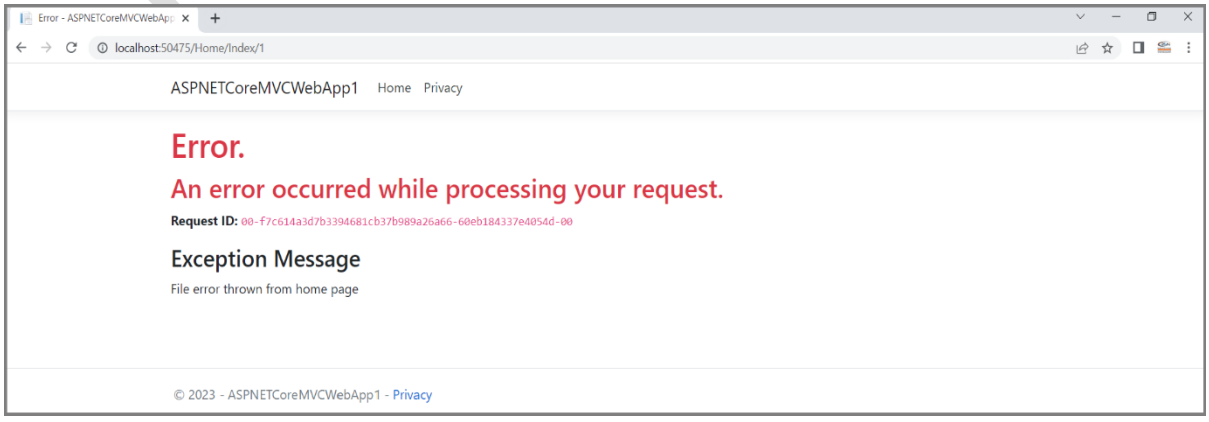
Views -> Shared -> Error.cshtml:

```
Error.cshtml*
1  @model ErrorViewModel
2  @{
3      ViewData["Title"] = "Error";
4  }
5
6  <h1 class="text-danger">Error.</h1>
7  <h2 class="text-danger">An error occurred while processing your request.</h2>
8
9  @if (Model.ShowRequestId)
10 {
11     <p>
12         <strong>Request ID:</strong> <code>@Model.RequestId</code>
13     </p>
14 }
15
16 <h3>Exception Message</h3>
17 <p>
18     @Model.ExceptionMessage
19 </p>
```

Warning: Do not serve sensitive error information to clients. Serving errors is a security risk.

To test the exception run the application:

- Set the environment to **Production**.
- Select '**Trigger an exception'** on the Home page.



2: Exception Handler Lambda

An alternative to a custom exception handler page is to provide a lambda to **UseExceptionHandler**. Using a lambda allows access to the error before returning the response.

The following code uses a lambda for exception handling:

Startup.cs file:

```
using Microsoft.AspNetCore.Diagnostics;
using Microsoft.AspNetCore.Http;
using System.IO;
using System.Net;
public void Configure(IApplicationBuilder app, IWebHostEnvironment env)
{
   if (env.IsDevelopment())
   {
      app.UseDeveloperExceptionPage();
   }
   else
   {
      app.UseExceptionHandler(errorApp =>
      {
         errorApp.Run(async context =>
         {
            context.Response.StatusCode = (int)HttpStatusCode.InternalServerError;
            context.Response.ContentType = "text/html";

            await context.Response.WriteAsync("<html lang=\"en\">\r\n<body>\r\n");
            await context.Response.WriteAsync("<h1 style='color:red'>ERROR!</h1>\r\n");
            await context.Response.WriteAsync(
                              "<h2 style='color:red'>An error occurred while processing your request.</h2>\r\n");

            var exceptionHandlerPathFeature =
               context.Features.Get<IExceptionHandlerPathFeature>();

            if (exceptionHandlerPathFeature?.Error is FileNotFoundException)
            {
               await context.Response.WriteAsync("<h3 style='color:red'>File error thrown!</h3>\r\n");
            }

            await context.Response.WriteAsync("<a href=\"/\">Home</a><br/>\r\n");
            await context.Response.WriteAsync("</body>\r\n</html>\r\n");
         });
      });
   }
   app.UseStaticFiles();

   app.UseRouting();

   app.UseAuthorization();

   app.UseEndpoints(endpoints =>
   {
      endpoints.MapControllerRoute(
         name: "default",
         pattern: "{controller=Home}/{action=Index}/{id?}");
   });
}
```

Warning:

Do not serve sensitive error information from **IExceptionHandlerFeature** or **IExceptionHandlerPathFeature** to clients. Serving errors is a security risk.

To test the exception handling lambda run the application:

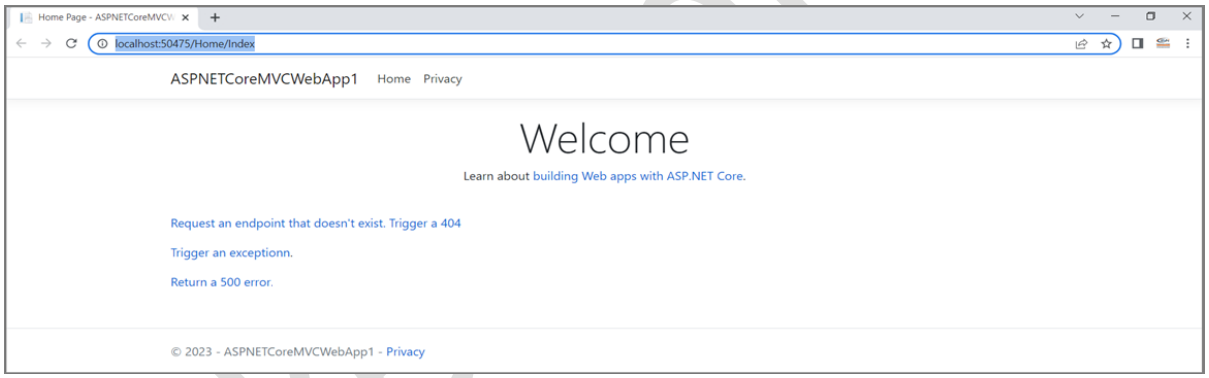
- Set the environment to **Production**.
- Select '**Trigger an exception**' on the Home page.

Approach 2: UseStatusCodePages

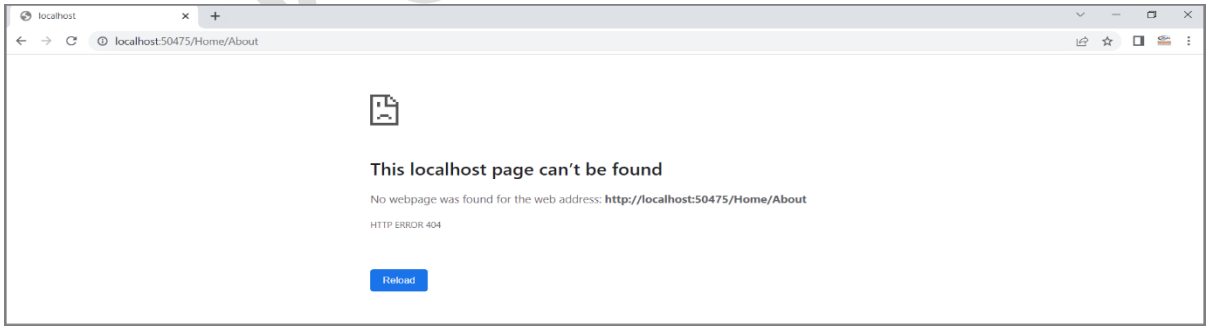
The above techniques discussed so far deal with the unhandled exceptions arising from code. However, that's not the only source of errors. Many times errors are generated due to internal server errors, non-existent pages, web server authorization issues and so on. These errors are reflected by the HTTP status codes such as 500, 404 and 401.

By default, an ASP.NET Core app doesn't provide a status code page for HTTP error status codes, such as 404 - Not Found. When the app sets an HTTP 400-599 error status code that doesn't have a body, it returns the status code and an empty response body.

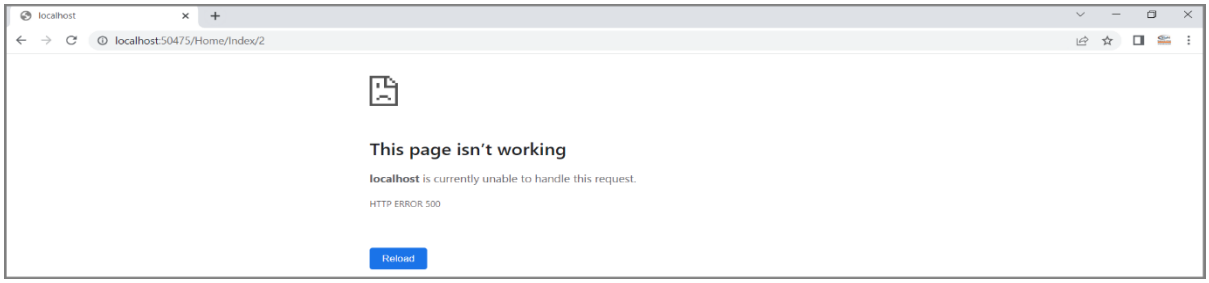
Click the link: Request an endpoint that doesn't exist. Trigger a 404 below,



We will get as shown below:



While clicking Return a 500 error



To deal with such errors we can use the status code pages middleware which provides status code pages.

To enable default text-only handlers for common error status codes, call **UseStatusCodePages()** in the **Startup**.**Configure** method:

```
public void Configure(IApplicationBuilder app, IWebHostEnvironment env)
{
    if (env.IsDevelopment())
    {
        app.UseDeveloperExceptionPage();
    }
    else
    {
        app.UseExceptionHandler("/Home/Error");
    }

    app.UseStatusCodePages();

    app.UseStaticFiles();

    app.UseRouting();

    app.UseAuthorization();

    app.UseEndpoints(endpoints =>
    {
        endpoints.MapControllerRoute(
            name: "default",
            pattern: "{controller=Home}/{action=Index}/{id?}");
    });
}
```

Call **UseStatusCodePages** before request handling middleware.

For example, call **UseStatusCodePages** before the Static File Middleware and the Endpoints Middleware.

When **UseStatusCodePages** isn't used, navigating to a URL without an endpoint returns a browser-dependent error message indicating the endpoint can't be found. For example, navigating to **Home/About**. When **UseStatusCodePages** is called, the browser returns:

And below for 500 error:

UseStatusCodePages isn't typically used in production because it returns a message that isn't useful to users.

Note

The status code pages middleware does not catch exceptions. To provide a custom error handling page, use the **exception handler page**.

UseStatusCodePages with format string:

To customize the response content type and text, use the overload of **UseStatusCodePages** that takes a content type and format string:

```
public void Configure(IApplicationBuilder app, IWebHostEnvironment env)
{
    if (env.IsDevelopment())
    {
        app.UseDeveloperExceptionPage();
    }
    else
    {
        app.UseExceptionHandler("/Home/Error");
    }

    app.UseStatusCodePages("text/plain", "Status code page, status code: {0}");

    app.UseStaticFiles();

    app.UseRouting();

    app.UseAuthorization();

    app.UseEndpoints(endpoints =>
    {
        endpoints.MapControllerRoute(
            name: "default",
            pattern: "{controller=Home}/{action=Index}/{id?}");
    });
}
```

In the preceding code, {0} is a placeholder for the error code.

Run the app, the result will be shown as below:

404 error:

And below for 500 error:

UseStatusCodePages with a **format string** isn't typically used in production because it returns a message that isn't useful to users.

UseStatusCodePages with lambda:

To specify custom error-handling and response-writing code, use the overload of **UseStatusCodePages** that takes a lambda expression:

```
app.UseStatusCodePages(async context =>
{
    context.HttpContext.Response.ContentType = "text/plain";

    await context.HttpContext.Response.WriteAsync("Status code page, status code: " + context.HttpContext.Response.StatusCode);
});
```

Run the app, the result will be shown as below:

404 error:

And below for 500 error:

UseStatusCodePages with a lambda isn't typically used in production because it returns a message that isn't useful to users.

UseStatusCodePagesWithRedirects:

The UseStatusCodePagesWithRedirects extension method:

- Sends a **302 - Found** status code to the client.
- Redirects the client to the error handling endpoint provided in the URL template. The error handling endpoint typically displays error information and returns HTTP 200.

```
public void Configure(IApplicationBuilder app, IWebHostEnvironment env)
{
    if (env.IsDevelopment())
    {
        app.UseDeveloperExceptionPage();
    }
    else
    {
        app.UseExceptionHandler("/Home/Error");
    }

    app.UseStatusCodePagesWithRedirects("/Home/MyStatusCode?code={0}");

    app.UseStaticFiles();

    app.UseRouting();

    app.UseAuthorization();

    app.UseEndpoints(endpoints =>
    {
        endpoints.MapControllerRoute(
            name: "default",
            pattern: "{controller=Home}/{action=Index}/{id?}");
    });
}
```

The URL template can include a **{0}** placeholder for the status code, as shown in the preceding code. If the URL template starts with **~** (tilde), the **~** is replaced by the app's **PathBase**.

This method is commonly used when the app:

- Should redirect the client to a different endpoint, usually in cases where a different app processes the error. For web apps, the client's browser address bar reflects the redirected endpoint.
- Shouldn't preserve and return the original status code with the initial redirect response.

Add an Action method in HomeController as shown below:

```
public IActionResult MyStatusCode(int code)
{
        if (code == 404)
        {
          ViewBag.ErrorMessage = "The requested page not found.";
        }
        else if (code == 500)
        {
          ViewBag.ErrorMessage = "My custom 500 error message.";
        }
        else
        {
          ViewBag.ErrorMessage = "An error occurred while processing your request.";
        }

        ViewBag.RequestId = Activity.Current?.Id ?? HttpContext.TraceIdentifier;
        ViewBag.ShowRequestId = !string.IsNullOrEmpty(ViewBag.RequestId);
        ViewBag.ErrorStatusCode = code;

        return View();
}
```

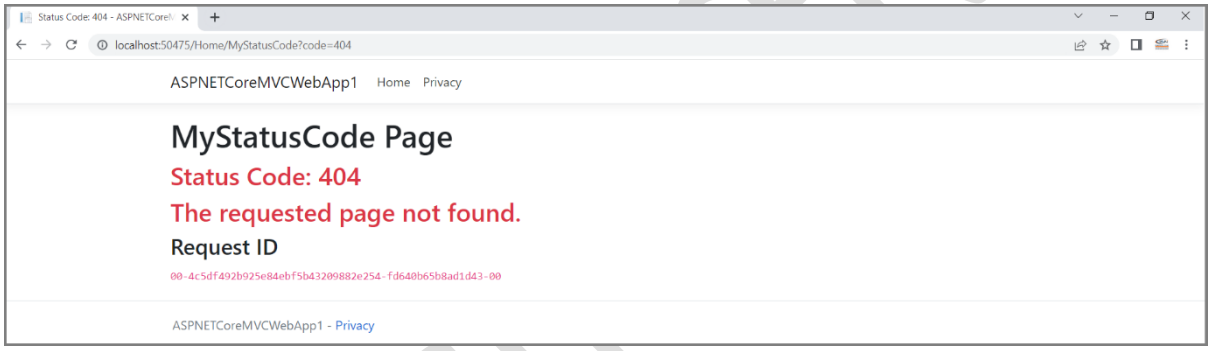
Create a view for the Action: Views/Home/MyStatusCode.cshtml

MyStatusCode.cshtml*

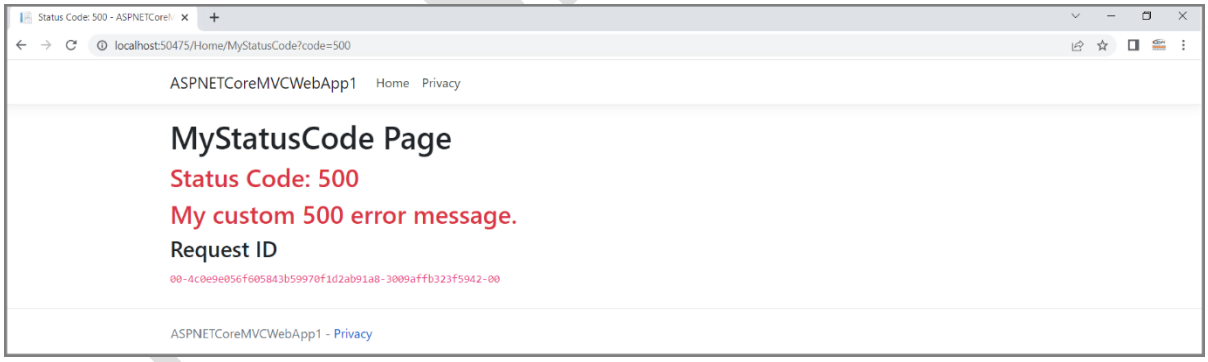
```
@{
    ViewData["Title"] = "Status Code" + ViewBag.ErrorStatusCode;
}

<h1>MyStatusCode Page</h1>
<h2 class="text-danger">Status Code: @ViewBag.ErrorStatusCode</h2>
<h2 class="text-danger"> @ViewBag.ErrorMessage</h2>

@if (ViewBag.ShowRequestId)
{
    <h3>Request ID</h3>
    <p>
        <code>@ViewBag.RequestId</code>
    </p>
}
```

Run the app, the result will be shown as below:

404 error:



And below for 500 error:



Note

The link is redirected to a new link that is endpoint provided.

UseStatusCodePagesWithReExecute:

The **UseStatusCodePagesWithReExecute** extension method:

- Returns the original status code to the client.
- Generates the response body by re-executing the request pipeline using an alternate path.

In Configure() method of Startup file (Startup.cs):

```
app.UseStatusCodePagesWithReExecute("/Home/MyStatusCode2", "?code={0}");
```

Ensure **UseStatusCodePagesWithReExecute** is placed before **UseRouting** so the request can be rerouted to the status page.

This method is commonly used when the app should:

- Process the request without redirecting to a different endpoint. For web apps, the client's browser address bar reflects the originally requested endpoint.
- Preserve and return the original status code with the response.

The URL and query string templates may include a placeholder **{0}** for the status code. The URL template must start with **/**.

The endpoint that processes the error can get the original URL that generated the error, as shown in the following example:

Add an Action method in HomeController as shown below:

```
public IActionResult MyStatusCode2(int code)
{
        var statusCodeReExecuteFeature = HttpContext.Features.Get<IStatusCodeReExecuteFeature>();

        if (statusCodeReExecuteFeature != null)
        {
           ViewBag.OriginalURL =
              statusCodeReExecuteFeature.OriginalPathBase
              + statusCodeReExecuteFeature.OriginalPath
              + statusCodeReExecuteFeature.OriginalQueryString;
        }

        ViewBag.RequestId = Activity.Current?.Id ?? HttpContext.TraceIdentifier;
        ViewBag.ShowRequestId = !string.IsNullOrEmpty(ViewBag.RequestId);
        ViewBag.ShowOriginalURL = !string.IsNullOrEmpty(ViewBag.OriginalURL);
        ViewBag.ErrorStatusCode = code;

        return View();
}
```

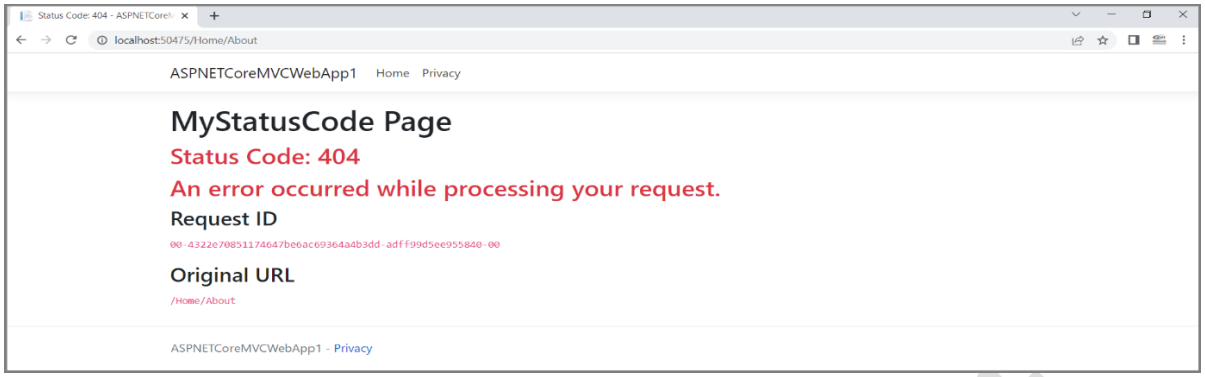
Create a view for the Action: Views/Home/MyStatusCode2.cshtml

MyStatusCode2.cshtml*

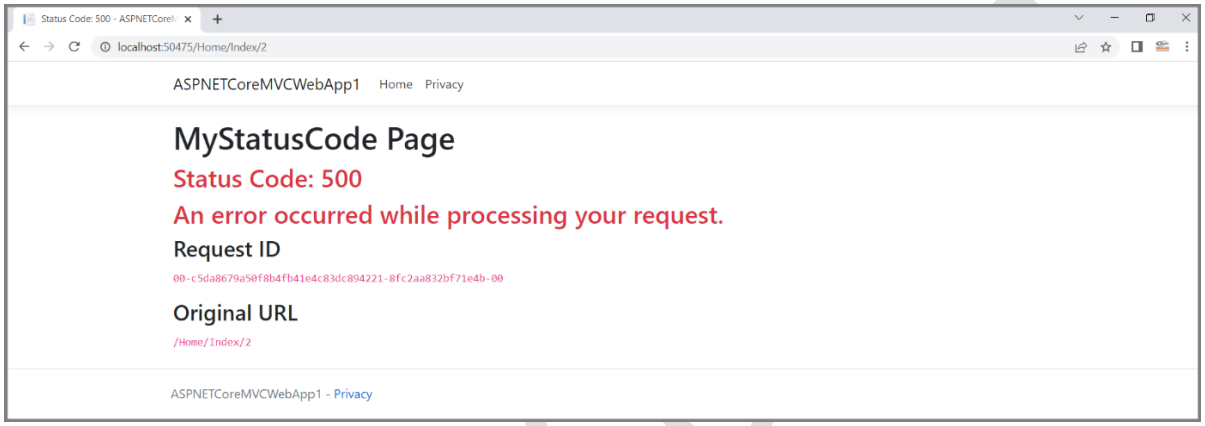
```
@{
    ViewData["Title"] = "Status Code: " + ViewBag.ErrorStatusCode;
}

<h1>MyStatusCode Page</h1>
<h2 class="text-danger">Status Code: @ViewBag.ErrorStatusCode</h2>
<h2 class="text-danger">An error occurred while processing your request.</h2>

@if (ViewBag.ShowRequestId)
{
    <h3>Request ID</h3>
    <p>
        <code>@ViewBag.RequestId</code>
    </p>}

@if (ViewBag.ShowOriginalURL)
{
    <h3>Original URL</h3>
    <p>
        <code>@ViewBag.OriginalURL</code>
    </p>
}
```

Run the app, the result will be shown as below:

404 error:



And below for 500 error:



Note

The link is kept the same with original one.

<u>Disable status code pages</u>:

To disable status code pages for an MVC controller or action method, use the [**SkipStatusCodePages**] attribute.

To disable status code pages for specific requests in a Razor Pages handler method or in an MVC controller, use **IStatusCodePagesFeature**:

```
// using Microsoft.AspNetCore.Diagnostics;
var statusCodePagesFeature = HttpContext.Features.Get<IStatusCodePagesFeature>();

if (statusCodePagesFeature != null)
{
    statusCodePagesFeature.Enabled = false;
}
```

Example:

```
[SkipStatusCodePages]
public IActionResult Index(int? id = null)
{
    if (id.HasValue)
    {
        if (id == 1)
        {
            throw new FileNotFoundException
                ("File not found exception thrown in Home/Index");
        }
        else if (id == 2)
        {
            return StatusCode(500);
        }
    }
    return View();
}
```

OR

```
public IActionResult Index(int? id = null)
{
    if (id.HasValue)
    {
        if (id == 1)
        {
            throw new FileNotFoundException
                ("File not found exception thrown in Home/Index");
        }
        else if (id == 2)
        {
            // using Microsoft.AspNetCore.Diagnostics;
            var statusCodePagesFeature = HttpContext.Features
                                                .Get<IStatusCodePagesFeature>();

            if (statusCodePagesFeature != null)
            {
                statusCodePagesFeature.Enabled = false;
            }

            return StatusCode(500);
        }
    }
    return View();
}
```

Exception-handling code:

Code in exception handling pages can also throw exceptions. Production error pages should be tested thoroughly and take extra care to avoid throwing exceptions of their own.

Response headers:

Once the headers for a response are sent:

- The app can't change the response's status code.
- Any exception pages or handlers can't run. The response must be completed or the connection aborted.

Server exception handling:

In addition to the exception handling logic in an app, the HTTP server implementation can handle some exceptions. If the server catches an exception before response headers are sent, the server sends a **500 - Internal Server Error** response without a response body. If the server catches an exception after response headers are sent, the server closes the connection. Requests that aren't handled by the app are handled by the server. Any exception that occurs when the server is handling the request is handled by the server's exception handling. The app's custom error pages, exception handling middleware, and filters don't affect this behavior.

Database error page:

3.1:

Database Error Page Middleware captures database-related exceptions that can be resolved by using Entity Framework migrations. When these exceptions occur, an HTML response with details of possible actions to resolve the issue is generated. This page should be enabled only in the Development environment. Enable the page by adding code to **Startup.Configure**:

```
if (env.IsDevelopment())
{
    app.UseDatabaseErrorPage();
}
```

UseDatabaseErrorPage requires the Microsoft.AspNetCore.Diagnostics.EntityFrameworkCore NuGet package.

Note: The **DatabaseErrorPageMiddleware** and its associated extension methods were marked as obsolete in ASP.NET Core 5.0. The middleware and extension methods will be removed in ASP.NET Core 6.0. The functionality will instead be provided by **DatabaseDeveloperPageExceptionFilter** and its extension methods available in **Microsoft.Extensions.DependencyInjection** namespace.

5.0:

DatabaseDeveloperPageExceptionFilter:

In combination with **UseDeveloperExceptionPage**, this captures database-related exceptions that can be resolved by using Entity Framework migrations. When these exceptions occur, an HTML response is generated with details of possible actions to resolve the issue. This page is enabled only in the Development environment.

Add the database developer page exception filter to the services collection. For example, call the **AddDatabaseDeveloperPageExceptionFilter** method in **Startup.ConfigureServices**:

```
public void ConfigureServices(IServiceCollection services)
{
   services.AddControllersWithViews();
   services.AddDbContext<ApplicationDbContext>(options =>
   {
            options.UseSqlServer(Configuration.GetConnectionString("conStr"));
  });
  services.AddDatabaseDeveloperPageExceptionFilter();
}
```

6.0/7.0:

Program.cs:

```
var builder = WebApplication.CreateBuilder(args);
builder.Services.AddDatabaseDeveloperPageExceptionFilter();
builder.Services.AddControllersWithViews();
```

Exception filters:

General Discussion

In MVC apps, exception filters can be configured globally or on a per-controller or per-action basis. In Razor Pages apps, they can be configured globally or per page model. These filters handle any unhandled exceptions that occur during the execution of a controller action or another filter.

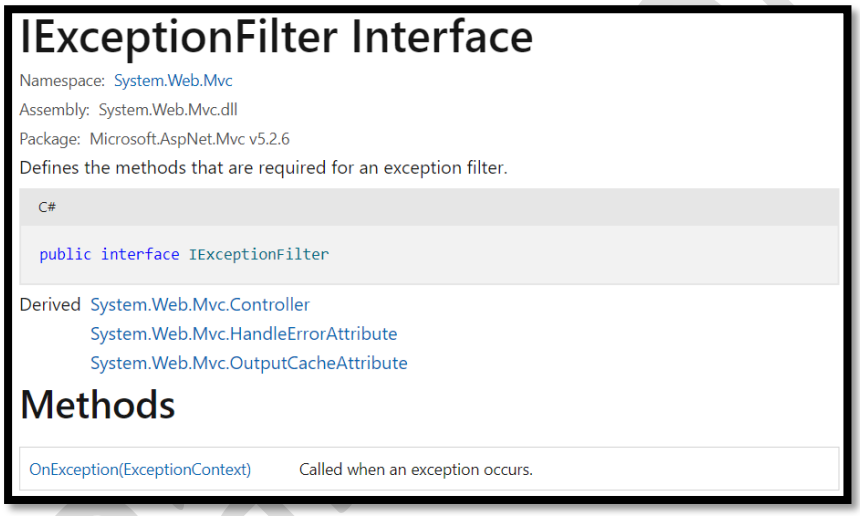
Exception filters are useful for trapping exceptions that occur within MVC actions, but they're not as flexible as the built-in exception handling middleware, UseExceptionHandler. Microsoft recommend using UseExceptionHandler, unless you need to perform error handling differently based on which MVC action is chosen.

Difference from ASP.NET MVC

In ASP.NET MVC, Exception Filter is the major approach for exception handling, while for ASP.NET Core MVC, as Microsoft suggested, the built-in exception handling middleware, UseExceptionHandler, is more flexible and suitable.

IExceptionFilter Interface for ASP.NET MVC is derived by System.Web.Mvc.HandleErrorAttribute and System.Web.Mvc.Controller, therefore, we can either overriding OnException method from a class derived from HandleErrorAttribute class, or directly overriding OnException method from a controller. However, **IExceptionFilter** Interface for ASP.NET Core is only derived by **Microsoft.AspNetCore.Mvc.Filters.ExceptionFilterAttribute**, not by Controller any more. So, we have to implement IExceptionFilter interface directly or from ExceptionFilterAttribute class, but not from Controller directly any more.

ASP.NET MVC



IExceptionFilter Interface

Namespace: System.Web.Mvc

Assembly: System.Web.Mvc.dll

Package: Microsoft.AspNet.Mvc v5.2.6

Defines the methods that are required for an exception filter.

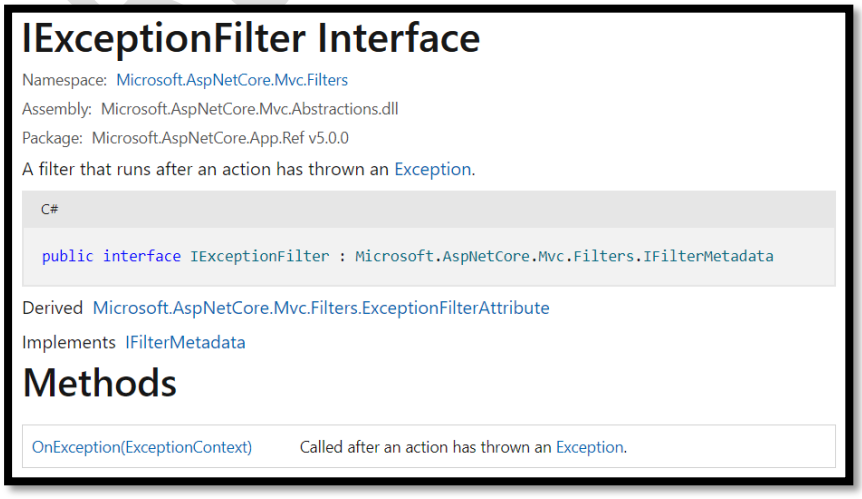
```
C#
public interface IExceptionFilter
```

Derived System.Web.Mvc.Controller
System.Web.Mvc.HandleErrorAttribute
System.Web.Mvc.OutputCacheAttribute

Methods

OnException(ExceptionContext)	Called when an exception occurs.

ASP.NET Core:



IExceptionFilter Interface

Namespace: Microsoft.AspNetCore.Mvc.Filters

Assembly: Microsoft.AspNetCore.Mvc.Abstractions.dll

Package: Microsoft.AspNetCore.App.Ref v5.0.0

A filter that runs after an action has thrown an Exception.

```
C#
public interface IExceptionFilter : Microsoft.AspNetCore.Mvc.Filters.IFilterMetadata
```

Derived Microsoft.AspNetCore.Mvc.Filters.ExceptionFilterAttribute

Implements IFilterMetadata

Methods

OnException(ExceptionContext)	Called after an action has thrown an Exception.

Exception filters

- Implement **IExceptionFilter** or **IAsyncExceptionFilter**.
- Can be used to implement common error handling policies.

The following sample exception filter uses a custom error view to display details about exceptions that occur when the app is in development:

Implementation

Step 1

Create a Custom Exception Filter Class: **Filters -> CustomExceptionFilter.cs**

```
using Microsoft.AspNetCore.Mvc;
using Microsoft.AspNetCore.Mvc.Filters;
using Microsoft.AspNetCore.Mvc.ModelBinding;
using Microsoft.AspNetCore.Mvc.ViewFeatures;

namespace ASPNETCoreMVCWebApp1.Filters
{
  public class CustomExceptionFilter: IExceptionFilter
  {
    private readonly IModelMetadataProvider _modelMetadataProvider;

    public CustomExceptionFilter(IModelMetadataProvider modelMetadataProvider)
    {
      _modelMetadataProvider = modelMetadataProvider;
    }

    public void OnException(ExceptionContext context)
    {
      var result = new ViewResult { ViewName = "CustomError" };
      result.ViewData = new ViewDataDictionary(_modelMetadataProvider, context.ModelState);
      result.ViewData.Add("Exception", context.Exception);

      // Here we can pass additional detailed data via ViewData
      context.ExceptionHandled = true; // mark exception as handled
      context.Result = result;
    }
  }
}
```

Step 2

Create a CustomError View: **Views/Shared/CustomError.cshtml**

```
@{
    ViewData["Title"] = "Custom Error";

    var exception = ViewData["Exception"] as Exception;
}

<h1 class="text-danger">An Error has Occurred.</h1>
<h2>Please try after some time.</h2>

<p>@exception.Message</p>
```

Step 3

Register in either locally in Controller level or Action level, e.g.

```
[TypeFilter(typeof(CustomExceptionFilter))]
public IActionResult Index(int? id = null)
{
    if (id.HasValue)
    {
        if (id == 1)
        {
            throw new FileNotFoundException
                ("File not found exception thrown in Home/Index");
        }
        else if (id == 2)
        {

            return StatusCode(500);
        }
    }
    return View();
}
```

Or global level in Startup.ConfigureServices,

```
public void ConfigureServices(IServiceCollection services)
{
  services.AddControllersWithViews();

  services.AddControllersWithViews(config =>
                                       config.Filters.Add(typeof(CustomExceptionFilter)));
}
```

Run the app, and Test it: **Click Trigger an exception**

Note: You must either register the Exception filter locally in action or controller or globally.