Exercise 11 : Error Handling

Objective

In this exercise you will create an error handling strategy for the QAForum application.

This exercise will take around 30 minutes.

Referenced material

This exercise is based on material from the chapter "Error Handling".

Saving Exception Data to the Database

	Open the 'Begin' solution in Visual Studio and compile (Shift Ctrl+B). The 'Begin' solution is identical to the 'End' solution from the previous lab.

	The error handling requirements for the QAForum application are that, in addition to errors being logged to the standard logging locations, any unhandled exceptions in the application must also be stored in the database. This will enable the support team to run a richer set of queries on the exceptions that occur than they would with the standard logging that is built into ASP.Net.
	
	Let's start by creating a model class that represents an unhandled exception. Add this class to the Models folder:
	`public class UnhandledException`
	`{`
	`    public int UnhandledExceptionId { get; set; }`
	`    public DateTime ExceptionDateTime { get; set; }`
	`    public string Path { get; set; }`
	`    public string ExceptionMessage { get; set; }`
	`    public string BaseExceptionMessage { get; set; }`
	`    public string StackTrace { get; set; }`
	
	`}`

	Now add the new class to the ForumDbContext class, which is located in EF/ForumDbContext.cs:
	`public class ForumDbContext : DbContext`
	`{`
	`    ....`
	`    public DbSet<Forum> Forums { get; set; }`
	`    public DbSet<Thread> Threads { get; set; }`
	`    public DbSet<Post> Posts { get; set; }`
	`    public DbSet<UnhandledException> UnhandledExceptions { get; set; }`

	The previous time we changed our entity framework model, we used the SeedDatabase program to apply our changes, although we noted that this technique caused our database to be deleted, together with any data in it.
	
	This time, we will use a different technique, just so that you can see a range of different ways of doing the same thing.
	Open the Package Manager Console window. Into that window type these two commands:
	`add-migration AddUnhandledExceptions -context ForumDbContext`
	`update-database -context ForumDbContext`
	
	(Note: if you get any errors when running the update-database command, run SeedDatabase as we have done in previous labs instead. This will result in your data being deleted, and would not be suitable for real life, but it will get you going with the rest of the lab. In real life, we would need to investigate the cause of the error, which might be an issue with entity framework versioning, or a mismatch between the model in the program and the database.)

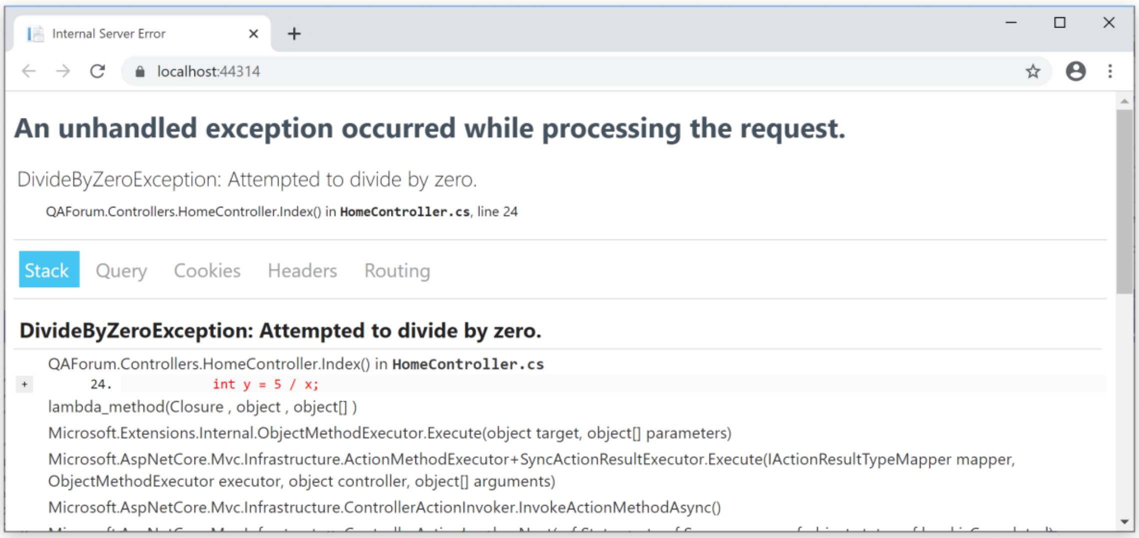
Let’s introduce a silly error into our program, just for the purpose of testing error handling. In /Controllers/HomeController.cs, modify the Index action as follows:

```
        public IActionResult Index()
        {
            int x = 0;
            int y = 5 / x;
            return View();
        }
```

That will cause a divide by zero error.

Run the program (you may prefer to use Ctrl-F5, rather than F5, so that the program doesn’t enter debug mode when the exception occurs. If you use F5, then just press F5 again or click Continue when the exception happens and Visual Studio enters debug mode).

What you will see is the Developer Exception Page:



An unhandled exception occurred while processing the request.

DivideByZeroException: Attempted to divide by zero.

QAForum.Controllers.HomeController.Index() in HomeController.cs, line 24

DivideByZeroException: Attempted to divide by zero.

```
QAForum.Controllers.HomeController.Index() in HomeController.cs
    24.             int y = 5 / x;
lambda_method(Closure , object , object[] )
Microsoft.Extensions.Internal.ObjectMethodExecutor.Execute(object target, object[] parameters)
Microsoft.AspNetCore.Mvc.Infrastructure.ActionMethodExecutor+SyncActionResultExecutor.Execute(IActionResultTypeMapper mapper, ObjectMethodExecutor executor, object controller, object[] arguments)
Microsoft.AspNetCore.Mvc.Infrastructure.ControllerActionInvoker.InvokeActionMethodAsync()
```

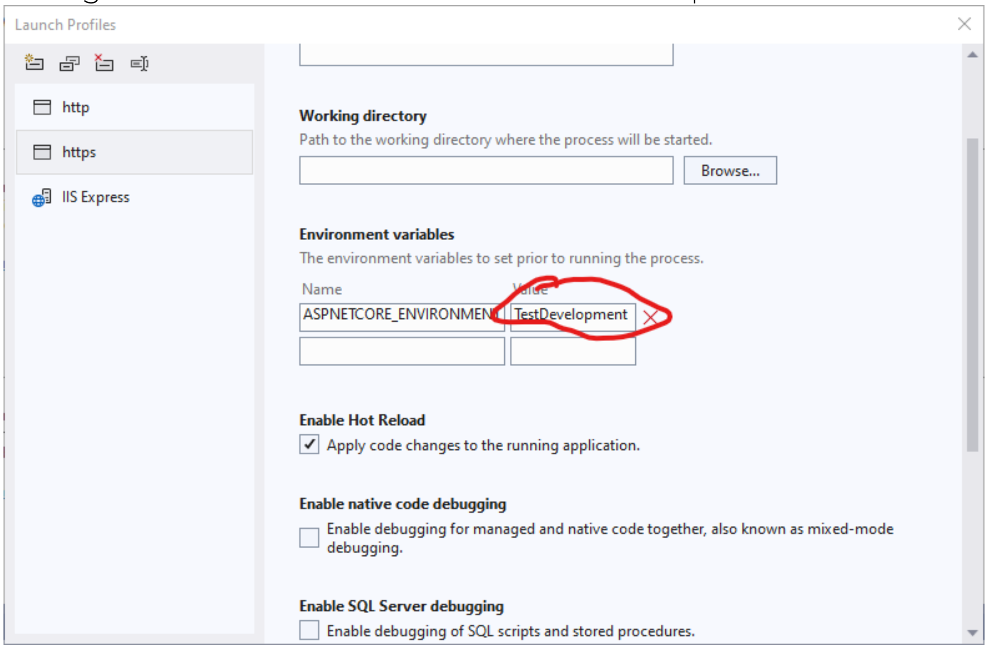
Click on each of the tabs (Stack, Query, Cookies, Headers and Routing) to see what each tab shows.

There's a lot of information there, and most of it is the kind of information which we would not wish to show to a user. Let's change out of development mode, and see what an end user would see by default.

In Solution Explorer, right-click on the QAForum project, and select Properties.

Go to the Debug tab, and look at the Environment Variables section. You should see an environment variable called ASPNETCORE_ENVIRONMENT, which is set to the value Development. Note whether you are setting the http, https or iis Express version.

Change the value of that variable to TestDevelopment:



Working directory
Path to the working directory where the process will be started.

Environment variables
The environment variables to set prior to running the process.

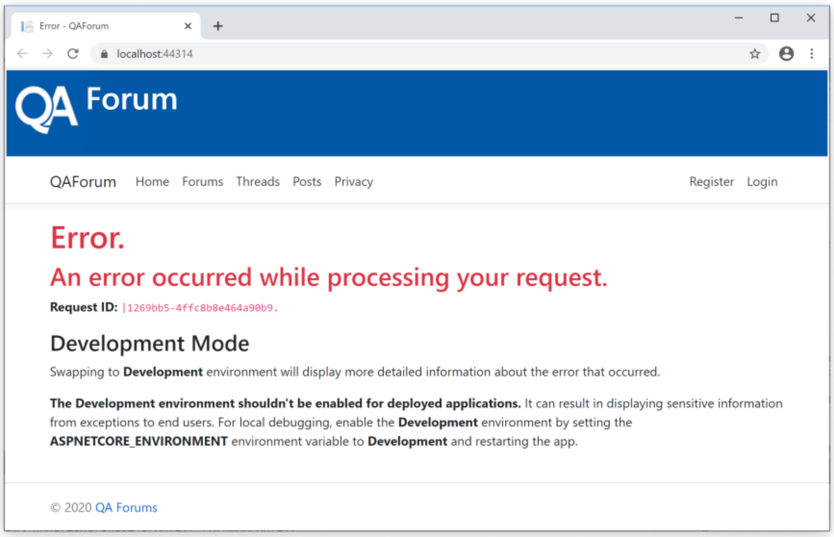
Name	Value
ASPNETCORE_ENVIRONMENT	TestDevelopment

Enable Hot Reload
✓ Apply code changes to the running application.

Enable native code debugging
Enable debugging for managed and native code together, also known as mixed-mode debugging.

Enable SQL Server debugging
Enable debugging of SQL scripts and stored procedures.

Run the program again, and look at the error screen now. That's better!



Error.

An error occurred while processing your request.

Request ID: |1269bb5-4ffc8b8e464a90b9.

Development Mode

Swapping to **Development** environment will display more detailed information about the error that occurred.

The Development environment shouldn't be enabled for deployed applications. It can result in displaying sensitive information from exceptions to end users. For local debugging, enable the **Development** environment by setting the **ASPNETCORE_ENVIRONMENT** environment variable to **Development** and restarting the app.

© 2020 QA Forums

	Stop your program, and open /Controllers/HomeController.cs. Have a look at the Error() method. This is the action which handles errors. Have a look in Program.cs. Find the line which says: `app.UseExceptionHandler("/Home/Error");` This is the line which inserts the middleware that redirects the user to the Error action. Note how that line is inside an "if/else" statement, so that it only runs if the program is not in development mode.

	Back in the home controller, right-click on the Error action, and choose Go To View.
	Find the `<h3>Development Mode</h3>` tag. Delete this tag, and everything below it. When we're not in development mode, there is no need to tell our users about the existence of development mode!
	
	Instead, add the following code to the bottom of the file:
	`@if (Model.ShowRequestId)`
	`{`
	`    <p>`
	`        <strong>Request ID:</strong> <code>@Model.RequestId</code>`
	`    </p>`
	`}`
	
	`<p class="text-danger">@ViewBag.Message</p>`

Now, let's modify the Error action in the Home controller.

The Error action will need access to our DbContext – but it's the only action in the Home controller which needs this. So it makes sense to inject it using parameter injection, rather than the more normal constructor injection, so that it's only injected on the rare occasions when it's needed.

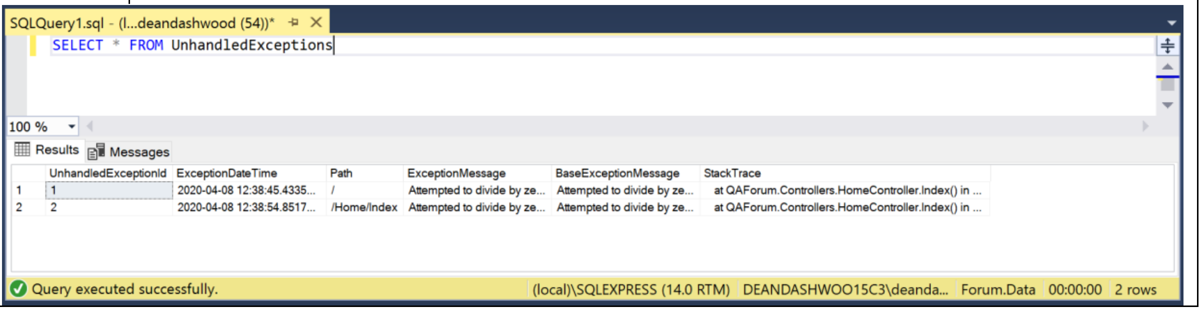
Modify the Error action as shown here:

```
        public IActionResult Error([FromServices]ForumDbContext context)
        {
            var exceptionHandlerPathFeature =
                HttpContext.Features.Get<IExceptionHandlerPathFeature>();
            var exception = exceptionHandlerPathFeature?.Error;

            var unhandledException = new UnhandledException
            {
                ExceptionDateTime = DateTime.Now,
                Path = exceptionHandlerPathFeature.Path,
                ExceptionMessage = exception?.Message,
                BaseExceptionMessage = exception?.GetBaseException()?.Message,
                StackTrace = exception?.StackTrace
            };

            try
            {
                context.UnhandledExceptions.Add(unhandledException);
                context.SaveChanges();
            }
            catch
            {
                ViewBag.Message = "This error was NOT logged, due to a further
error occurring while saving the error";
            }

            return View(new ErrorViewModel { RequestId = Activity.Current?.Id ??
HttpContext.TraceIdentifier });
        }
```

Note that it's vital that your error handler does not throw any errors itself! We have been careful to catch any exceptions that occur while saving to the database.

Run the program again.

After seeing the error message, use SQL Server Management Studio to query the contents of the UnhandledException table. You should be able to see details of the exception in the database:



```
SELECT * FROM UnhandledExceptions
```

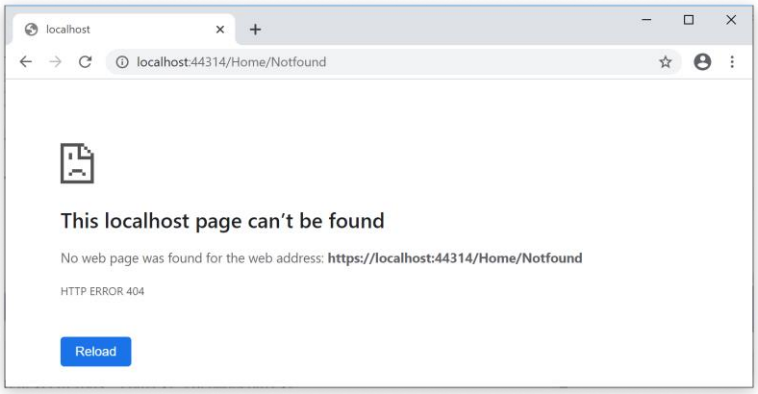
	UnhandledExceptionId	ExceptionDateTime	Path	ExceptionMessage	BaseExceptionMessage	StackTrace
1	1	2020-04-08 12:38:45.4335...	/	Attempted to divide by ze...	Attempted to divide by ze...	at QAForum.Controllers.HomeController.Index() in ...
2	2	2020-04-08 12:38:54.8517...	/Home/Index	Attempted to divide by ze...	Attempted to divide by ze...	at QAForum.Controllers.HomeController.Index() in ...

Query executed successfully. (local)\SQLEXPRESS (14.0 RTM) DEANDASHWOO15C3\deanda... Forum.Data 00:00:00 2 rows

Adding Status Code Pages

1 With the application running, modify the URL in your web browser to:
http://localhost:xxxx/Home/NotFound

2 Since there is no such action as NotFound, this will result in a response being sent to the web browser with a 404 status code (the code for Not Found). But the way the error is presented in the web browser is far from pleasing:



This localhost page can't be found

No web page was found for the web address: **https://localhost:44314/Home/Notfound**

HTTP ERROR 404

When MVC returns an error code to the web browser, it does not attach any content to that error code by default, which is why the web browser is forced to display a basic message such as that shown here.

To fix this, add the following middleware to the pipeline, beneath the declaration of the app object in Program.cs

```
    var app = builder.Build();
    app.UseStatusCodePagesWithReExecute("/Home/Status", "?code={0}");
    // Configure the HTTP request pipeline.
    if (app.Environment.IsDevelopment())
    {
                ....
```

Adding this line to the very start of the Configure method means that every other piece of middleware will get a chance to deal with the problem first, and only if they can't will this middleware component re-execute the request, but with a different path to allow the Status action to run instead of whatever action the user requested.

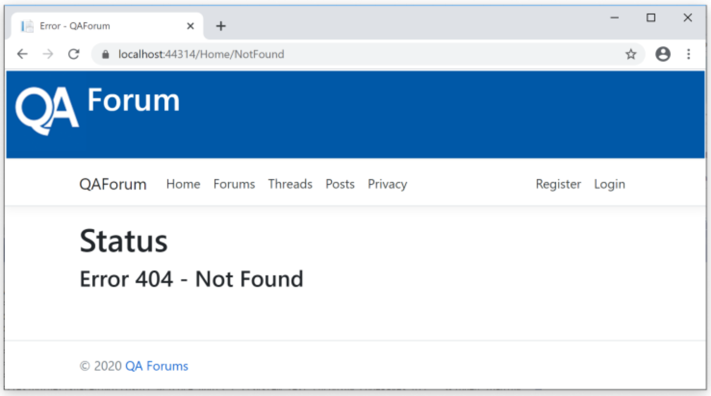
Next, add the Status action to the Home controller:

```
        public IActionResult Status(int code)
        {
            string message = Enum.GetName(typeof(HttpStatusCode), code);
            if (message == null)
            {
                ViewBag.Message = $"Error {code}";
            }
            else
            {
                // Turn the message from PascalCase to normal spacing
                message = Regex.Replace(message, ".[A-Z]",
                                m => $"{m.Value[0]} {m.Value[1]}");
                ViewBag.Message = $"Error {code} - {message}";
            }

            return View();
        }
```

Right-click in the Status action, and choose Add View. Select the Empty template, and add a basic view to show the error message to the user:

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

	```
	}

	<h1>Status</h1>

	<h3>@ViewBag.Message</h3>
	```
	 Note that, as per any other view, this view will make use of the Layout page, and therefore it will look like an integral part of our website, with the same styling as the rest of the site.

	Run the program again. Once the web browser loads (it will still show the error message from the home page), change the URL to once again try to access an Action that doesn't exist. This time you should see a nicely styled error page:  Error - QAForum localhost:44314/Home/NotFound QA Forum QAForum Home Forums Threads Posts Privacy Register Login Status Error 404 - Not Found © 2020 QA Forums

If You Have Time: Per-Action Error Handling

The techniques we have looked at in this chapter apply error handling across the whole of the web site.

Sometimes, it's preferable to customise the error handling for a specific controller, or even a specific action.

Add a folder to the root of the QAForum project, called Filters. Into that folder, add a class called CustomErrorAttribute:

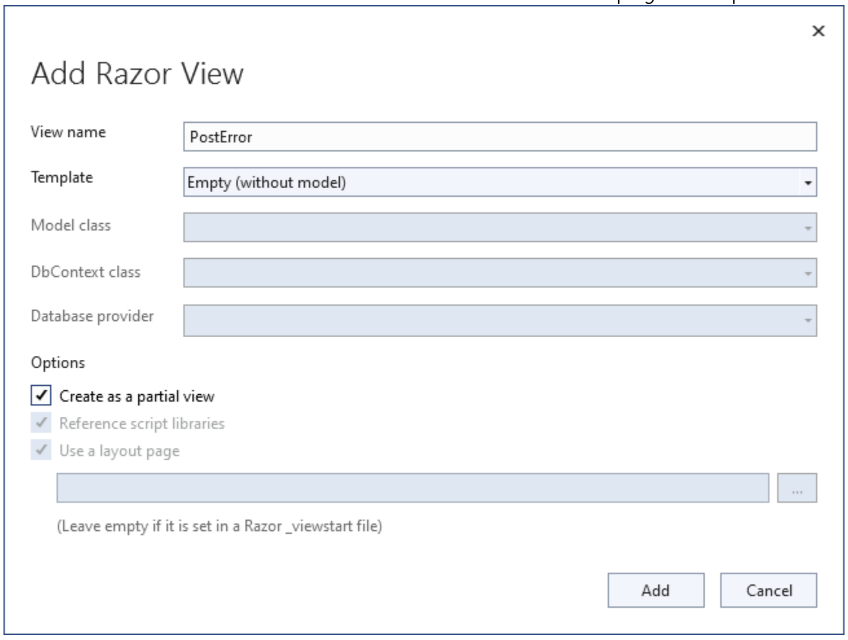
```
    public class CustomErrorAttribute : Attribute, IExceptionFilter
    {
        private readonly string viewName;

        public CustomErrorAttribute(string viewName)
        {
            this.viewName = viewName;
        }

        public void OnException(ExceptionContext context)
        {
            context.Result = new ViewResult
            {
                ViewName = viewName
            };
        }
    }
```

Let's test our custom error handling. We'll pick an action to test it on – it really doesn't matter which action we choose, but we'll choose the Post/Index action (in Areas/Users/Controllers/PostController). We'll add our new attribute to the action, and we'll also add code into the action to generate an error:

```
        [CustomError("PostError")]
        public IActionResult Index()
        {
            int x = 0;
            int y = 5 / x;
            return View();
        }
```

Notice how, in the first line (where we apply the attribute), we have specified the name of the view we want to use. Also, we could have applied the attribute to a controller class (instead of an action inside the controller class), and it would have applied to all actions in that controller.

Right-click in the Post/Index action, and select Add View. Ensure that you change the view name to PostError. Leave the Empty template selected:



Add Razor View

View name: PostError

Template: Empty (without model)

Model class:

DbContext class:

Database provider:

Options

- [✓] Create as a partial view
- [✓] Reference script libraries
- [✓] Use a layout page

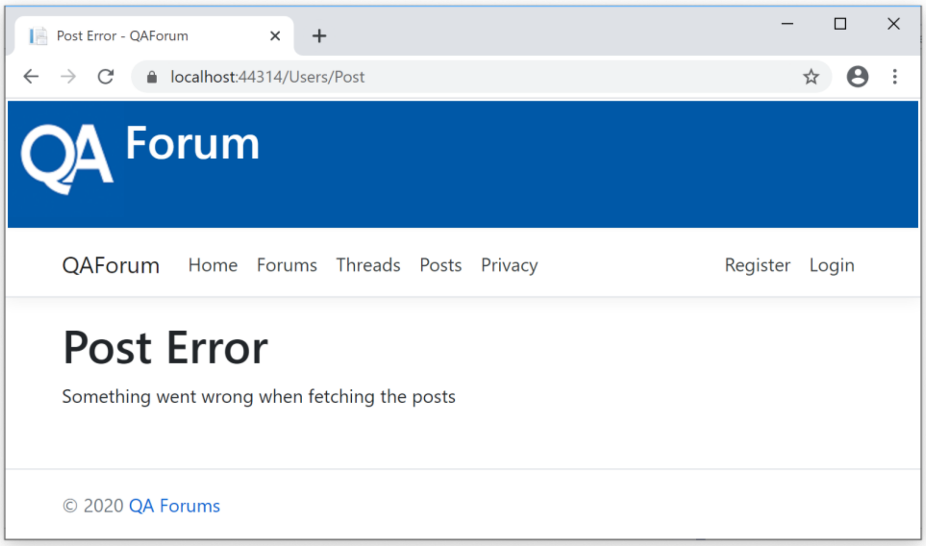
(Leave empty if it is set in a Razor _viewstart file)

Add some text to the view:

```
@{
    ViewData["Title"] = "Post Error";
}

<h1>Post Error</h1>

<p>Something went wrong when fetching the posts</p>
```

Run your program. You should see the standard error message, because of the error that's still being generated in Home/Index.

At the top of the error screen is the standard navigation bar. All the links should work – except for the Posts link, which should now show our custom error that's specific to the Posts action:



Post Error - QAForum

localhost:44314/Users/Post

Post Error

Something went wrong when fetching the posts

© 2020 QA Forums

Let’s make one more change. We’ll make it so that our custom error is only applied when not in development.

First of all, change the environment back to development. (Right-click on the project, and on the Debug tab, change ASPNETCORE_ENVIRONMENT to Development).

Then, run your program again and check that the custom error message is still shown. (You may have to enter the URL for the Posts page manually, since there is no navigation bar when the developer error message is shown for the home page. The URL is http://localhost:xxxx/Users/Post - replace xxxx with the correct port number that’s shown in your URL bar when you run the program.)

In order to find out whether the current environment is a development environment, the CustomErrorAttribute is going to need to use dependency injection to get hold of an IWebHostEnvironment object. It will do that using dependency injection:

```
    public class CustomErrorAttribute : Attribute, IExceptionFilter
    {
        private readonly string viewName;
        private readonly IWebHostEnvironment hostingEnvironment;

        public CustomErrorAttribute(string viewName,
                                    IWebHostEnvironment hostingEnvironment)
        {
            this.viewName = viewName;
            this.hostingEnvironment = hostingEnvironment;
        }

        public void OnException(ExceptionContext context)
        {
            if (hostingEnvironment.IsDevelopment())
            {
                return;
            }

            context.Result = new ViewResult
            {
                ViewName = viewName
            };
        }
    }
```

	Switch to the PostController class, and notice that there's now an error where you applied the CustomError attribute, because you haven't supplied enough arguments for its constructor.
	The C# language doesn't facilitate using dependency injection to create attributes. But MVC provides us with an attribute type which can be used to create other attributes, and which will use dependency injection to create the constructor parameters for those attributes.
	
	So, modify the attribute as follows:
	`[TypeFilter(typeof(CustomErrorAttribute),`
	`    Arguments = new object[] { "PostError"})]`
	`public IActionResult Index()`
	`{`
	`    ....`
	Note that, when using [TypeFilter], you don't need to register your filter with the service provider. [TypeFilter] does not use the service provider to create your filter, it only uses it to generate the constructor arguments for your filter.

	Test your program. You should now find that, because you are in a development environment, the developer exception page is shown when you go to the Posts/Index page. If you stop your program and change your environment, then re-test it, the custom error page will be shown.
	Change the ASPNETCORE_ENVIRONMENT back to Development if necessary. Remove the two lines of code that generate the error from the Home Controller's Index method (i.e. the lines that declare and initialise the variables x and y). Remove the two lines of code that generate the error from the Post Controller's Index method.

Congratulations, you have now created an exception handling strategy for your application!