Build an ASP.NET Core Service, and App with Core 2.2 Two-Day Hand-On Lab

Lab 8

This lab is the third in a series that builds the SpyStore RESTful service. This lab creates and configures the Exception filter. Prior to starting this lab, you must have completed Lab 7.

Part 1: Create and Apply the Exception Filter

Exception filters come into play when an unhandled exception is thrown in an action method (or bubbles up to an action method).

Step 1: Create the Exception Filter

1) Add a new folder named Filters into the `SpyStore.Service.cs` class.
2) Add a new class named `SpyStoreExceptionFilter.cs` in the Filters directory. Make the class public and inherit `ExceptionFilterAttribute`, as shown here:

```
public class SpyStoreExceptionFilter : ExceptionFilterAttribute
{
}
```

3) Add a constructor that takes an instance of `IHostingEnvironment` and assigns it to a private class variable:

```
private readonly IHostingEnvironment _hostingEnvironment;
public SpyStoreExceptionFilter(IHostingEnvironment hostingEnvironment)
{
  _hostingEnvironment = hostingEnvironment;
}
```

4) The `ExceptionFilter` has only one method to be implemented, `OnException`. Override this from the base class:

```
public override void OnException(ExceptionContext context)
{
}
```

5) The `ExceptionContext` provides information about the `ActionContext`, `Exception` thrown, the `HttpContext`, `ModelState`, and `RouteData`. For this application, you will use the `Exception` information to build up a customer `Response` message. If the environment is development, the stack trace is included. Update the `OnException` code to the following:

```
public override void OnException(ExceptionContext context)
{
  var ex = context.Exception;
  string stackTrace = _hostingEnvironment.IsDevelopment()
    ? context.Exception.StackTrace
    : string.Empty;
  string message = ex.Message;
  string error;
  IActionResult actionResult;
  switch (ex)
  {
    case SpyStoreInvalidQuantityException iqe:
      //Returns a 400
      error = "Invalid quantity request.";
      actionResult = new BadRequestObjectResult(
        new { Error = error, Message = message, StackTrace = stackTrace });
      break;
    case DbUpdateConcurrencyException ce:
      //Returns a 400
      error = "Concurrency Issue.";
      actionResult = new BadRequestObjectResult(
        new { Error = error, Message = message, StackTrace = stackTrace });
      break;
    case SpyStoreInvalidProductException ipe:
      //Returns a 400
      error = "Invalid Product Id.";
      actionResult = new BadRequestObjectResult(
        new { Error = error, Message = message, StackTrace = stackTrace });
      break;
    case SpyStoreInvalidCustomerException ice:
      //Returns a 400
      error = "Invalid Customer Id.";
      actionResult = new BadRequestObjectResult(
        new { Error = error, Message = message, StackTrace = stackTrace });
      break;
    default:
      error = "General Error.";
      actionResult = new ObjectResult(
        new { Error = error, Message = message, StackTrace = stackTrace })
      {
        StatusCode = 500
      };
      break;
  }
  //context.ExceptionHandled = true; //If this is uncommented, the exception is swallowed
  context.Result = actionResult;
}
```

Step 2: Apply the Exception Filter

1) Open the Startup.cs class and add the following using statement:

```
using SpyStore.Hol.Service.Filters;
```

2) Navigate to the `ConfigureServices` method. Update the lines that configure the JsonFormatter to include globally applying the `ExceptionFilter`:

```
services.AddMvcCore(config =>config.Filters.Add(new SpyStoreExceptionFilter(_env)))
        .AddJsonFormatters(j =>
        {
          j.ContractResolver = new DefaultContractResolver();
          j.Formatting = Formatting.Indented;
        });
```

Step 3: Test the Exception Filter

1) Open the ValuesController and navigate to the Get method that takes an integer parameter. Add an exception to the action method, like this:

```
// GET api/values/5
[HttpGet("{id}")]
public ActionResult<string> Get(int id)
{
  throw new Exception("Test Exception");
  return "value";
}
```

2) Run the application and use the SwaggerUI to test the Get method. You will get a result as follows:

```
{
  "Error": "General Error.",
  "Message": "Test Exception",
  "StackTrace": "   at SpyStore.Hol.Service.Controllers.ValuesController.Get(Int32 id) in
C:\\GitHub\\dotnetcore_hol\\TwoDay\\2.2\\Code\\Completed\\Lab8\\SpyStore.Hol.Service\\Controllers\
\ValuesController.cs:line 24\r\n   at lambda_method(Closure , Object , Object[] )\r\n   at
Microsoft.Extensions.Internal.ObjectMethodExecutor.Execute(Object target, Object[] parameters)\r\n
at
Microsoft.AspNetCore.Mvc.Internal.ActionMethodExecutor.SyncObjectResultExecutor.Execute(IActionRes
ultTypeMapper mapper, ObjectMethodExecutor executor, Object controller, Object[] arguments)\r\n
at Microsoft.AspNetCore.Mvc.Internal.ControllerActionInvoker.InvokeActionMethodAsync()\r\n   at
Microsoft.AspNetCore.Mvc.Internal.ControllerActionInvoker.InvokeNextActionFilterAsync()\r\n   at
Microsoft.AspNetCore.Mvc.Internal.ControllerActionInvoker.Rethrow(ActionExecutedContext
context)\r\n   at Microsoft.AspNetCore.Mvc.Internal.ControllerActionInvoker.Next(State& next,
Scope& scope, Object& state, Boolean& isCompleted)\r\n   at
Microsoft.AspNetCore.Mvc.Internal.ControllerActionInvoker.InvokeInnerFilterAsync()\r\n   at
Microsoft.AspNetCore.Mvc.Internal.ResourceInvoker.InvokeNextExceptionFilterAsync()"
}
```

Summary

This lab created and configured an ExceptionFilter for the service.

Next steps

In the next part of this tutorial series, you will add the controllers for the service.