Handling Errors in Web API Using Exception Filters and Exception Handlers

Exception Filters

Exception filters are types of filters used in Web API to catch unhandled exceptions that occur from within a controller's action. If you're familiar with MVC then this should not be new for you, and you'll agree that when designing an API, it's always good to have another tool in your arsenal to meet the variety of situations that can arise. They're best used for specific controllers but can be used globally as well, although Exception handlers are likely more suited in that case (see below).

Exception filters can be created by implementing IExceptionFilter, or deriving from the abstract class ExceptionFilterAttribute (which actually implements IExceptionFilter anyway). Like other aspects of Web API, they can be registered globally or applied to actions within a controller as an attribute.

If you are deriving from ExceptionFilterAttribute, then there is one method that you can override to provide functionality for an unhandled exception, called OnException:

```
public override void OnException(HttpActionExecutedContext a
```

actionExecutedContext gives you access to the type of exception as well as the ability to override the response that comes back to the server. This means you could suppress a particular error message or even affect the response code to be something different. Maybe your error message reveals some sensitive trace information and you'd like to return something more user friendly, for example.

To demonstrate, let's create a sample application and get started.

1. Create a new ASP.NET project in Visual Studio, and select ASP.NET Web Application.
2. From the ASP.NET templates, choose 'Empty' and click the Web API checkbox in the bottom section underneath 'Add folders and core references for:'
3. You should now have your basic Web API project setup and ready to run.

From here, add a new controller to your 'Controllers' folder, and select 'Web API 2 Controller with read/write actions' from the list. I named my controller DemoController, thus giving me routes such as /api/Demo.

Now create a new folder called Filters, and add a new class named DivideByZeroExceptionFilter with the following code:

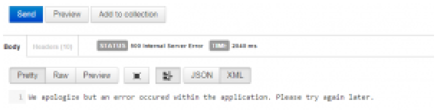
```
public class DivideByZeroExceptionFilter : ExceptionFilterA
  {
      public override void OnException(HttpActionExecutedCo
      {
          if (actionExecutedContext.Exception is DivideByZe
          {
              actionExecutedContext.Response = new HttpResp
                  { Content = new StringContent("We apologi
          }
      }

  }
```

Place a breakpoint within the class (within the If condition for checking the exception type, for example). Now let's add this attribute to our DemoController, throw an exception, and watch it hit our breakpoint. I decided to use the Delete action for my example.

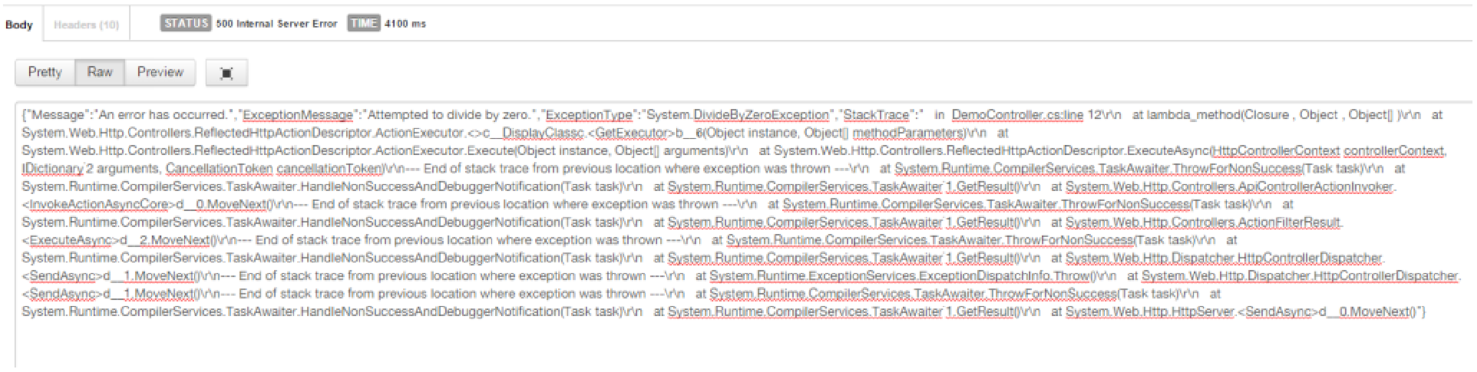
```
[DivideByZeroExceptionFilter]
       public void Delete(int id)
       {
            throw new DivideByZeroException();
       }
```

Using **Postman**, I issued a DELETE to my DemoController, which is decorated with the [DivideByZeroExceptionFilter]. Because of this, I throw the above DivideByZeroException, the breakpoint hits, and we can catch our DivideByZeroException within the custom filter.
You’ll notice that although the same 500 response is issued (certainly that could be changed based on your needs), a more friendly error message is returned:



```
We apologize but an error occured within the application. Please try again later.
```

Comment out the [DivideByZeroExceptionFilter] attribute above the Delete action, and observe what happens now:



```
{"Message":"An error has occurred.","ExceptionMessage":"Attempted to divide by zero.","ExceptionType":"System.DivideByZeroException","StackTrace":"   in DemoController.cs:line 12\r\n   at lambda_method(Closure , Object , Object[] )\r\n   at
System.Web.Http.Controllers.ReflectedHttpActionDescriptor.ActionExecutor.<>c__DisplayClassc.<GetExecutor>b__6(Object instance, Object[] methodParameters)\r\n   at
System.Web.Http.Controllers.ReflectedHttpActionDescriptor.ActionExecutor.Execute(Object instance, Object[] arguments)\r\n   at System.Web.Http.Controllers.ReflectedHttpActionDescriptor.ExecuteAsync(HttpControllerContext controllerContext,
IDictionary`2 arguments, CancellationToken cancellationToken)\r\n--- End of stack trace from previous location where exception was thrown ---\r\n   at System.Runtime.CompilerServices.TaskAwaiter.ThrowForNonSuccess(Task task)\r\n   at
System.Runtime.CompilerServices.TaskAwaiter.HandleNonSuccessAndDebuggerNotification(Task task)\r\n   at System.Runtime.CompilerServices.TaskAwaiter`1.GetResult()\r\n   at System.Web.Http.Controllers.ApiControllerActionInvoker.
<InvokeActionAsyncCore>d__0.MoveNext()\r\n--- End of stack trace from previous location where exception was thrown ---\r\n   at System.Runtime.CompilerServices.TaskAwaiter.ThrowForNonSuccess(Task task)\r\n   at
System.Runtime.CompilerServices.TaskAwaiter.HandleNonSuccessAndDebuggerNotification(Task task)\r\n   at System.Runtime.CompilerServices.TaskAwaiter`1.GetResult()\r\n   at System.Web.Http.Controllers.ActionFilterResult.
<ExecuteAsync>d__2.MoveNext()\r\n--- End of stack trace from previous location where exception was thrown ---\r\n   at System.Runtime.CompilerServices.TaskAwaiter.ThrowForNonSuccess(Task task)\r\n   at
System.Runtime.CompilerServices.TaskAwaiter.HandleNonSuccessAndDebuggerNotification(Task task)\r\n   at System.Runtime.CompilerServices.TaskAwaiter`1.GetResult()\r\n   at System.Web.Http.Dispatcher.HttpControllerDispatcher.
<SendAsync>d__1.MoveNext()\r\n--- End of stack trace from previous location where exception was thrown ---\r\n   at System.Runtime.ExceptionServices.ExceptionDispatchInfo.Throw()\r\n   at System.Web.Http.Dispatcher.HttpControllerDispatcher.
<SendAsync>d__1.MoveNext()\r\n--- End of stack trace from previous location where exception was thrown ---\r\n   at System.Runtime.CompilerServices.TaskAwaiter.ThrowForNonSuccess(Task task)\r\n   at
System.Runtime.CompilerServices.TaskAwaiter.HandleNonSuccessAndDebuggerNotification(Task task)\r\n   at System.Runtime.CompilerServices.TaskAwaiter`1.GetResult()\r\n   at System.Web.Http.HttpServer.<SendAsync>d__0.MoveNext()"}
```

The entire stack trace is returned! By using the filter, we have control over the HTTP response code returned, as well as any sort of customized error message we want.

While many times exception filters are decorated on specific actions, you can also register an exception filter globally. This is done by going to your WebApiConfiguration.cs, and adding the following line of code within your Register method:

```
config.Filters.Add(new DivideByZeroExceptionFilter());
```

Now when you run the API and invoke the same call, it will still catch the error. In fact, any time a DivideByZero exception occurs from within any controller, it will get picked up by the exception filter. Obviously this is not just within your controller, but any method further down the hierarchy originating from your controller.

While this is great, there are a few key points about Exception Filters:

- HttpResponseExceptions do not cause exception filters to fire – they are a special case designed for returning an HTTP response.
- Exceptions thrown from the constructor of a controller will not be caught within your filter.
- Similarly, exceptions thrown from other filters, or from delegating handlers will not be caught.
- Exceptions in routing or during content serialization will not be caught.

Because of this, Exception Filters are not an ideal choice for serving as a truly global mechanism to catch and take specific action with all exceptions that occur within your API.
For these scenarios, you would rely on an Exception Handler.

Exception Handlers

Now that you've seen how Exception Filters work, let's get into Web API 2′s Exception Handlers. If an unhandled error occurs from anywhere within the application, your Exception Handler will catch it and allow you to take specific action. Like exception filters, they won't fire if you have a try..catch block (the operative word was 'unhandled' exception), nor will they fire if you are using an exception filter on a specific controller. This means if you have a global exception filter registered, but a specific controller's action has an exception filter, then the exception filter will fire and the exception handler will not. In the case where your exception filter in turn throws an error, then the exception handler *will* catch this and execute whatever code you have prepared.

Let's add an Exception Handler class and use this instead to demonstrate what takes place when our DivideByZero exception occurs. Add a new class called MyExceptionHandler with the following code:

```
//A global exception handler that will be used to catch any
public class MyExceptionHandler : ExceptionHandler
    {
        //A basic DTO to return back to the caller with dat
        private class ErrorInformation
        {
            public string Message { get; set; }
            public DateTime ErrorDate { get; set; }
        }

        public override void Handle(ExceptionHandlerContext
        {

            //Return a DTO representing what happened
            context.Result = new ResponseMessageResult(cont
              new ErrorInformation { Message="We apologize

            //This is commented out, but could also serve t
            //context.Result = new ResponseMessageResult(co
        }
    }
```

Now go to WebApiConfig.cs, and add the following line:

```
config.Services.Replace(typeof(IExceptionHandler), new MyExc
```

This will ensure that all exceptions occurring globally are processed by MyExceptionHandler. Place a breakpoint somewhere within the Handle method and run the app again, invoking the same request as last time to cause an exception. The DivideByZeroException will be raised, and once you press continue, it will move in to the handler. Without it, a full stack trace would be returned, and with it in place, your custom error message takes the place, along with any response code you want!

In conclusion, exception filters and handlers are two powerful mechanisms provided by Web API to catch unhandled exceptions and take an appropriate action.
I'd like to cover logging in another post, but as you could imagine there is a very relevant tie in here to log your errors appropriately.

Learn more **development** tips from the Karbyn team. We are always looking for bright and talented people, if want to join our team, check out our **open positions**.