Exception Handling in ASP.NET Web API

03/12/20124 minutes to read +2

In this article

by Mike Wasson

This article describes error and exception handling in ASP.NET Web API.

- HttpResponseException
- Exception Filters
- Registering Exception Filters
- HttpError

HttpResponseException

What happens if a Web API controller throws an uncaught exception? By default, most exceptions are translated into an HTTP response with status code 500, Internal Server Error.

The **HttpResponseException** type is a special case. This exception returns any HTTP status code that you specify in the exception constructor. For example, the following method returns 404, Not Found, if the *id* parameter is not valid.

C# Copy

```
public Product GetProduct(int id)
{
    Product item = repository.Get(id);
    if (item == null)
    {
        throw new HttpResponseException(HttpStatusCode.NotFound);
    }
    return item;
}
```

For more control over the response, you can also construct the entire response message and include it with the **HttpResponseException:**

C# Copy

```
public Product GetProduct(int id)
{
    Product item = repository.Get(id);
    if (item == null)
    {
        var resp = new HttpResponseMessage(HttpStatusCode.NotFound)
        {
            Content = new StringContent(string.Format("No product with ID = {0}", id)),
            ReasonPhrase = "Product ID Not Found"
        };
        throw new HttpResponseException(resp);
    }
    return item;
}
```

Exception Filters

You can customize how Web API handles exceptions by writing an *exception filter*. An exception filter is executed when a controller method throws any unhandled exception that is *not* an **HttpResponseException** exception. The **HttpResponseException** type is a special case, because it is designed specifically for returning an HTTP response.

Exception filters implement the **System.Web.Http.Filters.IExceptionFilter** interface. The simplest way to write an exception filter is to derive from the **System.Web.Http.Filters.ExceptionFilterAttribute** class and override the **OnException**method.

Note

Exception filters in ASP.NET Web API are similar to those in ASP.NET MVC. However, they are declared in a separate namespace and function separately. In particular, the **HandleErrorAttribute** class used in MVC does not handle exceptions thrown by Web API controllers.

Here is a filter that converts **NotImplementedException** exceptions into HTTP status code 501, Not Implemented:

C# Copy

```
namespace ProductStore.Filters
{
    using System;
    using System.Net;
    using System.Net.Http;
    using System.Web.Http.Filters;

    public class NotImplExceptionFilterAttribute : ExceptionFilterAttribute
    {
        public override void OnException(HttpActionExecutedContext context)
        {
            if (context.Exception is NotImplementedException)
            {
                context.Response = new
HttpResponseMessage(HttpStatusCode.NotImplemented);
            }
        }
    }
}
```

The **Response** property of the **HttpActionExecutedContext** object contains the HTTP response message that will be sent to the client.

Registering Exception Filters

There are several ways to register a Web API exception filter:

- By action
- By controller
- Globally

To apply the filter to a specific action, add the filter as an attribute to the action:

C# Copy

```
public class ProductsController : ApiController
{
    [NotImplExceptionFilter]
    public Contact GetContact(int id)
    {
        throw new NotImplementedException("This method is not implemented");
    }
}
```

To apply the filter to all of the actions on a controller, add the filter as an attribute to the controller class:

C# Copy

```
[NotImplExceptionFilter]
public class ProductsController : ApiController
{
    // ...
}
```

To apply the filter globally to all Web API controllers, add an instance of the filter to the **GlobalConfiguration.Configuration.Filters** collection. Exception filters in this collection apply to any Web API controller action.

C# Copy

```
GlobalConfiguration.Configuration.Filters.Add(
    new ProductStore.NotImplExceptionFilterAttribute());
```

If you use the "ASP.NET MVC 4 Web Application" project template to create your project, put your Web API configuration code inside the `WebApiConfig` class, which is located in the App_Start folder:

C# Copy

```
public static class WebApiConfig
{
    public static void Register(HttpConfiguration config)
    {
        config.Filters.Add(new ProductStore.NotImplExceptionFilterAttribute());


        // Other configuration code...
    }
}
```

HttpError

The **HttpError** object provides a consistent way to return error information in the response body. The following example shows how to return HTTP status code 404 (Not Found) with an **HttpError** in the response body.

C# Copy

```
public HttpResponseMessage GetProduct(int id)
{
    Product item = repository.Get(id);
    if (item == null)
    {
        var message = string.Format("Product with id = {0} not found", id);
        return Request.CreateErrorResponse(HttpStatusCode.NotFound, message);
    }
    else
    {
        return Request.CreateResponse(HttpStatusCode.OK, item);
    }
}
```

CreateErrorResponse is an extension method defined in the **System.Net.Http.HttpRequestMessageExtensions** class. Internally, **CreateErrorResponse**creates an **HttpError** instance and then creates an **HttpResponseMessage** that contains the **HttpError**.

In this example, if the method is successful, it returns the product in the HTTP response. But if the requested product is not found, the HTTP response contains an **HttpError** in the request body. The response might look like the following:

Console

Copy

```
HTTP/1.1 404 Not Found
Content-Type: application/json; charset=utf-8
Date: Thu, 09 Aug 2012 23:27:18 GMT
Content-Length: 51

{
  "Message": "Product with id = 12 not found"
}
```

Notice that the **HttpError** was serialized to JSON in this example. One advantage of using **HttpError** is that it goes through the same content-negotiation and serialization process as any other strongly-typed model.

HttpError and Model Validation

For model validation, you can pass the model state to **CreateErrorResponse**, to include the validation errors in the response:

C#

Copy

```
public HttpResponseMessage PostProduct(Product item)
{
    if (!ModelState.IsValid)
    {
        return Request.CreateErrorResponse(HttpStatusCode.BadRequest, ModelState);
    }

    // Implementation not shown...
}
```

This example might return the following response:

Console

Copy

```
HTTP/1.1 400 Bad Request
Content-Type: application/json; charset=utf-8
Content-Length: 320

{
  "Message": "The request is invalid.",
  "ModelState": {
    "item": [
      "Required property 'Name' not found in JSON. Path '', line 1, position 14."
    ],
    "item.Name": [
      "The Name field is required."
    ],
    "item.Price": [
      "The field Price must be between 0 and 999."
    ]
  }
}
```

For more information about model validation, see Model Validation in ASP.NET Web API.

Using HttpError with HttpResponseException

The previous examples return an **HttpResponseMessage** message from the controller action, but you can also use **HttpResponseException** to return an **HttpError**. This lets you return a strongly-typed model in the normal success case, while still returning **HttpError** if there is an error:

C# Copy

```csharp
public Product GetProduct(int id)
{
    Product item = repository.Get(id);
    if (item == null)
    {
        var message = string.Format("Product with id = {0} not found", id);
        throw new HttpResponseException(
            Request.CreateErrorResponse(HttpStatusCode.NotFound, message));
    }
    else
    {
        return item;
    }
}
```