CHAPTER 30

Using Filters

Filters inject extra logic into request processing. Filters are like middleware that is applied to a single endpoint, which can be an action or a page handler method, and they provide an elegant way to manage a specific set of requests. In this chapter, I explain how filters work, describe the different types of filter that ASP.NET Core supports, and demonstrate the use of custom filters and the filters provided by ASP.NET Core. Table 30-1 summarizes the chapter.

Table 30-1. *Chapter Summary*

Problem	Solution	Listing
Implementing a security policy	Use an authorization filter	15, 16
Implementing a resource policy, such as caching	Use a resource filter	17–19
Altering the request or response for an action method	Use an action filter	20–23
Altering the request or response for a page handler method	Use a page filter	24–26
Inspecting or altering the result produced by an endpoint	Use a result filter	27–29
Inspecting or altering uncaught exceptions	Use an exception filter	30–31
Altering the filter lifecycle	Use a filter factory or define a service	32–35
Applying filters throughout an application	Use a global filter	36, 37
Changing the order in which filters are applied	Implement the `IOrderedFilter` interface	38–42

Preparing for This Chapter

This chapter uses the WebApp project from Chapter 29. To prepare for this chapter, open a new PowerShell command prompt, navigate to the `WebApp` project folder, and run the command shown in Listing 30-1 to remove the files that are no longer required.

Listing 30-1. Removing Files from the Project

```
Remove-Item -Path Controllers,Views,Pages -Recurse -Exclude _*,Shared
```

This command removes the controllers, views, and Razor Pages, leaving behind the shared layouts, data model, and configuration files.

Tip You can download the example project for this chapter—and for all the other chapters in this book—from `https://github.com/apress/pro-asp.net-core-3`. See Chapter 1 for how to get help if you have problems running the examples.

A. Freeman, *Pro ASP.NET Core 3*, https://doi.org/10.1007/978-1-4842-5440-0_30

Create the WebApp/Controllers folder and add a class file named HomeController.cs to the Controllers folder with the code shown in Listing 30-2.

Listing 30-2. The Contents of the HomeController.cs File in the Controllers Folder

```
using Microsoft.AspNetCore.Mvc;

namespace WebApp.Controllers {

    public class HomeController : Controller {

        public IActionResult Index() {
            return View("Message",
                "This is the Index action on the Home controller");
        }
    }
}
```

The action method renders a view called Message and passes a string as the view data. I added a Razor view named Message.cshtml with the content shown in Listing 30-3.

Listing 30-3. The Contents of the Message.cshtml File in the Views/Shared Folder

```
@{ Layout = "_SimpleLayout"; }

@if (Model is string) {
    @Model
} else if (Model is IDictionary<string, string>) {
    var dict = Model as IDictionary<string, string>;
    <table class="table table-sm table-striped table-bordered">
        <thead><tr><th>Name</th><th>Value</th></tr></thead>
        <tbody>
            @foreach (var kvp in dict) {
                <tr><td>@kvp.Key</td><td>@kvp.Value</td></tr>
            }
        </tbody>
    </table>
}
```

Add a Razor Page named Message.cshtml to the Pages folder and add the content shown in Listing 30-4.

Listing 30-4. The Contents of the Message.cshtml File in the Pages Folder

```
@page "/pages/message"
@model MessageModel
@using Microsoft.AspNetCore.Mvc.RazorPages
@using System.Collections.Generic

@if (Model.Message is string) {
    @Model.Message
} else if (Model.Message is IDictionary<string, string>) {
    var dict = Model.Message as IDictionary<string, string>;
    <table class="table table-sm table-striped table-bordered">
        <thead><tr><th>Name</th><th>Value</th></tr></thead>
        <tbody>
            @foreach (var kvp in dict) {
                <tr><td>@kvp.Key</td><td>@kvp.Value</td></tr>
            }
        </tbody>
```

```
    </table>
}

@functions {
    public class MessageModel : PageModel {

        public object Message { get; set; } = "This is the Message Razor Page";
    }
}
```

Enabling HTTPS Connections

Some of the examples in this chapter require the use of SSL. Add the configuration entries shown in Listing 30-5 to the launchSettings.json file in the Properties folder to enable SSL and set the port to 44350.

Listing 30-5. Enabling HTTPS in the launchSettings.json File in the Properties Folder

```
{
  "iisSettings": {
    "windowsAuthentication": false,
    "anonymousAuthentication": true,
    "iisExpress": {
      "applicationUrl": "http://localhost:5000",
      "sslPort": 44350
    }
  },
  "profiles": {
    "IIS Express": {
      "commandName": "IISExpress",
      "launchBrowser": true,
      "environmentVariables": {
        "ASPNETCORE_ENVIRONMENT": "Development"
      }
    },
    "WebApp": {
      "commandName": "Project",
      "launchBrowser": true,
      "environmentVariables": {
        "ASPNETCORE_ENVIRONMENT": "Development"
      },
      "applicationUrl": "http://localhost:5000;https://localhost:44350"
    }
  }
}
```

The .NET Core runtime includes a test certificate that is used for HTTPS requests. Run the commands shown in Listing 30-6 in the WebApp folder to regenerate and trust the test certificate.

Listing 30-6. Regenerating the Development Certificates

```
dotnet dev-certs https --clean

dotnet dev-certs https --trust
```

Click Yes to the prompts to delete the existing certificate that has already been trusted and click Yes to trust the new certificate, as shown in Figure 30-1.

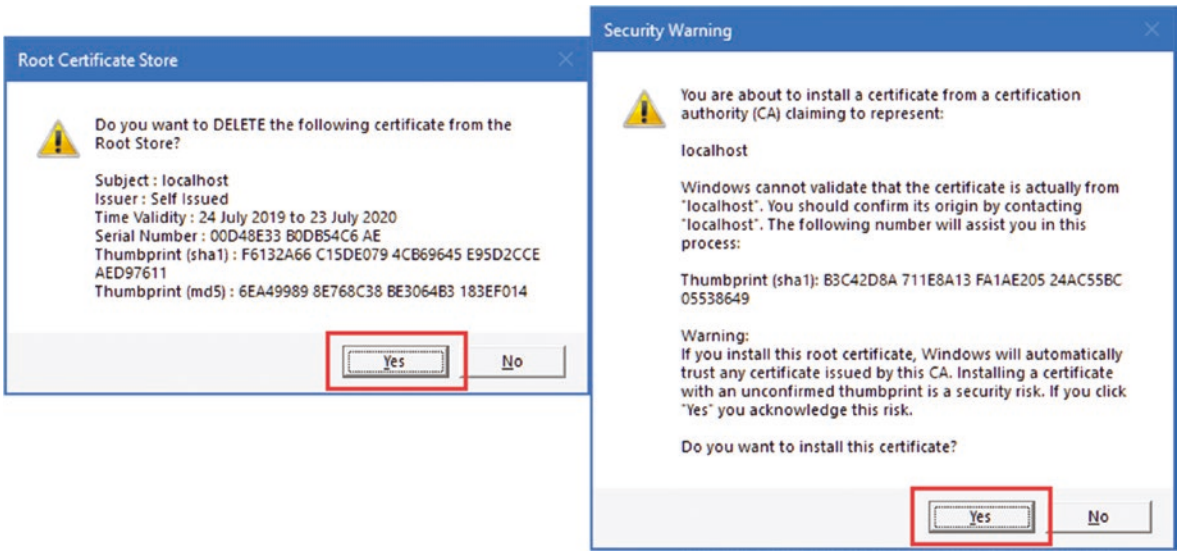

Figure 30-1. *Regenerating the HTTPS certificate*

Dropping the Database

Open a new PowerShell command prompt, navigate to the folder that contains the `WebApp.csproj` file, and run the command shown in Listing 30-7 to drop the database.

Listing 30-7. Dropping the Database

```
dotnet ef database drop --force
```

Running the Example Application

Select Start Without Debugging or Run Without Debugging from the Debug menu or use the PowerShell command prompt to run the command shown in Listing 30-8.

Listing 30-8. Running the Example Application

```
dotnet run
```

Use a browser to request `http://localhost:5000` and `https://localhost:44350`. Both URLs will be handled by the `Index` action defined by the `Home` controller, producing the responses shown in Figure 30-2.

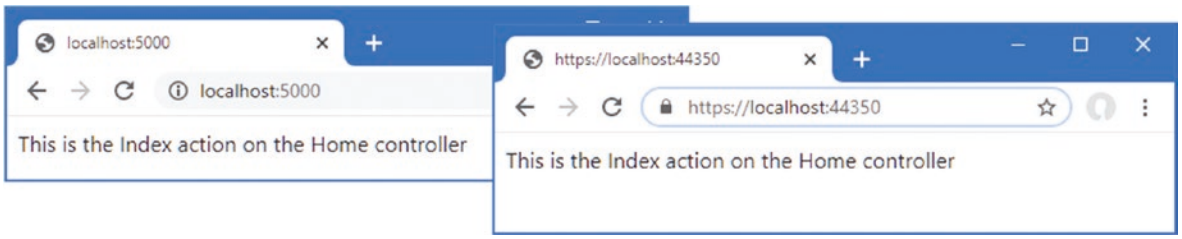

Figure 30-2. *Responses from the Home controller*

Request `http://localhost:5000/pages/message` and `https://localhost:44350/pages/message` to see the response from the `Message` Razor Page, delivered over HTTP and HTTPS, as shown in Figure 30-3.

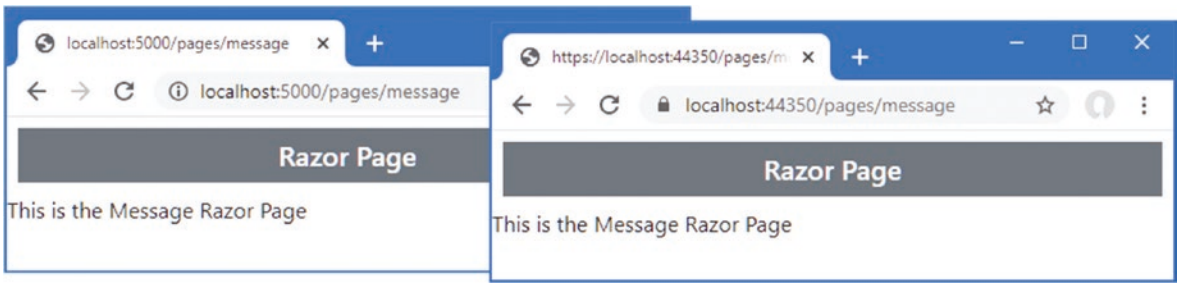

Figure 30-3. *Responses from the Message Razor Page*

Using Filters

Filters allow logic that would otherwise be applied in a middleware component or action method to be defined in a class where it can be easily reused.

Imagine that you want to enforce HTTPS requests for some action methods. In Chapter 16, I showed you how this can be done in middleware by reading the `IsHttps` property of the `HttpRequest` object. The problem with this approach is that the middleware would have to understand the configuration of the routing system to know how to intercept requests for specific action methods. A more focused approach would be to read the `HttpRequest.IsHttps` property within action methods, as shown in Listing 30-9.

Listing 30-9. Selectively Enforcing HTTPS in the HomeController.cs File in the Controllers Folder

```
using Microsoft.AspNetCore.Mvc;
using Microsoft.AspNetCore.Http;

namespace WebApp.Controllers {

    public class HomeController : Controller {

        public IActionResult Index() {
            if (Request.IsHttps) {
                return View("Message",
                    "This is the Index action on the Home controller");
            } else {
                return new StatusCodeResult(StatusCodes.Status403Forbidden);
            }
        }
    }
}
```

Restart ASP.NET Core and request `http://localhost:5000`. This method now requires HTTPS, and you will see an error response. Request `https://localhost:44350`, and you will see the message output. Figure 30-4 shows both responses.

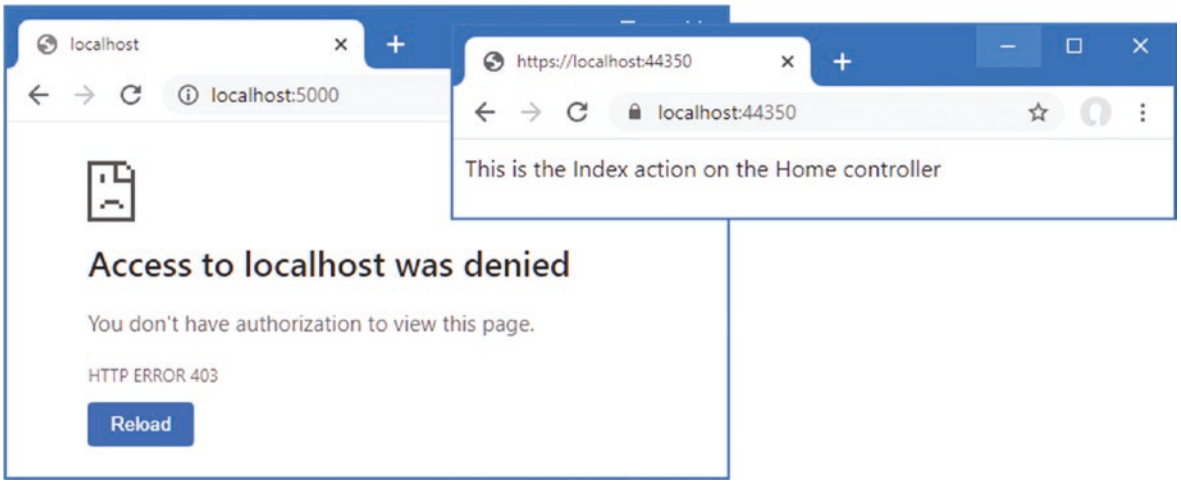

Figure 30-4. *Enforcing HTTPS in an action method*

■ **Tip** Clear your browser's history if you don't get the results you expect from the examples in this section. Browsers will often refuse to send requests to servers that have previously generated HTTPS errors, which is a good security practice but can be frustrating during development.

This approach works but has problems. The first problem is that the action method contains code that is more about implementing a security policy than about handling the request. A more serious problem is that including the HTTP-detecting code within the action method doesn't scale well and must be duplicated in every action method in the controller, as shown in Listing 30-10.

Listing 30-10. Adding Action Methods in the HomeController.cs File in the Controllers Folder

```
using Microsoft.AspNetCore.Mvc;
using Microsoft.AspNetCore.Http;

namespace WebApp.Controllers {

    public class HomeController : Controller {

        public IActionResult Index() {
            if (Request.IsHttps) {
                return View("Message",
                    "This is the Index action on the Home controller");
            } else {
                return new StatusCodeResult(StatusCodes.Status403Forbidden);
            }
        }

        public IActionResult Secure() {
            if (Request.IsHttps) {
                return View("Message",
                    "This is the Secure action on the Home controller");
```

```
            } else {
                return new StatusCodeResult(StatusCodes.Status403Forbidden);
            }
        }
    }
}
```

I must remember to implement the same check in every action method in every controller for which I want to require HTTPS. The code to implement the security policy is a substantial part of the—admittedly simple—controller, which makes the controller harder to understand, and it is only a matter of time before I forget to add it to a new action method, creating a hole in my security policy.

This is the type of problem that filters address. Listing 30-11 replaces my checks for HTTPS and implements a filter instead.

Listing 30-11. Applying a Filter in the HomeController.cs File in the Controllers Folder

```
using Microsoft.AspNetCore.Mvc;
using Microsoft.AspNetCore.Http;

namespace WebApp.Controllers {

    public class HomeController : Controller {

        [RequireHttps]
        public IActionResult Index() {
            return View("Message",
                "This is the Index action on the Home controller");
        }

        [RequireHttps]
        public IActionResult Secure() {
            return View("Message",
                "This is the Secure action on the Home controller");
        }
    }
}
```

The `RequireHttps` attribute applies one of the built-in filters provided by ASP.NET Core. This filter restricts access to action methods so that only HTTPS requests are supported and allows me to remove the security code from each method and focus on handling the successful requests.

■ **Note** The `RequireHttps` filter doesn't work the same way as my custom code. For `GET` requests, the `RequireHttps` attribute redirects the client to the originally requested URL, but it does so by using the `https` scheme so that a request to `http://localhost:5000` will be redirected to `https://localhost:5000`. This makes sense for most deployed applications but not during development because HTTP and HTTPS are on different local ports. The `RequireHttpsAttribute` class defines a protected method called `HandleNonHttpsRequest` that you can override to change the behavior. Alternatively, I re-create the original functionality from scratch in the "Understanding Authorization Filters" section.

I must still remember to apply the `RequireHttps` attribute to each action method, which means that I might forget. But filters have a useful trick: applying the attribute to a controller class has the same effect as applying it to each individual action method, as shown in Listing 30-12.

Listing 30-12. Applying a Filter to All Actions in the HomeController.cs File in the Controllers Folder

```
using Microsoft.AspNetCore.Mvc;
using Microsoft.AspNetCore.Http;

namespace WebApp.Controllers {

    [RequireHttps]
    public class HomeController : Controller {

        public IActionResult Index() {
            return View("Message",
                "This is the Index action on the Home controller");
        }

        public IActionResult Secure() {
            return View("Message",
                "This is the Secure action on the Home controller");
        }
    }
}
```

Filters can be applied with differing levels of granularity. If you want to restrict access to some actions but not others, then you can apply the `RequireHttps` attribute to just those methods. If you want to protect all the action methods, including any that you add to the controller in the future, then the `RequireHttps` attribute can be applied to the class. If you want to apply a filter to every action in an application, then you can use *global filters*, which I describe later in this chapter.

Using Filters in Razor Pages

Filters can also be used in Razor Pages. To implement the HTTPS-only policy in the `Message` Razor Pages, for example, I would have to add a handler method that inspects the connection, as shown in Listing 30-13.

Listing 30-13. Checking Connections in the Message.cshtml File in the Pages Folder

```
@page "/pages/message"
@model MessageModel
@using Microsoft.AspNetCore.Mvc.RazorPages
@using System.Collections.Generic
@using Microsoft.AspNetCore.Http

@if (Model.Message is string) {
    @Model.Message
} else if (Model.Message is IDictionary<string, string>) {
    var dict = Model.Message as IDictionary<string, string>;
    <table class="table table-sm table-striped table-bordered">
        <thead><tr><th>Name</th><th>Value</th></tr></thead>
        <tbody>
            @foreach (var kvp in dict) {
                <tr><td>@kvp.Key</td><td>@kvp.Value</td></tr>
            }
        </tbody>
    </table>
}
```

```
@functions {

    public class MessageModel : PageModel {

        public object Message { get; set; } = "This is the Message Razor Page";

        public IActionResult OnGet() {
            if (!Request.IsHttps) {
                return new StatusCodeResult(StatusCodes.Status403Forbidden);
            } else {
                return Page();
            }
        }
    }
}
```

The handler method works, but it is awkward and presents the same problems encountered with action methods. When using filters in Razor Pages, the attribute can be applied to the handler method or, as shown in Listing 30-14, to the entire class.

Listing 30-14. Applying a Filter in the Message.cshtml File in the Pages Folder

```
@page "/pages/message"
@model MessageModel
@using Microsoft.AspNetCore.Mvc.RazorPages
@using System.Collections.Generic
@using Microsoft.AspNetCore.Http

@if (Model.Message is string) {
    @Model.Message
} else if (Model.Message is IDictionary<string, string>) {
    var dict = Model.Message as IDictionary<string, string>;
    <table class="table table-sm table-striped table-bordered">
        <thead><tr><th>Name</th><th>Value</th></tr></thead>
        <tbody>
            @foreach (var kvp in dict) {
                <tr><td>@kvp.Key</td><td>@kvp.Value</td></tr>
            }
        </tbody>
    </table>
}

@functions {

    [RequireHttps]
    public class MessageModel : PageModel {

        public object Message { get; set; } = "This is the Message Razor Page";
    }
}
```

You will see a normal response if you request `https://localhost:44350/pages/message`. If you request the regular HTTP URL, `http://localhost:5000/pages/messages`, the filter will redirect the request, and you will see an error (as noted earlier, the `RequireHttps` filter redirects the browser to a port that is not enabled in the example application).

Understanding Filters

ASP.NET Core supports different types of filters, each of which is intended for a different purpose. Table 30-2 describes the filter categories.

Table 30-2. *The Filter Types*

Name	Description
Authorization filters	This type of filter is used to apply the application's authorization policy.
Resource filters	This type of filter is used to intercept requests, typically to implement features such as caching.
Action filters	This type of filter is used to modify the request before it is received by an action method or to modify the action result after it has been produced. This type of filter can be applied only to controllers and actions.
Page filters	This type of filter is used to modify the request before it is received by a Razor Page handler method or to modify the action result after it has been produced. This type of filter can be applied only to Razor Pages.
Result filters	This type of filter is used to alter the action result before it is executed or to modify the result after execution.
Exception filters	This type of filter is used to handle exceptions that occur during the execution of the action method or page handler.

Filters have their own pipeline and are executed in a specific order, as shown in Figure 30-5.

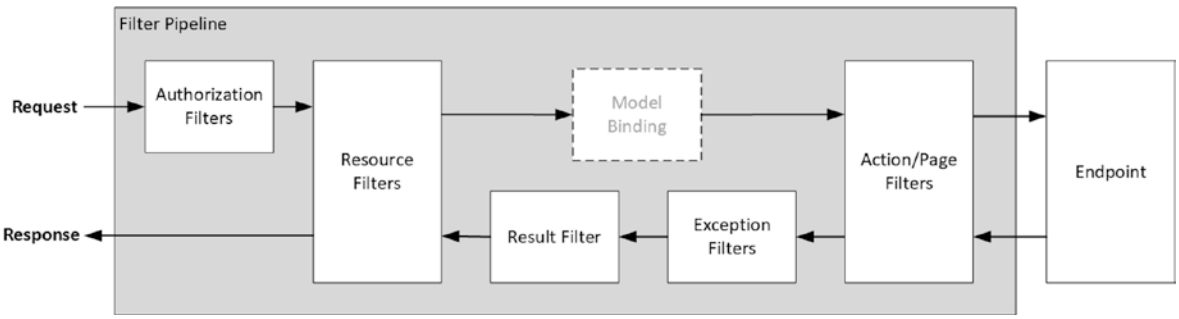

Figure 30-5. *The filter pipeline*

Filters can short-circuit the filter pipeline to prevent a request from being forwarded to the next filter. For example, an authorization filter can short-circuit the pipeline and return an error response if the user is unauthenticated. The resource, action, and page filters are able to inspect the request before and after it has been handled by the endpoint, allowing these types of filter to short-circuit the pipeline; to alter the request before it is handled; or to alter the response. (I have simplified the flow of filters in Figure 30-5. Page filters run before and after the model binding process, as described in the "Understanding Page Filters" section.)

Each type of filter is implemented using interfaces defined by ASP.NET Core, which also provides base classes that make it easy to apply some types of filters as attributes. I describe each interface and the attribute classes in the sections that follow, but they are shown in Table 30-3 for quick reference.

Table 30-3. *The Filter Types, Interfaces, and Attribute Base Classes*

Filter Type	Interfaces	Attribute Class
Authorization filters	`IAuthorizationFilter` `IAsyncAuthorizationFilter`	No attribute class is provided.
Resource filters	`IResourceFilter` `IAsyncResourceFilter`	No attribute class is provided.
Action filters	`IActionFilter` `IAsyncActionFilter`	`ActionFilterAttribute`
Page filters	`IPageFilter` `IAsyncPageFilter`	No attribute class is provided.

Result filters	`IResultFilter`
	`IAsyncResultFilter`
	`IAlwaysRunResultFilter`
	`IAsyncAlwaysRunResultFilter`
		`ResultFilterAttribute`

Exception Filters	`IExceptionFilter` `IAsyncExceptionFilter`	`ExceptionFilterAttribute`

Creating Custom Filters

Filters implement the `IFilterMetadata` interface, which is in the `Microsoft.AspNetCore.Mvc.Filters` namespace. Here is the interface:

```
namespace Microsoft.AspNetCore.Mvc.Filters {
    public interface IFilterMetadata { }
}
```

The interface is empty and doesn't require a filter to implement any specific behaviors. This is because each of the categories of filter described in the previous section works in a different way. Filters are provided with context data in the form of a `FilterContext` object. For convenience, Table 30-4 describes the properties that `FilterContext` provides.

Table 30-4. *The FilterContext Properties*

Name	Description
`ActionDescriptor`	This property returns an `ActionDescriptor` object, which describes the action method.
`HttpContext`	This property returns an `HttpContext` object, which provides details of the HTTP request and the HTTP response that will be sent in return.
`ModelState`	This property returns a `ModelStateDictionary` object, which is used to validate data sent by the client.
`RouteData`	This property returns a `RouteData` object that describes the way that the routing system has processed the request.
`Filters`	This property returns a list of filters that have been applied to the action method, expressed as an `IList<IFilterMetadata>`.

Understanding Authorization Filters

Authorization filters are used to implement an application's security policy. Authorization filters are executed before other types of filter and before the endpoint handles the request. Here is the definition of the `IAuthorizationFilter` interface:

```
namespace Microsoft.AspNetCore.Mvc.Filters {

    public interface IAuthorizationFilter : IFilterMetadata {

        void OnAuthorization(AuthorizationFilterContext context);
    }
}
```

The OnAuthorization method is called to provide the filter with the opportunity to authorize the request. For asynchronous authorization filters, here is the definition of the IAsyncAuthorizationFilter interface:

```
using System.Threading.Tasks;

namespace Microsoft.AspNetCore.Mvc.Filters {

    public interface IAsyncAuthorizationFilter : IFilterMetadata {

        Task OnAuthorizationAsync(AuthorizationFilterContext context);
    }
}
```

The OnAuthorizationAsync method is called so that the filter can authorize the request. Whichever interface is used, the filter receives context data describing the request through an AuthorizationFilterContext object, which is derived from the FilterContext class and adds one important property, as described in Table 30-5.

Table 30-5. *The AuthorizationFilterContext Property*

Name	Description
Result	This IActionResult property is set by authorization filters when the request doesn't comply with the application's authorization policy. If this property is set, then ASP.NET Core executes the IActionResult instead of invoking the endpoint.

Creating an Authorization Filter

To demonstrate how authorization filters work, I created a Filters folder in the example project, added a class file called HttpsOnlyAttribute.cs, and used it to define the filter shown in Listing 30-15.

Listing 30-15. The Contents of the HttpsOnlyAttribute.cs File in the Filters Folder

```
using System;
using Microsoft.AspNetCore.Http;
using Microsoft.AspNetCore.Mvc;
using Microsoft.AspNetCore.Mvc.Filters;

namespace WebApp.Filters {
    public class HttpsOnlyAttribute : Attribute, IAuthorizationFilter {

        public void OnAuthorization(AuthorizationFilterContext context) {
            if (!context.HttpContext.Request.IsHttps) {
                context.Result =
                    new StatusCodeResult(StatusCodes.Status403Forbidden);
            }
        }
    }
}
```

An authorization filter does nothing if a request complies with the authorization policy and inaction allows ASP.NET Core to move on to the next filter and, eventually, to execute the endpoint. If there is a problem, the filter sets the Result property of the AuthorizationFilterContext object that is passed to the OnAuthorization method. This prevents further execution from happening and provides a result to return to the client. In the listing, the HttpsOnlyAttribute class inspects the IsHttps property of the HttpRequest context object and sets the Result property to interrupt execution if the request has been made without HTTPS. Authorization filters can be applied to controllers, action methods, and Razor Pages. Listing 30-16 applies the new filter to the Home controller.

Listing 30-16. Applying a Custom Filter in the HomeController.cs File in the Controllers Folder

```
using Microsoft.AspNetCore.Mvc;
using Microsoft.AspNetCore.Http;
using WebApp.Filters;

namespace WebApp.Controllers {

    [HttpsOnly]
    public class HomeController : Controller {

        public IActionResult Index() {
            return View("Message",
                "This is the Index action on the Home controller");
        }

        public IActionResult Secure() {
            return View("Message",
                "This is the Secure action on the Home controller");
        }
    }
}
```

This filter re-creates the functionality that I included in the action methods in Listing 30-10. This is less useful in real projects than doing a redirection like the built-in `RequireHttps` filter because users won't understand the meaning of a 403 status code, but it does provide a useful example of how authorization filters work. Restart ASP.NET Core and request `http://localhost:5000`, and you will see the effect of the filter, as shown in Figure 30-6. Request `https://localhost:44350`, and you will receive the response from the action method, also shown in the figure.

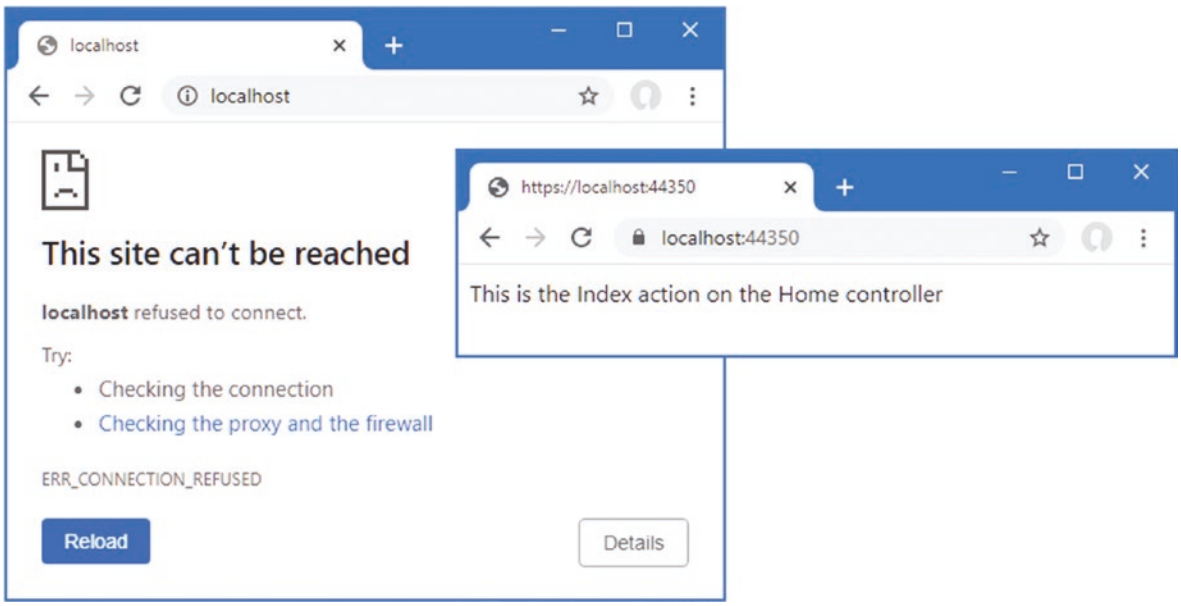

Figure 30-6. *Applying a custom authorization filter*

Understanding Resource Filters

Resource filters are executed twice for each request: before the ASP.NET Core model binding process and again before the action result is processed to generate the result. Here is the definition of the `IResourceFilter` interface:

```
namespace Microsoft.AspNetCore.Mvc.Filters {
    public interface IResourceFilter : IFilterMetadata {

        void OnResourceExecuting(ResourceExecutingContext context);

        void OnResourceExecuted(ResourceExecutedContext context);
    }
}
```

The `OnResourceExecuting` method is called when a request is being processed, and the `OnResourceExecuted` method is called after the endpoint has handled the request but before the action result is executed. For asynchronous resource filters, here is the definition of the `IAsyncResourceFilter` interface:

```
namespace Microsoft.AspNetCore.Mvc.Filters {
    public interface IAsyncResourceFilter : IFilterMetadata {

        Task OnResourceExecutionAsync(ResourceExecutingContext context,
            ResourceExecutionDelegate next);
    }
}
```

This interface defines a single method that receives a context object and a delegate to invoke. The resource filter is able to inspect the request before invoking the delegate and inspect the response before it is executed. The `OnResourceExecuting` method is provided with context using the `ResourceExecutingContext` class, which defines the property shown in Table 30-6 in addition to those defined by the `FilterContext` class.

Table 30-6. *The Property Defined by the ResourceExecutingContext Class*

Name	Description
`Result`	This `IActionResult` property is used to provide a result to short-circuit the pipeline.

The `OnResourceExecuted` method is provided with context using the `ResourceExecutedContext` class, which defines the properties shown in Table 30-7, in addition to those defined by the `FilterContext` class.

Table 30-7. *The Properties Defined by the ResourceExecutedContext Class*

Name	Description
`Result`	This `IActionResult` property provides the action result that will be used to produce a response.
`ValueProviderFactories`	This property returns an `IList<IValueProviderFactory>`, which provides access to the objects that provide values for the model binding process.

Creating a Resource Filter

Resource filters are usually used where it is possible to short-circuit the pipeline and provide a response early, such as when implementing data caching. To create a simple caching filter, add a class file called `SimpleCacheAttribute.cs` to the `Filters` folder with the code shown in Listing 30-17.

FILTERS AND DEPENDENCY INJECTION

Filters that are applied as attributes cannot declare dependencies in their constructors unless they implement the `IFilterFactory` interface and take responsibility for creating instances directly, as explained in the "Creating Filter Factories" section later in this chapter.

Listing 30-17. The Contents of the SimpleCacheAttribute.cs File in the Filters Folder

```
using Microsoft.AspNetCore.Http;
using Microsoft.AspNetCore.Mvc;
using Microsoft.AspNetCore.Mvc.Filters;
using System;
using System.Collections.Generic;

namespace WebApp.Filters {

    public class SimpleCacheAttribute : Attribute, IResourceFilter {
        private Dictionary<PathString, IActionResult> CachedResponses
            = new Dictionary<PathString, IActionResult>();

        public void OnResourceExecuting(ResourceExecutingContext context) {
            PathString path = context.HttpContext.Request.Path;
            if (CachedResponses.ContainsKey(path)) {
                context.Result = CachedResponses[path];
                CachedResponses.Remove(path);

            }
        }

        public void OnResourceExecuted(ResourceExecutedContext context) {
            CachedResponses.Add(context.HttpContext.Request.Path, context.Result);
        }
    }
}
```

This filter isn't an especially useful cache, but it does show how a resource filter works. The `OnResourceExecuting` method provides the filter with the opportunity to short-circuit the pipeline by setting the context object's `Result` property to a previously cached action result. If a value is assigned to the `Result` property, then the filter pipeline is short-circuited, and the action result is executed to produce the response for the client. Cached action results are used only once and then discarded from the cache. If no value is assigned to the `Result` property, then the request passes to the next step in the pipeline, which may be another filter or the endpoint.

The `OnResourceExecuted` method provides the filter with the action results that are produced when the pipeline is not short-circuited. In this case, the filter caches the action result so that it can be used for subsequent requests. Resource filters can be applied to controllers, action methods, and Razor Pages. Listing 30-18 applies the custom resource filter to the `Message` Razor Page and adds a timestamp that will help determine when an action result is cached.

Listing 30-18. Applying a Resource Filter in the Message.cshtml File in the Pages Folder

```
@page "/pages/message"
@model MessageModel
@using Microsoft.AspNetCore.Mvc.RazorPages
@using System.Collections.Generic
@using Microsoft.AspNetCore.Http
@using WebApp.Filters
```

```
@if (Model.Message is string) {
    @Model.Message
} else if (Model.Message is IDictionary<string, string>) {
    var dict = Model.Message as IDictionary<string, string>;
    <table class="table table-sm table-striped table-bordered">
        <thead><tr><th>Name</th><th>Value</th></tr></thead>
        <tbody>
            @foreach (var kvp in dict) {
                <tr><td>@kvp.Key</td><td>@kvp.Value</td></tr>
            }
        </tbody>
    </table>
}

@functions {

    [RequireHttps]
    [SimpleCache]
    public class MessageModel : PageModel {

        public object Message { get; set; } =
            $"{DateTime.Now.ToLongTimeString()}: This is the Message Razor Page";
    }
}
```

To see the effect of the resource filter, restart ASP.NET Core and request `https://localhost:44350/pages/message`. Since this is the first request for the path, there will be no cached result, and the request will be forwarded along the pipeline. As the response is processed, the resource filter will cache the action result for future use. Reload the browser to repeat the request, and you will see the same timestamp, indicating that the cached action result has been used. The cached item is removed when it is used, which means that reloading the browser will generate a response with a fresh timestamp, as shown in Figure 30-7.

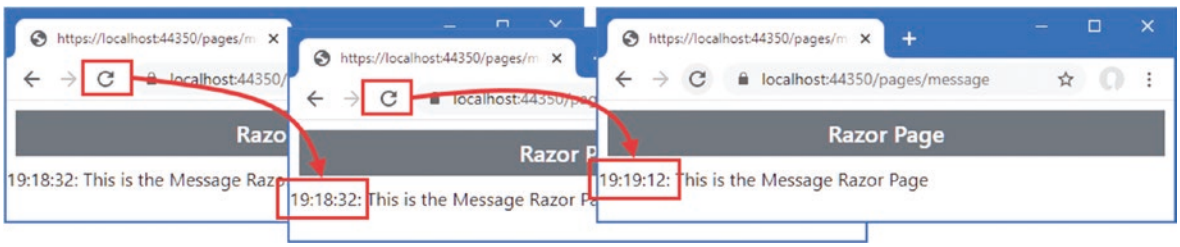

Figure 30-7. *Using a resource filter*

Creating an Asynchronous Resource Filter

The interface for asynchronous resource filters uses a single method that receives a delegate used to forward the request along the filter pipeline. Listing 30-19 reimplements the caching filter from the previous example so that it implements the `IAsyncResourceFilter` interface.

Listing 30-19. Creating an Asynchronous Filter in the SimpleCacheAttribute.cs File in the Filters Folder

```
using Microsoft.AspNetCore.Http;
using Microsoft.AspNetCore.Mvc;
using Microsoft.AspNetCore.Mvc.Filters;
using System;
using System.Collections.Generic;
using System.Threading.Tasks;
```

```
namespace WebApp.Filters {

    public class SimpleCacheAttribute : Attribute, IAsyncResourceFilter {
        private Dictionary<PathString, IActionResult> CachedResponses
            = new Dictionary<PathString, IActionResult>();

        public async Task OnResourceExecutionAsync(ResourceExecutingContext context,
                ResourceExecutionDelegate next) {
            PathString path = context.HttpContext.Request.Path;
            if (CachedResponses.ContainsKey(path)) {
                context.Result = CachedResponses[path];
                CachedResponses.Remove(path);
            } else {
                ResourceExecutedContext execContext = await next();
                CachedResponses.Add(context.HttpContext.Request.Path,
                    execContext.Result);
            }
        }
    }
}
```

The `OnResourceExecutionAsync` method receives a `ResourceExecutingContext` object, which is used to determine whether the pipeline can be short-circuited. If it cannot, the delegate is invoked without arguments and asynchronously produces a `ResourceExecutedContext` object when the request has been handled and is making its way back along the pipeline. Restart ASP.NET Core and repeat the requests described in the previous section, and you will see the same caching behavior, as shown in Figure 30-7.

■ **Caution** It is important not to confuse the two context objects. The action result produced by the endpoint is available only in the context object that is returned by the delegate.

Understanding Action Filters

Like resource filters, action filters are executed twice. The difference is that action filters are executed after the model binding process, whereas resource filters are executed before model binding. This means that resource filters can short-circuit the pipeline and minimize the work that ASP.NET Core does on the request. Action filters are used when model binding is required, which means they are used for tasks such as altering the model or enforcing validation. Action filters can be applied only to controllers and action methods, unlike resource filters, which can also be used with Razor Pages. (The Razor Pages equivalent to action filters is the page filter, described in the "Understanding Page Filters" section.) Here is the `IActionFilter` interface:

```
namespace Microsoft.AspNetCore.Mvc.Filters {

    public interface IActionFilter : IFilterMetadata {

        void OnActionExecuting(ActionExecutingContext context);

        void OnActionExecuted(ActionExecutedContext context);
    }
}
```

When an action filter has been applied to an action method, the `OnActionExecuting` method is called just before the action method is invoked, and the `OnActionExecuted` method is called just after. Action filters are provided with context data through two different context classes: `ActionExecutingContext` for the `OnActionExecuting` method and `ActionExecutedContext` for the `OnActionExecuted` method.

The ActionExecutingContext class, which is used to describe an action that is about to be invoked, defines the properties described in Table 30-8, in addition to the FilterContext properties.

Table 30-8. *The ActionExecutingContext Property*

Name	Description
Controller	This property returns the controller whose action method is about to be invoked. (Details of the action method are available through the ActionDescriptor property inherited from the base classes.)
ActionArguments	This property returns a dictionary of the arguments that will be passed to the action method, indexed by name. The filter can insert, remove, or change the arguments.
Result	If the filter assigns an IActionResult to this property, then the pipeline will be short-circuited, and the action result will be used to generate the response to the client without invoking the action method.

The ActionExecutedContext class is used to represent an action that has been executed and defines the properties described in Table 30-9, in addition to the FilterContext properties.

Table 30-9. *The ActionExecutedContext Properties*

Name	Description
Controller	This property returns the Controller object whose action method will be invoked.
Canceled	This bool property is set to true if another action filter has short-circuited the pipeline by assigning an action result to the Result property of the ActionExecutingContext object.
Exception	This property contains any Exception that was thrown by the action method.
ExceptionDispatchInfo	This method returns an ExceptionDispatchInfo object that contains the stack trace details of any exception thrown by the action method.
ExceptionHandled	Setting this property to true indicates that the filter has handled the exception, which will not be propagated any further.
Result	This property returns the IActionResult produced by the action method. The filter can change or replace the action result if required.

Asynchronous action filters are implemented using the IAsyncActionFilter interface.

```
namespace Microsoft.AspNetCore.Mvc.Filters {

    public interface IAsyncActionFilter : IFilterMetadata {

        Task OnActionExecutionAsync(ActionExecutingContext context,
            ActionExecutionDelegate next);
    }
}
```

This interface follows the same pattern as the IAsyncResourceFilter interface described earlier in the chapter. The OnActionExecutionAsync method is provided with an ActionExecutingContext object and a delegate. The ActionExecutingContext object describes the request before it is received by the action method. The filter can short-circuit the pipeline by assigning a value to the ActionExecutingContext.Result property or pass it along by invoking the delegate. The delegate asynchronously produces an ActionExecutedContext object that describes the result from the action method.

Creating an Action Filter

Add a class file called ChangeArgAttribute.cs to the Filters folder and use it to define the action filter shown in Listing 30-20.

Listing 30-20. The Contents of the ChangeArgAttribute.cs File in the Filters Folder

```
using Microsoft.AspNetCore.Mvc.Filters;
using System;
using System.Threading.Tasks;

namespace WebApp.Filters {
    public class ChangeArgAttribute : Attribute, IAsyncActionFilter {

        public async Task OnActionExecutionAsync(ActionExecutingContext context,
                ActionExecutionDelegate next) {

            if (context.ActionArguments.ContainsKey("message1")) {
                context.ActionArguments["message1"] = "New message";
            }
            await next();
        }
    }
}
```

The filter looks for an action argument named message1 and changes the value that will be used to invoke the action method. The values that will be used for the action method arguments are determined by the model binding process. Listing 30-21 adds an action method to the Home controller and applies the new filter.

Listing 30-21. Applying a Filter in the HomeController.cs File in the Controllers Folder

```
using Microsoft.AspNetCore.Mvc;
using Microsoft.AspNetCore.Http;
using WebApp.Filters;

namespace WebApp.Controllers {

    [HttpsOnly]
    public class HomeController : Controller {

        public IActionResult Index() {
            return View("Message",
                "This is the Index action on the Home controller");
        }

        public IActionResult Secure() {
            return View("Message",
                "This is the Secure action on the Home controller");
        }

        [ChangeArg]
        public IActionResult Messages(string message1, string message2 = "None") {
            return View("Message", $"{message1}, {message2}");
        }
    }
}
```

Restart ASP.NET Core and request `https://localhost:44350/home/messages?message1=hello&message2=world`. The model binding process will locate values for the parameters defined by the action method from the query string. One of those values is then modified by the action filter, producing the response shown in Figure 30-8.

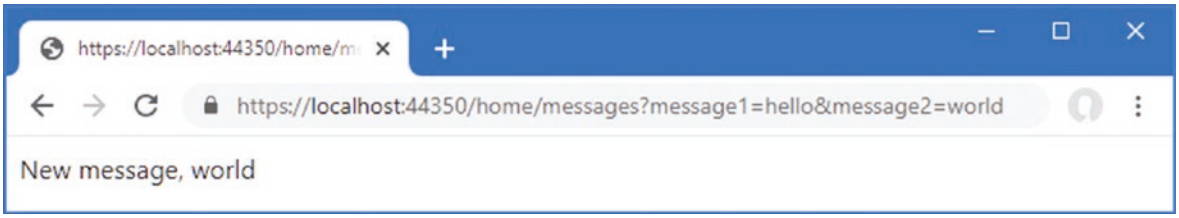

Figure 30-8. *Using an action filter*

Implementing an Action Filter Using the Attribute Base Class

Action attributes can also be implemented by deriving from the `ActionFilterAttribute` class, which extends `Attribute` and inherits both the `IActionFilter` and `IAsyncActionFilter` interfaces so that implementation classes override just the methods they require. In Listing 30-22, I have reimplemented the `ChangeArg` filter so that it is derived from `ActionFilterAttribute`.

Listing 30-22. Using a Filter Base Class in the ChangeArgsAttribute.cs File in the Filters Folder

```
using Microsoft.AspNetCore.Mvc.Filters;
using System;
using System.Threading.Tasks;

namespace WebApp.Filters {
    public class ChangeArgAttribute : ActionFilterAttribute {

        public override async Task OnActionExecutionAsync(
            ActionExecutingContext context,
                ActionExecutionDelegate next) {

            if (context.ActionArguments.ContainsKey("message1")) {
                context.ActionArguments["message1"] = "New message";
            }
            await next();
        }
    }
}
```

This attribute behaves in just the same way as the earlier implementation, and the use of the base class is a matter of preference. Restart ASP.NET Core and request `https://localhost:44350/home/messages?message1=hello&message2=world`, and you will see the response shown in Figure 30-8.

Using the Controller Filter Methods

The `Controller` class, which is the base for controllers that render Razor views, implements the `IActionFilter` and `IAsyncActionFilter` interfaces, which means you can define functionality and apply it to the actions defined by a controller and any derived controllers. Listing 30-23 implements the `ChangeArg` filter functionality directly in the `HomeController` class.

Listing 30-23. Using Action Filter Methods in the HomeController.cs File in the Controllers Folder

```
using Microsoft.AspNetCore.Mvc;
using Microsoft.AspNetCore.Http;
using WebApp.Filters;
using Microsoft.AspNetCore.Mvc.Filters;

namespace WebApp.Controllers {

    [HttpsOnly]
    public class HomeController : Controller {

        public IActionResult Index() {
            return View("Message",
                "This is the Index action on the Home controller");
        }

        public IActionResult Secure() {
            return View("Message",
                "This is the Secure action on the Home controller");
        }

        //[ChangeArg]
        public IActionResult Messages(string message1, string message2 = "None") {
            return View("Message", $"{message1}, {message2}");
        }

        public override void OnActionExecuting(ActionExecutingContext context) {
            if (context.ActionArguments.ContainsKey("message1")) {
                context.ActionArguments["message1"] = "New message";
            }
        }
    }
}
```

The Home controller overrides the Controller implementation of the OnActionExecuting method and uses it to modify the arguments that will be passed to the execution method.

Restart ASP.NET Core and request https://localhost:44350/home/messages?message1=hello&message2=world, and you will see the response shown in Figure 30-8.

Understanding Page Filters

Page filters are the Razor Page equivalent of action filters. Here is the IPageFilter interface, which is implemented by synchronous page filters:

```
namespace Microsoft.AspNetCore.Mvc.Filters {

    public interface IPageFilter : IFilterMetadata {

        void OnPageHandlerSelected(PageHandlerSelectedContext context);

        void OnPageHandlerExecuting(PageHandlerExecutingContext context);

        void OnPageHandlerExecuted(PageHandlerExecutedContext context);
    }
}
```

The OnPageHandlerSelected method is invoked after ASP.NET Core has selected the page handler method but before model binding has been performed, which means the arguments for the handler method have not been determined. This method receives context through the PageHandlerSelectedContext class, which defines the properties shown in Table 30-10, in addition to those defined by the FilterContext class. This method cannot be used to short-circuit the pipeline, but it can alter the handler method that will receive the request.

Table 30-10. *The PageHandlerSelectedContext Properties*

Name	Description
ActionDescriptor	This property returns the description of the Razor Page.
HandlerMethod	This property returns a HandlerMethodDescriptor object that describes the selected handler method.
HandlerInstance	This property returns the instance of the Razor Page that will handle the request.

The OnPageHandlerExecuting method is called after the model binding process has completed but before the page handler method is invoked. This method receives context through the PageHandlerExecutingContext class, which defines the properties shown in Table 30-11.

Table 30-11. *The PageHandlerExecutingContext Properties*

Name	Description
HandlerArguments	This property returns a dictionary containing the page handler arguments, indexed by name.
Result	The filter can short-circuit the pipeline by assigning an IActionResult object to this property.

The OnPageHandlerExecuted method is called after the page handler method has been invoked but before the action result is processed to create a response. This method receives context through the PageHandlerExecutedContext class, which defines the properties shown in Table 30-12 in addition to the PageHandlerSelectedContext properties.

Table 30-12. *The PageHandlerExecutedContext Properties*

Name	Description
Canceled	This property returns true if another filter short-circuited the filter pipeline.
Exception	This property returns an exception if one was thrown by the page handler method.
ExceptionHandled	This property is set to true to indicate that an exception thrown by the page handler has been handled by the filter.
Result	This property returns the action result that will be used to create a response for the client.

Asynchronous page filters are created by implementing the IAsyncPageFilter interface, which is defined like this:

```
namespace Microsoft.AspNetCore.Mvc.Filters {
    public interface IAsyncPageFilter : IFilterMetadata {

        Task OnPageHandlerSelectionAsync(PageHandlerSelectedContext context);

        Task OnPageHandlerExecutionAsync(PageHandlerExecutingContext context,
            PageHandlerExecutionDelegate next);
    }
}
```

The OnPageHandlerSelectionAsync is called after the handler method is selected and is equivalent to the synchronous OnPageHandlerSelected method. The OnPageHandlerExecutionAsync is provided with a PageHandlerExecutingContext object that allows it to short-circuit the pipeline and a delegate that is invoked to pass on the request. The delegate produces a PageHandlerExecutedContext object that can be used to inspect or alter the action result produced by the handler method.

Creating a Page Filter

To create a page filter, add a class file named ChangePageArgs.cs to the Filters folder and use it to define the class shown in Listing 30-24.

Listing 30-24. The Contents of the ChangePageArgs.cs File in the Filters Folder

```
using Microsoft.AspNetCore.Mvc.Filters;
using System;

namespace WebApp.Filters {
    public class ChangePageArgs : Attribute, IPageFilter {

        public void OnPageHandlerSelected(PageHandlerSelectedContext context) {
            // do nothing
        }

        public void OnPageHandlerExecuting(PageHandlerExecutingContext context) {
            if (context.HandlerArguments.ContainsKey("message1")) {
                context.HandlerArguments["message1"] = "New message";
            }
        }

        public void OnPageHandlerExecuted(PageHandlerExecutedContext context) {
            // do nothing
        }
    }
}
```

The page filter in Listing 30-24 performs the same task as the action filter I created in the previous section. In Listing 30-25, I have modified the Message Razor Page to define a handler method and have applied the page filter. Page filters can be applied to individual handler methods or, as in the listing, to the page model class, in which case the filter is used for all handler methods. (I also disabled the SimpleCache filter in Listing 30-25. Resource filters can work alongside page filters. I disabled this filter because caching responses makes some of the examples more difficult to follow.)

Listing 30-25. Using a Page Filter in the Message.cshtml File in the Pages Folder

```
@page "/pages/message"
@model MessageModel
@using Microsoft.AspNetCore.Mvc.RazorPages
@using System.Collections.Generic
@using Microsoft.AspNetCore.Http
@using WebApp.Filters

@if (Model.Message is string) {
    @Model.Message
} else if (Model.Message is IDictionary<string, string>) {
    var dict = Model.Message as IDictionary<string, string>;
    <table class="table table-sm table-striped table-bordered">
        <thead><tr><th>Name</th><th>Value</th></tr></thead>
        <tbody>
```

```
            @foreach (var kvp in dict) {
                <tr><td>@kvp.Key</td><td>@kvp.Value</td></tr>
            }
        </tbody>
    </table>
}

@functions {

    [RequireHttps]
    //[SimpleCache]
    [ChangePageArgs]
    public class MessageModel : PageModel {

        public object Message { get; set; } =
            $"{DateTime.Now.ToLongTimeString()}: This is the Message Razor Page";

        public void OnGet(string message1, string message2) {
            Message = $"{message1}, {message2}";
        }
    }
}
```

Restart ASP.NET Core and request `https://localhost:44350/pages/message?message1=hello&message2=world`. The page filter will replace the value of the `message1` argument for the `OnGet` handler method, which produces the response shown in Figure 30-9.

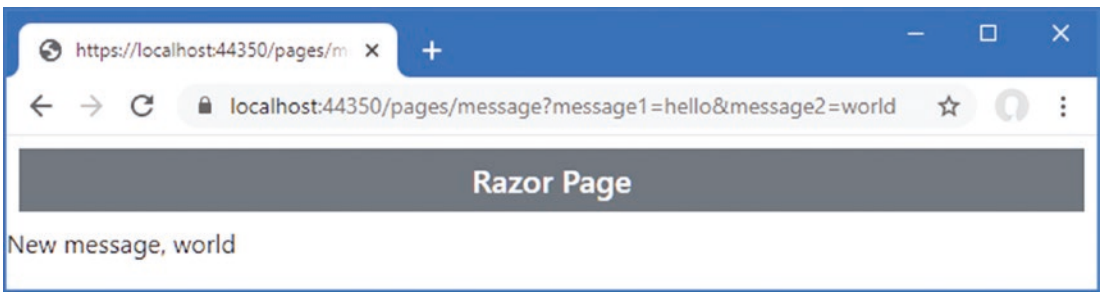

Figure 30-9. *Using a page filter*

Using the Page Model Filter Methods

The `PageModel` class, which is used as the base for page model classes, implements the `IPageFilter` and `IAsyncPageFilter` interfaces, which means you can add filter functionality directly to a page model, as shown in Listing 30-26.

Listing 30-26. Using the PageModel Filter Methods in the Message.cshtml File in the Pages Folder

```
@page "/pages/message"
@model MessageModel
@using Microsoft.AspNetCore.Mvc.RazorPages
@using System.Collections.Generic
@using Microsoft.AspNetCore.Http
@using WebApp.Filters
@using Microsoft.AspNetCore.Mvc.Filters
```

```
@if (Model.Message is string) {
    @Model.Message
} else if (Model.Message is IDictionary<string, string>) {
    var dict = Model.Message as IDictionary<string, string>;
    <table class="table table-sm table-striped table-bordered">
        <thead><tr><th>Name</th><th>Value</th></tr></thead>
        <tbody>
            @foreach (var kvp in dict) {
                <tr><td>@kvp.Key</td><td>@kvp.Value</td></tr>
            }
        </tbody>
    </table>
}

@functions {

    [RequireHttps]
    //[SimpleCache]
    //[ChangePageArgs]
    public class MessageModel : PageModel {

        public object Message { get; set; } =
            $"{DateTime.Now.ToLongTimeString()}: This is the Message Razor Page";

        public void OnGet(string message1, string message2) {
            Message = $"{message1}, {message2}";
        }

        public override void OnPageHandlerExecuting(
                PageHandlerExecutingContext context) {
            if (context.HandlerArguments.ContainsKey("message1")) {
                context.HandlerArguments["message1"] = "New message";
            }
        }
    }
}
```

Request `https://localhost:44350/pages/message?message1=hello&message2=world`. The method implemented by the page model class in Listing 30-26 will produce the same result as shown in Figure 30-9.

Understanding Result Filters

Result filters are executed before and after an action result is used to generate a response, allowing responses to be modified after they have been handled by the endpoint. Here is the definition of the `IResultFilter` interface:

```
namespace Microsoft.AspNetCore.Mvc.Filters {
    public interface IResultFilter : IFilterMetadata {

        void OnResultExecuting(ResultExecutingContext context);

        void OnResultExecuted(ResultExecutedContext context);
    }
}
```

The `OnResultExecuting` method is called after the endpoint has produced an action result. This method receives context through the `ResultExecutingContext` class, which defines the properties described in Table 30-13, in addition to those defined by the `FilterContext` class.

Table 30-13. *The ResultExecutingContext Class Properties*

Name	Description
`Result`	This property returns the action result produced by the endpoint.
`ValueProviderFactories`	This property returns an `IList<IValueProviderFactory>`, which provides access to the objects that provide values for the model binding process.

The `OnResultExecuted` method is called after the action result has been executed to generate the response for the client. This method receives context through the `ResultExecutedContext` class, which defines the properties shown in Table 30-14, in addition to those it inherits from the `FilterContext` class.

Table 30-14. *The ResultExecutedContext Class*

Name	Description
`Canceled`	This property returns `true` if another filter short-circuited the filter pipeline.
`Controller`	This property returns the object that contains the endpoint.
`Exception`	This property returns an exception if one was thrown by the page handler method.
`ExceptionHandled`	This property is set to `true` to indicate that an exception thrown by the page handler has been handled by the filter.
`Result`	This property returns the action result that will be used to create a response for the client. This property is read-only.

Asynchronous result filters implement the `IAsyncResultFilter` interface, which is defined like this:

```
namespace Microsoft.AspNetCore.Mvc.Filters {

    public interface IAsyncResultFilter : IFilterMetadata {

        Task OnResultExecutionAsync(ResultExecutingContext context,
            ResultExecutionDelegate next);
    }
}
```

This interface follows the pattern established by the other filter types. The `OnResultExecutionAsync` method is invoked with a context object whose `Result` property can be used to alter the response and a delegate that will forward the response along the pipeline.

Understanding Always-Run Result Filters

Filters that implement the `IResultFilter` and `IAsyncResultFilter` interfaces are used only when a request is handled normally by the endpoint. They are not used if another filter short-circuits the pipeline or if there is an exception. Filters that need to inspect or alter the response, even when the pipeline is short-circuited, can implement the `IAlwaysRunResultFilter` or `IAsyncAlwaysRunResultFilter` interface. These interfaces derived from `IResultFilter` and `IAsyncResultFilter` but define no new features. Instead, ASP.NET Core detects the always-run interfaces and always applies the filters.

Creating a Result Filter

Add a class file named `ResultDiagnosticsAttribute.cs` to the `Filters` folder and use it to define the filter shown in Listing 30-27.

Listing 30-27. The Contents of the ResultDiagnosticsAttribute.cs File in the Filters Folder

```
using Microsoft.AspNetCore.Mvc;
using Microsoft.AspNetCore.Mvc.Filters;
using Microsoft.AspNetCore.Mvc.ModelBinding;
using Microsoft.AspNetCore.Mvc.RazorPages;
using Microsoft.AspNetCore.Mvc.ViewFeatures;
using System;
using System.Collections.Generic;
using System.Threading.Tasks;

namespace WebApp.Filters {

    public class ResultDiagnosticsAttribute : Attribute, IAsyncResultFilter {

        public async Task OnResultExecutionAsync(
                ResultExecutingContext context, ResultExecutionDelegate next) {

            if (context.HttpContext.Request.Query.ContainsKey("diag")) {
                Dictionary<string, string> diagData =
                    new Dictionary<string, string> {
                        {"Result type", context.Result.GetType().Name }
                    };
                if (context.Result is ViewResult vr) {
                    diagData["View Name"] = vr.ViewName;
                    diagData["Model Type"] = vr.ViewData.Model.GetType().Name;
                    diagData["Model Data"] = vr.ViewData.Model.ToString();
                } else if (context.Result is PageResult pr) {
                    diagData["Model Type"] = pr.Model.GetType().Name;
                    diagData["Model Data"] = pr.ViewData.Model.ToString();
                }
                context.Result = new ViewResult() {
                    ViewName = "/Views/Shared/Message.cshtml",
                    ViewData = new ViewDataDictionary(
                                    new EmptyModelMetadataProvider(),
                                    new ModelStateDictionary()) {
                        Model = diagData
                    }
                };
            }
            await next();
        }
    }
}
```

This filter examines the request to see whether it contains a query string parameter named `diag`. If it does, then the filter creates a result that displays diagnostic information instead of the output produced by the endpoint. The filter in Listing 30-27 will work with the actions defined by the `Home` controller or the `Message` Razor Page. Listing 30-28 applies the result filter to the `Home` controller.

■ **Tip** Notice that I use a fully qualified name for the view when I create the action result in Listing 30-27. This avoids a problem with filters applied to Razor Pages, where ASP.NET Core tries to execute the new result as a Razor Page and throws an exception about the model type.

Listing 30-28. Applying a Result Filter in the HomeController.cs File in the Controllers Folder

```
using Microsoft.AspNetCore.Mvc;
using Microsoft.AspNetCore.Http;
using WebApp.Filters;
using Microsoft.AspNetCore.Mvc.Filters;

namespace WebApp.Controllers {

    [HttpsOnly]
    [ResultDiagnostics]
    public class HomeController : Controller {

        public IActionResult Index() {
            return View("Message",
                "This is the Index action on the Home controller");
        }

        public IActionResult Secure() {
            return View("Message",
                "This is the Secure action on the Home controller");
        }

        //[ChangeArg]
        public IActionResult Messages(string message1, string message2 = "None") {
            return View("Message", $"{message1}, {message2}");
        }

        public override void OnActionExecuting(ActionExecutingContext context) {
            if (context.ActionArguments.ContainsKey("message1")) {
                context.ActionArguments["message1"] = "New message";
            }
        }
    }
}
```

Restart ASP.NET Core and request `https://localhost:44350/?diag`. The query string parameter will be detected by the filter, which will generate the diagnostic information shown in Figure 30-10.

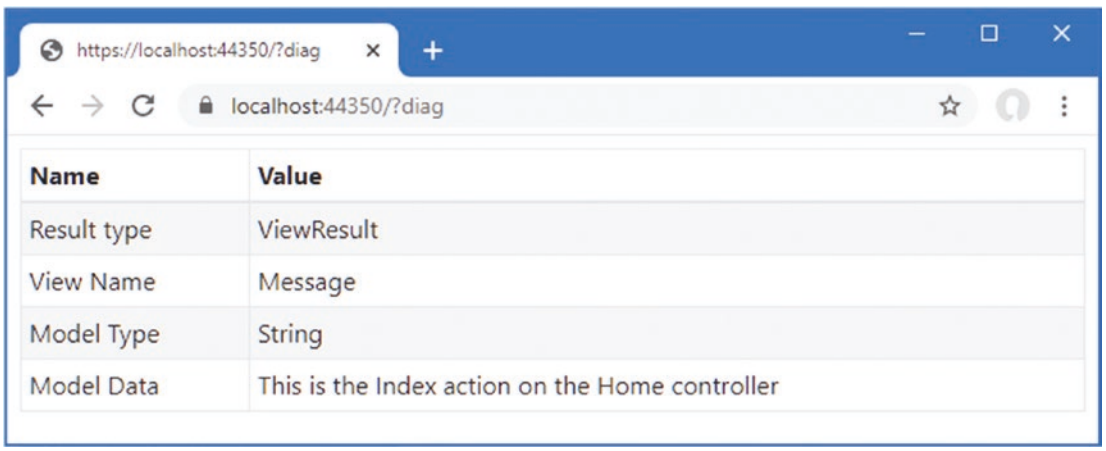

Figure 30-10. *Using a result filter*

Implementing a Result Filter Using the Attribute Base Class

The `ResultFilterAttribute` class is derived from `Attribute` and implements the `IResultFilter` and `IAsyncResultFilter` interfaces and can be used as the base class for result filters, as shown in Listing 30-29. There is no attribute base class for the always-run interfaces.

Listing 30-29. Using the Attribute Base Class in the ResultDiagnosticsAttribute.cs File in the Filters Folder

```
using Microsoft.AspNetCore.Mvc;
using Microsoft.AspNetCore.Mvc.Filters;
using Microsoft.AspNetCore.Mvc.ModelBinding;
using Microsoft.AspNetCore.Mvc.RazorPages;
using Microsoft.AspNetCore.Mvc.ViewFeatures;
using System;
using System.Collections.Generic;
using System.Threading.Tasks;

namespace WebApp.Filters {

    public class ResultDiagnosticsAttribute : ResultFilterAttribute {

        public override async Task OnResultExecutionAsync(
                ResultExecutingContext context, ResultExecutionDelegate next) {

            if (context.HttpContext.Request.Query.ContainsKey("diag")) {
                Dictionary<string, string> diagData =
                    new Dictionary<string, string> {
                        {"Result type", context.Result.GetType().Name }
                    };
                if (context.Result is ViewResult vr) {
                    diagData["View Name"] = vr.ViewName;
                    diagData["Model Type"] = vr.ViewData.Model.GetType().Name;
                    diagData["Model Data"] = vr.ViewData.Model.ToString();
                } else if (context.Result is PageResult pr) {
                    diagData["Model Type"] = pr.Model.GetType().Name;
                    diagData["Model Data"] = pr.ViewData.Model.ToString();
                }
                context.Result = new ViewResult() {
                    ViewName = "/Views/Shared/Message.cshtml",
                    ViewData = new ViewDataDictionary(
                                    new EmptyModelMetadataProvider(),
                                    new ModelStateDictionary()) {
                        Model = diagData
                    }
                };
            }
            await next();
        }
    }
}
```

Restart ASP.NET Core and request `https://localhost:44350/?diag`. The filter will produce the output shown in Figure 30-10.

Understanding Exception Filters

Exception filters allow you to respond to exceptions without having to write `try...catch` blocks in every action method. Exception filters can be applied to controller classes, action methods, page model classes, or handler methods. They are invoked when an exception is not handled by the endpoint or by the action, page, and result filters that have been applied to the endpoint. (Action, page, and result filters can deal with an unhandled exception by setting the `ExceptionHandled` property of their context objects to `true`.) Exception filters implement the `IExceptionFilter` interface, which is defined as follows:

```
namespace Microsoft.AspNetCore.Mvc.Filters {

    public interface IExceptionFilter : IFilterMetadata {

        void OnException(ExceptionContext context);
    }
}
```

The `OnException` method is called if an unhandled exception is encountered. The `IAsyncExceptionFilter` interface can be used to create asynchronous exception filters. Here is the definition of the asynchronous interface:

```
using System.Threading.Tasks;

namespace Microsoft.AspNetCore.Mvc.Filters {

    public interface IAsyncExceptionFilter : IFilterMetadata {

        Task OnExceptionAsync(ExceptionContext context);
    }
}
```

The `OnExceptionAsync` method is the asynchronous counterpart to the `OnException` method from the `IExceptionFilter` interface and is called when there is an unhandled exception. For both interfaces, context data is provided through the `ExceptionContext` class, which is derived from `FilterContext` and defines the additional properties shown in Table 30-15.

Table 30-15. *The ExceptionContext Properties*

Name	Description
`Exception`	This property contains any `Exception` that was thrown.
`ExceptionHandled`	This `bool` property is used to indicate if the exception has been handled.
`Result`	This property sets the `IActionResult` that will be used to generate the response.

Creating an Exception Filter

Exception filters can be created by implementing one of the filter interfaces or by deriving from the `ExceptionFilterAttribute` class, which is derived from `Attribute` and implements both the `IExceptionFilter` and `IAsyncException` filters. The most common use for an exception filter is to present a custom error page for a specific exception type in order to provide the user with more useful information than the standard error-handling capabilities can provide.

To create an exception filter, add a class file named `RangeExceptionAttribute.cs` to the `Filters` folder with the code shown in Listing 30-30.

Listing 30-30. The Contents of the RangeExceptionAttribute.cs File in the Filters Folder

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

```
namespace WebApp.Filters {
    public class RangeExceptionAttribute : ExceptionFilterAttribute {

        public override void OnException(ExceptionContext context) {
            if (context.Exception is ArgumentOutOfRangeException) {
                context.Result = new ViewResult() {
                    ViewName = "/Views/Shared/Message.cshtml",
                    ViewData = new ViewDataDictionary(
                        new EmptyModelMetadataProvider(),
                        new ModelStateDictionary()) {
                        Model = @"The data received by the
                                application cannot be processed"
                    }
                };
            }
        }
    }
}
```

This filter uses the `ExceptionContext` object to get the type of the unhandled exception and, if the type is `ArgumentOutOfRangeException`, creates an action result that displays a message to the user. Listing 30-31 adds an action method to the `Home` controller to which I have applied the exception filter.

Listing 30-31. Applying an Exception Filter in the HomeController.cs File in the Controllers Folder

```
using Microsoft.AspNetCore.Mvc;
using Microsoft.AspNetCore.Http;
using WebApp.Filters;
using Microsoft.AspNetCore.Mvc.Filters;
using System;

namespace WebApp.Controllers {

    [HttpsOnly]
    [ResultDiagnostics]
    public class HomeController : Controller {

        public IActionResult Index() {
            return View("Message",
                "This is the Index action on the Home controller");
        }

        public IActionResult Secure() {
            return View("Message",
                "This is the Secure action on the Home controller");
        }

        //[ChangeArg]
        public IActionResult Messages(string message1, string message2 = "None") {
            return View("Message", $"{message1}, {message2}");
        }

        public override void OnActionExecuting(ActionExecutingContext context) {
            if (context.ActionArguments.ContainsKey("message1")) {
                context.ActionArguments["message1"] = "New message";
            }
        }
```

```
        [RangeException]
        public ViewResult GenerateException(int? id) {
            if (id == null) {
                throw new ArgumentNullException(nameof(id));
            } else if (id > 10) {
                throw new ArgumentOutOfRangeException(nameof(id));
            } else {
                return View("Message", $"The value is {id}");
            }
        }
    }
}
```

The `GenerateException` action method relies on the default routing pattern to receive a nullable `int` value from the request URL. The action method throws an `ArgumentNullException` if there is no matching URL segment and throws an `ArgumentOutOfRangeException` if its value is greater than 50. If there is a value and it is in range, then the action method returns a `ViewResult`.

Restart ASP.NET Core and request `https://localhost:44350/Home/GenerateException/100`. The final segment will exceed the range expected by the action method, which will throw the exception type that is handled by the filter, producing the result shown in Figure 30-11. If you request `/Home/GenerateException`, then the exception thrown by the action method won't be handled by the filter, and the default error handling will be used.

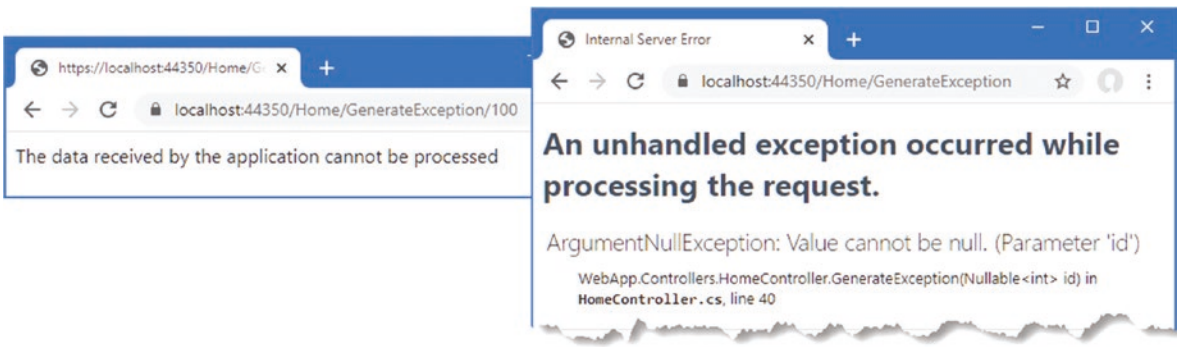

Figure 30-11. *Using an exception filter*

Managing the Filter Lifecycle

By default, ASP.NET Core manages the filter objects it creates and will reuse them for subsequent requests. This isn't always the desired behavior, and in the sections that follow, I describe different ways to take control of how filters are created. To create a filter that will show the lifecycle, add a class file called `GuidResponseAttribute.cs` to the `Filters` folder, and use it to define the filter shown in Listing 30-32.

Listing 30-32. The Contents of the GuidResponseAttribute.cs File in the Filters Folder

```
using Microsoft.AspNetCore.Mvc;
using Microsoft.AspNetCore.Mvc.Filters;
using Microsoft.AspNetCore.Mvc.ModelBinding;
using Microsoft.AspNetCore.Mvc.ViewFeatures;
using System;
using System.Collections.Generic;
using System.Threading.Tasks;
```

```
namespace WebApp.Filters {

    [AttributeUsage(AttributeTargets.Method | AttributeTargets.Class,
         AllowMultiple = true)]
    public class GuidResponseAttribute : Attribute, IAsyncAlwaysRunResultFilter {
        private int counter = 0;
        private string guid = Guid.NewGuid().ToString();

        public async Task OnResultExecutionAsync(ResultExecutingContext context,
            ResultExecutionDelegate next) {

            Dictionary<string, string> resultData;
            if (context.Result is ViewResult vr
                && vr.ViewData.Model is Dictionary<string, string> data) {
                    resultData = data;
            } else {
                resultData = new Dictionary<string, string>();
                context.Result = new ViewResult() {
                    ViewName = "/Views/Shared/Message.cshtml",
                    ViewData = new ViewDataDictionary(
                                    new EmptyModelMetadataProvider(),
                                    new ModelStateDictionary()) {
                        Model = resultData
                    }
                };
            }
            while (resultData.ContainsKey($"Counter_{counter}")) {
                counter++;
            }
            resultData[$"Counter_{counter}"] = guid;
            await next();
        }
    }
}
```

This result filter replaces the action result produced by the endpoint with one that will render the Message view and display a unique GUID value. The filter is configured so that it can be applied more than once to the same target and will add a new message if a filter earlier in the pipeline has created a suitable result. Listing 30-33 applies the filter twice to the Home controller. (I have also removed all but one of the action methods for brevity.)

Listing 30-33. Applying a Filter in the HomeController.cs File in the Controllers Folder

```
using Microsoft.AspNetCore.Mvc;
using Microsoft.AspNetCore.Http;
using WebApp.Filters;
using Microsoft.AspNetCore.Mvc.Filters;
using System;

namespace WebApp.Controllers {

    [HttpsOnly]
    [ResultDiagnostics]
    [GuidResponse]
    [GuidResponse]
    public class HomeController : Controller {
```

```
        public IActionResult Index() {
            return View("Message",
                "This is the Index action on the Home controller");
        }
    }
}
```

To confirm that the filter is being reused, restart ASP.NET Core and request `https://localhost:44350/?diag`. The response will contain GUID values from the two `GuidResponse` filter attributes. Two instances of the filter have been created to handle the request. Reload the browser, and you will see the same GUID values displayed, indicating that the filter objects created to handle the first request have been reused (Figure 30-12).

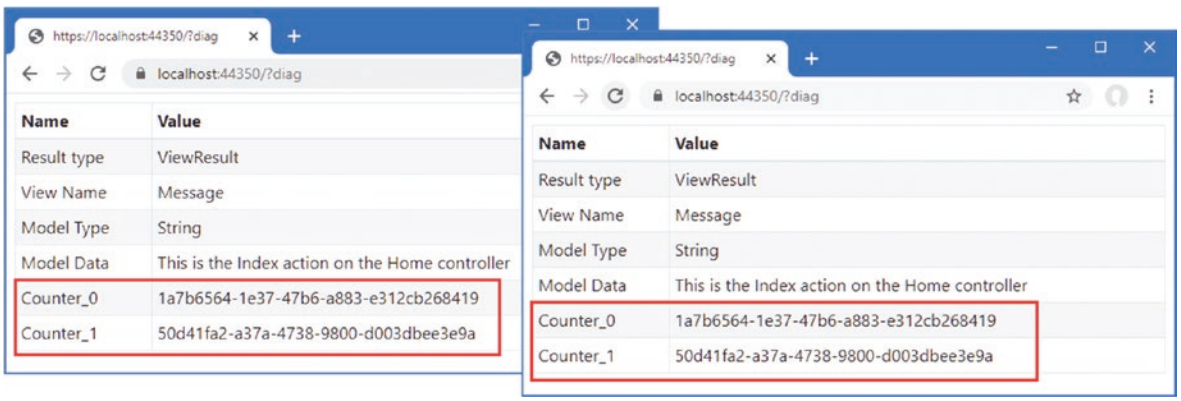

Figure 30-12. *Demonstrating filter reuse*

Creating Filter Factories

Filters can implement the `IFilterFactory` interface to take responsibility for creating instances of filters and specify whether those instances can be reused. The `IFilterFactory` interface defines the members described in Table 30-16.

Table 30-16. *The IFilterFactory Members*

Name	Description
`IsReusable`	This `bool` property indicates whether instances of the filter can be reused.
`CreateInstance(serviceProvider)`	This method is invoked to create new instances of the filter and is provided with an `IServiceProvider` object.

Listing 30-34 implements the `IFilterFactory` interface and returns `false` for the `IsReusable` property, which prevents the filter from being reused.

Listing 30-34. Implementing an Interface in the GuidResponseAttribute.cs File in the Filters Folder

```
using Microsoft.AspNetCore.Mvc;
using Microsoft.AspNetCore.Mvc.Filters;
using Microsoft.AspNetCore.Mvc.ModelBinding;
using Microsoft.AspNetCore.Mvc.ViewFeatures;
using System;
using System.Collections.Generic;
using System.Threading.Tasks;
using Microsoft.Extensions.DependencyInjection;
```

```
namespace WebApp.Filters {

    [AttributeUsage(AttributeTargets.Method | AttributeTargets.Class,
         AllowMultiple = true)]
    public class GuidResponseAttribute : Attribute,
            IAsyncAlwaysRunResultFilter, IFilterFactory {
        private int counter = 0;
        private string guid = Guid.NewGuid().ToString();

        public bool IsReusable => false;

        public IFilterMetadata CreateInstance(IServiceProvider serviceProvider) {
            return ActivatorUtilities
                .GetServiceOrCreateInstance<GuidResponseAttribute>(serviceProvider);
        }

        public async Task OnResultExecutionAsync(ResultExecutingContext context,
            ResultExecutionDelegate next) {

            Dictionary<string, string> resultData;
            if (context.Result is ViewResult vr
                && vr.ViewData.Model is Dictionary<string, string> data) {
                    resultData = data;
            } else {
                resultData = new Dictionary<string, string>();
                context.Result = new ViewResult() {
                    ViewName = "/Views/Shared/Message.cshtml",
                    ViewData = new ViewDataDictionary(
                                        new EmptyModelMetadataProvider(),
                                        new ModelStateDictionary()) {
                        Model = resultData
                    }
                };
            }
            while (resultData.ContainsKey($"Counter_{counter}")) {
                counter++;
            }
            resultData[$"Counter_{counter}"] = guid;
            await next();
        }
    }
}
```

I create new filter objects using the `GetServiceOrCreateInstance` method, defined by the `ActivatorUtilities` class in the `Microsoft.Extensions.DependencyInjection` namespace. Although you can use the `new` keyword to create a filter, this approach will resolve any dependencies on services that are declared through the filter's constructor.

To see the effect of implementing the `IFilterFactory` interface, restart ASP.NET Core and request `https://localhost:44350/?diag`. Reload the browser, and each time the request is handled, new filters will be created, and new GUIDs will be displayed, as shown in Figure 30-13.

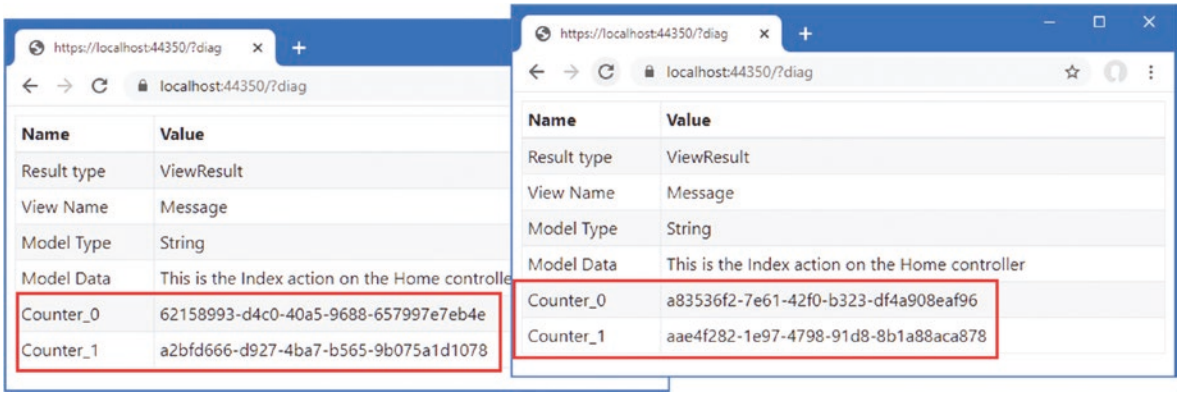

Figure 30-13. *Preventing filter reuse*

Using Dependency Injection Scopes to Manage Filter Lifecycles

Filters can be registered as services, which allows their lifecycle to be controlled through dependency injection, which I described in Chapter 14. Listing 30-35 registers the `GuidResponse` filter as a scoped service.

Listing 30-35. Creating a Filter Service in the Startup.cs File in the WebApp Folder

```
using Microsoft.AspNetCore.Builder;
using Microsoft.Extensions.DependencyInjection;
using Microsoft.Extensions.Configuration;
using Microsoft.EntityFrameworkCore;
using WebApp.Models;
using Microsoft.AspNetCore.Antiforgery;
using Microsoft.AspNetCore.Http;
using Microsoft.AspNetCore.Mvc;
using WebApp.Filters;

namespace WebApp {
    public class Startup {

        public Startup(IConfiguration config) {
            Configuration = config;
        }

        public IConfiguration Configuration { get; set; }

        public void ConfigureServices(IServiceCollection services) {
            services.AddDbContext<DataContext>(opts => {
                opts.UseSqlServer(Configuration[
                    "ConnectionStrings:ProductConnection"]);
                opts.EnableSensitiveDataLogging(true);
            });
            services.AddControllersWithViews().AddRazorRuntimeCompilation();
            services.AddRazorPages().AddRazorRuntimeCompilation();
            services.AddSingleton<CitiesData>();

            services.Configure<AntiforgeryOptions>(opts => {
                opts.HeaderName = "X-XSRF-TOKEN";
            });
```

```
            services.Configure<MvcOptions>(opts => opts.ModelBindingMessageProvider
                .SetValueMustNotBeNullAccessor(value => "Please enter a value"));

            services.AddScoped<GuidResponseAttribute>();
        }

        public void Configure(IApplicationBuilder app, DataContext context,
                IAntiforgery antiforgery) {

            // ...statements omitted for brevity...
        }
    }
}
```

By default, ASP.NET Core creates a scope for each request, which means that a single instance of the filter will be created for each request. To see the effect, restart ASP.NET Core and request `https://localhost:44350/?diag`. Both attributes applied to the `Home` controller are processed using the same instance of the filter, which means that both GUIDs in the response are the same. Reload the browser; a new scope will be created, and a new filter object will be used, as shown in Figure 30-14.

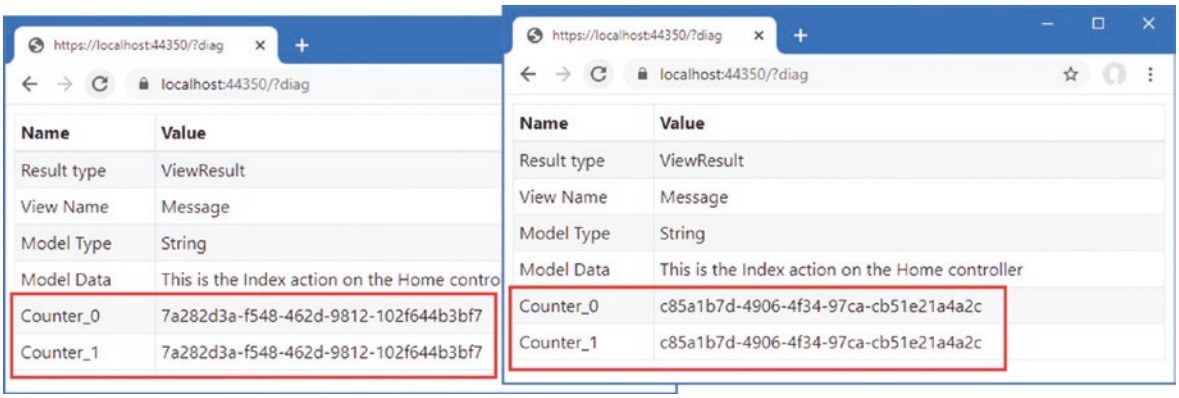

Figure 30-14. *Using dependency injection to manage filters*

USING FILTERS AS SERVICES WITHOUT THE IFILTERFACTORY INTERFACE

The change in lifecycle took effect immediately in this example because I used the `ActivatorUtilities.GetServiceOrCreateInstance` method to create the filter object when I implemented the `IFilterFactory` interface. This method will check to see whether there is a service available for the requested type before invoking its constructor. If you want to use filters as services without implementing `IFilterFactory` and using `ActivatorUtilities`, you can apply the filter using the `ServiceFilter` attribute, like this:

```
...
[ServiceFilter(typeof(GuidResponseAttribute))]
...
```

ASP.NET Core will create the filter object from the service and apply it to the request. Filters that are applied in this way do not have to be derived from the `Attribute` class.

Creating Global Filters

Global filters are applied to every request that ASP.NET Core handles, which means they don't have to be applied to individual controllers or Razor Pages. Any filter can be used as a global filter; however, action filters will be applied to requests only where the endpoint is an action method, and page filters will be applied to requests only where the endpoint is a Razor Page.

Global filters are set up using the options pattern in the `Startup` class, as shown in Listing 30-36.

Listing 30-36. Creating a Global Filter in the Startup.cs File in the WebApp Folder

```
...
public void ConfigureServices(IServiceCollection services) {
    services.AddDbContext<DataContext>(opts => {
        opts.UseSqlServer(Configuration[
            "ConnectionStrings:ProductConnection"]);
        opts.EnableSensitiveDataLogging(true);
    });
    services.AddControllersWithViews().AddRazorRuntimeCompilation();
    services.AddRazorPages().AddRazorRuntimeCompilation();
    services.AddSingleton<CitiesData>();

    services.Configure<AntiforgeryOptions>(opts => {
        opts.HeaderName = "X-XSRF-TOKEN";
    });

    services.Configure<MvcOptions>(opts => opts.ModelBindingMessageProvider
        .SetValueMustNotBeNullAccessor(value => "Please enter a value"));

    services.AddScoped<GuidResponseAttribute>();
    services.Configure<MvcOptions>(opts => opts.Filters.Add<HttpsOnlyAttribute>());
}
...
```

The `MvcOptions.Filters` property returns a collection to which filters are added to apply them globally, either using the `Add<T>` method or using the `AddService<T>` method for filters that are also services. There is also an `Add` method without a generic type argument that can be used to register a specific object as a global filter.

The statement in Listing 30-36 registers the `HttpsOnly` filter I created earlier in the chapter, which means that it no longer needs to be applied directly to individual controllers or Razor Pages, so Listing 30-37 removes the filter from the `Home` controller.

■ **Note** Notice that I have disabled the `GuidResponse` filter in Listing 30-37. This is an always-run result filter and will replace the result generated by the global filter.

Listing 30-37. Removing a Filter in the HomeController.cs File in the Controllers Folder

```
using Microsoft.AspNetCore.Mvc;
using Microsoft.AspNetCore.Http;
using WebApp.Filters;
using Microsoft.AspNetCore.Mvc.Filters;
using System;

namespace WebApp.Controllers {

    //[HttpsOnly]
    [ResultDiagnostics]
    //[GuidResponse]
    //[GuidResponse]
```

```
    public class HomeController : Controller {

        public IActionResult Index() {
            return View("Message",
                "This is the Index action on the Home controller");
        }
    }
}
```

Restart ASP.NET Core and request `http://localhost:5000` to confirm that the HTTPS-only policy is being applied even though the attribute is no longer used to decorate the controller. The global authorization filter will short-circuit the filter pipeline and produce the response shown in Figure 30-15.

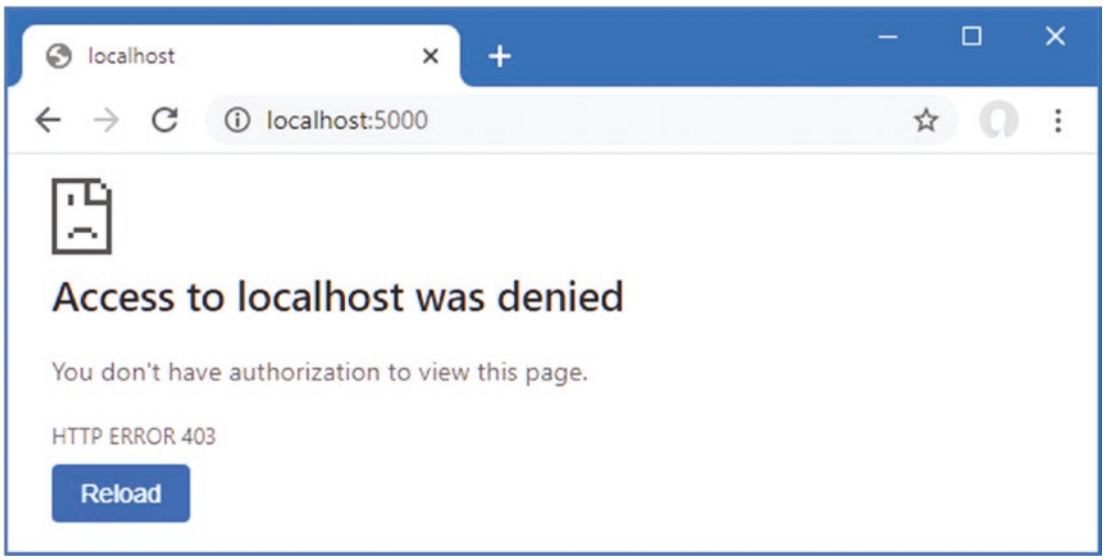

Figure 30-15. *Using a global filter*

Understanding and Changing Filter Order

Filters run in a specific sequence: authorization, resource, action, or page, and then result. But if there are multiple filters of a given type, then the order in which they are applied is driven by the scope through which the filters have been applied.

To demonstrate how this works, add a class file named `MessageAttribute.cs` to the `Filters` folder and use it to define the filter shown in Listing 30-38.

Listing 30-38. The Contents of the MessageAttribute.cs File in the Filters Folder

```
using Microsoft.AspNetCore.Mvc;
using Microsoft.AspNetCore.Mvc.Filters;
using Microsoft.AspNetCore.Mvc.ModelBinding;
using Microsoft.AspNetCore.Mvc.ViewFeatures;
using System;
using System.Collections.Generic;
using System.Threading.Tasks;
```

```
namespace WebApp.Filters {

    [AttributeUsage(AttributeTargets.Method | AttributeTargets.Class,
        AllowMultiple = true)]
    public class MessageAttribute : Attribute, IAsyncAlwaysRunResultFilter {
        private int counter = 0;
        private string msg;

        public MessageAttribute(string message) => msg = message;

        public async Task OnResultExecutionAsync(ResultExecutingContext context,
               ResultExecutionDelegate next) {
            Dictionary<string, string> resultData;
            if (context.Result is ViewResult vr
                && vr.ViewData.Model is Dictionary<string, string> data) {
                resultData = data;
            } else {
                resultData = new Dictionary<string, string>();
                context.Result = new ViewResult() {
                    ViewName = "/Views/Shared/Message.cshtml",
                    ViewData = new ViewDataDictionary(
                                    new EmptyModelMetadataProvider(),
                                    new ModelStateDictionary()) {
                        Model = resultData
                    }
                };
            }
            while (resultData.ContainsKey($"Message_{counter}")) {
                counter++;
            }
            resultData[$"Message_{counter}"] = msg;
            await next();
        }
    }
}
```

This result filter uses techniques shown in earlier examples to replace the result from the endpoint and allows multiple filters to build up a series of messages that will be displayed to the user. Listing 30-39 applies several instances of the `Message` filter to the `Home` controller.

Listing 30-39. Applying a Filter in the HomeController.cs File in the Controllers Folder

```
using Microsoft.AspNetCore.Mvc;
using Microsoft.AspNetCore.Http;
using WebApp.Filters;
using Microsoft.AspNetCore.Mvc.Filters;
using System;

namespace WebApp.Controllers {

    [Message("This is the controller-scoped filter")]
    public class HomeController : Controller {

        [Message("This is the first action-scoped filter")]
        [Message("This is the second action-scoped filter")]
        public IActionResult Index() {
            return View("Message",
```

```
                "This is the Index action on the Home controller");
        }
    }
}
```

Listing 30-40 registers the Message filter globally.

Listing 30-40. Creating a Global Filter in the Startup.cs File in the WebApp Folder

```
...
public void ConfigureServices(IServiceCollection services) {
    services.AddDbContext<DataContext>(opts => {
        opts.UseSqlServer(Configuration[
            "ConnectionStrings:ProductConnection"]);
        opts.EnableSensitiveDataLogging(true);
    });
    services.AddControllersWithViews().AddRazorRuntimeCompilation();
    services.AddRazorPages().AddRazorRuntimeCompilation();
    services.AddSingleton<CitiesData>();

    services.Configure<AntiforgeryOptions>(opts => {
        opts.HeaderName = "X-XSRF-TOKEN";
    });

    services.Configure<MvcOptions>(opts => opts.ModelBindingMessageProvider
        .SetValueMustNotBeNullAccessor(value => "Please enter a value"));

    services.AddScoped<GuidResponseAttribute>();
    services.Configure<MvcOptions>(opts => {
        opts.Filters.Add<HttpsOnlyAttribute>();
        opts.Filters.Add(new MessageAttribute("This is the globally-scoped filter"));
    });
}
...
```

There are four instances of the same filter. To see the order in which they are applied, restart ASP.NET Core and request https://localhost:44350, which will produce the response shown in Figure 30-16.

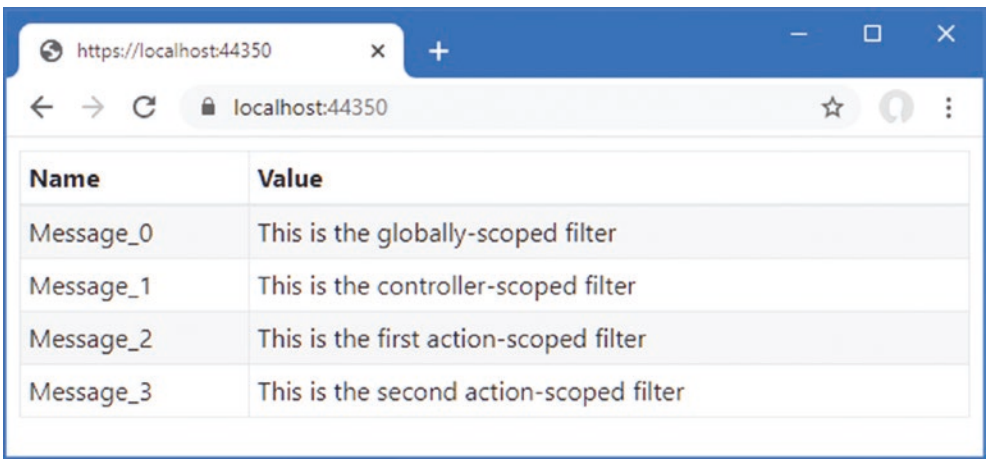

Figure 30-16. *Applying the same filter in different scopes*

By default, ASP.NET Core runs global filters, then filters applied to controllers or page model classes, and finally filters applied to action or handler methods.

Changing Filter Order

The default order can be changed by implementing the `IOrderedFilter` interface, which ASP.NET Core looks for when it is working out how to sequence filters. Here is the definition of the interface:

```
namespace Microsoft.AspNetCore.Mvc.Filters {

    public interface IOrderedFilter : IFilterMetadata {
        int Order { get; }
    }
}
```

The `Order` property returns an `int` value, and filters with low values are applied before those with higher `Order` values. In Listing 30-41, I have implemented the interface in the `Message` filter and defined a constructor argument that will allow the value for the `Order` property to be specified when the filter is applied.

Listing 30-41. Adding Ordering Support in the MessageAttribute.cs File in the Filters Folder

```
using Microsoft.AspNetCore.Mvc;
using Microsoft.AspNetCore.Mvc.Filters;
using Microsoft.AspNetCore.Mvc.ModelBinding;
using Microsoft.AspNetCore.Mvc.ViewFeatures;
using System;
using System.Collections.Generic;
using System.Threading.Tasks;

namespace WebApp.Filters {

    [AttributeUsage(AttributeTargets.Method | AttributeTargets.Class,
        AllowMultiple = true)]
    public class MessageAttribute : Attribute, IAsyncAlwaysRunResultFilter,
            IOrderedFilter {
        private int counter = 0;
        private string msg;

        public MessageAttribute(string message) => msg = message;

        public int Order { get; set; }

        public async Task OnResultExecutionAsync(ResultExecutingContext context,
               ResultExecutionDelegate next) {

            // ...statements omitted for brevity...
        }
    }
}
```

In Listing 30-42, I have used the constructor argument to change the order in which the filters are applied.

Listing 30-42. Setting Filter Order in the HomeController.cs File in the Controllers Folder

```
using Microsoft.AspNetCore.Mvc;
using Microsoft.AspNetCore.Http;
using WebApp.Filters;
```

```
using Microsoft.AspNetCore.Mvc.Filters;
using System;

namespace WebApp.Controllers {

    [Message("This is the controller-scoped filter", Order = 10)]
    public class HomeController : Controller {

        [Message("This is the first action-scoped filter", Order = 1)]
        [Message("This is the second action-scoped filter", Order = -1)]
        public IActionResult Index() {
            return View("Message",
                "This is the Index action on the Home controller");
        }
    }
}
```

`Order` values can be negative, which is a helpful way of ensuring that a filter is applied before any global filters with the default order (although you can also set the order when creating global filters, too). Restart ASP.NET Core and request `https://localhost:44350` to see the new filter order, which is shown in Figure 30-17.

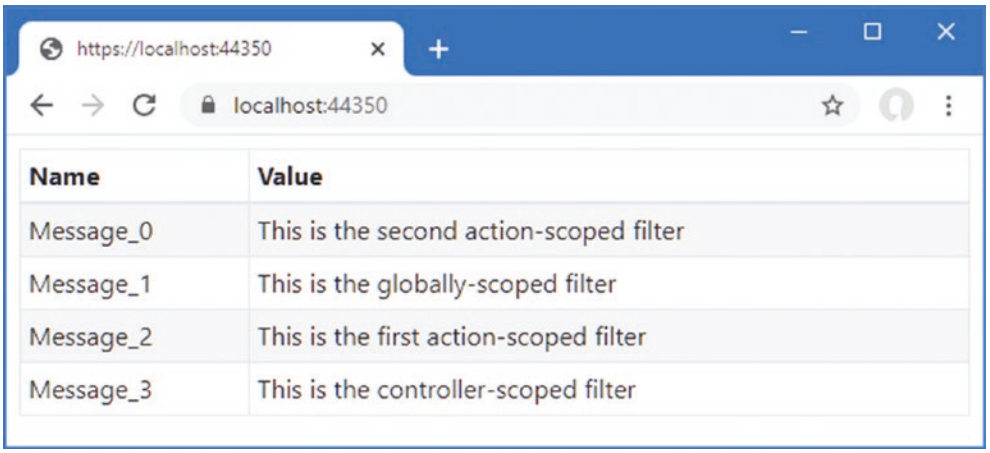

Figure 30-17. *Changing filter order*

Summary

In this chapter, I described the ASP.NET Core filter feature and explained how it can be used to alter requests and results for specific endpoints. I described the different types of filters and demonstrated how to create and apply each of them. I also showed you how to manage the lifecycle of filters and control the order in which they are executed. In the next chapter, I show you how to combine the features described in this part of the book to create form applications.