CHAPTER 18

Filters

Filters inject extra logic into MVC Framework request processing. They provide a simple and elegant way to implement *cross-cutting concerns*. This term refers to functionality that is used all over an application and doesn't fit neatly into any one place, where it would break the separation of concerns pattern. Classic examples of cross-cutting concerns are logging, authorization, and caching. In this chapter, I show you the different categories of filters that the MVC Framework supports, how to create and use custom filters, and how to control their execution. Table 18-1 provides the summary for this chapter.

Table 18-1. *Chapter Summary*

Problem	Solution	Listing
Inject extra logic into request processing	Apply filters to the controller or its action methods	1–8
Restrict action methods to specific users and groups	Use authorization filters	9–12
Authenticate requests	Use authentication filters	13–19
Process errors when executing requests	Use exception filters	20–30
Inject general-purpose logic into the request handling process	Use action filters	31–35
Inspect or alter the results generated by action methods	Use result filters	36–41
Use filters without attributes	Use the built-in controller methods	42
Define filters that apply to all action methods in the application	Define global filters	43–46
Control the order in which filters are executed	Use the `Order` parameter	47–49
Override global and controller filters for an action method	Use an override filter	50–54

Preparing the Example Project

For this chapter I created a new MVC project called `Filters` using the `Empty` template, checking the option to add the core MVC folders and references. I created a `Home` controller which has the action method shown in Listing 18-1. I am only focused on controllers in this chapter, so I return string values from the action methods, rather than `ActionResult` objects. This has the effect of causing the MVC Framework to send the string value directly to the browser, by passing the Razor view engine.

Listing 18-1. The Contents of the HomeController.cs File

```
using System.Web.Mvc;

namespace Filters.Controllers {
    public class HomeController : Controller {

        public string Index() {
            return "This is the Index action on the Home controller";
        }
    }
}
```

Later in the chapter, I'll be demonstrating how you can use a new MVC feature called *authentication filters*, and for that I need to be able to perform some simple user authentication. As I explained in Chapter 12, I don't cover the security features of the underling ASP.NET platform in this book, but Apress will freely distribute the security chapters from my forthcoming *Pro ASP.NET MVC 5 Platform* book when it is published in 2014. So, in order to demonstrate the authentication filters feature, which is part of the MVC Framework, I am going to use the same approach I adopted in Chapter 12 and define static user credentials in the `Web.config` file, as shown in Listing 18-2.

Listing 18-2. Defining User Credentials in the Web.config File

```
...
<system.web>
  <compilation debug="true" targetFramework="4.5.1" />
  <httpRuntime targetFramework="4.5.1" />
  <authentication mode="Forms">
    <forms loginUrl="~/Account/Login" timeout="2880">
      <credentials passwordFormat="Clear">
        <user name="user"  password="secret"/>
        <user name="admin" password="secret" />
      </credentials>
    </forms>
  </authentication>
</system.web>
...
```

I have defined two users, `user` and `admin`, and assigned them the same password, `secret`, to keep things simple. I am using `forms` authentication once again and I have used the `loginUrl` attribute to specify that unauthenticated requests should be redirected to the `/Account/Login` URL. In Listing 18-3, you can see the contents of the `Account` controller I added to the project and whose `Login` action will be targeted by the default routing configuration.

Listing 18-3. The Contents of the AccountController.cs File

```
using System.Web.Mvc;
using System.Web.Security;

namespace Filters.Controllers {
    public class AccountController : Controller {

        public ActionResult Login() {
            return View();
        }
```

```
        [HttpPost]
        public ActionResult Login(string username, string password, string returnUrl) {
            bool result = FormsAuthentication.Authenticate(username, password);
            if (result) {
                FormsAuthentication.SetAuthCookie(username, false);
                return Redirect(returnUrl ?? Url.Action("Index", "Admin"));
            } else {
                ModelState.AddModelError("", "Incorrect username or password");
                return View();
            }
        }
    }
}
```

To create the view that will gather candidate credentials from the user, create a `Views/Shared` folder and right-click it. Select `Add ➤ MVC 5 View Page (Razor)`, set the name `Login.cshtml` and click the `OK` button to create the view. Edit the new view to match the content shown in Listing 18-4.

■ **Note** I am creating a shared view because I'll be adding a second authentication controller later in the chapter and I want to reuse the view.

Listing 18-4. The Contents of the Login.cshtml File

```
@{
    Layout = null;
}

<!DOCTYPE html>

<html>
<head>
    <meta name="viewport" content="width=device-width" />
    <title></title>
</head>
<body>
    @using (Html.BeginForm()) {
        @Html.ValidationSummary()
        <p><label>Username:</label><input name="username" /></p>
        <p><label>Password:</label><input name="password" type="password"/></p>
        <input type="submit" value="Log in" />
    }
</body>
</html>
```

Setting the Start URL and Testing the Application

As with all of the example projects, I want Visual Studio to start with the root URL for the application rather than guess the URL based on the file that is being edited. Select `Filters Properties` from the Visual Studio `Project` menu, switch to the `Web` tab and check the `Specific Page` option in the `Start Action` section. You don't have to provide a value. Just checking the option is enough. If you start the example app, you will get the response shown in Figure 18-1.

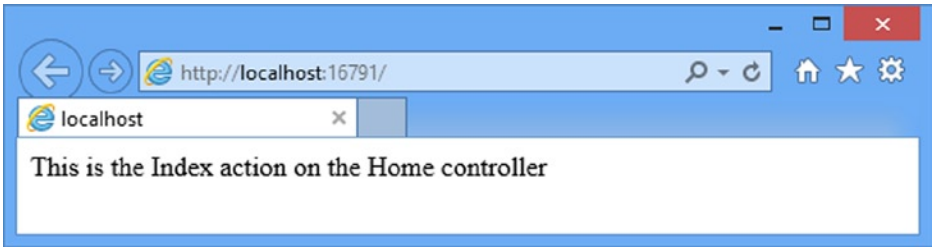

Figure 18-1. *Running the example application*

Using Filters

You have already seen an example of a filter in Chapter 12, when I applied authorization to the action methods of the SportsStore administration controller. I wanted the action method to be used only by users who had authenticated themselves, which presented me with a choice of approaches. I could have checked the authorization status of the request in each and every action method, as shown in Listing 18-5.

Listing 18-5. Explicitly Checking Authorization in Action Methods

```
namespace SportsStore.WebUI.Controllers {

    public class AdminController : Controller {

        // ... instance variables and constructor

        public ViewResult Index() {
            if (!Request.IsAuthenticated) {
                FormsAuthentication.RedirectToLoginPage();
            }
            // ...rest of action method
        }

        public ViewResult Create() {
            if (!Request.IsAuthenticated) {
                FormsAuthentication.RedirectToLoginPage();
            }
            // ...rest of action method
        }
```

```
        public ViewResult Edit(int productId) {
            if (!Request.IsAuthenticated) {
                FormsAuthentication.RedirectToLoginPage();
            }
            // ...rest of action method
        }

        // ... other action methods
    }
}
```

You can see that there is a lot of repetition in this approach, which is why I decided to use a filter instead, as shown in Listing 18-6.

Listing 18-6. Applying a Filter

```
namespace SportsStore.WebUI.Controllers {

    [Authorize]
    public class AdminController : Controller {

        // ... instance variables and constructor

        public ViewResult Index() {
            // ...rest of action method
        }

        public ViewResult Create() {
            // ...rest of action method
        }

        public ViewResult Edit(int productId) {
            // ...rest of action method
        }

        // ... other action methods
    }
}
```

Filters are .NET attributes that add extra steps to the request processing pipeline. I used the `Authorize` filter in Listing 18-6, which has the same effect as all of the duplicated checks in Listing 18-5.

Introducing the Filter Types

The MVC Framework supports five different types of filters. Each allows you to introduce logic at different points during request processing. The filter types are described in Table 18-2.

Table 18-2. *MVC Framework Filter Types*

Filter Type	Interface	Default Implementation	Description
Authentication	`IAuthenticationFilter`	`N/A`	Runs first, before any other filters or the action method, but can be run again after the authorization filters
Authorization	`IAuthorizationFilter`	`AuthorizeAttribute`	Runs second, after authentication, but before any other filters or the action method
Action	`IActionFilter`	`ActionFilterAttribute`	Runs before and after the action method
Result	`IResultFilter`	`ActionFilterAttribute`	Runs before and after the action result is executed
Exception	`IExceptionFilter`	`HandleErrorAttribute`	Runs only if another filter, the action method, or the action result throws an exception

Before the MVC Framework invokes an action, it inspects the method definition to see if it has attributes that implement the interfaces listed in Table 18-2. If so, then at the appropriate point in the request handling process, the methods defined by these interfaces are invoked. The framework includes default attribute classes that implement the filter interfaces. I will show you how to use these default classes later in this chapter.

■ **Tip** MVC 5 introduces a new interface, `IoverrideFilter`, which I describe in the Overriding Filters section later in the chapter.

The `ActionFilterAttribute` class implements both the `IActionFilter` and `IResultFilter` interfaces. This class is abstract, which forces you to provide an implementation. The `AuthorizeAttribute` and `HandleErrorAttribute` classes contain useful features and can be used without creating a derived class.

Applying Filters to Controllers and Action Methods

Filters can be applied to individual action methods or to an entire controller. In Listing 18-6, I applied the `Authorize` filter to the `AdminController` class, which has the same effect as applying it to each action method in the controller, as shown in Listing 18-7.

Listing 18-7. Applying a Filter to Action Methods Individually

```
namespace SportsStore.WebUI.Controllers {

    public class AdminController : Controller {

        // ... instance variables and constructor
        [Authorize]
        public ViewResult Index() {
            // ...rest of action method
        }
```

```
        [Authorize]
        public ViewResult Create() {
            // ...rest of action method
        }

        // ... other action methods
    }
}
```

You can apply multiple filters, and mix and match the levels at which they are applied, that is, whether they are applied to the controller or an individual action method. Listing 18-8 shows three different filters in use.

Listing 18-8. Applying Multiple Filters in a Controller Class

```
[Authorize(Roles="trader")] // applies to all actions
public class ExampleController : Controller {

    [ShowMessage]                   // applies to just this action
    [OutputCache(Duration=60)]      // applies to just this action
    public ActionResult Index() {
        // ... action method body
    }
}
```

Some of the filters in this listing take parameters. I will show you how these work as I describe the different kinds of filters.

■ **Note** If you have defined a custom base class for your controllers, any filters applied to the base class will affect the derived classes.

Using Authorization Filters

Authorization filters are run after the authentication filters, before action filters and before the action method is invoked. As the name suggests, these filters enforce your authorization policy, ensuring that action methods can be invoked only by approved users.

There is a somewhat involved relationship between the authentication and authorization filters which is easier to explain once you understand how authorization filters work. I explain this relationship in the *Using Authentication Filters* section later in the chapter. Authorization filters implement the `IAuthorizationFilter` interface, which is shown in Listing 18-9.

Listing 18-9. The IAuthorizationFilter Interface

```
namespace System.Web.Mvc {

    public interface IAuthorizationFilter {
        void OnAuthorization(AuthorizationContext filterContext);
    }
}
```

You can, if you are so minded, create a class that implements the `IAuthorizationFilter` interface and create your own security logic. See the sidebar on why this is a really bad idea.

WARNING: WRITING SECURITY CODE IS DANGEROUS

Programming history is littered with the wreckage of applications whose programmers thought they knew how to write good security code. That's actually a skill that few people possess. There is usually some forgotten wrinkle or untested corner case that leaves a gaping hole in the application's security. If you do not believe me, just Google the term `security bug` and start reading through the top results.

Wherever possible, I like to use security code that is widely tested and proven. In this case, the MVC Framework has provided a full-featured authorization filter, which can be derived to implement custom authorization policies. I try to use this whenever I can, and I recommend that you do the same. At the least, you can pass some of the blame to Microsoft when your secret application data is spread far and wide on the Internet.

A much safer approach is to create a subclass of the `AuthorizeAttribute` class which takes care of all of the tricky stuff and makes it easy to write custom authorization code. The best way to demonstrate this is to create a custom filter and, to this end, I have added an `Infrastructure` folder to the example project and created a new class file within it called `CustomAuthAttribute.cs`. You can see the content of this file in Listing 18-10.

Listing 18-10. The Contents of the CustomAuthAttribute.cs File

```
using System.Web;
using System.Web.Mvc;

namespace Filters.Infrastructure {
    public class CustomAuthAttribute : AuthorizeAttribute {
        private bool localAllowed;

        public CustomAuthAttribute(bool allowedParam) {
            localAllowed = allowedParam;
        }

        protected override bool AuthorizeCore(HttpContextBase httpContext) {
            if (httpContext.Request.IsLocal) {
                return localAllowed;
            } else {
                return true;
            }
        }
    }
}
```

This is a simple authorization filter. It allows you to prevent access to local requests (a local request is one where the browser and the application server are running on the same device, such as your development PC).

I have used the simplest approach to creating an authorization filter, which is to subclass the `AuthorizeAttribute` class and then override the `AuthorizeCore` method. This ensures that I benefit from the features built in to `AuthorizeAttribute`. The constructor for the filter takes a `bool` value, indicating whether local requests are permitted.

The interesting part of the filter class is the implementation of the AuthorizeCore method, which is how the MVC Framework checks to see if the filter will authorize access for a request. The argument to this method is an HttpContextBase object, through which I can get information about the request being processed. By taking advantage of the built-in features of the AuthorizeAttribute base class, I only have to focus on the authorization logic and return true from the AuthorizeCore method if I want to authorize a request and return false if I do not.

KEEPING AUTHORIZATION ATTRIBUTES SIMPLE

The AuthorizeCore method is passed an HttpContextBase object, which provides access to information about the request, but not about the controller or action method that the authorization attribute has been applied to. The main reason that developers implement the IAuthorizationFilter interface directly is to get access to the AuthorizationContext passed to the OnAuthorization method, through which a much wider range of information can be obtained, including routing details and the current controller and action method.

I do not recommend this approach, and not just because I think writing your own security code is dangerous. Although authorization is a cross-cutting concern, building logic into your authorization attributes which is tightly coupled to the structure of your controllers undermines the separation of concerns and causes testing and maintenance problems. Keep your authorization attributes simple and focused on authorization based on the request. Let the context of what is being authorized come from where the attribute is applied.

Applying the Custom Authorization Filter

To use the custom authorization filter, I simply apply an attribute to the action methods or controllers that I want to protect, as illustrated by Listing 18-11, which demonstrates the application of the filter to the Index action method in the Home controller.

Listing 18-11. Applying a Custom Authorization Filter in the HomeController.cs File

```
using System.Web.Mvc;
using Filters.Infrastructure;

namespace Filters.Controllers {
    public class HomeController : Controller {

        [CustomAuth(false)]
        public string Index() {
            return "This is the Index action on the Home controller";
        }
    }
}
```

I have set the constructor argument for the filter to false, which means that local requests will be denied access to the Index action method. You can test this by starting the application. The routing configuration will target the Index action method when the root URL is requested by the browser. If the browser making the request is on the machine running Visual Studio, then you will see the result in Figure 18-2. The filter has denied authorization for the request and the MVC Framework has responded in the only way it knows how: by prompting the user for credentials. Of course, a username or password won't change the fact that the request is coming from the local machine and there is nothing you can do at this point to get past the authentication challenge.

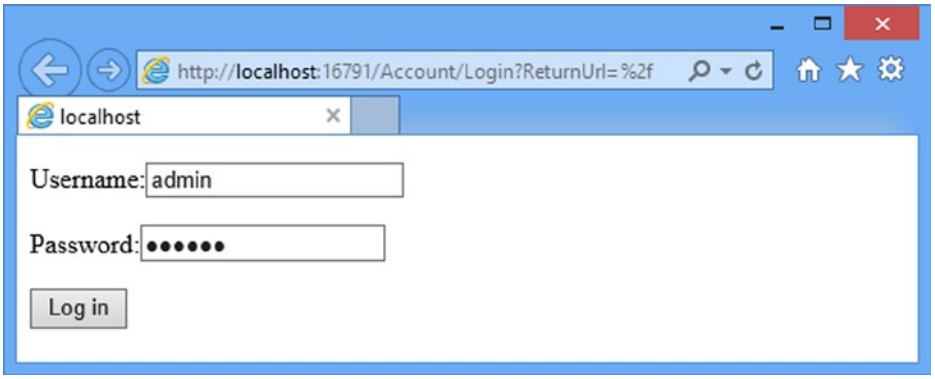

Figure 18-2. *Re-prompting for credentials for a local request by the custom authorization filter*

However, the filter will authorize the request if you change the constructor argument for the filter to `true` and restart the application. (You can't test by making a request from another machine because IIS Express, which runs the application, is configured to reject any connections that are not local.)

Using the Built-in Authorization Filter

Although I used the `AuthorizeAttribute` class as the base for the custom filter, it has its own implementation of the `AuthorizeCore` method which is useful for performing general purpose authorization tasks. When using the `AuthorizeAttribute` directly, I can specify an authorization policy using two public properties of this class, as shown in Table 18-3.

Table 18-3. *AuthorizeAttribute Properties*

Name	Type	Description
`Users`	`string`	A comma-separated list of usernames that are allowed to access the action method.
`Roles`	`string`	A comma-separated list of role names. To access the action method, users must be in at least one of these roles.

Listing 18-12 shows how I can use the built-in filter to protect an action method with one of these properties.

Listing 18-12. Using the Built-in Authorization Filter

```
using System.Web.Mvc;
using Filters.Infrastructure;

namespace Filters.Controllers {
    public class HomeController : Controller {

        [Authorize(Users = "admin")]
        public string Index() {
            return "This is the Index action on the Home controller";
        }
    }
}
```

I have specified that the admin user is authorized to invoke the Index action method, but there is an implicit condition as well: the request is authenticated. If I do not specify any users or roles, then any authenticated user can use the action method. For most applications, the authorization policy that AuthorizeAttribute provides is sufficient. If you want to implement something special, you can derive from this class as just I did earlier in the chapter or supplement your configuration with authentication filters, which I describe the next section.

Using Authentication Filters

Authentication filters are new in MVC version 5 and provide the means to provide fine-grain control over how users are authenticated for controllers and actions in an application.

Authentication filters have a relatively complex lifecycle. They are run before any other filter, which lets you define an authentication policy that will be applied before any other type of filter is used. Authentication filters can also be combined with authorization filters to provide authentication challenges for requests that don't comply to the authorization policy. Authentication filters are also run after an action method has been executed but before the ActionResult is processed. I explain how all of this works and provide some examples along the way.

Understanding the IAuthenticationFilter Interface

Authentication filters implement the IAuthenticationFilter interface, which is shown in Listing 18-13.

Listing 18-13. The IAuthenticationFilter Interface

```
namespace System.Web.Mvc.Filters {

    public interface IAuthenticationFilter {
        void OnAuthentication(AuthenticationContext context);
        void OnAuthenticationChallenge(AuthenticationChallengeContext context);
    }
}
```

The OnAuthenticationChallenge method is called by the MVC Framework whenever a request has failed the authentication or authorization policies for an action method. The OnAuthenticationChallenge method is passed an AuthenticationChallengeContext object, which is derived from the ControllerContext class I described in Chapter 17 and which defines the additional properties shown in Table 18-4.

Table 18-4. *The properties defined by the AuthenticationChallengeContext class*

Name	Description
ActionDescriptor	Returns an ActionDescriptor that describes the action method to which the filter has been applied
Result	Sets an ActionResult that expresses the result of the authentication challenge

The most important property is Result, because it allows the authentication filter to pass an ActionResult to the MVC Framework, a process known as *short-circuiting* that I will describe shortly. The best way of explaining how an authentication filter works is through an example. To my mind, the most interesting aspect of authentication filters is that they allow a single controller to define action methods that are authenticated in different ways, so my first step is to add a new controller that simulates Google logins. In Listing 18-14, you can see the definition of the GoogleAccountController.

Listing 18-14. The Contents of the GoogleAccountController.cs File

```
using System.Web.Mvc;
using System.Web.Security;

namespace Filters.Controllers {
    public class GoogleAccountController : Controller {

        public ActionResult Login() {
            return View();
        }

        [HttpPost]
        public ActionResult Login(string username, string password, string returnUrl) {
            if (username.EndsWith("@google.com") && password == "secret") {
                FormsAuthentication.SetAuthCookie(username, false);
                return Redirect(returnUrl ?? Url.Action("Index", "Home"));
            } else {
                ModelState.AddModelError("", "Incorrect username or password");
                return View();
            }
        }
    }
}
```

I don't want to implement real Google logins because it means delving into the dark world of third-party authentication, which is a topic in its own right. Instead, I have created a terrible hack that will authenticate any user name that ends with `@google.com` as long as it is provided with the password `secret`.

At the moment, my Google authentication controller isn't hooked up to the application, and that's where the authentication filter comes in. I created a new class file called `GoogleAuthAttribute.cs`, shown in Listing 18-15, in the `Infrastructure` folder. The `FilterAttribute` class, from which my `GoogleAuth` filter is derived, is the base for all filter classes.

Listing 18-15. The Contents of the GoogleAuthAttribute.cs File

```
using System;
using System.Web.Mvc;
using System.Web.Mvc.Filters;
using System.Web.Routing;

namespace Filters.Infrastructure {

    public class GoogleAuthAttribute : FilterAttribute, IAuthenticationFilter {

        public void OnAuthentication(AuthenticationContext context) {
            // not implemented
        }
```

```
        public void OnAuthenticationChallenge(AuthenticationChallengeContext context) {
            if (context.Result == null) {
                context.Result = new RedirectToRouteResult(new RouteValueDictionary {
                    {"controller", "GoogleAccount"},
                    {"action",  "Login"},
                    {"returnUrl", context.HttpContext.Request.RawUrl}
                });
            }
        }
    }
}
```

My implementation of the `OnAuthenticationChallenge` method checks to see if the `Result` property of the `AuthenticationChallengeContext` argument has been set. This allows me to avoid challenging the user when the filter is run after the action method has executed. Do not worry about that now. I explain why this is important in the *Handling the Final Challenge Request* section later in this chapter.

What's important for this section is that I use the `OnAuthenticationChallenge` method to challenge the user for credentials by redirecting their browser to my `GoogleAccount` controller with a `RedirectToRouteResult`. Authentication filters can use all of the `ActionResult` types that I described in Chapter 17, but the `Controller` convenience methods for creating them are not available, which is why I had to use a `RouteValueDictionary` object to specify the segment values so that a route to the challenge action method can be generated.

Implementing the Authentication Check

My authentication filter is ready to challenge users for their fake Google credentials, and now I can wire up the remaining behavior. The controller will call the `OnAuthentication` method before running any other kind of filter, providing an opportunity to perform a broad authentication check. You don't have to implement the `OnAuthentication` method, but I am going to do so in order to check that I am dealing with a Google account.

The `OnAuthentication` method is passed an `AuthenticationContext` object that, like the `AuthenticationChallengeContext` class, is derived from `ControllerContext` and provides access to all of the information I described in Chapter 17. The `AuthenticationContext` class also defines the properties shown in Table 18-5.

Table 18-5. *The properties defined by the AuthenticationContext class*

Name	Description
`ActionDescriptor`	Returns an `ActionDescriptor` that describes the action method to which the filter has been applied
`Principal`	Returns an `IPrincipal` implementation that identifies the current user, if they have already been authenticated.
`Result`	Sets an `ActionResult` that expresses the result of the authentication check

If the `OnAuthentication` sets a value for the `Result` property of the context object, then the MVC Framework will call the `OnAuthenticationChallenge` method. If the `OnAuthenticationChallenge` method doesn't set a value for the `Result` property on its context object, then the one from the `OnAuthentication` method will be executed.

I use the `OnAuthentication` method to create a result that reports an authentication error to the user, which can then be overridden by the `OnAuthenticationChallenge` method to challenge the user for credentials instead. This allows me to be sure that they see a meaningful response, even if no challenge can be issued (although I must admit that I have yet to encounter a situation where this has happened). In Listing 18-16, you can see how I have implemented the `OnAuthentication` method so that it checks that the request has been authenticated by using any Google credentials.

Listing 18-16. Implementing the OnAuthentication Method in the GoogleAuthAttribute.cs File

```
using System;
using System.Security.Principal;
using System.Web.Mvc;
using System.Web.Mvc.Filters;
using System.Web.Routing;

namespace Filters.Infrastructure {

    public class GoogleAuthAttribute : FilterAttribute, IAuthenticationFilter {

        public void OnAuthentication(AuthenticationContext context) {
            IIdentity ident = context.Principal.Identity;
            if (!ident.IsAuthenticated || !ident.Name.EndsWith("@google.com")) {
                context.Result = new HttpUnauthorizedResult();
            }
        }

        public void OnAuthenticationChallenge(AuthenticationChallengeContext context) {
            if (context.Result == null || context.Result is HttpUnauthorizedResult) {
                context.Result = new RedirectToRouteResult(new RouteValueDictionary {
                    {"controller", "GoogleAccount"},
                    {"action",  "Login"},
                    {"returnUrl", context.HttpContext.Request.RawUrl}
                });
            }
        }
    }
}
```

My implementation of the `OnAuthentication` method checks to see if the request has been authenticated using a username that ends with `@google.com`. If the request is not authenticated or the request is authenticated using a different kind of credential, then I set the `Result` property of the `AuthenticationContext` object to a new `HttpUnauthorizedResult`.

The `HttpUnauthorizedResult` is set as the `Result` value for the `AuthenticationChallengeContext` object that is passed to the `OnAuthenticationChallenge` method and you can see that I have updated this method to challenge the user when this happens, coordinating the actions of the two methods in the filter. The next step is to apply the filter to the controller, which you can see in Listing 18-17.

Listing 18-17. Applying the Authentication Filter in the HomeController.cs File

```
using System.Web.Mvc;
using Filters.Infrastructure;

namespace Filters.Controllers {
    public class HomeController : Controller {

        [Authorize(Users = "admin")]
        public string Index() {
            return "This is the Index action on the Home controller";
        }
```

```
        [GoogleAuth]
        public string List() {
            return "This is the List action on the Home controller";
        }
    }
}
```

I have defined a new action method called `List`, which I decorated with the `GoogleAuth` filter. The result is that access to the `Index` method is secured through the built-in support for forms authentication but that access to the `List` action method is secured through my custom fake Google authentication system.

You can see the effect by starting the application. By default the browser will target the `Index` action method, which will trigger the standard authentication and require you to log in using one of the usernames that I defined in the `Web.config` file. If you then request the `/Home/List` URL, then your existing credentials will be rejected and you will have to authenticate using a Google username.

Combining Authentication and Authorization Filters

You can combine authentication and authorization filters on the same action methods to narrow the scope of your security policy. The MVC Framework will call the `OnAuthentication` method of the authentication filter, just as in the previous example, and move on to run the authorization filter if the request passes the authentication check. If the request doesn't pass the authorization filter, then the `OnAuthenticationChallenge` method of the authentication filter will be called so that you can challenge the user for the required credentials. In Listing 18-18, you can see how I have combined the `GoogleAuth` and `Authorize` filters to restrict access to the `List` action in the `Home` controller.

Listing 18-18. Combining Authentication and Authorization Filters in the HomeController.cs File

```
using System.Web.Mvc;
using Filters.Infrastructure;

namespace Filters.Controllers {
    public class HomeController : Controller {

        [Authorize(Users = "admin")]
        public string Index() {
            return "This is the Index action on the Home controller";
        }

        [GoogleAuth]
        [Authorize(Users = "bob@google.com")]
        public string List() {
            return "This is the List action on the Home controller";
        }
    }
}
```

The `Authorize` filter restricts access to the `bob@google.com` account. If the action method is targeted by another Google account, then the authentication filter `OnAuthenticationChallenge` method will be passed an `AuthenticationChallengeContext` object whose `Result` property is set to an instance of the `HttpUnauthorizedResult` class (which is why I used the same class in the `OnAuthentication` method).

The filters in the Home controller restrict access to the Index method to the user admin, who is authenticated using the AccountController, and restrict access to the List method to the bob@google.com user, who is authenticated through the GoogleAccount controller.

Handling the Final Challenge Request

The MVC Framework calls the OnAuthenticationChallenge method one final time after the action method has been executed, but before the ActionResult is returned and executed. This provides authentication filters an opportunity to respond to the fact that the action has completed or even alter the result (something that is also possible with result filters, which I describe later in the chapter).

It is for this reason that I check the Result property of the AuthenticationChallengeContext object in the OnAuthenticationChallenge method. If I did not, I end up challenging the user for credentials once again, which makes little sense given that the action method has already been executed by this point.

The only reason I have found for responding to this last method call is to clear the authentication for the request, which can be useful when important action methods require temporarily elevated credentials that you want entered each and every time the action is to be executed. In Listing 18-19, you can see how I have implemented this feature.

Listing 18-19. Handling the Final Challenge Call in the GoogleAuthAttribute.cs File

```
using System;
using System.Security.Principal;
using System.Web.Mvc;
using System.Web.Mvc.Filters;
using System.Web.Routing;
using System.Web.Security;

namespace Filters.Infrastructure {

    public class GoogleAuthAttribute : FilterAttribute, IAuthenticationFilter {

        public void OnAuthentication(AuthenticationContext context) {
            IIdentity ident = context.Principal.Identity;
            if (!ident.IsAuthenticated || !ident.Name.EndsWith("@google.com")) {
                context.Result = new HttpUnauthorizedResult();
            }
        }

        public void OnAuthenticationChallenge(AuthenticationChallengeContext context) {
            if (context.Result == null || context.Result is HttpUnauthorizedResult) {
                context.Result = new RedirectToRouteResult(new RouteValueDictionary {
                    {"controller", "GoogleAccount"},
                    {"action",  "Login"},
                    {"returnUrl", context.HttpContext.Request.RawUrl}
                });
            } else {
                FormsAuthentication.SignOut();
            }
        }
    }
}
```

You can see the effect by starting the application and requesting the `Home/List` URL. You will be prompted to provide credentials and you will be able to execute the action method if you authenticate as `bob@google.com`. But you will be prompted for credentials again if you reload the browser, essentially targeting the `List` method a second time.

Using Exception Filters

Exception filters are run only if an unhandled exception has been thrown when invoking an action method. The exception can come from the following locations:

- Another kind of filter (authorization, action, or result filter)
- The action method itself
- When the action result is executed (see Chapter 17 for details on action results)

Creating an Exception Filter

Exception filters implement the `IExceptionFilter` interface, which is shown in Listing 18-20.

Listing 18-20. The IExceptionFilter Interface

```
namespace System.Web.Mvc {

    public interface IExceptionFilter {
        void OnException(ExceptionContext filterContext);
    }
}
```

The `OnException` method is called when an unhandled exception arises. The parameter for this method is an `ExceptionContext` object, which is derived from `ControllerContext` and provides a number of useful properties that you can use to get information about the request, as shown in Table 18-6.

Table 18-6. *Useful ControllerContext Properties*

Name	Type	Description
`Controller`	`ControllerBase`	Returns the controller object for this request
`HttpContext`	`HttpContextBase`	Provides access to details of the request and access to the response
`IsChildAction`	`bool`	Returns `true` if this is a child action (see Chapter 20)
`RequestContext`	`RequestContext`	Provides access to the `HttpContext` and the routing data, both of which are available through other properties
`RouteData`	`RouteData`	Returns the routing data for this request

In addition to the properties inherited from the `ControllerContext` class, the `ExceptionContext` class defines some additional properties which are useful with dealing with exceptions, as shown in Table 18-7.

Table 18-7. *Additional ExceptionContext Properties*

Name	Type	Description
`ActionDescriptor`	`ActionDescriptor`	Provides details of the action method
`Result`	`ActionResult`	The result for the action method; a filter can cancel the request by setting this property to a non-`null` value
`Exception`	`Exception`	The unhandled exception
`ExceptionHandled`	`bool`	Returns `true` if another filter has marked the exception as handled

The exception that has been thrown is available through the `Exception` property. An exception filter can report that it has handled the exception by setting the `ExceptionHandled` property to `true`. All of the exception filters applied to an action are invoked even if this property is set to `true`, so it is good practice to check whether another filter has already dealt with the problem, to avoid attempting to recover from a problem that another filter has resolved.

■ **Note** If none of the exception filters for an action method set the `ExceptionHandled` property to `true`, the MVC Framework uses the default ASP.NET exception handling procedure which will display the dreaded "yellow screen of death."

The `Result` property is used by the exception filter to tell the MVC Framework what to do. The two main uses for exception filters are to log the exception and to display a message to the user. To demonstrate how this all fits together, I have created a new class file called `RangeExceptionAttribute.cs` in the `Infrastructure` folder. The contents of this file are shown in Listing 18-21.

Listing 18-21. The Contents of the RangeExceptionAttribute.cs File

```
using System;
using System.Web.Mvc;

namespace Filters.Infrastructure {
    public class RangeExceptionAttribute : FilterAttribute, IExceptionFilter {

        public void OnException(ExceptionContext filterContext) {
            if (!filterContext.ExceptionHandled &&
                filterContext.Exception is ArgumentOutOfRangeException) {
                    filterContext.Result
                        = new RedirectResult("~/Content/RangeErrorPage.html");
                    filterContext.ExceptionHandled = true;
            }
        }
    }
}
```

This exception filter handles `ArgumentOutOfRangeException` instances by redirecting the user's browser to a file called `RangeErrorPage.html` in the `Content` folder. Notice that I have derived the `RangeExceptionAttribute` class from the `FilterAttribute` class, in addition to implementing the `IExceptionFilter` interface. In order for a .NET attribute

class to be treated as an MVC filter, the class has to implement the `IMvcFilter` interface. You can do this directly, but the easiest way to create a filter is to derive your class from `FilterAttribute`, which implements the required interface and provides some useful basic features, such as handling the default order in which filters are processed (which I will come to later in this chapter).

Applying the Exception Filter

I need to do some groundwork before I can test the exception filter. First, I need to create a `Content` folder in the example project and create the `RangeErrorPage.html` file within it. This is the file that I will direct users to when the exception is handled and you can see the contents of the file in Listing 18-22.

Listing 18-22. The Contents of the RangeErrorPage.html File

```
<!DOCTYPE html>
<html xmlns="http://www.w3.org/1999/xhtml">
<head>
    <title>Range Error</title>
</head>
<body>
    <h2>Sorry</h2>
    <span>One of the arguments was out of the expected range.</span>
</body>
</html>
```

Next, I need to add an action method to the `Home` controller which will throw the exception I want to demonstrate. You can see the addition in Listing 18-23.

Listing 18-23. Adding a New Action in the HomeController.cs File

```
using System;
using System.Web.Mvc;
using Filters.Infrastructure;

namespace Filters.Controllers {
    public class HomeController : Controller {

        [Authorize(Users = "admin")]
        public string Index() {
            return "This is the Index action on the Home controller";
        }

        [GoogleAuth]
        [Authorize(Users = "bob@google.com")]
        public string List() {
            return "This is the List action on the Home controller";
        }
```

```
        public string RangeTest(int id) {
            if (id > 100) {
                return String.Format("The id value is: {0}", id);
            } else {
                throw new ArgumentOutOfRangeException("id", id, "");
            }
        }
    }
}
```

You can see the default exception handling if you start the application and navigate to the `/Home/RangeTest/50` URL. The default routing that Visual Studio creates for an MVC project has a segment variable called `id` which will be set to 50 for this URL, triggering the response shown in Figure 18-3. (See Chapters 15 and 16 for details of routing and URL segments.)

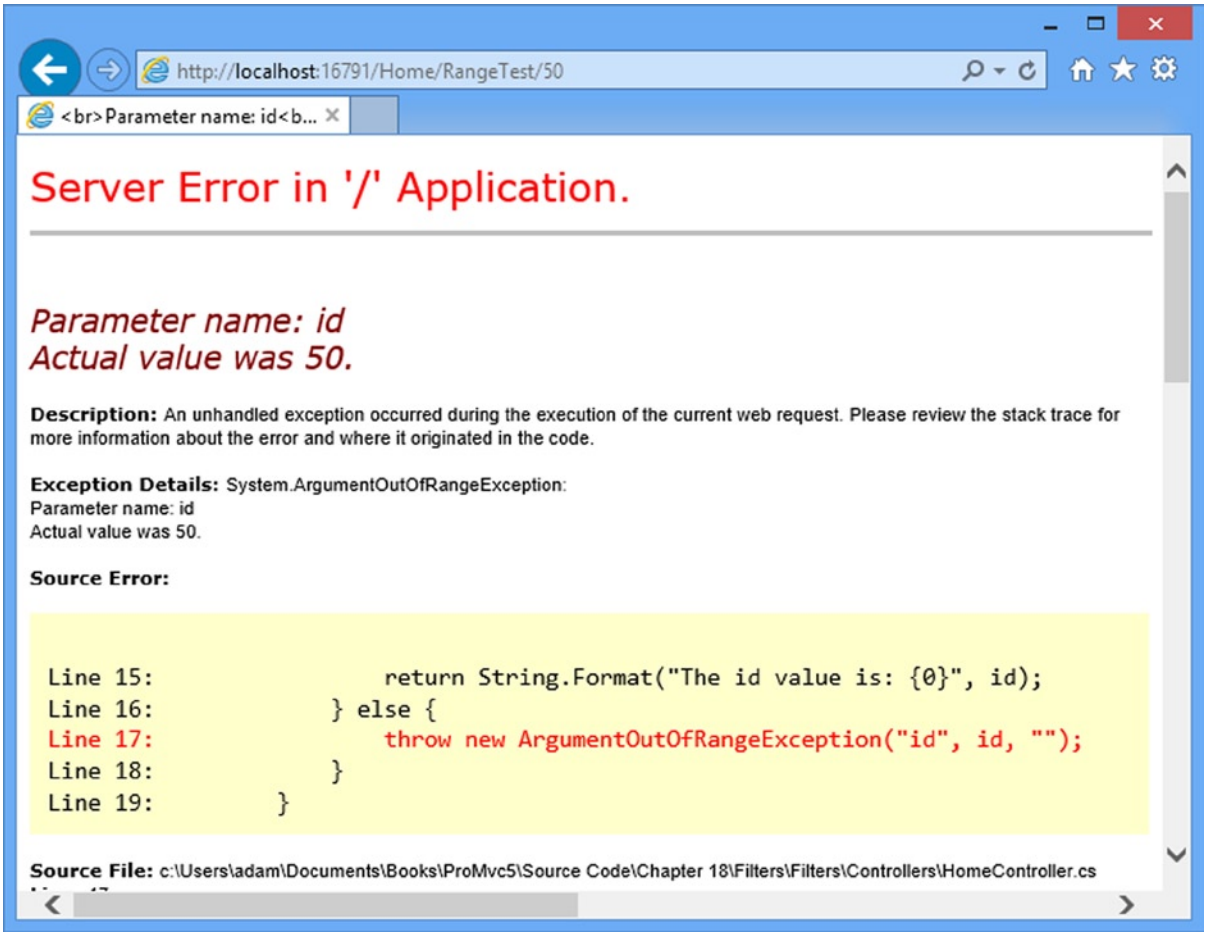

Figure 18-3. *The default exception handling response*

■ **Note** Visual Studio will detect the exception and break the debugger to give you control of the execution of the application. Press F5 or click the Continue button to continue the execution of the application and see the default exception handling behavior.

I can apply the exception filter to either controllers or individual actions, as shown in Listing 18-24.

Listing 18-24. Applying the Filter in the HomeController.cs File

```
...
[RangeException]
public string RangeTest(int id) {
    if (id > 100) {
        return String.Format("The id value is: {0}", id);
    } else {
        throw new ArgumentOutOfRangeException("id");
    }
}
...
```

You can see the effect if you restart the application and navigate to the /Home/RangeTest/50 URL again, as shown in Figure 18-4.

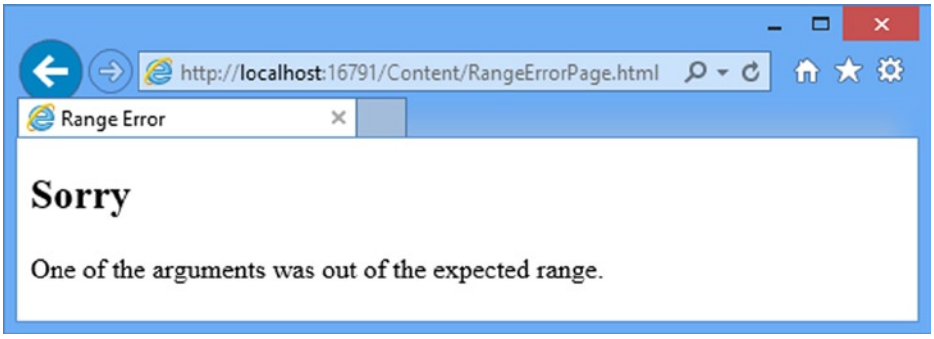

Figure 18-4. *The effect of applying the exception filter*

Using a View to Respond to an Exception

Depending on the exception you are dealing with, displaying a page of static content can be the simplest and safest thing to do. There is little chance of the process going wrong and causing additional problems. However, while you can be confident that the user will see the message, this approach is not especially useful to the user, who gets a generic warning and is dropped out of the application.

An alternative approach is to use a view to display details of the problem and present the user with some contextual information and options they can follow to sort things out. To demonstrate this, I have made some changes to the RangeExceptionAttribute class, as shown in Listing 18-25.

Listing 18-25. Returning a View from an Exception Filter in the RangeExceptionAttribute.cs File

```
using System;
using System.Web.Mvc;

namespace Filters.Infrastructure {
    public class RangeExceptionAttribute : FilterAttribute, IExceptionFilter {

        public void OnException(ExceptionContext filterContext) {

            if (!filterContext.ExceptionHandled &&
                    filterContext.Exception is ArgumentOutOfRangeException) {

                int val = (int)(((ArgumentOutOfRangeException)
                    filterContext.Exception).ActualValue);
                filterContext.Result = new ViewResult {
                    ViewName = "RangeError",
                    ViewData = new ViewDataDictionary<int>(val)};
                    filterContext.ExceptionHandled = true;
            }
        }
    }
}
```

I create a `ViewResult` object and set the values of the `ViewName` and `ViewData` properties to specify the name of the view and the model object that will be passed to it. This is messy code because I am working with the `ViewResult` object directly, rather than relying on the `View` method defined by the `Controller` class that is used in action methods. I am not going to go into this code because I cover views in depth in Chapter 20 and the built-in exception filter, which I describe in the next section, can be used to achieve the same effect more elegantly. I just want you to see how things work behind the scenes.

The `ViewResult` object specifies a view called `RangeError` and passes the `int` value of the argument that caused the exception as the view model object.

To display details of the error, I created the `Views/Shared` folder to the Visual Studio project and created the `RangeError.cshtml` file within it, the contents of which you can see in Listing 18-26.

Listing 18-26. The Contents of the RangeError.cshtml view File

```
@model int

<!DOCTYPE html>
<html>
<head>
    <meta name="viewport" content="width=device-width" />
    <title>Range Error</title>
</head>
<body>
    <h2>Sorry</h2>
    <span>The value @Model was out of the expected range.</span>
    <div>
        @Html.ActionLink("Change value and try again", "Index")
    </div>
</body>
</html>
```

The view file uses standard HTML and Razor tags to present a (slightly) more useful message to the user. The example application is pretty limited, so there is not anything useful I can direct the user to do to resolve the problem. But I have used the `ActionLink` helper method to create a link that targets another action method, just to demonstrate that you have the full set of view features available. You can see the result if you restart the application and navigate to the `/Home/RangeTest/50` URL, as shown in Figure 18-5.

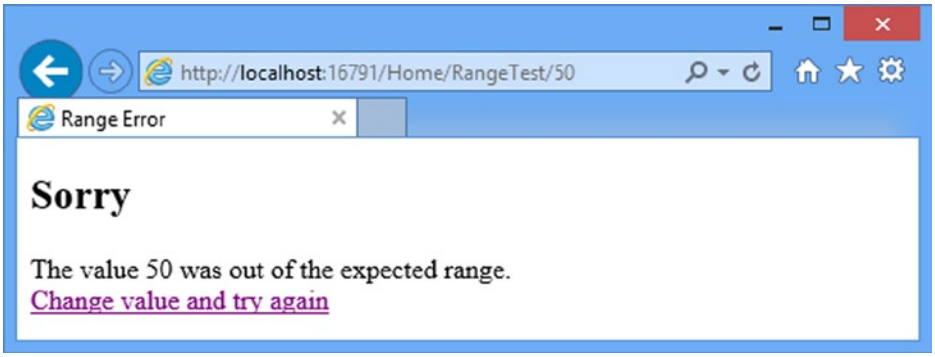

Figure 18-5. *Using a view to display an error message from an exception filter*

Avoiding the Wrong Exception Trap

The benefits of using a view to display an error are that you can use layouts to make the error message consistent with the rest of your application and generate dynamic content that will help the user understand what is going wrong and what they can do about it.

The drawback is that you must thoroughly test your view to make sure that you do not just generate another exception. I see this a lot, where the testing focus is on the main features of the application and does not properly cover the different error situations that can arise. As a simple demonstration, I have added a Razor code block to the `RangeError.cshtml` view that will throw an exception, as shown in Listing 18-27.

Listing 18-27. Adding Code That Will Throw an Exception to the RangeError.cshtml File

```
@model int

@{
    var count = 0;
    var number = Model / count;
}

<!DOCTYPE html>
<html>
<head>
    <meta name="viewport" content="width=device-width" />
    <title>Range Error</title>
</head>
<body>
    <h2>Sorry</h2>
    <span>The value @Model was out of the expected range.</span>
```

```
    <div>
        @Html.ActionLink("Change value and try again", "Index")
    </div>
</body>
</html>
```

When the view is rendered, the code will generate a `DivideByZeroException`. If you start the application and navigate to the `/Home/RangeTest/50` URL again, you will see the exception thrown while trying to render the view and not the one thrown by the controller, as shown in Figure 18-6.

Figure 18-6. *An exception thrown while rendering a view*

This isn't a realistic scenario, but it demonstrates what happens when there are problems in the view. The user sees a bewildering error that does not relate to the problem he or she encountered in the application. When using an exception filter that relies on a view, be careful to test that view thoroughly.

Using the Built-in Exception Filter

I showed you how to create an exception filter because I think understanding what happens behind the scenes in the MVC Framework is a good thing. But you do not often need to create your own filters in real projects because Microsoft has included the `HandleErrorAttribute` in the MVC Framework, which is a built-in implementation of the `IExceptionFilter` interface. With it, you can specify an exception and the names of a view and layout using the properties described in Table 18-8.

Table 18-8. *HandleErrorAttribute Properties*

Name	Type	Description
ExceptionType	Type	The exception type handled by this filter. It will also handle exception types that inherit from the specified value, but will ignore all others. The default value is System.Exception, which means that, by default, it will handle all standard exceptions.
View	string	The name of the view template that this filter renders. If you do not specify a value, it takes a default value of Error, so by default, it renders /Views/<currentControllerName>/Error.cshtml or /Views/Shared/Error.cshtml.
Master	string	The name of the layout used when rendering this filter's view. If you do not specify a value, the view uses its default layout page.

When an unhandled exception of the type specified by ExceptionType is encountered, this filter will render the view specified by the View property (using the default layout or the one specified by the Master property).

Preparing to Use the Built-in Exception Filter

The HandleErrorAttribute filter works only when custom errors are enabled in the Web.config file, which is done by adding a customErrors attribute inside the <system.web> node, as shown in Listing 18-28.

Listing 18-28. Enabling Custom Error in the Web.config File

```
...
<system.web>
  <compilation debug="true" targetFramework="4.5.1" />
  <httpRuntime targetFramework="4.5.1" />
  <authentication mode="Forms">
    <forms loginUrl="~/Account/Login" timeout="2880">
      <credentials passwordFormat="Clear">
        <user name="user"  password="secret"/>
        <user name="admin" password="secret" />
      </credentials>
    </forms>
  </authentication>
  <customErrors mode="On" defaultRedirect="/Content/RangeErrorPage.html"/>
</system.web>
...
```

The default value for the mode attribute is RemoteOnly, which means that connections made from the local machine will always receive the standard yellow page of death errors, which is a problem because IIS Express will only allow local connections. By setting the mode attribute to On, I am specifying that my error handling policy should always be applied, irrespective of where the connections originate. The defaultRedirect attribute specifies a default content page that will be displayed if all else fails.

Applying the Built-in Exception Filter

You can see how I applied the HandleError filter to the Home controller in Listing 18-29.

Listing 18-29. Using the HandleErrorAttribute Filter in the HomeController.cs File

```
...
[HandleError(ExceptionType = typeof(ArgumentOutOfRangeException),
    View = "RangeError")]
public string RangeTest(int id) {
    if (id > 100) {
        return String.Format("The id value is: {0}", id);
    } else {
        throw new ArgumentOutOfRangeException("id", id, "");
    }
}
...
```

I have recreated the situation I had with the custom filter, which is that an ArgumentOutOfRangeException will be dealt with by displaying the RangeError view to the user.

When rendering a view, the HandleErrorAttribute filter passes a HandleErrorInfo view model object, which is a wrapper around the exception that provides additional information that you use in your view. Table 18-9 describes the properties defined by the HandleErrorInfo class.

Table 18-9. *HandleErrorInfo Properties*

Name	Type	Description
ActionName	string	Returns the name of the action method that generated the exception
ControllerName	string	Returns the name of the controller that generated the exception
Exception	Exception	Returns the exception

You can see how I have updated the RangeError.cshtml view to use this model object in Listing 18-30.

Listing 18-30. Using a HandleErrorInfo Model Object in the RangeError.cshtml File

```
@model HandleErrorInfo

@{
    ViewBag.Title = "Sorry, there was a problem!";
}

<!DOCTYPE html>
<html>
<head>
    <meta name="viewport" content="width=device-width" />
    <title>Range Error</title>
</head>
<body>
    <h2>Sorry</h2>
    <span>The value @(((ArgumentOutOfRangeException)Model.Exception).ActualValue)
        was out of the expected range.</span>
```

```
    <div>
        @Html.ActionLink("Change value and try again", "Index")
    </div>
</body>
</html>
```

I have to cast the value of the `Model.Exception` property to the `ArgumentOutOfRangeException` type to be able to read the `ActualValue` property because the `HandleErrorInfo` class is a general-purpose model object that is used to pass any exception to a view.

Using Action Filters

Action filters are filters that can be used for any purpose. The built-in class for creating this kind of filter, `IActionFilter`, is shown in Listing 18-31.

Listing 18-31. The IActionFilter Interface

```
namespace System.Web.Mvc {

    public interface IActionFilter {
        void OnActionExecuting(ActionExecutingContext filterContext);
        void OnActionExecuted(ActionExecutedContext filterContext);
    }
}
```

This interface defines two methods. The MVC Framework calls the `OnActionExecuting` method *before* the action method is invoked. It calls the `OnActionExecuted` method *after* the action method has been invoked.

Implementing the OnActionExecuting Method

The `OnActionExecuting` method is called before the action method is invoked. You can use this opportunity to inspect the request and elect to cancel the request, modify the request, or start some activity that will span the invocation of the action. The parameter to this method is an `ActionExecutingContext` object, which subclasses the `ControllerContext` class and defines the two additional properties described in Table 18-10.

Table 18-10. *ActionExecutingContext Properties*

Name	Type	Description
`ActionDescriptor`	`ActionDescriptor`	Provides details of the action method
`Result`	`ActionResult`	The result for the action method; a filter can cancel the request by setting this property to a non-`null` value

You can use a filter to cancel the request by setting the `Result` property of the parameter to an action result. To demonstrate this, I have created my own action filter class file called `CustomActionAttribute.cs` in the `Infrastructure` folder of the example project, as shown in Listing 18-32.

Listing 18-32. The Contents of the CustomActionAttribute.cs File

```
using System.Web.Mvc;

namespace Filters.Infrastructure {
    public class CustomActionAttribute : FilterAttribute, IActionFilter {

        public void OnActionExecuting(ActionExecutingContext filterContext) {
            if (filterContext.HttpContext.Request.IsLocal) {
                filterContext.Result = new HttpNotFoundResult();
            }
        }

        public void OnActionExecuted(ActionExecutedContext filterContext) {
            // not yet implemented
        }
    }
}
```

In this example, I use the `OnActionExecuting` method to check whether the request has been made from the local machine. If it has, I return a `404–Not Found` response to the user.

■ **Note** You can see from Listing 18-32 that you do not need to implement both methods defined in the `IActionFilter` interface to create a working filter. Be careful not to throw a `NotImplementedException`, which Visual Studio adds to a class when you implement an interface. The MVC Framework calls both methods in an action filter and if an exception is thrown then you will trigger the exception filters. If you do not need to add any logic to a method, then just leave it empty.

You apply an action filter as you would any other attribute. To demonstrate the action filter I created in Listing 18-32, I have added a new action method to the `Home` controller, as shown in Listing 18-33.

Listing 18-33. Adding a New Action in the HomeController.cs File

```
using System;
using System.Web.Mvc;
using Filters.Infrastructure;

namespace Filters.Controllers {
    public class HomeController : Controller {

        [Authorize(Users = "admin")]
        public string Index() {
            return "This is the Index action on the Home controller";
        }

        [GoogleAuth]
        [Authorize(Users = "bob@google.com")]
        public string List() {
            return "This is the List action on the Home controller";
        }
```

```
        [HandleError(ExceptionType = typeof(ArgumentOutOfRangeException),
            View = "RangeError")]
        public string RangeTest(int id) {
            if (id > 100) {
                return String.Format("The id value is: {0}", id);
            } else {
                throw new ArgumentOutOfRangeException("id", id, "");
            }
        }

        [CustomAction]
        public string FilterTest() {
            return "This is the FilterTest action";
        }
    }
}
```

You can test the filter by starting the application and navigating to the `/Home/FilterTest` URL. The request from the browser will, of course, be a local connection and that will cause the custom action filter to generate a 404 error for the browser, as shown in Figure 18-7.

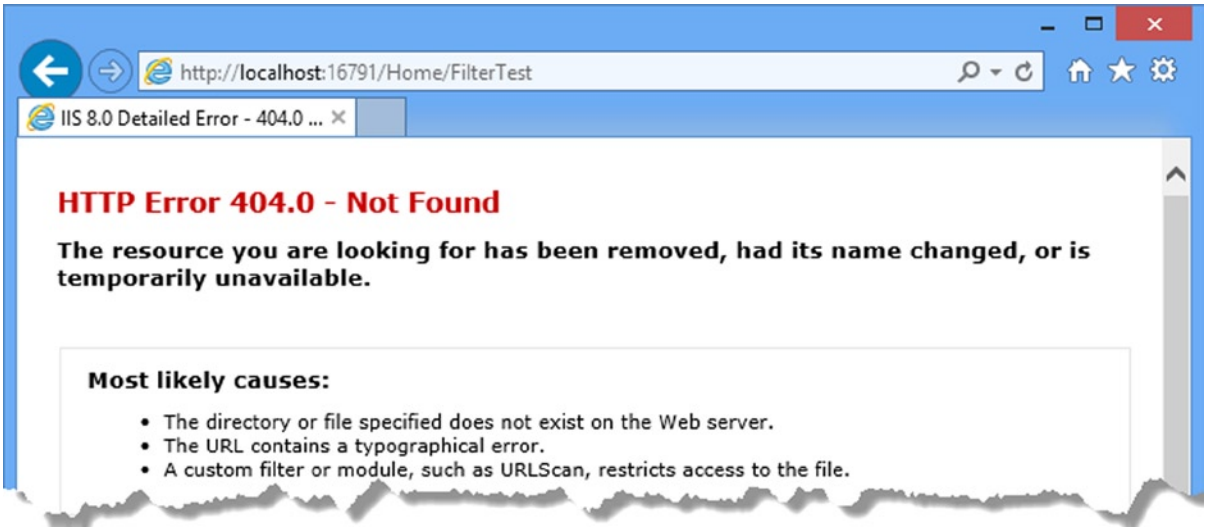

Figure 18-7. *The effect of using an action filter*

■ **Tip** If you want to be sure that it is the filter that is producing the error, simply remove the attribute from the `FilterTest` action method in the `Home` controller and try again.

Implementing the OnActionExecuted Method

You can also use the filter to perform some task that spans the execution of the action method. As a simple example, I have created a class file called `ProfileActionAttribute.cs` in the `Infrastructure` folder, and used it to define a class that measures the amount of time that an action method takes to execute. You can see the code for this filter in Listing 18-34.

Listing 18-34. The Contents of the ProFileActionAttribute.cs File

```
using System.Diagnostics;
using System.Web.Mvc;

namespace Filters.Infrastructure {

    public class ProfileActionAttribute : FilterAttribute, IActionFilter {
        private Stopwatch timer;

        public void OnActionExecuting(ActionExecutingContext filterContext) {
            timer = Stopwatch.StartNew();
        }

        public void OnActionExecuted(ActionExecutedContext filterContext) {
            timer.Stop();
            if (filterContext.Exception == null) {
                filterContext.HttpContext.Response.Write(
                    string.Format("<div>Action method elapsed time: {0:F6}</div>",
                        timer.Elapsed.TotalSeconds));
            }
        }
    }
}
```

In this example, I use the `OnActionExecuting` method to start a timer (using the high-resolution `Stopwatch` timer class in the `System.Diagnostics` namespace). The `OnActionExecuted` method is invoked when the action method has completed. Listing 18-35 shows how I applied the attribute to the `Home` controller. (I removed the previous filter I created so that local requests are not redirected.)

Listing 18-35. Applying the Action Filter in the HomeController.cs File

```
...
[ProfileAction]
public string FilterTest() {
    return "This is the ActionFilterTest action";
}
...
```

If you start the application and navigate to the `/Home/FilterTest` URL, you will see the results illustrated by Figure 18-8.

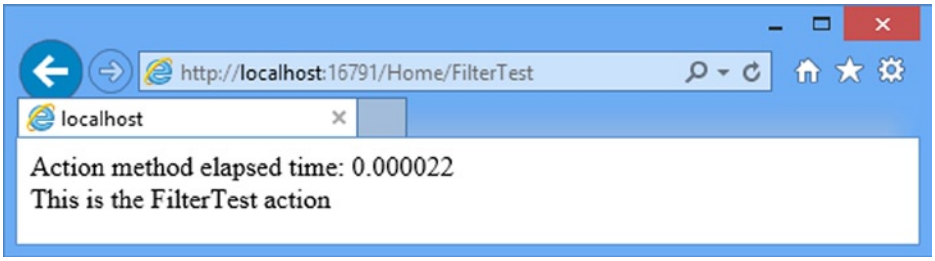

Figure 18-8. *Using an action filter to measure performance*

■ **Tip** Notice that the profile information is shown in the browser before the result of the action method. This is because the action filter is executed after the action method has completed but before the result is processed.

The parameter that is passed to the `OnActionExecuted` method is an `ActionExecutedContext` object. This class defines some useful properties, as shown in Table 18-11. The `Exception` property returns any exception that the action method has thrown, and the `ExceptionHandled` property indicates if another filter has dealt with it.

Table 18-11. *ActionExecutedContext Properties*

Name	Type	Description
`ActionDescriptor`	`ActionDescriptor`	Provides details of the action method
`Canceled`	`bool`	Returns `true` if the action has been canceled by another filter
`Exception`	`Exception`	Returns an exception thrown by another filter or by the action method
`ExceptionHandled`	`bool`	Returns `true` if the exception has been handled
`Result`	`ActionResult`	The result for the action method; a filter can cancel the request by setting this property to a non-`null` value

The `Canceled` property will return `true` if another filter has canceled the request (by setting a value for the `Result` property) since the time that the filter's `OnActionExecuting` method was invoked. The `OnActionExecuted` method will still be called, but only so that it can tidy up and release any resources the filter was using.

Using Result Filters

Result filters are general-purpose filters which operate on the results produced by action methods. Result filters implement the `IResultFilter` interface, which is shown in Listing 18-36.

Listing 18-36. The IResultFilter Interface

```
namespace System.Web.Mvc {

    public interface IResultFilter {
        void OnResultExecuting(ResultExecutingContext filterContext);
        void OnResultExecuted(ResultExecutedContext filterContext);
    }
}
```

In Chapter 17, I explained how action methods return action results, allowing separation between the intent of an action method and its execution. When I apply a result filter to an action method, the `OnResultExecuting` method is invoked *after* the action method has returned an action result but *before* the action result is executed. The `OnResultExecuted` method is invoked after the action result is executed.

The parameters to these methods are `ResultExecutingContext` and `ResultExecutedContext` objects, respectively, and they are similar to their action filter counterparts. They define the same properties, which have the same effects. (See Table 18-11.) To demonstrate a simple result filter, I created a class file called `ProfileResultAttribute.cs` in the `Infrastructure` folder and used it to define the class shown in Listing 18-37.

Listing 18-37. The Contents of the ProFileResultAttribute.cs File

```
using System.Diagnostics;
using System.Web.Mvc;

namespace Filters.Infrastructure {
    public class ProfileResultAttribute : FilterAttribute, IResultFilter {
        private Stopwatch timer;

        public void OnResultExecuting(ResultExecutingContext filterContext) {
            timer = Stopwatch.StartNew();
        }

        public void OnResultExecuted(ResultExecutedContext filterContext) {
            timer.Stop();
            filterContext.HttpContext.Response.Write(
                    string.Format("<div>Result elapsed time: {0:F6}</div>",
                        timer.Elapsed.TotalSeconds));
        }
    }
}
```

This result filter is the complement to the action filter I created in the previous section and measures the amount of time taken to execute the result. You can see how I applied this filter to the `Home` controller in Listing 18-38.

Listing 18-38. Applying the Result Filter in the HomeController.cs File

```
...
[ProfileAction]
[ProfileResult]
public string FilterTest() {
    return "This is the ActionFilterTest action";
}
...
```

Figure 18-9 shows the effect of starting the application and navigating to the /Home/FilterTest URL. You can see that both filters have added data to the response sent to the browser. The output from the result filter is shown after the result from the action method, of course, since the OnResultExecuted method cannot be executed by the MVC Framework until the result has been properly dealt with—which, in this case, means inserting a string value into the result.

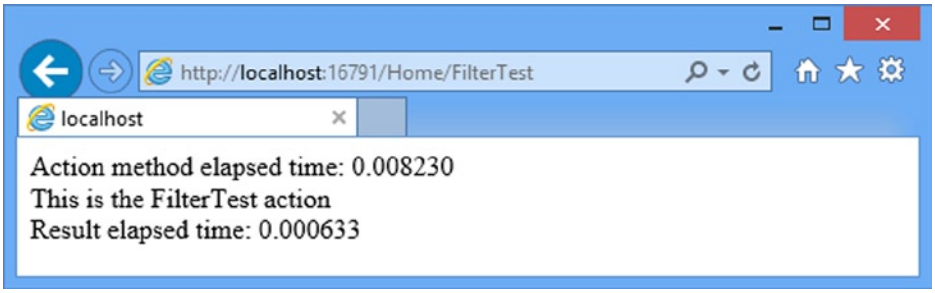

Figure 18-9. *The effect of applying a result filter*

Using the Built-in Action and Result Filter Class

The MVC Framework includes a built-in class that can be derived to create a class that is both an action and result filter. The class, called ActionFilterAttribute, is shown in Listing 18-39.

Listing 18-39. The ActionFilterAttribute Class

```
public abstract class ActionFilterAttribute : FilterAttribute, IActionFilter,
    IResultFilter{

        public virtual void OnActionExecuting(ActionExecutingContext filterContext) {
        }

        public virtual void OnActionExecuted(ActionExecutedContext filterContext) {
        }

        public virtual void OnResultExecuting(ResultExecutingContext filterContext) {
        }

        public virtual void OnResultExecuted(ResultExecutedContext filterContext) {
        }
    }
}
```

The only benefit to using this class is that you do not need to override and implement the methods that you do not intend to use. Otherwise, there is no advantage over implementing the filter interfaces directly.

To demonstrate the use of the ActionFilterAttribute class, I added a class file called ProfileAllAttribute.cs to the Infrastructure folder of the example project and used it to define the class shown in Listing 18-40.

Listing 18-40. The Contents of the ActionFilterAttribute.cs File

```
using System.Diagnostics;
using System.Web.Mvc;

namespace Filters.Infrastructure {
    public class ProfileAllAttribute : ActionFilterAttribute {
        private Stopwatch timer;

        public override void OnActionExecuting(ActionExecutingContext filterContext) {
            timer = Stopwatch.StartNew();
        }

        public override void OnResultExecuted(ResultExecutedContext filterContext) {
            timer.Stop();
            filterContext.HttpContext.Response.Write(
                    string.Format("<div>Total elapsed time: {0:F6}</div>",
                        timer.Elapsed.TotalSeconds));
        }
    }
}
```

The `ActionFilterAttribute` class implements the `IActionFilter` and `IResultFilter` interfaces, which means that the MVC Framework will treat derived classes as both types of filters, even if not all of the methods are overridden. In the example, I have implemented only the `OnActionExecuting` method from the `IActionFilter` interface and the `OnResultExecuted` method from the `IResultFilter` interface. This allows me to continue the profiling theme and measure the time it takes for the action method to execute and the result to be processed as a single unit. Listing 18-41 shows how I applied the filter to the `Home` controller.

Listing 18-41. Applying the Filter in the HomeController.cs File

```
...
[ProfileAction]
[ProfileResult]
[ProfileAll]
public string FilterTest() {
    return "This is the FilterTest action";
}
...
```

You can see the effect of all of these filters if you start the application and navigate to the `/Home/FilterTest` method. Figure 18-10 shows the result.

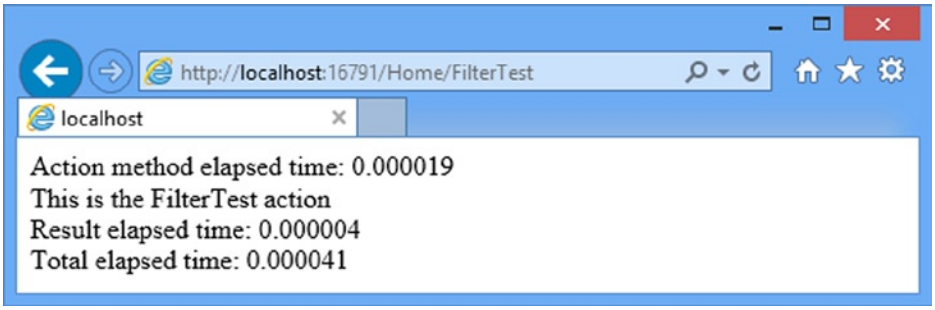

Figure 18-10. *The effect of adding the ProFileAll filter*

Using Other Filter Features

The previous examples have given you all the information you need to work effectively with filters. In the following sections, I will show you some of the advanced MVC Framework filtering capabilities, which are interesting but not as widely used.

Filtering Without Attributes

The normal way of using filters is to apply attributes, as I have demonstrated in the previous sections. However, there is an alternative. The `Controller` class implements the `IAuthenticationFilter`, `IAuthorizationFilter`, `IActionFilter`, `IResultFilter`, and `IExceptionFilter` interfaces. It also provides empty `virtual` implementations of each of the `On`*XXX* methods you have already seen, such as `OnAuthorization` and `OnException`. In Listing 18-42 I have updated the `Home` controller to use this feature and create a self-profiling controller class.

Listing 18-42. Using the Controller Filter Methods in the HomeController.cs File

```
using System;
using System.Diagnostics;
using System.Web.Mvc;
using Filters.Infrastructure;

namespace Filters.Controllers {
    public class HomeController : Controller {
        private Stopwatch timer;

        [Authorize(Users = "admin")]
        public string Index() {
            return "This is the Index action on the Home controller";
        }

        [GoogleAuth]
        [Authorize(Users = "bob@google.com")]
        public string List() {
            return "This is the List action on the Home controller";
        }
```

```
        [HandleError(ExceptionType = typeof(ArgumentOutOfRangeException),             View =
"RangeError")]
        public string RangeTest(int id) {
            if (id > 100) {
                return String.Format("The id value is: {0}", id);
            } else {
                throw new ArgumentOutOfRangeException("id", id, "");
            }
        }

        public string FilterTest() {
            return "This is the FilterTest action";
        }

        protected override void OnActionExecuting(ActionExecutingContext filterContext) {
            timer = Stopwatch.StartNew();
        }

        protected override void OnResultExecuted(ResultExecutedContext filterContext) {
            timer.Stop();
            filterContext.HttpContext.Response.Write(
                    string.Format("<div>Total elapsed time: {0}</div>",
                        timer.Elapsed.TotalSeconds));
        }
    }
}
```

I removed the filters from the `FilterTest` action method because they are no longer required. The `Home` controller will add the profile information to the response for any action method. Figure 18-11 shows the effect of starting the application and navigating to the `/Home/RangeTest/200` URL, which targets the `RangeTest` action without causing the exception I set up to demonstrate the `HandleError` filter.

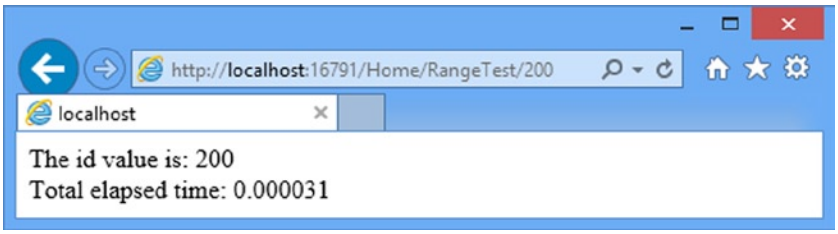

Figure 18-11. *The effect of implementing filter methods directly in the controller*

This technique is most useful when you are creating a base class from which multiple controllers in your project are derived. The whole point of filtering is to put code that is required *across* the application in one reusable location, so using these methods in a class that will not be used as a base for controllers does not make much sense.

■ **Tip** I prefer to use attributes. I like the separation between the controller logic and the filter logic. If you are looking for a way to apply a filter to all of your controllers, continue reading to see how to do that with global filters.

Using Global Filters

Global filters are applied to all of the action methods in all of the controllers in your application. There is a convention for setting up global filters, which is created by Visual Studio automatically when you use the MVC project template but which must be set up manually with the Empty template.

Application-wide configuration is done in classes added to the App_Start folder, which is why I defined routes in Chapters 15 and 16 in the App_Start/RouteConfig.cs file. To create the equivalent for filters, I added a new class file called FilterConfig.cs to the App_Start folder with the content shown in Listing 18-43.

Listing 18-43. The Content of the FilterConfig.cs File

```
using System.Web;
using System.Web.Mvc;

namespace Filters {
    public class FilterConfig {
        public static void RegisterGlobalFilters(GlobalFilterCollection filters) {
            filters.Add(new HandleErrorAttribute());
        }
    }
}
```

This is the same content that Visual Studio would have created for the MVC template. The FilterConfig class defines a static method called RegisterGlobalFilters that receives the collection of global filters, expressed as a GlobalFilterCollection object, to which new filters can be added.

There are two conventions to note in this file. The first is that the FilterConfig class is defined in the Filters namespace and not Filters.App_Start, which is what Visual Studio will use when it creates the file. The second is that the HandleError filter, which I described earlier in the chapter, is always defined as a global filter by calling the Add method on the GlobalFilterCollection object.

■ **Note** You don't have to set up the HandleError filter globally, but it defines the default MVC exception handling policy. This will render the /Views/Shared/Error.cshtml view when an unhandled exception arises. This exception handling policy is disabled by default for development. See the "Creating an Exception Filter" section earlier in the chapter for a note on how to enable it in the Web.config file.

I am going to apply my ProfileAll filter globally and I use the same method call that sets up the HandleError filter, as shown in Listing 18-44.

Listing 18-44. Adding a Global Filter to the FilterConfig.cs File

```
using System.Web;
using System.Web.Mvc;
using Filters.Infrastructure;
```

```
namespace Filters {
    public class FilterConfig {
        public static void RegisterGlobalFilters(GlobalFilterCollection filters) {
            filters.Add(new HandleErrorAttribute());
            filters.Add(new ProfileAllAttribute());
        }
    }
}
```

■ **Tip** Notice that I register the filter globally by creating an instance of the filter class, which means that I need to refer to the class name, including the `Attribute` suffix. The rule is that you omit `Attribute` when applying the filter as an attribute, but include it when directly creating an instance of its class.

The next step is to ensure that the `FilterConfig.RegisterGlobalFilters` method is called from the `Global.asax` file when the application starts. You can see the addition I have made to this file in Listing 18-45.

Listing 18-45. Setting Up Global Filters in the Global.asax File

```
using System;
using System.Collections.Generic;
using System.Linq;
using System.Web;
using System.Web.Mvc;
using System.Web.Routing;

namespace Filters {
    public class MvcApplication : System.Web.HttpApplication {
        protected void Application_Start() {
            AreaRegistration.RegisterAllAreas();
            RouteConfig.RegisterRoutes(RouteTable.Routes);
            FilterConfig.RegisterGlobalFilters(GlobalFilters.Filters);
        }
    }
}
```

To demonstrate the global filter, I have created a new controller called `Customer`, as shown in Listing 18-46. I have created a new controller because I want to use code to which I have not previously applied a filter.

Listing 18-46. The Contents of the CustomerController.cs File

```
using System.Web.Mvc;

namespace Filters.Controllers {
    public class CustomerController : Controller {

        public string Index() {
            return "This is the Customer controller";
        }
    }
}
```

This is a simple controller whose Index action returns a string. Figure 18-12 illustrates the effect of the global filter, which I achieved by starting the application and navigating to the /Customer URL. Even though I have not applied a filter directly to the controller, the global filter adds the profiling information shown in the figure.

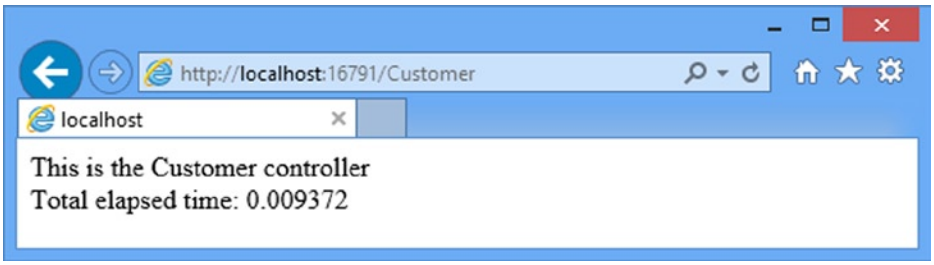

Figure 18-12. *The Effect of a Global Filter*

Ordering Filter Execution

I have already explained that filters are executed by type. The sequence is authentication filters, authorization filters, action filters, and then result filters. The framework executes exception filters at any stage if there is an unhandled exception. However, within each type category, you can take control of the order in which individual filters are used. Listing 18-47 shows a class file called SimpleMessageAttribute.cs that I added to the Infrastructure folder to define a simple filter so as to demonstrate ordering filter execution.

Listing 18-47. The Contents of the SimpleMessageAttribute.cs File

```
using System;
using System.Web.Mvc;

namespace Filters.Infrastructure {

    [AttributeUsage(AttributeTargets.Method, AllowMultiple=true)]
    public class SimpleMessageAttribute : FilterAttribute, IActionFilter {

        public string Message { get; set; }

        public void OnActionExecuting(ActionExecutingContext filterContext) {
            filterContext.HttpContext.Response.Write(
                string.Format("<div>[Before Action: {0}]<div>", Message));
        }

        public void OnActionExecuted(ActionExecutedContext filterContext) {
            filterContext.HttpContext.Response.Write(
                string.Format("<div>[After Action: {0}]<div>", Message));
        }
    }
}
```

This filter writes a message to the response when the `OnActionExecuting` and `OnActionExecuted` methods are invoked, part of which is specified using the `Message` property (which I will set when I apply the filter). I can apply multiple instances of this filter to an action method, as shown in Listing 18-48. (Notice that in the `AttributeUsage` attribute in Listing 18-47, I set the `AllowMultiple` property to `true`).

Listing 18-48. Applying Multiple Filters to an Action in the CustomerController.cs File

```
using System.Web.Mvc;
using Filters.Infrastructure;

namespace Filters.Controllers {
    public class CustomerController : Controller {

        [SimpleMessage(Message="A")]
        [SimpleMessage(Message="B")]
        public string Index() {
            return "This is the Customer controller";
        }
    }
}
```

I created two filters with different messages: the first has a message of `A`, and the other has a message of `B`. I could have used two different filters, but this approach allows me to demonstrate that you can configure global filters through properties. When you run the application and navigate to `/Customer` URL, you will see the result shown in Figure 18-13.

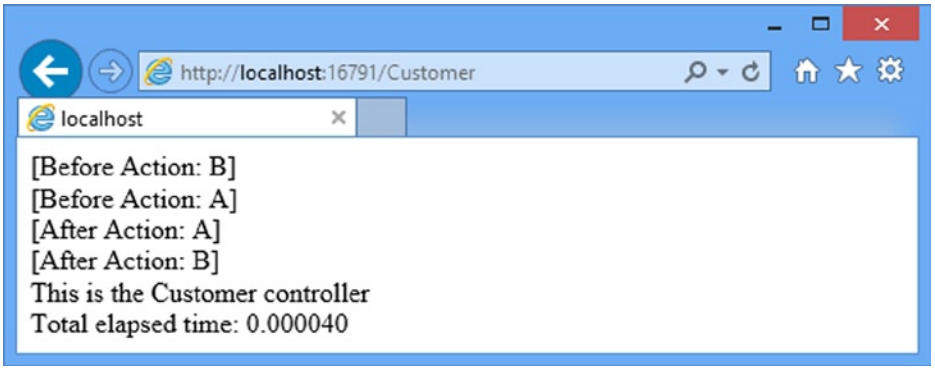

Figure 18-13. *Multiple filters on the same action method*

The MVC Framework executes the `B` filter before the `A` filter, but it could have been the other way around. The MVC Framework does not guarantee any particular order or execution. Most of the time, the order does not matter. When it does, you can use the `Order` property, as shown in Listing 18-49.

Listing 18-49. Using the Order Property in a Filter

```
using System.Web.Mvc;
using Filters.Infrastructure;

namespace Filters.Controllers {
    public class CustomerController : Controller {
```

```
        [SimpleMessage(Message = "A", Order = 1)]
        [SimpleMessage(Message = "B", Order = 2)]
        public string Index() {
            return "This is the Customer controller";
        }
    }
}
```

The `Order` parameter takes an `int` value, and the MVC Framework executes the filters in ascending order. In the listing, I have given the `A` filter the lowest value, so the framework executes it first, as shown in Figure 18-14.

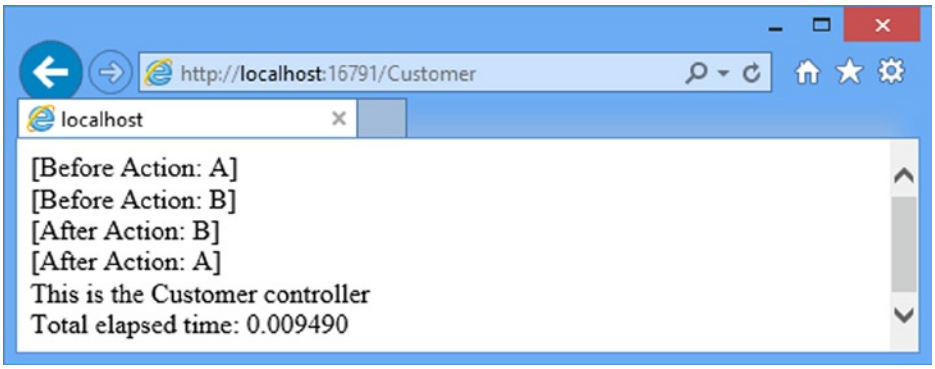

Figure 18-14. *Specifying the order of filter execution*

■ **Note** Notice that the `OnActionExecuting` methods are executed in the order I specified, but the `OnActionExecuted` methods are executed in the reverse order. The MVC Framework builds up a stack of filters as it executes them before the action method, and then unwinds the stack afterward. This unwinding behavior cannot be changed.

If I do not specify a value for the `Order` property, it is assigned a default value of `-1`. This means that if you mix filters so that some have `Order` values and others do not, the ones without these values will be executed first, since they have the lowest `Order` value.

If multiple filters of the same type (say, action filters) have the same `Order` value (say `1`), then the MVC Framework determines the execution order based on where the filter has been applied. Global filters are executed first, then filters applied to the controller class, and then filters applied to the action method.

■ **Note** The order of execution is reversed for exception filters. If exception filters are applied with the same `Order` value to the controller and to the action method, the filter on the action method will be executed first. Global exception filters with the same `Order` value are executed last.

Overriding Filters

There will be occasions when you want to apply a filter globally or at the controller level, but use a different filter for a specific action method. To demonstrate what I mean, I have updated the `SimpleMessage` filter so that it can be applied to an entire controller, as shown in Listing 18-50.

Listing 18-50. Adding Controller-Level Application in the SimpleMessageAttribute.cs File

```
using System;
using System.Collections.Generic;
using System.Linq;
using System.Web;
using System.Web.Mvc;

namespace Filters.Infrastructure {

    [AttributeUsage(AttributeTargets.Class | AttributeTargets.Method,
        AllowMultiple = true)]
    public class SimpleMessageAttribute : FilterAttribute, IActionFilter {

        public string Message { get; set; }

        public void OnActionExecuting(ActionExecutingContext filterContext) {
            filterContext.HttpContext.Response.Write(
                string.Format("<div>[Before Action: {0}]<div>", Message));
        }

        public void OnActionExecuted(ActionExecutedContext filterContext) {
            filterContext.HttpContext.Response.Write(
                string.Format("<div>[After Action: {0}]<div>", Message));
        }
    }
}
```

This change means that the filter can be applied to individual action methods or to the entire controller class. In Listing 18-51, you can see how I have changed the way that the filter is applied to the `Customer` controller.

Listing 18-51. Updating the CustomerController.cs File

```
using System.Web.Mvc;
using Filters.Infrastructure;

namespace Filters.Controllers {
    [SimpleMessage(Message = "A")]
    public class CustomerController : Controller {

        public string Index() {
            return "This is the Customer controller";
        }
```

```
        [SimpleMessage(Message = "B")]
        public string OtherAction() {
            return "This is the Other Action in the Customer controller";
        }
    }
}
```

I have applied the `SimpleMessage` filter to the controller class, meaning that the message `A` will be added to the response when either of the action methods is invoked. I have added a new `OtherAction` method, to which I have applied the `SimpleMessage` filter again, but this time with the message `B`.

The problem is that, by default, the `OtherAction` method is affected by both applications of the filter: at the controller and method level. You can see how this works by starting the application and navigating to `/Customer/OtherAction`, as shown in Figure 18-15.

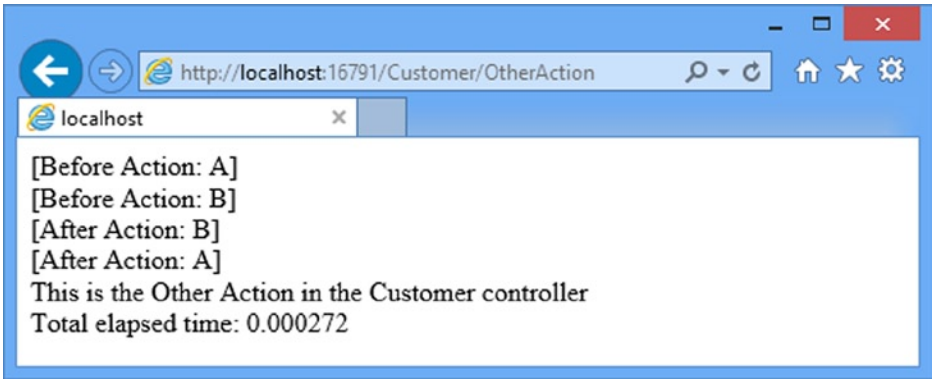

Figure 18-15. *The default filter behavior*

If you want an action method to just be affected by the filters that have been directly applied to it, then you can use a *filter override*. This tells the MVC Framework to ignore any filters that have been defined at a higher-level, such as the controller or globally. Filter overrides are attributes that implement the `IOverrideFilter` interface, which is shown in Listing 18-52.

Listing 18-52. The IOverrideFilter interface

```
namespace System.Web.Http.Filters {
    public interface IOverrideFilter : IFilter {

        Type FiltersToOverride { get; }
    }
}
```

The `FiltersToOverride` method returns the type of filter that will be overridden. I am interested in action filters for this example and to that end I created the `CustomOverrideActionFiltersAttribute.cs` file in the `Infrastructure`. As Listing 18-53 shows, I implemented the `FiltersToOverride` method so that my new attribute overrides the `IActionFilter` type.

■ **Caution** The MVC Framework comes with some built-in filter overrides in the `System.Web.Mvc.Filters` namespace: `OverrideAuthenticationAttribute`, `OverrideActionFiltersAttribute`, and so on. As I write this, these filters do not work. This is because they are derived from `Attribute` and not `FilterAttribute`. I assume that this will be resolved in a later release, but in the meantime you should create custom filter override attributes like the one I demonstrate below.

Listing 18-53. The Contents of the CustomOverrideActionFiltersAttribute.cs File

```
using System;
using System.Web.Mvc;
using System.Web.Mvc.Filters;

namespace Filters.Infrastructure {

    [AttributeUsage(AttributeTargets.Class | AttributeTargets.Method,
        Inherited = true, AllowMultiple = false)]
    public class CustomOverrideActionFiltersAttribute : FilterAttribute,
            IOverrideFilter {
        public Type FiltersToOverride {
            get { return typeof(IActionFilter); }
        }
    }
}
```

I can apply this filter to my controller to prevent the global and controller level action filters from taking effect, as shown in Listing 18-54.

Listing 18-54. Applying a Filter Override in the CustomerController.cs File

```
using System.Web.Mvc;
using Filters.Infrastructure;

namespace Filters.Controllers {
    [SimpleMessage(Message = "A")]
    public class CustomerController : Controller {

        public string Index() {
            return "This is the Customer controller";
        }

        [CustomOverrideActionFilters]
        [SimpleMessage(Message = "B")]
        public string OtherAction() {
            return "This is the Other Action in the Customer controller";
        }
    }
}
```

As Figure 18-16 shows, only the `SimpleMessage` attribute which I have applied directly to the `OtherAction` method is run.

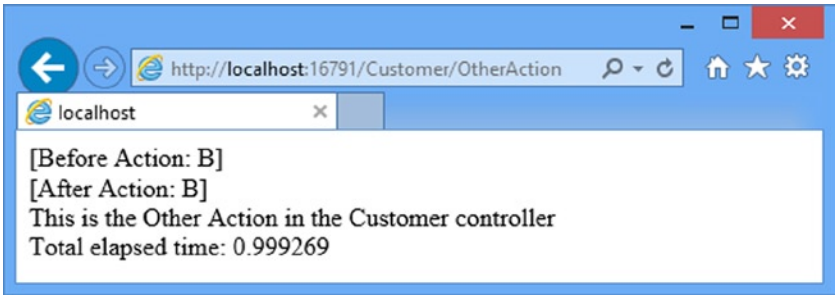

Figure 18-16. *The effect of overriding action filters*

Summary

In this chapter, you have seen how to encapsulate logic that addresses cross-cutting concerns as filters. I showed you the different kinds of filters available and how to implement each of them. You saw how filters can be applied as attributes to controllers and action methods, and how they can be applied as global filters. Filters are a means of extending the logic that is applied when a request is processed, without needing to include that logic in the action method. In the next chapter, I show you how to change and extend the way that the MVC Framework deals with controllers.