

Filtering in ASP.NET MVC

03/23/2012 • 9 minutes to read

In this article

[ASP.NET MVC Filter Types](#)

[Filters Provided in ASP.NET MVC](#)

[How To Create a Filter](#)

[Filter Providers](#)

[Filter Order](#)

[Canceling Filter Execution](#)

[Filter State](#)

[Related Topics](#)

[See Also](#)

In ASP.NET MVC, controllers define action methods that usually have a one-to-one relationship with possible user interactions, such as clicking a link or submitting a form. For example, when the user clicks a link, a request is routed to the designated controller, and the corresponding action method is called.

Sometimes you want to perform logic either before an action method is called or after an action method runs. To support this, ASP.NET MVC provides filters. Filters are custom classes that provide both a declarative and programmatic means to add pre-action and post-action behavior to controller action methods.

A Visual Studio project with source code is available to accompany this topic: [Download](#).

ASP.NET MVC Filter Types

ASP.NET MVC supports the following types of action filters:

- Authorization filters. These implement [IAuthorizationFilter](#) and make security decisions about whether to execute an action method, such as performing authentication or validating properties of the request. The [AuthorizeAttribute](#) class and the [RequireHttpsAttribute](#) class are examples of an authorization filter. Authorization filters run before any other filter.
- Action filters. These implement [IActionFilter](#) and wrap the action method execution. The [IActionFilter](#) interface declares two methods: [OnActionExecuting](#) and [OnActionExecuted](#). [OnActionExecuting](#) runs before the action method. [OnActionExecuted](#) runs after the action method and can perform additional processing, such as providing extra data to the action method, inspecting the return value, or canceling execution of the action method.
- Result filters. These implement [IResultFilter](#) and wrap execution of the [ActionResult](#) object. [IResultFilter](#) declares two methods: [OnResultExecuting](#) and [OnResultExecuted](#). [OnResultExecuting](#) runs before the [ActionResult](#) object is executed. [OnResultExecuted](#) runs after the result and can perform additional processing of the result, such as modifying the HTTP response. The [OutputCacheAttribute](#) class is one example of a result filter.
- Exception filters. These implement [IExceptionFilter](#) and execute if there is an unhandled exception thrown during the execution of the ASP.NET MVC pipeline. Exception filters can be used for tasks such as logging or displaying an error page. The [HandleErrorAttribute](#) class is one example of an exception filter.

The [Controller](#) class implements each of the filter interfaces. You can implement any of the filters for a specific controller by overriding the controller's `On<Filter>` method. For example, you can override the [OnAuthorization](#) method. The simple controller included in the downloadable sample overrides each of the filters and writes out diagnostic information when each filter runs. You can implement the following `On<Filter>` methods in a controller:

- [OnAuthorization](#)
- [OnException](#)
- [OnActionExecuting](#)
- [OnActionExecuted](#)

- [OnResultExecuting](#)
- [OnResultExecuted](#)

Filters Provided in ASP.NET MVC

ASP.NET MVC includes the following filters, which are implemented as attributes. The filters can be applied at the action method, controller, or application level.

- [AuthorizeAttribute](#). Restricts access by authentication and optionally authorization.
- [HandleErrorAttribute](#). Specifies how to handle an exception that is thrown by an action method.

⚠ Note

This filter does not catch exceptions unless the customErrors element is enabled in the Web.config file.

- [OutputCacheAttribute](#). Provides output caching.
- [RequireHttpsAttribute](#). Forces unsecured HTTP requests to be resent over HTTPS.

How To Create a Filter

You can create a filter in the following ways:

- Override one or more of the controller's On<Filter> methods.
- Create an attribute class that derives from [ActionFilterAttribute](#) and apply the attribute to a controller or an action method.
- Register a filter with the filter provider (the [FilterProviders](#) class).

- Register a global filter using the [GlobalFilterCollection](#) class.

A filter can implement the abstract [ActionFilterAttribute](#) class. Some filters, such as [AuthorizeAttribute](#), implement the [FilterAttribute](#) class directly. Authorization filters are always called before the action method runs and called before all other filter types. Other action filters, such as [OutputCacheAttribute](#), implement the abstract [ActionFilterAttribute](#) class, which enables the action filter to run either before or after the action method runs.

You can use the filter attribute declaratively with action methods or controllers. If the attribute marks a controller, the action filter applies to all action methods in that controller.

The following example shows the default implementation of the HomeController class. In the example, the `HandleError` attribute is used to mark the controller. Therefore, the filter applies to all action methods in the controller.

VB

 Copy

```
<HandleError()> _  
Public Class HomeController  
    Inherits System.Web.Mvc.Controller  
  
    Function Index() As ActionResult  
        ViewData("Message") = "Welcome to ASP.NET MVC!"  
  
        Return View()  
    End Function  
  
    Function About() As ActionResult  
        Return View()  
    End Function  
End Class
```

Filter Providers

Multiple filter providers can be registered. Filter providers are registered using the static [Providers](#) property. The [GetFilters\(ControllerContext, ActionDescriptor\)](#) method aggregates the filters from all of the providers into a single list. Providers can be registered in any order; the order they are registered has no impact on the order in which the filter run.

ⓘ Note

Filter providers are a new feature to ASP.NET MVC 3.

By default, ASP.NET MVC registers the following filter providers:

- [Filters](#) for global filters.
- [FilterAttributeFilterProvider](#) for filter attributes.
- [ControllerInstanceFilterProvider](#) for controller instances.

The [GetFilters](#) method returns all of the [IFilterProvider](#) instances in the service locator.

Filter Order

Filters run in the following order:

1. Authorization filters
2. Action filters
3. Response filters
4. Exception filters

For example, authorization filters run first and exception filters run last. Within each filter type, the [Order](#) value specifies the run order. Within each filter type and order, the [Scope](#) enumeration value specifies the order for filters. This enumeration

defines the following filter scope values (in the order in which they run):

1. [First](#)
2. [Global](#)
3. [Controller](#)
4. [Action](#)
5. [Last](#)

For example, an [OnActionExecuting\(ActionExecutingContext\)](#) filter that has the [Order](#) property set to zero and filter scope set to [First](#) runs before an action filter that has the [Order](#) property set to zero and filter scope set to [Action](#). Because exception filters run in reverse order, an exception filter that has the [Order](#) property set to zero and filter scope set to [First](#) runs *after* an action filter that has the [Order](#) property set to zero and filter scope set to [Action](#).

The execution order of filters that have the same type, order, and scope is undefined.

The [OnActionExecuting\(ActionExecutingContext\)](#), [OnResultExecuting\(ResultExecutingContext\)](#), and [OnAuthorization\(AuthorizationContext\)](#) filters run in forward order. The [OnActionExecuted\(ActionExecutedContext\)](#), [OnResultExecuting\(ResultExecutingContext\)](#), and [OnException\(ExceptionContext\)](#) filters run in reverse order.

ⓘ Note

In ASP.NET MVC version 3, the order of execution for exception filters has changed for exception filters that have the same [Order](#) value. In ASP.NET MVC 2 and earlier, exception filters on the controller with the same [Order](#) value as those on an action method were executed before the exception filters on the action method. This would typically be the case if exception filters are applied without a specified [Order](#) value. In ASP.NET MVC 3, this order has been reversed so that the most specific exception handler executes first. For more information, see the filter order section later in this document.

Canceling Filter Execution

You can cancel filter execution in the [OnActionExecuting](#) and [OnResultExecuting](#) methods by setting the [Result](#) property to a non-null value. Any pending [OnActionExecuted](#) and [OnActionExecuting](#) filters will not be invoked and the invoker will not call the [OnActionExecuted](#) method for the canceled filter or for pending filters. The [OnActionExecuted](#) filter for previously run filters will run. All of the [OnResultExecuting](#) and [OnResultExecuted](#) filters will run.

For example, imagine an ASP.NET MVC application that has a home controller and a simple controller. A global request timing filter is applied to the application. The request timing filter implements the four action and result filters ([OnActionExecuting](#), [OnActionExecuted](#), [OnResultExecuting](#) and [OnResultExecuted](#)). Each filter method in the application writes trace information with the name of the filter, the name of the controller, the name of the action and the type of the filter. In that case, the filter type is a request timing filter. A request to the Index action of the Home controller shows the following output in the trace listener (debug window).


Filter Method	Controller	Action	Filter type
OnActionExecuting	Home	Index	Request timing filter
OnActionExecuted	Home	Index	Request timing filter
OnResultExecuting	Home	Index	Request timing filter
OnResultExecuted	Home	Index	Request timing filter

Consider the same application where a trace action filter is applied to the simple controller. The simple controller also implements the four action and result filters. The simple controller has three filters, each of which implements the four action and result methods. A request to the Details action of the Simple controller shows the following output in the trace listener:

Filter Method	Controller	Action	Filter type
OnActionExecuting	Simple	Details	Simple Controller

Filter Method	Controller	Action	Filter type
OnActionExecuting	Simple	Details	Trace action
OnActionExecuting	Simple	Details	Request timing
OnActionExecuted	Simple	Details	Request timing
OnActionExecuted	Simple	Details	Trace action
OnActionExecuted	Simple	Details	Simple Controller
OnResultExecuting	Simple	Details	Simple Controller
OnResultExecuting	Simple	Details	Trace action
OnResultExecuting	Simple	Details	Request timing
OnResultExecuted	Simple	Details	Request timing
OnResultExecuted	Simple	Details	Trace action
OnResultExecuted	Simple	Details	Simple Controller

Now consider the same application where the trace action filter sets the result property to "Home/Index", as shown in the following example:

VB	 Copy
<pre>If filterContext.RouteData.Values.ContainsValue("Cancel") Then filterContext.Result = New RedirectResult("~/Home/Index")</pre>	


```
Trace.WriteLine(" Redirecting from Simple filter to /Home/Index")  
End If
```

The request to the Details action of the Simple controller with the canceled result shows the following output in the trace listener:

Filter Method	Controller	Action	Filter type
OnActionExecuting	Simple	Details	Simple Controller
OnActionExecuting	Simple	Details	Trace action
OnActionExecuted	Simple	Details	Simple Controller
OnResultExecuting	Simple	Details	Simple Controller
OnResultExecuting	Simple	Details	Trace action
OnResultExecuting	Simple	Details	Request timing
OnResultExecuted	Simple	Details	Request timing
OnResultExecuted	Simple	Details	Trace action
OnResultExecuted	Simple	Details	Simple Controller
OnActionExecuting	Home	Index	Request timing filter
OnActionExecuted	Home	Index	Request timing filter

Filter Method	Controller	Action	Filter type
OnResultExecuting	Home	Index	Request timing filter
OnResultExecuted	Home	Index	Request timing filter

The downloadable sample for this topic implements the filtering described in these tables.

Filter State

Do not store filter state in a filter instance. Per-request state would typically be stored in the [Items](#) property. This collection is initialized to empty for every request and is disposed when the request completes. Transient per-user state is typically stored in the user's session. User-state information is often stored in a database. The downloadable sample for this topic includes a timing filter that uses per-request state. For information about how to create stateful filters, see the video [Advanced MVC 3](#).

Related Topics

Title	Description
ASP.NET MVC 3 Service Location, Part 4: Filters	(Blog entry) Describes ASP.NET MVC filters in a service location context.
AuthorizeAttribute	Describes how to use the Authorize attribute to control access to an action method.
OutputCacheAttribute	Describes how to use the OutputCache attribute to provide output caching for an action method.
HandleErrorAttribute	Describes how to use the HandleError attribute to handle exceptions that are thrown by an action method.

Title	Description
Creating Custom Action Filters	Describes how to implement custom action filters.
How to: Create a Custom Action Filter	Explains how to add a custom action filter to an ASP.NET MVC application.

See Also

Other Resources

[Easily Add Performance Counters to Your MVC Application](#)

[Get to Know Action Filters in ASP.NET MVC 3 Using `HandleError`](#)

[ASP.NET MVC Overview](#)