Filters inject extra logic into MVC Framework request processing providing a 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 separation of concerns pattern. Classic examples of cross-cutting concerns are logging, authorisation, and caching.

Before the MVC Framework invokes an action, it inspects the method definition to see if it has attributes that implement any of the 5 possible filter type interfaces:

* IAuthenticationFilter (First executed)
* IAuthorizationFilter
* IActionFilter
* IResultFilter (Last executed)
* IExceptionFilter (executed at any stage if there is an unhandled exception).

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.

Filters can be applied to individual action methods or to an entire controller. You can apply multiple filters, and mix and match the levels at which they are applied. Some filters also take input parameters.

Authorization Filters

These are run after Authenication filters, before action filters and before the action method is invoked. Such action methods can only be invoked by authorized users.

WARNING: Writing security code is dangerous

You could create a class that implements IAuthorizationFilter and create your own security logic but this is a really bad idea. Programming history is littered with programmers who thought they knew how to write good security code but it’s a skill that few people actually possess. Quite often there will be hole in the application’s security code. Whenever possible, it’s best to use code that is widely tested and proven. The MVC Framework has a full-featured authorization filter, which can be derived to implement custom authorization policies.

Authentication Filters

These have a fairly complex lifecycle. They’re run before any other filters and can be used in combination 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.

The IAuthenticationFilter interface demands a OnAuthenicationChallenge method. This 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. Importantly, this defines the ActionDescripor and Result properties. Result allows the authentication filter to pass an ActionResult to the MVC Framework, a process known as short-circuiting.

Handling the final challenge request

The MVC Framework calls on 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. This is why we want to check whether Result is null when OnAuthenticationChallenge method is called as it would make little sense to check for credentials after the action method has been executed.

Using Exception filters

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

* Another kind of filter (authorisation, action or result filter)
* The action method itself
* When the action result is executed.

The exception filter implements the IExceptionFilter interface which has demands a method called OnException. Like other filters it passes the method a context object, ExceptionContext, which is derived from ControllerContext and has useful properties:

|  |  |  |
| --- | --- | --- |
| Name | Type | Description |
| Controller | ControllerBase | Returns the controller object for this request |
| HttpContext | HttpContextBase | Provides acces to details of the request and access to the response |
| IsChildAction | Bool | Returns true if this is a child action |
| 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 the ExceptionContext class has it’s own useful 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 the property to a non-null value |
| Exception | Exception | The unhandled exception |
| ExceptionHandled | Bool | Returns true if another filter has marked the exception as handled. |

Exception response using a view

Depending on the exception you’re dealing with, displaying a page of static content can be the simplest and safest thing to do. However, it’s not the best thing for the user to just be dropped from 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.

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.

However, the drawback is that you must thoroughly test your view to make sure that you don’t just generate another exception

Using the Built-in Exception Filter

It’s good to create your own exception filter as a way of understanding how they work behind the scense but you won’t often have to create your own filters in a real project because of the HandleErrorAttribute provided in the MVC framework. It has the following 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 often ignore others. The default value is System.Exception and so will by default handle all standard exceptions. |
| View | String | The name of the view template that this filter renders. If not specified 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. |

The HandleAttributeError filter works only when custom errors are enabled in the Web.Config file,

Action filters

The IActionFilter interface defines two methods, OnActionExecuting and OnActionExecuted w which the Framework calls before and after the action method is invoked respectively.

The parameter to the OnActionExecuting method is an ActionExecutingContext object, which subclasses the ControllerContext object and has two additional properties: ActionDescriptor and Result.

Using Result Filters

The IResultFilter interface defines two methods, OnResultExecuting and OnResultExecuted. OnResultExecuting 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.

Global filters

These are applied to all action methods in all of the contollers in your application. The MVC convention for global filters is a file in the App\_Start folder named FilterConfig.cs. However, it is not part of the App\_Start namespace in the same way that RouteConfig isn’t. This class defines a static method called RegisterGlobalFilters which receives a collection of filters, expressed as a GlobalFilterCollection object, to which new filters can be added. The second convention is that the HandleErrorAttribute filter is always defined as a global filter.

// We register a filter globally by declaring an instance of the filter class.

filters.Add(new HandleErrorAttribute()); // Always defined as a global filter!

filters.Add(new ProfileAllAttribute());

The next step is to ensure that the FilterConfig.RegisterGlobalFilters method is called from the Global.asax file when the application starts:

namespace Filters

{

public class MvcApplication : System.Web.HttpApplication

{

protected void Application\_Start()

{

AreaRegistration.RegisterAllAreas();

RouteConfig.RegisterRoutes(RouteTable.Routes);

FilterConfig.RegisterGlobalFilters(GlobalFilters.Filters);

}

}

}

Although filters are executed in order of type within each category, you can take control of the order in which individual filters are used.

Global filters are executed first, then filters applied to the Controller class then filters applied to the action method.

Overriding filters

What if we 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.