MANNING

ASP.NET
Core
IN ACTION

Andrew Lock

ASP.NET Core in Action
by Andrew Lock

Chapter 13

brief contents

The MVC filter pipeline

13

This chapter covers

- The MVC filter pipeline and how it differs from middleware
- Creating custom filters to refactor complex action methods
- Using authorization filters to protect your action methods
- Short-circuiting the filter pipeline to bypass action execution
- Injecting dependencies into filters

In part 1, I covered the MVC framework of ASP.NET Core in some detail. You learned about `MvcMiddleware` and how routing is used to select an action method to execute. You also saw model binding, validation, and how to generate a response by returning an `IActionResult` from your actions. In this chapter, I'm going to head deeper into `MvcMiddleware` and look at the *MVC filter pipeline*, sometimes called the *MVC action invocation pipeline*.

369

MVC uses a number of built-in filters to handle crosscutting concerns, such as authorization (controlling which users can access which action methods in your application). Any application that has the concept of users will use authorization filters as a minimum, but filters are much more powerful than this single use case.

This chapter describes the MVC filter pipeline in the context of an MVC request. You'll learn how to create custom filters that you can use in your own apps, and how you can use them to reduce duplicate code in your action methods. You'll learn how to customize your application's behavior for specific actions, as well as how to apply filters globally to modify all of the actions in your app.

Think of the filter pipeline as a mini middleware pipeline running inside `Mvc-Middleware`. Like the middleware pipeline in ASP.NET Core, the filter pipeline consists of a series of components connected as a pipe, so the output of one filter feeds into the input of the next.

This chapter starts by looking at the similarities and differences between MVC filters and middleware, and when you should choose one over the other. You'll learn about all the different types of filters and how they combine to create the filter pipeline for a request that reaches `MvcMiddleware`.

In section 13.2, I'll take you through each filter type in detail, how they fit into the MVC pipeline, and what to use them for. For each one, I'll provide example implementations that you might use in your own application.

A key feature of filters is the ability to short-circuit a request by generating a response and halting progression through the MVC filter pipeline. This is similar to the way short-circuiting works in middleware, but there are subtle differences. On top of that, the exact behavior is slightly different for each filter, which I cover in section 13.3.

You typically add MVC filters to the pipeline by implementing them as attributes added to your controller classes and action methods. Unfortunately, you can't easily use DI with attributes due to the limitations of C#. In section 13.4, I'll show you how to use the `ServiceFilterAttribute` and `TypeFilterAttribute` base classes to enable dependency injection in your filters.

Before we can start writing code, we should get to grips with the basics of the MVC filter pipeline. The first section of this chapter explains what the pipeline is, why you might want to use it, and how it differs from the middleware pipeline.

13.1   Understanding filters and when to use them

The MVC filter pipeline is a relatively simple concept, in that it provides *hooks* into the normal MVC request, as shown in figure 13.1. Say you wanted to ensure that users can create or edit products on an e-commerce app only if they're logged in. The app would redirect anonymous users to a login page instead of executing the action.

Without filters, you'd need to include the same code to check for a logged-in user at the start of each specific action method. With this approach, `MvcMiddleware` would still execute the model binding and validation, even if the user were not logged in.

With filters, you can use the *hooks* in the MVC request to run common code across all, or a subset of, requests. This way, you can do a wide range of things, such as

- Ensuring a user is logged in before an action method, model binding, or validation runs
- Customizing the output format of particular action methods
- Handling model validation failures before an action method is invoked
- Catching exceptions from an action method and handling them in a special way

In many ways, the MVC filter pipeline is like a middleware pipeline, but restricted to `Mvc-Middleware` only. Like middleware, filters are good for handling crosscutting concerns for your application and are a useful tool for reducing code duplication in many cases.

1. A request is received to the URL /recipe/1.

Request

2. The routing module matches the request to the RecipeController.View action and sets id=1.

Routing

3. A variety of different MVC filters run as part of the MVC middleware.

Model binding / validation

4. Filters run before model binding, before the action method runs, and before and after the IActionResult is executed.

Action

MVC Controller

ViewResult Execution
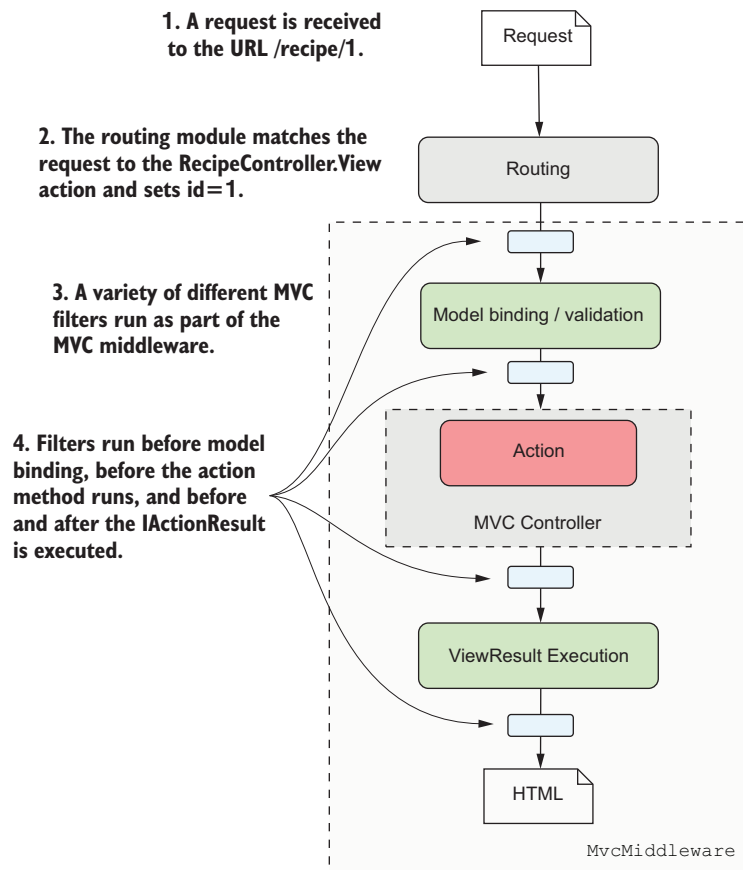
HTML

`MvcMiddleware`

Figure 13.1   Filters run at multiple points in `MvcMiddleware` in the normal handling of a request.

In this section, I'll describe the MVC filter pipeline and how it fits into the overall MVC request. You'll learn about the types of MVC filters, how you can add them to your own apps, and how to control the order in which they execute when handling a request.

13.1.1 The MVC filter pipeline

As you saw in figure 13.1, MVC filters run at a number of different points in the MVC request. This linear view of an MVC request and the filter pipeline that you've used so far doesn't *quite* match up with how these filters execute. There are five types of filter, each of which runs at a different *stage* in `MvcMiddleware`, as shown in figure 13.2.

Each stage lends itself to a particular use case, thanks to its specific location in `Mvc-Middleware`, with respect to model binding, action execution, and result execution.

- *Authorization filters*—These run first in the pipeline, so are useful for protecting your APIs and action methods. If an authorization filter deems the request unauthorized, it will short-circuit the request, preventing the rest of the filter pipeline from running.

- *Resource filters*—After authorization, resource filters are the next filters to run in the pipeline. They can also execute at the *end* of the pipeline, in much the same way that middleware components can handle both the incoming request and the outgoing response. Alternatively, they can completely short-circuit the request pipeline and return a response directly.

   Thanks to their early position in the pipeline, resource filters can have a variety of uses. You could add metrics to an action method, prevent an action method from executing if an unsupported content type is requested, or, as they run before model binding, control the way model binding works for that request.

- *Action filters*—Action filters run just before and after an action is executed. As model binding has already happened, action filters let you manipulate the arguments to the method—before it executes—or they can short-circuit the action completely and return a different `IActionResult`. Because they also run after the action executes, they can optionally customize `IActionResult` before it's executed.

- *Exception filters*—Exception filters can catch exceptions that occur in the filter pipeline and handle them appropriately. They let you write custom MVC-specific error-handling code, which can be useful in some situations. For example, you could catch exceptions in Web API actions and format them differently to exceptions in your MVC actions.

- *Result filters*—Result filters run before and after an action method's `IAction-Result` is executed. This lets you control the execution of the result, or even short-circuit the execution of the result.

Authorization filters run first for every MVC request. If the request isn't authorized, it will short-circuit the pipeline.

Resource filters also run at the end of the pipeline after the result has been executed.

Resource filters run next before model binding runs.

If an exception occurs somewhere in the pipeline, the ExceptionFilter will execute.

Action filters run before and after the action method executes. As they run after model binding, you can use them to customize the arguments passed to the action.

If the action method returns an IActionResult, the result filters will execute before and after the IActionResult is executed.
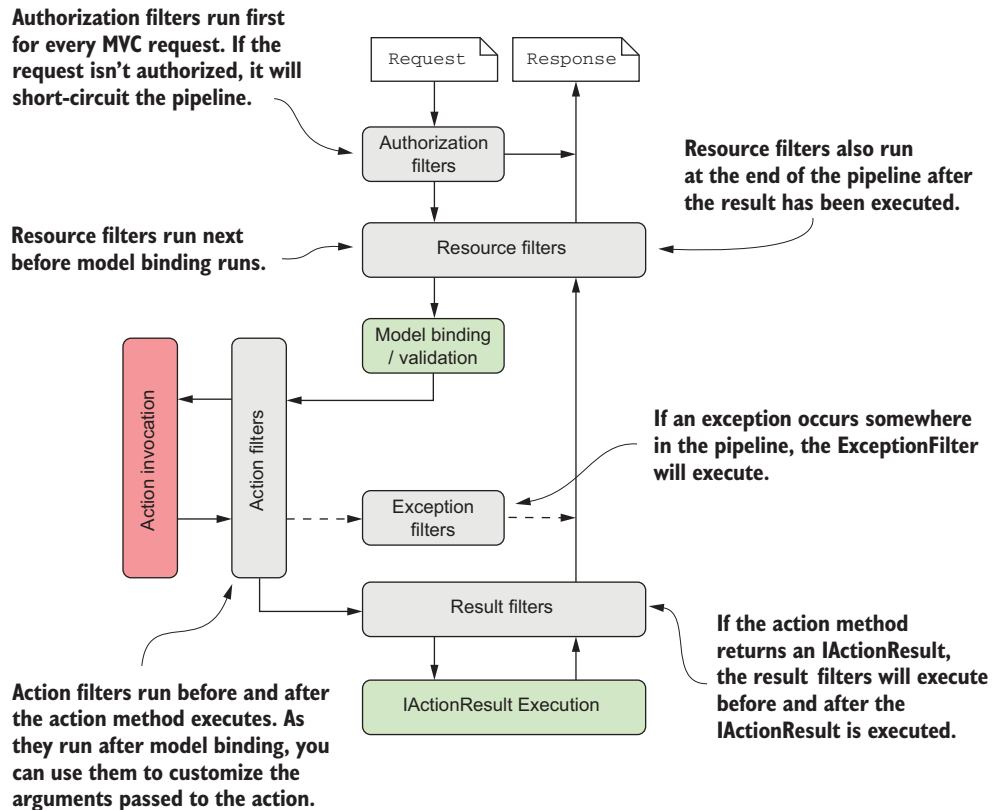
Figure 13.2 The MVC filter pipeline, including the five different filter stages. Some filter stages (resource, action, and result) run twice, before and after the remainder of the pipeline.

Exactly which filter you pick to implement will depend on the functionality you're trying to introduce. Want to short-circuit a request as early as possible? Resource filters are a good fit. Need access to the action method parameters? Use an action filter.

Think of the filter pipeline as a small middleware pipeline that lives by itself in `MvcMiddleware`. Alternatively, you could think of filters as *hooks* into the MVC action invocation process, which let you run code at a particular point in a request's lifecycle.

One of the main questions I hear when people learn about filters in ASP.NET Core is "Why do we need them?" If the filter pipeline is like a mini middleware pipeline, why not use a middleware component directly, instead of introducing the filter concept? That's an excellent point, which I'll tackle in the next section.

13.1.2 *Filters or middleware: which should you choose?*

The filter pipeline is similar to the middleware pipeline in many ways, but there are a number of subtle differences that you should consider when deciding which approach to use. When considering the similarities, they have three main parallels:

- *Requests pass through a middleware component on the way "in" and responses pass through again on the way "out."* Resource, action, and result filters are also two-way, though authorization and exception filters run only once for a request.
- *Middleware can short-circuit a request by returning a response, instead of passing it on to later middleware.* Filters can also short-circuit the MVC filter pipeline by returning a response.
- *Middleware is often used for crosscutting application concerns, such as logging, performance profiling, and exception handling.* Filters also lend themselves to crosscutting concerns.

In contrast, there are three main differences between middleware and filters:

- Middleware can run for all requests; filters will only run for requests that reach `MvcMiddleware`.
- Filters have access to MVC constructs such as `ModelState` and `IActionResults`. Middleware, in general, is independent from MVC, so can't use these concepts.
- Filters can be easily applied to a subset of requests; for example, all actions on a single controller. Middleware doesn't have this concept as a first-class idea (though you could achieve something similar with custom middleware components).

That's all well and good, but how should we interpret these differences? When should we choose one over the other?

I like to think of middleware versus filters as a question of specificity. Middleware is the more general concept, so has the wider reach. If the functionality you need has no MVC-specific requirements, then you should use a middleware component. Exception handling is a great example of this; exceptions could happen anywhere in your application, and you need to handle them, so using exception-handling middleware makes sense.

On the other hand, if you *do* need access to MVC constructs, or you want to behave differently for some MVC actions, then you should consider using a filter. Ironically, this can also be applied to exception handling. You don't want exceptions in your Web API controllers to automatically generate HTML error pages when the client is expecting JSON. Instead, you could use an exception filter on your Web API actions to render the exception to JSON, while letting the exception-handling middleware catch errors from elsewhere in your app.

> **TIP**   Where possible, consider using middleware for crosscutting concerns. Use filters when you need different behavior for different action methods, or where the functionality relies on MVC concepts like `ModelState` validation.

The middleware versus filters argument is a subtle one, and it doesn't matter which you choose as long as it works for you. You can even use middleware components *inside* the filter pipeline as filters, but that's outside the scope of this book.[1]

---

[1]   The "middleware as filters" feature was introduced in ASP.NET Core 1.1. If you're interested, I wrote an introduction to the feature here: http://mng.bz/Mg97.

> **TIP** The middleware as filters feature was introduced in ASP.NET Core 1.1 and is also available in 2.0. The canonical use case is for localizing requests to multiple languages. I have a blog series on how to use the feature here: http://mng.bz/a6Rb.

Filters can be a little abstract in isolation, so in the next section, we'll look at some code and learn how to write a custom filter in ASP.NET Core.

13.1.3 Creating a simple filter

In this section, I show how to create your first filters; in section 13.1.4, you'll see how to apply them to your MVC actions. We'll start small, creating filters that just write to the console, but in section 13.2, we'll look at some more practical examples and discuss some of their nuances.

You implement an MVC filter for a given stage by implementing one of a pair of interfaces—one synchronous (sync), one asynchronous (async):

- *Authorization filters*—IAuthorizationFilter or IAsyncAuthorizationFilter
- *Resource filters*—IResourceFilter or IAsyncResourceFilter
- *Action filters*—IActionFilter or IAsyncActionFilter
- *Exception filters*—IExceptionFilter or IAsyncExceptionFilter
- *Result filters*—IResultFilter or IAsyncResultFilter

You can use any POCO class to implement a filter, but you'll typically implement them as C# attributes, which you can use to decorate your MVC controllers and actions, as you'll see in section 13.1.4. You can achieve the same results with either the sync or async interface, so which you choose should depend on whether any services you call in the filter require async support.

> **NOTE** You should implement *either* the sync interface *or* the async interface, *not both*. If you implement both, then only the async interface will be used.

Listing 13.1 shows a resource filter that implements IResourceFilter and writes to the console when it executes. The OnResourceExecuting method is called when a request first reaches the resource filter stage of the filter pipeline. In contrast, the OnResourceExecuted method is called after the rest of the pipeline has executed; after model binding, action execution, result execution, and all intermediate filters have run.

Listing 13.1 Example resource filter implementing `IResourceFilter`

```
public class LogResourceFilter : Attribute, IResourceFilter
{
    public void OnResourceExecuting(
        ResourceExecutingContext context)
    {
        Console.WriteLine("Executing!");
    }
```

Executed at the start of the pipeline, after authorization filters.

The context contains the HttpContext, routing details, and information about the current action.

```
public void OnResourceExecuted(
    ResourceExecutedContext context)
{
    Console.WriteLine("Executed"");
}
}
```

Executed after model
binding, action execution,
and result execution.

Contains additional context
information, such as the
IActionResult returned by the action

The interface methods are simple and are similar for each stage in the filter pipeline,
passing a context object as a method parameter. Each of the two-method sync filters
has an *Executing and an *Executed method. The type of the argument is different
for each filter, but it contains all of the details for the filter pipeline.

For example, the ResourceExecutingContext passed to the resource filter con-
tains the HttpContext object itself, details about the route that selected this action,
details about the action itself, and so on. Contexts for later filters will contain addi-
tional details, such as the action method arguments for an action filter and the Model-
State.

The context object for the ResourceExecutedContext method is similar, but it also
contains details about how the rest of the pipeline executed. You can check whether
an unhandled exception occurred, you can see if another filter from the same stage
short-circuited the pipeline, or you can see the IActionResult used to generate the
response.

These context objects are powerful and are the key to advanced filter behaviors
like short-circuiting the pipeline and handling exceptions. We'll make use of them in
section 13.2 when creating more complex filter examples.

The async version of the resource filter requires implementing a single method, as
shown in listing 13.2. As for the sync version, you're passed a ResourceExecuting-
Context object as an argument, and you're passed a delegate representing the remain-
der of the filter pipeline. You must call this delegate (asynchronously) to execute the
remainder of the pipeline, which will return an instance of ResourceExecutedContext.

> ### Listing 13.2   Example resource filter implementing `IAsyncResourceFilter`

Executed at the start of the pipeline,
after authorization filters.

You're provided a delegate, which
encapsulates the remainder of the
MVC filter pipeline.

```
public class LogAsyncResourceFilter : Attribute, IAsyncResourceFilter
{
    public async Task OnResourceExecutionAsync(
        ResourceExecutingContext context,
        ResourceExecutionDelegate next)
    {
        Console.WriteLine("Executing async!");
        ResourceExecutedContext executedContext = await next();
        Console.WriteLine("Executed async!");
    }
}
```

Called before
the rest of
the pipeline
executes.

Called after the rest of
the pipeline executes.

Executes the rest of the
pipeline and obtains a
ResourceExecutedContext
instance

The sync and async filter implementations have subtle differences, but for most purposes they're identical. I recommend implementing the sync version if possible, and only falling back to the async version if you need to.

You've created a couple of filters now, so we should look at how to use them in the application. In the next section, we'll tackle two specific issues: how to control which requests execute your new filters and how to control the order in which they execute.

13.1.4 Adding filters to your actions, your controllers, and globally

In section 13.1.2, I discussed the similarities and differences between middleware and filters. One of those differences is that filters can be scoped to specific actions or controllers, so that they only run for certain requests. Alternatively, you can apply a filter globally, so that it runs for every MVC action.

By adding filters in different ways, you can achieve a number of different results. Imagine you have a filter that forces you to log in to view an action. How you add the filter to your app will significantly change your app's behavior:

- *Apply the filter to a single action*—Anonymous users could browse the app as normal, but if they tried to access the protected action, they would be forced to log in.
- *Apply the filter to a controller*—Anonymous users could access actions from other controllers, but accessing any action on the protected controller would force them to log in.
- *Apply the filter globally*—Users couldn't use the app without logging in. Any attempt to access an action would redirect the user to the login page.

> **NOTE**   ASP.NET Core comes with just such a filter out of the box, `Authorize-Filter`. I'll discuss this filter in section 13.2.1, and you'll be seeing a lot more of it in chapter 15.

As I described in the previous section, you normally create filters as attributes, and for good reason—it makes applying them to MVC controllers and actions easy. In this section, you'll see how to apply `LogResourceFilter` from listing 13.1 to an action, a controller, and globally. The level at which the filter applies is called its *scope*.

> **DEFINITION**   The *scope* of a filter refers to how many different actions it applies to. A filter can be scoped to the action method, to the controller, or globally.

You'll start at the most specific scope—applying filters to a single action. The following listing shows an example of an MVC controller that has two action methods: one with `LogResourceFilter` and one without.

Listing 13.3   Applying filters to an action method

```
public class HomeController : Controller
{
    [LogResourceFilter]
    public IActionResult Index()
    {
        return View();
    }
```

> **LogResourceFilter will run as part of the pipeline when executing this action.**

```
    public IActionResult About()
    {                                    This action method has no
        return View();                   filters at the action level.
    }
}
```

Alternatively, if you want to apply the same filter to every action method, you could add the attribute at the controller scope, as in the next listing. Every action method in the controller will use `LogResourceFilter`, without having to specifically decorate each method:

Listing 13.4   Applying filters to a controller

```
[LogResourceFilter]                      The LogResourceFilter Is
public class HomeController : Controller  added to every action on
{                                         the controller.
    public IActionResult Index ()
    {
        return View();
    }                                    Every action in the
                                         controller is decorated
    public IActionResult SendInvoice()   with the filter.
    {
        return View();
    }
}
```

Filters you apply as attributes to controllers and actions are automatically discovered by `MvcMiddleware` when your application starts up. For common attributes, you can go one step further and apply filters globally, without having to decorate individual controllers.

You add global filters in a different way to controller- or action-scoped filters—by adding a filter directly to the MVC services, when configuring MVC in `Startup`. This listing shows three equivalent ways to add a globally scoped filter.

Listing 13.5   Applying filters globally to an application

```
public class Startup
{                                                      Adds filters using
    public void ConfigureServices(IServiceCollection services)  the MvcOptions
    {                                                  object
        services.AddMvc(options =>
        {
You can pass    options.Filters.Add(new LogResourceFilter());
an instance     options.Filters.Add(typeof(LogAsyncResourceFilter));
of the filter   options.Filters.Add<LogAsyncResourceFilter>();
directly...                                            Alternatively, the
        });                                            framework can
    }                                                  create a global
}                                                      filter using a
                                                       generic type
  ... or pass in the Type of the filter                parameter.
  and let the framework create it.
```

With three different scopes in play, you'll often find action methods that have multiple filters applied to them, some applied directly to the action method, and others inherited from the controller or globally. The question then becomes: Which filters run first?

13.1.5 *Understanding the order of filter execution*

You've seen that the filter pipeline contains five different *stages*, one for each type of filter. These stages always run in the fixed order I described in section 13.1.1. But within each stage, you can also have multiple filters of the same type (for example, multiple resource filters) that are part of a single action method's pipeline. These could all have multiple *scopes*, depending on how you added them, as you saw in the last section.

In this section, we're thinking about the *order of filters within a given stage* and how scope affects this. We'll start by looking at the default order, then move on to ways to customize the order to your own requirements.

THE DEFAULT SCOPE EXECUTION ORDER

When thinking about filter ordering, it's important to remember that resource, action, and result filters implement two methods: an *Executing before method and an *Executed after method. The order in which each method executes depends on the scope of the filter, as shown in figure 13.3 for the resource filter stage.

By default, filters execute from the broadest scope (global) to the narrowest (action) when running the *Executing method for each stage. The *Executed methods run in reverse order, from the narrowest scope (action) to the broadest (global).

You'll often find you need a bit more control over this order, especially if you have, for example, multiple action filters applied at the same scope. The MVC framework caters to this requirement by way of the IOrderedFilter interface.

Global filters run first in each filter stage.

Filters scoped to the controller level run after global filters and before action filters.

Filters scoped to the action level run last in a filter stage.

Global scope filter

Controller scope filter

Base controller filter

Action scope filter

The scope of the filters determines the order in which they run within a single stage.

Filters can also be applied to base Controller classes. Base class scoped filters run later than filters on the derived controllers.
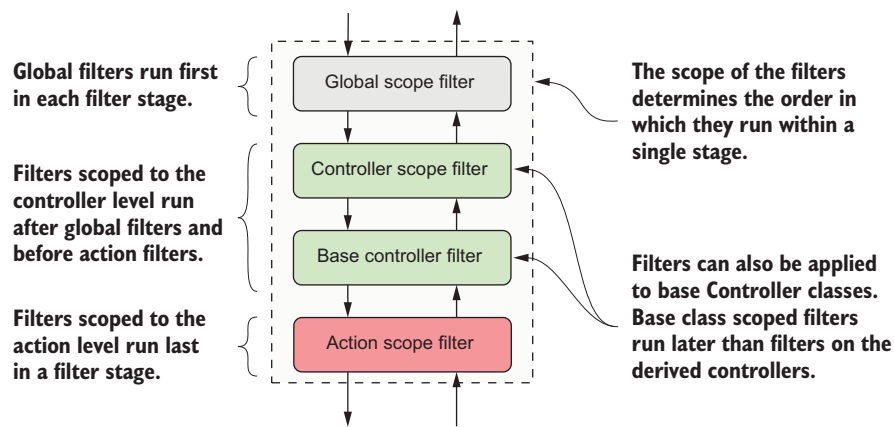
Figure 13.3 The default filter ordering within a given stage, based on the scope of the filters. For the *Executing method, globally scoped filters run first, followed by controller-scoped, and finally, action-scoped filters. For the *Executed method, the filters run in reverse order.

OVERRIDING THE DEFAULT ORDER OF FILTER EXECUTION WITH IORDEREDFILTER

Filters are great for extracting crosscutting concerns from your controller actions, but if you have multiple filters applied to an action, then you'll often need to control the precise order in which they execute.

Scope can get you some of the way, but for those other cases, you can implement `IOrderFilter`. This interface consists of a single property, `Order`:

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

You can implement this property in your filters to set the order in which they execute. `MvcMiddleware` will order the filters for a stage based on this value first, from lowest to highest, and use the default scope order to handle ties, as shown in figure 13.4.

The filters for `Order = -1` execute first, as they have the lowest `Order` value. The controller filter executes first because it has a broader scope than the action filter. The filters with `Order=0` execute next, in the default scope order, as shown in figure 13.4. Finally, the filter with `Order=1` executes.

By default, if a filter doesn't implement `IOrderedFilter`, it's assumed to have `Order = 0`. All of the filters that ship as part of ASP.NET Core have `Order = 0`, so you can implement your own filters relative to these.

This section has covered most of the technical details you need to use filters and create custom implementations for your own application. In the next section, you'll

Filters with the lowest Order number run first. Scope is used to decide tie breaks.

Controller scope filter
Order = -1

Action scope filter
Order = -1

The Order and scope of the filters determines the order in which they run within a single stage.

Global scope filter
Order = 0

By default, filters have an Order of 0.

Controller scope filter
Order = 0

Action scope filter
Order = 0

Filters with the highest value of Order run last for the stage.
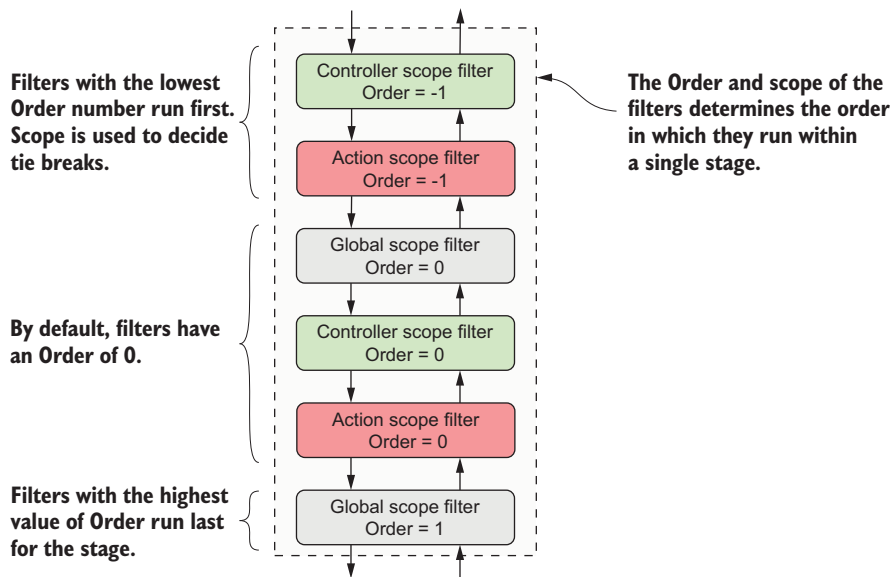
Global scope filter
Order = 1

Figure 13.4   Controlling the filter order for a stage using the `IOrderedFilter` interface. Filters are ordered by the `Order` property first, and then by scope.

see some of the built-in filters provided by ASP.NET Core, as well as some practical examples of filters you might want to use in your own applications.

13.2 Creating custom filters for your application

ASP.NET Core includes a number of filters that you can use, but often, the most useful filters are the custom ones that are specific to your own apps. In this section, you'll work through each of the five types of filters. I'll explain in more detail what they're for and when you should use them. I'll point out examples of these filters that are part of ASP.NET Core itself and you'll see how to create custom filters for an example application.

To give you something realistic to work with, you'll start with a Web API controller for accessing the recipe application from chapter 12. This controller contains two actions: one for fetching a `RecipeDetailViewModel` and another for updating a `Recipe` with new values. This listing shows your starting point for this chapter, including both of the action methods.

> **Listing 13.6 Recipe Web API controller before refactoring to use filters**

```
[Route("api/recipe")]
public class RecipeApiController : Controller        These fields would be passed in as
{                                                    configuration values and are used
    private const bool IsEnabled = true;  ◁────────  to control access to actions.
    public RecipeService _service;
    public RecipeApiController(RecipeService service)
    {
        _service = service;
    }

    [HttpGet("{id}")]
    public IActionResult Get(int id)
    {                                                        If the API isn't enabled,
        if (!IsEnabled) { return BadRequest(); }  ◁───────  block further execution.

        try
        {
            if (!_service.DoesRecipeExist(id))          If the requested Recipe
            {                                           doesn't exist, return a
                return NotFound();                      404 response.
            }
            var detail = _service.GetRecipeDetail(id);

            Response.GetTypedHeaders().LastModified =       Sets the Last-Modified
                detail.LastModified;                        response header to the
                                                            value in the model
            return Ok(detail);
        }
        catch (Exception ex)                        If an exception occurs, catch it,
        {                                           and return the error in an
            return GetErrorResponse(ex);            expected format, as a 500 error.
        }
    }
```

Fetch RecipeDetailViewModel.

Returns the view model with a 200 response

```
[HttpPost("{id}")]
public IActionResult Edit(
    int id, [FromBody] UpdateRecipeCommand command)
{                                                      If the API isn't enabled,
    if (!IsEnabled) { return BadRequest(); }     ◁──  block further execution.

    try
    {
        if (!ModelState.IsValid)                Validates the binding model
        {                                       and returns a 400 response
            return BadRequest(ModelState);      if there are errors
        }

        if (!_service.DoesRecipeExist(id))       If the requested Recipe
        {                                        doesn't exist, return a
            return NotFound();                   404 response.
        }

        _service.UpdateRecipe(command);      Updates the Recipe from the command
        return Ok();                         and returns a 200 response
    }
    catch (Exception ex)
    {                                        If an exception occurs, catch it,
        return GetErrorResponse(ex);         and return the error in an
    }                                        expected format, as a 500 error.
}

private static IActionResult GetErrorResponse(Exception ex)
{
    var error = new
    {
        Success = false,
        Errors = new[]
        {
            ex.Message
        }
    };

    return new ObjectResult(error)
    {
        StatusCode = 500
    };
}
}
```

These action methods currently have a lot of code to them, which hides the intent of each action. There's also quite a lot of duplication between the methods, such as checking that the Recipe entity exists, and formatting exceptions.

   In this section, you're going to refactor this controller to use filters for all the code in the methods that's unrelated to the intent of each action. By the end of the chapter, you'll have a much simpler controller that's far easier to understand, as shown here.

Listing 13.7   Recipe Web API controller after refactoring to use filters

```
[Route("api/recipe")]
[ValidateModel, HandleException, FeatureEnabled(IsEnabled = true)]   ◁
public class RecipeApiController : Controller
{
    public RecipeService _service;
    public RecipeApiController(RecipeService service)
    {
        _service = service;
    }

    [HttpGet("{id}"), EnsureRecipeExists, AddLastModifedHeader]
    public IActionResult Get(int id)
    {

        var detail = _service.GetRecipeDetail(id);
        return Ok(detail);

    }

    [HttpPost("{id}"), EnsureRecipeExists]   ◁
    public IActionResult Edit(
        int id, [FromBody] UpdateRecipeCommand command)
    {
        _service.UpdateRecipe(command);
        return Ok();
    }
}
```

The filters encapsulate the majority of logic common to multiple action methods.

Placing filters at the action level limits them to a single action.

The intent of the action, return a Recipe view model, is much clearer.

Placing filters at the action level can be used to control the order in which they execute.

The intent of the action, update a Recipe, is much clearer.

I think you'll have to agree, the controller in listing 13.7 is much easier to read! In this section, you'll refactor the controller bit by bit, removing crosscutting code to get to something more manageable. All the filters I'll create in this section will use the sync filter interfaces—I'll leave it as an exercise for the reader to create their async counterparts. You'll start by looking at authorization filters and how they relate to security in ASP.NET Core.

13.2.1   Authorization filters: protecting your APIs

Authentication and *authorization* are related, fundamental concepts in security that we'll be looking at in detail in chapters 14 and 15.

> **DEFINITION**   *Authentication* is concerned with determining *who* made a request. *Authorization* is concerned with *what* a user is allowed to access.

Authorization filters run first in the MVC filter pipeline, before any other filters. They control access to the action method by immediately short-circuiting the pipeline when a request doesn't meet the necessary requirements.

ASP.NET Core has a built-in authorization framework that you should use when you need to protect your MVC application or your Web APIs. You can configure this framework with custom policies that let you finely control access to your actions.

TIP   It's possible to write your own authorization filters by implementing `IAuthorizationFilter` or `IAsyncAuthorizationFilter`, but I strongly advise against it. The ASP.NET Core authorization framework is highly configurable and should meet all your needs.

At the heart of the ASP.NET Core authorization framework is an Authorization filter, `AuthorizeFilter`, which you can add to the filter pipeline by decorating your actions or controllers with the `[Authorize]` attribute. In its simplest form, adding the `[Authorize]` attribute to an action, as in the following listing, means the request must be made by an authenticated user to be allowed to continue. If you're not logged in, it will short-circuit the pipeline, returning a 401 Unauthorized response to the browser.

Listing 13.8   Adding `[Authorize]` to an action method

```
public class RecipeApiController : Controller        The Get method has no
{                                                     [Authorize] attribute, so can
    public IActionResult Get(int id)          ◁       be executed by anyone.
    {
        // method body
    }                                                 Adds the AuthorizeFilter to the MVC
                                                      filter pipeline using [Authorize]
    [Authorize]
    public IActionResult Edit(             ◁
        int id, [FromBody] UpdateRecipeCommand command)    The Edit method can
    {                                                      only be executed if
        // method body                                     you're logged in.
    }
}
```

As with all filters, you can apply the `[Authorize]` attribute at the controller level to protect all the actions on a controller, or even globally, to protect every method in your app.

NOTE   We'll explore authorization in detail in chapter 15, including how to add more detailed requirements, so that only specific sets of users can execute an action.

The next filters in the pipeline are resource filters. In the next section, you'll extract some of the common code from `RecipeApiController` and see how easy it is to create a short-circuiting filter.

13.2.2   Resource filters: short-circuiting your action methods

Resource filters are the first general-purpose filters in the MVC filter pipeline. In section 13.1.3, you saw minimal examples of both sync and async resource filters, which logged to the console. In your own apps, you can use resource filters for a wide range of purposes, thanks to the fact they execute so early (and late) in the filter pipeline.

The ASP.NET Core framework includes a few different implementations of resource filters you can use in your apps, for example:

- `ConsumesAttribute`—Can be used to restrict the allowed formats an action method can accept. If your action is decorated with `[Consumes("application/json")]` but the client sends the request as XML, then the resource filter will short-circuit the pipeline and return a 415 Unsupported Media Type response.
- `DisableFormValueModelBindingAttribute`—This filter prevents model binding from binding to form data in the request body. This can be useful if you know an action method will be handling large file uploads that you need to manage manually yourself. The resource filters run before model binding, so you can disable the model binding for a single action in this way.[2]

resource filters are useful when you want to ensure the filter runs early in the pipeline, before model binding. They provide an early hook into the pipeline for your logic, so you can quickly short-circuit the request if you need to.

Look back at listing 13.6 and see if you can refactor any of the code into a Resource filter. One candidate line appears at the start of both the `Get` and `Edit` methods:

```
if (!IsEnabled) { return BadRequest(); }
```

This line of code is a *feature toggle* that you can use to disable the availability of the whole API, based on the `IsEnabled` field. In practice, you'd probably load the `IsEnabled` field from a database or configuration file so you could control the availability dynamically at runtime but, for this example, I'm using a hardcoded value.

This piece of code is self-contained, crosscutting logic, which is somewhat tangential to the main action method intent—a perfect candidate for a filter. You want to execute the feature toggle early in the pipeline, before any other logic, so a resource filter makes sense.

> **TIP** Technically, you could also use an Authorization filter for this example, but I'm following my own advice of "Don't write your own Authorization filters!"

The next listing shows an implementation of `FeatureEnabledAttribute`, which extracts the logic from the action methods and moves it into the filter. I've also exposed the `IsEnabled` field as a property on the filter.

> **Listing 13.9 The `FeatureEnabledAttribute` resource filter**

```
public class FeatureEnabledAttribute : Attribute, IResourceFilter
{
    public bool IsEnabled { get; set; }    ⟵——— Defines whether the feature is enabled
```

---

[2] For details on handling file uploads, see http://mng.bz/2rrk.

```
    public void OnResourceExecuting(              Executes before model binding,
        ResourceExecutingContext context)        early in the filter pipeline

    {
        if (!IsEnabled)                          If the feature isn't enabled,
        {                                        short-circuits the pipeline
            context.Result = new BadRequestResult();   by setting the
        }                                        context.Result property
    }
    public void OnResourceExecuted(              Must be implemented to satisfy
        ResourceExecutedContext context) { }     IResourceFilter, but not needed
}                                                in this case.
```

This simple resource filter demonstrates a number of important concepts, which are applicable to most filter types:

- The filter is an attribute as well as a filter. This lets you decorate your controller and action methods with it using `[FeatureEnabled(IsEnabled = true)]`.
- The filter interface consists of two methods—`*Executing` that runs before model binding and `*Executed` that runs after the result has been executed. You must implement both, even if you only need one for your use case.
- The filter execution methods provide a `context` object. This provides access to, among other things, the `HttpContext` for the request and metadata about the action method the middleware will execute.
- To short-circuit the pipeline, set the `context.Result` property to `IAction-Result`. The MVC pipeline will execute this result to generate the response, bypassing any remaining filters in the pipeline and the action method itself. In this example, if the feature isn't enabled, you bypass the pipeline by returning `BadRequestResult`, which will return a 400 error to the client.

By moving this logic into the resource filter, you can remove it from your action methods, and instead decorate the whole API controller with a simple attribute:

```
[Route("api/recipe"), FeatureEnabled(IsEnabled = true)]
public class RecipeApiController : Controller
```

You've only extracted two lines of code from your action methods so far, but you're on the right track. In the next section, we'll move on to Action filters and extract two more filters from the action method code.

13.2.3   *Action filters: customizing model binding and action results*

Action filters run just after model binding, before the action method executes. Thanks to this positioning, action filters can access all the arguments that will be used to execute the action method, which makes them a powerful way of extracting common logic out of your actions.

On top of this, they also run just after the action method has executed and can completely change or replace the `IActionResult` returned by the action if you want. They can even handle exceptions thrown in the action.

The ASP.NET Core framework includes a number of action filters out of the box. One of these commonly used filters is `ResponseCacheFilter`, which sets HTTP caching headers on your action-method responses.

> **TIP** Caching is a broad topic that aims to improve the performance of an application over the naive approach. But caching can also make debugging issues difficult and may even be undesirable in some situations. Consequently, I often apply `ResponseCacheFilter` to my action methods to set HTTP caching headers that *disable* caching! You can read about this and other approaches to caching in the docs at http://mng.bz/AMSQ.

The real power of action filters comes when you build filters tailored to your own apps by extracting common code from your action methods. To demonstrate, I'm going to create two custom filters for `RecipeApiController`:

- `ValidateModelAttribute`—This will return `BadRequestResult` if the model state indicates that the binding model is invalid and will short-circuit the action execution.
- `EnsureRecipeExistsAttribute`—This will use each action method's id argument to validate that the requested `Recipe` entity exists before the action method runs. If the `Recipe` doesn't exist, the filter will return `NotFoundResult` and will short-circuit the pipeline.

As you saw in chapter 6, `MvcMiddleware` automatically validates your binding models for you before your actions execute, but it's up to you to decide what to do about it. For Razor pages, this can be somewhat complicated by the need to build a view model, so you'll often need some custom code in each action method to handle this. But for Web API controllers, it's common to return a 400 Bad Request response containing a list of the errors, as shown in figure 13.5.

The request is POSTed to the RecipeApiController.

The request body is bound to the action method's binding model.

A 400 Bad Request response is sent, indicating that validation failed for the request.

The response body is sent as a JSON object, providing the name of each field and the error.
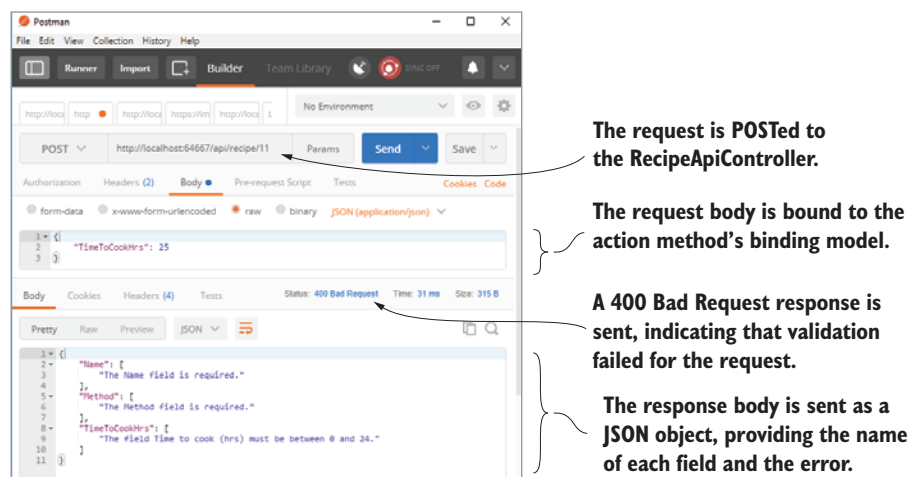
Figure 13.5 Posting data to a Web API using Postman. The data is bound to the action method's binding model and validated. If validation fails, it's common to return a 400 BadRequest response with a list of the validation errors.

It's likely that all of your Web API controllers will use this approach, so an action filter that automatically validates your binding models is a perfect fit. It's normally the first filter I create for any new project. Listing 13.10 shows a simple implementation that you can use in your own apps.[3]

> **Listing 13.10   The action filter for validating `ModelState`**

**For convenience, you derive from
the ActionFilterAttribute base class.**

```
public class ValidateModelAttribute : ActionFilterAttribute
{
    public override void OnActionExecuting(
        ActionExecutingContext context)
    {
        if (!context.ModelState.IsValid)
        {
            context.Result =
                new BadRequestObjectResult(context.ModelState);
        }
    }
}
```

**Overrides the Executing
method to run the filter
before the Action executes**

**If the model isn't valid, set the
Result property; this short-
circuits the action execution.**

**Model binding and validation have already
run at this point, so you can check the state.**

This attribute is self-explanatory and follows a similar pattern to the resource filter in section 13.2.2, but with a few interesting points:

- I have derived from the abstract `ActionFilterAttribute`. This class implements `IActionFilter` and `IResultFilter`, as well as their async counterparts, so you can override the methods you need as appropriate. This avoids needing to add an unused `OnActionExecuted()` method, but is entirely optional and a matter of preference.
- Action filters run after model binding has taken place, so `context.ModelState` contains the validation errors if validation failed.
- Setting the `Result` property on `context` short-circuits the pipeline. But, due to the position of the action filter stage, only the action method execution and later action filters are bypassed; all the other stages of the pipeline run as though the action had executed as normal.

If you apply this action filter to your `RecipeApiController`, you can remove

```
if (!ModelState.IsValid)
{
    return BadRequest(ModelState);
}
```

---

[3]   ASP.NET Core 2.1 added this behavior by default for controllers decorated with the [ApiController] attribute. For details, see https://blogs.msdn.microsoft.com/webdev/2018/02/27/asp-net-core-2-1-web-apis/.

from the start of both the action methods, as it will run automatically in the filter pipe-line. You'll use a similar approach to remove the duplicate code checking whether the id provided as an argument to the action methods corresponds to an existing Recipe entity.

This listing shows the `EnsureRecipeExistsAttribute` action filter. This uses an instance of `RecipeService` to check whether the `Recipe` exists and returns a 404 Not Found if it doesn't.

Listing 13.11   An action filter to check whether a Recipe exists

```
public class EnsureRecipeExistsAtribute : ActionFilterAttribute
{
    public override void OnActionExecuting(
        ActionExecutingContext context)
    {
        var service = (RecipeService) context.HttpContext
            .RequestServices.GetService(typeof(RecipeService));

        var recipeId = (int) context.ActionArguments["id"];
        if (!service.DoesRecipeExist(recipeId))
        {
            context.Result = new NotFoundResult();
        }
    }
}
```

- Fetches an instance of RecipeService from the DI container
- Retrieves the id parameter that will be passed to action method when it executes
- If it doesn't exist, returns a 404 Not Found result and short-circuits the pipeline.
- Checks whether a Recipe entity with the given RecipeId exists

---

As before, you've derived from `ActionFilterAttribute` for simplicity and overridden the `OnActionExecuting` method. The main functionality of the filter relies on the `DoesRecipeExist()` method of `RecipeService`, so the first step is to obtain an instance of `RecipeService`. The `context` parameter provides access to the `Http-Context` for the request, which in turn lets you access the DI container and use `RequestServices.GetService()` to return an instance of `RecipeService`.

> **WARNING**   This technique for obtaining dependencies is known as *service loca-tion* and is generally considered an antipattern.[4] In section 13.4, I'll show a much better way to use the DI container to inject dependencies into your filters.

As well as `RecipeService`, the other piece of information you need is the id argument of the `Get` and `Edit` action methods. In action filters, model binding has already occurred, so the arguments that the MVC middleware will use to execute the action method are already known and are exposed on `context.ActionArguments`.

The action arguments are exposed as `Dictionary<string, object>`, so you can obtain the id parameter using the `"id"` string key. Remember to cast the object to the correct type.

---

[4]   For a detailed discussion on DI patterns and antipatterns, see *Dependency Injection in .NET* by Mark Seemann (Manning, 2012) https://livebook.manning.com/#!/book/dependency-injection-in-dot-net/chapter-5/.

> **TIP**   Whenever I see magic strings like this, I always like to try to replace them by using the `nameof` operator. Unfortunately, `nameof` won't work for method arguments like this, so be careful when refactoring your code. I suggest explicitly applying the action filter to the action method (instead of globally, or to a controller) to remind you about that implicit coupling.

With `RecipeService` and `id` in place, it's a case of checking whether the identifier corresponds to a `Recipe` entity and, if not, setting `context.Result` to `NotFoundResult`. This will short-circuit the pipeline and bypass the action method altogether.

> **NOTE**   Remember, you can have multiple action filters running in a single stage. Short-circuiting the pipeline by setting `context.Result` will prevent later filters in the stage from running, as well as bypassing the action method execution.

Before we move on, it's worth mentioning a special case for action filters. The `Controller` base class implements `IActionFilter` and `IAsyncActionFilter` itself. If you find yourself creating an action filter for a single controller and you want to apply it to every action in that controller, then you can override the appropriate methods on your controller.

Listing 13.12   Overriding action filter methods directly on a `Controller`

```
public class HomeController : Controller     ⟵── Derives from the Controller base class
{
    public override void OnActionExecuting(
        ActionExecutingContext context)          Runs before any other action filters
    { }                                          for every action in the controller

    public override void OnActionExecuted(
        ActionExecutedContext context)           Runs after all other action filters
    { }                                          for every action in the controller
}
```

If you override these methods on your controller, they'll run in the action filter stage of the filter pipeline for every action on the controller. The `OnActionExecuting` `Controller` method runs before any other action filters, regardless of ordering or scope, and the `OnActionExecuted` method runs after all other filters.

> **TIP**   The controller implementation can be useful in some cases, but you can't control the ordering related to other filters. Personally, I generally prefer to break logic into explicit, declarative filter attributes but, as always, the choice is yours.

With the resource and action filters complete, your controller is looking much tidier, but there's one aspect in particular that would be nice to remove: the exception handling. In the next section, we'll look at how to create a custom exception filter for your controller, and why you might want to do this instead of using exception-handling middleware.

13.2.4 *Exception filters: custom exception handling for your action methods*

In chapter 3, I went into some depth about types of error-handling middleware you can add to your apps. These let you catch exceptions thrown from any later middleware and handle them appropriately. If you're using exception-handling middleware, you may be wondering why we need exception filters at all!

The answer to this is pretty much the same as I outlined in section 13.1.2: filters are great for crosscutting concerns, when you need behavior that's either specific to MVC or should only apply to certain routes.

Both of these can apply in exception handling. Exception filters run within `Mvc-Middleware`, so they have access to the context in which the error occurred, such as the action that was executing. This can be useful for logging additional details when errors occur, such as the action parameters that caused the error.

> **WARNING** If you use exception filters to record action method arguments, make sure you're not storing sensitive data, such as passwords or credit card details, in your logs.

You can also use exception filters to handle errors from different routes in different ways. Imagine you have both MVC and Web API controllers in your app, as we do in the recipe app. What happens when an exception is thrown by an MVC controller?

As you saw in chapter 3, the exception travels back up the middleware pipeline, and is caught by exception-handler middleware. The exception-handler middleware will re-execute the pipeline and generate an MVC error page.

That's great for your MVC controllers, but what about exceptions in your Web API controllers? If your API throws an exception, and consequently returns HTML generated by the exception-handler middleware, that's going to break a client who has called the API expecting a JSON response!

Instead, exception filters let you handle the exception inside `MvcMiddleware` and generate an appropriate response body. The exception handler middleware only intercepts errors without a body, so it will let the modified Web API response pass untouched.

Exception filters can catch exceptions from more than your action methods. They'll run if an exception occurs in `MvcMiddleware`

- During model binding or validation
- When the action method is executing
- When an action filter is executing

You should note that exception filters won't catch exceptions thrown in any filters other than action filters, so it's important your resource and result filters don't throw exceptions. Similarly, they won't catch exceptions thrown when executing `IAction-Result` itself.

Now that you know why you might want an exception filter, go ahead and implement one for `RecipeApiController`, as shown next. This lets you safely remove the `try-catch` block from your action methods, knowing that your filter will catch any errors.

Listing 13.13   The `HandleExceptionAttribute` exception filter

ExceptionFilterAttibute is an abstract base class that implements IExceptionFilter.

There's only a single method to override for IExceptionFilter.

```
public class HandleExceptionAttribute : ExceptionFilterAttribute
{
    public override void OnException(ExceptionContext context)
    {
        var error = new
        {
            Success = false,
            Errors = new [] { context.Exception.Message }
        };

        context.Result = new ObjectResult(error)
        {
            StatusCode = 500
        };
        context.ExceptionHandled = true;
    }
}
```

Building a custom object to return in the response

Creates an ObjectResult to serialize the error object and to set the response status code

Marks the exception as handled to prevent it propagating out of MvcMiddleware

It's quite common to have one or two different exception filters in your application, one for your MVC controllers and one for your Web API controllers, but they're not always necessary. If you can handle all the exceptions in your application with a single piece of middleware, then ditch the exception filters and go with that instead.

You're coming to the last type of filter now, result filters, and with it, you're almost done refactoring your `RecipeApiController`. Custom result filters tend to be relatively rare in the apps I've written, but they have their uses, as you'll see.

13.2.5   Result filters: customizing action results before they execute

If everything runs successfully in the pipeline, and there's no short-circuiting, then the next stage of the pipeline after action filters are result filters. These run just before and after the `IActionResult` returned by the Action method (or action filters) is executed.

> **WARNING**   If the pipeline is short-circuited by setting `context.Result`, the result filter stage won't be run, but `IActionResult` will still be executed to generate the response. The one exception to this rule is action filters—these only short-circuit the action execution, as you saw in figure 13.2, and so result filters run as normal, as though the action itself generated the response.

Result filters run immediately after action filters, so many of their use cases are similar, but you typically use result filters to customize the way the IActionResult executes. For example, ASP.NET Core has several result filters built into its framework:

- ProducesAttribute—This forces the Web API result to be serialized to a specific output format. For example, decorating your action method with [Produces("application/xml")] forces the formatters to try to format the response as XML, even if the client doesn't list XML in its Accept header.

- FormatFilterAttribute—Decorating an action method with this filter tells the formatter to look for a route value or query string parameter called format, and to use that to determine the output format. For example, you could call /api/recipe/11?format=json and FormatFilter will format the response as JSON, or call api/recipe/11?format=xml and get the response as XML.[5]

As well as controlling the output formatters, you can use result filters to make any last-minute adjustments before IActionResult is executed and the response is generated.

As an example of the kind of flexibility available to you, in the following listing I demonstrate setting the Last-Modified header, based on the object returned from the action. This is a somewhat contrived example—it's specific enough to a single action that it doesn't warrant being moved to a result filter—but, hopefully, you get the idea!

> ### Listing 13.14   Setting a response header in a result filter

ResultFilterAttribute provides a useful base class you can override

You could also override the Executed method, but the response would already be sent by then.

```
public class AddLastModifedHeaderAttribute : ResultFilterAttribute
{
    public override void OnResultExecuting(
        ResultExecutingContext context)
    {
        if (context.Result is OkObjectResult result
            && result.Value is RecipeDetailViewModel detail)
        {
            var viewModelDate = detail.LastModified;
            context.HttpContext.Response
                .GetTypedHeaders().LastModified = viewModelDate;
        }
    }
}
```

Checks whether the action result returned a 200 Ok result with a view model

Checks whether the view model type is RecipeDetailViewModel . . .

. . . if it is, fetches the LastModified field and sets the Last-Modified header in the response

I've used another helper base class here, ResultFilterAttribute, so you only need to override a single method to implement the filter. Fetch the current IActionResult,

---

[5]   Remember, you need to explicitly configure the XML formatters if you want to serialize to XML. For details, see http://mng.bz/0J2z.

exposed on `context.Result`, and check that it's `OkObjectResult` with `RecipeDetail-ViewModel`. If it is, then fetch the `LastModified` field from the view model and add a `Last-Modified` header to the response.

> **TIP** `GetTypedHeaders()` is an extension method that provides strongly typed access to request and response headers. It takes care of parsing and formatting the values for you. You can find it in the `Microsoft.AspNetCore.Http` namespace.

As with resource and action filters, result filters can implement a method that runs *after* the result has been executed, `OnResultExecuted`. You can use this method, for example, to inspect exceptions that happened during the execution of `IAction-Result`.

> **WARNING** Generally, you can't modify the response in the `OnResult-Executed` method, as `MvcMiddleware` may have already started streaming the response to the client.

That brings us to the end of this detailed look at each of the filters in the MVC pipeline. Looking back and comparing listings 13.6 and 13.7, you can see filters allowed us to refactor the controllers and make the intent of each action method much clearer. Writing your code in this way makes it easier to reason about, as each filter and action has a single responsibility.

In the next section, I'll take a slight detour into exactly what happens when you short-circuit a filter. I've described *how* to do this, by setting the `context.Result` property on a filter, but I haven't yet described exactly what happens. For example, what if there are multiple filters in the stage when it's short-circuited? Do those still run?

13.3 *Understanding pipeline short-circuiting*

A brief warning: the topic of filter short-circuiting can be a little confusing. Unlike middleware short-circuiting, which is cut-and-dried, the MVC filter pipeline is a bit more nuanced. Luckily, you won't often find you need to dig into it, but when you do, you'll be glad of the detail.

You short-circuit the authorization, resource, action, and result filters by setting `context.Result` to `IActionResult`. Setting an action result in this way causes some, or all, of the remaining pipeline to by bypassed. But the filter pipeline isn't entirely linear, as you saw in figure 13.2, so short-circuiting doesn't always do an about-face back down the pipeline. For example, short-circuited action filters only bypass action method execution—the result filters and result execution stages still run.

The other difficulty is what happens if you have more than one type of filter. Let's say you have three resource filters executing in a pipeline. What happens if the second filter causes a short circuit? Any remaining filters are bypassed, but the first resource filter has already run its *Executing command, as shown in figure 13.6. This earlier filter gets to run its *Executed command too, with `context.Cancelled = true`, indicating that a filter in that stage (the resource filter stage) short-circuited the pipeline.

OnResourceExecuting   OnResourceExecuted

1. Resource filter 1 runs its *Executing function.

✓ Filter1 ✓   `context.cancelled=true`

OnResourceExecuting

2. Resource filter 2 runs its *Executing function and short-circuits the pipeline by setting context.Result.

✓ Filter2 ✗

`context.Result`

3. Resource filter 3 (or the rest of the pipeline) never runs.

✗ Filter 3 ✗

5. Resource filter 1 runs its *Executed function. Cancelled is set to true, indicating the pipeline was cancelled.

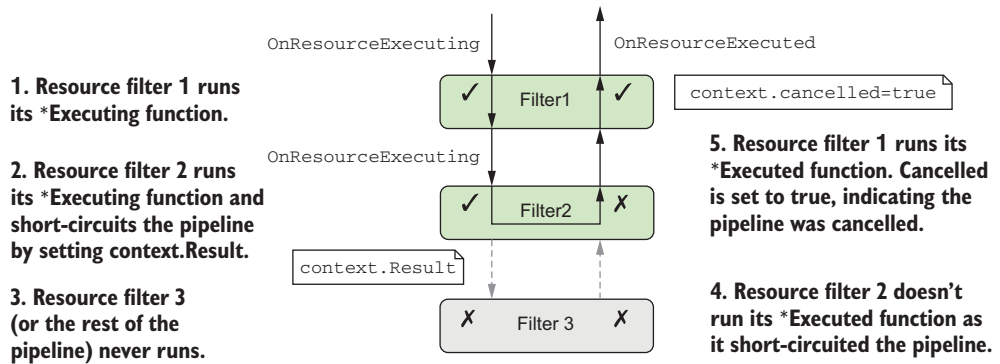**4. Resource filter 2 doesn't run its *Executed function as it short-circuited the pipeline.**

Figure 13.6   The effect of short-circuiting a resource filter on other resource filters in that stage. Later filters in the stage won't run at all, but earlier filters run their `OnResourceExecuted` function.

Understanding which other filters will run when you short-circuit a filter can be somewhat of a chore, but I've summarized each filter in table 13.1. You'll also find it useful to refer to figure 13.2 to visualize the shape of the pipeline when thinking about short-circuits.

Table 13.1   The effect of short-circuiting filters on filter-pipeline execution

Filter type	How to short-circuit?	What else runs?
Authorization filters	Set `context.Result`	Nothing, the pipeline is immediately halted.
Resource filters	Set `context.Result`	Resource-filter *Executed functions from earlier filters run with `context.Cancelled = true`.
Action filters	Set `context.Result`	Only bypasses action method execution. Action filters earlier in the pipeline run their *Executed methods with `context.Cancelled = true`, then result filters, result execution, and resource filters' *Executed methods all run as normal.
Exception filters	Set `context.Result` and `Exception.Handled = true`	All resource-filter *Executed functions run.
Result filters	Set `context.Cancelled = true`	Result filters earlier in the pipeline run their *Executed functions with `context.Cancelled = true`. All resource-filter *Executed functions run as normal.

The most interesting point here is that short-circuiting an action filter doesn't short-circuit much of the pipeline at all. In fact, it only bypasses later action filters and the action method execution itself. By primarily building action filters, you can ensure that other filters, such as result filters that define the output format, run as usual, even when your action filters short-circuit.

The last thing I'd like to talk about in this chapter is how to use DI with your filters. You saw in chapter 10 that DI is integral to ASP.NET Core, and in the next section, you'll see how to design your filters so that the framework can inject service dependencies into them for you.

13.4 *Using dependency injection with filter attributes*

The previous version of ASP.NET used filters, but they suffered from one problem in particular: it was hard to use services from them. This was a fundamental issue with implementing them as attributes that you decorate your actions with. C# attributes don't let you pass dependencies into their constructors (other than constant values), and they're created as singletons, so there's only a single instance for the lifetime of your app.

In ASP.NET Core, this limitation is still there in general, in that filters are typically created as attributes that you add to your controller classes and action methods. What happens if you need to access a transient or scoped service from inside the singleton attribute?

Listing 13.11 showed one way of doing this, using a pseudo service locator pattern to reach into the DI container and pluck out `RecipeService` at runtime. This works but is generally frowned upon as a pattern, in favor of proper DI. How can you add DI to your filters?

The key is to split the filter into two. Instead of creating a class that's both an attribute and a filter, create a filter class that contains the functionality and an attribute that tells `MvcMiddleware` when and where to use the filter.

Let's apply this to the action filter from listing 13.11. Previously, I derived from `ActionFilterAttribute` and obtained an instance of `RecipeService` from the `context` passed to the method. In the following listing, I show two classes, `EnsureRecipeExists-Filter` and `EnsureRecipeExistsAttribute`. The filter class is responsible for the functionality and takes in `RecipeService` as a constructor dependency.
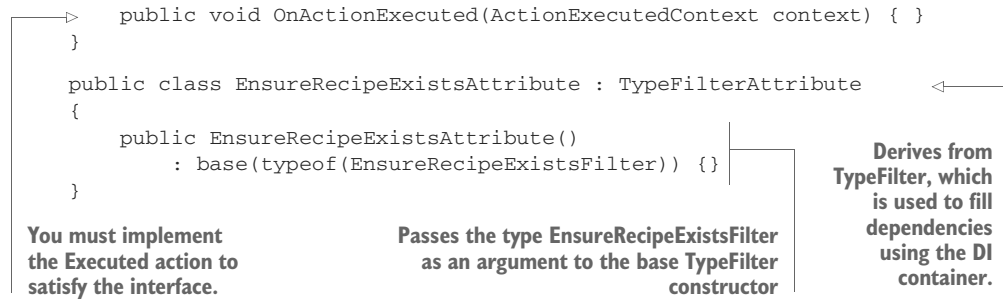
> **Listing 13.15   Using DI in a filter by not deriving from `Attribute`**

```
public class EnsureRecipeExistsFilter : IActionFilter        ◁──   Doesn't derive from
{                                                                  an Attribute class
    private readonly RecipeService _service;
    public EnsureRecipeExistsFilter(RecipeService service)
    {
        _service = service;
    }

    public void OnActionExecuting(ActionExecutingContext context)
    {
        var recipeId = (int) context.ActionArguments["id"];
        if (!_service.DoesRecipeExist(recipeId))
        {
            context.Result = new NotFoundResult();
        }
    }
```

RecipeService is injected into the constructor.

The rest of the method remains the same.

```
        public void OnActionExecuted(ActionExecutedContext context) { }
    }

    public class EnsureRecipeExistsAttribute : TypeFilterAttribute
    {
        public EnsureRecipeExistsAttribute()
            : base(typeof(EnsureRecipeExistsFilter)) {}
    }
```

You must implement the Executed action to satisfy the interface.

Passes the type EnsureRecipeExistsFilter as an argument to the base TypeFilter constructor

Derives from TypeFilter, which is used to fill dependencies using the DI container.

`EnsureRecipeExistsFilter` is a valid filter; you could use it on its own by adding it as a global filter (as global filters don't need to be attributes). But you can't use it directly by decorating controller classes and action methods, as it's not an attribute. That's where `EnsureRecipeExistsAttribute` comes in.

You can decorate your methods with `EnsureRecipeExistsAttribute` instead. This attribute inherits from `TypeFilterAttribute` and passes the `Type` of filter to create as an argument to the base constructor. This attribute acts as a *factory* for `EnsureRecipe-ExistsFilter` by implementing `IFilterFactory`.

When `MvcMiddleware` initially loads your app, it scans your actions and controllers, looking for filters and filter factories. It uses these to form a filter pipeline for every action in your app, as shown in figure 13.7.

When an action decorated with `EnsureRecipeExistsAttribute` is called, `Mvc-Middleware` calls `CreateInstance()` on the attribute. This creates a new instance of `EnsureRecipeExistsFilter` and uses the DI container to populate its dependencies (`RecipeService`).

The MvcMiddleware scans your app looking for filters or attributes that implement IFilterFactory.

Attributes that implement filter interfaces are added directly to the pipeline.

```
public class HomeController
{
  [ValidateModel]
  [EnsureRecipeExistsFilter]
  public IActionResult Index()
  {
    return View();
  }
}
```

`ValidateModelAttribute`

`IFilterFactory`

`EnsureRecipeExistsFilter`

CreateInstance()

The MvcMiddleware calls CreateInstance () on each IFilterFactory when a request is received to create a filter instance, which is added to the pipeline.
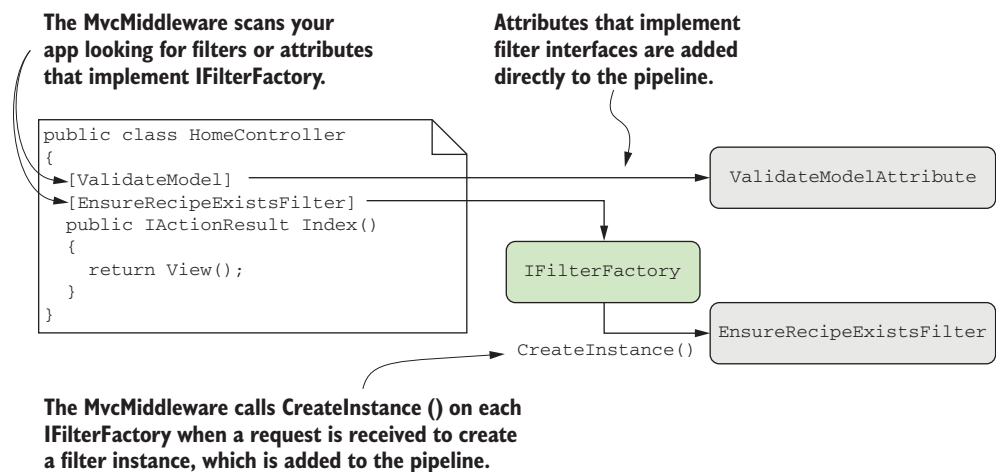
Figure 13.7   The `MvcMiddleware` scans your app on startup to find both filters and attributes that implement `IFilterFactory`. At runtime, the middleware calls `CreateInstance()` to get an instance of the filter.

By using this `IFilterFactory` approach, you get the best of both worlds; you can decorate your controllers and actions with attributes, and you can use DI in your filters. Out of the box, two similar classes provide this functionality, which have slightly different behaviors:

- `TypeFilterAttribute`—Loads all of the filter's dependencies from the DI container and uses them to create a new instance of the filter.
- `ServiceFilterAttribute`—Loads the filter *itself* from the DI container. The DI container takes care of the service lifetime and building the dependency graph. Unfortunately, you also have to explicitly register your filter with the DI container in `ConfigureServices` in `Startup`:

  ```
  services.AddTransient<EnsureRecipeExistsFilter>();
  ```

Whether you choose to use `TypeFilterAttribute` or `ServiceFilterAttribute` is somewhat a matter of preference, and you can always implement a custom `IFilter-Factory` if you need to. The key takeaway is that you can now use DI in your filters. If you don't need to use DI for a filter, then implement it as an attribute directly for simplicity.

> **TIP** I like to create my filters as a nested class of the attribute class when using this pattern. This keeps all the code nicely contained in a single file and indicates the relationship between the classes.

That brings us to the end of this chapter on the filter pipeline. Filters are a somewhat advanced topic, in that they aren't strictly necessary for building basic apps, but I find them extremely useful for ensuring my controller and action methods are simple and easy to understand.

In the next chapter, we'll take our first look at securing your app. We'll discuss the difference between authentication and authorization, the concept of identity in ASP.NET Core, and how you can use the ASP.NET Core Identity system to let users register and log in to your app.

Summary

- The filter pipeline executes as part of `MvcMiddleware` after routing has selected an action method.
- The filter pipeline consists of authorization filters, resource filters, action filters, exception filters, and Result filters. Each filter type is grouped into a *stage*.
- Resource, action, and result filters run twice in the pipeline: an `*Executing` method on the way in and an `*Executed` method on the way out.
- Authorization and exception filters only run once as part of the pipeline; they don't run after a response has been generated.
- Each type of filter has both a sync and an async version. For example, resource filters can implement either the `IResourceFilter` interface or the `IAsync-ResourceFilter` interface. You should use the synchronous interface unless your filter needs to use asynchronous method calls.

- You can add filters globally, at the controller level, or at the action level. This is called the *scope* of the filter.
- Within a given stage, global-scoped filters run first, then controller-scoped, and finally, action-scoped.
- You can override the default order by implementing the `IOrderedFilter` interface. Filters will run from lowest to highest `Order` and use scope to break ties.
- Authorization filters run first in the pipeline and control access to APIs. ASP.NET Core includes an `[Authorization]` attribute that you can apply to action methods so that only logged-in users can execute the action.
- Resource filters run after authorization filters, and again after a result has been executed. They can be used to short-circuit the pipeline, so that an action method is never executed. They can also be used to customize the model binding process for an action method.
- Action filters run after model binding has occurred, just before an action method executes. They also run after the action method has executed. They can be used to extract common code out of an action method to prevent duplication.
- The `Controller` base class also implements `IActionFilter` and `IAsyncActionFilter`. They run at the start and end of the action filter pipeline, regardless of the ordering or scope of other action filters.
- Exception filters execute after action filters, when an action method has thrown an exception. They can be used to provide custom error handling specific to the action executed.
- Generally, you should handle exceptions at the middleware level, but exception filters let you customize how you handle exceptions for specific actions or controllers.
- Result filters run just before and after an `IActionResult` is executed. You can use them to control how the action result is executed, or to completely change the action result that will be executed.
- You can use `ServiceFilterAttribute` and `TypeFilterAttribute` to allow dependency injection in your custom filters. `ServiceFilterAttribute` requires that you register your filter and all its dependencies with the DI container, whereas `TypeFilterAttribute` only requires that the filter's dependencies have been registered.

.NET DEVELOPMENT

ASP.NET Core IN ACTION

Andrew Lock

The dev world has permanently embraced open platforms with flexible tooling, and ASP.NET Core has changed with it. This free, open source web framework delivers choice without compromise. You can enjoy the benefits of a mature, well-supported stack and the freedom to develop and deploy from and onto any cloud or on-prem platform.

ASP.NET Core in Action opens up the world of cross-platform web development with .NET. You'll start with a crash course in .NET Core, immediately cutting the cord between ASP.NET and Windows. Then, you'll begin to build amazing web applications step by step, systematically adding essential features like logins, configuration, dependency injection, and custom components. Along the way, you'll mix in important process steps like testing, multiplatform deployment, and security.

What's Inside

- Covers ASP.NET Core 2.0
- Dynamic page generation with the Razor templating engine
- Developing ASP.NET Core apps for non-Windows servers
- Clear, annotated examples in C#

Readers need intermediate experience with C# or a similar language.

Andrew Lock has been developing professionally with ASP.NET for the last seven years. His focus is currently on the ASP.NET Core framework.

To download their free eBook in PDF, ePub, and Kindle formats, owners of this book should visit manning.com/books/asp-net-core-in-action

Free eBook
See first page

"Comprehensive coverage of the latest and greatest .NET technology."
—Jason Pike
Atlas RFID Solutions

"A thorough and easy-to-read training guide to the future of Microsoft cross-platform web development."
—Mark Harris, Microsoft

"An outstanding presentation of the concepts and best practices. Explains not only what to do, but why to do it."
—Mark Elston, Advantest America

"Superb starting point for .NET Core 2.0 with valid and relevant real-world examples."
—George Onofrei, Devex

MANNING $49.99 / Can $65.99 [INCLUDING eBOOK]