ASP.NET Core - True Ultimate Guide

Section 21: Filters

Filters in ASP.NET Core MVC

Filters are powerful components in ASP.NET Core MVC that allow you to intercept and execute code before, after, or even around the execution of your controller actions or Razor Pages. They provide a clean and modular way to implement cross-cutting concerns like:

- **Authentication:** Verifying user identities before granting access to certain actions.
- **Authorization:** Checking if a user is authorized to perform a specific action.
- **Caching:** Caching responses to improve performance.
- **Error Handling:** Handling exceptions and generating appropriate error responses.
- **Logging:** Recording information about requests and responses.
- **Action Filtering:** Modifying action parameters or results.

Purpose of Filters

- **Encapsulation:** Filters encapsulate reusable logic that can be applied to multiple actions or controllers, reducing code duplication and promoting maintainability.
- **Clean Separation:** They help keep your controller actions focused on their core responsibility of handling requests and generating responses, while delegating cross-cutting concerns to filters.
- **Extensibility:** ASP.NET Core MVC provides a framework for creating custom filters tailored to your application's specific needs.

Best Practices

- **Choose the Right Filter Type:** Select the appropriate filter type (action, authorization, resource, exception, result) based on the desired interception point and the kind of logic you want to implement.
- **Dependency Injection (DI):** Leverage DI to inject services and dependencies into your filters, ensuring loose coupling and testability.
- **Keep Filters Small and Focused:** Each filter should have a single, well-defined responsibility. Avoid putting too much logic into a single filter.
- **Consider Performance:** Be mindful of the potential performance impact of filters, especially if you have many filters or complex logic within them.
- **Order Matters:** The order in which filters are executed is important. Understand the default order and how to customize it using the Order property if necessary.

Action Filters

Action filters, implemented using the IActionFilter interface, are invoked before and after an action method executes. They provide hooks where you can:

- **Pre-Action Logic (OnActionExecuting):**
    - Inspect and modify the action arguments or the ActionExecutingContext object.
    - Short-circuit the action execution (e.g., by setting the Result property of the context).
- **Post-Action Logic (OnActionExecuted):**
    - Inspect and modify the action result or the ActionExecutedContext object.
    - Log information about the action's execution.

ViewData in Action Filters

While not as common as using strongly typed models or view models, you can use the ViewData dictionary within action filters to pass additional data to your views.

- **Setting Values:** In the OnActionExecuting method, you can add data to the ViewData dictionary using context.Controller.ViewData["key"] = value.
- **Accessing in Views:** This data will then be available in the corresponding view.

Serilog Structured Logging in Filters

Serilog, with its emphasis on structured logging, integrates seamlessly with filters, allowing you to capture valuable contextual information about the action's execution.

- **Inject ILogger:** Use dependency injection to get an ILogger instance in your filter.
- **Log Messages:** Use _logger.LogInformation, _logger.LogWarning, etc., to log messages with structured data (key-value pairs).

Code Explanation

```
// PersonsListActionFilter.cs (Action Filter)

public class PersonsListActionFilter : IActionFilter

{
```

```
    private readonly ILogger<PersonsListActionFilter> _logger; // Injected logger


    public PersonsListActionFilter(ILogger<PersonsListActionFilter> logger)
    {
        _logger = logger;
    }


    public void OnActionExecuted(ActionExecutedContext context)
    {
        _logger.LogInformation("PersonsListActionFilter.OnActionExecuted method");
    }


    public void OnActionExecuting(ActionExecutingContext context)
    {
        _logger.LogInformation("PersonsListActionFilter.OnActionExecuting method");
    }
}
```

In this action filter:

1. **ILogger Injection:** The constructor receives an ILogger instance through dependency injection. The generic type parameter PersonsListActionFilter specifies the category name for the logger, allowing for targeted configuration.
2. **OnActionExecuting and OnActionExecuted:**
    - These methods are called before and after the action method executes, respectively.
    - In this simple example, they just log informational messages using the injected logger.

Applying the Filter

You can apply this filter in a few ways:

- **Globally:** Register the filter in Program.cs (or Startup.cs) to apply it to all controllers and actions in your application.

- **Controller-Level:** Apply the [PersonsListActionFilter] attribute to your controller class to apply it to all actions within that controller.
- **Action-Level:** Apply the [PersonsListActionFilter] attribute to specific action methods to filter only those actions.

Example Usage

```
// PersonsController.cs

[Route("[controller]")]

[PersonsListActionFilter] // Apply the filter to the entire controller

public class PersonsController : Controller

{

  // ... your actions ...

}
```

In this scenario, the PersonsListActionFilter will be executed before and after every action method within the PersonsController.

Remember that you can enhance this action filter by adding more meaningful logic to the OnActionExecuting and OnActionExecuted methods, such as:

- **OnActionExecuting:**
    - Checking authorization or authentication.
    - Validating or modifying action parameters.
    - Setting up logging context or other data.
- **OnActionExecuted:**
    - Logging the outcome of the action.
    - Modifying the action result (if needed).
    - Performing cleanup or other post-processing tasks.

Filter Arguments

Filters often require additional data or configuration to perform their tasks effectively. You can provide this information through *filter arguments*.

- **How to Pass Arguments:**
    - When applying a filter using the [TypeFilter] or [ServiceFilter] attributes, you can specify an Arguments property, which takes an array of objects.
- **Example:**

[TypeFilter(typeof(ResponseHeaderActionFilter), Arguments = new object[] { "My-Key-From-Action", "My-Value-From-Action", 1 })]

In this example, the ResponseHeaderActionFilter receives three arguments: a key, a value, and an order number.

Global Filters

Global filters are applied to all controllers and actions in your ASP.NET Core MVC application. They are useful for implementing cross-cutting concerns that need to be enforced consistently throughout your application.

- **How to Register Global Filters:**
    - In Program.cs (or Startup.cs in older versions), add your filter to the Filters collection within the MvcOptions configuration:
- builder.Services.AddControllersWithViews(options => {
- // ...
-
- // Registering a global filter
- options.Filters.Add(new ResponseHeaderActionFilter(logger, "My-Key-From-Global", "My-Value-From-Global", 2));

});

- **Example:**
    - A global authorization filter that requires users to be authenticated for all actions except those explicitly marked with [AllowAnonymous].
    - A global exception filter that logs all unhandled exceptions.

Custom Order of Filters: Controlling Execution Sequence

By default, ASP.NET Core executes filters in a predefined order. However, you can customize this order to suit your application's needs.

- **Order Property:**
    - Both the [TypeFilter] and [ServiceFilter] attributes have an Order property that you can use to specify the execution order of your filter.
    - Filters with lower Order values are executed first.
- **IOrderedFilter Interface:**
    - For more fine-grained control over filter ordering, implement the IOrderedFilter interface in your filter class.
    - This interface requires you to implement the Order property, which returns an integer value representing the filter's order.

Code Explanation

Let's break down the relevant parts of your code:

PersonsListActionFilter

```
// ...


public void OnActionExecuting(ActionExecutingContext context)
{
   // ...


   if (context.ActionArguments.ContainsKey("searchBy"))
   {
      string? searchBy = Convert.ToString(context.ActionArguments["searchBy"]);


      if (!string.IsNullOrEmpty(searchBy))
      {
         // ... (validation logic to ensure 'searchBy' is valid) ...
      }
   }
}
```

```
public void OnActionExecuted(ActionExecutedContext context)
{
    // ...

    // Accessing arguments passed to the action method using HttpContext.Items
    IDictionary<string, object?>? parameters = (IDictionary<string,
object?>?)context.HttpContext.Items["arguments"];

    if (parameters != null)
    {
        // ... (logic to populate ViewData with the action arguments)
    }

    // ... (populating ViewBag.SearchFields) ...
}
```

- **OnActionExecuting:**
    - It stores the ActionArguments (parameters passed to the action) in HttpContext.Items so they can be accessed later in OnActionExecuted.
    - It also validates the searchBy parameter to ensure it's one of the allowed values.
- **OnActionExecuted:**
    - Retrieves the action arguments from HttpContext.Items.
    - Populates ViewData with the values of the action arguments (if present).
    - Sets up ViewBag.SearchFields with a dictionary of search options.

ResponseHeaderActionFilter

```
public class ResponseHeaderActionFilter : IActionFilter, IOrderedFilter
{
    // ...

    public int Order { get; } // Implementing IOrderedFilter
```

```
    public ResponseHeaderActionFilter(ILogger<ResponseHeaderActionFilter> logger, string key, string value, int order)

    {

        // ...

        Order = order; // Set the order of the filter

    }


    // ... (OnActionExecuting and OnActionExecuted methods)

}
```

- **IOrderedFilter Implementation:** This filter implements IOrderedFilter, allowing you to explicitly control its execution order.
- **Order Property:** The Order property is set in the constructor and determines the filter's position in the execution sequence.

PersonsController

```
[Route("[controller]")]

[TypeFilter(typeof(ResponseHeaderActionFilter), Arguments = new object[] { "My-Key-From-Controller", "My-Value-From-Controller", 3 }, Order = 3)]

public class PersonsController : Controller

{

    // ...


    [Route("[action]")]

    [Route("/")]

    [TypeFilter(typeof(PersonsListActionFilter), Order = 4)] // Applied to Index action with Order = 4

    [TypeFilter(typeof(ResponseHeaderActionFilter), Arguments = new object[] { "MyKey-FromAction", "MyValue-From-Action", 1 }, Order = 1)]

    public async Task<IActionResult> Index(string searchBy, string? searchString, string sortBy = nameof(PersonResponse.PersonName), SortOrderOptions sortOrder = SortOrderOptions.ASC)


    {

        // ...
```

```
    }


    // ... (other actions) ...
}
```

- **Controller-level filter:** The ResponseHeaderActionFilter is applied at the controller level with Order = 3. It will be executed for all actions in this controller.
- **Action-level filters:** The Index action has two filters applied:
    - PersonsListActionFilter with Order = 4.
    - Another instance of ResponseHeaderActionFilter with Order = 1.

Filter Execution Order

For the Index action, the filters will be executed in the following order:

1. ResponseHeaderActionFilter (from action, Order = 1)
2. ResponseHeaderActionFilter (from controller, Order = 3)
3. PersonsListActionFilter (Order = 4)

Notes

- **ActionArguments vs. ViewData:**
    - ActionArguments represent the original input parameters to the action method.
    - ViewData is used to pass data from the controller (or filters) to the view.
- **HttpContext.Items:** This dictionary is a useful way to pass data between different middleware components or filters within the same request.
- **Serilog:** Use structured logging to capture valuable context information in your filters (e.g., action name, parameter values).
- **Testability:** Write unit tests for your filters to ensure they behave correctly in isolation.

Asynchronous Filters

In the realm of modern web development, asynchronous operations (e.g., database calls, API requests) are prevalent. ASP.NET Core MVC filters support asynchronous execution through the IAsyncActionFilter and IAsyncResultFilter interfaces. These interfaces allow you to perform asynchronous tasks within your filter logic, making your code more efficient and responsive.

- **IAsyncActionFilter:**
    - **Methods:**
        - OnActionExecutionAsync(ActionExecutingContext context, ActionExecutionDelegate next)
    - **Purpose:** Intercepts the action execution pipeline asynchronously.
    - **Notes:**
        - The next delegate now returns a Task that represents the execution of the rest of the filter pipeline and the action method.
        - You can await this task to wait for the subsequent operations to complete.
        - You can perform asynchronous tasks within the filter's logic and await their completion before or after calling next.

- **IAsyncResultFilter:**
    - **Methods:**
        - OnResultExecutionAsync(ResultExecutingContext context, ResultExecutionDelegate next)
    - **Purpose:** Intercepts the result execution pipeline asynchronously.
    - **Notes:**
        - Similar to IAsyncActionFilter, but it operates on the action result rather than the action itself.
        - You can use it for asynchronous tasks related to transforming or modifying the result before it's sent to the client.

Short-Circuiting Action Filters

Short-circuiting is a technique where an action filter can decide to terminate the filter pipeline early and directly return a result without invoking the action method or any subsequent filters. This can be useful for:

- **Validation:** If model validation fails, you can short-circuit and return a validation error response immediately.
- **Authentication/Authorization:** If a user isn't authenticated or authorized, you can short-circuit and return an appropriate response.
- **Caching:** If a cached response is available, you can short-circuit and return it directly.
- **How to Short-Circuit:**
    - In an IActionFilter, set the context.Result property to an IActionResult.
    - In an IAsyncResultFilter, set the context.Cancel property to true and provide a new context.Result.

Code Explanation: PersonCreateAndEditPostActionFilter

```
// PersonCreateAndEditPostActionFilter.cs (Async Action Filter)
public class PersonCreateAndEditPostActionFilter : IAsyncActionFilter
{
  // ... (_countriesService injection)

  public async Task OnActionExecutionAsync(ActionExecutingContext context,
ActionExecutionDelegate next)
  {
    // ... (potential before logic, not shown in this example)

    if (context.Controller is PersonsController personsController)
    {
      if (!personsController.ModelState.IsValid)
      {
        // ... (Prepare countries for ViewBag) ...
        // ... (Populate ViewBag.Errors with validation errors) ...

        var personRequest = context.ActionArguments["personRequest"];
        context.Result = personsController.View(personRequest); // Short-circuit
      }
      else
```

```
        {
            await next(); // Continue the pipeline if model is valid
        }
    }
    else
    {
        await next();
    }

    // ... (potential after logic, not shown in this example)
  }
}
```

In this code:

1. **IAsyncActionFilter Implementation:** The filter implements the IAsyncActionFilter interface for asynchronous operation.
2. **OnActionExecutionAsync:** This method is called asynchronously during the action execution pipeline.
3. **Controller Check:** It checks if the controller is of type PersonsController.
4. **Model State Validation:** If the ModelState is invalid, it prepares the necessary data for the view (ViewBag.Countries, ViewBag.Errors) and short-circuits the pipeline by setting context.Result to the View result with the original request data (personRequest).
5. **Continue Pipeline:** If the model state is valid or the controller is not PersonsController, it calls await next() to proceed with the action method and subsequent filters.

Notes

- **Async Filters:** Use IAsyncActionFilter or IAsyncResultFilter when you need to perform asynchronous tasks within your filters.
- **Short-Circuiting:** Use context.Result or context.Cancel to terminate the filter pipeline early if certain conditions are met.
- **HttpContext.Items:** This is a useful dictionary for passing data between middleware components or filters within the same request.
- **ActionArguments vs. ViewData:**
    - ActionArguments are the original input parameters passed to the action.
    - ViewData is used to pass additional data from the controller or filters to the view.

- **Testing:** Remember to write unit tests for your filters, including scenarios that test short-circuiting behavior.

Result Filters

Result filters, implemented by the IResultFilter or IAsyncResultFilter interfaces, are triggered just before and after the execution of an action result (the IActionResult returned by the action method). They give you the opportunity to:

- **Inspect and Modify the Result:** You can examine and potentially modify the result before it is sent to the client. This is useful for adding headers, transforming the result's content, or making other adjustments based on specific conditions.
- **Perform Post-Processing:** After the result has been executed (sent to the client), you can carry out tasks like logging or cleaning up resources.
- **IResultFilter:**
    - **Methods:** OnResultExecuting(ResultExecutingContext context) (before), OnResultExecuted(ResultExecutedContext context) (after)
    - **Synchronous execution:** Best suited for synchronous operations on the result.
- **IAsyncResultFilter:**
    - **Methods:** OnResultExecutionAsync(ResultExecutingContext context, ResultExecutionDelegate next)
    - **Asynchronous execution:** Enables asynchronous operations on the result using await.

Resource Filters

Resource filters, implementing IResourceFilter or IAsyncResourceFilter, are executed before and after model binding and action execution. They are ideal for tasks related to resource access and management.

- **Methods:**
    - IResourceFilter: OnResourceExecuting(ResourceExecutingContext context), OnResourceExecuted(ResourceExecutedContext context)

    - IAsyncResourceFilter: OnResourceExecutionAsync(ResourceExecutingContext context, ResourceExecutionDelegate next)
- **Common Use Cases:**
    - **Caching:** Implement caching logic to avoid unnecessary action execution.
    - **Performance Monitoring:** Measure the execution time of actions.
    - **Resource Cleanup:** Release resources acquired during action execution.

Authorization Filters: Guarding Your Actions

Authorization filters, implementing the IAuthorizationFilter interface, are responsible for determining whether a user is allowed to access a particular action method. They run before model binding and action execution.

- **Method:** OnAuthorization(AuthorizationFilterContext context)
- **Purpose:** Check authentication and authorization policies.
- **Short-Circuiting:** If authorization fails, you typically set context.Result to an appropriate result (e.g., 401 Unauthorized or a redirect to the login page).

Exception Filters

Exception filters, implemented through the IExceptionFilter interface, handle exceptions thrown during the execution of action methods or other filters.

- **Method:** OnException(ExceptionContext context)
- **Purpose:** Log exceptions, create custom error responses, or perform other error-handling tasks.
- **Short-Circuiting:** By setting context.ExceptionHandled = true and providing a new context.Result, you can prevent the exception from propagating further and control the error response sent to the client.

Impact of Short-Circuiting

- **Action Filters:** Short-circuiting an action filter prevents the action method and any subsequent action filters from executing. Control is transferred directly to the result execution pipeline.
- **Result Filters:** Short-circuiting a result filter bypasses the execution of any subsequent result filters. Control is returned to the previous filter or the MVC framework.

- **Resource Filters:** Short-circuiting a resource filter prevents the action method, action filters, and result filters from executing. The response is generated based on the IActionResult set in the context.Result.
- **Authorization Filters:** Short-circuiting an authorization filter prevents all subsequent steps in the pipeline, including model binding, action execution, and result execution.

Code Explanation

Let's analyze the filters in your code:

1. **TokenAuthorizationFilter:**
    - An authorization filter that checks for the presence and validity of an "Auth-Key" cookie.
    - If the cookie is missing or invalid, it short-circuits the pipeline and returns a 401 Unauthorized status code.
2. **HandleExceptionFilter:**
    - An exception filter that logs errors and, in development environments, returns a ContentResult with the exception message and a 500 Internal Server Error status code.
3. **FeatureDisabledResourceFilter:**
    - A resource filter that checks if a feature is disabled.
    - If disabled, it short-circuits the pipeline and returns a 501 Not Implemented status code.
4. **PersonsListResultFilter:**
    - A result filter that adds a "Last-Modified" header to the response after the action result has been executed.
5. **TokenResultFilter:**
    - A result filter that appends an "Auth-Key" cookie to the response before it is sent to the client.

Notes

- **Filter Types:** Understand the different filter types and when to use each one.
- **Async Filters:** Use async filters for asynchronous operations within your filter logic.
- **Short-Circuiting:** Use this technique to control the filter pipeline and return early responses.
- **Order of Execution:** Be aware of the default filter execution order and how to customize it using the Order property or IOrderedFilter.
- **Dependency Injection:** Utilize DI to inject services and dependencies into your filters.

IAlwaysRunResultFilter

The IAlwaysRunResultFilter interface inherits from the standard IResultFilter interface and introduces a crucial distinction: its methods (OnResultExecuting and OnResultExecuted) are **guaranteed to execute** even if another filter in the pipeline short-circuits the result. This makes it an indispensable tool for scenarios where you need to perform actions or modifications on the response regardless of whether the action method or other filters completed successfully.

Notes:

- **Inheritance:** IAlwaysRunResultFilter extends IResultFilter, inheriting its two methods:
    - OnResultExecuting(ResultExecutingContext context): Called before the action result is executed.
    - OnResultExecuted(ResultExecutedContext context): Called after the action result has been executed.
- **Guaranteed Execution:**
    - The core feature of IAlwaysRunResultFilter is that its methods are always invoked, even if:
        - An exception occurs in the action method or a previous filter.
        - Another filter short-circuits the pipeline by setting context.Cancel = true and providing a new context.Result.
        - The action method itself returns a result that short-circuits the pipeline (e.g., return new EmptyResult();).
- **Use Cases:**
    - **Logging:** Log the final outcome of the request, including any exceptions or short-circuited results.
    - **Response Modification:** Apply modifications to the response headers or content, even if the action was not executed.
    - **Cleanup:** Perform essential cleanup tasks or release resources, regardless of the action's success or failure.

Code Explanation

```
// PersonAlwaysRunResultFilter.cs
public class PersonAlwaysRunResultFilter : IAlwaysRunResultFilter
{
   // (You might inject dependencies here if needed)

   public void OnResultExecuted(ResultExecutedContext context)
   {
      // Logic to execute after the result has been executed (sent to the client)
   }

   public void OnResultExecuting(ResultExecutingContext context)
   {
      // Logic to execute before the result is executed
   }
}
```

- **Empty Implementation:** In this example, the OnResultExecuted and OnResultExecuting methods are empty. You would replace these placeholders with your actual filter logic.

Applying the Filter

```
// PersonsController.cs (HttpPost Edit action)
[HttpPost]
[Route("[action]/{personID}")]
[TypeFilter(typeof(PersonCreateAndEditPostActionFilter))]
[TypeFilter(typeof(TokenAuthorizationFilter))]
[TypeFilter(typeof(PersonAlwaysRunResultFilter))]

// Apply the filter
public async Task<IActionResult> Edit(PersonUpdateRequest personRequest)
{
  // ...
}
```

The PersonAlwaysRunResultFilter is applied to the Edit (POST) action method using the [TypeFilter] attribute.

Example Scenarios

1. **Logging the Final Response:** Even if an exception occurs within the action method or a previous filter, the OnResultExecuted method of PersonAlwaysRunResultFilter will still be called, allowing you to log the final status code and any error details.
2. **Adding Headers:** You could use OnResultExecuting to add custom headers to the response, ensuring they are included even if the action is short-circuited.
3. **Resource Cleanup:** If your action acquires resources (e.g., database connections, file handles), you can use OnResultExecuted to release them reliably, even if an exception occurs.

Caveats

- **Can't Change the Result:** While IAlwaysRunResultFilter guarantees execution, it cannot change the IActionResult once it has been set by another filter or the action method itself. Its primary purpose is to observe or modify the response or perform cleanup tasks.
- **Order:** IAlwaysRunResultFilter is executed at the very end of the result filter pipeline, after all other result filters, regardless of their specified order.

Filter Overrides

In ASP.NET Core MVC, filter overrides empower you to selectively disable or alter the behavior of filters on a per-action or per-controller basis. This granular control is essential for situations where you need to:

- **Bypass a Filter:** Skip the execution of a specific filter for a particular action or controller.
- **Modify Filter Behavior:** Change the arguments or settings of a filter for a specific action or controller.

Key Mechanisms for Filter Overrides

1. **[NonAction] Attribute:**
    - **Purpose:** Prevents a method from being treated as an action method by the MVC framework. This effectively bypasses all filters (action filters, result filters, etc.) that would otherwise be applied to the method.
    - **Usage:** Apply this attribute to methods within your controller that you don't want to be considered action methods.

2. **[SkipFilter] Attribute (Custom):**
    - **Purpose:** A custom attribute that allows you to skip the execution of a specific filter or a group of filters for a particular action or controller.
    - **Implementation:** In your code example, the SkipFilter attribute implements the IFilterMetadata interface, which is a marker interface used to identify filter attributes.
3. **Overriding Filter Arguments:**
    - **Purpose:** Modify the arguments passed to a filter for a specific action or controller.
    - **Mechanism:** When applying the filter attribute, provide the updated arguments.

Code Explanation

SkipFilter Attribute

```
// SkipFilter.cs

public class SkipFilter : Attribute, IFilterMetadata

{

}
```

This simple attribute acts as a marker that you can apply to actions or controllers to indicate that you want to skip specific filters.

PersonAlwaysRunResultFilter with Override Logic

```
// PersonAlwaysRunResultFilter.cs

public void OnResultExecuting(ResultExecutingContext context)

{

  if (context.Filters.OfType<SkipFilter>().Any())

  {

    return; // Skip this filter if the SkipFilter attribute is present

  }


  // ... Your filter logic here ...

}
```

In this modified filter:

1. **Check for SkipFilter:** The OnResultExecuting method checks if the context.Filters collection contains any instances of the SkipFilter attribute.

2. **Skip if Present:** If SkipFilter is found, the filter's logic is bypassed by simply returning from the method.
3. **Execute Otherwise:** If SkipFilter is not present, the filter proceeds with its normal execution.

Applying the SkipFilter Attribute

```
// In your controller

[SkipFilter] // Skip specific filters for this action

public IActionResult SomeAction()

{

   // ...

}
```

By applying the SkipFilter attribute to an action method, you instruct any filters that check for this attribute (like the modified PersonAlwaysRunResultFilter) to bypass their logic for that specific action.

Notes

- **[NonAction]:** Use to completely exclude a method from being treated as an action.
- **Custom Attributes:** Create custom attributes like SkipFilter to provide more fine-grained control over filter execution.
- **Override Arguments:** Modify filter behavior by providing different arguments when applying the filter attribute.
- **Flexibility:** Filter overrides allow you to adapt your filter pipeline to specific actions or controllers, enhancing your control and customization options.

Service Filters

Service filters, applied using the [ServiceFilter] attribute, offer a powerful mechanism for injecting dependencies directly into your filters. This is essential when your filters require access to services like loggers, configuration settings, or data repositories to perform their tasks effectively.

- **[ServiceFilter] Attribute:**
    - **Purpose:** Instructs ASP.NET Core MVC to resolve the filter instance from the dependency injection (DI) container.
    - **Usage:** Apply this attribute to controllers or actions, specifying the type of the filter you want to inject.
    - **Benefits:**
        - Promotes loose coupling and testability by allowing filters to receive their dependencies through constructor injection.
        - Enables you to manage filter lifetimes (transient, scoped, singleton) using the DI container.

Filter Attribute Classes: Encapsulating Filter Metadata

Filter attribute classes serve as a convenient way to package metadata about a filter and apply it declaratively to controllers or actions. They typically inherit from the Attribute class and implement one or more filter interfaces (e.g., IActionFilter, IAuthorizationFilter, etc.).

- **Purpose:**
    - Provide a concise and readable way to apply filters.
    - Encapsulate filter-specific settings or configurations within the attribute's properties.
- **Example:**

```
public class MyActionFilterAttribute : Attribute, IActionFilter
{
   // Filter properties (e.g., configuration settings)
   public string SomeSetting { get; set; }

   // ... implementation of IActionFilter methods ...
}
```

IFilterFactory Interface

The IFilterFactory interface allows you to create filter instances dynamically at runtime, offering flexibility and customization beyond what's possible with simple attribute classes.

- **IsReusable Property:** Indicates whether the created filter instance can be reused across multiple requests (if true) or should be a new instance for each request (if false).
- **CreateInstance(IServiceProvider serviceProvider) Method:** This method is responsible for creating the actual filter instance. It receives an IServiceProvider which you can use to resolve dependencies from the DI container.

Code Explanation

Let's break down the relevant filter-related code in your examples:

PersonsListActionFilter

- This is a standard action filter (IActionFilter) that performs some logic before and after the action method executes.
- It utilizes an ILogger (injected through the constructor) for logging and modifies the ViewData in the OnActionExecuted method.

ResponseHeaderFilterFactoryAttribute and ResponseHeaderActionFilter

- **ResponseHeaderFilterFactoryAttribute:**
    - Implements the IFilterFactory interface.
    - IsReusable is set to false, meaning a new filter instance will be created for each request.
    - The CreateInstance method:
        1. Resolves the ResponseHeaderActionFilter from the DI container.
        2. Sets the Key, Value, and Order properties of the filter based on the attribute's constructor arguments.
        3. Returns the configured filter instance.
- **ResponseHeaderActionFilter:**
    - An async action filter (IAsyncActionFilter) that adds a custom header to the response after the action method executes.
    - Its Key, Value, and Order properties are set by the ResponseHeaderFilterFactoryAttribute.

PersonsController (Filter Application)

```
[Route("[action]")]

[Route("/")]

[ServiceFilter(typeof(PersonsListActionFilter), Order = 4)]

[ResponseHeaderFilterFactory("MyKey-FromAction", "MyValue-FromAction",  1)] // Using the filter factory

[TypeFilter(typeof(PersonsListResultFilter))]

[SkipFilter]

public async Task<IActionResult> Index(string searchBy, string? searchString, string sortBy = nameof(PersonResponse.PersonName),  SortOrderOptions sortOrder = SortOrderOptions.ASC)

{

   // ...

}
```

In the Index action:

- [ServiceFilter(typeof(PersonsListActionFilter))]: Applies the PersonsListActionFilter, resolving it from the DI container.
- [ResponseHeaderFilterFactory(...)]: Uses the filter factory to create and apply an instance of ResponseHeaderActionFilter with the specified arguments.

- [SkipFilter]: This custom attribute is used to skip specific filters that check for its presence (as demonstrated in a previous example).

Notes

- **ServiceFilter:** Use for filters that require dependencies from the DI container.
- **Filter Attribute Classes:** A convenient way to encapsulate filter metadata and apply filters declaratively.
- **IFilterFactory:** Enables dynamic filter creation and customization.
- **IsReusable:** Control whether filter instances are reused or created anew for each request.
- **CreateInstance:** Implement this method to create and configure your filter instances.

Key Points to Remember

Filters - Core Concepts

- **Purpose:** Intercept and execute code before, after, or around controller actions or Razor Pages.
- **Benefits:**
    - Modularize cross-cutting concerns (auth, logging, caching, etc.)
    - Clean separation of concerns
    - Reusability
    - Extensibility
- **Types:**
    - IActionFilter: Before and after action execution.
    - IAuthorizationFilter: Authorization checks before action execution.
    - IResourceFilter: Before and after model binding and action execution.
    - IExceptionFilter: Handles exceptions.
    - IResultFilter: Before and after action result execution.

Key Filter Interfaces and Attributes

- **IActionFilter, IAsyncResultFilter:** Intercept action execution (sync and async).
- **IAuthorizationFilter:** Perform authorization checks.
- **IResourceFilter, IAsyncResourceFilter:** Manage resource access and lifecycle (sync and async).
- **IExceptionFilter:** Handle exceptions gracefully.
- **IResultFilter, IAsyncResultFilter:** Work with the action result (sync and async).
- **IAlwaysRunResultFilter:** Guaranteed execution, even if other filters short-circuit.
- **[TypeFilter]:** Apply a filter by its type, with optional arguments.
- **[ServiceFilter]:** Resolve a filter from the DI container.

Filter Overrides

- **[NonAction]:** Exclude a method from being treated as an action (bypasses filters).
- **Custom Attributes:** Create your own attributes to skip or modify filter behavior.

Filter Factories

- **IFilterFactory Interface:** Create filter instances dynamically at runtime.
- **IsReusable Property:** Control whether the filter instance can be reused.
- **CreateInstance Method:** Implement to create and configure filter instances.

Short-Circuiting

- **Action Filters:** Set context.Result to bypass the action and subsequent filters.
- **Result Filters:** Set context.Cancel = true and provide a new context.Result.
- **Resource Filters:** Set context.Result to bypass action execution and result filters.
- **Authorization Filters:** Set context.Result to bypass the entire pipeline (including model binding).

Best Practices

- **Choose the Right Filter:** Select the appropriate filter type for your needs.
- **Dependency Injection:** Use DI to inject services into your filters.
- **Keep Filters Small and Focused:** Each filter should have a single responsibility.
- **Performance:** Be mindful of the potential performance impact of filters.
- **Order Matters:** Understand the default filter execution order and how to customize it.

Interview Tips

- **Explain Filter Types:** Clearly articulate the purpose and use cases for each filter type.
- **Short-Circuiting:** Demonstrate how and when to use short-circuiting effectively.
- **Custom Filters:** Be prepared to discuss scenarios where you would create custom filters and how you would implement them.
- **Filter Ordering:** Explain the default order and how to customize it if needed.
- **Best Practices:** Showcase your knowledge of filter best practices and common pitfalls.