

In ASP.NET Core, filters are methods that execute before and after the action method execution. They run as part of the filter pipeline, which executes as soon as the action selector runs. First, there is a request pipeline that executes a set of middleware one by one, as we have seen in the middleware section. One of these middleware components is the routing middleware. Whenever the routing middleware detects that the incoming URL matches a particular route, it enters the MVC processing flow. It first tries to pick the appropriate action based on the incoming route. For example, if the URL is /home, it may pick up the Index action method in the HomeController. As soon as the routing middleware selects a specific action method, it executes the filter pipeline. The filter pipeline is a collection of filters. Some filters execute before the action method, while others execute after.

Types of Filters

There are multiple types of filters, including:

- Authorization Filters
- Resource Filters
- Action Filters
- Exception Filters
- Result Filters

Some filters contain two types of methods—one that runs before the action method and another that runs after.

For example, resource filters have two methods:

- OnResourceExecuting (executes before the action method)
- OnResourceExecuted (executes after the action method)

Similarly, action filters have:

- OnActionExecuting (executes before the action method)
- OnActionExecuted (executes after the action method)

Result filters, such as OnResultExecuting and OnResultExecuted, both execute after the action method.

However, some filters have only one method:

- Authorization Filter → OnAuthorization
- Exception Filter → OnException

Why Use Filters?

Different types of filters exist for different purposes, depending on when they execute and their intended function.

Microsoft designed these filters to handle various tasks within the MVC model-view-controller structure.

For example:

- Authorization Filters → Verify user permissions for accessing resources.
- Action Filters → Manipulate action method parameters (reading, checking, validating).
- Other Filters → Modify request/response headers, caching, and model state validation.

Each filter serves a unique role in preparing the response sent to the browser.

Developer Perspective

As a developer, understanding the different types of filters, their execution flow, and their use cases is crucial. You need to know:

- When each filter executes
- How it executes
- What operations it performs

For instance, understanding what you can achieve with an authorization filter versus an action filter is essential.

Additionally, knowing best practices for using filters ensures optimal application performance and maintainability.

What's Next?

In the next lecture, we will focus on **action filters** first. After that, we will explore other filter types, such as authorization and resource filters, one by one.



Action Filters in ASP.NET Core

In this lecture, let's focus on **action filters**.

As we have seen, the entire filter pipeline can be a bit confusing. However, let's narrow our focus to action filters.

Execution Flow of Action Filters

Action filters contain two types of methods:

1. OnActionExecuting
2. OnActionExecuted

The sequence of execution is as follows:

- Once the **routing middleware** selects a particular action method in a controller, it determines which action method needs to be executed.
- Before executing the action method, the OnActionExecuting method runs.
- After the action method completes execution (i.e., when a response is returned), the OnActionExecuted method runs.

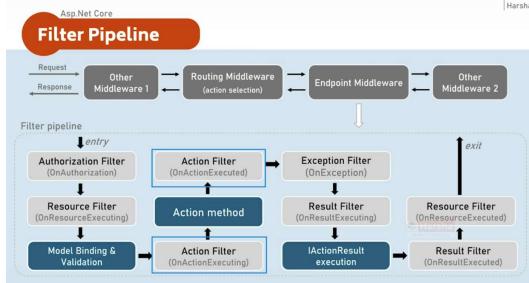
You can define both OnActionExecuting and OnActionExecuted in a single **Action Filter class**.

Understanding the Action Filter Class

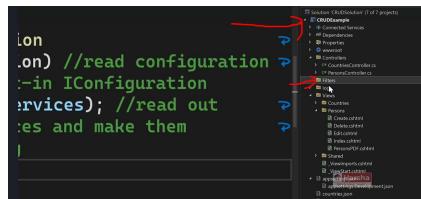
An **action filter class** contains logic for handling action method execution before and after it runs.

However, before action filters execute, other filters—such as **resource filters**, **authorization filters**, and **model binding**—take place. We will discuss these later.

For now, let's focus on **implementing an action filter** in this lecture.



Filter concept is completely related to this asp.net core project. It is not related to repository layer or service layer.



```
1  using Microsoft.AspNetCore.Mvc.Filters;
2
3  namespace CRUDExample.Filters.ActionFilters
4  {
5      //we want to apply this action Filter to the PersonList action method in the HomeController
6      public class PersonListActionFilter : IActionFilter
7      {
8          public void OnActionExecuted(ActionExecutedContext context)
9          {
10              throw new NotImplementedException();
11          }
12
13          public void OnActionExecuting(ActionExecutingContext context)
14          {
15              throw new NotImplementedException();
16          }
17
18      }
19  }
```

```
namespace CRUDExample.Filters.ActionFilters
{
    //we want to apply this action filter to the PersonsList action method in the HomeController
    public class PersonsListActionFilter : IActionFilter
    {
        private readonly ILogger<PersonsListActionFilter> _logger;
        public PersonsListActionFilter(ILogger<PersonsListActionFilter> logger)
        {
            _logger = logger;
        }

        //To do: add before logic here
        public void OnActionExecuted(ActionExecutedContext context)
        {
            _logger.LogInformation("PersonsListActionFilter: OnActionExecuted method");
        }

        //To do: add before logic here
        public void OnActionExecuting(ActionExecutingContext context)
        {
            _logger.LogInformation("PersonsListActionFilter: OnActionExecuting method");
        }
    }
}
```

Next, we have to add this filter to a particular action method where we want to execute

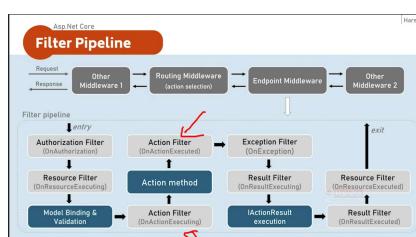
```
    public void testGetPersonList() {
        // Given
        PersonSearchCriteria personSearchCriteria = new PersonSearchCriteria();
        personSearchCriteria.setPersonName("John Doe");
        personSearchCriteria.setPersonEmail("john.doe@example.com");
        personSearchCriteria.setPersonMobile("1234567890");
        personSearchCriteria.setPersonGender("Male");
        personSearchCriteria.setPersonBirthDate("1990-01-01");
        logger = logger();
        // When
        List<PersonResponse> persons = await _personSearcherService.getFilteredPersons(personSearchCriteria);
        // Then
        assertEquals(1, persons.size());
        assertEquals("John Doe", persons.get(0).getPersonName());
        assertEquals("john.doe@example.com", persons.get(0).getPersonEmail());
        assertEquals("1234567890", persons.get(0).getPersonMobile());
        assertEquals("Male", persons.get(0).getPersonGender());
        assertEquals("1990-01-01", persons.get(0).getPersonBirthDate());
    }
}
```

Now, let's add breakpoint to our controller and filer and understand the flow of execution.

```
24     _countriesService = countriesService;
25     _logger = logger;
26   }
27   /**
28    * @url persons/index
29    * @param [PersonFilter] filter
30    * @route("/")
31    * @typeFilter([typeof(PersonsListActionFilter)])
32   public async Task<ActionResult<Index>�新型> Index(string searchBy,
33   string? searchString, string sortBy = nameof(
34   PersonResponse.PersonName), SortOrderOptions sortOrder =
35   SortOrderOptions.ASC)
36   {
37     _logger.LogInformation("Index action method of PersonsController");
38     logger.LogInformation($"----- {searchString} -----");
39     logger.LogInformation("----- adding a breakpoint in the index action -----");
40   }
```

```
    public void OnActionExecuting(ActionExecutingContext context)
    {
        // To do: add after logic here
        _logger.LogInformation
        ("PersonsListActionFilter.OnActionExecuting method");
    }
}

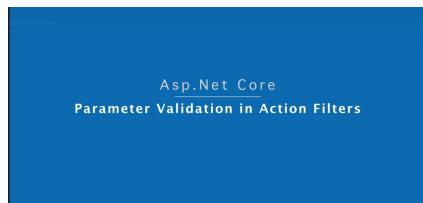
[Filter]
public void OnActionExecuted(ActionExecutedContext context)
```



Open SEQ and see the log.

But what's the purpose of the [action filter](#)?

The screenshot shows the 'Action Filters' section of the Asp.NET Core documentation. It contains two main sections: 'OnActionExecuting' method and 'OnActionExecuted' method. The 'OnActionExecuting' method is described as running immediately before an action method executes, with bullet points: 'It can access the action method parameters, read them & do necessary manipulations on them.', 'It can validate action method parameters.', and 'It can short-circuit the action (prevent action method from execution) and return a different IActionResult.' The 'OnActionExecuted' method is described as running immediately after an action method executes, with bullet points: 'It can manipulate the ViewData.', 'It can change the result returned from the action method.', and 'It can throw exceptions to either return the exception to the exception filter (if exists); or return the error response to the browser.'



We are in the process of understanding the practical use cases of action filters. One of the common tasks that can be performed with action filters is that you can read the action method parameters and validate them before the actual action method execution. For example, in this Index action method of the PersonsController, we have parameters like searchBy, sortBy, etc. There is a need to validate these parameter values to ensure the user supplies the correct values. So, it would be better if we validate these parameters before they reach the actual action method.

```
//URL: persons/index  
[Route("action")]  
[Route("")]  
[TypeFilter(typeof(PersonsListActionFilter))]  
public async Task<IActionResult> Index(string searchBy,  
string? searchString, string sortBy = nameof(  
(PersonResponse.PersonName), SortOrderOptions sortOrder =  
SortOrderOptions.ASC)  
{  
    _logger.LogInformation("Index action method of  
    PersonsController");  
    _logger.LogDebug($"searchBy: {searchBy}, searchString:  
    {searchString}, sortBy: {sortBy}, sortOrder:  
    {sortOrder}");  
    //actual action method
```

Using this 'context', you can access current 'action' name, controller name, the action method parameters, you can short circuit by giving a result etc.

```
78     public void OnActionExecuting(ActionExecutingContext context) =  
79     {  
80         //To do: add before logic here  
81         _logger.LogInformation("PersonsListActionFilter.OnActionExecuting method");  
82         context.  
83     }
```

```
1  //In all action logic  
2  public void OnActionExecuting(ActionExecutingContext context)  
3  {  
4      _logger.LogInformation("PersonsListActionFilter.OnActionExecuting method");  
5      if(context.ActionArguments.Contains("sortBy"))  
6      {  
7          string? sortBy = Convert.ToString(context.ActionArguments["sortBy"]);  
8          //Validate the sortBy parameter value  
9          if (string.IsNullOrEmpty(sortBy))  
10          {  
11              context.Response.StatusCode = 400;  
12              context.Response.ContentType = "text/html";  
13              context.Response.ContentLength = 0;  
14              context.Response.OutputStream.Close();  
15              context.Response.Close();  
16              context.Response.AddError("Sort By parameter is required");  
17          }  
18          //Count the sortBy parameter value  
19          if (Convert.ToInt32(context.ActionArguments["sortBy"]) != sortBy)  
20          {  
21              _logger.LogWarning("sortBy actual value ({sortBy}), sortBy: {sortBy}");  
22              context.ActionArguments["sortBy"] = Convert.ToString(PersonResponse.PersonName);  
23              _logger.LogInformation("sortBy updated value ({sortBy}), sortBy: {sortBy}");  
24          }  
25      }  
26  }
```

Let's try to send some wrong value.

```
localhost:5298/Persons/Index?searchBy=abcdPersonName&searchString=  
localhost:5298/Persons/Index?searchBy=abcdPersonName&searchString= - Google Search
```

Solutions

Person [Download as PDF](#) [Download as CSV](#) [Download as](#)

A screenshot of a search interface. At the top, there are links for 'Person', 'Download as PDF', 'Download as CSV', and 'Download as'. Below that is a search bar with the placeholder 'Person Name'. A dropdown menu is open over the search bar, showing the option 'Person Name'. At the bottom left, it says 'Now debug'.


```

21     public void OnActionExecuting(ActionExecutingContext context)
22     {
23         //To do: add before logic here
24         logger.LogInformation("PersonListActionFilter.OnActionExecuting method");
25     }
26     if (context.ActionArguments.ContainsKey("searchBy"))
27     {
28         string? searchBy = Convert.ToString(context.ActionArguments["searchBy"]);
29         //validate the searchBy parameter value
30         if (!string.IsNullOrEmpty(searchBy))
31             Harsha

```

```

//To do: add before logic here
public void OnActionExecuting(ActionExecutingContext context)
{
    logger.LogInformation("PersonListActionFilter.OnActionExecuting method");
    if (context.ActionArguments.ContainsKey("searchBy"))
    {
        string? searchBy = Convert.ToString(context.ActionArguments["searchBy"]);
        //validate the searchBy parameter value
        if (!string.IsNullOrEmpty(searchBy))
        {
            var searchByOptions = new List<string>()
            {
                nameof(PersonResponse.Name),
                nameof(PersonResponse.Email),
                nameof(PersonResponse.DateOfBirth),
                nameof(PersonResponse.Gender),
                nameof(PersonResponse.Address),
                nameof(PersonResponse.CountryID),
                nameof(PersonResponse.StateID)
            };
            //reset the searchBy parameter value if searchBy is not in the list
            if (!searchByOptions.Contains(searchBy))
            {
                context.ActionArguments["searchBy"] = nameof(PersonResponse.PersonName);
                logger.LogWarning("searchBy parameter value {searchBy} is invalid", searchBy);
            }
        }
    }
}

```



```

//To do: add before logic here
public void OnActionExecuted(ActionExecutedContext context)
{
    logger.LogInformation("PersonListActionFilter.OnActionExecuted method");
    PersonController personController = (PersonController)context.Controller;
    IDictionary<string, object?> parameters = (IDictionary<string, object?>)context.HttpContext.Items["arguments"];
    if (parameters != null)
    {
        if (parameters.ContainsKey("searchBy"))
        {
            personController.ViewData["searchBy"] = Convert.ToString(parameters["searchBy"]);
        }
        if (parameters.ContainsKey("searchString"))
        {
            personController.ViewData["searchString"] = Convert.ToString(parameters["searchString"]);
        }
        if (parameters.ContainsKey("sortBy"))
        {
            personController.ViewData["sortBy"] = Convert.ToString(parameters["sortBy"]);
        }
        if (parameters.ContainsKey("sortOrder"))
        {
            personController.ViewData["sortOrder"] = Convert.ToString(parameters["sortOrder"]);
        }
    }
}

```

```

//([Index])
//([Person])
//([Sort])
//([Search])
public async Task<ActionResult> Index([Bind("searchBy", "searchString", "sortBy", "sortOrder")])
{
    logger.LogInformation("Index action method of PersonController");
    logger.LogDebug("Index, {searchBy}, {searchString}, {sortBy}, {sortOrder}", searchBy, searchString, sortBy, sortOrder);
    //Search
    ViewData["SortOrderField"] = new Dictionary<string, string>
    {
        {nameof(PersonResponse.Name), "Person Name"}, 
        {nameof(PersonResponse.Email), "Email"}, 
        {nameof(PersonResponse.DateOfBirth), "Date of Birth"}, 
        {nameof(PersonResponse.Gender), "Gender"}, 
        {nameof(PersonResponse.CountryID), "Country"}, 
        {nameof(PersonResponse.Address), "Address"}
    };
    List<PersonResponse> persons = await _personService.GetFilteredPersons(searchBy, searchString);
    //ViewBag CurrentSortBy = searchBy;
    //ViewBag CurrentSearchString = searchString;
    PersonResponse sortedPersons = await _personService.GetSortedPersons(persons, sortBy, sortOrder);
    ViewBag.CurrentSortBy = sortBy;
    ViewBag.CurrentSearchString = searchString;
    return View(sortedPersons); //Views/Persons/Index.cshtml
}

/// <summary>

```

Let's run our application and see the execution sequences.

```

//([Index])
//([Person])
//([Sort])
//([Search])
public async Task<ActionResult> Index([Bind("searchBy", "searchString", "sortBy", "sortOrder")])
{
    logger.LogInformation("Index action method of PersonController");
    logger.LogDebug("Index, {searchBy}, {searchString}, {sortBy}, {sortOrder}", searchBy, searchString, sortBy, sortOrder);
    ViewData["SortOrderField"] = new Dictionary<string, string>
    {
        {nameof(PersonResponse.Name), "Person Name"}, 
        {nameof(PersonResponse.Email), "Email"}, 
        {nameof(PersonResponse.DateOfBirth), "Date of Birth"}, 
        {nameof(PersonResponse.Gender), "Gender"}, 
        {nameof(PersonResponse.CountryID), "Country"}, 
        {nameof(PersonResponse.Address), "Address"}
    };
    List<PersonResponse> persons = await _personService.GetFilteredPersons(searchBy, searchString);
    //ViewBag CurrentSortBy = searchBy;
    //ViewBag CurrentSearchString = searchString;
    PersonResponse sortedPersons = await _personService.GetSortedPersons(persons, sortBy, sortOrder);
    //ViewBag CurrentSortBy = sortBy;
    //ViewBag CurrentSearchString = searchString;
    return View(sortedPersons); //Views/Persons/Index.cshtml
}

/// <summary>

```







It is not advisable to write this kind of static log messages in the Serilog.
Always try to structure your logs.
That means, if you find any searchable keywords in your log message, you should supply them as parameters.
Let me show an example.

```

11  public PersonsListActionFilter(
12    ILogger<PersonsListActionFilter> logger)
13  {
14    _logger = logger;
15  }
16
17  public void OnActionExecuted(ActionExecutedContext context)
18  {
19    //To do: add after logic here
20    _logger.LogInformation(
21      ("PersonsListActionFilter.OnActionExecuted method");
22
23    PersonsController personsController = (PersonsController) context.Controller;
24
25    IDictionary<string, object?>? parameters =
26    (IDictionary<string, object?>?) context.HttpContext.Items[<!-- arguments -->];
  
```

For Example in this log message if we can notice, this is the name of the class and this is the name of the method.

```

1  public void OnActionExecuted(ActionExecutedContext context)
2
3  //To do: add after logic here
4  _logger.LogInformation(
5    ("PersonsListActionfilter.OnActionExecuted method");
6
7  PersonsController personsController = (PersonsController) context.Controller;
8
9  IDictionary<string, object?>? parameters = [Harsha]
  
```

Now based on this parameter name, we can search

```

13  _logger = logger;
14
15
16  public void OnActionExecuted(ActionExecutedContext context)
17  {
18    //To do: add after logic here
19    _logger.LogInformation("FilterName.{MethodName} method",
20      nameof(PersonsListActionfilter), nameof(OnActionExecuted));
21
22    PersonsController personsController = (PersonsController) context.Controller;
23
  
```

```

//To do: add before logic here
public void OnActionExecuting(ActionExecutingContext context)
{
  context.HttpContext.Items["arguments"] = context.ActionArguments;

  _logger.LogInformation("FilterName.{MethodName} method",nameof(PersonsListActionFilter), nameof(OnActionExecuting));

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

    //Validate the searchby parameter value
    if (!string.IsNullOrEmpty(searchby))
  
```

Now, open SEQ server. Instead of looking into the huge log messages, I would like to see only the log messages from action filter.



| 07 Mar 2025 08:58:15:96 | PersonalListActionFilter.OnActionExecuted method |
|--------------------------|---|
| Event ▾ | Leaf (Information) ▾ |
| | Type (OnActionExecuting) ▾ |
| ✓ x ActionId | d70094f-89af-47d9-b657-899c1ed89979 |
| ✓ x ActionName | CRUDDeepApp.Controllers.PersonController.Index |
| ✓ x ApplicationName | CRUD Deep App |
| ✓ x ConnectionId | 0e0448c57807 |
| ✓ x FilterType | OnActionExecutionFilter |
| ✓ x MethodName | OnActionExecuted |
| ✓ x RequestId | 0e0448c57807:00000001 |
| ✓ x RequestPath | CRUDsample.Filters.ActionFilters.PersonListActionFilter |
| ✓ x SourceObject | PersonalListActionFilter.OnActionExecuted |
| 07 Mar 2025 08:58:15:943 | PersonalListActionFilter.OnActionExecuting method |



When it runs

You can supply an array of arguments that will be supplied as constructor arguments of the filter class.

```
[TypeFilter(typeof(FilterClassName),
    Arguments = new object[] { arg1, arg2 })]
public IActionResult ActionMethod()
```

How to send arguments in controller

How to receive arguments in filter's constructor

```
public FilterClassName(IService service, type param1, type param2)
```

Generally, you invoke the filter class using the `TypeFilter` attribute just above the action method. While applying or attaching the filter to an action method, you can optionally supply argument values to that filter. These values are received by the constructor of the filter.

Why Should You Supply Argument Values?

Assume your filter provides a specific functionality and is customizable.

For example:

- When applying the filter to the Index action method, you might want to pass arguments like **1** and **2**.
- When applying the same filter to the Create action method, you might want to pass different arguments, like **3** and **4**.

Since different action methods may require different arguments, passing argument values through the `TypeFilter` attribute allows you to customize the filter behavior accordingly.

Handling Arguments in the Filter Constructor

- The first parameter receives the first argument value, the second parameter receives the second argument value, and so on.
- If your filter requires dependency injection, place your injected services before the parameters in the constructor.
- If no dependencies need to be injected, you can directly define the parameters to receive the argument values.

This structured approach ensures flexibility and reusability of your filters across multiple action methods. ↗

```
using Microsoft.AspNetCore.Mvc.Filters;
namespace CRUDsample.Filters.ActionFilters
{
    public class ResponseHeaderActionFilter : IActionFilter
    {
        private ILogger<ResponseHeaderActionFilter> _logger;
        private readonly string Key;
        private readonly string Value;

        public ResponseHeaderActionFilter	ILogger<ResponseHeaderActionFilter> logger, string key, string value)
        {
            this._logger = logger;
            this.Key = key;
            this.Value = value;
        }

        //before
        public void OnActionExecuting(ActionExecutingContext context)
        {
            _logger.LogInformation($"{nameof(ResponseHeaderActionFilter)} {nameof(OnActionExecuting)}");
        }

        //after
        public void OnActionExecuted(ActionExecutedContext context)
        {
            _logger.LogInformation($"{nameof(ResponseHeaderActionFilter)} {nameof(OnActionExecuted)}");
            context.HttpContext.Response.Headers[key] = Value;
        }
    }
}
```

Now we have to supply the parameter, but we do not need to supply ILogger as it is service instances.

```
// [Route("index")]
// [HttpGet("action")]
// [TypeFilter(typeof(PersonListActionFilter))]
// [TypeFilter(typeof(PersonResponseHeaderActionFilter))]
public async Task<IActionResult> Index(string searchBy, string sortBy= nameof(PersonResponse.PersonName), SortOrderOptions sortOrder= SortOrderOptions.Asc)
{
    _logger.LogInformation("Index action method of PersonController");
    _logger.LogInformation("searchBy: {searchBy}, sortBy: {sortBy}, sortOrder: {sortOrder}");
    //Search
    //IEnumerable<Person> result = new Dictionary<string, string>();
    //foreach (var item in result)
    //{
    //    if (item.Key == "Person Name")
    //    {
    //        item.Value = item.Value + ", " + item.Key;
    //    }
    //}
    var result = await _context.Persons
        .Where(p => p != null)
        .Select(p => new PersonResponse()
        {
            PersonName = p.Name,
            Email = p.Email
        })
        .ToListAsync();
    return View(result);
}
```

```
28     //URL: persons/index
29     [Route("{action}")]
30     [Route("{*}")]
31     [TypeFilter(typeof(PersonListActionFilter))]
32     [TypeFilter(typeof(ResponseHeaderActionFilter)), Arguments = new object[] { "X-Custom-Key", "Custom-Value" }]
33     public async Task<IActionResult> Index(string searchBy, string sortBy= nameof(PersonResponse.PersonName), SortOrderOptions sortOrder= SortOrderOptions.Asc)
34     {
35         _logger.LogInformation("Index action method of PersonController");
36         _logger.LogInformation("searchBy: {searchBy}, sortBy: {sortBy}, sortOrder: {sortOrder}");
37         var result = await _context.Persons
38             .Where(p => p != null)
39             .Select(p => new PersonResponse()
40             {
41                 PersonName = p.Name,
42                 Email = p.Email
43             })
44             .ToListAsync();
45         return View(result);
46     }
47
48     // Executes when the user clicks on "Create Person" hyperlink
49     [HttpPost("create")]
50     [TypeFilter(typeof(PersonListActionFilter))]
51 }
```







In ASP.NET Core, filters can be applied at **three levels or scopes**:

1. **Method Level** – Applied to a specific action method.
2. **Controller Level** – Applied to all action methods within a controller.
3. **Global Level** – Applied to all controllers and action methods across the entire project.

How Filters Work at Each Level

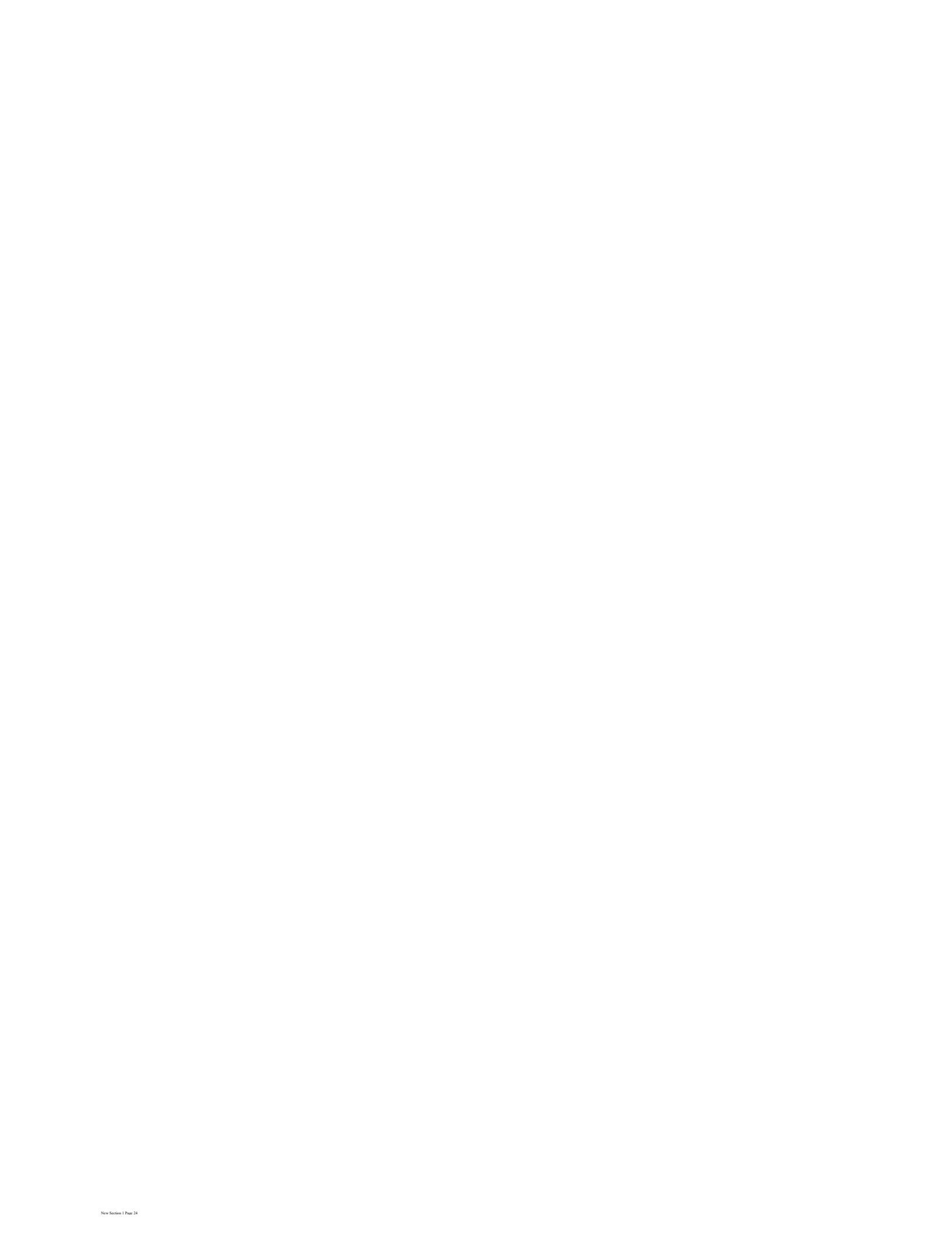
- If a filter is applied to a **specific action method**, it only executes for that method, not for other methods in the same controller.
 - If a filter is applied at the **controller level**, it executes for all action methods within that controller.
 - If a filter is applied **globally**, it affects all controllers and action methods in the project.
- This structured approach helps in managing cross-cutting concerns like logging, authentication, and exception handling efficiently. ↗



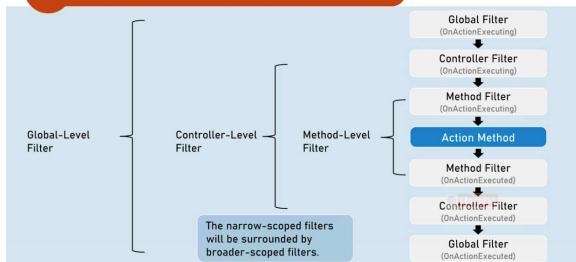
```
ClassLevel
01  using ServiceContracts.DTO;
02  using ServiceContracts.Dtos;
03
04  namespace CMUDExample.Controllers
05
06  [Route("api/[controller]")]
07  [TypeFilter(typeof(CustomTypeFilter), Arguments = new object[] { "x-Custom-Key", "Custom-Value" })]
08  public class PersonController : Controller
09  {
10      //private fields
11      private readonly IPersonsAdderService _personsAdderService;
12      private readonly IPersonsDeleteService _personsDeleteService;
13      private readonly IPersonsGetService _personsGetService;
14      private readonly IPersonsUpdateService _personsUpdateService;
15      private readonly IPersonsOrderService _personsOrderService;
16
17      private readonly ICountriesGetterService _countriesGetterService;
18      private readonly ILogger<PersonController> _logger;
19
20      public PersonController(IPersonsAdderService personsAdderService, IPersonsDeleteService
21      
```

```
builder.Services.AddControllersWithViews(options =>
{
    options.Filters.Add<FilterClassName>(); //add by type
    //or
    options.Filters.Add(new FilterClassName()); //add filter instance
});
```



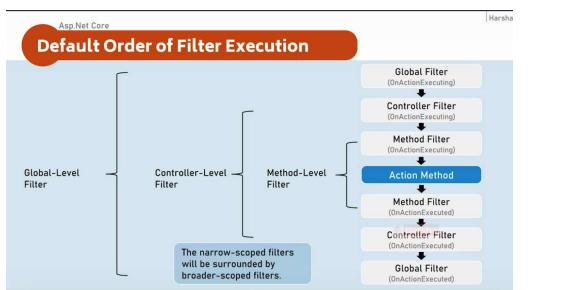


Default Order of Filter Execution



Asp.Net Core Custom Order of Filters

This is the default order of filter execution.



Custom Order of Filter Execution



Custom Order of Filters - Example



Default Order of Filter Execution in ASP.NET Core

In ASP.NET Core, the order of filter execution follows a specific hierarchy:

1. Broader Scope Executes First → The order starts with the **global filters**, followed by **controller-level filters**, and then **method-level filters**.
2. Narrower Scope Executes Next → The process moves from the broader scope down to the action method.
3. Reverse Order for OnActionExecuted → Once the action method completes, the filters execute in the **reverse order**.

Modifying Filter Execution Order

You can **override the default order** by setting the `Order` property when applying filters.





- ◊ The **default order** for all filters is **0**.
 - ◊ You can assign any **integer value** (positive, negative, or zero) to change execution priority.
 - ◊ The filter with the **lowest order value executes first** (ascending order).
 - ◊ When filters have the **same order value**, the **broader scope executes first** (Global → Controller → Method).

Example Scenario:

- **Filter A:** Global filter (Order = 0)
 - **Filter B:** Controller-level filter (Order = 0)
 - **Filter C:** Method-level filter (Order = 0)
 - **Filter D:** Another method-level filter (Order = 1)

Execution Order for OnActionExecuting:

- 1 Global Filter (Order = 0)**
 - 2 Controller Filter (Order = 0)**
 - 3 Method Filter (Order = 0)**

Execution Order for OnAction

- Execution Order for OnAction
 - Method Filter (Order = 1)
 - Method Filter (Order = 0)
 - Controller Filter (Order = 0)
 - Global Filter (Order = 0)

Changing Filter Order Manually

If you want to change the execution order, explicitly set Order values.

```
[ServiceFilter(typeof(MyGlobalFilter), Order = 2)]
[ServiceFilter(typeof(MyControllerFilter), Order = 1)]
[ServiceFilter(typeof(MyMethodFilter), Order = 0)]
public IActionResult Index()
{
    return View();
}
```

Now, **Method Filter** runs first, followed by **Controller Filter**, and **Global Filter** runs last. The reverse happens during `OnActionExecuted`. This mechanism ensures **flexibility** in controlling the filter pipeline. ☀

In our project, at program.cs file, default order for global filter is 0. So, I am not changing anything for global filter.

```
19
20
21 //it adds controllers and views as services
22 builder.Services.AddControllersWithViews(options => {
23     options.Filters.AddResponseHeaderActionFilter>();
24 }
25
26 var logger = builder.Services.BuildServiceProvider()
27     .GetRequiredService<ILogger<ResponseHeaderActionFilter>>()
28     ;
29
30
31 //options.Filters.Add<ResponseHeaderActionFilter>(logger,
32 //    "My-Key-From-Global", "My-Value-From-Global");
33
34 };
35
36
37 //add services into IoC container
38 builder.Services.AddScopedIICountriesRepository,
39 CountriesRepository>();
```

At PersonsController.cs file

```
[Route("{index}")]
[Route("{action}")]
[Route("{*path}")]
[TypeFilter(typeof(PersonsListActionFilter))]
[TyperFilter(typeof(ResponseBodyActionFilter)), Arguments = new object[] { "My-Key-From-Action", "My-Value-From-Action", Order = 1}]
public async Task< IActionResult> Index(string searchBy, string searchString, string sortBy= nameof(PersonResponse.PersonName), SortOrderOptions sortOrder = SortOrderOptions.Ascending)
{
    _logger.LogInformation("Index action method of PersonsController");
    _logger.LogDebug($"SearchBy: {searchBy}, searchString: {searchString}, sortBy: {sortBy}, sortOrder: {sortOrder}");
    //Search
    //ViewBag.SearchFields = new Dictionary<string, string>()
    //{
    //    [nameof(PersonResponse.PersonName)] = "Person Name",
    //    [nameof(PersonResponse.Email)] = "Email",
    //    [nameof(PersonResponse.DateOfBirth)] = "Date of Birth",
    //    [nameof(PersonResponse.Gender)] = "Gender",
    //};
}
```

If you want to execute your filter sooner than others, you will set a lower value ; may be negative value

```
29 //Url: "persons/index"
30 [Route("{action}")]
31 [Route("Filtered")]
32 [TypeFilter(typeof(PersonsListActionFilter))]
33 [TypeFilter(typeof(ResponseHeaderActionFilter)), Arguments =
  new object[] { "MyKey-FromAction", "MyValue-From-Action" },
  Order = 20]
34 public async Task<IActionResult> Index(string searchBy,
  string searchString, string sortBy = nameof
  (PersonResponse.PersonName), SortOrderOptions sortOrder =
  SortOrderOptions.ASC)
35 {
36   _logger.LogInformation("Index action method of
  PersonsController");
37 
38   _logger.LogDebug($"Search by: {searchBy}, searchString:
  {searchString}, sortBy: {sortBy}, sortOrder:
  {sortOrder}");
39 }
```

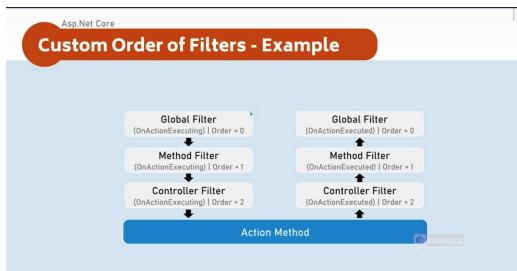
Now last order at Contoller 2

```
 8     using ServiceContracts.Enums;
 9 
10    namespace CRUDExample.Controllers
11    {
12        [Route("{controller}")]
13        [TypeFilter("MyCustomHeaderActionFilter"), Arguments = new object[] { "My-Key-From-Controller", "My-Value-From-Controller", Order = 2}]
14        public class PersonController : Controller
15        {
16            //private fields
17            private readonly IPersonsSetterService _personsSetterService;
18            private readonly IPersonsAdderService _personsAdderService;
19            private readonly IPersonsSorterService _personsSorterService;
20            private readonly IPersonsUpdaterService _personsUpdaterService;
21            private readonly IPersonsDeleterService _personsDeleterService;
22 
23            private readonly ICountrysSetterService _countrysSetterService;
24 
25            private readonly ICountrysAdderService _countrysAdderService;
26            private readonly ICountrysSorterService _countrysSorterService;
27            private readonly ICountrysUpdaterService _countrysUpdaterService;
28            private readonly ICountrysDeleterService _countrysDeleterService;
29 
30            private readonly ICitiesSetterService _citiesSetterService;
31 
32            private readonly ICitiesAdderService _citiesAdderService;
33            private readonly ICitiesSorterService _citiesSorterService;
34            private readonly ICitiesUpdaterService _citiesUpdaterService;
35            private readonly ICitiesDeleterService _citiesDeleterService;
36 
37            private readonly IRegionsSetterService _regionsSetterService;
38 
39            private readonly IRegionsAdderService _regionsAdderService;
40            private readonly IRegionsSorterService _regionsSorterService;
41            private readonly IRegionsUpdaterService _regionsUpdaterService;
42            private readonly IRegionsDeleterService _regionsDeleterService;
43 
44            private readonly IContinentSetterService _continentSetterService;
45 
46            private readonly IContinentAdderService _continentAdderService;
47            private readonly IContinentSorterService _continentSorterService;
48            private readonly IContinentUpdaterService _continentUpdaterService;
49            private readonly IContinentDeleterService _continentDeleterService;
50 
51            private readonly ICountrysDeleterService _countrysDeleterService;
52 
53            private readonly ICitiesDeleterService _citiesDeleterService;
54 
55            private readonly IRegionsDeleterService _regionsDeleterService;
56 
57            private readonly IContinentDeleterService _continentDeleterService;
58 
59            private readonly IPersonsDeleterService _personsDeleterService;
60 
61            private readonly IPersonsAdderService _personsAdderService;
62 
63            private readonly IPersonsSorterService _personsSorterService;
64 
65            private readonly IPersonsUpdaterService _personsUpdaterService;
66 
67            private readonly IPersonsSetterService _personsSetterService;
68 
69            private readonly IPersonsDeleterService _personsDeleterService;
70 
71            private readonly IPersonsAdderService _personsAdderService;
72 
73            private readonly IPersonsSorterService _personsSorterService;
```

As per our discussion.







Now Run our application set a debugger at the Filter constructor and analyze

```
string value)
{
    _logger = logger;
    Key = key;
    Value = value;
}
```

Actually in this example, there is no need of changing the order. But sometimes, the data read from one action filter need to be accessed by another action filter. So in that particular cases only we have to change the order of execution.

But in general cases, there is no need of filter execution.

But if you follow this approach, you can't set the 'Order' in global filter. We have to implement 'IOrderedFilter' interface.

```
19
20
21 //it adds controllers and views as services
22 builder.Services.AddControllersWithViews(options => {
23     options.Filters.Add<ResponseHeaderActionFilter>();
24
25     var logger = builder.Services.BuildServiceProvider()
26         .GetRequiredService<ILogger<ResponseHeaderActionFilter>>();
27
28     options.Filters.Add(new ResponseHeaderActionFilter(logger,
29         "My-Key-From-Global", "My-Value-From-Global"));
30
31 //add services into IoC container
32 builder.Services.AddScoped<ICountriesRepository,
```



As we have seen already, we can by default set the order property value in the type filter attribute, either in the controller or at the method level. See, for example, in this person's controller, while using the type filter, you can set the desired order value without a problem.

```
4 using Rotativa.AspNetCore;
5 using ServiceContracts;
6 using ServiceContracts.DTO;
7 using ServiceContracts.Enums;
8 using System.IO;
9
10 namespace CRUDExample.Controllers
11 {
12     [Route("[controller]")]
13     [TypeFilter(typeof(ResponseHeaderActionFilter), Arguments = >
14         new object[] { "My-Key-From-Controller", "My-Value-From-Controller" }, Order = 20)]
15     public class PersonsController : Controller
16     {
17         //private fields
18         private readonly IPersonsService _personsService;
19         private readonly ICountriesService _countriesService;
```

But it is not possible by default to use this type filter for global filters. For example, in Program.cs, while adding the filter instance, we cannot set the order property value by default. To overcome this problem, there is an alternative way to set the order, which is generally preferred and widely used. That is by using the predefined interface called IOrderedFilter.

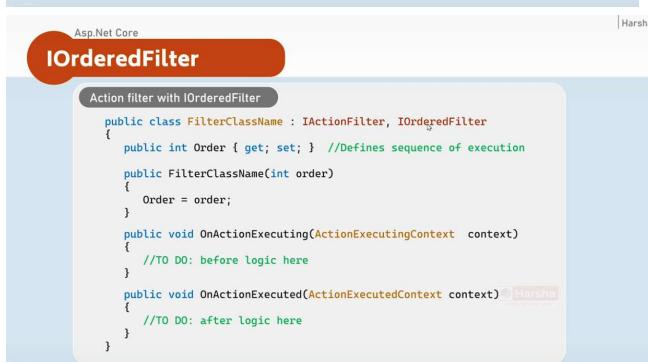




```

19
20
21 //it adds controllers and views as services
22 builder.Services.AddControllersWithViews(options => {
23     //options.Filters.Add<ResponseHeaderActionFilter>();
24 }
25
26 var logger = builder.Services.BuildServiceProvider()
27     .GetRequiredService<ILogger<ResponseHeaderActionFilter>>()
28 );
29
30 //add services into IoC container
31 builder.Services.AddScoped<ICountriesRepository,
32 CountriesRepository>();

```



```

1 using Microsoft.AspNetCore.Mvc.Filters;
2
3 namespace CRUDExample.Filters.ActionFilters
4 {
5     public class ResponseHeaderActionFilter : IActionFilter, IOrderedFilter
6     {
7         private ILogger<ResponseHeaderActionFilter> _logger;
8         private readonly string Key;
9         private readonly string Value;
10
11         public int Order { get; set; }
12
13         public ResponseHeaderActionFilter(ILogger<ResponseHeaderActionFilter> logger, string key, string value, int order)
14         {
15             this._logger = logger;
16             this.Key = key;
17             this.Value = value;
18             Order = order;
19         }
20
21         //before
22         public void OnActionExecuting(ActionExecutingContext context)
23         {
24             _logger.LogInformation($"[FilterName] ({MethodofName}) method", nameof(ResponseHeaderActionFilter), nameof(OnActionExecuting));
25         }
26
27         //after
28         public void OnActionExecuted(ActionExecutedContext context)
29         {
30             _logger.LogInformation($"[FilterName] ({MethodofName}) method", nameof(ResponseHeaderActionFilter), nameof(OnActionExecuted));
31             context.HttpContext.Response.Headers[key] = Value;
32         }
33     }
34 }

```

```

27
28 //Url: persons/index
29 [Route("action")]
30 [Route("")]
31 [TypeFilter(typeof(PersonsListActionFilter))]
32 [TypeFilter(typeof(ResponseHeaderActionFilter))], Arguments =
33 new object[] { "MyKey-FromAction", "MyValue-From-Action",
34 1 }];
35
36 public async Task<IActionResult> Index(string searchBy,
37 string? searchString, string sortBy = nameof(
38 PersonResponse.PersonName), SortOrderOptions sortOrder =
39 SortOrderOptions.ASC)
40 {
41     _logger.LogInformation("Index action method of PersonsController");
42
43     _logger.LogInformation("so we are supplying the three values", searchString);
44 }

```

Setting order for global filter





```

    builder.Services.AddControllersWithViews();
    //It adds controllers and views as services
    builder.Services.AddControllersWithViews(options =>
    {
        //options.Filters.Add<ResponseHeaderActionFilter>(); // but you can supply parameter in this way
        //options.Filters.Add<ResponseHeaderActionFilter>(logger, "My-Key-From-Global", "My-Value-From-Global", 2);
        options.Filters.Add(new ResponseHeaderActionFilter(logger, "My-Key-From-Global", "My-Value-From-Global", 2));
    });
    //Add services into IoC container
    builder.Services.AddScoped<ICountriesGetterService, CountriesGetterService>();

```

Setting order at the Controller Level

```

1  using ServiceContracts.DTO;
2  using ServiceContracts.Enums;
3
4  namespace CRUDExample.Controllers
5  {
6      [Route("[controller]")]
7      [TypeFilter(typeof(ResponseHeaderActionFilter), Arguments = new object[] { "My-Key-From-Controller", "My-Value-From-Controller", 3 })]
8      public class PersonsController : Controller
9      {
10         //private fields
11         private readonly IPersonsGetterService _personsGetterService;
12     }

```

Keep in mind, sometimes ASP.NET CORE picks up Order from the 'TypeFilter', even though we have implemented 'IOrderedFilter' in our filter class.

It's a bug of ASP.NET Core.

```

9  namespace CRUDExample.Controllers
10 {
11     [Route("[controller]")]
12     [TypeFilter(typeof(ResponseHeaderActionFilter), Arguments = new object[] { "My-Key-From-Controller", "My-Value-From-Controller", 3 })]
13     public class PersonsController : Controller
14     {
15         //private fields
16     }

```

```

1  using ServiceContracts.Enums;
2
3  namespace CRUDExample.Controllers
4  {
5      [Route("[controller]")]
6      [TypeFilter(typeof(ResponseHeaderActionFilter), Arguments = new object[] { "My-Key-From-Controller", "My-Value-From-Controller", 3 }, Order = 3)]
7      public class PersonsController : Controller
8      {
9          //private fields
10         private readonly IPersonsService _personsService;
11     }

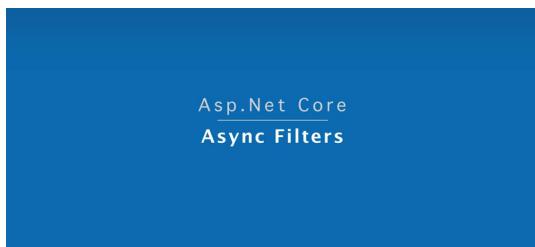
```

No problem at global filter

```

25     var logger = builder.Services.BuildServiceProvider()
26         .GetRequiredService<ILogger<ResponseHeaderActionFilter>>();
27     options.Filters.Add(new ResponseHeaderActionFilter(logger,
28         "My-Key-From-Global", "My-Value-From-Global", 2));
29 }
30 //add services into IoC container

```



```

Asp.Net Core
Async Filter

Asynchronous Action Filter
public class FilterClassName : IAsyncActionFilter, IOrderedFilter
{
    public int Order { get; set; } //Defines sequence of execution
    public FilterClassName(int order)
    {
        Order = order;
    }
    public async Task OnActionExecutionAsync(ActionExecuting context,
                                             ActionExecutionDelegate next)
    {
        //TO DO: before logic here
        await next();
        //TO DO: after logic here
    }
}

```

If you want to call any asynchronous methods in your filter, meaning you need to perform asynchronous operations within your filter method, you must convert your filter class into an asynchronous filter.

Changes Required:

- Instead of implementing the `IActionFilter` interface, implement the `IAsyncActionFilter` interface to make it an asynchronous action filter.
- Instead of writing two separate methods (`OnActionExecuting` and `OnActionExecuted`), combine both logics into a single method: `OnActionExecutionAsync`.
- Much like middleware, you will call the next method, which is of type `ActionExecutionDelegate`.

Execution Flow:

- The `context` object contains all HTTP-related information, including the controller, request, and result data.





- No changes are required in the execution context.
- Optionally, you can implement `IOrderedFilter` if you need to define the order of execution (this is optional).

Key Differences:

- The logic that you previously wrote in `OnActionExecuting` should now be placed **above** the next method call (before execution logic).
- The logic that you previously wrote in `OnActionExecuted` should be placed **after** the next method call (post-execution logic).
- The next method represents the subsequent filter. If no other filters exist, it directly executes the actual action method.

How It Works:

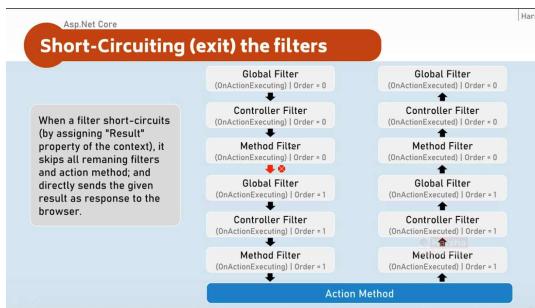
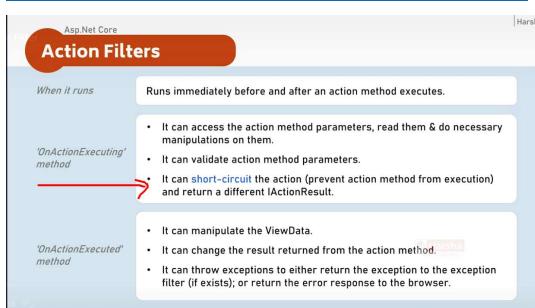
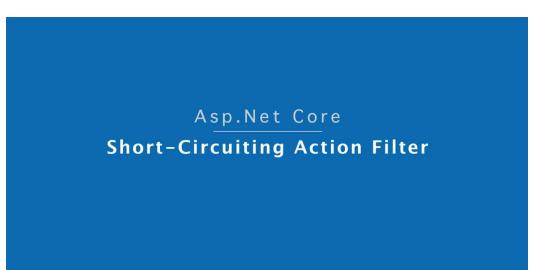
- If multiple filters are applied, filter 1 executes first.
- After executing its pre-action logic, it calls filter 2 using the next method.
- Filter 2 then follows the same pattern, calling the next filter or executing the action method.
- This chaining mechanism is similar to how middleware works.

Benefit of Asynchronous Filters:

By converting the filter into an asynchronous filter, you can now call any asynchronous methods, such as:

- Database calls
- File system operations
- Asynchronous service calls

This improves efficiency, especially when dealing with I/O operations.



Understanding Result Filters in ASP.NET Core

Result filters allow you to execute logic **before** and **after** an action result is processed. They function similarly to action filters but specifically target the result execution phase rather than the action method itself.

Execution Flow

1. Action Filters Execute First:

- `OnActionExecuting` runs **before** the action method.





- OnActionExecuted runs after the action method.

2. Result Filters Execute Next:

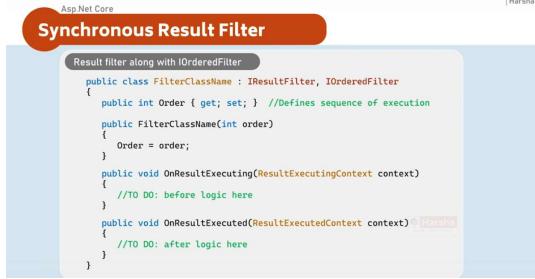
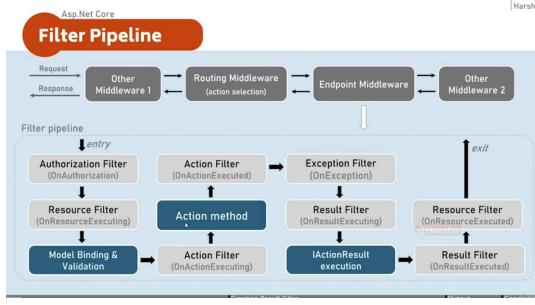
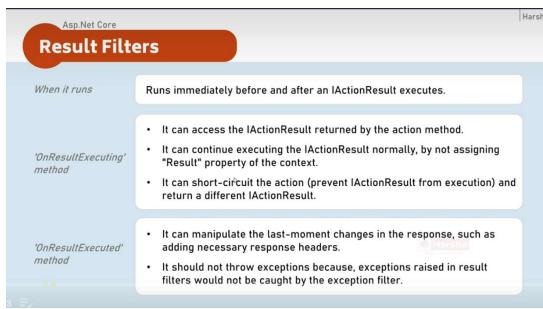
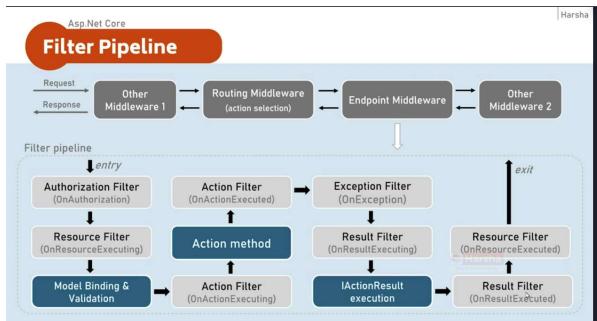
- OnResultExecuting runs before the action result is generated.
- OnResultExecuted runs after the action result has been executed.

Purpose of Result Filters

- Short-circuiting Execution:** You can override the original result before it is sent to the client.
- Modifying View Data:** Additional data can be injected into the view before it is rendered.
- Last-Minute Response Changes:** Headers or response content can be modified after execution.

Implementation Approaches

- Synchronous Filters:** Implement IResultFilter with OnResultExecuting and OnResultExecuted.
- Asynchronous Filters:** Implement IAsyncResultFilter with OnResultExecutionAsync, where the next delegate calls the next filter or the action result itself.







Suppose our requirement is that if we talk about the person's controller Index Action method, after execution of the view result, I would like to add a response header. After the view executes.

```

12 [TypeFilter(typeof(ResponseHeaderActionFilter)), Arguments = 
13 new object[] { "My-Key-From-Controller", "My-Value-From- 
14 Controller", 3 }, Order = 3]
15
16 private readonly IPersonsService _personsService;
17 private readonly ICountriesService _countriesService;
18 private readonly ILogger<PersonsController> _logger;
19
20
21 public PersonsController(IPersonsService personsService,
22 ICountriesService countriesService,
23 ILogger<PersonsController> logger)
24 {
25     _personsService = personsService;
26     _countriesService = countriesService;
27 }

```

```

1 using Microsoft.AspNetCore.Mvc.Filters;
2
3 namespace CRUDExample.Filters.ResultFilters
4 {
5     public class PersonListResultFilter : IActionResultFilter
6     {
7         private readonly ILogger<PersonListResultFilter> _logger;
8
9         public PersonListResultFilter(ILogger<PersonListResultFilter> logger)
10        {
11            _logger = logger;
12        }
13
14        public async Task OnResultExecutionAsync(ResultExecutingContext context, ResultExecutionDelegate next)
15        {
16            // TO DO: before logic
17            _logger.LogInformation($"[FilterName] - before", nameof(PersonListResultFilter), nameof(OnResultExecution));
18
19            await next(); // call the subsequent filter or IActionResult
20
21            // TO DO: after logic
22            _logger.LogInformation($"[FilterName] - after", nameof(PersonListResultFilter), nameof(OnResultExecution));
23
24        }
25    }
26 }

```

```

39 //Route["index"]
40 [Route("{"action"}")]
41 [Route("{*}")]
42 [TypeFilter(typeof(PersonsListActionFilter), Order = 0)] // 0 as per the presentation
43 [TypeFilter(typeof(ResponseHeaderActionFilter)), Arguments = new object[] { "My-Key-From-Action", "My-Value-From-Action", 1 }, Order = 1]
44 [TypeFilter(typeof(PersonsListResultFilter))]
45 public async Task<ActionResult> Index(string searchBy, string? searchString, string sortBy = nameof(PersonResponse.PersonName), SortOrderOption sortOrder = SortOrderOption.Ascending)
46 {
47     _logger.LogInformation("Index action method of PersonsController");
48     _logger.LogDebug($"searchBy: {searchBy}, searchString: {searchString}, sortBy: {sortBy}, sortOrder: {sortOrder}");
49
50     //Search
51     ViewBag.SearchFields = new Dictionary<string, string>()
52     //{
53     //    {nameof(PersonResponse.PersonName), "Person Name"}, 
54     //    {nameof(PersonResponse.Email), "Email"}, 
55     //    {nameof(PersonResponse.DateOfBirth), "Date of Birth"}, 
56     //    {nameof(PersonResponse.Gender), "Gender"}, 
57     //    {nameof(PersonResponse.CountryID), "Country"}, 
58     //    {nameof(PersonResponse.Address), "Address"}, 
59     //};
59
60
61     List<PersonResponse> persons = await _personsGetterService.GetFilteredPersons(searchBy, searchString);
62     ViewBag.CurrentSearchBy = searchBy;
63     ViewBag.CurrentSearchString = searchString;
64
65     //Sort
66     List<PersonResponse> sortedPersons = await _personsSorterService.GetSortedPersons(persons, sortBy, sortOrder);
67     ViewBag.CurrentSortBy = sortBy;
68     ViewBag.CurrentSortOrder = sortOrder.ToString();
69 }
70

```

| Name | Value |
|-----------|-------|
| localhost | |
| Styles... | |
| all.m... | |
| aspnet... | |
| logo.p... | |
| browse... | |
| negoti... | |
| connec... | |
| CRUDE... | |

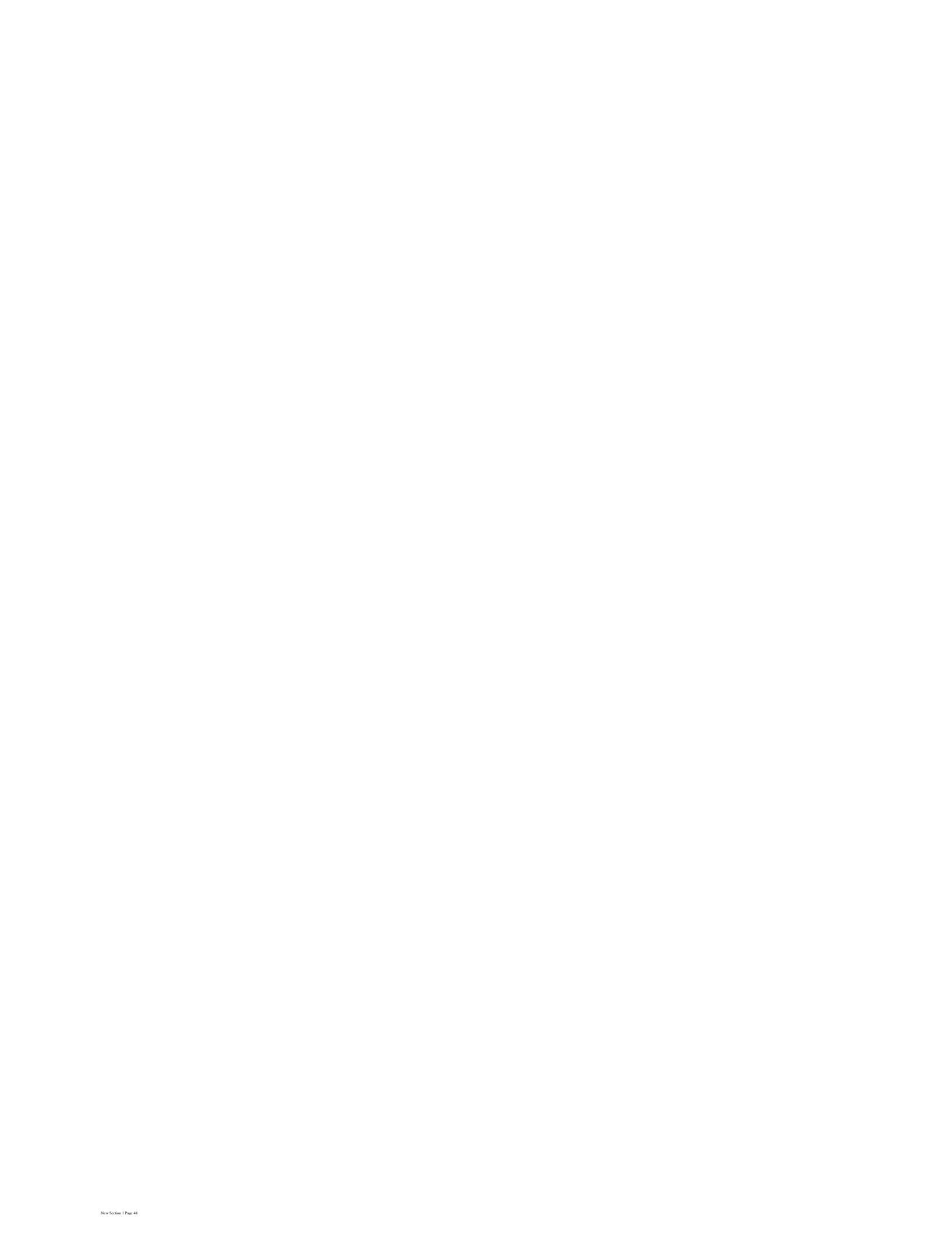
Headers

| Name | Value |
|------------------------|---------------------------------|
| Status Code | 200 OK |
| Remote Address | [:1]:5298 |
| Referrer Policy | strict-origin-when-cross-origin |
| Last-Modified | 2022-07-30 22:38 |
| My-Key-From-Controller | My-Value-From-Controller |
| My-Key-From-Global | My-Value-From-Global |

In general, result filters are rarely used in real-world projects, except in specific cases where you need to prevent the execution of a result. For example, instead of executing a `ViewResult`, you might return a `ContentResult` or `JsonResult`. Additionally, result filters can be useful when transforming the response body from one format to another, such as converting an XML response into a JSON response.

Asp.Net Core Resource Filter





Resource Filters

If you need to prevent the execution of the filter pipeline, resource filters are the best option. Their placement in the filter pipeline is very early, as they execute immediately after the authorization filter and almost at the end after the result filters are completed.

In other words, resource filters wrap the entire process, including model binding, action filters, action methods, and result filters.

- The OnResourceExecuting method runs before model binding.
- The OnResourceExecuted method runs after the result filters.

Usage in Real-World Projects

- The OnResourceExecuted method is rarely used, but OnResourceExecuting has many practical applications.
- Its main purpose is to prevent or short-circuit the remaining filter pipeline based on certain conditions.

For example:

- If the requested content type is not supported by the server (e.g., the browser expects text/plain, but the application does not support it), we can short-circuit the request and return an error response or a ContentResult.

Since OnResourceExecuting runs before model binding, it can modify the request before the action method executes.

Example: Adjusting the request body, adding extra values, or modifying request headers.

Another use case: Tracking execution time by logging the duration of the action method.

When to Use OnResourceExecuted?

Though it's rarely used, one case where OnResourceExecuted is useful is caching the response body.

Synchronous vs. Asynchronous Resource Filters

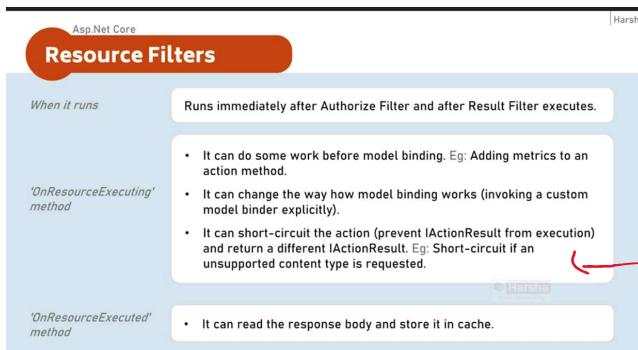
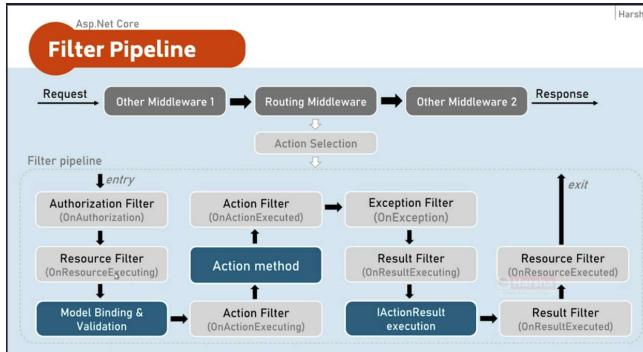
- If you need a synchronous resource filter, implement IResourceFilter (just like IResultFilter or IActionFilter).
 - It contains two methods: OnResourceExecuting and OnResourceExecuted, handling before and after logic, respectively.

- If you need an asynchronous resource filter, implement IAsyncResourceFilter.

The before logic goes above the next delegate.

The after logic goes below the next delegate.

>Note: Implementing IOrderedFilter is optional unless you have multiple resource filters and need to control their execution order.







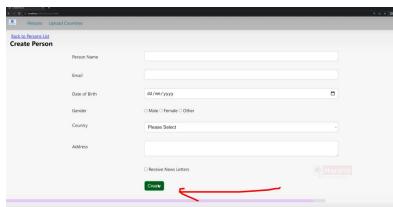
Asynchronous Resource Filter

```
Asynchronous Resource Filter
public class FilterClassName : IAsyncResourceFilter, IOOrderedFilter
{
    public int Order { get; set; } //Defines sequence of execution
    public FilterClassName(int order)
    {
        Order = order;
    }
    public async Task OnResourceExecutionAsync(ResourceExecutingContext context,
                                                ResourceExecutionDelegate next)
    {
        //TO DO: before logic here
        await next();
        //TO DO: after logic here
    }
}
```

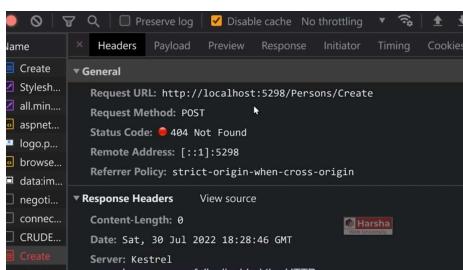
Suppose, I want an resource filter that temporarily disables an action method.

```
FeatureDisabledResourceFilter.cs
using Microsoft.AspNetCore.Mvc;
using Microsoft.AspNetCore.Mvc.Filters;
namespace CMSSample.Filters.ResourceFilters
{
    public class FeatureDisabledResourceFilter : IAsyncResourceFilter
    {
        private readonly ILogger<FeatureDisabledResourceFilter> _logger;
        private readonly bool _isDisabled;
        public FeatureDisabledResourceFilter(ILogger<FeatureDisabledResourceFilter> logger, bool isDisabled = true)
        {
            _logger = logger;
            _isDisabled = isDisabled;
        }
        public async Task OnResourceExecutionAsync(ResourceExecutingContext context, ResourceExecutionDelegate next)
        {
            //TO DO: before logic
            _logger.LogInformation($"[filterName] - before", nameof(FeatureDisabledResourceFilter), nameof(OnResourceExecutionAsync));
            if (_isDisabled)
            {
                //Assigning any non null value into this property will stop the execution of the pipeline (short-circuiting)
                //context.Result = new BadRequestResult(); //If context.Result returns HTTP 400 error response
                else
                {
                    await next();
                }
            }
            //TO DO: after logic
            _logger.LogInformation($"[filterName] - after", nameof(FeatureDisabledResourceFilter), nameof(OnResourceExecutionAsync));
        }
    }
}
```

```
CreatePersonController.cs
    [HttpPost]
    [ValidateAntiForgeryToken]
    [TypeFilter(typeof(PersonCreateAndEditPostActionFilter))]
    [TypeFilter(typeof(FeatureDisabledResourceFilter))]
    public async Task Create(PersonAddRequest personRequest)
    {
        //if (!ModelState.IsValid) //before executing this controller method, model validation gets executed
        //{
        //    List<CountryResponse> countries = await _countriesGetterService.GetAllCountries();
        //    ViewBag.Countries = countries.Select(temp => new SelectListItem()
        //{
        //        Text = temp.CountryName,
        //        Value = temp.CountryID.ToString(),
        //});
        //}
        return View();
    }
}
```



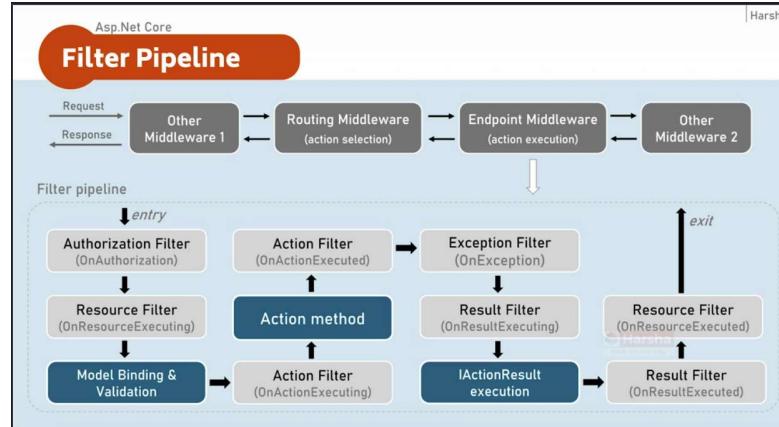
Click on this button and see whether our Create Action method works or not







Asp .Net Core Authorization Filter



Authorization Filters in ASP.NET Core

The first filter that executes in the filter pipeline is the **Authorization Filter**.

- As soon as a request reaches the endpoint middleware, the first filter to execute is the **authorization filter**, thanks to its **early position** in the pipeline.

Purpose of Authorization Filters

- Used to validate whether a user is logged in or not.
- Ensures that only **authenticated users** can access certain resources.
- If the user is not logged in, the request is **short-circuited**, and an **error response** is returned **immediately**.

Example Use Case

Imagine a mailing application with an **Inbox** page that displays a list of emails.

Rule:

- The user **must be logged in** to access the Inbox page.

⌚ What happens when an unauthorized request is made?

1. The request reaches the **Authorization Filter**.
2. The filter checks whether the user is **logged in**.
3. If the user is **not logged in**, they are **redirected** to the **login page**.
4. This prevents further execution of the **action method** and other filters.

How Does It Work?

Authentication is typically handled using **cookies**.

◦ What are cookies?

- Cookies are stored in the **browser**, but the **server** can send cookies to the browser.
- Once stored, cookies are **automatically** sent with all **subsequent requests** to the same domain.
- This allows the server to verify if a user is **logged in** for every request.

Authentication vs. Authorization

Term Definition

Authentication Checks if the user is **logged in**.

Authorization Checks if the user has **permission** to access a specific resource.

For example:

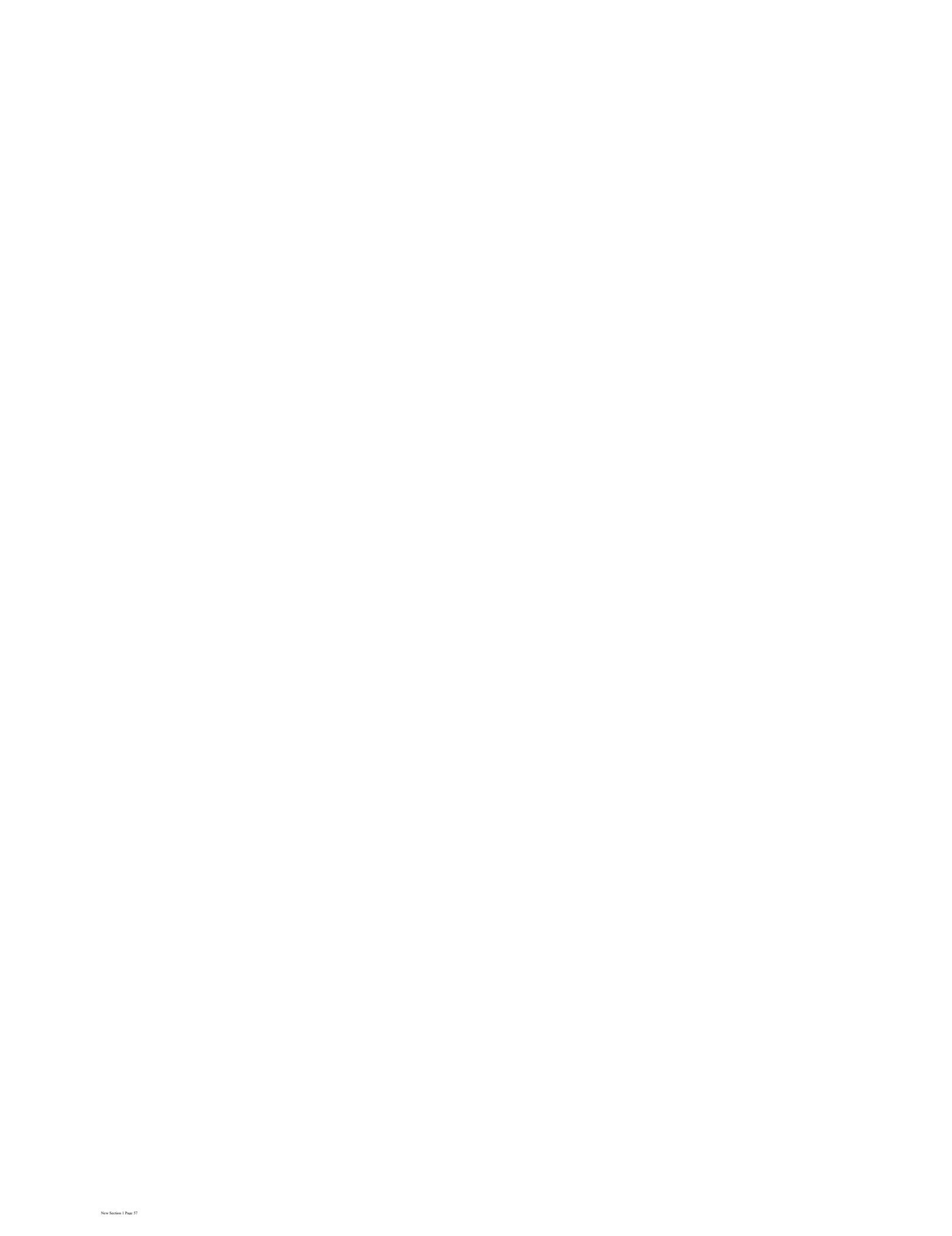
- **Authentication** ensures that a user is logged in.
- **Authorization** ensures that the logged-in user has permission to view the **Inbox page**.
- If the user lacks permission, they are **redirected** or given an **error response**.

Implementation of Authorization Filters

❖ Synchronous Authorization Filter

- Implement **IAuthorizationFilter**
- Override **OnAuthorization** method





❖ Asynchronous Authorization Filter

- Implement `IAsyncAuthorizationFilter`
- Allows calling **asynchronous** methods

💡 Note:

- Prefer **asynchronous filters** (`IAsyncAuthorizationFilter`) when calling async methods.
- **Do not throw exceptions** in the `OnAuthorization` method, as **exception filters** will **not** catch them.

Now, let's move on to implementing this **practically** in ASP.NET Core! 🔒

Asp.Net Core | Harsha

Authorization Filters

When it runs

Runs before any other filters in the filter pipeline.

'OnAuthorize' method

- Determines whether the user is authorized for the request.
- Short-circuits the pipeline if the request is NOT authorized.
- Don't throw exceptions in `OnAuthorize` method, as they will not be handled by exception filters.

Harsha
www.harshaindia.com

Asp.Net Core | Harsha

Synchronous Authorization Filter

Authorization filter

```
public class FilterClassName : IAuthorizationFilter
{
    public void OnAuthorization(AuthorizationFilterContext context)
    {
        //TO DO: authorization logic here
    }
}
```

Harsha
www.harshaindia.com

Asp.Net Core | Harsha

Asynchronous Authorization Filter

Authorization filter

```
public class FilterClassName : IAsyncAuthorizationFilter
{
    public async Task OnAuthorizationAsync(AuthorizationFilterContext context)
    {
        //TO DO: authorization logic here
    }
}
```

Harsha
www.harshaindia.com





```

1  using Microsoft.AspNetCore.Mvc;
2  using Microsoft.AspNetCore.Mvc.Filters;
3
4  namespace CRUDEExample.Filters.AuthorizationFilter
5  {
6      public class TokenAuthorizationFilter : IAuthorizationFilter
7      {
8          public void OnAuthorization(AuthorizationFilterContext context)
9          {
10             //assume that we have created a cookie and sent the same to the browser. Then the browser stored the cookies into the browser memory.
11             // Once the cookie is sent the browser, same cookie will automatically be submitted for all the subsequent request to the server.
12             if (context.HttpContext.Request.Cookies.ContainsKey("Auth-Key") == false)
13             {
14                 //as soon as we set an not null value to 'context.Result', that means we are short-circuiting the request pipeline and the request will not reach the controller action.
15                 context.Result = new StatusCodeResult(StatusCodes.Status401Unauthorized);
16             }
17
18             //Suppose the requirement is 'Auth-Key' value should be "A100"
19             if (context.HttpContext.Request.Cookies["Auth-Key"] != "A100")
20             {
21                 context.Result = new StatusCodeResult(StatusCodes.Status401Unauthorized);
22             }
23         }
24     }
25 }

```

```

1  return View(personUpdateRequest);
2
3  //on clicking the submit button from edit page
4  [HttpPost]
5  [Route("{action}/{personID}")]
6  [TypeFilter(typeof(PersonCreateAndEditPostActionFilter))]
7  [TypeFilter(typeof(TokenAuthorizationFilter))]
8  public async Task<ActionResult> Edit(PersonUpdateRequest personRequest)
9  {
10    PersonResponse? personResponse = await _personsGetterService.GetPersonByPersonId(personRequest.PersonID);
11
12    if (personResponse == null)
13    {
14        return RedirectToAction("Index");
15    }
16
17    //if (ModelState.IsValid)
18    //{
19    //PersonResponse updatedPerson = await _personsUpdaterService.UpdatePerson(personRequest);
20    //return RedirectToAction("Index");
21    //}
22    //else
23    //{
24    //    List<CountryResponse> countries = await _countriesGetterService.GetAllCountries();
25    //    ViewBag.Countries = countries.Select(temp =>
26    //        new SelectListItem() { Text = temp.CountryName, Value = temp.CountryID.ToString() });
27    //}
28
29 }

```

Now run our application and click on 'Edit' Page

| Person Name | Email | Date of Birth | Age | Gender | Country | Address | Receive News Letters | Options |
|-------------|-----------------------------|---------------|-----|--------|-----------------------|---------------------|----------------------|---|
| Angie | asavar3@dropbox.com | 09 Jan 1987 | 36 | Male | China | 83187 Merry Drive | True | Edit Delete |
| Franchot | fbowsher2@howstuffworks.com | 10 Feb 1995 | 28 | Male | Philippines | 73 Heath Avenue | True | Edit Delete |
| Hansain | hmoscov@tripod.com | 20 Sep 1990 | 32 | Male | China | 413 Sactjen Way | True | Edit Delete |
| Lombard | lwoodwing9@wix.com | 25 Sep 1997 | 25 | Male | Palestinian Territory | 484 Clarendon Court | False | Edit Delete |
| Maddy | mjarrell6@wisc.edu | 16 Feb 1983 | 40 | Male | China | 57449 Brown Way | True | Edit Delete |
| Marguerite | mwebsdale0@people.com.cn | 28 Aug 1989 | 33 | Female | Thailand | 4 Parkside Point | False | Edit Delete |
| Minta | mconachya@va.gov | 24 May 1990 | 32 | Female | China | 2 Warrior Avenue | True | Edit Delete |

After essential modification, click on update

Person Name: Angie

Email: asavar3@dropbox.com

Date of Birth: 09/01/1987

Gender: Male

Country: China

Address: 83187 Merry Drive

Receive News Letters:

[Update](#)



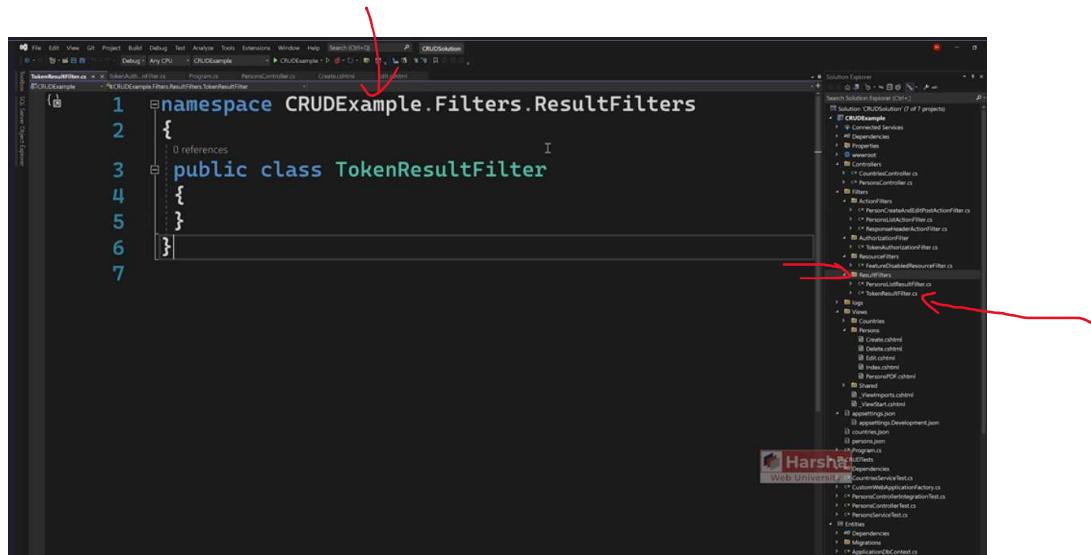

```
6     reference
7     public class TokenAuthorizationFilter : IAuthorizationFilter
8     {
9         public void OnAuthorization(AuthorizationFilterContext context)
10        {
11            if (context.HttpContext.Request.Cookies.ContainsKey("Auth-
12                Key") == false)
13            {
14                context.Result = new StatusCodeResult
15                    (StatusCodes.Status401Unauthorized);
16                return;
17            }
18
19            if (context.HttpContext.Request.Cookies["Auth-key"] != "A100")
20            {
21            }
22        }
23    }

```



Now the question is how do you create the cookie?

Let's create the cookie by using a Result Filter. Because in general, result filters are used to perform last moment changes to the response such as adding a cookie or adding a response header.







```

using Microsoft.AspNetCore.Mvc.Filters;
namespace CRUDExample.Filters.ResultFilters
{
    public class TokenResultFilter : IResultFilter
    {
        public void OnResultExecuting(ResultExecutingContext context)
        {
        }
        public void OnResultExecuted(ResultExecutedContext context)
        {
            //this cookie will be sent to the browser. The browser will store the cookie into the browser memory.
            context.HttpContext.Response.Cookies.Append("Auth-Key", "A100");
        }
    }
}

```

Let's apply this 'TokenResultFilter' in the Person's Controller Edit Action method.

```

another get request to "persons/index"
return RedirectToAction("Index", "Persons");
}

[HttpGet]
[Route("[action]/[{personID}]]") //Eg: /persons/
edit/1
[TypeFilter(typeof(TokenResultFilter))]
public async Task<IActionResult> Edit(Guid personID) ...

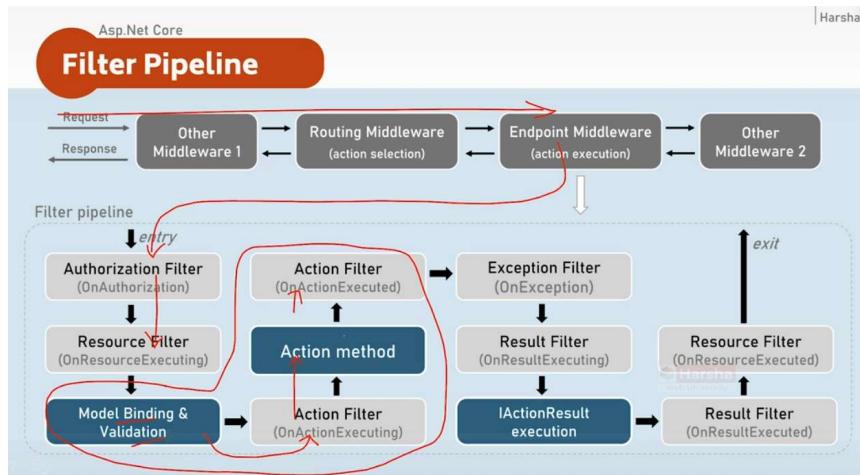
[HttpPost]
[Route("[action]/[{personID}]]")
[TypeFilter(typeof(
PersonCreateAndEditPostActionFilter))]

```

Let's run this. The cookies has been sent and received by the browser.

| | Auth-Key | A100 | lo... | / | S... |
|-----------------------------|------------|-------|-------|------|------|
| AspNetCore.Antiforgery.G... | CfDJ8AA... | lo... | / | S... | |

Asp.Net Core Exception Filter



Exception Filter in ASP.NET Core

Whenever an exception is raised later on, by default, ASP.NET Core sends an **HTTP 500 response** to the browser, which indicates a **runtime server error**—meaning something went wrong on the server side. If you specifically want to catch exceptions that occur during **model binding**, **action filters**, or **action methods**, you can use the **exception filter**.

For example, let's say we make a request to the /persons/index route.

- The request passes through all middleware layers and reaches the **endpoint middleware**.
- Then, the request enters the **filter pipeline**.
- The **authorization filter** executes first. If it **does not short-circuit**, the remaining filters continue executing.
- If a **resource filter** is applied, its `OnResourceExecuting` method runs.
- After that, the **controller object is automatically created**.
- Then, **model binding** and **model validation** are performed.
- Next, the **action filter's OnActionExecuting method** runs.
- Then, the **action method** itself executes.
- Finally, the **action filter's OnActionExecuted method** executes.

Handling Exceptions with Exception Filters

If an **exception occurs** during:

- **Model binding**,
- **Action filter execution**, or



- **Action method execution,**

Then, the **exception filter** automatically executes.

If no exception occurs in this specific area, then the exception filter **does not execute**.

For example, suppose a request encounters an exception:

- During **model binding**
- During **model validation**
- Or inside the **action method** (e.g., a database connection issue raising an SQL exception).

In such cases, the **exception filter executes** to catch the error.

How Exception Filters Work

Once an exception filter catches an exception, it has **two options**:

1. Short-circuit the request

- The filter **logs the exception** (e.g., using **Serilog**).
- It then **returns a JSON result or content result** to the browser.
- This stops further filter execution and sends the response **immediately**.

2. Allow the request to continue

- Some filters may choose **not** to short-circuit.
- After execution, the remaining filters and **result execution pipeline** will continue as usual.

Scope of Exception Filters

- **Exception filters can only catch exceptions in model binding, action filters, and action methods.**

- They **cannot** catch exceptions raised in:
 - **Middleware execution**
 - **Authorization filters**
 - **Resource filters**
 - **Result filters**

If you need to catch exceptions occurring **anywhere in the application**, you should use **exception handling middleware** instead.

When to Use Exception Filters vs. Exception Handling Middleware

- Use **exception filters** for **specific action methods or controllers**.
- Use **exception handling middleware** to **handle exceptions globally** for all requests and middleware.

Synchronous and Asynchronous Exception Filters

1. Synchronous Exception Filter

- Implement the **IExceptionFilter** interface.
 - Define the **OnException** method.
- ```
public class CustomExceptionFilter : IExceptionFilter
{
 public void OnException(ExceptionContext context)
 {
 // Log exception
 context.Result = new JsonResult(new { message = "An error occurred" });
 context.ExceptionHandled = true; // Short-circuit the request
 }
}
```

### 2. Asynchronous Exception Filter

- Implement the **IAsyncExceptionFilter** interface.
  - Define the **OnExceptionAsync** method.
- ```
public class CustomAsyncExceptionFilter : IAsyncExceptionFilter
{
    public async Task OnExceptionAsync(ExceptionContext context)
    {
        // Log exception
        context.Result = new JsonResult(new { message = "An error occurred" });
        context.ExceptionHandled = true; // Short-circuit the request
        await Task.CompletedTask; // Indicate task completion
    }
}
```




```
}
```

Conclusion

- Exception filters execute only when an exception occurs in model binding, action filters, or action methods.
- They do not catch exceptions from middleware, authorization filters, resource filters, or result filters.
- For global exception handling, use exception handling middleware.
- For handling exceptions in specific controllers or actions, use exception filters.

Let's now implement this practically! 🚀

The slide has a header 'Asp.Net Core' and a footer 'Harsha'. The title 'Exception Filters' is in a red bar. A callout box says 'Runs when an exception is raised during the filter pipeline.' Below it, under 'When it runs', is a list of bullet points:

- Handles unhandled exceptions that occur in controller creation, model binding, action filters or action methods.
- Doesn't handle the unhandled exceptions that occur in authorization filters, resource filters, result filters or IActionResult execution.
- Recommended to be used only when you want a different error handling and generate different result for specific controllers; otherwise, ErrorHandlingMiddleware is recommended over Exception Filters.

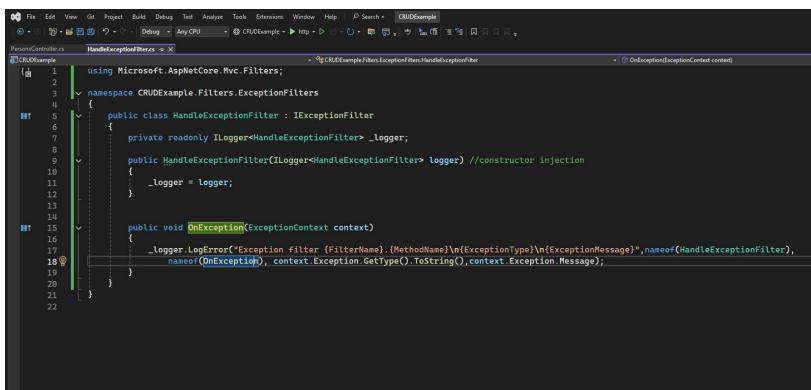
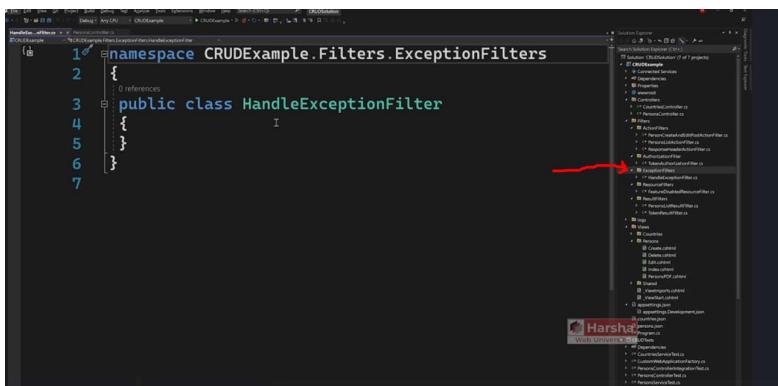
The slide has a header 'Asp.Net Core' and a footer 'Harsha'. The title 'Synchronous Exception Filter' is in a red bar. A callout box says 'Exception filter'. Below it is a code snippet:

```
public class FilterClassName : IExceptionFilter
{
    public void OnException(ExceptionContext context)
    {
        //TO DO: exception handling logic here, as follows
        context.Result = some_action_result;
        //or
        context.ExceptionHandled = true;
    }
}
```

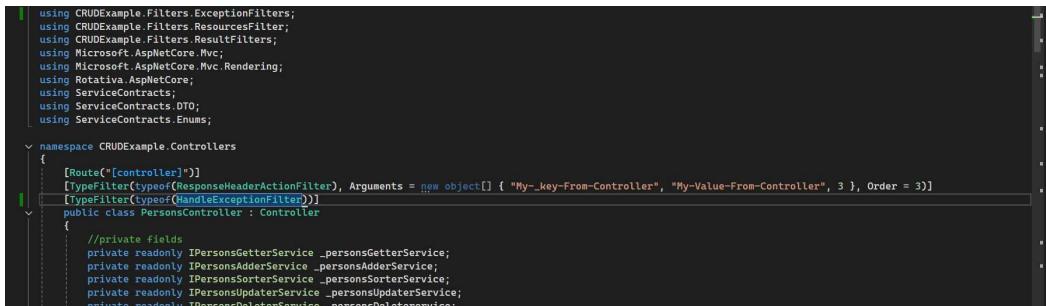

Asynchronous Exception Filter

Exception filter

```
public class FilterClassName : IAsyncExceptionFilter
{
    public async Task OnExceptionAsync(ExceptionFilterContext context)
    {
        //TO DO: exception handling logic here, as follows
        context.Result = some_action_result;
        //or
        context.ExceptionHandled = true;
        return Task.CompletedTask;
    }
}
```

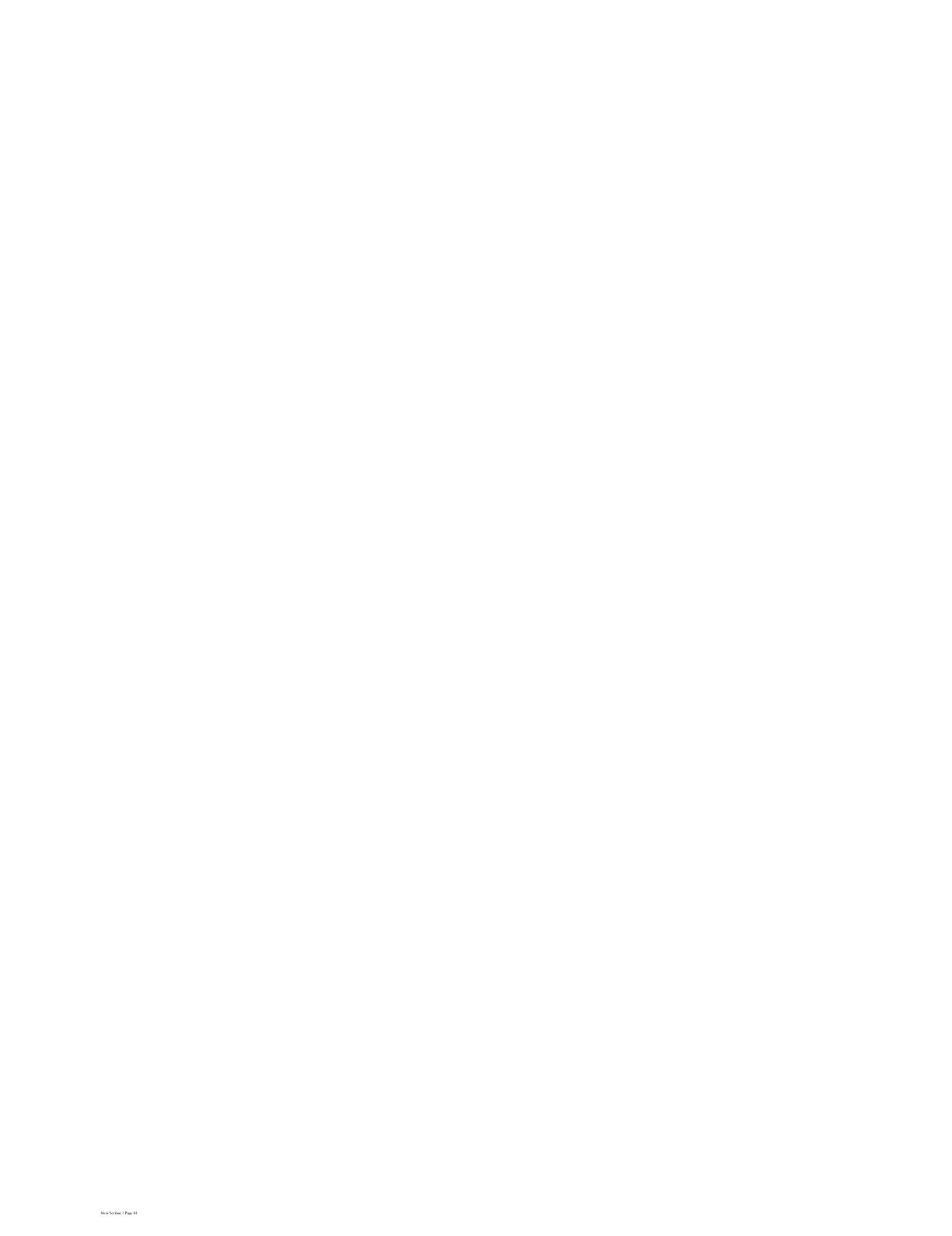


Apply that filter on 'PersonsController'



Now let's make a mistake. Let's change the database name in appSettings.json





```

appsettings.json
{
  "Logging": {
    "LogLevel": {
      "Default": "Information",
      "Microsoft.AspNetCore": "Warning"
    }
  },
  "AllowedHosts": "*",
  "ConnectionStrings": {
    "DefaultConnection": "Data Source=(localdb)\\MSSQLLocalDB;Initial Catalog=PersonsDatabase1;Integrated Security=True;Connect Timeout=30;Encrypt=False;Trust Server Cer"
  },
  "EPPlus": {
    "ExcelPackage": {
      "LicenseContext": "NonCommercial"
    }
  },
  "Serilog": {
    "MinimumLevel": "Information",
    "Using": [
      "Serilog.Sinks.Console",
      "Serilog.Sinks.File",
      "Serilog.Sinks.MSSqlServer",
      "Serilog.Sinks.Seq"
    ],
    "Properties": {}
  }
}

```

Of-course it is a wrong database name, we should get a exception. First We have reached 'Index' action method.

```

PersonController.cs
public class PersonController : Controller
{
    private readonly ILogger<PersonController> _logger;
    public PersonController(ILogger<PersonController> logger)
    {
        _logger = logger;
    }

    public void OnException(ExceptionContext context)
    {
        _logger.LogError("Exception filter {FilterName}. {MethodName}\n{ExceptionType}\n{ExceptionMessage}", nameof(HandleExceptionFilter), nameof(OnException),
            context.Exception.GetType().ToString(),
            context.Exception.Message);
    }
}

```

An unhandled exception occurred while processing the request.

```

SqlException: Cannot open database "PersonsDatabase1" requested by the login. The login failed.  

Login failed for user 'HARSHA-STUDIO\Harsha'.  

Microsoft.Data.SqlClient.TdsParser.ThrowExceptionAndWarning(TdsParserStateObject stateObj, bool callerHasConnectionLock, bool asyncClose)  

InvalidOperationException: An exception has been raised that is likely due to a transient failure. Consider enabling transient error resiliency by adding 'EnableRetryOnFailure' to the 'UseSqlServer' call.  

Microsoft.EntityFrameworkCore.Storage.Internal.SqlServerExecutionStrategy.ExecuteAsync<TState, TResult>(TState state, Func<IDbContext, TState, CancellationToken, Task<TResult>> operation, Func<IDbContext, CancellationToken, Task<ExecutionResult<TResult>> verifySucceeded, CancellationToken cancellationToken)  

Stack Query Cookies Headers Routing

```

SqlException: Cannot open database "PersonsDatabase1" requested by the login. The login failed. Login failed for user 'HARSHA-STUDIO\Harsha'.

```

Microsoft.Data.SqlClient.TdsParser.ThrowExceptionAndWarning(TdsParserStateObject stateObj, bool callerHasConnectionLock, bool asyncClose)  

Microsoft.Data.SqlClient.TdsParser.TryRun(RunBehavior runBehavior, SqlCommand cmdHandler, SqlDataReader dataStream, BulkCopySimpleResultSet bulkCopyHandler, TdsParserStateObject stateObj)  

Microsoft.Data.SqlClient.TdsParser.RunBehavior.RunBehavior(SqlCommand cmdHandler, SqlDataReader dataStream, BulkCopySimpleResultSet bulkCopyHandler, TdsParserStateObject stateObj)  

Microsoft.Data.SqlClient.TdsParser.RunBehavior.CompleteLogin(SqlObject stateObj)  

Microsoft.Data.SqlClient.TdsParserInternalConnectionTds.LoginOrFail(ServerInfo serverInfo, string newPassword, SecureString newSecurePassword, bool ignoreSqlOpenTimeout, TimeoutTimer timeout, bool withFailover)  

Microsoft.Data.SqlClient.TdsParserInternalConnectionTds.LoginOrFail(ServerInfo serverInfo, string newPassword, SecureString newSecurePassword, bool redirectedUserInstance, SqlConnectionString connectionOptions, SqlCredential credential, TimeoutTimer timeout)  

Microsoft.Data.SqlClient.TdsParserInternalConnectionTds.OpenLoginIfNotTimeout(TimeSpan timeout, SqlConnectionString connectionOptions, SqlCredential credential, string newPassword, SecureString newSecurePassword, bool redirectedUserInstance)  

Microsoft.Data.SqlClient.TdsParserInternalConnectionTds.CreateConnection(SqlConnectionString connectionOptions, SqlCredential credential, object providerInfo, string newPassword, SecureString newSecurePassword, bool redirectedUserInstance, SqlConnectionString userConnectionString, SessionData reconnectSessionData, bool applyTransientFaultHandling, string accessToken, SqlConnectionPool pool)  

Microsoft.Data.SqlClient.SqlConnectionFactory.CreateConnection(DbConnectionOptions options, DbConnectionPoolKey poolKey, object poolGroupProviderInfo, DbConnectionPool pool, DbConnection owningConnection, DbConnectionOptions userOptions)  

Microsoft.Data.ProviderBase.DbConnectionFactory.CreatePooledConnection(DbConnectionPool pool, DbConnection owningObject, DbConnectionOptions options, DbConnectionPoolKey poolKey, DbConnectionOptions userOptions)  

Microsoft.Data.ProviderBase.DbConnectionPool.CreateObject(DbConnection owningObject, DbConnectionOptions userOptions, DbConnectionInternal oldConnection)  

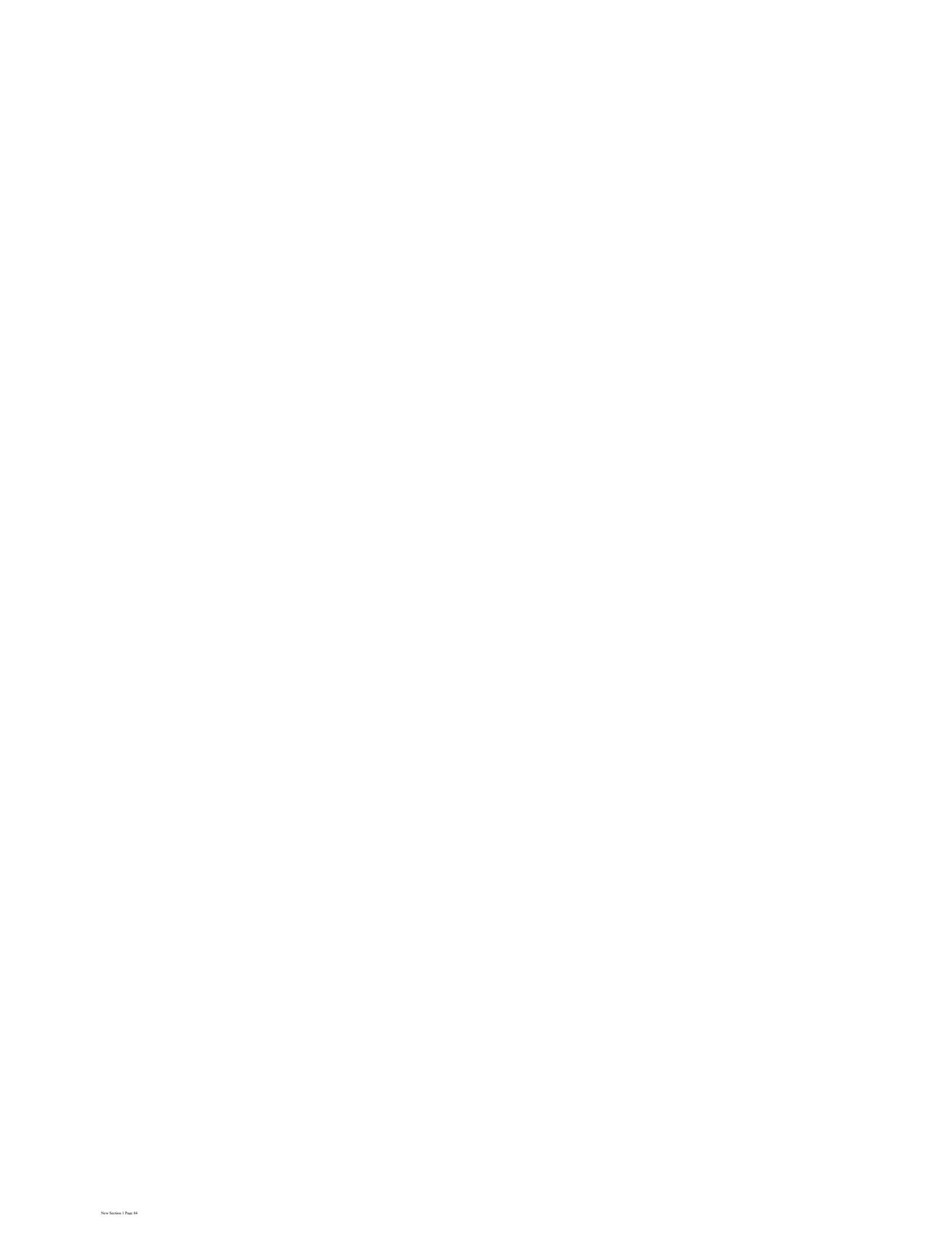
Microsoft.Data.ProviderBase.DbConnectionPool.UserCreateRequest(DbConnection owningObject, DbConnectionOptions userOptions, DbConnectionInternal oldConnection)  

Microsoft.Data.ProviderBase.DbConnectionPool.TryGetConnection(DbConnection owningObject, int userWaitForMultipleObjectsTimeout, bool allowCreate, bool onlyOneCheckConnection, DbConnectionOptions userOptions, out DbConnectionInternal connection)  

Microsoft.Data.ProviderBase.DbConnectionPool.WaitForPendingOpen()

```





Asp.Net Core

Impact of Short-Circuiting

Short-Circuiting in Filters

Do you ever wonder **what exactly happens** when you **short-circuit a request** in a filter? Let's explore this topic in detail.

Short-Circuiting in the Authorization Filter

Assume we make a request to the **server**, and it reaches the **endpoint middleware**.

- The request then **enters the authorization filters** first.
- Here, we apply **short-circuiting**, which means we **assign a non-null value** to the **Result** property.
- This value can be **any type of action result**, such as:
 - ContentResult
 - FileResult
 - JsonResult
 - Any other valid action result

What Happens When You Short-Circuit?

When short-circuiting occurs:

1. **All remaining filters and action methods are skipped.**
 - This includes **model binding, action execution, and all subsequent filters**.
 - In a visual representation, these skipped parts appear as **red-colored filters** in the filter pipeline.
2. **The result is executed immediately.**
 - The assigned result (e.g., ContentResult) **executes directly**.
 - The response is **sent to the browser immediately**, bypassing all remaining pipeline steps.

Example of Short-Circuiting in an Authorization Filter

If, as part of the authorization filter, we assign a **ContentResult** object to the **Result** property:

```
public class CustomAuthorizationFilter : IAuthorizationFilter
{
    public void OnAuthorization(AuthorizationFilterContext context)
    {
        // Short-circuiting the request
        context.Result = new ContentResult
        {
            Content = "Unauthorized access",
            StatusCode = 403 // Forbidden
        };
    }
}
```



}

In this case:

- The authorization filter assigns a ContentResult.
- All remaining filters and the action method are skipped.
- The response is immediately returned to the browser with "Unauthorized access" and a 403 Forbidden status code.

Conclusion

- Short-circuiting means skipping all remaining filters and directly returning a response.
- It is commonly used in authorization filters to prevent unauthorized access.
- By setting the Result property, you effectively bypass the rest of the pipeline and send a response immediately.

This is how short-circuiting works in filters! 🎉

Asp.Net Core

| Harsh

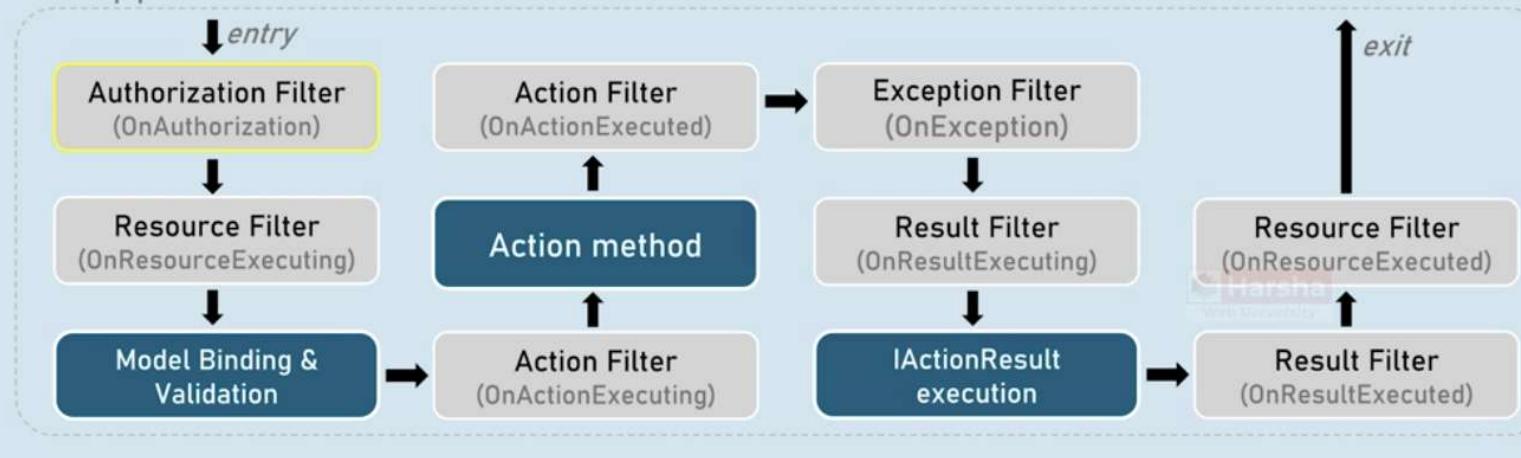
Short-circuiting Authorization Filter

Authorization Filter - OnAuthorization():

How to short-circuit

```
context.Result = some_action_result;
```

Filter pipeline



Asp.Net Core

| Harsha

Short-circuiting Authorization Filter

Authorization Filter - OnAuthorization():

How to short-circuit

```
context.Result = some_action_result;
```

Filter pipeline

sha



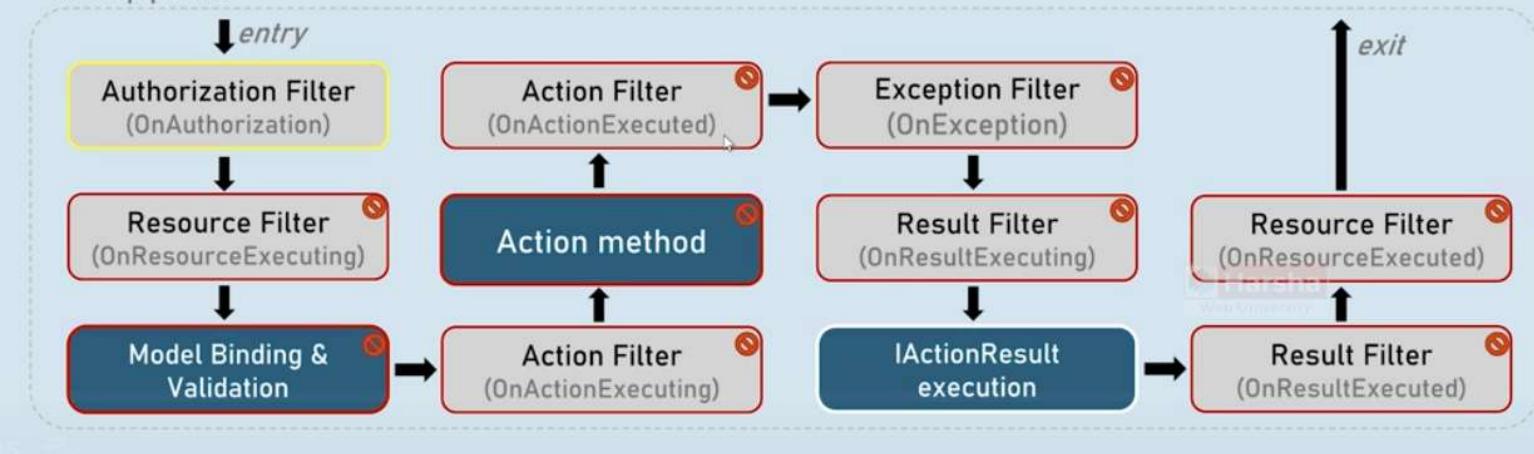
Short-circuiting Authorization Filter

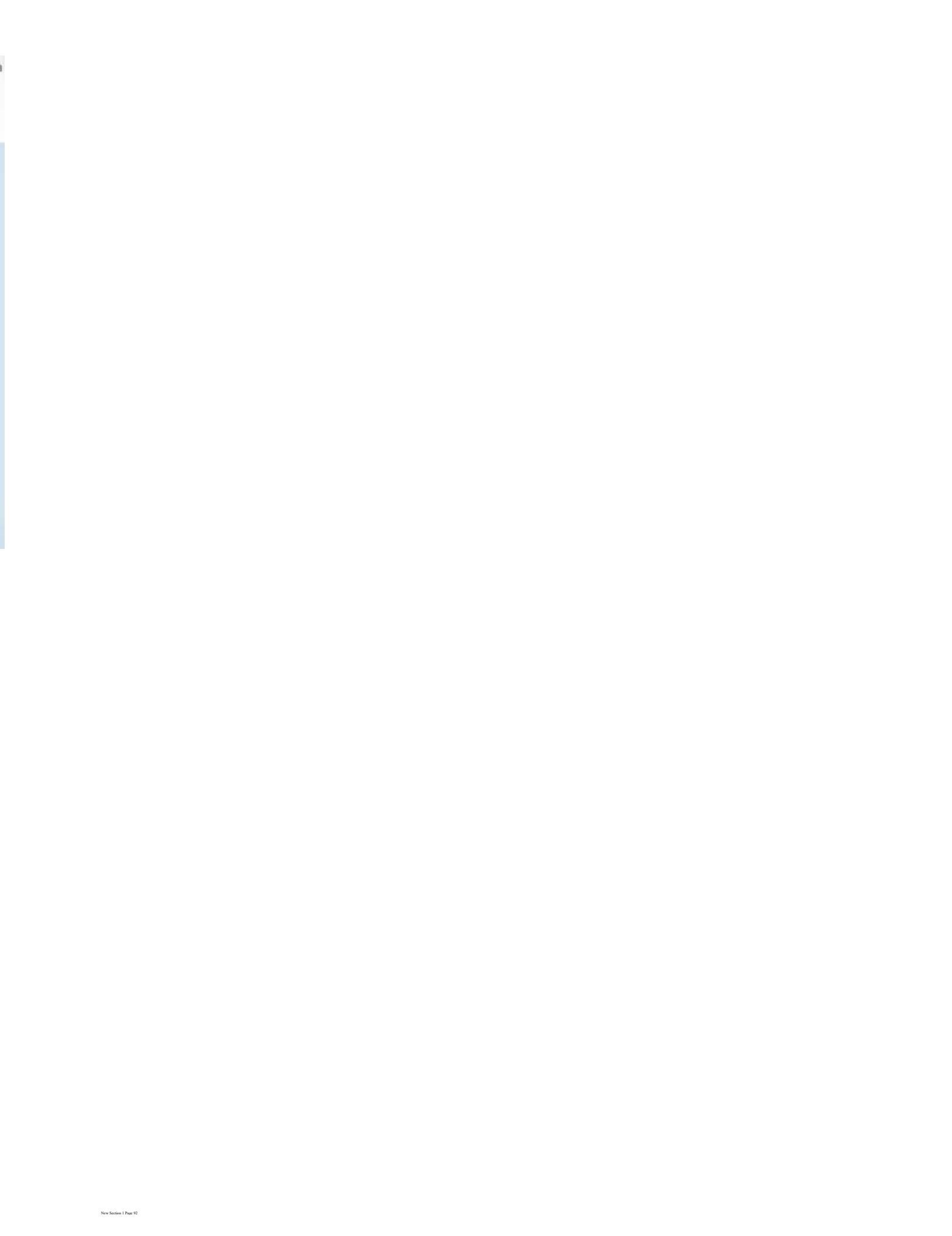
Authorization Filter - OnAuthorization():

How to short-circuit

```
context.Result = some_action_result;
```

Filter pipeline







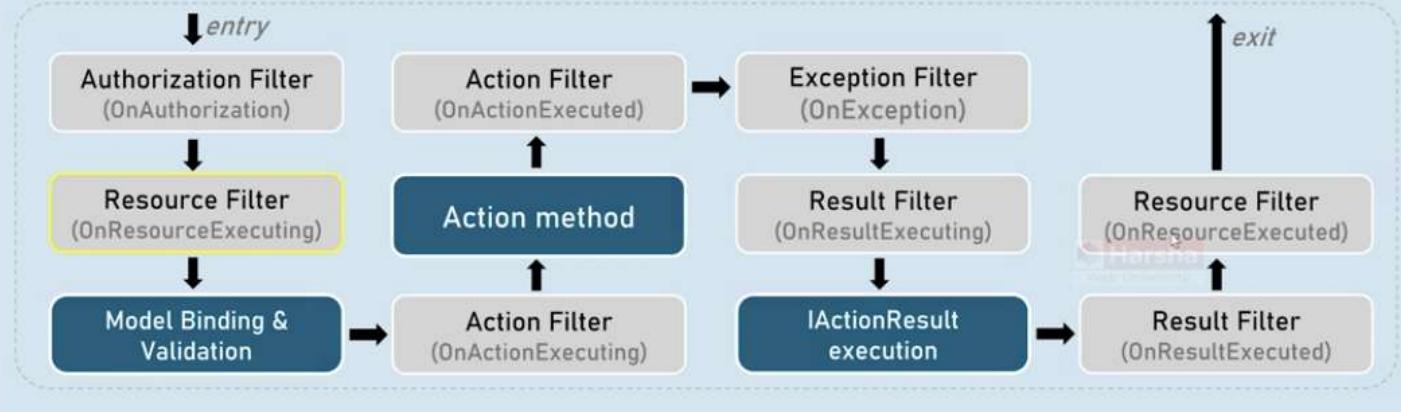
Short-circuiting Resource Filter

Resource Filter - OnResourceExecuting():

How to short-circuit

```
context.Result = some_action_result;
```

Filter pipeline



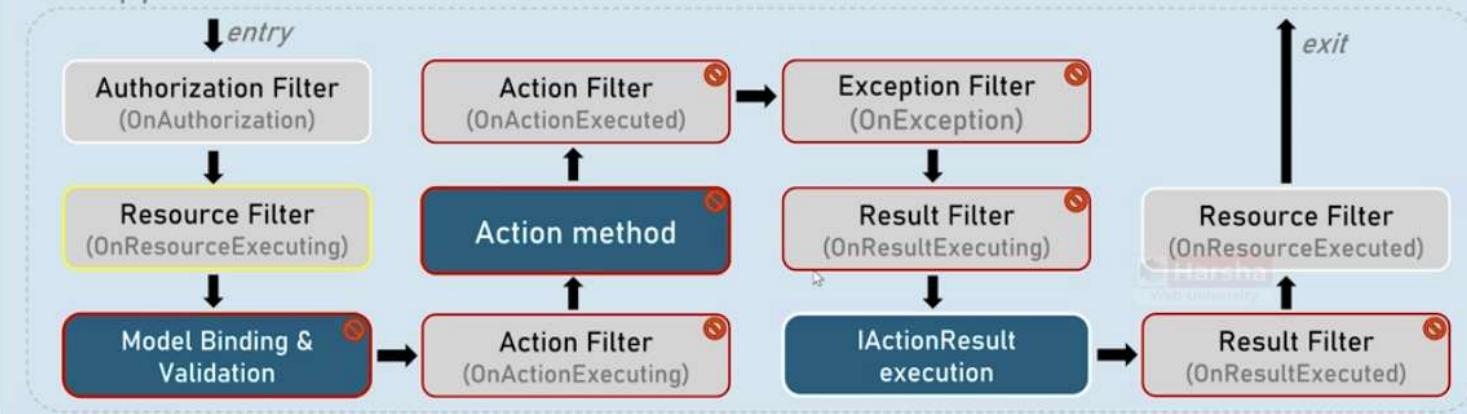
Short-circuiting Resource Filter

Resource Filter - OnResourceExecuting():

How to short-circuit

```
context.Result = some_action_result;
```

Filter pipeline



Short-Circuiting in the Resource Filter

Just like in the authorization filter, short-circuiting can also occur in the resource filter. However, it behaves differently.
Where Does Short-Circuiting Happen in a Resource Filter?

- Short-circuiting typically happens in the **OnResourceExecuting** method.
- It does not make sense to short-circuit in the **OnResourceExecuted** method because, by that time, all filters and the action method have already executed.
- Even though it's technically possible to short-circuit in **OnResourceExecuted**, there is no real benefit in doing so.

What Happens When You Short-Circuit in OnResourceExecuting?





When short-circuiting occurs in OnResourceExecuting:

1. **The model binding, action method, and all remaining filters (except OnResourceExecuted) are skipped.**
2. **The action method does not execute.**
3. **Instead, an alternative response is sent**—for example, an error message or a cached response.
4. **After the alternative response is sent, execution jumps to OnResourceExecuted**, bypassing the normal request pipeline.

Why Short-Circuit in a Resource Filter?

The main reason for short-circuiting in a resource filter is to:

- **Prevent unnecessary execution** of action methods and result filters.
- **Return a cached response** instead of reprocessing the request.
- **Send an early error response** without executing the action method.

Example of Short-Circuiting in a Resource Filter

Here's how you might implement short-circuiting in a **resource filter**:

```
public class CacheResourceFilter : IResourceFilter
{
    public void OnResourceExecuting(ResourceExecutingContext context)
    {
        // Short-circuiting the request if a cached response is available
        var cachedResponse = GetCachedResponse(context.HttpContext);
        if (cachedResponse != null)
        {
            context.Result = new ContentResult
            {
                Content = cachedResponse,
                ContentType = "application/json",
                StatusCode = 200
            };
        }
    }

    public void OnResourceExecuted(ResourceExecutedContext context)
    {
        // This executes after the response is sent
    }

    private string? GetCachedResponse(HttpContext context)
    {
        // Logic to retrieve cached response, if available
        return null; // Example: returning null means no cache found
    }
}
```

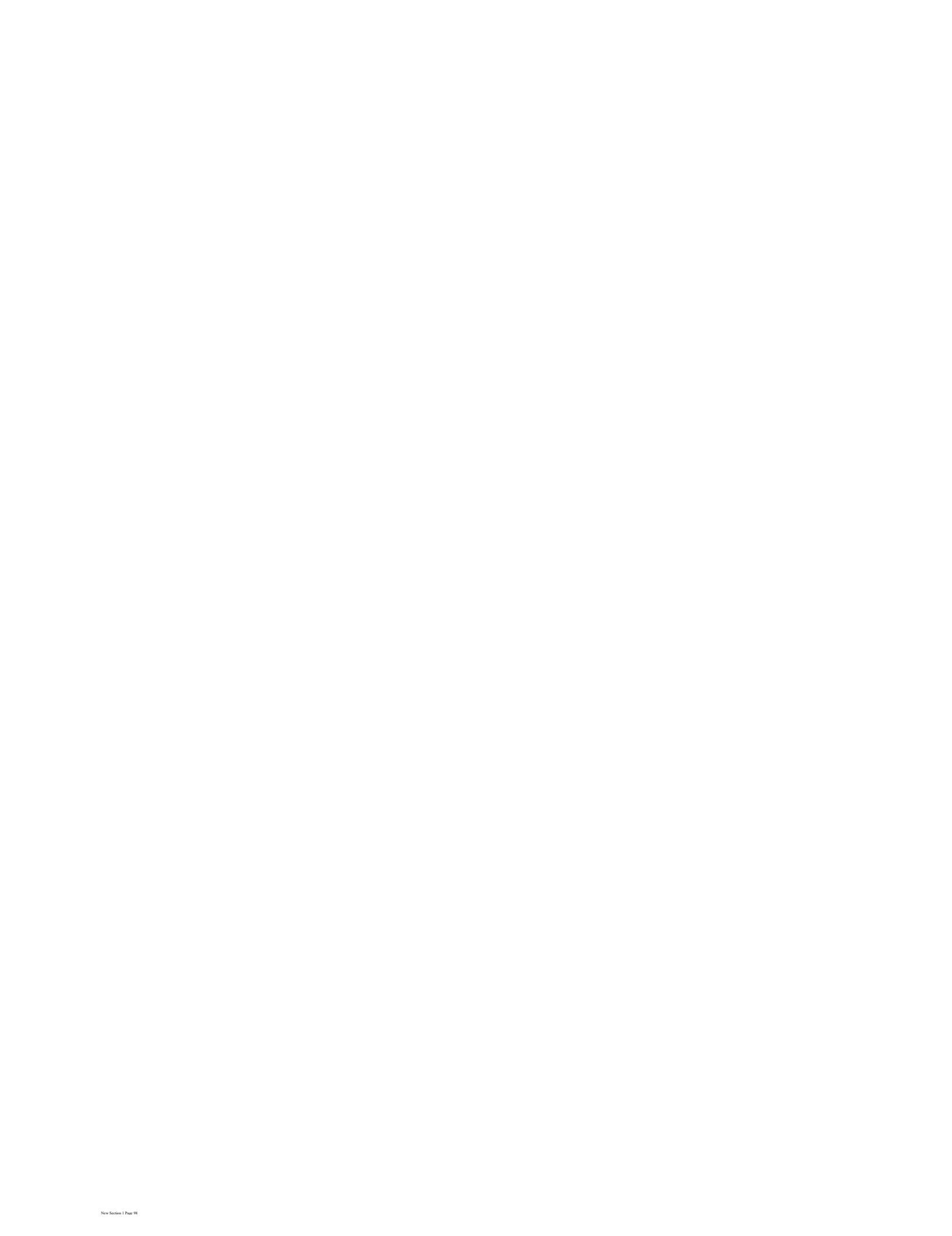
What Happens in This Example?

1. **Before executing the action method**, we check if a cached response is available.
2. If a **cached response exists**, it is **sent immediately** without executing the action method.
3. If no cache is found, the request proceeds normally.

Conclusion

- **Short-circuiting in a resource filter prevents the action method from executing.**
- **It is useful for caching mechanisms or sending early error messages.**
- **Unlike the authorization filter, the request still reaches OnResourceExecuted after short-circuiting.**

This approach **optimizes performance** by avoiding unnecessary processing when a cached response is available! 🚀





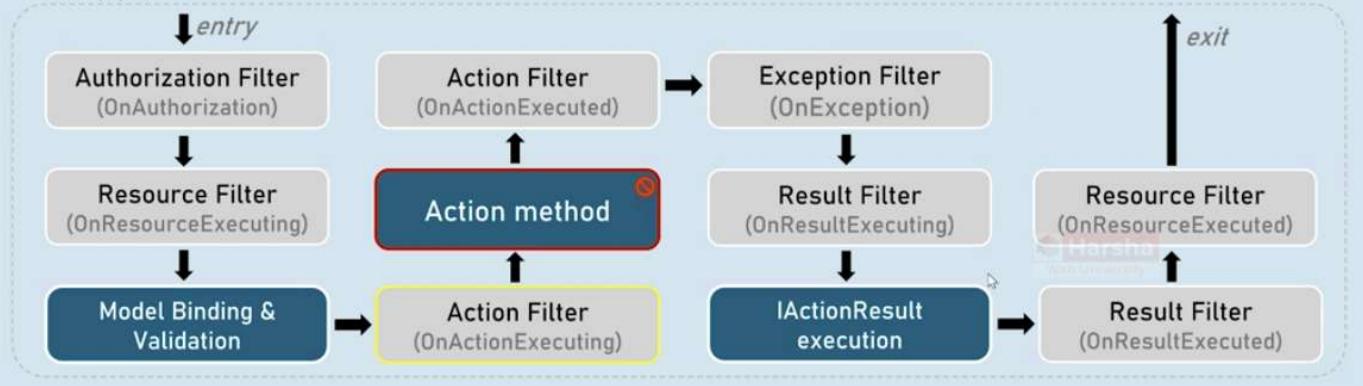
Short-circuiting Action Filter

Action Filter - OnActionExecuting():

How to short-circuit

```
context.Result = some_action_result;
```

Filter pipeline



But when a short circuit the request in the action filter, for example we have assigned result equal to some action result in this on action executing method of the action filter in this case only and only the action method of the action filter in this case only and only the action method will be short circuited. All the remaining filters and result execution runs normally.

For example as a part of this on action executing we have assigned a Json result into this result property then after that it continues executing this on action executing method in this case of an exception, then exception filter, if not it will be skipped of course then it will continue normally the result filter executing method then action method that means the Json conversion, then all the remaining filters will be executed as shown in this picture. This is the impact of short circuiting inside an action filter.

There is no point of short circuiting in the on action executed method. Because by this time already the action method has been executed. So there is no benefit of that.

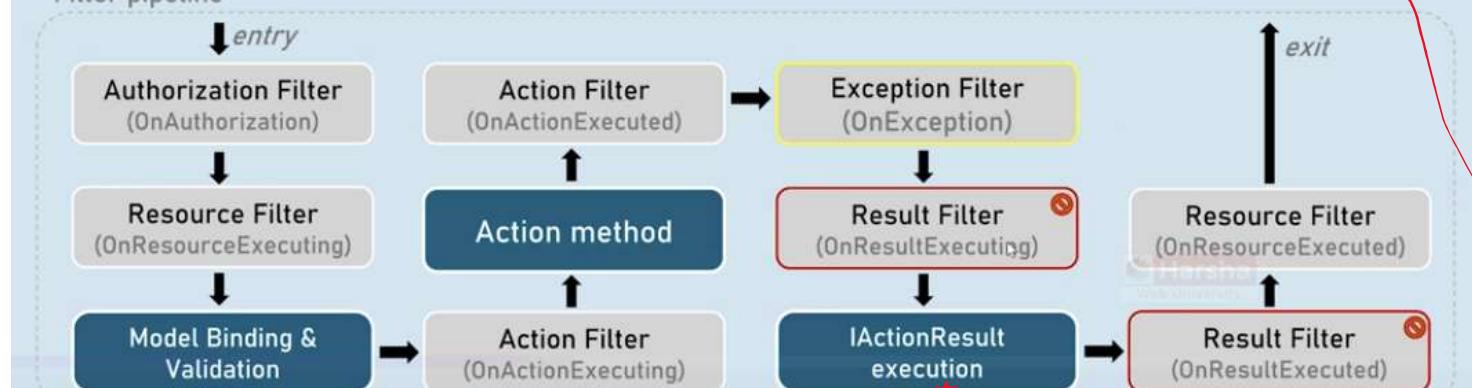
Short-circuiting Exception Filter

Exception Filter - OnException():

How to short-circuit

```
context.Result = some_action_result;  
//or  
context.ExceptionHandled = true;
```

Filter pipeline



he
ng

ecuted
n result
ter.

fit or



Now if you short circuit the request in the exception filter, for example we have assigned a Content result into the result properly or we have assigned exception handled equal to true if you do either or these it will bypass executing the result filter. But what about the results that we have assigned into this result properly that result will be executed normally.

But do remember it executes the results that we have assigned into this result property.

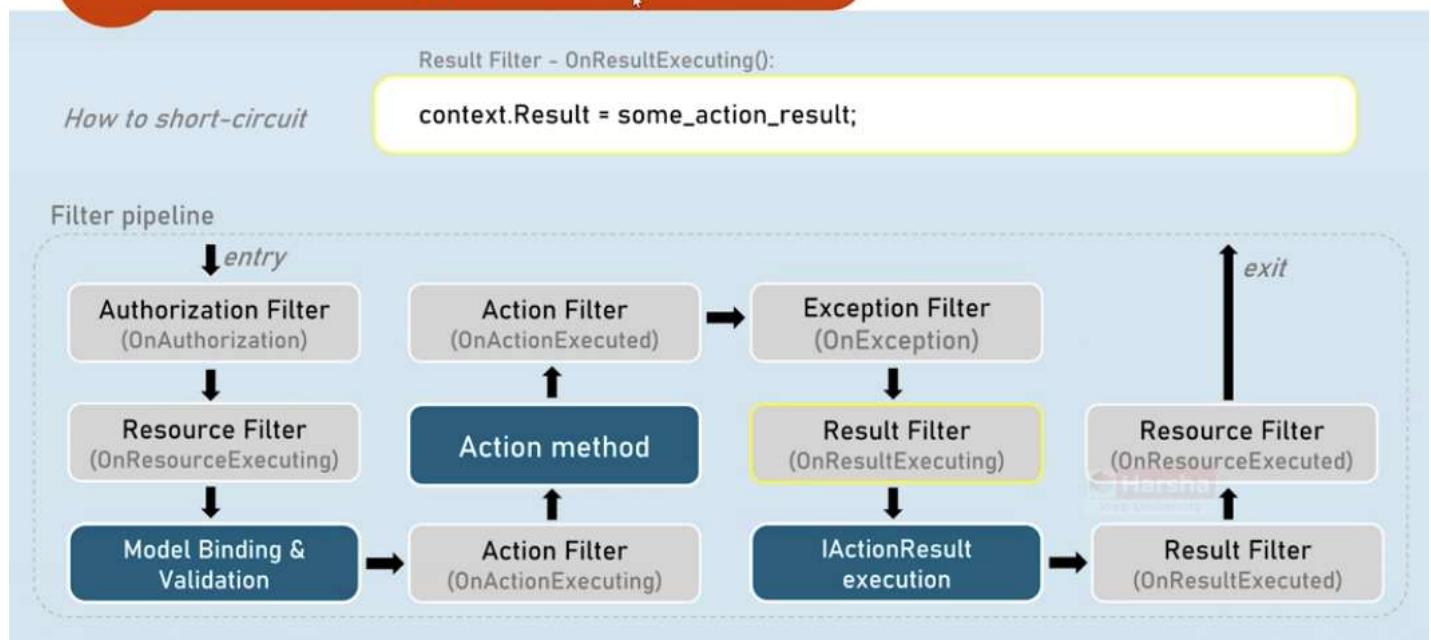
Not the actual method returned from the action method.

For example the action method returns a file result but in this exception filter we have given content result. No guess what? What Result will be executed? Here or content result? It is content result because it is the most recently returned result in this exception filter. So it overrides the previous result that was actually returned by the action method.

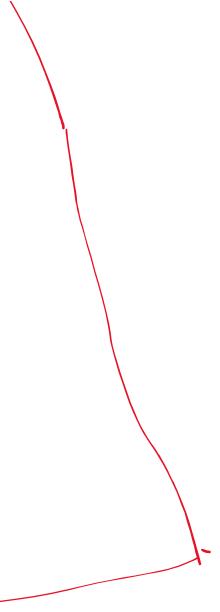
Asp.Net Core

Harsha

Short-circuiting Result Filter



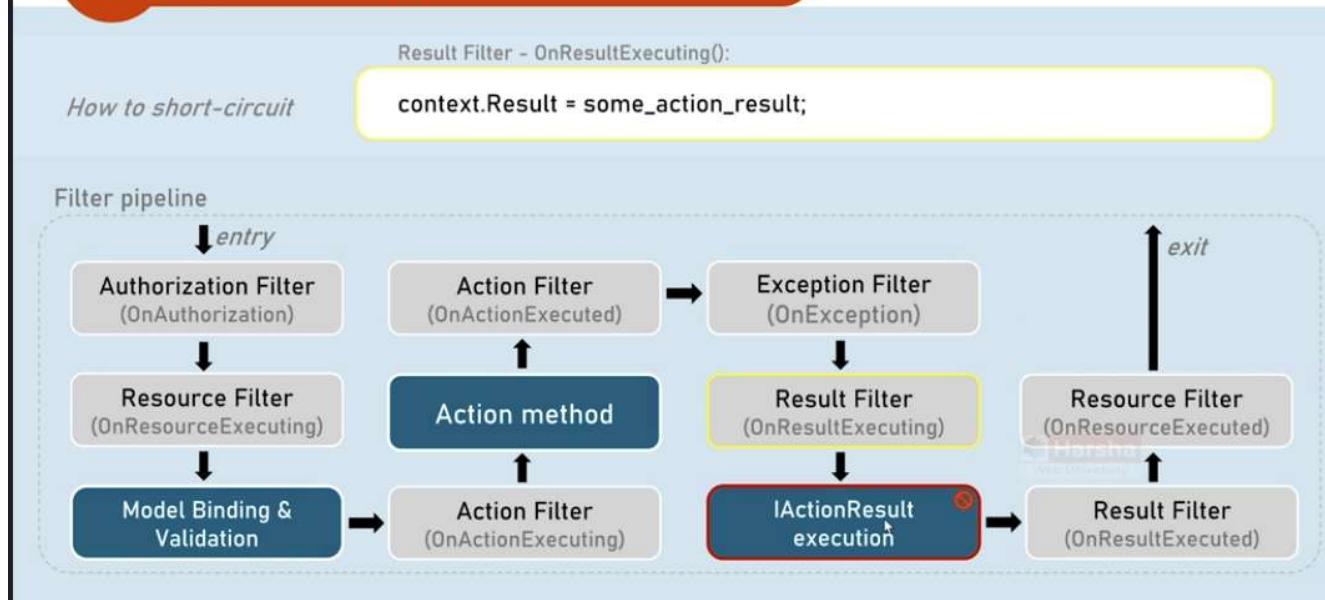
In a similar way, if you short circuit in the result filter, then it short circuits only the result execution. But the result executed method and on resource executed method will be executed normally.



either file result
rned from the

thod execute

Short-circuiting Result Filter



Here we will experience what exactly happens when you short circuit the action filter practically.

```

7   namespace CRUDExample.Filters.ActionFilters
8   {
9       public class PersonCreateAndEditPostActionFilter : IAsyncActionFilter
10      {
11          private readonly ICountriesGetterService _countriesGetterService;
12          private readonly ILogger<PersonCreateAndEditPostActionFilter> _logger;
13
14          public PersonCreateAndEditPostActionFilter(ICountriesGetterService countriesGetterService, ILogger<PersonCreateAndEditPostActionFilter> logger)
15          {
16              _countriesGetterService = countriesGetterService;
17              _logger = logger;
18          }
19
20          public async Task OnActionExecutionAsync(ActionExecutingContext context, ActionExecutionDelegate next)
21          {
22              //TO DO: before logic
23              if(context.Controller is PersonsController personsController)
24              {
25                  if (!personsController.ModelState.IsValid) //before executing this controller method, model validation gets executed
26                  {
27                      List<CountryResponse> countries = await _countriesGetterService.GetAllCountries();
28                      personsController.ViewBag.Countries = countries.Select(temp => new SelectListItem() { Text = temp.CountryName, Value = temp.CountryID.ToString()});
29
30                      personsController.ViewBag.E // dynamic Represents an object whose operations will be resolved at runtime.
31                      Many(v => v.Errors).Select(e => e.ErrorMessage).ToList();
32
33                      var personAddRequest = context.ActionArguments["personRequest"];
34
35                      context.Result = personsController.View(personAddRequest); // short-circuits or skips the subsequent action filters & action method
36                  }
37                  else
38                  {
39                      await next(); //invokes the subsequent filters or action method
40                  }
41              }
42              else
43              {
44                  await next(); //invokes the subsequent filters or action method
45              }
46              //TO DO: after logic
47
48              _logger.LogInformation("In after logic of PersonsCreateAndEdit Action filter");
49          }
50      }

```

Let's run our application. Click on 'Create'. We have already disabled front-end side validation.

The screenshot shows a web application titled 'Create Person' on a page 'localhost:5298/Persons/Create'. The form contains fields for Person Name, Email, Date of Birth (dd/mm/yyyy), Gender (Male, Female, Other), Country (dropdown menu with 'Please Select'), Address, and a checkbox for 'Receive News Letters'. Below the form is a green 'Create' button. In the top right corner, there is a watermark for 'Harsha Web University'.

That request will be received by 'OnResourceExecutionAsync' method. Click on continue button.

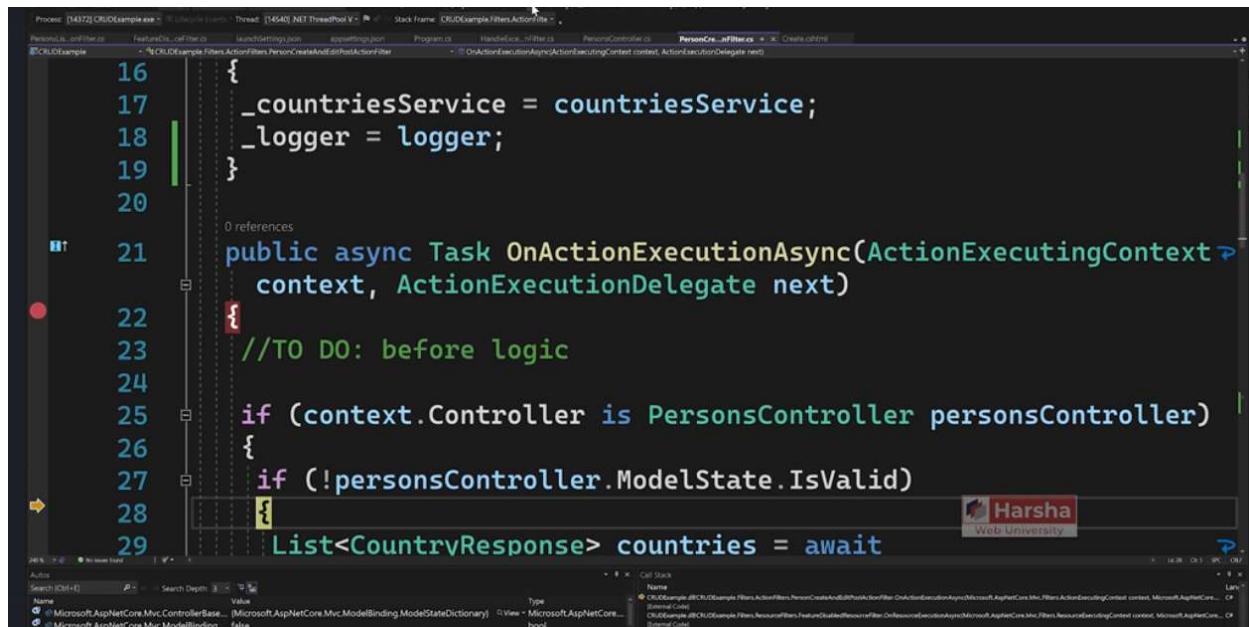
```

15     _logger = logger;
16     _isEnabled = isEnabled;
17 }
18 public async Task OnResourceExecutionAsync
19     (ResourceExecutingContext context, ResourceExecutionDelegate next)
20 {
21     //TO DO: before logic
22     _logger.LogInformation("{FilterName}. {MethodName} - before",
23         nameof(FeatureDisabledResourceFilter), nameof(
24             OnResourceExecutionAsync));
25
26     if (_isEnabled)
27     {
28         await next();
29     }
30 }

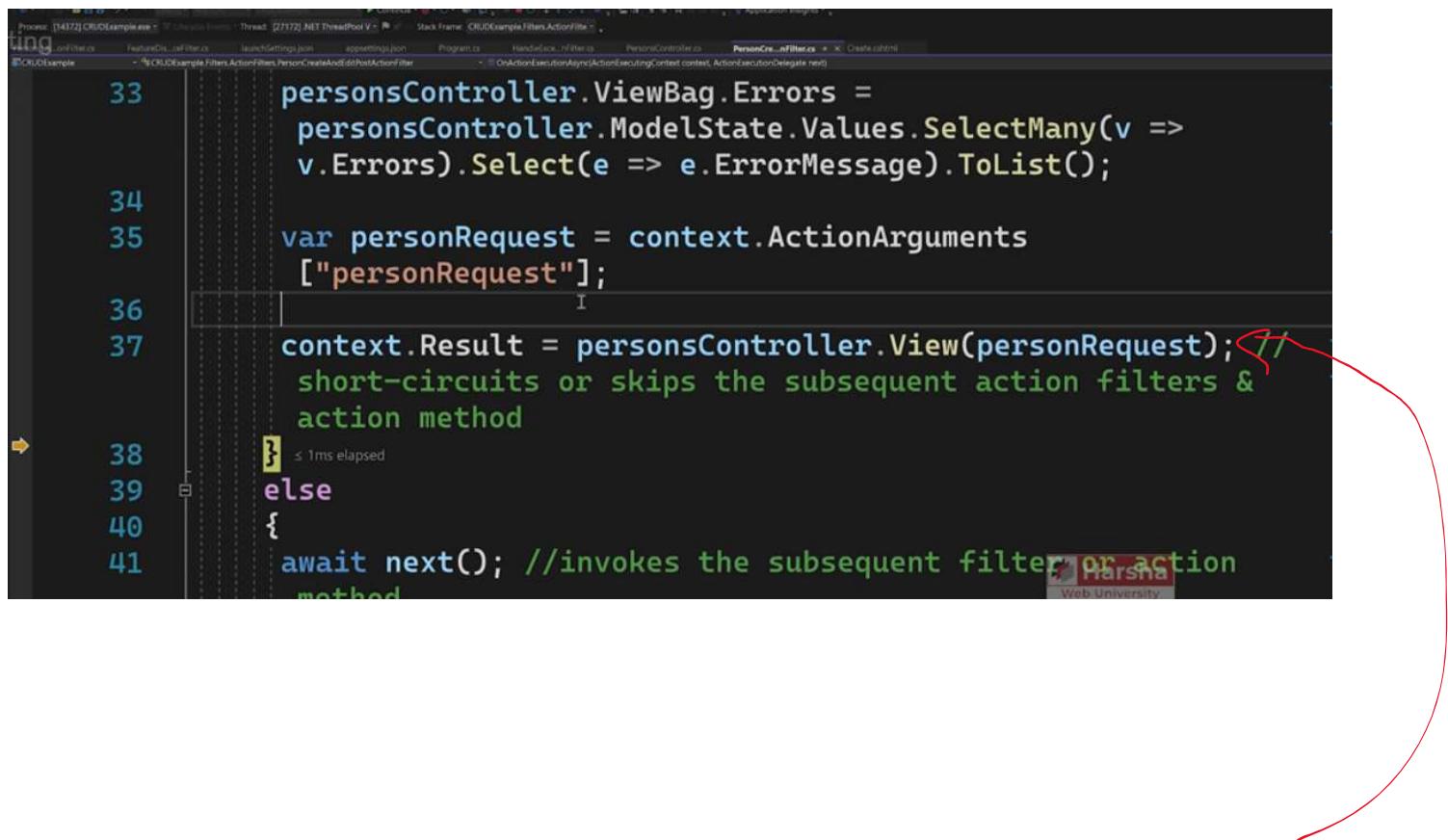
```

The screenshot shows a debugger interface with the code for the 'OnResourceExecutionAsync' method. The code is written in C# and uses the await keyword. A tooltip 'RESOURCE EXECUTING METHOD' is displayed near the cursor. The debugger shows the current line of code being executed.

We are inside the 'Action Filter'.



```
16
17     _countriesService = countriesService;
18     _logger = logger;
19
20
21     public async Task OnActionExecutionAsync(ActionExecutingContext context, ActionExecutionDelegate next)
22     {
23         //TO DO: before logic
24
25         if (context.Controller is PersonsController personsController)
26         {
27             if (!personsController.ModelState.IsValid)
28             {
29                 List<CountryResponse> countries = await
```



```
33             personsController.ViewBag.Errors =
34                 personsController.ModelState.Values.SelectMany(v =>
35                     v.Errors).Select(e => e.ErrorMessage).ToList();
36
37             var personRequest = context.ActionArguments
38                 ["personRequest"];
39
40             context.Result = personsController.View(personRequest); // short-circuits or skips the subsequent action filters &
41             // action method
42         }
43     }
44
45     else
46     {
47         await next(); //invokes the subsequent filter or action
48             method
```



```
38     }
39     else
40     {
41         await next(); //invokes the subsequent filter or action
42         method
43     }
44     else
45     {
46         await next(); //calls the subsequent filter or action method
47     }
48
49 //TO DO: after logic
50 _logger.LogInformation("In after logic of PersonsCreateAndEdit Action filter");

```

See currently we are in the after logic. This is equivalent to 'OnActionExecuted' method.

Click continue. We are in the Resource Filter.

```
1  using Microsoft.AspNetCore.Mvc;
2  using Microsoft.AspNetCore.Mvc.Filters;
3
4  namespace CRUDExample.Filters.ResourceFilters
5  {
6
7      public class FeatureDisabledResourceFilter : IAsyncResourceFilter
8      {
9          private readonly ILogger<FeatureDisabledResourceFilter>
10             _logger;
11          private readonly bool _isEnabled;
12
13      public FeatureDisabledResourceFilter

```

see currently we are inside the feature


```
    context.Result = new StatusCodeResult(501); //501 - NOT Implemented
}
else
{
    await next();
}

//TO DO: after logic
_logger.LogInformation($"{FilterName}.{MethodName} - after",
    nameof(FeatureDisabledResourceFilter), nameof(
    (OnResourceExecutionAsync)));
}
```

it executes after logic of the resource

Asp.Net Core

Harsha

Short-circuiting the filters

How to short-circuit?

Authorization filter:

```
context.Result = some_action_result;
```

What else runs?

Bypasses all filters, action method execution & result execution.

How to short-circuit?

Resource filter:

```
context.Result = some_action_result;
```

What else runs?

Bypasses model binding, action filters, action method, result execution & result filters.

Resource filters' "Executed" methods run with `context.Cancelled = true`.

Short-circuiting the filters

Action filter:

How to short-circuit?

```
context.Result = some_action_result;
```

What else runs?

Bypasses only action method execution.

Other action filters' "Executed" methods with context.Cancelled = true; and also all result filters, resource filters run normally.

Exception filter:

```
context.Result = some_action_result;  
//or  
context.ExceptionHandled = true;
```

Bypasses result execution & result filters.

All resource filters' "Executed" methods run.

Short-circuiting the filters

Result filter:

How to short-circuit?

```
context.Result = some_action_result;
```

What else runs?

Bypasses only result execution.

Other result filters' "Executed" methods & all resource filters run normally.

Asp.Net Core

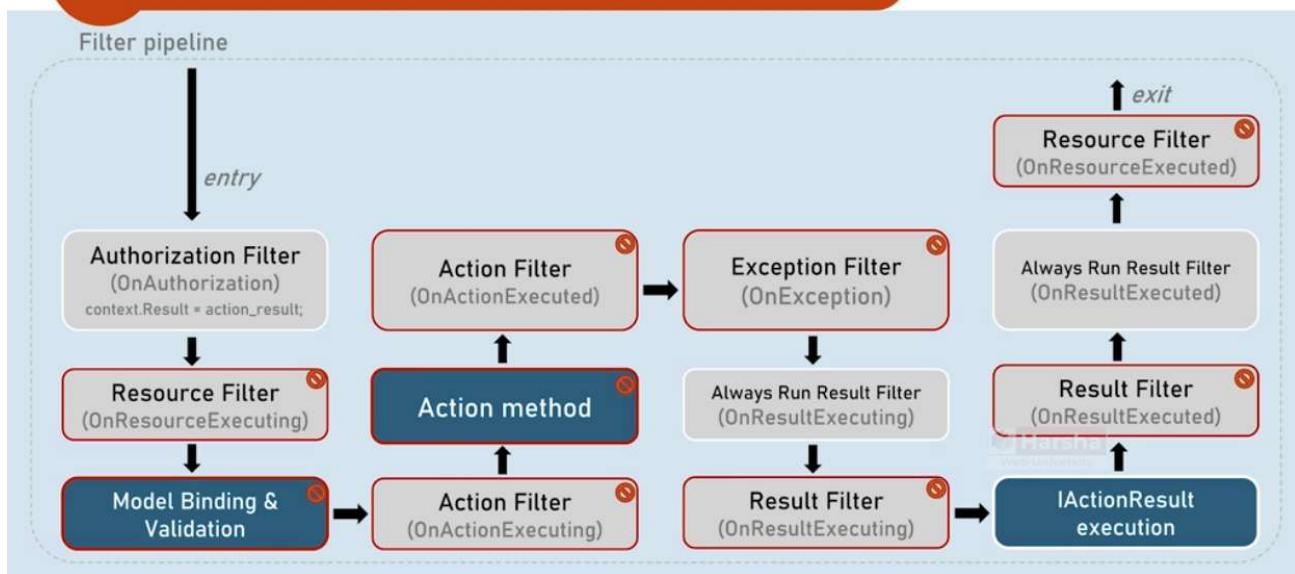
IAlwaysRunResultFilter

I have a question for you. Does the filter execute in case when you short circuit the authorization filter or in other filters ? Mostly no Right?

Asp.Net Core

Harsha

Short-circuiting Authorization Filter



in

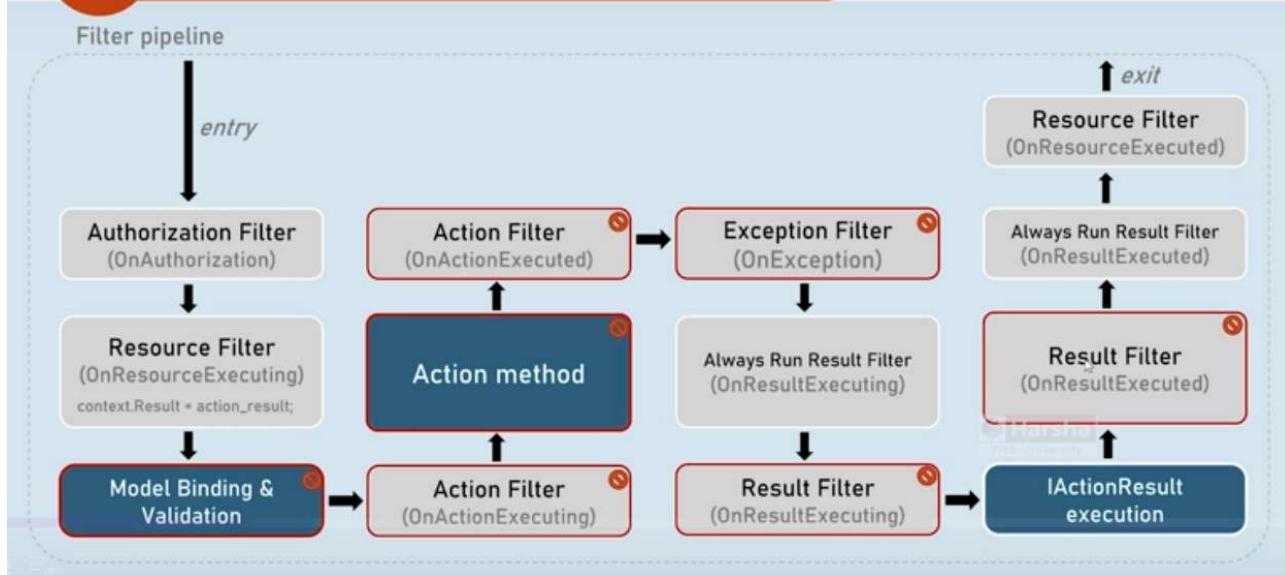
Short-circuiting Exception Filter



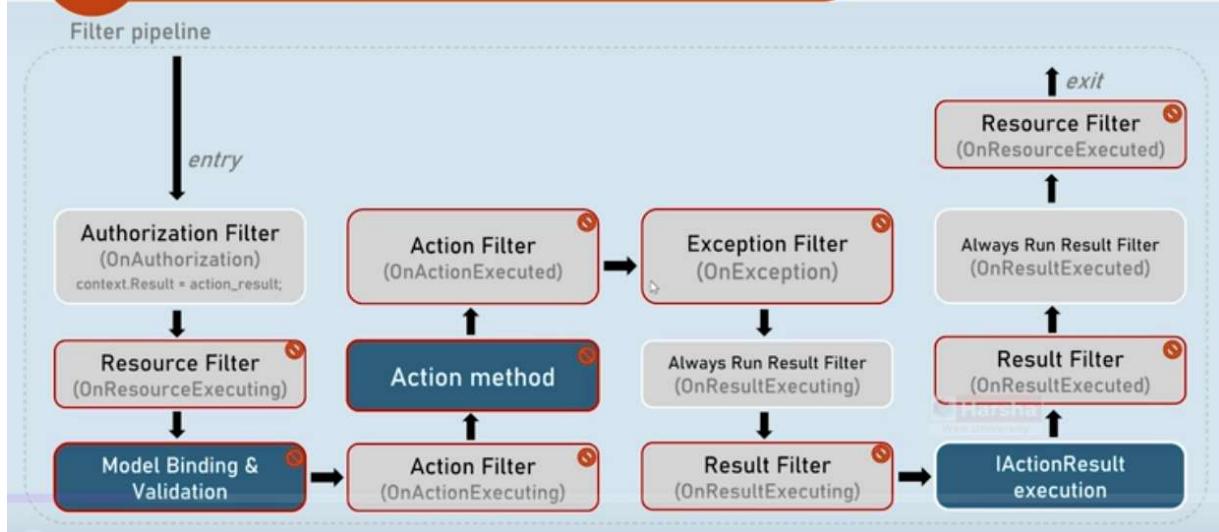
Short-circuiting Authorization Filter



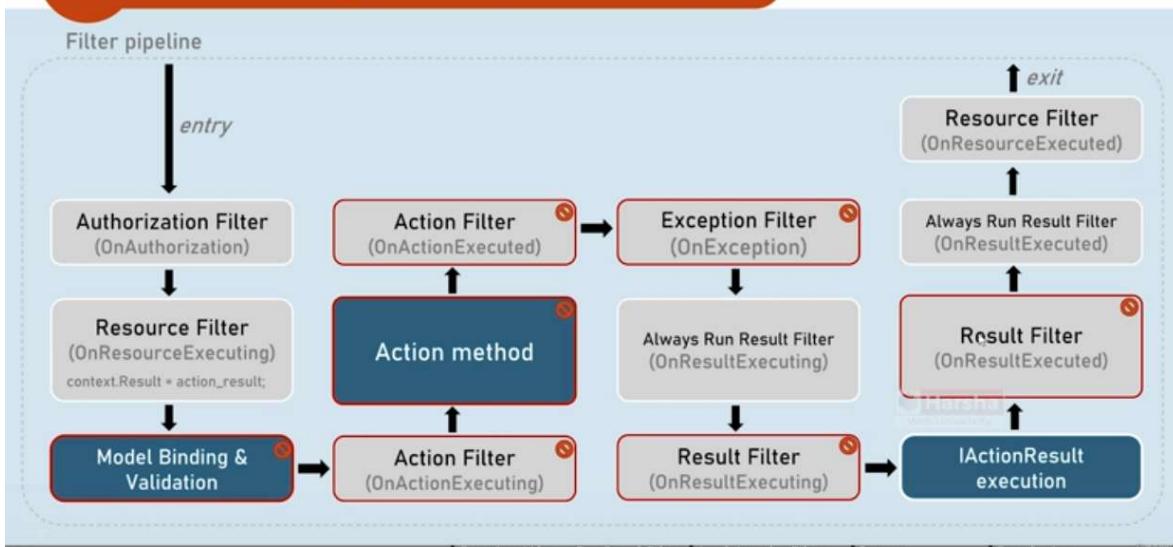
Short-circuiting Resource Filter



Short-circuiting Authorization Filter



Short-circuiting Resource Filter



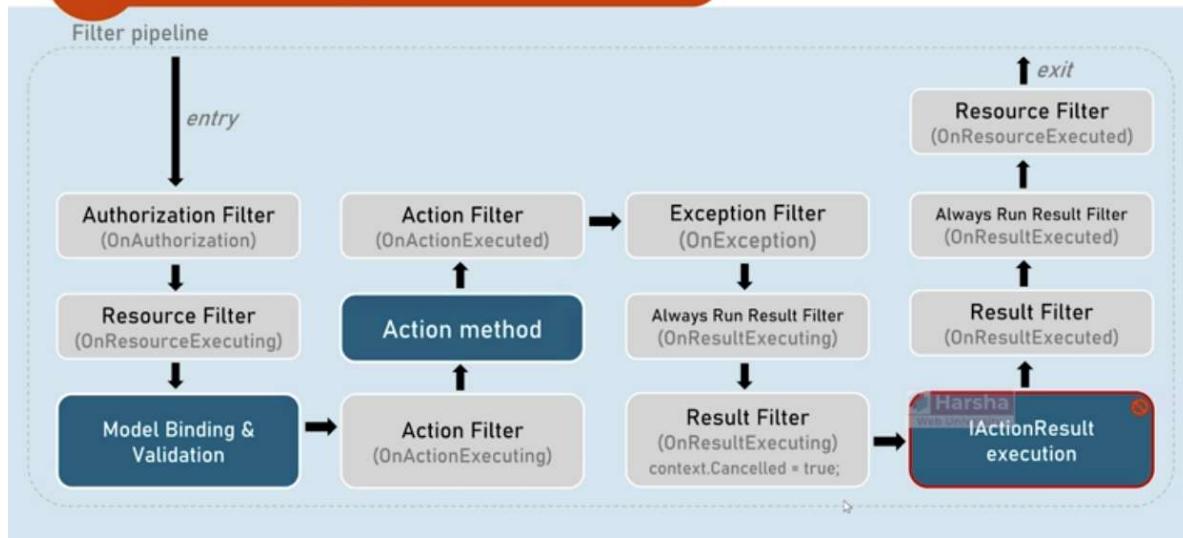
Short-circuiting Action Filter



Short-circuiting Exception Filter



Short-circuiting Result Filter



Does the Result Filter Execute When You Short-Circuit in Other Filters?

In most cases, when you short-circuit in the Authorization, Resource, or Exception filter, the Result Filter does NOT execute.

☞ The "Always Run" Result Filter executes even when the request is short-circuited.

Breakdown of Short-Circuiting & Result Filter Execution

① Short-Circuiting in the Authorization Filter

- When you short-circuit in the Authorization filter, all subsequent filters, including Action filters, Result filters, and the Action method itself, are skipped.
- However, the Always Run Result Filter will still execute.

•T execute. However, there is an **exception** to this rule:

2 Short-Circuiting in the Resource Filter

- When you short-circuit in the Resource filter, the request skips:
 - Model binding
 - Action method execution
 - Other Result filters
- However, the Always Run Result Filter will execute.

3 Short-Circuiting in the Exception Filter

- When a request is short-circuited in the Exception filter, the normal Result filter does not execute.
- But the Always Run Result Filter still executes.

4 Short-Circuiting in the Result Filter

- If you short-circuit in the OnResultExecuting method of a Result Filter,
 - The IActionResult execution is skipped
 - But the Always Run Result Filter will execute

What is the Always Run Result Filter?

- The Always Run Result Filter is a special type of Result Filter that executes no matter what—even if the request is short-circuited.
- It wraps around the normal Result filter, ensuring it always runs before and after the result execution.

Why Use the Always Run Result Filter?

You might want to execute certain actions unconditionally, such as:

- Clearing cache on the server
- Removing cookies
- Resetting performance counters
- Logging performance metrics
- Releasing resources

Example: Creating an Always Run Result Filter

Here's how you can implement an Always Run Result Filter in ASP.NET Core

```
public class AlwaysRunResultFilter : IAlwaysRunResultFilter
{
    public void OnResultExecuting(ResultExecutingContext context)
    {
        // Code that always runs before the result executes
        Console.WriteLine("Always Run Result Filter - Before Execution");
    }

    public void OnResultExecuted(ResultExecutedContext context)
    {
        // Code that always runs after the result executes
        Console.WriteLine("Always Run Result Filter - After Execution");
    }
}
```

Registering the Always Run Result Filter in ASP.NET Core

To apply the filter globally, add it in Program.cs:

```
builder.Services.AddControllers(options =>
{
    options.Filters.Add<AlwaysRunResultFilter>();
});
```

Conclusion

- If you short-circuit in Authorization, Resource, or Exception filters, normal Result filters do NOT execute.
- The "Always Run Result Filter" still executes, no matter what.
- Use it when you need to execute critical cleanup tasks like cache clearing, logging, or resource cleanup.
- This ensures that important finalization logic runs even when requests are short-circuited!

Code Implementation:


```

1  using Microsoft.AspNetCore.Mvc.Filters;
2
3  namespace CRUDEExample.Filters.ResultFilters
4  {
5      public class PersonsAlwaysRunResultFilter : IAlwaysRunResultFilter
6      {
7          public void OnResultExecuted(ResultExecutedContext context)
8          {
9          }
10
11         public void OnResultExecuting(ResultExecutingContext context)
12         {
13         }
14     }
15 }
16
17

```

In our project, we have commented out at line no 1. So it will not generate and send cookie to Browser. When we click on edit from person list page, no 1 marker

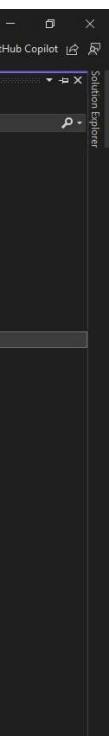
Then after we click on the 'Update' button from edit page. 2 no filter will be executed. So we are short circuited at 'AuthorizationFilter'. Now let's see if our 'Alw

```

89
90
91
92 [HttpGet]
93 [Route("[action]/[personID]")] //Eg: /persons/edit/1
94 // [TypeFilter(typeof(TokenResultFilter))] ←
95 public async Task<IActionResult> Edit(Guid personID) ...
96
97
98
99
100
101
102 [HttpPost]
103 [Route("[action]/[personID]")]
104 [TypeFilter(typeof(PersonCreateAndEditPostActionFilter))]
105 [TypeFilter(typeof(TokenAuthorizationFilter))] ←
106 public async Task<IActionResult> Edit(PersonUpdateRequest
107 personRequest)
108 {
109     PersonResponse? personResponse = await
110     _personsService.GetPersonByPersonID(personRequest.PersonID);
111 }
112
113
114
115
116
117
118

```

Updated Code:



d action method will execute.

'aysRunResultFilter' gets executed or not.


```

//on clicking the submit button from edit page
[HttpPost]
[Route("[action]/{personID}")]
[TypeFilter(typeof(PersonCreateAndEditPostActionFilter))]
[TypeFilter(typeof(TokenAuthorizationFilter))]
[TypeFilter(typeof(PersonsAlwaysRunResultFilter))]
public async Task<IActionResult> Edit(PersonUpdateRequest personRequest)
{
    PersonResponse? personResponse = await _personsGetterService.GetPersonByPersonId(personRequest.PersonID);

    if (personResponse == null)
    {
        return RedirectToAction("Index");
    }

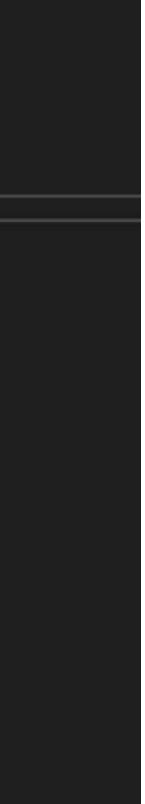
    //if (ModelState.IsValid)
    //{
    PersonResponse updatedPerson = await _personsUpdaterService.UpdatePerson(personRequest);
    return RedirectToAction("Index");
    //}
    //else
    //{
    //    List<CountryResponse> countries = await _countriesGetterService.GetAllCountries();
    //    ViewBag.Countries = countries.Select(temp =>
    //    new SelectListItem() { Text = temp.CountryName, Value = temp.CountryID.ToString() });
}

```

The screenshot shows an 'Edit Person' form with the following fields and values:

- Person Name: Angie
- Email: asarvar3@dropbox.com
- Date of Birth: 09/01/1987
- Gender: Male (radio button selected)
- Country: China
- Address: 83187 Merry Drive
- Receive News Letters

A red arrow points from the "Receive News Letters" label to the "Update" button.



The screenshot shows the Visual Studio IDE with the code editor open. The file is CRUDEExample.Filters.AuthorizationFilter.cs. The code defines a class TokenAuthorizationFilter that implements the IAuthorizationFilter interface. It checks if a cookie named "Auth-Key" exists in the request. If it doesn't, it returns a 401 Unauthorized response.

```
4
5 namespace CRUDEExample.Filters.AuthorizationFilter
6 {
7     public class TokenAuthorizationFilter : IAuthorizationFilter
8     {
9         public void OnAuthorization(AuthorizationFilterContext context)
10        {
11            if (context.HttpContext.Request.Cookies.ContainsKey("Auth-
12            Key") == false)
13            {
14                context.Result = new StatusCodeResult
15                (StatusCodes.Status401Unauthorized);
16                return;
17            }
18        }
19    }
20 }
```

Call Stack:

- Name: CRUDEExample.CRUDExample.Filters.AuthorizationFilter.TokenAuthorizationFilter.OnAuthorization<Microsoft.AspNetCore.Mvc.Filters.AuthorizationFilterContext> Line 11

Auto

Search (Ctrl+F)

Call Stack

Name

Type

context

context.HttpContext.Request.Cookies

Microsoft.AspNetCore.Mvc.Infrastructure.ResourceInvoker.AuthorizationFilterContext

Microsoft.AspNetCore.Http.RequestCookieCollection

View - Microsoft.AspNetCore...

Harsha

Web University

The screenshot shows the Visual Studio IDE with the code editor open. The file is CRUDEExample.Filters.ResultFilter.cs. The code defines a class PersonAlwaysRunResultFilter that implements the IAlwaysRunResultFilter interface. It has two methods: OnResultExecuting and OnResultExecuted, both of which are currently empty.

```
1 reference
2
3
4
5 public class PersonAlwaysRunResultFilter : IAlwaysRunResultFilter
6 {
7     public void OnResultExecuted(ResultExecutingContext context)
8     {
9     }
10
11     public void OnResultExecuting(ResultExecutingContext context)
12     {
13     }
14 }
15
16 }
```

Call Stack:

- Name: CRUDEExample.CRUDExample.Filters.ResultFilter.PersonAlwaysRunResultFilter.OnResultExecuting<Microsoft.AspNetCore.Mvc.Filters.ResultFilterContext> Line 12

Auto

Search (Ctrl+F)

Call Stack

Name

Type

PersonAlwaysRunResultFilter

ResultExecutingContext

Microsoft.AspNetCore...

Harsha

Web University

Always Run Result Filters

When it runs

Runs immediately before and after result filters.

Result filters:

Doesn't execute when authorization filter, resource filter or exception filter short-circuits.

AlwaysRunResult filter:

Execute always even when authorization filter, resource filter or exception filter short-circuits.

'OnResultExecuting'
method

Same as Result filter

'OnResultExecuted'
method

Same as Result filter

Synchronous Always Run Result Filter

Synchronous Always Run Result filter

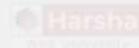
```
public class FilterClassName : IAlwaysRunResultFilter
{
    public void OnResultExecuting(ResultExecutingContext context)
    {
        //TO DO: before logic here
    }

    public void OnResultExecuted(ResultExecutedContext context)
    {
        //TO DO: after logic here
    }
}
```


Asynchronous Always Run Result Filter

Asynchronous Always Run Result Filter

```
public class FilterClassName : IAsyncAlwaysRunResultFilter
{
    public async Task OnResultExecutionAsync(ResultExecutingContext context,
                                              ResultExecutionDelegate next)
    {
        //TO DO: before logic here
        await next();
        //TO DO: after logic here
    }
}
```



Asp.Net Core Filter Overrides

Did you think of this scenario

Assume that we have applied the filter at the controller level or at the global level but we want to skip its execution for a specific action.



a particular action method.

Let's say in the controller we have two actions, but we don't want to execute that filter for the action 2 we want to skip it.

But how is that possible to skip the filter for the particular action method?
That concept is called as **filter overrides**.

To be frank there is no any straightforward feature in order to implement this functionality in ASP.NET Core.

But there is a possible workaround.

You have to write your own **if statement** in your filter to identify whether the particular attribute is applied for the action.

If it is applied for example, we have created a **skip filter attribute**, if this attribute is applied for the action method then portion of the code will be skipped.

You can keep the same kind of condition in both **OnActionExecuting** as well as **OnActionExecuted**, or in any other filter.

So basically our goal is to **skip the filter execution** for a particular action method which you want in specific.

Filter Overrides

```
[TypeFilter(typeof(FilterClassName))] //filter applied at controller level
public class ControllerName : Controller
{
    public IActionResult Action1() //requirement: The filter SHOULD execute
    {
    }

    public IActionResult Action2() //requirement: The filter SHOULD NOT execute
    {
    }
}
```

t.

on.

In we have to **return from this method**, so the remaining

After you can use the same functionality.



Filter Overrides

Attribute to be applied to desired action method

```
public class SkipFilterAttribute : Attribute, IFilterMetadata
{
}
```

Filter that respects 'SkipFilterAttribute'

```
public class FilterClassName : IActionFilter //or any other filter interface
{
    public void OnActionExecuting(ActionExecutingContext context)
    {
        //get list of filters applied to the current working action method
        if (context.Filters.OfType<SkipResultFilter>().Any())
        {
            return;
        }
        //TO DO: before logic here
    }

    public void OnActionExecuted(ActionExecutedContext context)
    {
        //TO DO: after logic here
    }
}
```

Let's say for example, we have created a 'PersonAlwaysRunResultFilter'.

Action method

[SkipFilter]

```
public IActionResult ActionMethod()  
{  
}
```

It skips execution of code of a filter, for specific action methods.

The screenshot shows the Visual Studio IDE with the file 'PersonAlwaysRunResultFilter.cs' open. The code defines a class 'PersonAlwaysRunResultFilter' that implements the interface 'IAlwaysRunResultFilter'. It contains two methods: 'OnResultExecuted' and 'OnResultExecuting'. The 'OnResultExecuted' method has a body with a single brace {}, while the 'OnResultExecuting' method is currently empty. The code is color-coded with syntax highlighting for keywords and comments.

```
public class PersonAlwaysRunResultFilter : IAlwaysRunResultFilter
{
    public void OnResultExecuted(ResultExecutedContext context)
    {
    }

    public void OnResultExecuting(ResultExecutingContext context)
    {
    }
}
```

And assume we have applied this filter at controller level. So instead of applying

The screenshot shows the Visual Studio IDE with the file 'PersonController.cs' open. The code defines an 'Edit' action method. At the top of the method, there is an annotation '[TypeFilter(typeof(PersonAlwaysRunResultFilter))]' with a tooltip 'TypeFilter(typeof(PersonAlwaysRunResultFilter))'. The rest of the code for the 'Edit' method is present below it.

```
[TypeFilter(typeof(PersonAlwaysRunResultFilter))]
public async Task<IActionResult> Edit(PersonRequest personRequest)
{
    PersonResponse? personResponse = await
        _personsService.GetPersonByPersonID(personRequest.ID);
    if (personResponse == null)
        return NotFound();
    var result = await _personsService.UpdatePersonAsync(
        personRequest.ID, personRequest);
    return Ok(result);
}
```

We have applied this filter here at controller level. That means all the action m

The screenshot shows the Visual Studio IDE with the file 'PersonController.cs' open. The code defines an 'Edit' action method. It includes two annotations: '[TypeFilter(typeof(HandleExceptionFilter))]' and '[TypeFilter(typeof(PersonAlwaysRunResultFilter))]' with a tooltip 'TypeFilter(typeof(HandleExceptionFilter))'. The rest of the code for the 'Edit' method is present below the annotations.

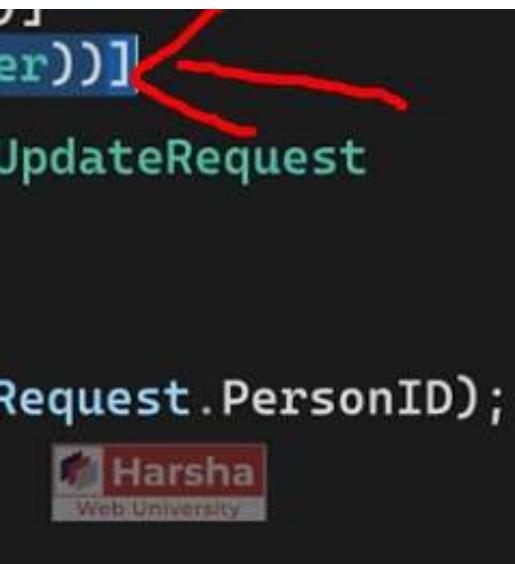
```
[TypeFilter(typeof(HandleExceptionFilter))]
[TypeFilter(typeof(PersonAlwaysRunResultFilter))]
public async Task<IActionResult> Edit(PersonRequest personRequest)
{
    PersonResponse? personResponse = await
        _personsService.GetPersonByPersonID(personRequest.ID);
    if (personResponse == null)
        return NotFound();
    var result = await _personsService.UpdatePersonAsync(
        personRequest.ID, personRequest);
    return Ok(result);
}
```



```
    context)
}

context)
```

g this filter here,

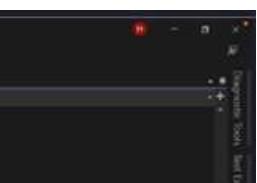


```
er))]
```

UpdateRequest

```
Request.PersonID);
```

method of this controller have this filter





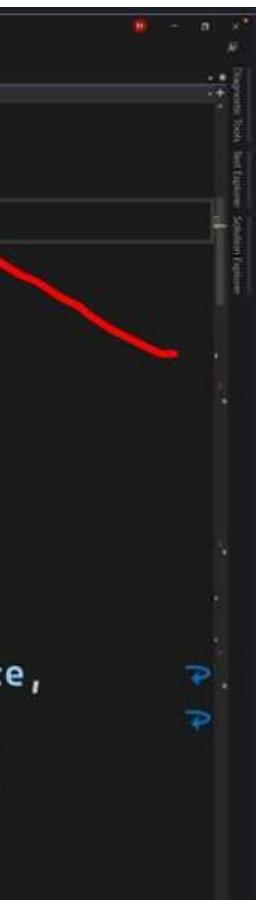
```
File Edit View Git Project Build Debug Test Analyze Tools Extensions Window Help Search (Ctrl+Q) CRUDExample
  Debug Any CPU CRUDExample PersonController.cs
  PersonController.cs
  PersonController.cs
  CRUDExample.Controllers.PersonController
  _personsService
Controller", 3 }, Order = 3)]
17 [TypeFilter(typeof(HandleExceptionFilter))]
18 [TypeFilter(typeof(PersonAlwaysRunResultFilter))]
12 references
19 public class PersonsController : Controller
20 {
21     //private fields
22     private readonly IPersonsService _personsService;
23     private readonly ICountriesService _countriesService;
24     private readonly ILogger<PersonsController> _logger;
25
26     //constructor
3 references
27     public PersonsController(IPersonsService personsService,
28         ICountriesService countriesService,
29         ILogger<PersonsController> logger)
30     {
31         _personsService = personsService;
32         _countriesService = countriesService;
33     }
34 }
```

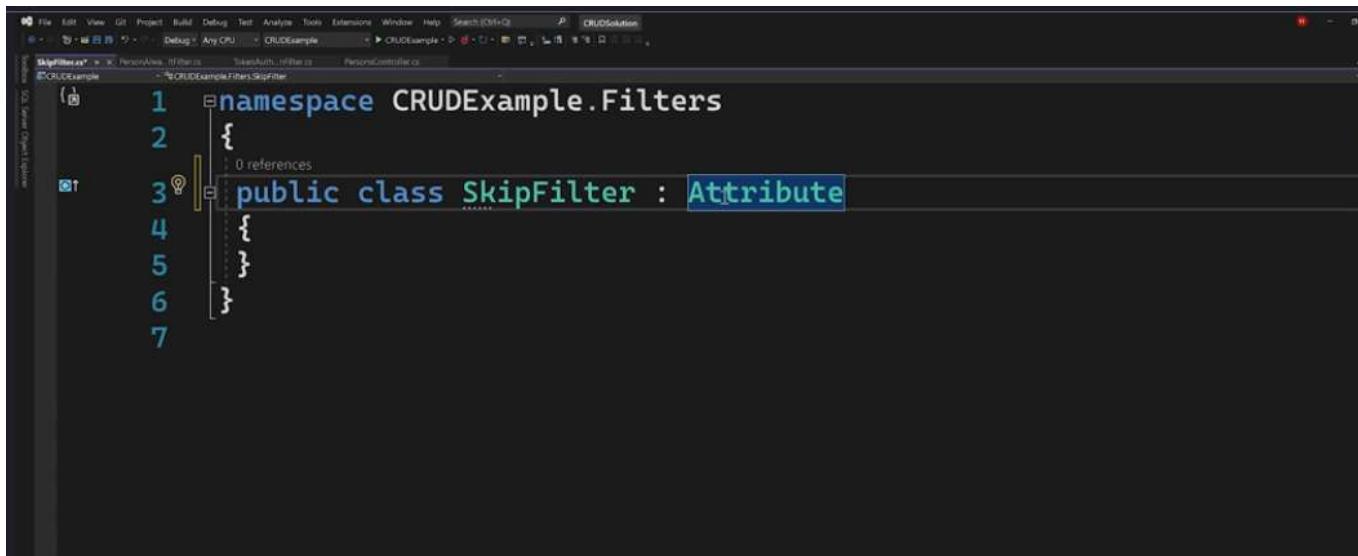
But here my goal is that, I don't want to execute that filter on 'Index' action method.



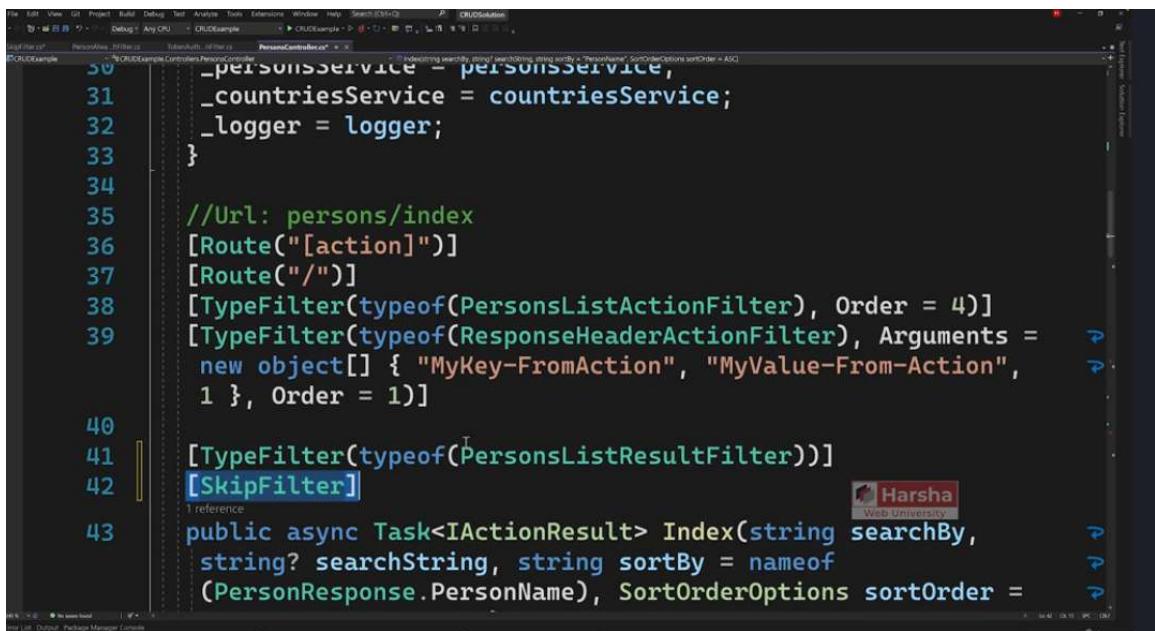
```
new object[] { "MyKey-FromAction", "MyValue-From-Action",
1 }, Order = 1)]
39
40 [TypeFilter(typeof(PersonsListResultFilter))]
1 reference
41 public async Task<IActionResult> Index(string searchBy,
42     string? searchString, string sortBy = nameof
43     (PersonResponse.PersonName), SortOrderOptions sortOrder =
44     SortOrderOptions.ASC)
45 {
46     _logger.LogInformation("Index action method of
47     PersonsController");
48
49     _logger.LogDebug($"searchBy: {searchBy}, searchString:
50     {searchString}, sortBy: {sortBy}, sortOrder: {sortOrder}");
51 }
```

Whenever we inherit a pre-defined class called 'System.Attribute', this class can be applied as an attribute for the controller or action method.





```
namespace CRUDExample.Filters
{
    [AttributeUsage(AttributeTargets.Class | AttributeTargets.Method)]
    public class SkipFilter : Attribute
    {
    }
}
```



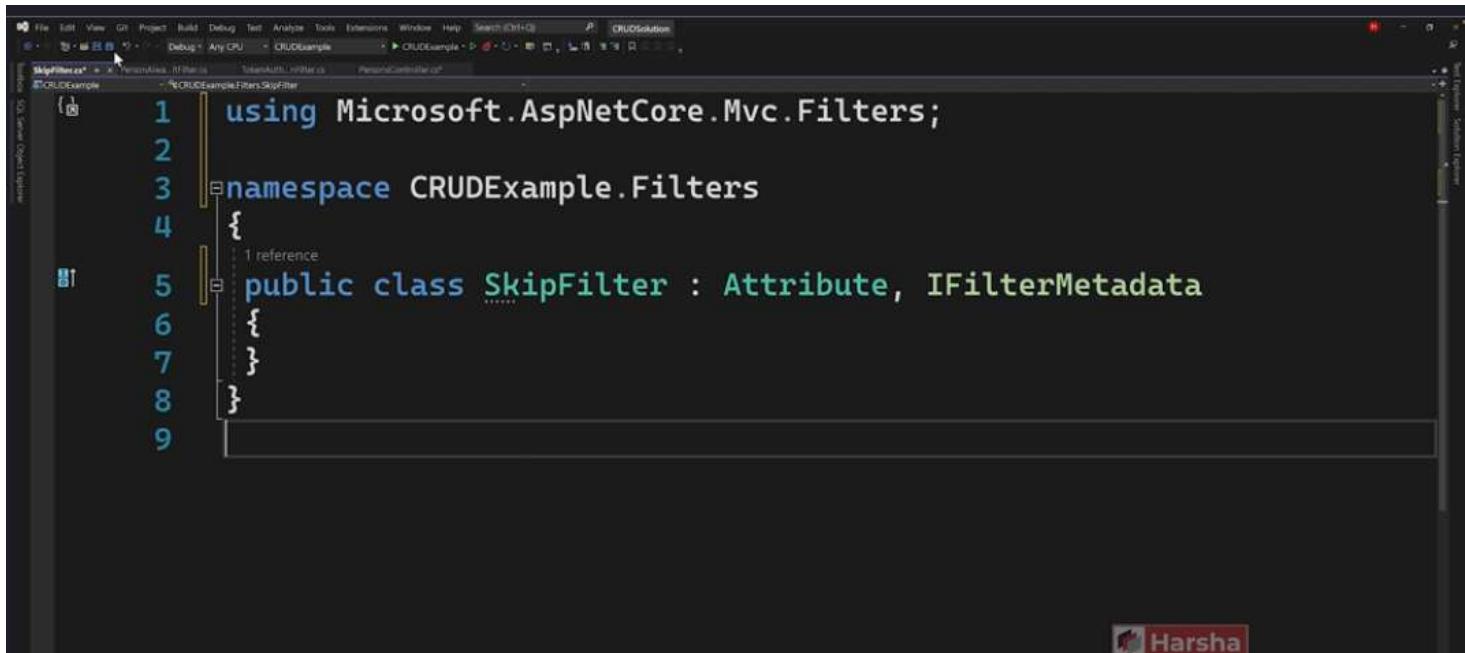
```
private readonly IPersonService _personService;
private readonly ICountriesService _countriesService;
private readonly ILogger<PersonController> _logger;
}

//Url: persons/index
[Route("[action]")]
[Route("/")]
[TypeFilter(typeof(PersonsListActionFilter), Order = 4)]
[TypeFilter(typeof(ResponseHeaderActionFilter), Arguments =
    new object[] { "MyKey-FromAction", "MyValue-From-Action",
    1 }, Order = 1)]
[TypeFilter(typeof(PersonsListResultFilter))]
[SkipFilter]
public async Task<IActionResult> Index(string searchBy,
    string? searchString, string sortBy = nameof(
        PersonResponse.PersonName), SortOrderOptions sortOrder =
```

And moreover, optionally,
Implement another interface called **IFilterMetadata**.

The **IFilterMetadata** interface represents that this class can act as a filter.
So when you are trying to read the list of filters that are applied for the **Index** action method,

You can get this **Skip Filter** as a part of that.



The screenshot shows the Visual Studio IDE with the code editor open. The file is named `SkipFilter.cs` and contains the following C# code:

```
1 using Microsoft.AspNetCore.Mvc.Filters;
2
3 namespace CRUDExample.Filters
4 {
5     public class SkipFilter : Attribute, IFilterMetadata
6     {
7     }
8 }
9
```

Harsha

Let us see the same.

Go to the **PersonAlwaysRunResultFilter**.

Assume that we have some functionality, and this logic should be skipped for the **Index** action method.
In order to skip it, we have to check the **context** parameter.

Get the list of filters that are applied for the current working action method.

Assume that we made a request to the **Index** action method.

So, the current working action method is **Index**.

The **Filters** property provides the list of filters applied for the **Index** action method, whether at the controller level or globally.
In that list, we are searching for a particular type of filter, which is **SkipFilter**.

The LINQ method **Any()** returns **true** if the **SkipFilter** is applied to the **Index** action method. Otherwise, it returns **false**.

So, if the **SkipFilter** is applied, we return from the method, skipping the remaining portion of the code.

Okay, return—that's it! Since it is **void**.

Now, let me demonstrate this practically, then you will understand the code better.


```
public class PersonAlwaysRunResultFilter : IAlwaysRunResultFilter
{
    public void OnResultExecuted(ResultExecutedContext context)
    {
    }

    public void OnResultExecuting(ResultExecutingContext context)
    {
        if (context.Filters.OfType<SkipFilter>().Any())
        {
            return;
        }

        //TO DO: before logic here
    }
}
```

Now let's run our application.

```
(PersonResponse.PersonName), sortOrderOptions sortorder = SortOrderOptions.ASC)

_logger.LogInformation("Index action method of PersonsController");

_logger.LogDebug($"searchBy: {searchBy}, searchString: {searchString}, sortBy: {sortBy}, sortOrder: {sortOrder}");

//Search
List<PersonResponse> persons = await _personsService.GetFilteredPersons(searchBy, searchString);

//Sort
List<PersonResponse> sortedPersons = await
```

we are going to return the view result

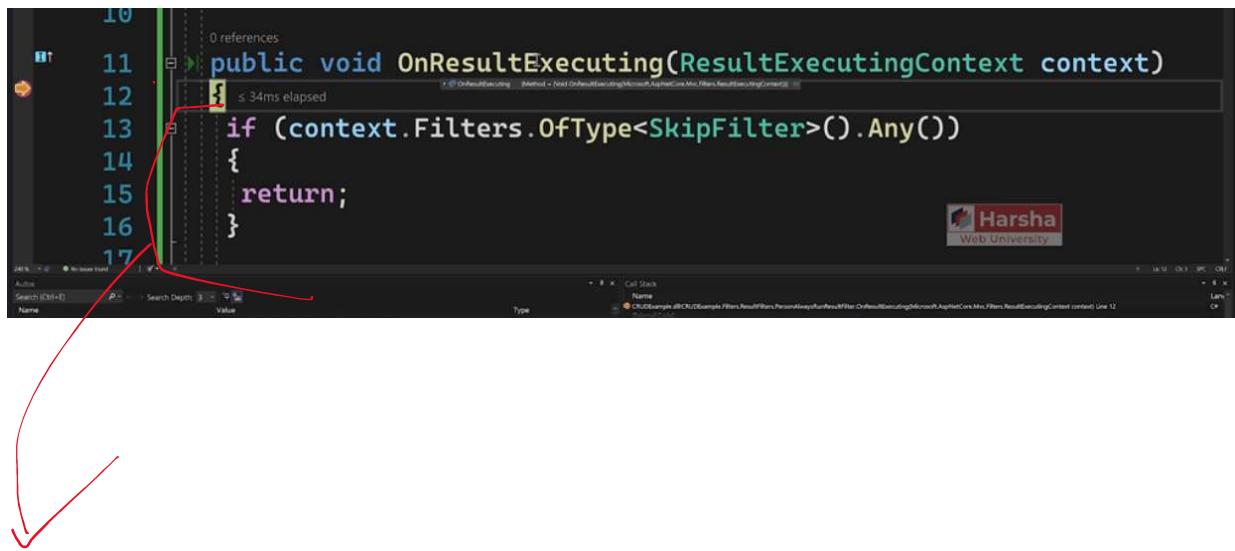
Before executing this view


```
51     List<PersonResponse> persons = await
52         _personsService.GetFilteredPersons(searchBy, search
53             //Sort
54     List<PersonResponse> sortedPersons = await
55         _personsService.GetSortedPersons(persons, sortBy,
56     return View(sortedPersons); //Views/Persons/Index.cs
57 }
58
59
60 // Executes when the user clicks on "Create Person" by
```

It runs 'onActionExecuted' method.

```
11     public PersonsListActionFilter	ILogger<PersonsListActionFilter>
12         logger)
13     {
14         _logger = logger;
15     }
16
17     public void OnActionExecuted(ActionExecutedContext context)
18     {
19         //To do: add after logic here
20         _logger.LogInformation("{FilterName}. {MethodName} method",
21             nameof(PersonsListActionFilter), nameof(OnActionExecuted));
22
23         PersonsController personsController = (PersonsController)
24             context.Controller;
```

```
5     public class PersonAlwaysRunResultFilter :
6         IAlwaysRunResultFilter
7     {
8     }
9
10
11    public void OnResultExecuting(ResultExecutingContext context)
12    {
13    }
```

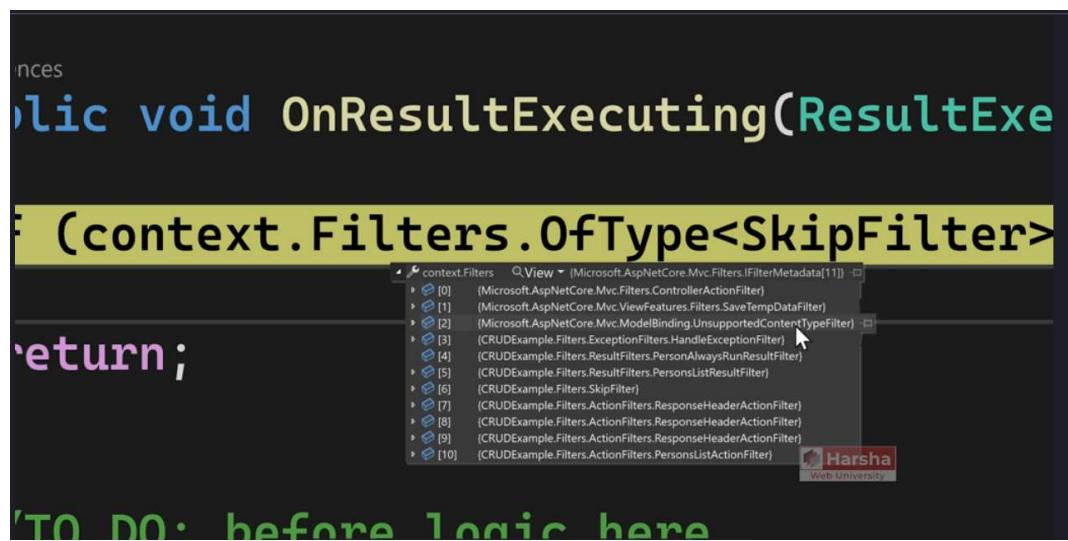



```
10
11 public void OnResultExecuting(ResultExecutingContext context)
12 {
13     if (context.Filters.OfType<SkipFilter>().Any())
14     {
15         return;
16     }
17 }
```

If this 'if statement' logic is not present here, actual before logic will be executed.

If the 'skip filter' is applied for the current working action method.

Some of the filters are automatically applied from 'ASP.NET Core'. We get all filters. (Controller level, Global level, method level)



```
10
11 public void OnResultExecuting(ResultExe
12
13     if (context.Filters.OfType<SkipFilter>
14
15         return;
16
17 
```

'TO DO: before logic here'

Call stack details:

- [0] Microsoft.AspNetCore.Mvc.Filters.ControllerActionFilter
- [1] Microsoft.AspNetCore.Mvc.ViewFeatures.Filters.SaveTempDataFilter
- [2] Microsoft.AspNetCore.Mvc.ModelBinding.UnsupportedContentTypeFilter
- [3] CRUDExample.Filters.ExceptionFilters.HandleExceptionFilter
- [4] CRUDExample.Filters.ResultFilters.PersonAlwaysRunResultFilter
- [5] CRUDExample.Filters.ResultFilters.PersonsListResultFilter
- [6] CRUDExample.Filters.SkipFilter
- [7] CRUDExample.Filters.ActionFilters.ResponseHeaderActionFilter
- [8] CRUDExample.Filters.ActionFilters.ResponseHeaderActionFilter
- [9] CRUDExample.Filters.ActionFilters.ResponseHeaderActionFilter
- [10] CRUDExample.Filters.ActionFilters.PersonsListActionFilter

Asp.Net Core

Service Filter

In **ASP.NET Core**, there are two attributes to apply a filter to a particular action method or controller:

- 1 TypeFilter**
- 2 ServiceFilter**

So far, we have used **TypeFilter**, but there is another one called **ServiceFilter**.

What's the difference between them?

Both are used for the **same purpose**, and their syntax is also **the same**, but there is **one small difference**:

- TypeFilter** → Allows you to supply arguments to the filter **constructor**.
- ServiceFilter** → Does not allow you to supply arguments to the filter **constructor**.

That's the main distinction!

ServiceFilterAttribute [vs] TypeFilterAttribute

Common purpose

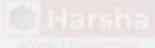
Both are used to apply a filter a controller or action method.

Type Filter
Attribute

```
//can supply arguments to filter
[TypeFilter(typeof(FilterClassName)),
 Arguments = new object[] { arg1, arg2 })]
public IActionResult ActionMethod()
{
    ...
}
```

Service Filter
Attribute

```
//can't supply arguments to filter
[ServiceFilter(typeof(FilterClassName))]
public IActionResult ActionMethod()
{
    ...
}
```



If you look at our project. In this filter there are no parameters to be supplied

| Harsha

The screenshot shows the Visual Studio IDE with the file 'PersonsListResultFilter.cs' open. The code defines a class 'PersonsListResultFilter' that implements the interface 'IAsyncResultFilter'. It uses dependency injection to get an 'ILogger<PersonsListResultFilter>' named 'logger'. The 'OnResultExecutionAsync' method is annotated with the 'ServiceFilterAttribute'.

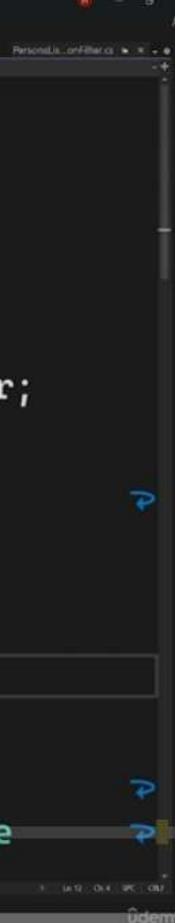
```
1  using Microsoft.AspNetCore.Mvc.Filters;
2
3  namespace CRUDEExample.Filters.ResultFilters
4  {
5      public class PersonsListResultFilter : IAsyncResultFilter
6      {
7          private readonly ILogger<PersonsListResultFilter> _logger;
8
9          public PersonsListResultFilter(ILogger<PersonsListResultFilter> logger)
10         {
11             _logger = logger;
12         }
13
14         public async Task OnResultExecutionAsync(
15             ResultExecutingContext context, ResultExecutionDelegate next)
```



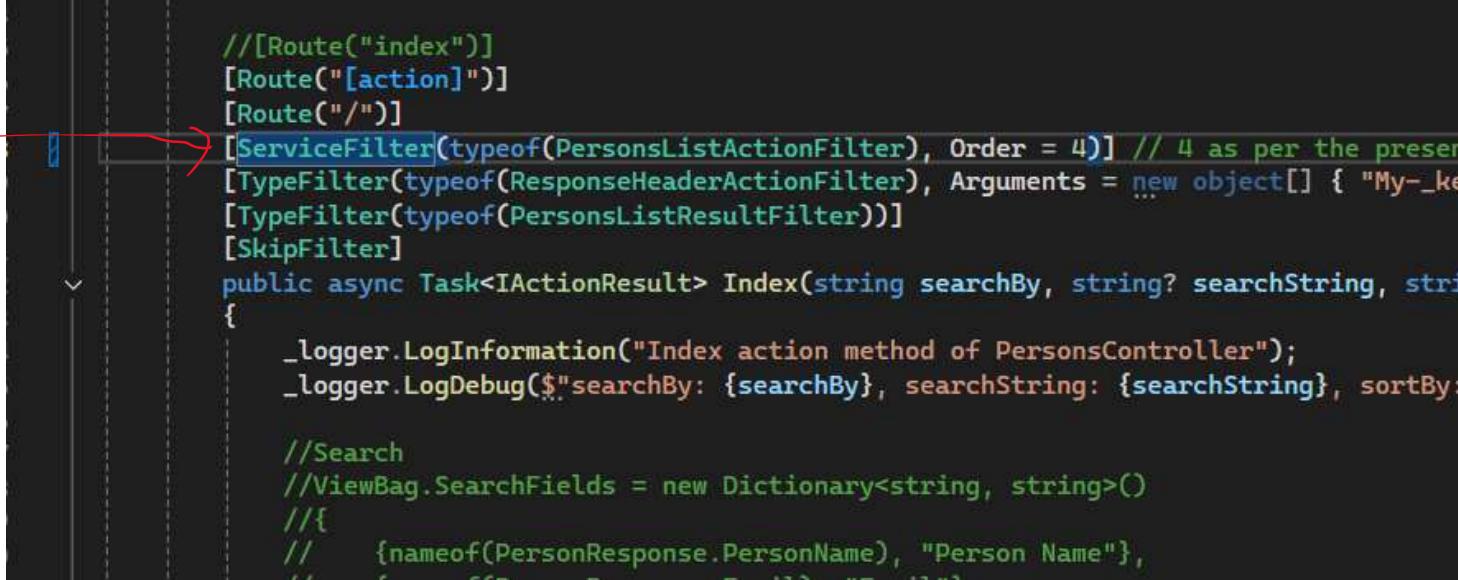
The screenshot shows the 'PersonsController.cs' code. A red arrow points to the section where filters are applied to the 'Index' action. The filters include 'PersonsListActionFilter' (order 4), 'ResponseHeaderActionFilter' (arguments: "My_key-From-Action", "My-Value-From-Action"), and 'PersonsListResultFilter'. The 'Index' action itself logs information about the search parameters.

```
//[Route("index")]
[Route("[action]")]
[Route("/")]
[TypeFilter(typeof(PersonsListActionFilter), Order = 4)] // 4 as per the presentation
[TypeFilter(typeof(ResponseHeaderActionFilter), Arguments = new object[] { "My_key-From-Action", "My-Value-From-Action" })
[TypeFilter(typeof(PersonsListResultFilter))]
[SkipFilter]
public async Task<IActionResult> Index(string searchBy, string? searchString, string sortBy = nameof(PersonResponse.PersonName))
{
    _logger.LogInformation("Index action method of PersonsController");
    _logger.LogDebug($"searchBy: {searchBy}, searchString: {searchString}, sortBy: {sortBy}, sortOrder: {sortOrder}");
}

//Search
//ViewBag.SearchFields = new Dictionary<string, string>()
//{
//    {nameof(PersonResponse.PersonName), "Person Name"},
//    {nameof(PersonResponse.Email), "Email"},
//    {nameof(PersonResponse.DateOfBirth), "Date of Birth"},
//    {nameof(PersonResponse.Gender), "Gender"},
//    {nameof(PersonResponse.CountryID), "Country"},
//    {nameof(PersonResponse.Address), "Address"},
//};
```



```
, 1 }, Order = 1)]  
  
onName), SortOrderOptions sortOrder = Sor
```

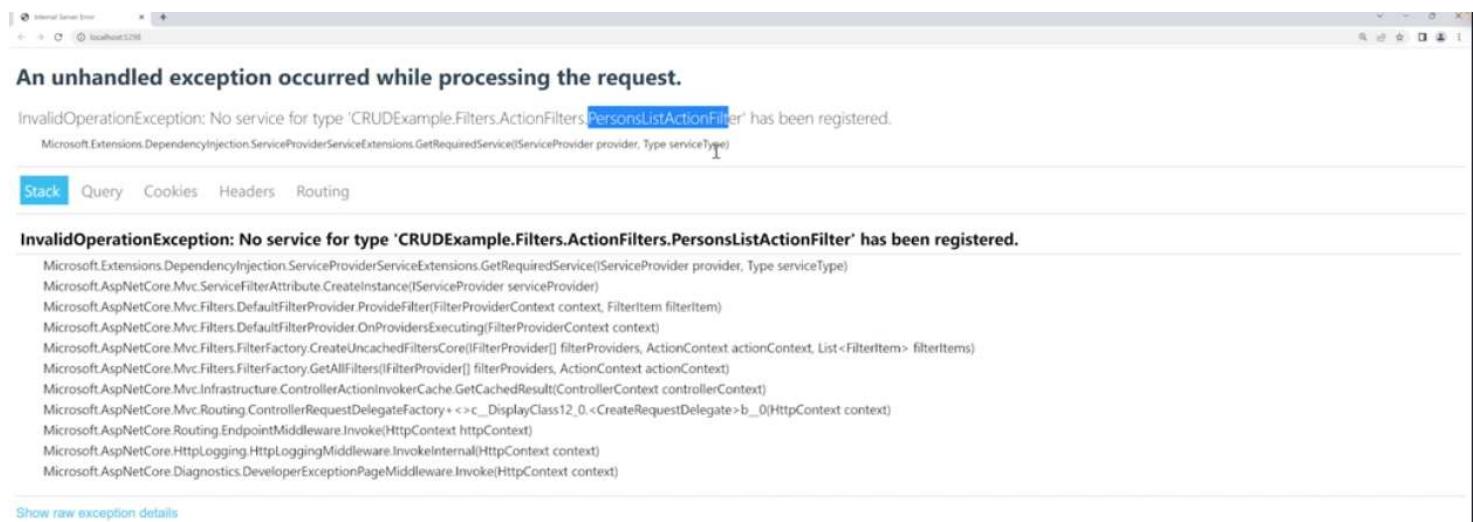



```
//[Route("index")]
[Route("[action]")]
[Route("/")]
[ServiceFilter(typeof(PersonsListActionFilter), Order = 4)] // 4 as per the present requirement
[TypeFilter(typeof(ResponseHeaderActionFilter)), Arguments = new object[] { "My-kebab" }]
[TypeFilter(typeof(PersonsListResultFilter))]
[SkipFilter]
public async Task<IActionResult> Index(string searchBy, string? searchString, string? sortBy)
{
    _logger.LogInformation("Index action method of PersonsController");
    _logger.LogDebug($"searchBy: {searchBy}, searchString: {searchString}, sortBy: {sortBy}");

    //Search
    ViewBag.SearchFields = new Dictionary<string, string>()
    //{
    //    {nameof(PersonResponse.PersonName), "Person Name"},
    //    {nameof(PersonResponse.Surname), "Surname"}
    //};
}
```

It works in a same way.

Now run our application and wait, we have got an error.



Because in case of Service Filter, It assumes, expects and requires all the filters in the container. If not you will get such error.

```
ntation
ey-From-Action", "My-Value-From-Action",

ing sortBy = nameof(PersonResponse.Person
: {sortBy}, sortOrder: {sortOrder}");
```

ters added as a services in the IOC


```
32     _logger = logger;
33 }
34
35 //Url: persons/index
36 [Route("[action]")]
37 [Route("/")]
38 [ServiceFilter(typeof(PersonsListActionFilter), Order = 4)]
39 [TypeFilter(typeof(ResponseHeaderActionFilter), Arguments =
    new object[] { "MyKey-FromAction", "MyValue-From-Action",
    1 }, Order = 1)]
40
41 [TypeFilter(typeof(PersonsListResultFilter))]
42 [SkipFilter]
43 public async Task<IActionResult> Index(string searchBy,
    string? searchString, string sortBy = nameof
    (PersonResponse.PersonName), SortOrderOptions sortOrder =
    SortOrderOptions.ASC)
```

The point is, here the ServiceFilter Attribute tries to inject this particular list injection. It will not create a direct object of this class. But tries to inject the

```
33     builder.Services.AddScoped<ICountriesService, CountriesService>()
34     builder.Services.AddScoped<IPersonsService, PersonsService>();
35
36     builder.Services.AddDbContext<ApplicationDbContext>(options =>
37     {
38         options.UseSqlServer(builder.Configuration.GetConnectionString
            ("DefaultConnection"));
39     });
40
41     builder.Services.AddTransient<PersonsListActionFilter>();
42
43     builder.Services.AddHttpLogging(options =>
44     {
45         options.LoggingFields =
46             Microsoft.AspNetCore.HttpLogging.HttpLoggingFields.RequestProp
```



action filter through dependency
same through dependency injection.



Now run again.

Asp.Net Core

ServiceFilter [vs] TypeFilter

Type Filter

Can supply arguments to the filter.

Filter instances are created by using
Microsoft.Extensions.DependencyInjection.
ObjectFactory.

They're NOT created using DI (Dependency Injection).

The lifetime of filter instances is by default transient (a
new filter instance gets created every time when it is
invoked).

But optionally, you can re-use the same instance of
filter class across multiple requests, by setting a
property called TypeFilterAttribute.IsReusable to 'true'.

Service Filter

Can't supply arguments to the filter.

Filter instances are created by using
ServiceProvider (using DI).

The lifetime of filter instances is the a
of the filter class added in the IoC con

Eg:

If the filter class is added to the IoC c
AddScoped() method, then its instanc

| Harsha

actual lifetime
container.

container with
es are scoped.

ServiceFilter [vs] TypeFilter

Type Filter

Filter classes **NEED NOT** be registered (added) to the IoC container.

Filter classes **CAN** inject services using both constructor injection or method injection.

Service Filter

Filter class **SHOULD** be registered (a) container, much like any other service.

Filter classes **CAN** inject services using both constructor injection or method injection.

There is no particular benefit in either of these.

Asp.Net Core Filter Attribute Classes

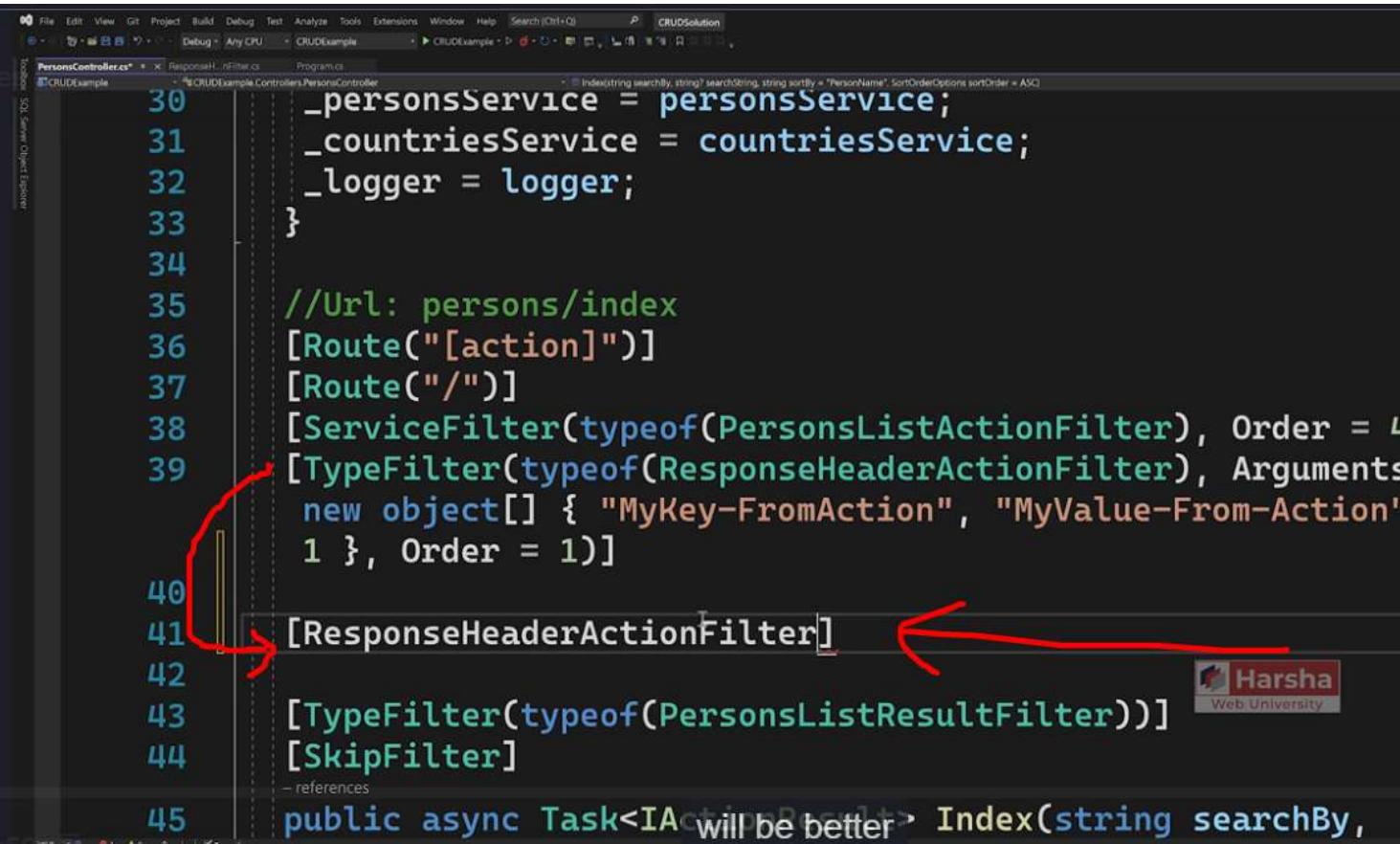
In almost all the lectures of the section, we are applying a filter either as a ty

added) to the IoC
ce.

using both
ction.

ype filter or a service filter, right? But if

somehow we can apply the same filter in a more concise and shortcut way—particular type filter, I would like to apply my filter like this: I would like to do this. Currently, it is not possible, but if somehow I can make it possible, that



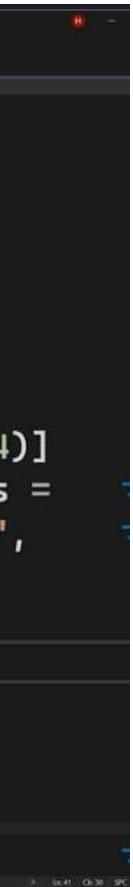
```
File Edit View Git Project Build Debug Test Analyze Tools Extensions Window Help Search (Ctrl + Q) CRUDsolution
File Edit View Git Project Build Debug Test Analyze Tools Extensions Window Help Search (Ctrl + Q) CRUDsolution
PersonController.cs  ResponseHeaderFilter.cs Programs.cs
CRUDExample CRUDExample.Controllers.PersonController
30     _personsService = personsService;
31     _countriesService = countriesService;
32     _logger = logger;
33 }
34
35 //Url: persons/index
36 [Route("[action]")]
37 [Route("/")]
38 [ServiceFilter(typeof(PersonsListActionFilter), Order = 1)]
39 [TypeFilter(typeof(ResponseHeaderActionFilter), Arguments =
40     new object[] { "MyKey-FromAction", "MyValue-From-Action" },
41     Order = 1)]
42 [ResponseHeaderActionFilter] ←
43 [TypeFilter(typeof(PersonsListResultFilter))]
44 [SkipFilter]
45 public async Task<IA<will be better> Index(string searchBy,
```

Harsha
Web University

Because in this case, if I would like to supply the arguments easily, I can simply do this is my goal. But how do you transform this particular statement into this? Is this possible? That's the topic of today's lecture.

In order to use your filter as an attribute, you have to change the way you create

-for example, instead of writing this
use my filter as an attribute directly like
would be better.



A screenshot of a terminal window or code editor showing a snippet of Java code. The code includes a constructor call with parameters and a return statement. The cursor is positioned at the end of the return statement, and the IDE is displaying code completion suggestions in a dropdown menu below the cursor. The suggestions include various methods and fields from the current class and its superclass.

```
    @]  
    s =  
    ,
```

simply provide the same values like this. So,
? How do you make attribute filters

Create your filter class.


```
30     _personsService = personsService;
31     _countriesService = countriesService;
32     _logger = logger;
33 }
34
35 //Url: persons/index
36 [Route("[action]")]
37 [Route("/")]
38 [ServiceFilter(typeof(PersonsListActionFilter), Order = 4)]
39 [TypeFilter(typeof(ResponseHeaderActionFilter), Arguments =
    new object[] { "MyKey-FromAction", "MyValue-From-Action",
    1 }, Order = 1)]
40
41 [ResponseHeaderActionFilter("MyKey-FromAction", "MyValue-From-
    Action", 1)]
42
43 [TypeFilter(typeof(PersonsListResultFilter))]
44 [SkipFilter]
```

Traditionally, from the beginning, we have been creating our filter class by implementing the filter interface. For example, we implement `IActionFilter`, `IExceptionFilter`, and other filters. Similarly, for asynchronous filters, you implement `IAsyncActionFilter` and `IAsyncExceptionFilter`.

However, instead of implementing a filter interface, you can inherit from filter attribute classes, such as `ActionFilterAttribute` or `ResultFilterAttribute`.

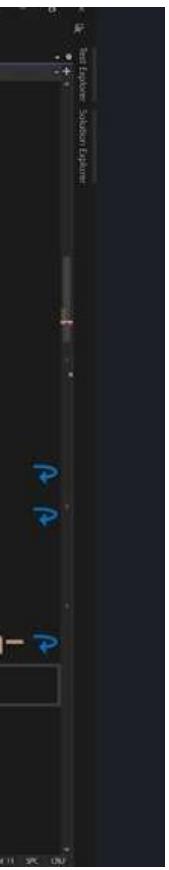
The key difference between these two approaches is that `ActionFilterAttribute` does not support constructor injection. Instead, if you want to inject services through constructor injection, you can prefer using a filter attribute.

Here, I am referring to all three classes as **filter attribute classes**, meaning they can be `ActionFilterAttribute`, `ExceptionFilterAttribute`, or `ResultFilterAttribute`.

Notice that there is no `AuthorizationFilterAttribute` or `ResourceFilterAttribute`. Microsoft has not provided these, probably because they are not commonly used in practical scenarios.

The most commonly implemented custom filters are **action filters, exception filters, and result filters**. For authorization filters, the built-in ones usually handle even complex cases well—something we will explore in the next section. Implementing a resource filter is rare.

With that said, our focus here is on understanding the difference between implementing filter interfaces like `IActionFilterAttribute` and filter attribute classes like `ActionFilterAttribute`.



you can implement their respective

erAttribute, ExceptionFilterAttribute,

on. So, if you do not need to inject

ceptionFilterAttribute, or

possibly because they are not frequently

the **Identity** section later. Similarly,

onFilter and inheriting from

IActionFilter [vs] ActionFilterAttribute

[versus]

Action filter that implements 'IActionFilter'

```
public class FilterClassName : IActionFilter,  
    IOrderedFilter  
{  
    //supports constructor DI  
}
```

Action filter that inherits from 'ActionFilterAttribute'

```
public class FilterClassName : ActionFilterAttribute  
{  
    //doesn't support constructor DI  
}
```

Filter interfaces:

IAuthorizationFilter

IAsyncAuthorizationFilter

IResourceFilter

IAsyncResourceFilter

IActionFilter

IAsyncActionFilter

IExceptionFilter

IAsyncExceptionFilter

IResultFilter

IAsyncResultFilter

Filter attributes:

ActionFilterAttribute

ExceptionFilterAttribute

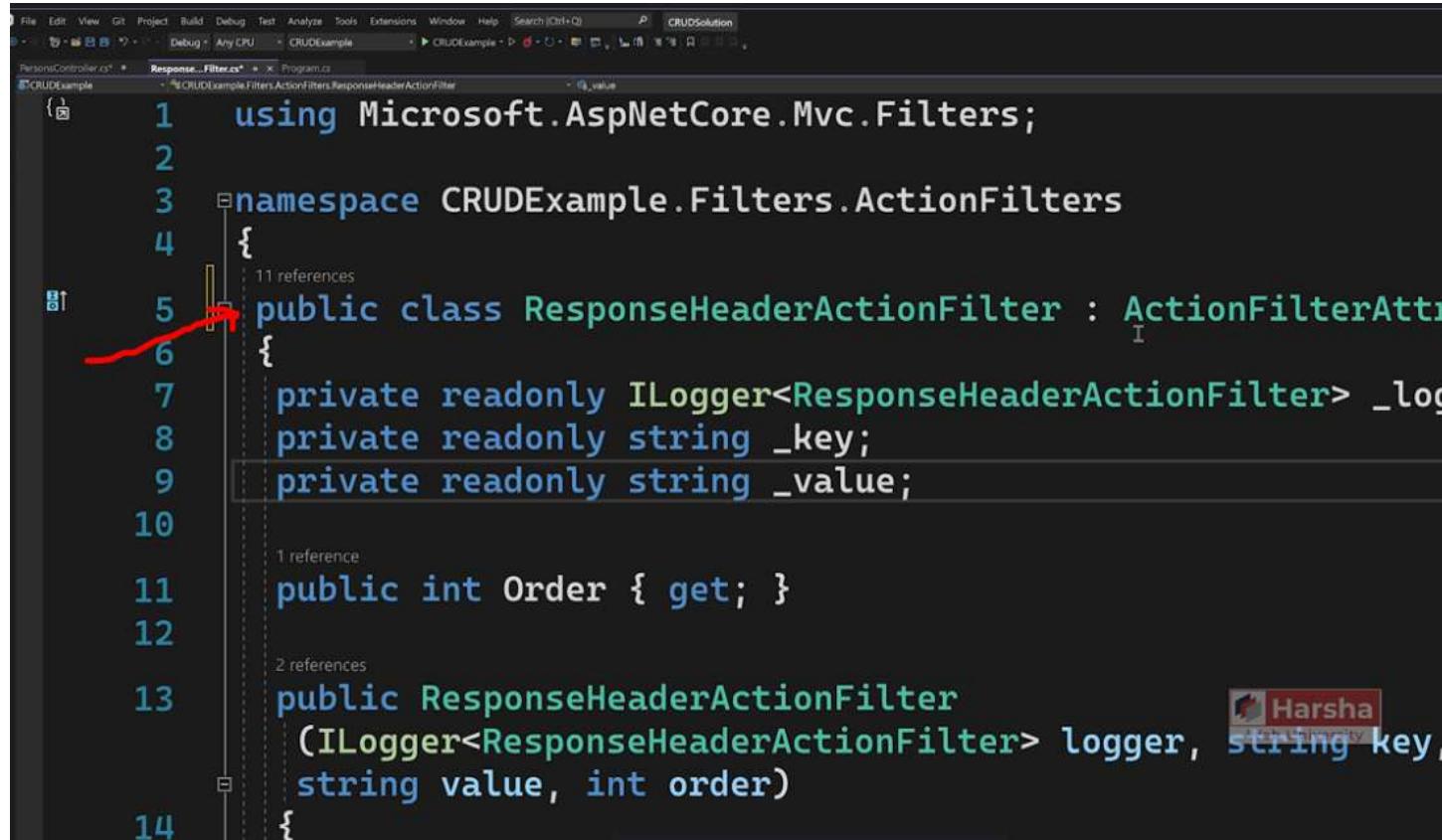
ResultFilterAttribute

'ActionFilterAttribute'

Name : ActionFilterAttribute

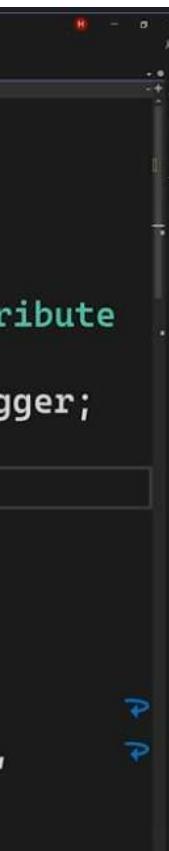
structor DI


```
1  using Microsoft.AspNetCore.Mvc.Filters;
2
3  namespace CRUDExample.Filters.ActionFilters
4  {
5      public class ResponseHeaderActionFilter : IAsyncActionFilter, IOrderedFilter
6      {
7          private ILogger<ResponseHeaderActionFilter> _logger;
8          private readonly string _key;
9          private readonly string Value;
10
11         public int Order { get; set; }
12
13         public ResponseHeaderActionFilter(ILogger<ResponseHeaderActionFilter> logger, string key, string value, int order)
14         {
15             this._logger = logger;
16             this._key = key;
17             this.Value = value;
18             Order = order;
19         }
20
21         public async Task OnActionExecutionAsync(ActionExecutingContext context, ActionExecutionDelegate next)
22         {
23             _logger.LogInformation("{FilterName}.{{MethodName}} method - before", nameof(ResponseHeaderActionFilter), nameof(OnActionExecution));
24
25             await next(); // call the next subsequent filter which is added in the chain. If there is no subsequent filter, it will call the controller action
26
27             _logger.LogInformation("{FilterName}.{{MethodName}} method - after", nameof(ResponseHeaderActionFilter), nameof(OnActionExecution));
28             context.HttpContext.Response.Headers[_key] = Value;
29         }
30     }
31 }
```



```
1  using Microsoft.AspNetCore.Mvc.Filters;
2
3  namespace CRUDExample.Filters.ActionFilters
4  {
5      public class ResponseHeaderActionFilter : ActionFilterAttribute
6      {
7          private readonly ILogger<ResponseHeaderActionFilter> _logger;
8          private readonly string _key;
9          private readonly string _value;
10
11         public int Order { get; } = 1;
12
13         public ResponseHeaderActionFilter(ILogger<ResponseHeaderActionFilter> logger, string key, string value)
14         {
15             _logger = logger;
16             _key = key;
17             _value = value;
18         }
19
20         public void OnActionExecuting(ActionExecutingContext context)
21         {
22             _logger.LogInformation("{FilterName}.{{MethodName}} method - before", nameof(ResponseHeaderActionFilter), nameof(OnActionExecuting));
23
24             context.HttpContext.Response.Headers[_key] = _value;
25
26             _logger.LogInformation("{FilterName}.{{MethodName}} method - after", nameof(ResponseHeaderActionFilter), nameof(OnActionExecuting));
27         }
28
29         public void OnActionExecuted(ActionExecutedContext context)
30         {
31             _logger.LogInformation("{FilterName}.{{MethodName}} method - after", nameof(ResponseHeaderActionFilter), nameof(OnActionExecuted));
32         }
33     }
34 }
```

```
ecutionAsync));  
all the action method.  
ecutionAsync));
```



IActionFilter [vs] ActionFilterAttribute

Action filter that implements 'IActionFilter'

```
public class FilterClassName : IActionFilter,  
                           IOrderedFilter  
{  
    public int Order { get; set; }  
  
    public FilterClassName(IService service, type arg)  
    {  
    }  
  
    public void  
    OnActionExecuting(ActionExecutingContext context)  
    {  
    }  
  
    public void  
    OnActionExecuted(ActionExecutedContext context)  
    {  
    }  
}
```

```
[TypeFilter(typeof(FilterClassName),  
           Arguments = new object[] { arg1, ... })]
```

Action filter that inherits 'ActionFilterAttribute'

```
public class FilterClassName : ActionFilterAttribute  
{  
    public FilterClassName(type arg)  
    {  
    }  
  
    public override void  
    OnActionExecuting(ActionExecutingContext context)  
    {  
    }  
  
    public override void  
    OnActionExecuted(ActionExecutedContext context)  
    {  
    }  
}
```

```
[FilterClassName(arg1, ... )]
```

rAttribute'

nFilterAttribute

Context context)

ontext context)

ia

The screenshot shows the Visual Studio IDE with the code editor open to the `ResponseHeaderActionFilter.cs` file. The file is located in the `CRUDEExample.Filters.ActionFilters` namespace. The code defines a class `ResponseHeaderActionFilter` that implements the `ActionFilterAttribute`. It includes fields for `ILogger<ResponseHeaderActionFilter>`, `_key`, and `Value`. The constructor takes `logger`, `key`, and `value` as parameters. The `OnActionExecutionAsync` method logs information before and after the action execution, and sets the response header `Value`.

```
1  using Microsoft.AspNetCore.Mvc.Filters;
2
3  namespace CRUDEExample.Filters.ActionFilters
4  {
5      public class ResponseHeaderActionFilter : ActionFilterAttribute
6      {
7          private ILogger<ResponseHeaderActionFilter> _logger;
8          private readonly string _key;
9          private readonly string Value;
10
11         //public int Order { get; set; }
12
13         //you can't inject constructor injection here
14         //public ResponseHeaderActionFilter(ILogger<ResponseHeaderActionFilter> logger, string key, string value)
15         public ResponseHeaderActionFilter(string key, string value)
16         {
17             //this._logger = logger;
18             this._key = key;
19             this.Value = value;
20             //Order = order;
21         }
22
23         public override async Task OnActionExecutionAsync(ActionExecutingContext context, ActionExecutionDelegate next)
24         {
25             //_Logger.LogInformation("{FilterName}.{MethodName} method - before", nameof(ResponseHeaderActionFilter), nameof(OnActionExecutionAsync));
26
27             await next(); // call the next subsequent filter which is added in the chain. If there is no subsequent filter, it will call the action method
28
29             //_Logger.LogInformation("{FilterName}.{MethodName} method - after", nameof(ResponseHeaderActionFilter), nameof(OnActionExecutionAsync));
30             context.HttpContext.Response.Headers[_key] = Value;
31         }
32     }
33 }
34
```

The screenshot shows the `PersonsController.cs` file in Visual Studio. A red box highlights two specific filter application points: one for the `Index` action and another for the `PersonList` result. The `Index` action is annotated with `[ServiceFilter(typeof(PersonsListActionFilter), Order = 4)]` and `[TypeFilter(typeof(ResponseHeaderActionFilter), Arguments = new object[] { "My_key-From-Action", "My-Value-From-Action" })]`. The `PersonList` result is annotated with `[ResponseHeaderActionFilter("My_key-From-Action", "My-Value-From-Action")]` and `[TypeFilter(typeof(PersonsListResultFilter))]`. The `Index` action also includes logging and search logic.

```
1  //Route("index")
2  [Route("[action]")]
3  [Route("/")]
4  [ServiceFilter(typeof(PersonsListActionFilter), Order = 4)] // 4 as per the presentation
5  //TypeFilter(typeof(ResponseHeaderActionFilter), Arguments = new object[] { "My_key-From-Action", "My-Value-From-Action" })
6  [ResponseHeaderActionFilter("My_key-From-Action", "My-Value-From-Action")]
7  [TypeFilter(typeof(PersonsListResultFilter))]
8  [SkipFilter]
9  public async Task<IActionResult> Index(string searchBy, string? searchString, string sortBy = nameof(PersonResponse.PersonName))
10 {
11     _logger.LogInformation("Index action method of PersonsController");
12     _logger.LogDebug($"searchBy: {searchBy}, searchString: {searchString}, sortBy: {sortBy}, sortOrder: {sortOrder}");
13
14     //Search
15     //ViewBag.SearchFields = new Dictionary<string, string>()
16     //{
17     //    {"nameof(PersonResponse.PersonName)", "Person_Name"}
18 }
```

ActionFilter...

hod.

, 1 }, Order = 1)]

lame), SortOrderOptions sortOrder = So

Filter interface [vs] FilterAttribute class

Filter interface [such as `IActionFilter`, `IResultFilter` etc.]

Filter class **MUST** implement all methods - both "Executing" and "Executed" methods.

Filter class **CAN** have DI with either constructor injection or method injection.

FilterAttribute class [such as `ActionFilterAttribute`]

Filter class **MAY** override desired (e.g. `OnActionExecuting`) methods - "Executing" and "Executed".

Filter class **CAN'T** have DI with neither constructor injection nor method injection.

Filter interface [vs] FilterAttribute class

Filter interface [such as `IActionFilter`, `IResultFilter` etc.]

Doesn't implement "Attribute" class.

Filter should be applied to controller or action methods by using `[ServiceFilter]` or `[TypeFilter]` attributes; otherwise can be applied as global filter in the `Program.cs`.

Eg: `[TypeFilter(typeof(FilterClassName))]` **//lengthy**

FilterAttribute class [such as `ActionFilterAttribute`]

Implements "Attribute" class by default.

Filter can be applied to controller or action methods directly using the filter class name in `[FilterName]` or using `[ServiceFilter]` or `[TypeFilter]` attributes; otherwise can be applied as global filter in the `Program.cs`.

Eg: `[FilterClassName]` **//simple**

To conclude, generally, as per the Microsoft documentation and most common practices, it is preferred to use **filter interfaces** over **filter attributes**. The biggest advantage of using filter interfaces is that we can use **constructor injection** directly in the filter classes with these interfaces.

| Harsha

FilterAttribute etc.]

either or both
ed") methods.

her constructor

ha

| Harsha

erAttribute etc.]

tion methods
tself (without
ibutes);
r in the

non practices, it's
age is that you can

This is important because you may want to call a service method to verify or at least be able to inject a logger. That's why the implementation of filter interface has been done so far.

If you want to have both benefits. Then prefer considering 'IFilterFactory'



Asp.Net Core _____ **IFilterFactory**

Here's the formatted version of the text:

As I mentioned earlier in the previous lecture, I would like to get both benefits. You want to apply the filter class as an attribute with a simple syntax, while at the same time be able to use **dependency injection** in the filter class.

To achieve both benefits, instead of using the **Action Filter Attribute**, you can migrate the same functionality using the **IFilterFactory** interface.

Let's see how.

or access data, or at
interfaces, like we've



efits. That is, we
the same time being

I should try to

IFilterFactory [vs] ActionFilterAttribute

Filter factory that inherits 'IFilterFactory'

```
public class FilterClassNameAttribute : Attribute,  
    IFilterFactory  
{  
    public type Prop1 { get; set; }  
  
    public FilterClassName(type arg1, type arg2)  
    {  
        this.Prop1 = arg1; this.Prop2 = arg2;  
    }  
  
    public IFilterMetadata CreateInstance(IServiceProvider  
serviceProvider)  
    {  
        FilterClassName filter =  
            serviceProvider.GetRequiredService  
            <FilterClassName>(); //instantiate the filter  
        filter.Property1 = Prop1;  
        ...  
        return filter;  
    }  
}
```

[FilterClassName(arg1, arg2, ...)]

Action filter that inherits 'ActionFilter'

```
public class FilterClassName : ActionFilter  
{  
    public FilterClassName(type arg1, type arg2)  
    {  
    }  
  
    public override void  
    OnActionExecuting(ActionExecutingContext context)  
    {  
    }  
  
    public override void  
    OnActionExecuted(ActionExecutedContext context)  
    {  
    }  
}
```

[FilterClassName(arg1, arg2, ...)]

That's the point. In order to use the **IFilterFactory**, for every filter class, we need an additional class, which we can call the **Filter Factory** class.

For example, for the **Response Header Action Filter**, we are going to create a **Header Filter Factory** class. Optionally, you can suffix the class name with "Attribute" and ensure that it inherits from the **Attribute** class, so that we can use this **Header Filter Factory** class as an attribute in the controller.

At the same time, we will implement the **IFilterFactory**, which means we can use this **Response Header Filter Factory** in the controller.

| Harsha

```
Attribute'  
    filterAttribute  
        arg2)  
  
context context)  
  
xt context)
```

We have to create an

ate a **Response**
with the word
can apply this **Filter**

are going to use


```
1  using Microsoft.AspNetCore.Mvc.Filters;
2
3  namespace CRUDExample.Filters.ActionFilters
4  {
5      public class ResponseHeaderFilterFactory : IFilterFactory
6      {
7          }
8      }
9
10     public class ResponseHeaderActionFilter : IOrderedFilter
11     {
12         private readonly string _key;
13         private readonly string _value;
14     }
15 }
```

Now, switch back to Controller just for a moment.

rs

Attribute : Attribute, ↪

: IAsyncActionFilter, ↪




```
controller.cs*  X PersonListFilter.cs  ResponseHeaderFilter.cs  Program.cs
role    ~%ORUDEExample.Controllers.PersonsController
58  |  LServiceFilterTypeFor<PersonsListAction>
59  |
39  |
40  // [TypeFilter(typeof(ResponseHeaderActionFilter),
41  new object[] { "MyKey-FromAction", "1" }, Order = 1)
42  [ResponseHeaderFilterFactory("MyKey-From-Action", 1)]
43  |
44  [TypeFilter(typeof(PersonsListResultFilter))]
45  [SkipFilter]
46  public async Task<IActionResult> Index(
47  string? searchString, string sortBy =
        (PersonResponse.PersonName), SortOrderOptions sortOrder =
        SortOrderOptions.ASC)
48  {
49  }
```

Let's also change at Controller level

```
ASCIISorter), Order = 4jj  
tionFilter), Arguments = ?  
MyValue-From-Action", ?  
  
FromAction", "MyValue- ?  
  
filter)])  
  
x(string searchBy, ?  
= nameof ?  
erOptions sortOrder = ?  
Harsha  
Web-University
```



```
12  [using ServiceContracts.Enums;
13
14  namespace CRUDExample.Controllers
15  {
16      [Route("[controller]")]
17      // [TypeFilter(typeof(ResponseHeaderActionFilter), 
18      // new object[] { "My-Key-From-Controller", "My-Value-From-Controller", 3 }, Order = 3)]
19      [ResponseHeaderFilterFactory("My-Key-From-Controller", "My-Value-From-Controller", 3)]
20      [TypeFilter(typeof(HandleExceptionFilter))]
21      [TypeFilter(typeof(PersonAlwaysRunResultFilter))]
22      public class PersonsController : Controller
23      {
24          // private fields
25          private readonly IPersonsService _personsService;
26          private readonly ICountriesService _countriesService;
```

A screenshot of a dark-themed IDE interface, likely Visual Studio Code. The main area shows a snippet of C# code:

```
rguments = args  
-From-  
r", "My-  
rsha  
iversity  
ce.
```

The word "args" is highlighted in green. Three red arrows point from the bottom of the image towards the three parameter names in the code: "args", "From", and "r". A red line also extends from the bottom right towards the "r" parameter. The status bar at the bottom left shows "New Section 1 Page 239".


```
4     {
5         public class ResponseHeaderFilterFactoryAttribute : Attribute, IFilterMetadata
6         {
7             bool IFilterFactory.IsReusable => false; //Currently ignore it
8
9             private string? Key { get; set; }
10            private string? Value { get; set; }
11            private int Order { get; set; }
12
13            public ResponseHeaderFilterFactoryAttribute(string key, string value, int order)
14            {
15                this.Key = key;
16                this.Value = value;
17                this.Order = order;
18            }
19
20
21            //Controller -> FilterFactory -> Filter (Controller will call
22            //will be called by the framework to create the filter object
23            IFilterMetadata IFilterFactory.CreateInstance(IServiceProvider provider)
24            {
25                //return filter object
26                return new ResponseHeaderActionFilter(Key, Value, Order);
27            }
28
29
30            public class ResponseHeaderActionFilter : IAsyncActionFilter, IOrderedFilter
31            {
32                private ILogger<ResponseHeaderActionFilter> _logger;
33                private readonly string _key;
34                private readonly string Value;
35
36                public int Order { get; set; }
37
38                //you can't inject constructor injection here
39                public ResponseHeaderActionFilter(string key, string value, int order)
40                {
41                    //this._logger = logger;
42                    this._key = key;
43                    this.Value = value;
44                }
45            }
46        }
47    }
48}
```

FilterFactory

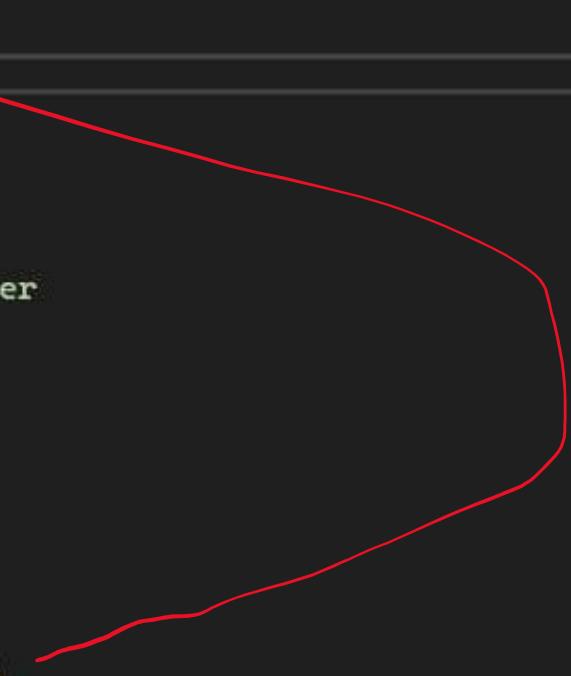
..

g value, int order)

FilterFactory to create the filter object)
(run-time)
or serviceProvider)

nderedFilter

nt order)



What problem does IFilter

FilterAttribute class [such as ActionFilterAttribute etc.]

Filter **CAN** be applied as an attribute to the controller or action method.

Eg: `[FilterClassName]`

Filter class **CAN'T** have DI with neither constructor injection nor method injection.

Filter class **CAN** receive arguments either through constructor parameters or filter class's properties.

Factory solve?

IFilterFactory

Filter **CAN** be applied as an attribute to the controller or action method.

Eg: [FilterClassName]

Filter class **CAN** have DI with either constructor injection or method injection.

Filter class **CAN** receive arguments only through filter class's properties, if it is instantiated through ServiceProvider (using DI).

Alternatively, if you don't need to inject services using DI in the filter class; you can instantiate the filter class with 'new' keyword, in the CreateInstance() method of IFilterFactory; then the filter class can receive arguments either as constructor parameters or properties.

A single, thin, horizontal red line is positioned near the bottom left corner of the page.

But you can not supply arguments for the dependency to the constructor of the class.
You have to convert the same constructor parameters into properties.

filter class like this.


```
controller.cs * ResponseHeaderFilter.cs [X] Program.cs
sample -> CRUDEExample.Filters.ActionFilters.ResponseHeaderFilterFactoryAttribute
CreateInstance(IServiceProvider serviceProvider)
14     Key = key;
15     Value = value;
16     Order = order;
17 }
18
19 //Controller -> FilterFactory
0 references
20 public IFilterMetadata Create(
21     serviceProvider)
22 {
23     var filter = new ResponseHeaderFilter(
24         Order);
25     //return filter object
26     return filter;
27 }
3 references
```

```
key -> Filter  
    .ServiceInstance(IServiceProvider  
        .AddFilter((key, value) =>  
            new HeaderActionFilter(key, value, filterOrder))  
    );  
  
    return services;  
}
```




```
ResponseHeaderFilters.cs  Program.cs
  ↵C:\UDExample\Filters\ActionFilters\ResponseHeaderActionFilter.cs
    ↵Order
21  {
22      var filter = new ResponseHeaderActionFilter();
23      //return filter object
24      return filter;
25  }
26 }
27
28 public class ResponseHeaderActionFilter : IOrderedFilter
29 {
30     private readonly string _key;
31     private readonly string _value;
32
33     public int Order { get; }
```

```
onFilter(Key, Value,  
er : IAsyncActionFilter,
```



3 references

public class ResponseHeader : IOrderedFilter

{

2 references

public string Key { get; set; }

2 references

public string Value { get; set; }

1 reference

public int Order { get; set; }

1 reference

2 references



```
    rActionFilter : IAsyncActionFilter  
  
    set; }  
  
; set; }  
  
}
```

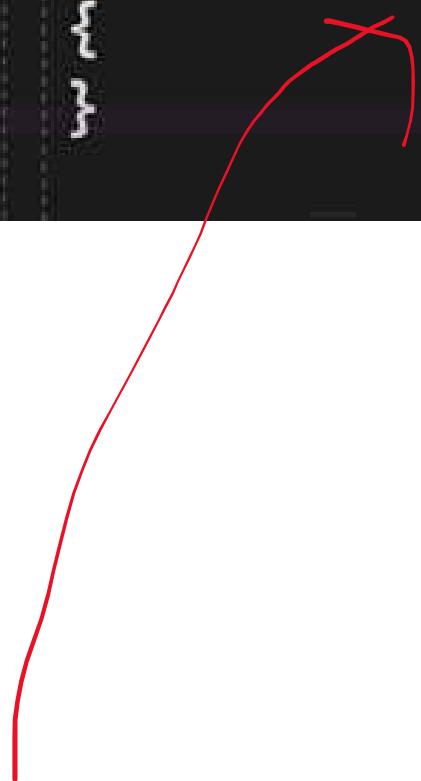


```
27  
28     public class ResponseHeaderActionFilter : IOrderedFilter  
29     {  
30         public string Key { get; set; }  
31         public string Value { get; set; }  
32         public int Order { get; }  
33  
34         public ResponseHeaderActionFilter(string key, string value, int order)  
35         {  
36             Key = key;  
37             Value = value;  
38             Order = order;  
39         }  
40     }
```

lter : IAsyncActionFilter




```
25      }
26  }
27
28  public class ResponseHeaderAction
29      : IOrderedFilter
30  {
31      public string Key { get; set; }
32      public string Value { get; set; }
33      public int Order { get; }
34
35  }
36
37
```



```
    nFilter : IAsyncActionFilter,    ?
```

```
}
```

```
er()
```



No need of initializing them here

```
public IFilterMetadata CreateFilter(IFilterMetadata filter, string key, string value, int order, IServiceProvider serviceProvider)
{
    var filter = new ResponseHeaderFilter();
    filter.Key = key;
    filter.Value = value;
    filter.Order = order;
    //return filter object
    return filter;
}
```

3 references

```
public class ResponseHeaderFilter : IFilterMetadata
```

```
CreateInstance(IServiceProvider  
    eHeaderActionFilter(Key, Value,
```

I

```
ActionFilter : IAsyncActionFilter
```



er,



Now, instead of you create an object of the filter class directly, you

It creates an object of 'ResponseHeaderActionFilter' through Depen

can inject the same through constructor injection through **IServiceProvider**.
dency injection.


```
10 //Controller -> FilterFactory
11
12 public IFilterMetadata CreateFilter(IFilterMetadataKey key,
13     IServiceProvider serviceProvider)
14 {
15     var filter =
16         serviceProvider.GetRequiredService<IFilterMetadataProvider>()
17             .GetFilter(key);
18
19     filter.Key = Key;
20     filter.Value = Value;
21     filter.Order = Order;
22
23     return filter;
24 }
25
26 }
```



```
ry -> Filter
```

```
CreateInstance(IServiceProvider
```

```
edService<ResponseHeaderActionFilte
```



It creates an object of 'ResponseHeaderActionFilter' through DI and make sure whether this class is added to the IOC container collection.

In 'Program.cs' file

```
31
32
33     builder.Services.AddTransient<ResponseHeaderAct
34
35     //it adds controllers and views as services
36     builder.Services.AddControllersWithViews(option
37     {
38         //options.Filters.Add<ResponseHeaderActionF
39
40         var logger = builder.Services.BuildServiceP
41
42         options.Filters.Add(new ResponseHeaderActio
43         {
44             Key = "My_key-From-Global",
45             Value = "My-Value-From-Global",
46             Order = 2
47         });
48     });
49 
```

Now run our application.

'Dependency Injection'. But we have to
lection.

```
ionFilter>();  
  
s =>  
  
ilter>(5); // we can also pass Order as a parameter in this way, then  
provider().GetRequiredService<ILogger<ResponseHeaderActionFilter>>();  
  
nFilter(logger)
```

we do not need to implement 'IOrderedFilter'

As Filters

ASP.NET Core

[VS] Middleware

Asp.Net Core

Filter Pipeline



Filter pipeline

↓ *entry*

Authorization Filter
(OnAuthorization)

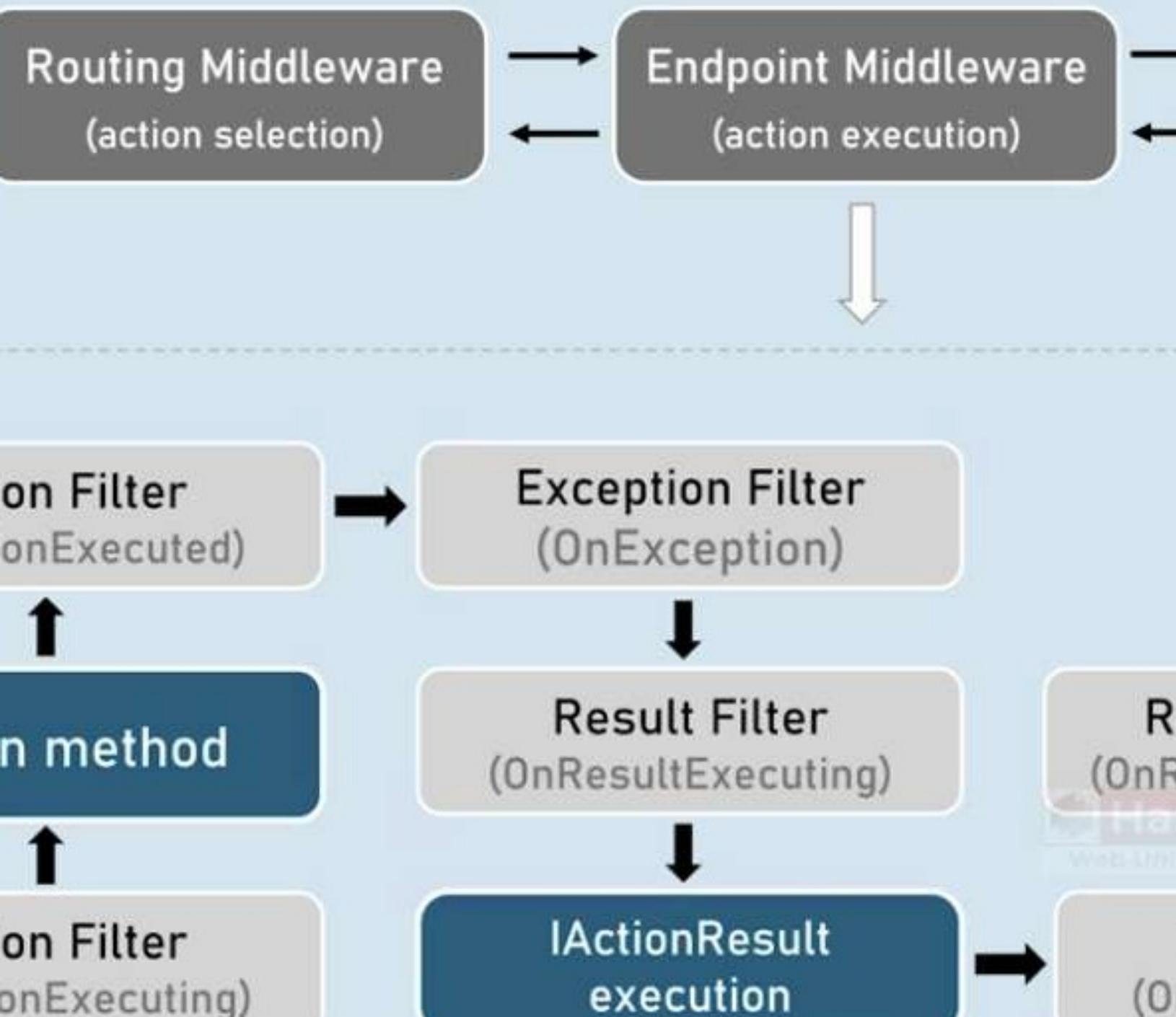
Action
(OnActionExecuting)

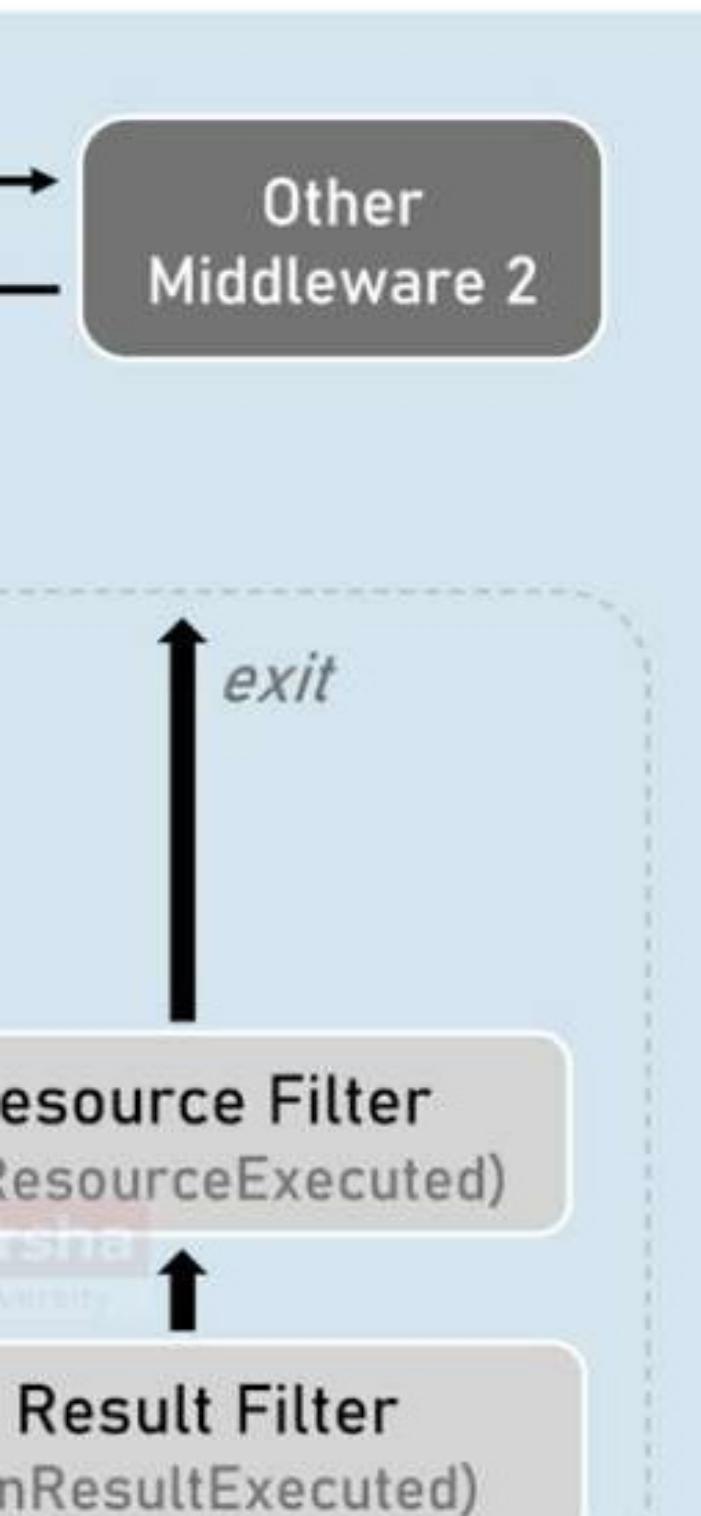
Resource Filter
(OnResourceExecuting)

Action
(OnActionExecuted)

Model Binding &
Validation

Action
(OnActionExecuted)





Asp.Net Core

Filters [vs] Middle

Middleware

Middleware pipeline is a superset of which contains the full-set of all middleware registered to the ApplicationBuilder in the application's code (Program.cs).

Middleware pipeline execute for all requests.

Middleware handles application-level logic.

eware

Filter

Filter pipeline,
Middlewares added
application's startup

requests.

l functionality

Filter pipeline is a subset of M
which executes under "EndPoint

In addition, filter pipeline exect
reach "EndPoint Middleware".

Result Executed)

Harsha

Middleware pipeline
point Middleware".

executes for requests that

Middleware handles application-level concerns such as Logging, HTTPS redirection, Request/Response modification, Application Profiling, Exception handling, Static file serving, Session management, Authentication etc., by accessing low-level abstractions such as HttpContext.

The difference between **middleware** and **filters** can help you answer the interview question:

1. Middleware

- **Purpose:** Middleware handles **application-level** concerns such as Logging, HTTPS redirection, Request/Response modification, Application Profiling, Exception handling, Static file serving, Session management, Authentication etc., by accessing low-level abstractions such as HttpContext.
- **Scope:** It's **global**—it applies to the entire request.
- **Examples:** Things like **authentication**, **authorization**, **logging** and **profiling** are handled by middleware.
- **Execution:** Middleware is executed **in the request pipeline**. Once all the middleware is processed, the request reaches the controller.

l functionality
Performance
iles,
-level

Filters handle MVC-specific functionality by manipulating or accessing View ModelState, Action result, ActionContext

can be a bit tricky, but let's break it down so that it's clear an

el functionality. It operates on **all requests** that pass through

est pipeline for the app.

Authorization, logging, error handling, CORS (Cross-Origin Resource Sharing)

uest pipeline. Once a request enters the pipeline, it goes through routing and eventually the endpoint (controller action).

unctionality such as
ewData, ViewBag,
ction parameters etc.

d easy to understand. Here's how I would

the system, regardless of which route or

source Sharing), and HTTPS redirection

ugh the series of middlewares, and once all

2. Filters

- **Purpose:** Filters perform **MVC-specific functions**.
- **Scope:** Filters are **narrower in scope** than middleware; they can access **controller parameters, view data, model state**, etc.
- **Examples:** **Authorization filters**, **action filters** (run before the controller action is run), and **exception filters** (run before the result is sent back).
- **Execution:** Filters are executed during the **endpoints' execution**; they are **run only for specific actions**, not for every request.

Key Differences

- **Middleware** is the **super-set**, handling **global things** like security or logging.
- **Filters** are the **sub-set**, focused on **controller actions** and **HTTP methods**.

When to Use Each

- **Use middleware** when you need **global functionality** that applies to all requests/responses globally.
- **Use filters** when you need to perform **specific logic** on specific **controller actions**, **HTTP methods**, **data**, or running code before/after the controller action.

onality and are usually concerned with **action methods** in co

iddleware. They run only for a **specific controller action** and a
e, and other MVC-specific details.

s (for logic before/after the action method runs), **result filters**
ilters (to handle exceptions in controller actions).

point middleware phase of the pipeline, after routing but bef
uest.

tasks that apply to all requests, and it's executed across the en

ction execution and specific to the MVC framework.

onality that applies to the entire application, like security, log

logic on an action method or during the MVC lifecycle, like w
r action.

ntrollers.

re primarily designed to interact with

s (for logic after the action method runs but

ore the action method itself executes. They

tire pipeline.

ging, error handling, or modifying

validating model data, manipulating view

Conclusion

- **Middleware is for application-wide concerns,**
- They can **never** be used interchangeably because they are specific to a process.

and **filters** are for **controller action-specific** concerns.
use they serve **different purposes** in the pipeline, though both

are part of the overall request handling

Config

Asp.Net Core

Azure Services Extension

on

```
File Edit View Git Project Build Debug Test Analyze Tools Extensions Window Help Search (F5)
Debug - Any CPU CRUDEexample CRUDEexample
Program.cs
50 options.Logging
51 Microsoft.AspNetCore.Mvc
52 properties |
53 Microsoft.AspNetCore.Mvc
54 propertiesAndHead
55 });
56 var app = builder
57 app.UseSerilogRe
58 if (builder.Env:
59 {
60     app.UseDevelop
61 }
62 app.UseHttpsOnc
63
```

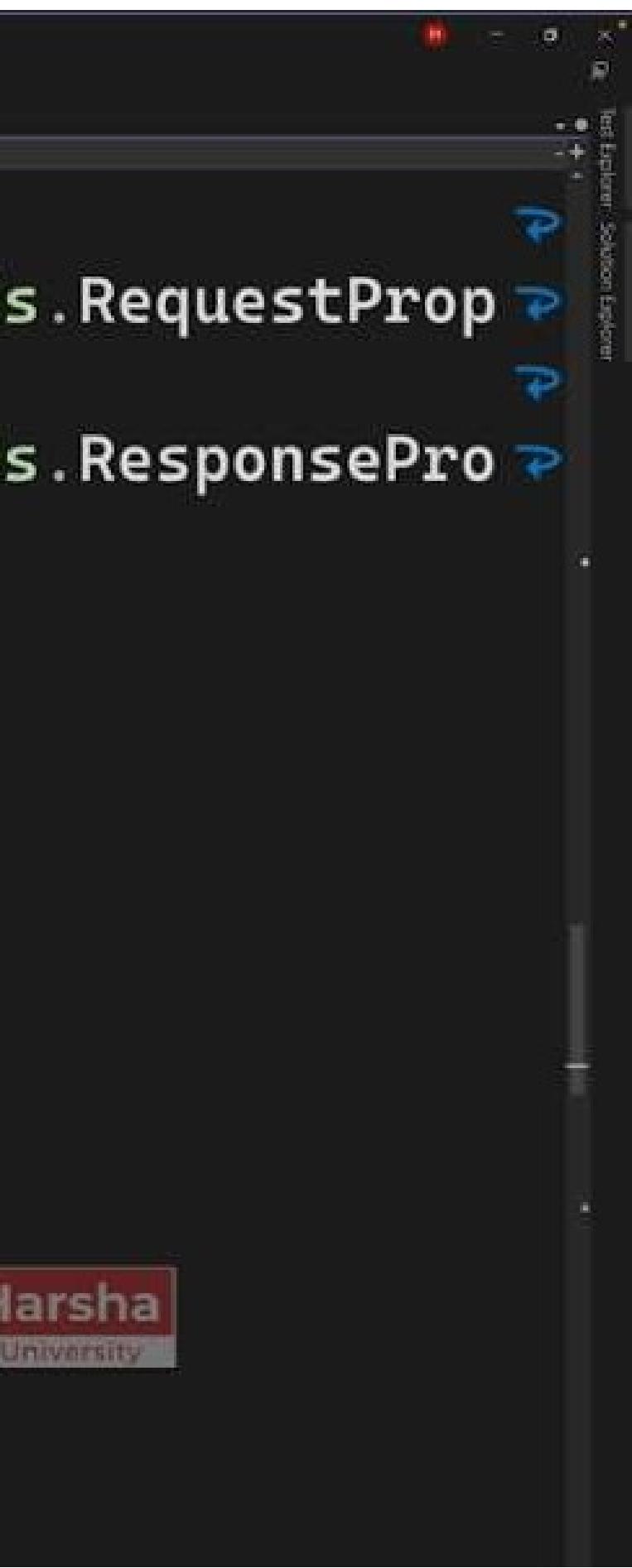
```
    <!-- stop-level-statements-entry-point -->
    loggingFields =
        NetCore.HttpLogging.HttpLoggingFieldHeaders;
}

builder.Build();

requestLogging();

// Application pipeline
if (env.Environment.IsDevelopment())
{
    builderExceptionPage();
}
```






```
CRUDSolution
New Statement Entry Point
addScoped<IPersonsRepository,
>();
addScoped<ICountriesService, Countries
>();
addScoped<IPersonsService, PersonsService
>();
addDbContext<ApplicationDbContext>(opt
ionBuilder => opt
    .AddDbContext<ApplicationDbContext>(builder
        .Configure(builde
        .ConnectionString("DefaultConnection"))
    );
    .AddTransient<PersonsListActionFilter>(
        builder =>
            builder
                .AddHttpLogging(options =>
                    options
                        .ConfigureHttpLogging(log =>
                            log
                                .SetCategoryLevel(LogLevel.Information)
                                .SetProviderLevel(LogLevel.Information)
                                .SetFilter(filter =>
                                    filter
                                        .IncludeFormattedMessage()
                                        .IncludeResponseContent()
                                )
                        )
                )
    );
}

```

It is a service collection type

A screenshot of a dark-themed IDE interface, likely Visual Studio Code, showing code completion suggestions. The code editor displays the following C++ code:

```
Service>    ↵
ice>();
```

The cursor is positioned after the opening parenthesis of the constructor call. A code completion dropdown menu is open, listing several suggestions:

- Service< (highlighted)
- Service<::Service<
- Service<::Service<::Service<
- Service<::Service<::Service<::Service<
- Service<::Service<::Service<::Service<::Service<
- Service<::Service<::Service<::Service<::Service<::Service<

The background of the interface is dark, and the code completion UI has a light gray background with blue outlines for the suggestions.

```
    builder.Services.AddApplicationServices();
    builder.Services.AddInfrastructureServices();
    builder.Services.AddWebApplicationServices();

    var options = builder.Services
        .GetService<IOptions<DefaultConnectionOptions>>()
        .Value;
```

```
    services.AddScoped<IPersonService>(sp =>
        sp.GetService<IPersonRepository>()
    );
    services.AddDbContext<ApplicationDbContext>(options =>
        options.UseSqlServer(builder.Configuration.GetConnectionString("DefaultConnection"))
    );
}
```





A screenshot of a .NET Core application's startup code in `Program.cs`. The code uses the `IWebHostBuilder` interface to configure services and middleware. The `AddServices` method is being called on the `builder` object. The configuration includes adding services from `PersonsRepository`, `PersonsService`, `DbOptions` (using `SqlServer` with `DefaultConnection`), and `TracingService`.

```
36 builder.Services.AddService<IPersonsRepository>();
37
38 builder.Services.AddService<IPersonsService>();
39 builder.Services.AddService<IDbOptions>();
40
41 builder.Services.AddDbContext<DefaultContext>(
42     options =>
43         options.UseSqlServer("DefaultConnection"));
44
45
46 builder.Services.AddTracing();
```

You're absolutely right! In real-world applications, the service registration and middleware logic in the `ConfigureServices` and `Configure` methods can become quite complex and hard to maintain. Let's go through how we can make this code more readable, and easier to maintain. Let's go through **extension methods**.

```
coped<IPersonsRepository,>
coped<ICountriesService, CountriesService>
coped<IPersonsService, PersonsService>();
>Context<ApplicationDbContext>(options =>
(builder.Configuration.GetConnectionString
));
I
transient<PersonsListActionFilter>();
```

, the Program.cs file can quickly become **too lengthy and hard to read**. To keep the code in one place. Splitting the code into **multiple files** makes you able to reuse the code in other parts of your application. In this section, we will learn how you can **refactor** the code and move the service registration logic to a separate file.



rd to maintain if you put all
ur project more modular,
tion to a separate file using

Steps to Refactor the Program.cs File

1. Create an Extension Method for Service Registration

First, you'll need to create an extension method for your service class file, say ServiceExtensions.cs.

ServiceExtensions.cs:

using Microsoft.Extensions.DependencyInjection;
public static class ServiceExtensions

{

 public static void AddCustomServices(this IServiceProvider services)

{

 // Add your services here

 // Example:

 services.AddScoped<IMyService, MyService>();

 services.AddSingleton<ILoggingService, LogService>();

 // You can also add other service configurations, like this:

}

}

2. Refactor Program.cs to Use the Extension Method

Now, in your Program.cs, you just need to **call this extension method**, which is focused on **high-level configuration**.

stration

or `IServiceCollection` to handle the service registrations. You can

`ServiceCollection services)`

`e>();`

`loggingService>();`

like adding a database context or caching

method

This extension **method** to add all your services, keeping the Program

can put this code in a separate

gram.cs file cleaner and more

Program.cs (Refactored):

```
Program.cs (Refactored):  
C#  
  
using Microsoft.AspNetCore.Builder;  
using Microsoft.Extensions.DependencyInjection;  
using Microsoft.Extensions.Hosting;  
  
var builder = WebApplication.CreateBuilder();  
  
// Add services to the container using the  
builder.Services.AddCustomServices(); // T  
  
// Configure the HTTP request pipeline.  
var app = builder.Build();  
  
if (app.Environment.IsDevelopment())  
{  
    app.UseDeveloperExceptionPage();  
}  
else
```

 Copy  Edit

tion;

(args);

extension method

This line now just refers to your extension method



```
else
{
    app.UseExceptionHandler("/Home/Error");
    app.UseHsts();
}

app.UseHttpsRedirection();
app.UseStaticFiles();

app.UseRouting();

app.MapControllerRoute(
    name: "default",
    pattern: "{controller=Home}/{action=Index}/{id?}");

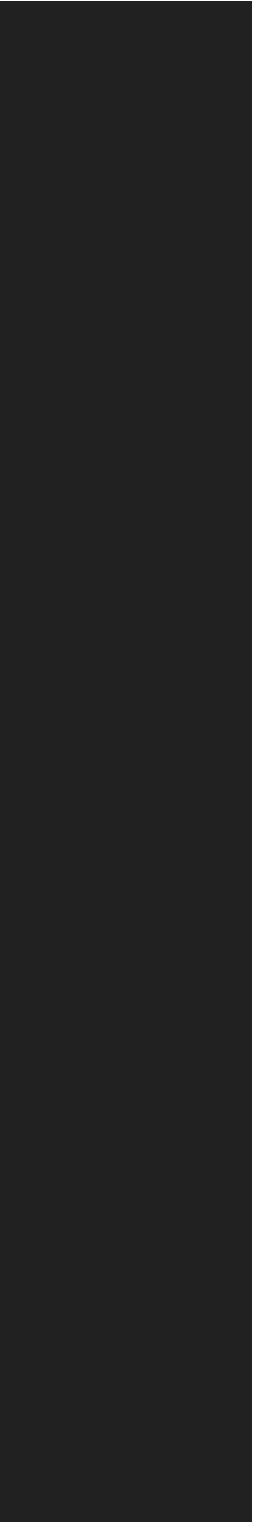
app.Run();

```

3. Benefits of This Refactoring

dex}/{id?});





- **Cleaner Code:** Your Program.cs file is now much cleaner.
- **Separation of Concerns:** The services-related logic is now separated from the main application logic.
- **Easier Maintenance:** In larger projects, managing multiple service registrations grow.
- **Scalability:** As the project grows, you can further abstract the logic into separate services (e.g., data access, logging, caching, etc.).

4. Additional Tips

- You can keep adding more methods to the Service class, like additional service registrations or middleware configuration, etc.
- For complex middleware setups, consider using extension methods to keep the code clean and readable.

Conclusion

By using extension methods like this, you can easily manage both service registrations and middleware logic in one place, making your application easier to maintain and scalable.

In shorter and more readable, focusing only on high-level configuration logic is moved to a separate class, keeping each file focused on managing dependencies is easier when they are in their own files, especially if you break down ServiceExtensions.cs into more specialized files per service. You can also use extension methods to handle different aspects of the application's configuration.

extension methods for them as well to keep Program.cs clean.

Finally, keep your **Program.cs** file lean and maintainable, and you'll be in good place. This is a common practice in larger applications to ensure that the main entry point is clean and focused on starting the application.

figuration.

a single responsibility.

ecially as your service

for different areas of

tion, like middleware setup,

don't have to keep all your
ure the codebase is organized

```
    ced<IPersonsService
```

```
extension
```

```
    ontext<ApplicationD
```

```
        ilder.Configuration
```

```
    );
```

```
    nsient<PersonsListA
```

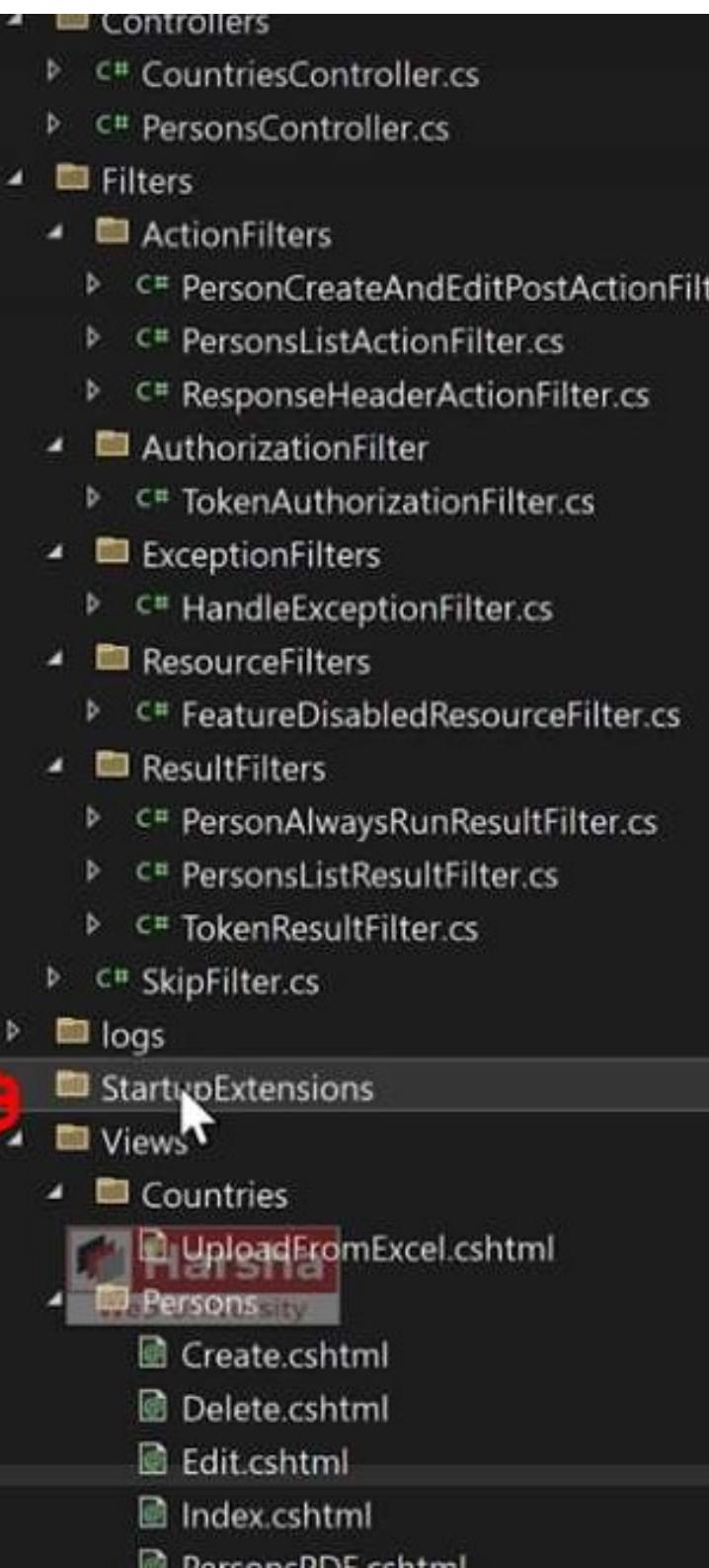
```
    loggingsOptions ->
```

e, PersonsSe

```
DbContext>()
```

```
on.GetConnec
```

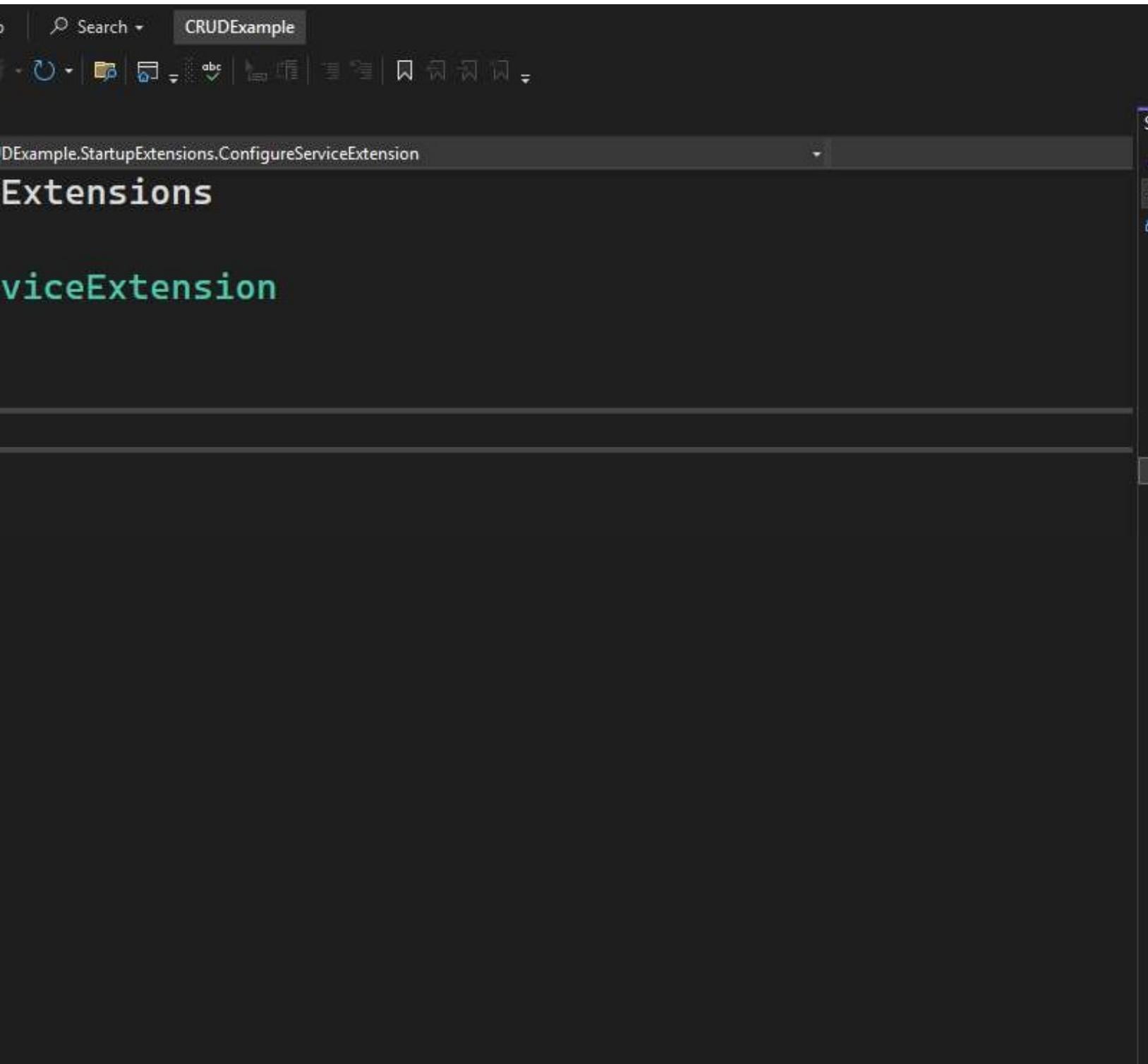
ActionFilte

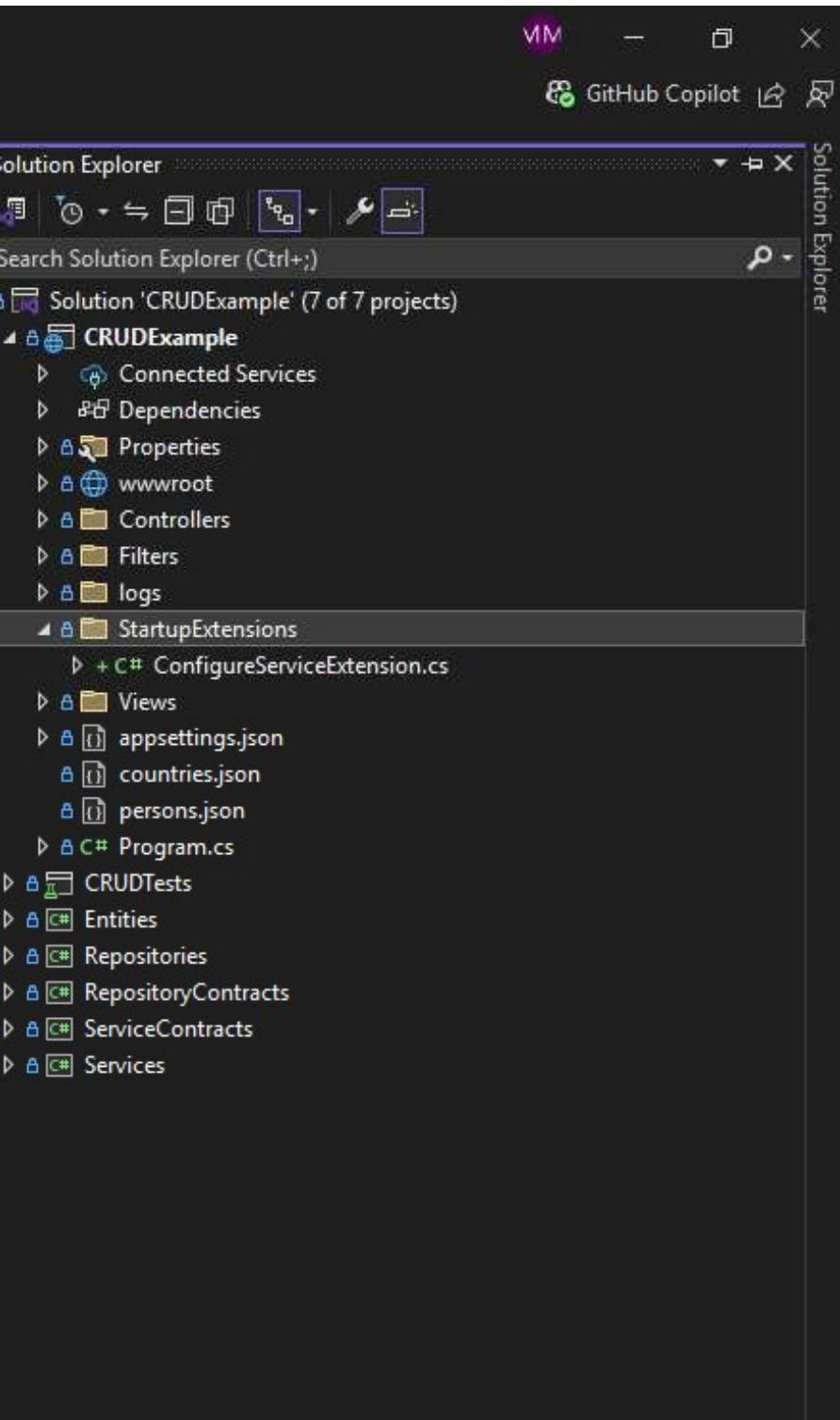


The screenshot shows the Visual Studio IDE interface with the following details:

- Menu Bar:** File, Edit, View, Git, Project, Build, Debug, Test, Analyze, Tools, Extensions, Window, Help.
- Toolbar:** Standard icons for New, Open, Save, Print, etc.
- Status Bar:** Debug, Any CPU, CRUDEExample, http.
- Code Editor:** The file `ConfigureServiceExtension.cs` is open. The code is as follows:

```
namespace CRUDEExample.Startup
{
    public class ConfigureServices
    {
    }
}
```
- Solution Explorer:** Shows the project `CRUDEExample`.
- Task List:** Shows the task `CRUDEExample`.

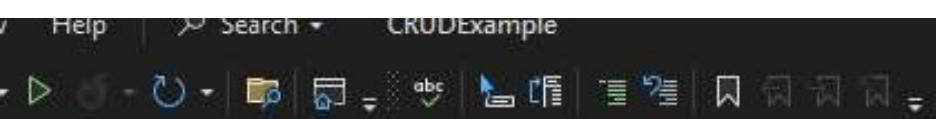




A screenshot of the Visual Studio IDE interface. The menu bar includes File, Edit, View, Git, Project, Build, Debug, Test, Analyze, Tools, Extensions, and Windows. The toolbar contains icons for file operations like Open, Save, and Print. The status bar shows "Debug Any CPU CRUDEExample http://". The code editor window has tabs for "ConfigureServiceExtension.cs" and "X". The title bar says "CRUDEExample". The code itself is:

```
1  namespace CRUDEExample.StartUp
2  {
3      public static class Configuration
4      {
5          public static void ConfigureServices(IServiceCollection services)
6          {
7              // Configuration logic here
8          }
9      }
10 }
11
```

The code editor features color-coded syntax highlighting and a vertical green margin line on the left side. A blue screwdriver icon is positioned near the top of the margin line. A cursor is visible at the start of the "ConfigureServices" method definition.



CRUDExample.StartupExtensions.ConfigureServiceExtension

ConfigureService

tupExtensions

ConfigureServiceExtension

```
ConfigureServices(this IServiceCollection services)
```

es(IServiceCollection services)

es)

The screenshot shows the Visual Studio IDE interface with the following details:

- Menu Bar:** File, Edit, View, Git, Project, Build, Debug, Test, Analyze, Tools, Extensions, Window, Help.
- Toolbar:** Standard icons for file operations like Open, Save, Print, etc.
- Status Bar:** Debug, Any CPU, CRUDEExample, http.
- Code Editor:** The file "Program.cs" is open, showing C# code for a .NET application. The code includes configuration for Serilog logging and services.
- Code Content:**

```
20      //    //LoggingProvider.AddDe
21      //    //loggingProvider.AddEv
22      //});
23
24      //replace existing logging me
25      builder.Host.UseSerilog((Host
26      loggerConfiguration logger
27      {
28          loggerConfiguration
29              .ReadFrom.Configuration()
30              .ReadFrom.Services(service
31      });
32
33  builder.Services.ConfigureSer
34
35  builder.Services.AddTransient(
36
37      //it adds controllers and vie
```

Search → CRUDExample

bug();
entLog();

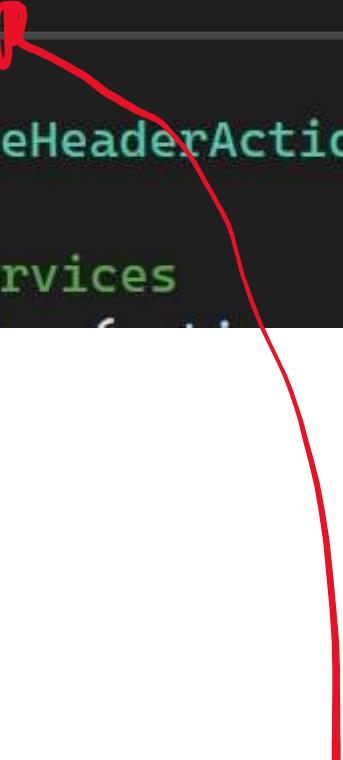
chanism with serilog
BuilderContext context, IServiceProvider services,
Configuration) =>

context.Configuration) // means, reading the configurat
ces); // this statemenet makes our service collection a

vices();

<ResponseHeaderActionFilter>();

ws as services



ion from
available