🔷 SECTION 7: FILTERS IN ASP.NET CORE (THEORY) 🔷

This is a high-value interview section. Many candidates confuse middleware vs filters, or don't know execution order — interviewers love this topic.

1️⃣ What are Filters?

🧠 Filters are like checkpoints **around a controller action**.

Before an action runs:

- Check permissions
- Validate input
- Log information

After an action runs:

- Modify response
- Handle errors

They run **only for MVC actions**, not for every request.

🎯 **Filters are MVC-specific components that:**

- Execute at specific stages of action execution
- Surround controller action logic
- Handle cross-cutting concerns within MVC

📌 **Filters run after routing but before/after action execution.**

🔁 Q: Filters vs Middleware?

✔️ Middleware → global

✔️ Filters → MVC only

2️⃣ Types of Filters (INTERVIEW MUST)

ASP.NET Core provides **5 types of filters**:

1️⃣ Authorization
2️⃣ Resource
3️⃣ Action
4️⃣ Exception
5️⃣ Result

3️⃣ Authorization Filters

🧠 Authorization filters check: “Is this user allowed to access this action?” 🎯

- Runs first
- Before model binding
- Used for authorization logic
- Example: `[Authorize]`

📌 If authorization fails → request never reaches action.

🔁 Q: Can authorization filters short-circuit execution? ✔ Yes

4️⃣ Resource Filters

🧠 Resource filters wrap **everything** related to an action. 🎯

- Runs before model binding
- Runs after result execution
- Used for:
    - Caching
    - Resource initialization

📌 **Executes only once, unlike action filters.**

🔁 Q: Why use resource filter over action filter?

✔ To cover model binding + result execution together

5️⃣ Action Filters

🧠 Action filters run:

- Before action method
- After action method

🎯 Used for:

- Logging
- Validation
- Modifying inputs or outputs

Implements:

```
IActionFilter
IAsyncActionFilter
```

📌 **Cannot catch exceptions thrown by result filters.**

🔁 Q: Can action filters access ModelState? ✔ Yes

6️⃣ Exception Filters

🧠 Exception filters catch errors from actions.

🎯

- Runs when action throws exception
- MVC-specific error handling
- Does NOT catch middleware exceptions

📌 **Prefer middleware for global exception handling.**

🔁 Q: Why middleware is better for exceptions?

✔️ Covers whole pipeline, not just MVC

7️⃣ Result Filters

🧠 Result filters run around the response.

🎯 Used to:

- Modify response headers
- Log response data
- Runs before & after action result execution.

8️⃣ Filter Execution Order (VERY IMPORTANT)

Order (Memorize this)

```
Authorization Filter
Resource Filter
Action Filter
Action Method
Exception Filter
Result Filter
```

📌 **Resource filters wrap action + result.**

🔁 Q: Do exception filters catch middleware exceptions?

❌ No

✔️ Only MVC action exceptions

9️⃣ Filter Scope (INTERVIEW FAVORITE)

Filters can be applied at:

1️⃣ Global
2️⃣ Controller
3️⃣ Action

Priority: **Action > Controller > Global**

🔟 Sync vs Async Filters

- Prefer async filters
- Avoid blocking code
- Improves scalability

🧠 ONE-LINE INTERVIEW ANSWER (MEMORIZE)

“Filters in ASP.NET Core allow executing custom logic at specific stages of MVC request processing, such as authorization, action execution, exception handling, and result processing.”

✅ YOU NOW UNDERSTAND

✓ What filters are	✓ Filter types	✓ Execution order
✓ Middleware vs filters	✓ Scope & priority	✓ Interview traps

🔧 HANDS-ON 7: FILTERS IN ASP.NET CORE 🔧

🧱 STEP 0: Project Setup

```
dotnet new webapi -n FiltersDemo
cd FiltersDemo
```

🧱 STEP 1: Action Filter (Before & After Action)

📁 Create folder: **Filters**

```
public class LoggingActionFilter : IActionFilter
{
    public void OnActionExecuting(ActionExecutingContext context)
    {
        Console.WriteLine("➡️ Action executing");
    }

    public void OnActionExecuted(ActionExecutedContext context)
    {
        Console.WriteLine("⬅️ Action executed");
    }
}
```

Register Globally

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

📌 **Action filters run around action execution and can access ModelState.**

🧱 STEP 2: Resource Filter (Wraps Model Binding)

```
public class TimingResourceFilter : IResourceFilter
{
    private Stopwatch _stopwatch;

    public void OnResourceExecuting(ResourceExecutingContext context)
    {
        _stopwatch = Stopwatch.StartNew();
        Console.WriteLine("⏱ Resource executing");
    }
    public void OnResourceExecuted(ResourceExecutedContext context)
    {
        _stopwatch.Stop();
        Console.WriteLine($"⏱ Resource executed in
{_stopwatch.ElapsedMilliseconds} ms");
    }
}
```

Register:

```
options.Filters.Add<TimingResourceFilter>();
```

📌 **Resource filters wrap model binding and result execution.**

🧱 STEP 3: Authorization Filter (Custom)

```
public class ApiKeyAuthorizationFilter : IAuthorizationFilter
{
    public void OnAuthorization(AuthorizationFilterContext context)
    {
        if (!context.HttpContext.Request.Headers.ContainsKey("x-api-
key"))
        {
            context.Result = new UnauthorizedResult();
        }
    }
}
```

Register:

```
options.Filters.Add<ApiKeyAuthorizationFilter>();
```

📌 **Authorization filters run first and can short-circuit execution.**

🧱 STEP 4: Exception Filter (MVC-level Error Handling)

```
public class GlobalExceptionFilter : IExceptionFilter
{
    public void OnException(ExceptionContext context)
    {
        context.Result = new ObjectResult("Handled by Exception Filter")
        {
            StatusCode = 500
        };

        context.ExceptionHandled = true;
    }
}
```

Register:

```
options.Filters.Add<GlobalExceptionFilter>();
```

📌 **Exception filters catch only MVC action exceptions.**

🧱 STEP 5: Result Filter (Response-Level)

```
public class ResponseHeaderFilter : IResultFilter
{
    public void OnResultExecuting(ResultExecutingContext context)
    {
        context.HttpContext.Response.Headers.Add("X-Filtered", "true");
        Console.WriteLine("📦 Result executing");
    }

    public void OnResultExecuted(ResultExecutedContext context)
    {
        Console.WriteLine("📦 Result executed");
    }
}
```

Register:

```
options.Filters.Add<ResponseHeaderFilter>();
```

🧱 STEP 6: Test Controller

```
[ApiController]
[Route("api/test")]
public class TestController : ControllerBase
{
    [HttpGet]
    public IActionResult Get()
    {
        Console.WriteLine("🎯 Action method");
        return Ok("Success");
    }

    [HttpGet("error")]
    public IActionResult Error()
    {
        throw new Exception("Boom!");
    }
}
```

🧱 STEP 7: Observe Execution Order (INTERVIEW FAVORITE)

Call: `GET /api/test`

Console output order:

```
Authorization Filter
Resource executing
Action executing
🎯 Action method
Action executed
Result executing
Result executed
Resource executed
```

📌 **"Resource filters wrap the entire MVC execution."**

🧱 STEP 8: Exception Flow

Call: `GET /api/test/error`

Handled by: ✓ Exception Filter ❌ Middleware (if not configured)

📌 **Exception filters handle MVC exceptions only.**

🧱 STEP 9: Filter Scope & Priority

Apply at Controller Level

```
[ServiceFilter(typeof(LoggingActionFilter))]
public class ScopedController : ControllerBase
{
}
```

Priority: **Action > Controller > Global**

📌 **Local filters override global ones.**

🧱 STEP 10: Async Filters (BEST PRACTICE)

```
public class AsyncActionFilter : IAsyncActionFilter
{
    public async Task OnActionExecutionAsync(
        ActionExecutingContext context,
        ActionExecutionDelegate next)
    {
        Console.WriteLine("Async before");
        await next();
        Console.WriteLine("Async after");
    }
}
```

📌 **Prefer async filters for non-blocking behavior.**

🧠 FINAL INTERVIEW SUMMARY (MEMORIZE)

"ASP.NET Core filters allow injecting logic at different stages of MVC execution such as authorization, resource handling, action execution, exception handling, and result processing. Execution order and scope are critical for correct behavior."

✅ YOU NOW MASTER (HANDS-ON)

✓ All filter types	✓ Execution order	✓ Short-circuiting
✓ Exception handling	✓ Scope & priority	✓ Async filters