

Application Flow, Filters & Middleware

Application Fundamentals, Errors, Filters, Middleware



SoftUni Team
Technical Trainers



SoftUni



Software University

<https://softuni.bg>

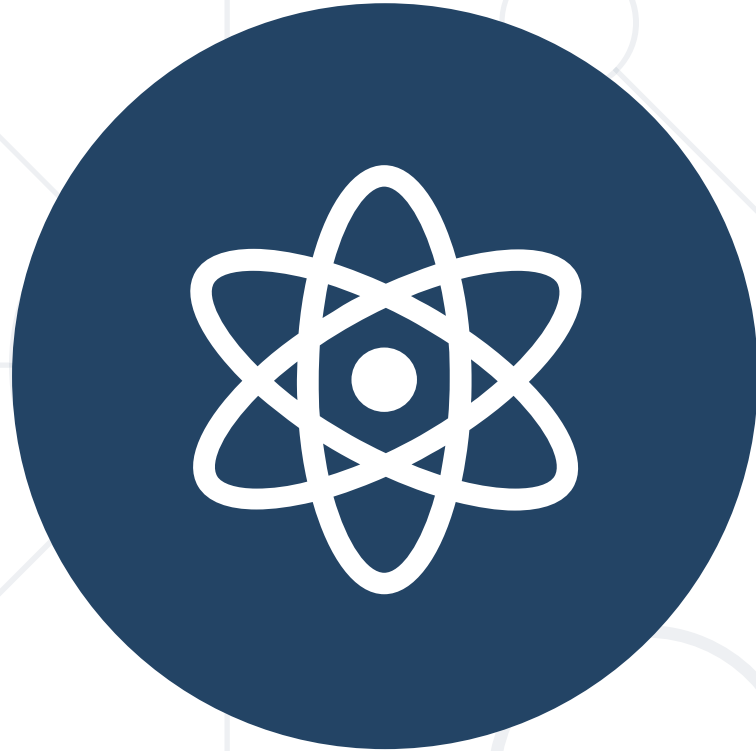
sli.do

#csharp-web

Table of Contents

1. Application Flow
2. Middleware
3. Filters

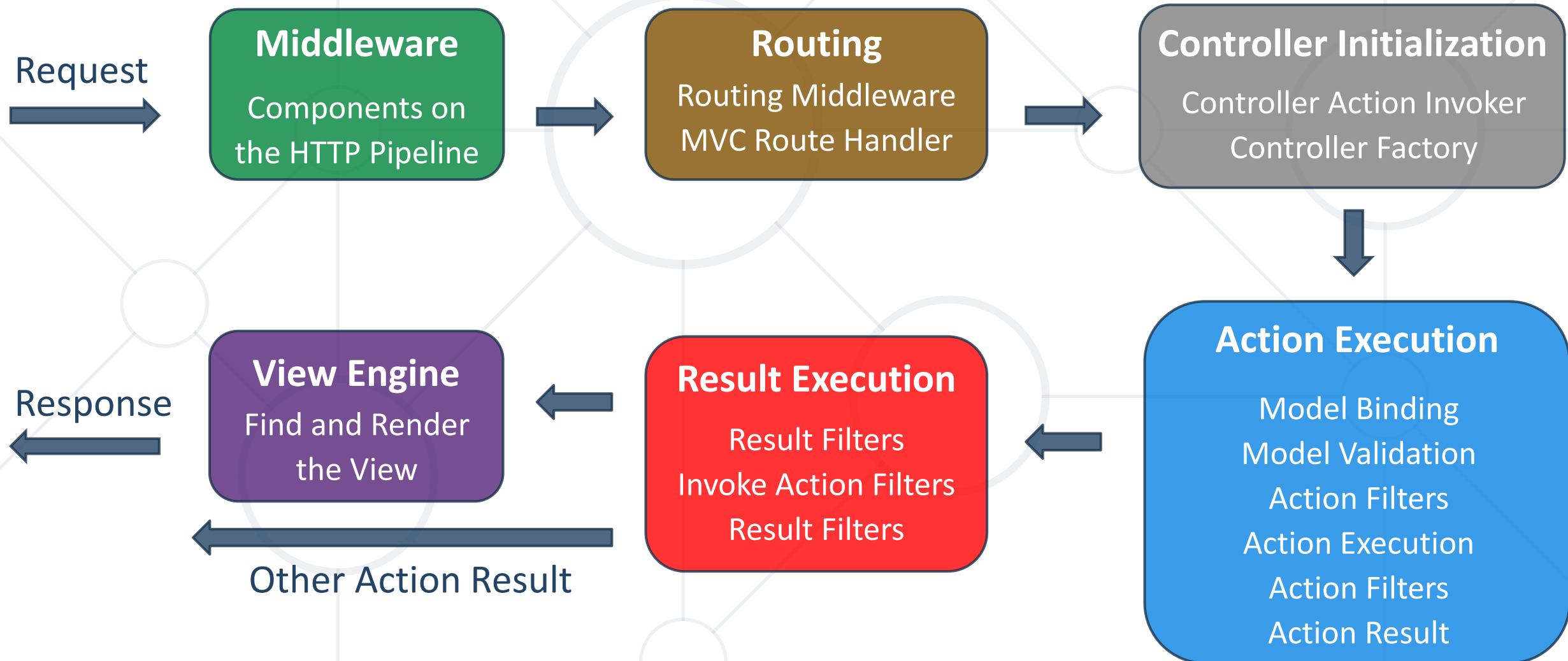




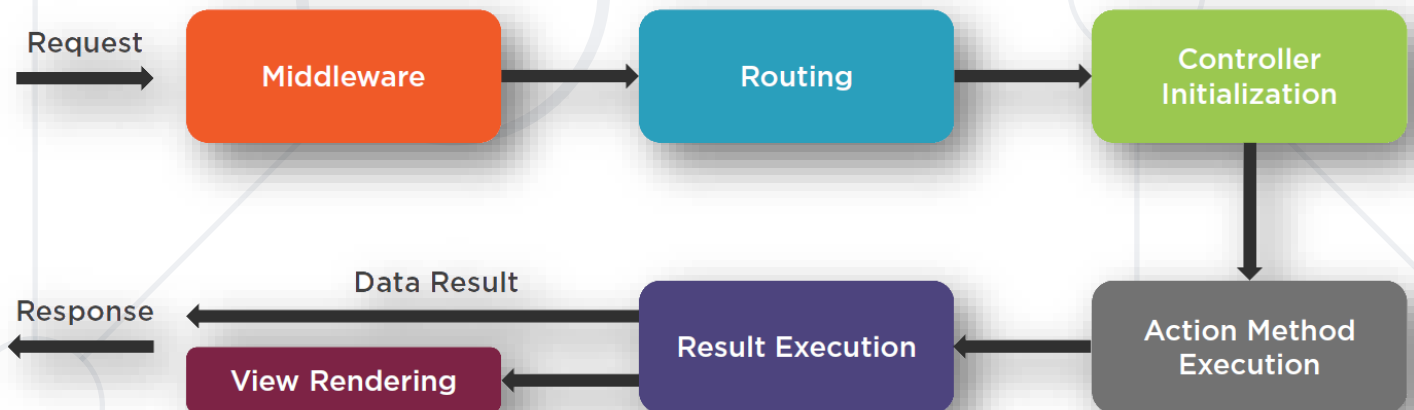
Application Flow

- **Web Applications** handle **Requests** and produce **Responses**
 - The whole process is naturally ordered in some kind of **pipeline**
 - In most cases, the process is **extensible** and **modifiable**
- **Web Applications** have different **deployment environments**
 - The environments determine the **behavior** of the application
 - The environments may also affect the pipeline
- **Web Applications** have initial **configuration**
 - Host, Security, Directories, Conventions, etc.

MVC Request Lifecycle



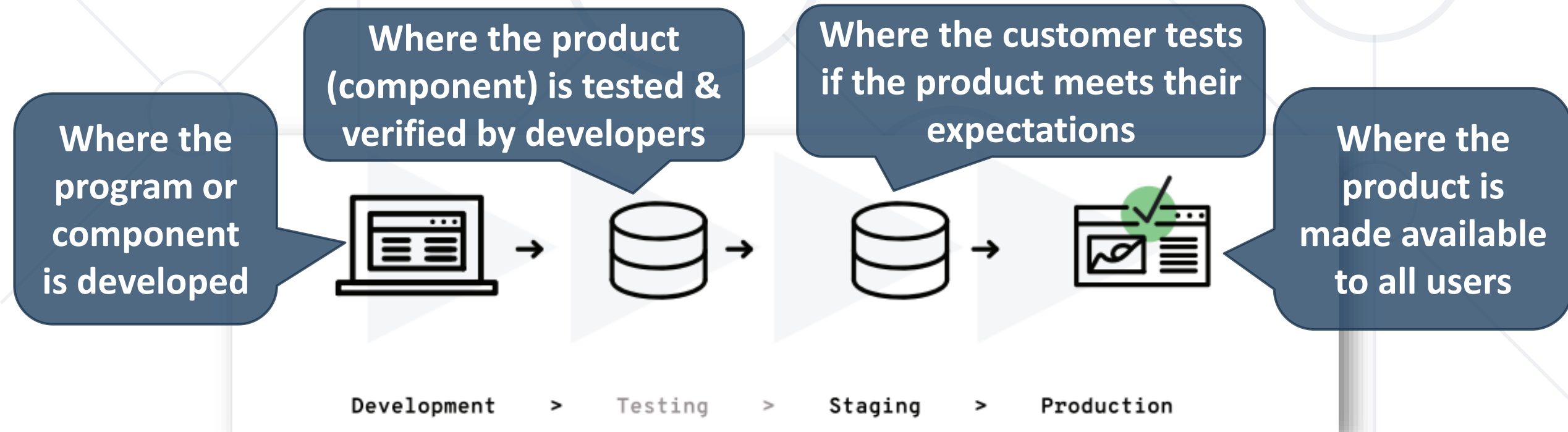
- The **Controller** is one of the main components in the **Request pipeline**
 - Each **Controller** has its own **ControllerContext**
 - A set of useful properties containing data about the current **Request**
- **ControllerContext** properties
 - **ActionDescriptor**
 - **HttpContext (Request, Response)**
 - **ModelState**
 - **RouteData**
 - **ValidProviderFactories**



- **ASP.NET Core** uses the **Program.cs** class to
 - Configure the HTTP Request Pipeline
 - Define behavior for different environments

```
if (!app.Environment.IsDevelopment())  
{  
    app.UseExceptionHandler("/Home/Error");  
    app.UseHsts();  
}  
  
app.UseHttpsRedirection();  
app.UseStaticFiles();  
app.UseCookiePolicy();  
  
app.UseMvcWithDefaultRoute();
```


- **Software Deployment** is usually distributed into several **environments**
 - Multi-stage deployment is a **MUST** in Enterprise applications
 - A **computer system** (**virtual** or **real**) which runs your software

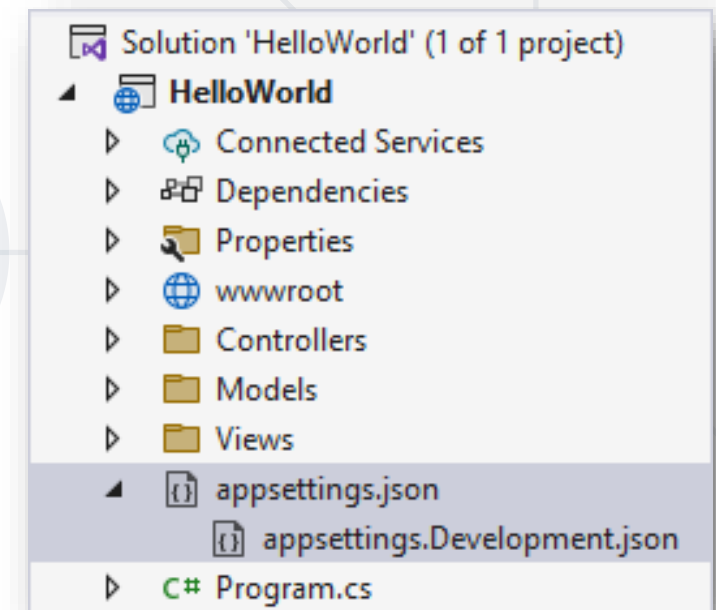


- Most environment architectures use the following environments
 - **Dev**
 - The program or component is developed
 - **Test**
 - The product (component) is tested & verified by developers
 - **Stage**
 - The customer tests if the product meets their expectations
 - **Production**
 - The product is made available to all users

- ASP.NET Core configures app behavior based on **runtime environment**
 - Supports 3 environments – **Development**, **Staging** and **Production**
 - Reads the **Environment variable** "**ASPNETCORE_ENVIRONMENT**"
 - Environment value is stored in the **EnvironmentName** property of the **Environment** property of the **WebApplication**, returned by the **Build()** method of the **WebApplicationBuilder**
- The environment can be set to **any value**. The default environment is **Production**

```
if(app.Environment.IsDevelopment()) // TODO: Do Development
if(app.Environment.IsStaging()) // TODO: Do Staging
if(app.Environment.IsProduction()) // TODO: Do Production
if(app.Environment.IsEnvironment("some_environment")) // TODO: Do Something
```

- App configuration in **ASP.NET Core** is based on **key-value** pairs
 - App configurations are specified in **configuration providers**
- **Configuration providers** read data from **configuration sources**
 - Azure Key Vault, Command-line arguments
 - Custom providers (installed or created)
 - Directory files, Environment variables, etc.
- One of the default sources is **appsettings.json**



- App configuration is read at **app startup** from the providers
 - Configuration properties are mapped in **IConfiguration**
 - **IConfiguration** is available in the app's **DI container**

appsettings.json

```
{
  "Greeting": "Hello!",
  "Config": {
    "Secret":
      "Can't touch this"
  },
  ...
}
```

HomeController.cs

```
public class HomeController : Controller
{
    private readonly IConfiguration config;
    public HomeController(IConfiguration config)
    {
        this.config = config;
    }
    public IActionResult Config()
    {
        return Content(this.config["Greeting"]);
        // Hello!
    }
}
```

- **Configuration options**, by convention, are set in **Program.cs**
 - Configured before **WebApplicationBuilder** builds the app
 - Accessed by the **Configuration** property of the **WebApplicationBuilder**, returned by the **WebApplication.CreateBuilder(args)** method
 - Typical pattern is adding the service and then configuring it
- Adding services to the service container
 - Makes them available within the app
 - Resolved via **Dependency Injection**
 - Makes them available within the **Program** class

- Services can be configured for **Dependency Injection** differently

```
// Transient objects are always different  
// A new instance is provided to every controller and service  
builder.Services.AddTransient<DataService>();  
  
// Scoped objects are the same within a request  
// They are different across different requests  
builder.Services.AddScoped(typeof(DataService));  
  
// Singleton objects are the same for every object and request.  
builder.Services.AddSingleton<DataService>();
```



Diagnostics & Custom Error Handlers

Error Handling

- There are several ways to configure **Error handling** in **ASP.NET Core**
 - Developer Exception Page
 - Exception Handler
 - Status Code Pages
- **ASP.NET Core MVC** apps have additional options for handling errors
 - Exception Filters
 - Model Validation (**ModelState**)

Error Handling (Developer Exception Page)

Internal Server Error x +
https://localhost:44317

An unhandled exception occurred while processing the request.

Exception: Something went wrong...

WebApplication1.Controllers.HomeController.Index() in HomeController.cs, line 31

Stack Query Cookies Headers

Exception: Something went wrong...

```
WebApplication1.Controllers.HomeController.Index() in HomeController.cs
31.         throw new Exception("Something went wrong...");
lambda_method(Closure , object , object[])
Microsoft.Extensions.Internal.ObjectMethodExecutor.Execute(object target, object[] parameters)
```

Internal Server Error x +
https://localhost:44317

An unhandled exception occurred while processing the request.

Exception: Something went wrong...

WebApplication1.Controllers.HomeController.Index() in HomeController.cs, line 31

Stack Query Cookies Headers

No QueryString data.

Internal Server Error x +
https://localhost:44317

An unhandled exception occurred while processing the request.

Exception: Something went wrong...

WebApplication1.Controllers.HomeController.Index() in HomeController.cs, line 31

Stack Query Cookies Headers

Variable	Value
Accept	text/html,application/xhtml+xml,application/xml;q=0.9,image/webp,image/apng,*/*;q=0.8
Accept-Encoding	gzip, deflate, br
Accept-Language	en-US,en;q=0.9
Connection	Keep-Alive

Internal Server Error x +
https://localhost:44317

An unhandled exception occurred while processing the request.

Exception: Something went wrong...

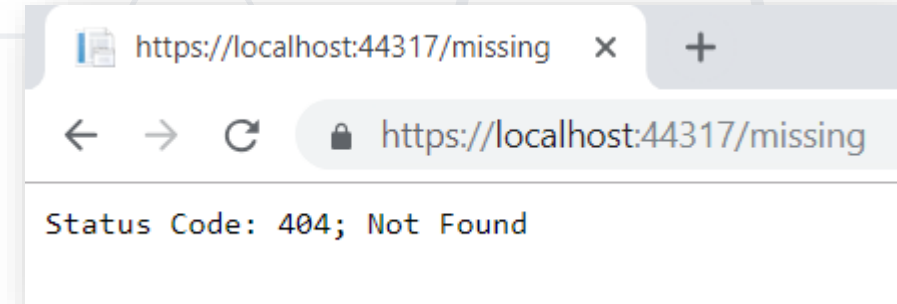
WebApplication1.Controllers.HomeController.Index() in HomeController.cs, line 31

Stack Query Cookies Headers

No cookie data.

Error Handling (Status Code Pages)

- **ASP.NET Core** apps do not provide **rich** status code pages
 - To provide such, you have to use the **Status Code Pages Middleware**
 - **`app.UseStatusCodePages()`**
- The Middleware can easily be customized
 - Supports several extension methods. For example:
 - **`app.UseStatusCodePagesWithRedirects(...)`**
 - **`app.UseStatusCodePagesWithReExecute(...)`**



Error Handling (Custom Error Handler)

- Configuring a custom exception page is done by using the **ExceptionHandlerMiddleware**

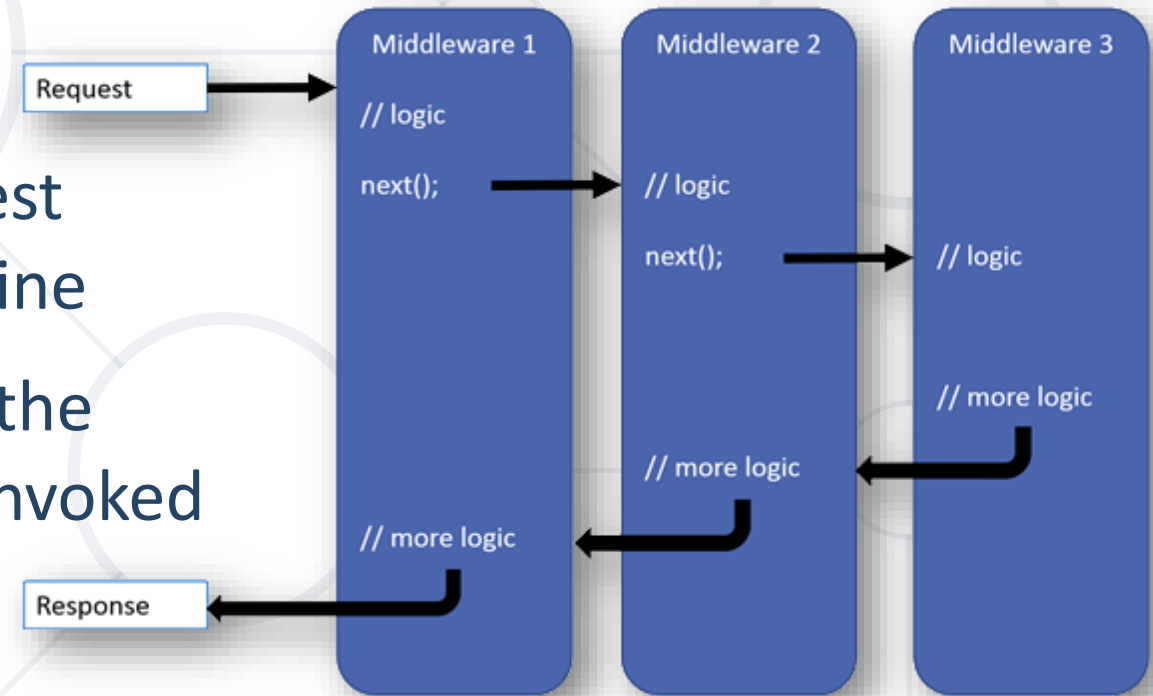
```
app.UseExceptionHandler("/Home/Error");
```

- You can then implement a **handler** for that **route**
 - It can be a **Controller Action**, a **Razor Page** or other handler

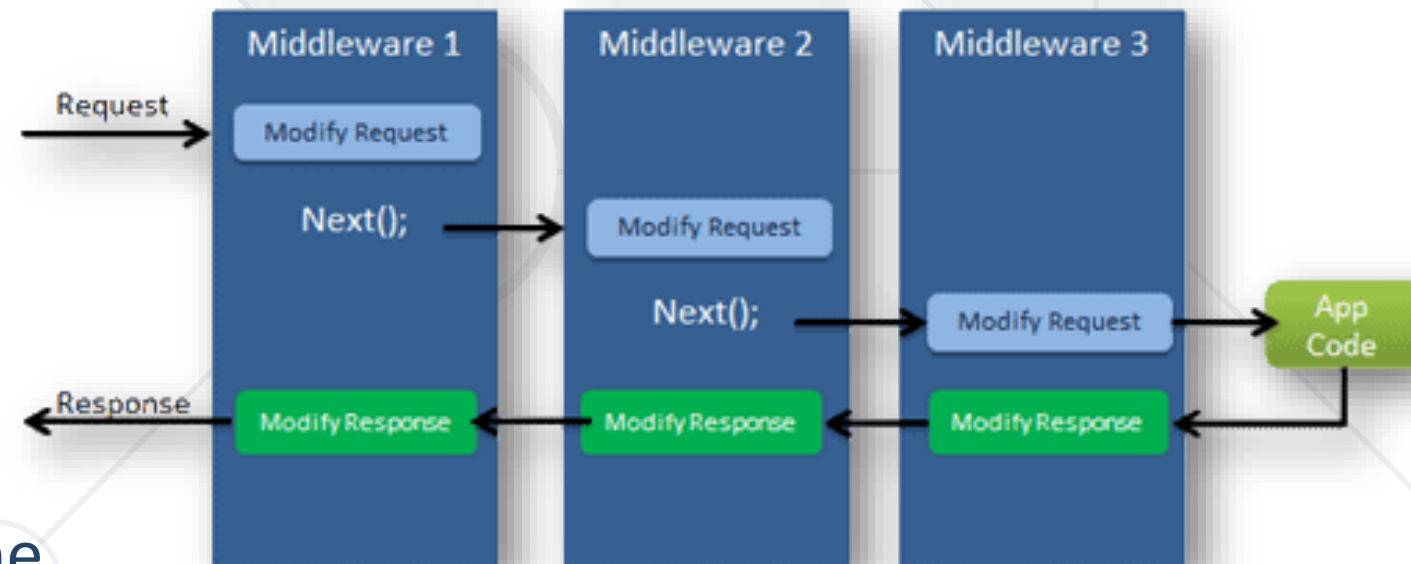


Middleware

- Middleware is a **software**, assembled into an **app pipeline**
- Each **component**
 - Handles **Requests** and **Responses**
 - Chooses whether to pass the request to the next component in the pipeline
 - May perform work **before** or **after** the next component in the pipeline is invoked
- In ASP.NET Core, **request delegates** build the **request pipeline**



- **Request delegates** handle each **HTTP request**
 - Are configured using the **extension** methods **Run()**, **Map()** and **Use()**
- **Request delegates** (also called **middleware components**) can be:
 - Specified in-line as an anonymous method (called **inline middleware**)
 - Defined in a reusable class
- Each **middleware component** should
 - Invoke the **next component** in the pipeline
 - Or **short-circuiting** the pipeline



- The **Use()** method is used to **chain** multiple **delegates** together
 - It can **short-circuit** the pipeline (if it does not invoke **next()**)
- The first **Run()** delegate terminates the pipeline
 - **Run()** is a convention
 - Some middleware expose **Run{Middleware}** methods
 - These methods run at the end of the pipeline
- The **Map()** method is used to **branch** the pipeline
 - The **request pipeline** is branched – based on the given **request path**

Creating Your Own Middleware (Inline)

- The **ASP.NET Core Request Pipeline** consists of a sequence of **Request Delegates**, called one after another
- Custom **Request Delegates** are created using the **IApplicationBuilder**

```
app.Use(async (context, next) =>
{
    // Do work that doesn't write to the Response.
    await next();
    // Do logging or other work that doesn't write to the Response.
});
//Other code below...
```

Creating Your Own Middleware (Class)

- **Request delegates** can also be defined as classes

```
public class CustomMiddleware
{
    private readonly RequestDelegate next;

    public CustomMiddleware(RequestDelegate next)
    {
        this.next = next;
    }

    // IMyService is injected into InvokeAsync
    public async Task InvokeAsync(HttpContext httpContext, IMyService svc)
    {
        svc.MyProperty = 1000;
        await this.next(httpContext);
    }
}
```

The next delegate
in the **pipeline**

Third-party **dependencies**
are injected through **DI**

Creating Your Own Middleware (Class)

- The custom **Middleware** class needs to be included into the **pipeline**

```
public static class CustomMiddlewareExtensions
{
    public static IApplicationBuilder UseCustom(this IApplicationBuilder builder)
    {
        return builder.UseMiddleware<CustomMiddleware>();
    }
}
```

- **Program.cs**

```
app.UseCustom();
```

- Some built-in middleware in ASP.NET Core are

Middleware	Import
Authentication	<code>app.UseAuthentication()</code>
Cookie Policy	<code>app.UseCookiePolicy()</code>
CORS	<code>app.UseCors()</code>
Diagnostics	<code>app.UseDevelopmentExceptionPage()</code> <code>app.UseExceptionHandler(...)</code> <code>app.UseStatusCodePages()</code>
HTTPS Redirection	<code>app.UseHttpsRedirection()</code>
HSTS	<code>app.UseHsts()</code>
Static Files	<code>app.UseStaticFiles()</code>

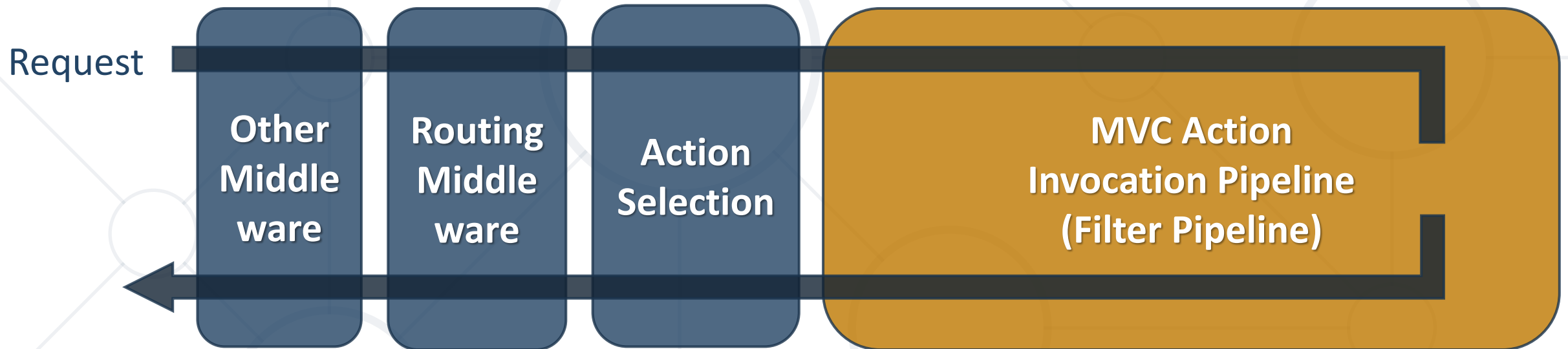
Middleware	Import
Response Caching	<code>app.UseResponseCaching()</code>
Response Compression	<code>app.UseResponseCompression()</code>
Request Localization	<code>app.UseRequestLocalization(...)</code>
Routing	<code>app.UseRouter(...)</code>
Session	<code>app.UseSession()</code>
URL Rewriting	<code>app.UseRewriter(...)</code>
WebSockets	<code>app.UseWebSockets(...)</code>
Others	<code>app.UseWelcomePage()</code>

- Many more are available on **NuGet**

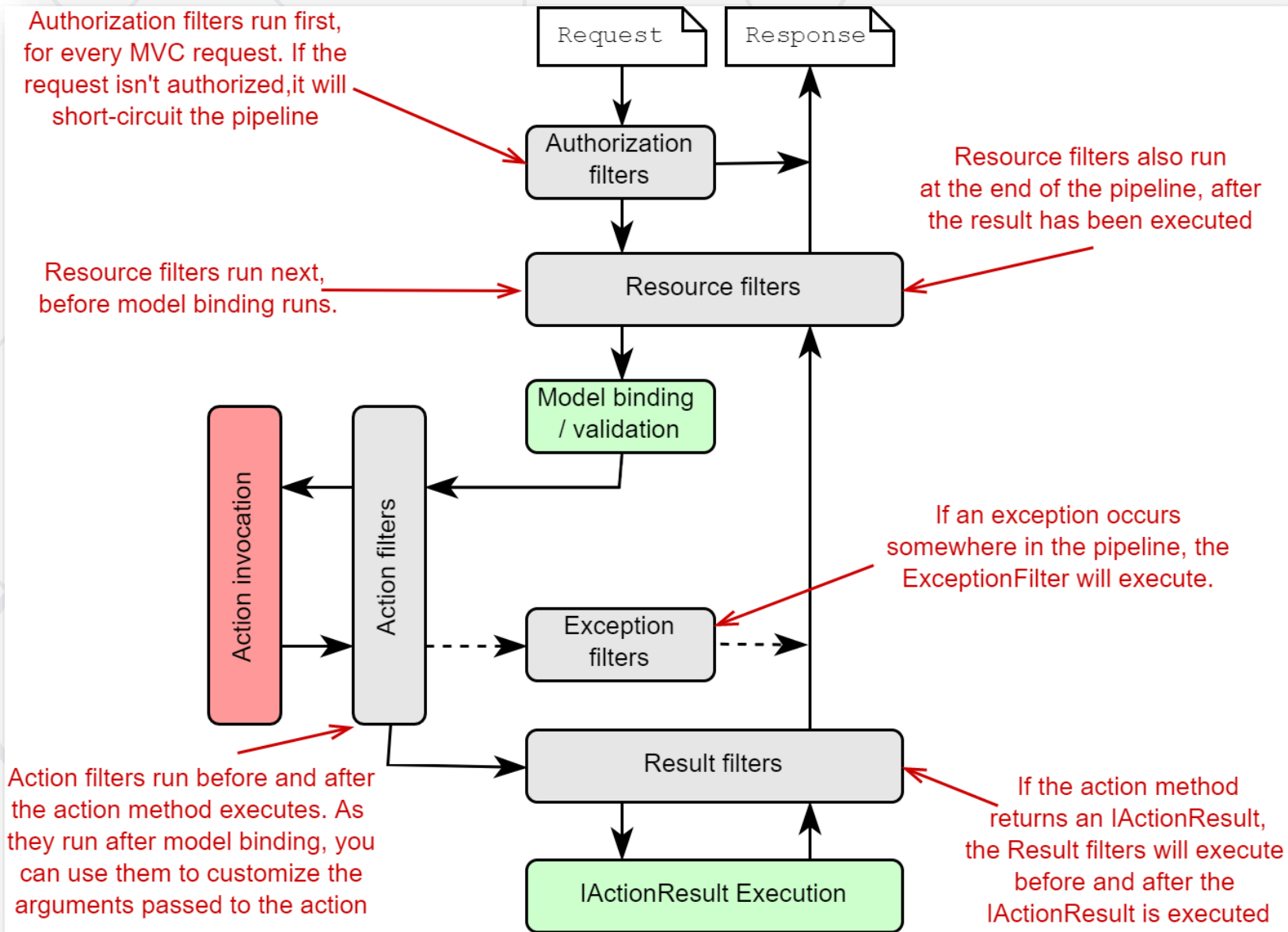


Filters

- **Filters** allow you to run code **before** or **after** specific stages in the **Request Processing Pipeline**



- **Filters** are similar but **NOT** the **same** as **Middleware**
 - Middleware operate on the level of **ASP.NET Core**
 - Filters operate only on the level of **MVC**



- There are several types of **Filters**
 - Each is executed on a **different stage** of the **Filter Pipeline**
 - There are **Authorization, Resource, Action, Exception** and **Result Filters**

Filter	Description
Authorization	Run first. Determine if the Client is authorized to access the Requested functionality
Resource	Run immediately after Authorization . Can run code before and after the rest of the pipeline
Action	Run immediately before and after an individual Action Method is invoked.
Exception	Used to apply global policies for unhandled errors that occur.
Result	Run immediately before and after execution of individual Action Results .

- **ASP.NET Core MVC Filters** can be both **synchronous** and **asynchronous**

Synchronous

```
public class SampleActionFilter : IActionFilter
{
    public void OnActionExecuting(
        ActionExecutingContext context)
    {
        // DO before the action executes
    }

    public void OnActionExecuted(
        ActionExecutedContext context)
    {
        // DO after the action executes
    }
}
```

Asynchronous

```
public class SampleAsyncActionFilter :
    IAsyncActionFilter
{
    public async Task OnActionExecutionAsync(
        ActionExecutingContext context,
        ActionExecutionDelegate next)
    {
        // DO before the action executes
        var resultContext = await next();
        // DO after the action executes
        // resultContext.Result will be set
    }
}
```

Adding Filters to the Pipeline (Global)

- **Filters** are added **globally** in the **MvcOption.Services**
 - Will be applied to all **Controllers** and **Actions**

```
builder.Services.AddMvc(options => {  
    options.Filters.Add(new SampleActionFilter()); // instant  
    options.Filters.Add(typeof(SampleActionFilter)); // by type  
    ...  
});
```

- **ASP.NET Core** also includes built-in attribute-based **Filters**

```
public class AddHeaderAttribute : ResultFilterAttribute
{
    private readonly string name;
    private readonly string value;

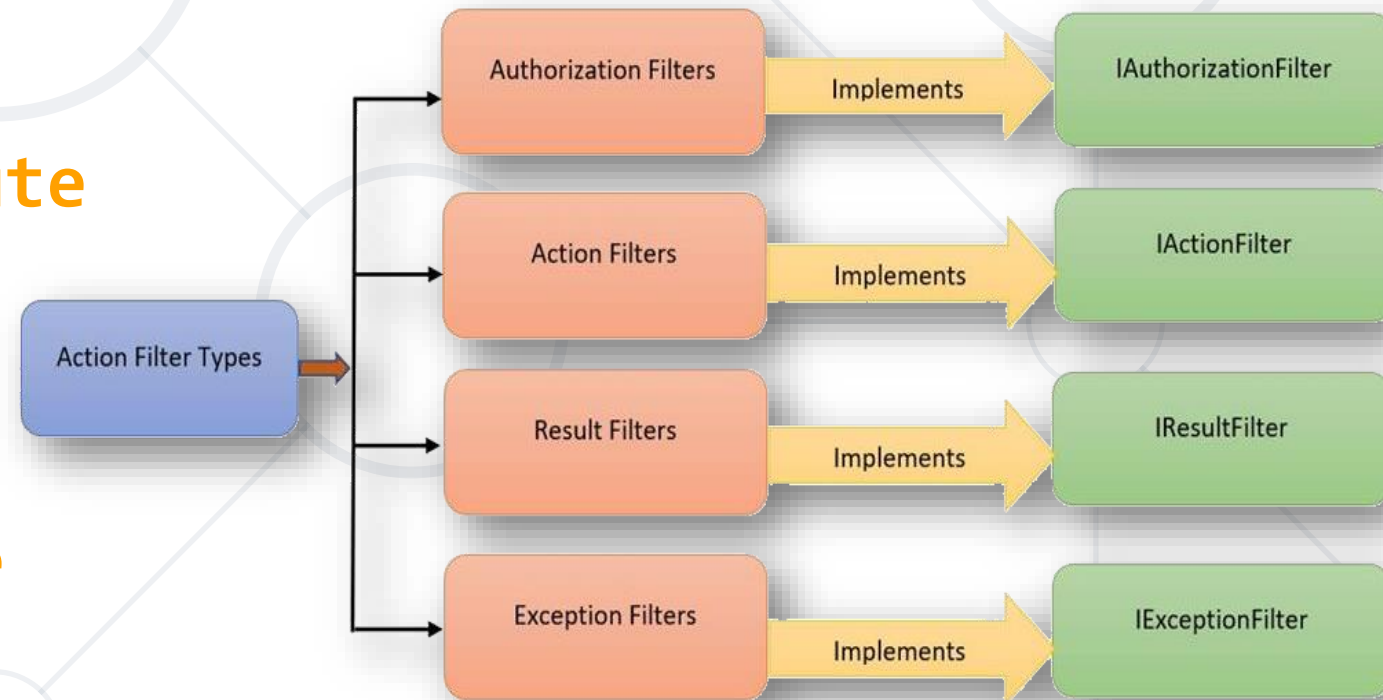
    public AddHeaderAttribute(string name, string value)
    {
        this.name = name;
        this.value = value;
    }

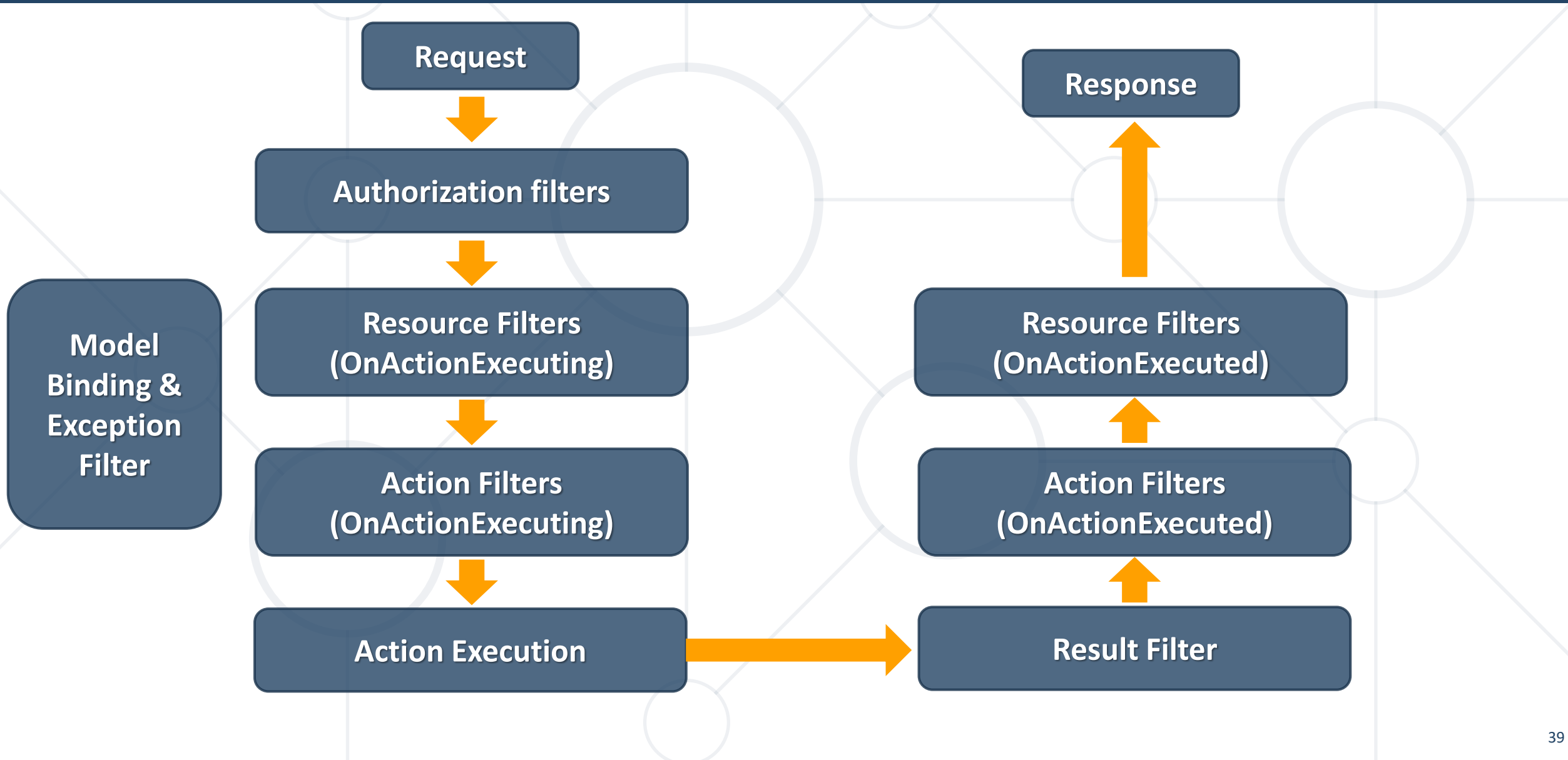
    public override void OnResultExecuting(ResultExecutingContext context)
    {
        context.HttpContext.Response.Headers.Add(this.name,
            new string[] { this.value });
        base.OnResultExecuting(context);
    }
}
```

- **Attributes** allow **Filters** to accept **arguments**
- This particular **Filter** will attach the given **Header** and its **value** to every **Result** in the **Controller**

```
[AddHeader("Author", "Steve Smith @ardalis")]  
public class SampleController : Controller  
{  
    public IActionResult Index()  
    {  
        return Content("Examine the headers using developer tools.");  
    }  
  
    public IActionResult Test()  
    {  
        return Content("Header will be present here too.");  
    }  
}
```

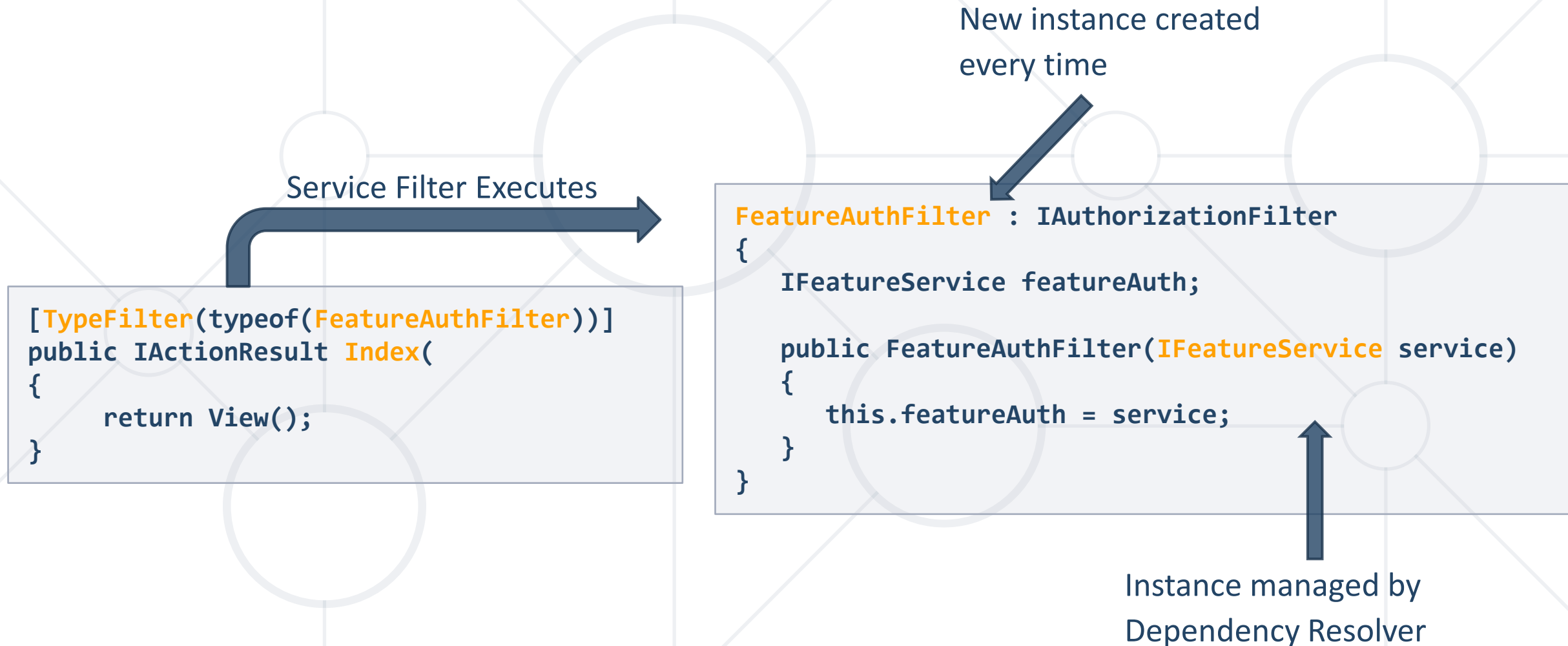
- Several of the **Filter** interfaces have corresponding **Attributes**
 - These can be used as **base classes** for custom implementation
- Filter Attributes
 - **ActionFilterAttribute**
 - **ExceptionHandlerAttribute**
 - **ResultFilterAttribute**
 - **FormatFilterAttribute**
 - **ServiceFilterAttribute**
 - **TypeFilterAttribute**





- **Filters** that are implemented as **Attributes**
 - Are added directly to **Controller** classes or **Action** methods
 - Cannot have **constructor dependencies** provided by **DI**
 - **Parameters** must be supplied where the **attributes** are applied
 - This is a limitation of how filters **attributes** work
- There are several approaches to include **DI** in **Filter Attributes**
 - **ServiceFilterAttribute**
 - **TypeFilterAttribute**

- Service filter implementation types are registered in **DI**
 - **ServiceFilterAttribute** retrieves an instance of the filter from **DI**
 - Used only for **Filters** that are registered as **Services**
- **TypeFilterAttribute** is similar to **ServiceFilterAttribute**
 - The type is not resolved directly from the DI container
 - Type is instantiated using **ObjectFactory**
- There are ways to control the **reusability** of the instances
 - There is no guarantee that a **single instance** will be created



```
builder.Services.AddSingleton<IFeatureService, FeatureService>();  
builder.Services.AddSingleton<FeatureFilter>();
```

Instance managed by
Dependency Resolver

Service Filter Executes

```
[TypeFilter(typeof(FeatureAuthFilter))]  
public IActionResult Index()  
{  
    return View();  
}
```

```
FeatureAuthFilter : IAuthorizationFilter  
{  
    IFeatureService featureAuth;  
  
    public FeatureAuthFilter(IFeatureService service)  
    {  
        this.featureAuth = service;  
    }  
}
```

Instance managed by
Dependency Resolver

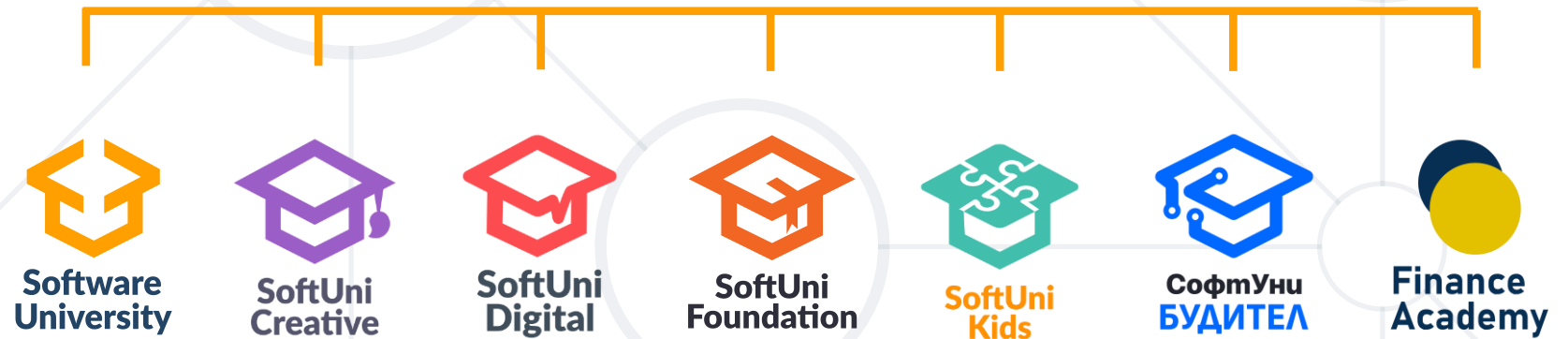
- **Application Flow**
 - Application Environments
 - Request Lifecycle
 - Error Handling
- **Middleware** == software, assembled into an app pipeline
- **Filters** == allow to run code before or after specific stages in the Request Processing Pipeline



SoftUni Diamond Partners



Questions?



- Software University – High-Quality Education, Profession and Job for Software Developers
 - softuni.bg, softuni.org
- Software University Foundation
 - softuni.foundation
- Software University @ Facebook
 - facebook.com/SoftwareUniversity



- This course (slides, examples, demos, exercises, homework, documents, videos and other assets) is **copyrighted content**
- Unauthorized copy, reproduction or use is illegal
- © SoftUni – <https://softuni.org>
- © Software University – <https://softuni.bg>

