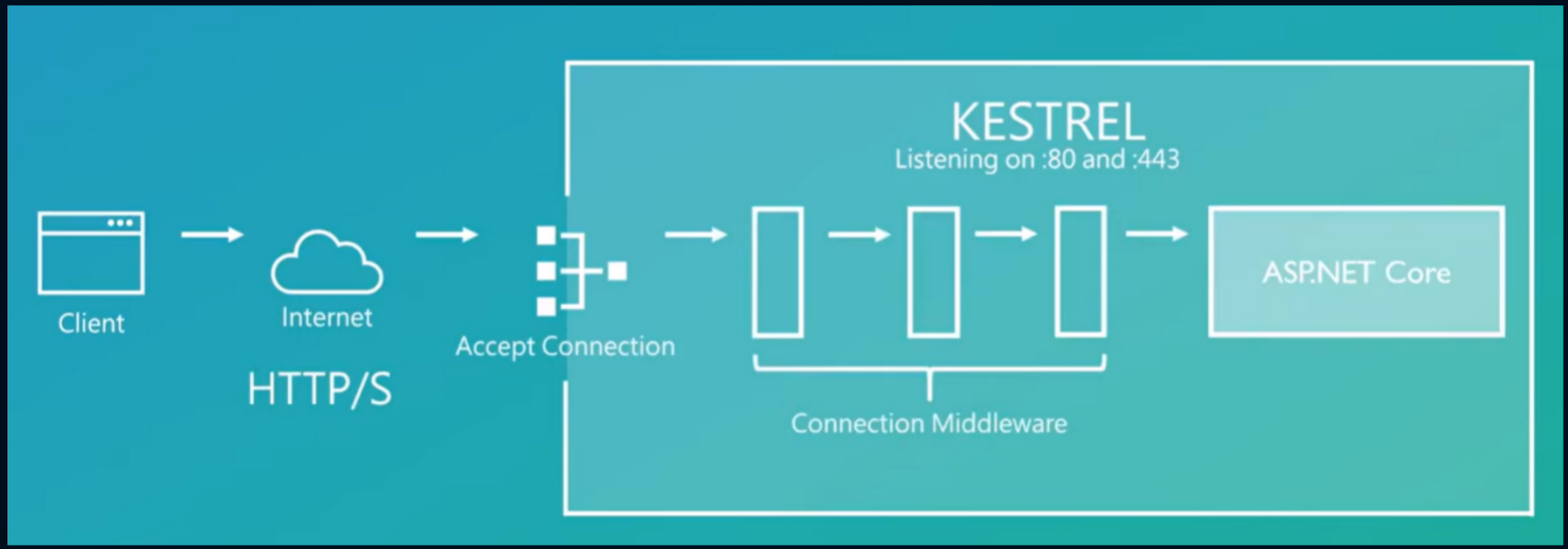
# Middlewares

In client-server applications, middlewares play a crucial role in handling communication and data processing between the client and server components. Middlewares act as intermediaries that intercept and process requests and responses, allowing developers to add custom logic, security checks, data transformation, and other functionalities to the communication flow. It is important to mention that middlewares can be present at client -side or server-side




Client-server using Kestrel

# ASP Net Core Middlewares

**Components that form a pipeline for handling HTTP requests and responses in an ASP.NET Core application. Each middleware in the pipeline performs a specific operation on the request and/or response before passing it along to the next middleware in the pipeline. Middlewares enable developers to add custom logic, cross-cutting concerns, and additional functionality to the request/response processing flow. Middlewares serve diferent ends here are a few:**

### Request Middleware:

Request middlewares are executed in sequence when an HTTP request is received by the application.
Each middleware can inspect and modify the incoming request.
E.g. Auth, Routing, Custom

### Response Middleware:

Executed in reverse order when the HTTP response is generated by the application.
e.g. Exception, Compression, Custom

### Chaining and Order:

Ordered in the pipeline based on their registration order.
Each one can decide whether to pass the request to the next middleware or short-circuit the pipeline.

### Reusability:

Middlewares can be developed and packaged as reusable components, making it easy to apply common functionality across multiple applications.

### Customization and Extension:

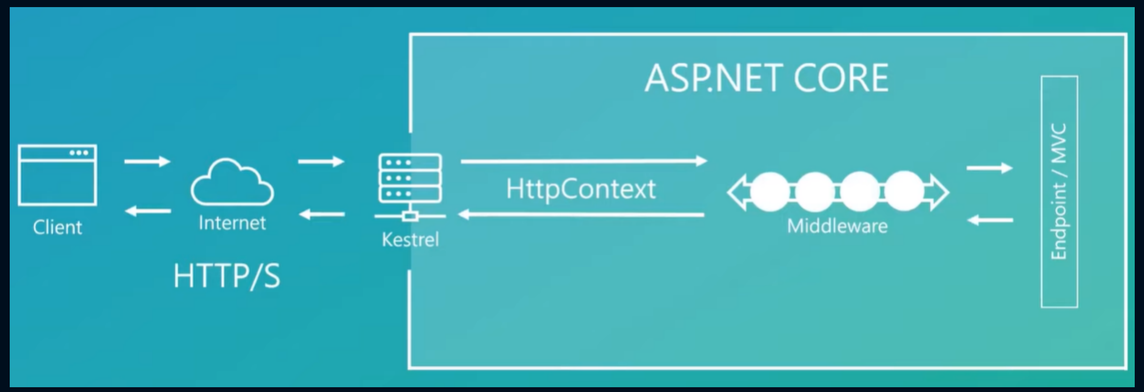
ASP.NET Core provides a set of built-in middlewares, but developers can also create custom middlewares to meet specific requirements.

### Asynchronous Support:

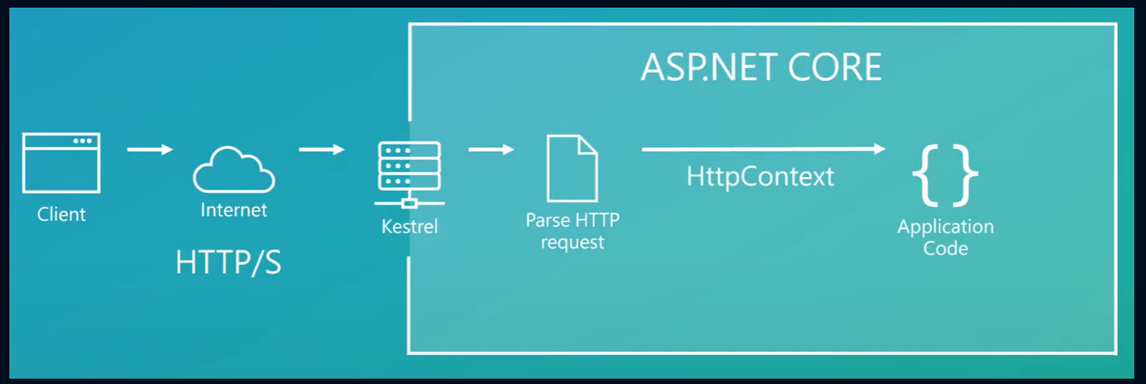
Middlewares can be implemented as asynchronous to leverage the benefits of asynchronous programming and improve application performance.

# ASP Net Core Middlewares

HttpContext is a crucial component in ASP.NET Core that represents the current HTTP request and response being processed by the application. It encapsulates all the relevant information related to the incoming request and outgoing response, allowing middlewares to inspect, modify, and control the request-response flow.



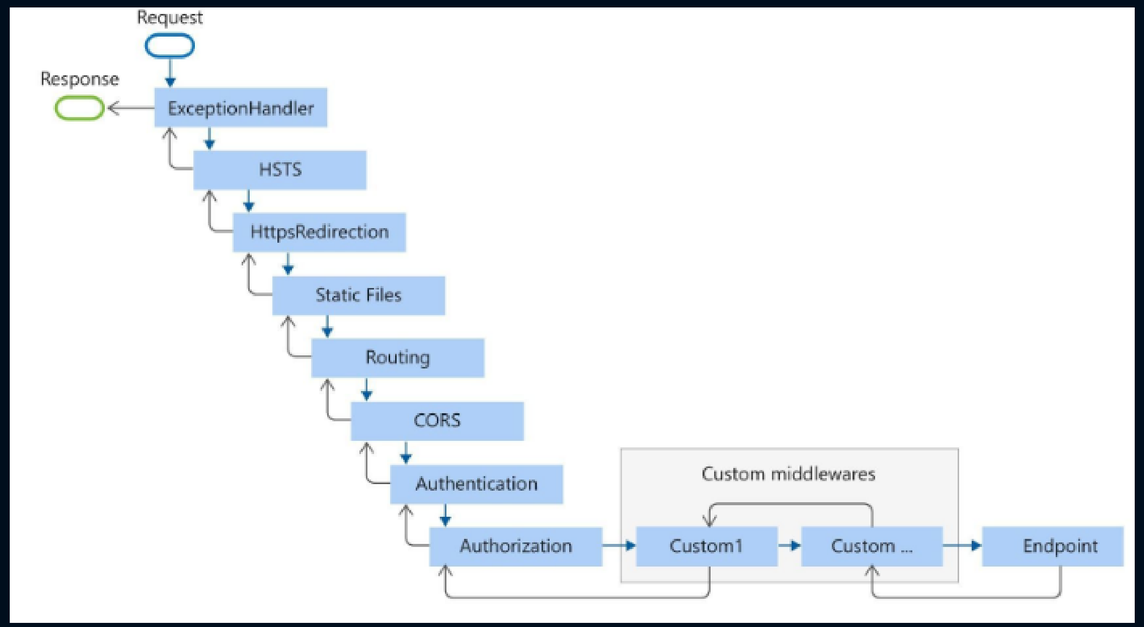

ASP.Net Core middlewares




Overview of ASP.Net Core structure

# Custom middleware

Custom middlewares in ASP.NET Core are components that developers create to extend or modify the behavior of the request/response processing pipeline. These middlewares allow you to add specific functionality or handle cross-cutting concerns in a modular and reusable manner.




Middleware pipeline order

```
public class UrlChangerMiddleware
{
  private readonly RequestDelegate _next;

  public UrlChangerMiddleware(RequestDelegate next)
  {
    _next = next;
  }

  public async Task InvokeAsync(HttpContext context)
  {
    // Call the next delegate/middleware in the pipeline.
    await _next(context);
  }
}
```

**External Middleware Class**

**Using lambda's is ok if your replacing the entire pipeline, for reusable middleware assemblies however, a minimum class shown above should be used.**

Custom middleware implementation, external class

# Global Exception Handling

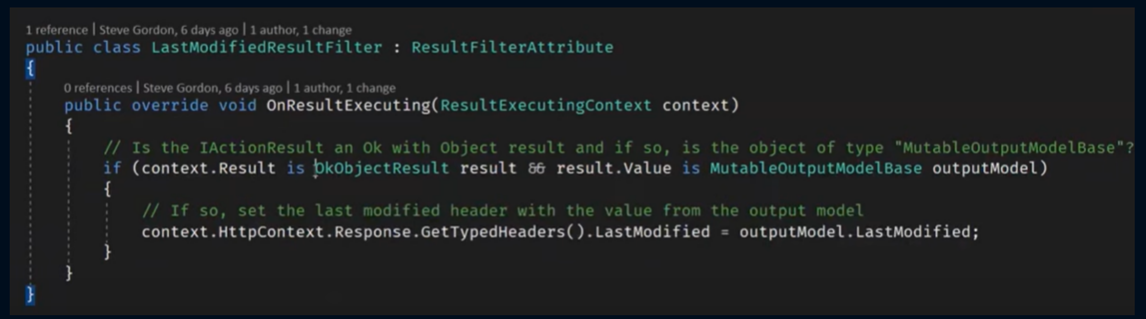
The UseExceptionHandler middleware can be used in the Configure method of the Startup class to handle exceptions and return an appropriate error response.

```
1 reference
public static WebApplication ConfigurePipeline(this WebApplication app)
{
    if (app.Environment.IsDevelopment())
    {
        app.UseDeveloperExceptionPage();
    }
    else
    {
        app.UseExceptionHandler(appBuilder =>
        {
            appBuilder.Run(async context =>
            {
                context.Response.StatusCode = 500;
                await context.Response.WriteAsync(
                    "An unexpected fault happened. Try again later.");
            });
        });
    }
```

Implementation of a simple exception handler

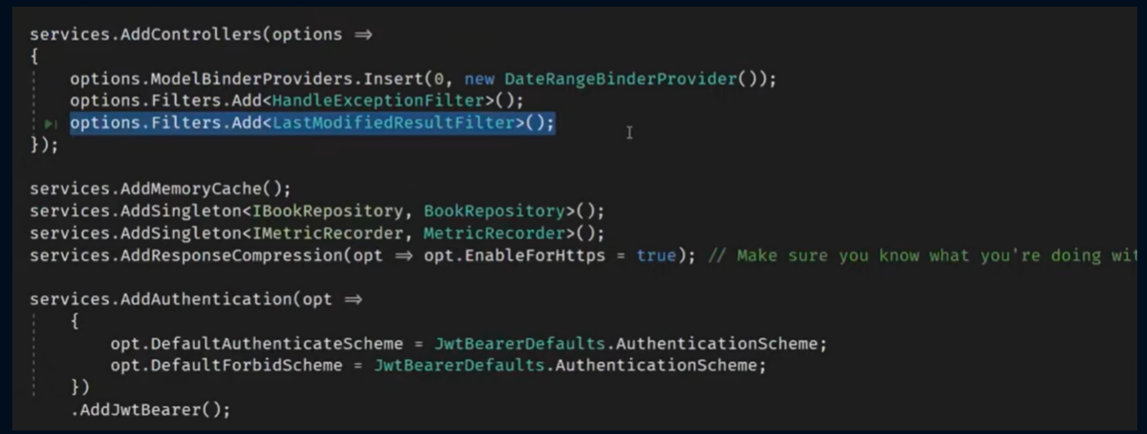
# Filters

Attributes or classes that can be applied to controllers or actions to modify the behavior of the action execution or the result returned by the action.
They can be used for tasks like authorization, logging, caching, exception handling, and more.
Filters execute before and after the action method is executed and provide hooks to perform pre- and post-processing tasks.
There are several types of filters in ASP.NET Core, including authorization filters, resource filters, action filters, and exception filters.



```
1 reference | Steve Gordon, 6 days ago | 1 author, 1 change
public class LastModifiedResultFilter : ResultFilterAttribute
{
    0 references | Steve Gordon, 6 days ago | 1 author, 1 change
    public override void OnResultExecuting(ResultExecutingContext context)
    {
        // Is the IActionResult an Ok with Object result and if so, is the object of type "MutableOutputModelBase"?
        if (context.Result is OkObjectResult result && result.Value is MutableOutputModelBase outputModel)
        {
            // If so, set the last modified header with the value from the output model
            context.HttpContext.Response.GetTypedHeaders().LastModified = outputModel.LastModified;
        }
    }
}
```

Custom filter class

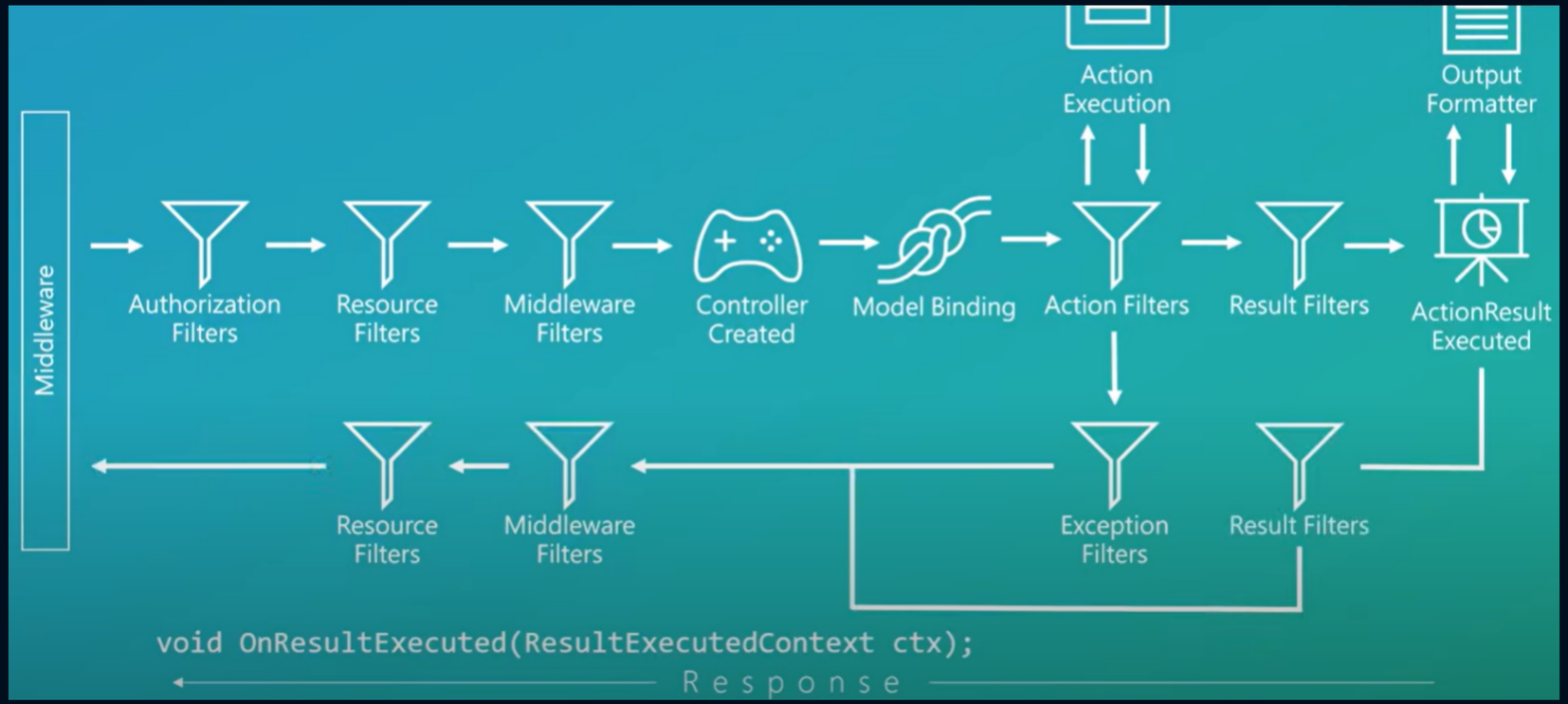


```
services.AddControllers(options =>
{
    options.ModelBinderProviders.Insert(0, new DateRangeBinderProvider());
    options.Filters.Add<HandleExceptionFilter>();
    options.Filters.Add<LastModifiedResultFilter>();
});

services.AddMemoryCache();
services.AddSingleton<IBookRepository, BookRepository>();
services.AddSingleton<IMetricRecorder, MetricRecorder>();
services.AddResponseCompression(opt => opt.EnableForHttps = true); // Make sure you know what you're doing wit

services.AddAuthentication(opt =>
    {
        opt.DefaultAuthenticateScheme = JwtBearerDefaults.AuthenticationScheme;
        opt.DefaultForbidScheme = JwtBearerDefaults.AuthenticationScheme;
    })
    .AddJwtBearer();
```

Register Custom filter

# Filters

Both middlewares and filters are powerful features in ASP.NET Core that enable developers to implement various functionalities in a modular and customizable way. Depending on the scenario, you can choose between using middlewares, filters, or a combination of both to achieve the desired behavior in your web application.




Filter pipeline

# Global Exception Handling via Filters

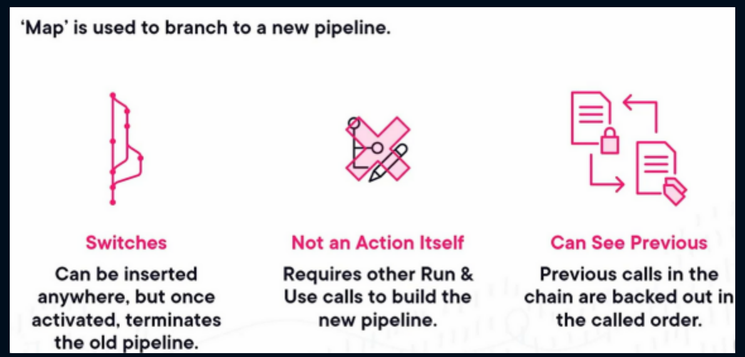
Filters in ASP.NET Core are components that allow developers to implement cross-cutting concerns in a flexible and reusable way. They are designed to intercept and modify the behavior of HTTP requests and responses, allowing you to implement common functionalities across multiple actions or controllers in your application.

```
public class HandleExceptionFilter : IExceptionFilter
{
    private readonly IMetricRecorder _metricRecorder;
    private readonly IHostEnvironment _hostEnvironment;

    0 references | Steve Gordon, 6 days ago | 1 author, 1 change
    public HandleExceptionFilter(IMetricRecorder metricRecorder, IHostEnvironment hostEnvironment)
    {
        _metricRecorder = metricRecorder;
        _hostEnvironment = hostEnvironment;
    }

    0 references | Steve Gordon, 3 hours ago | 1 author, 2 changes
    public void OnException(ExceptionContext context)
    {
        // Record a metric to our monitoring system so we can alert on unhandled exceptions
        _metricRecorder.RecordException(context.Exception);

        // If we're in production, send the custom formatted content with a general error
        // If we're not in production, send the custom formatted content with the exception message
        context.Result = _hostEnvironment.IsProduction()
            ? new JsonResult(new ApiError("An unhandled error occurred.")) { StatusCode = 500}
            : new JsonResult(new ApiError(context.Exception.Message)) {StatusCode = 500};

        context.ExceptionHandled = true;
    }
```

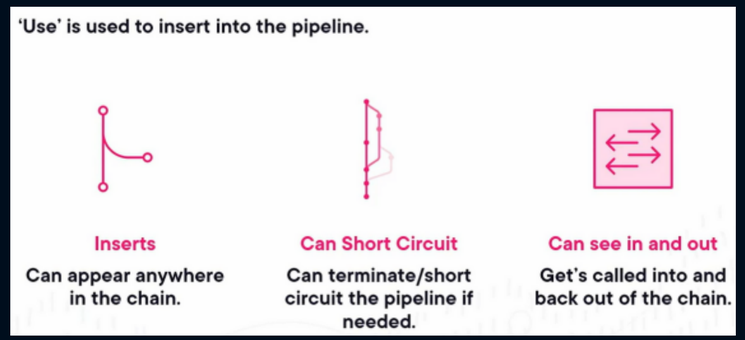
Global exception handling via Filters

# Middleware methods

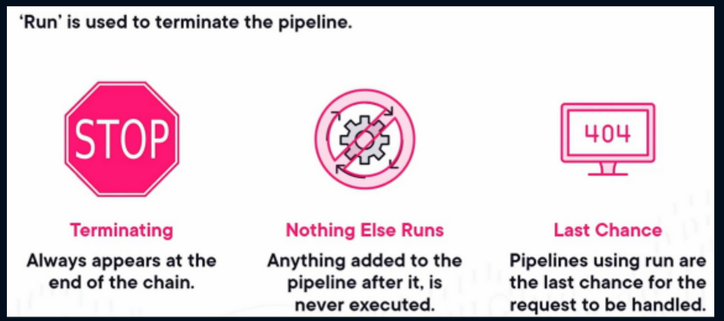
In ASP.NET Core, custom middlewares can be added to the middleware pipeline using several extension methods provided by the IApplicationBuilder interface. Each method has a different purpose and allows you to insert your custom middleware at different stages of the request/response processing pipeline.




## Map method




## Use method




## Run method

# Middleware methods

These methods provide different ways to add your custom middleware to the pipeline, giving you the flexibility to control the request/response processing based on specific conditions or URL patterns.

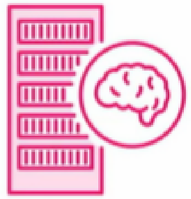
'MapWhen' is used to branch to a new pipeline on a conditional basis.

**Switches**

Like 'Map' can be inserted anywhere, but terminates old pipeline.

**Not an Action Itself**

Like 'Map' requires other Run & Use calls to build a new pipeline.

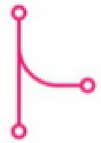


**ASP.Net Controlled**

Like 'UseWhen' the decision is made by ASP.Net NOT user code.

## Map when method

'UseWhen' is used to insert into the pipeline on a conditional basis.

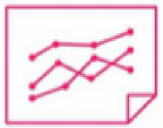


**Inserts**

Like 'Use' can appear anywhere in the chain.

**ASP.Net decides**

Unlike 'Use' this is not called if the criteria doesn't match.



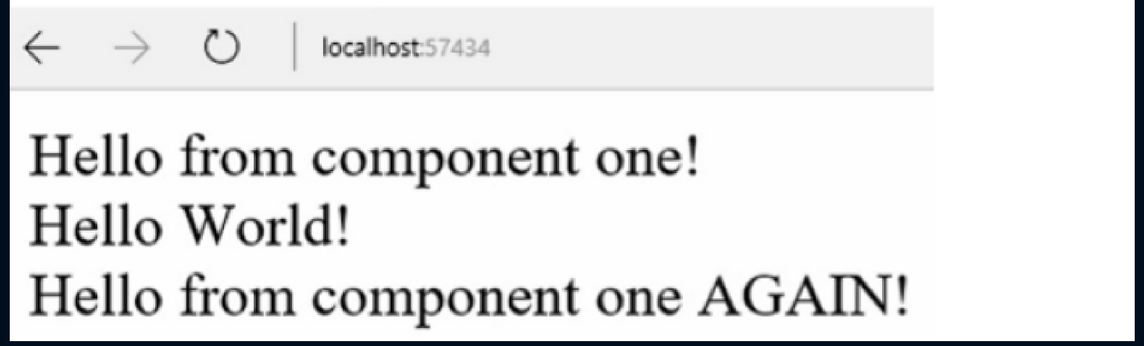
**Is Performant**

ASP.Net does the hard work of making the decision NOT user code.

## Use when method

# Middleware methods

```
app.Use(async (context, next) =>
{
    await context.Response.WriteAsync("Hello from component one!");
    await next.Invoke();
    await context.Response.WriteAsync("Hello from component one AGAIN
});

app.Run(async (context) =>
{
    await context.Response.WriteAsync("Hello World!");
});
```




Examples of Use and Run

```
app.Use(async (context, next) =>
{
    await context.Response.WriteAsync("Hello from component or
    await next.Invoke();
    await context.Response.WriteAsync("Hello from component or
});

app.Map("/mymapbranch", (appBuilder) =>
{
    appBuilder.Run(async (context) =>
    {
        await context.Response.WriteAsync("Greetings from my
    });
});

app.Run(async (context) =>
{
    await context.Response.WriteAsync("Hello World!<br/>");
});
```

```
app.MapWhen(context => context.Request.Query.ContainsKey("querybra
{
    appBuilder.Run(async (context) =>
    {
        await context.Response.WriteAsync("You have arrived at you
    });
});
```

Examples of Map and MapWhen

# Demo

**We will check the following:**

Check middlewares

You can check the following Repository for some examples:

C# fundamentals

You can check the following Repository(ies) for some examples:

AspNetCoreAnatomySamples

# Homework

**Exercise/ Homework**

This is a continuation of the previous exercise.

Implement the following Middlewares:

- Global Exception Handler Middleware
- Response Compression using gzip