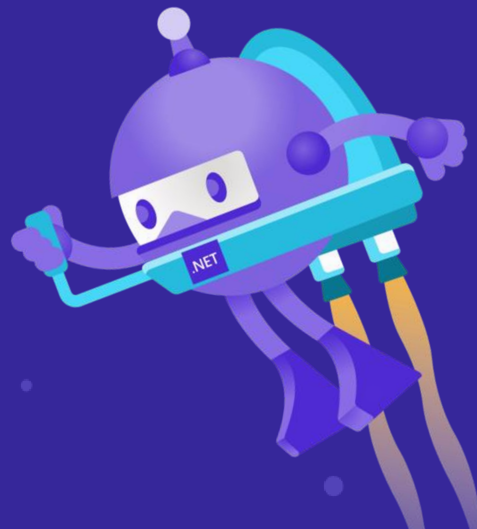
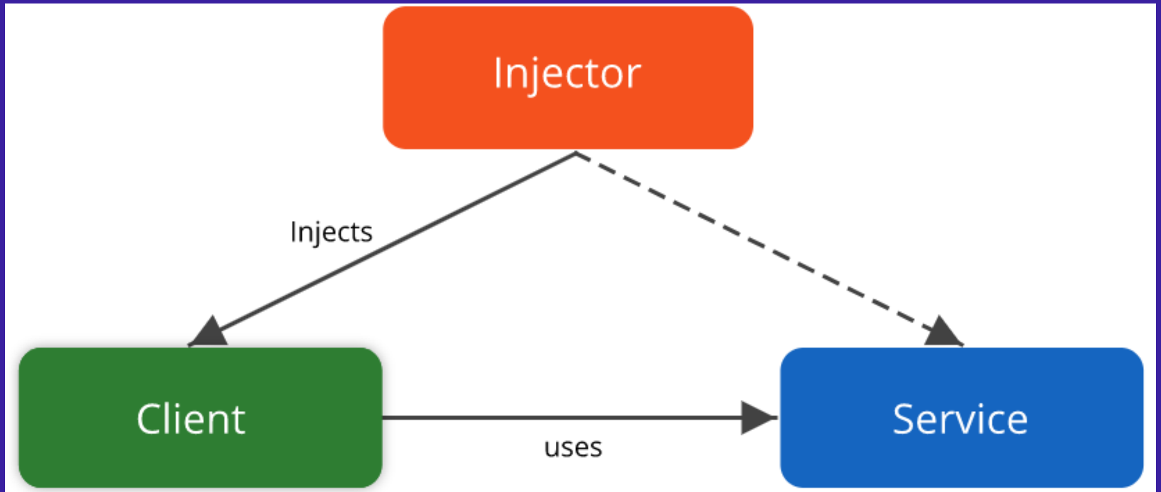
Day 9 : ASP.NET Core Web API – DI, Middleware, Routing



What is Dependency Injection(DI)

- Is a design pattern
- Allows the creation of dependent objects outside of a class and provides those objects to a class through different ways
- Involves three components



Dependency Injection(DI)

- ASP.NET Core is designed from scratch to support DI
- Come with a DI container out of the box
- Dependencies managed by the container are called ***Services***
- Two types of Services are there
  - Framework Services
  - Application Services

Configuring DI

- Install the framework NuGet package

  ```
  Install-Package Microsoft.Extensions.DependencyInjection
  ```

- Register your dependencies in Startup.cs

  ```
  public void ConfigureServices(IServiceCollection services)
  {
      services.AddScoped<ISecurityService, SecurityService>();
      services.AddSingleton<ICachingService, CachingService>();
   }
  ```

- Usage

  ```
  private readonly ICachingService _cachingService;

  public DefaultController(ICachingService cachingService)
      {
          _cachingService = cachingService;
      }
  ```

DI – Service Lifetimes

- DI Container allows you to control the lifetime of registered services
- Three types of lifetimes are available
    - Singleton – Creates only one instance
    - Transient - Creates an instance each time it is requested from the container
    - Scoped – Instances are created once per client request(HTTP Request)

DI – Accessing Service instance

```
public class Startup
{
    // ...code ...

        public void ConfigureServices(IServiceCollection services)
        {
            services.AddControllers();

            services.AddScoped<Interfaces.IOrderSender, HttpOrderSender>();
            services.AddScoped<Interfaces.IOrderManager, OrderManager>();
        }

    // ...code ...

}
```

Built-in DI Container - Limitations

- Features not supported
  - Property Injection
  - Injection based on name
  - Child Containers
  - Custom lifetime management
  - Convention-based registration

Built-in DI Container – Third Party Support

- Autofac
- Dryloc
- Grace
- LightInject
- Lamar
- Stashbox
- Unity

DI – Using Third Party Providers

```
public IServiceProvider ConfigureServices(IServiceCollection services)
{
    // Add framework services.
    // Configure third-party DI container
    // return container-specific IServiceProvider implementation
}
```

```
public class Startup {
  public IServiceProvider ConfigureServices(IServiceCollection services) {
    // setup the Autofac container
    var builder = new ContainerBuilder();
    builder.Populate(services);
    builder.RegisterType<ArticleService>().As<IArticleService>();
    var container = builder.Build();
    // return the IServiceProvider implementation
    return new AutofacServiceProvider(container);
  }
```

Middleware

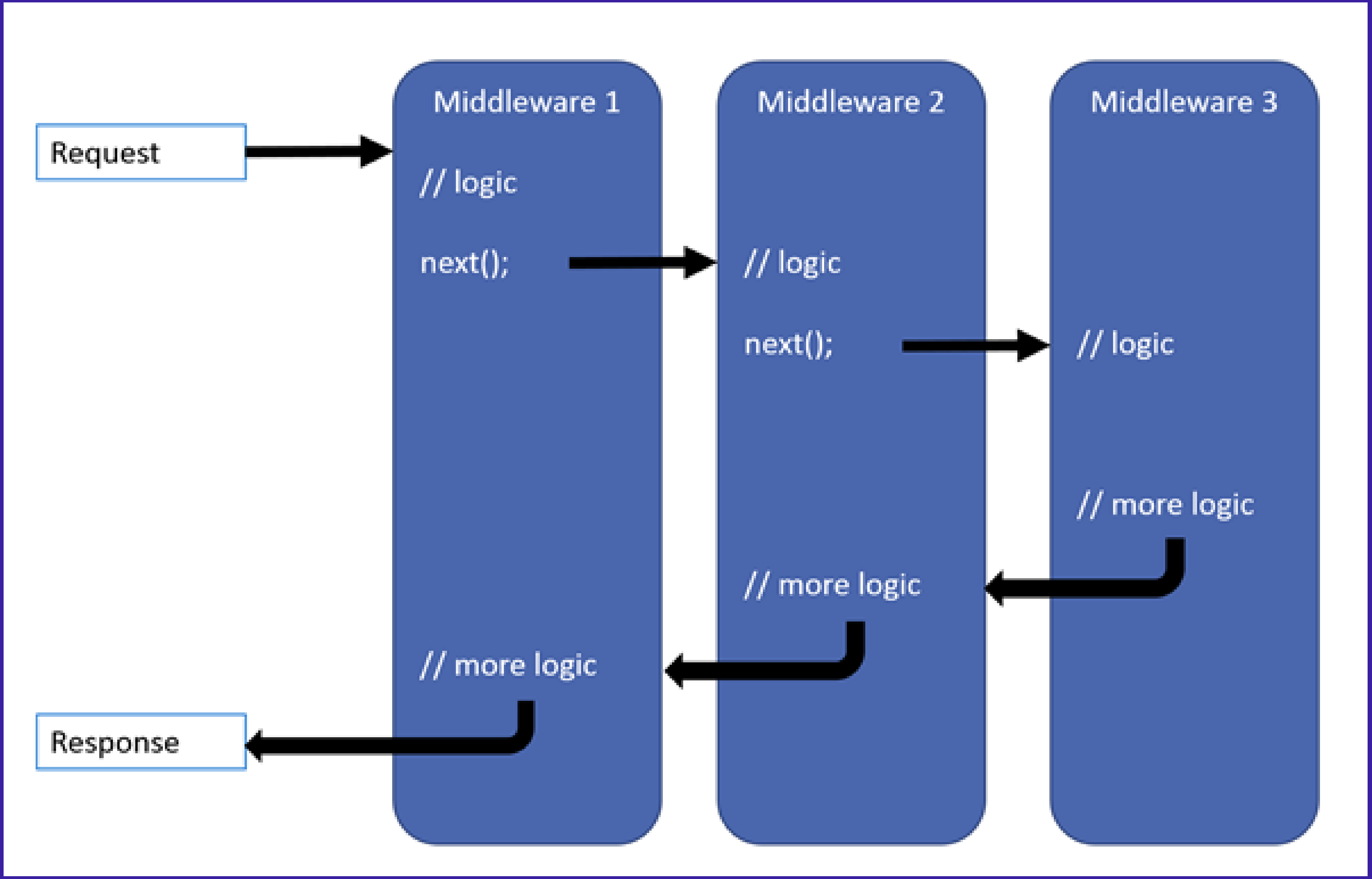
- Is a piece of logic or code, that's assembled into an app pipeline to handle requests and responses.

- Each component:
  - Chooses whether to pass the request to the next component in the pipeline.
  - Can perform work before and after the next component in the pipeline.

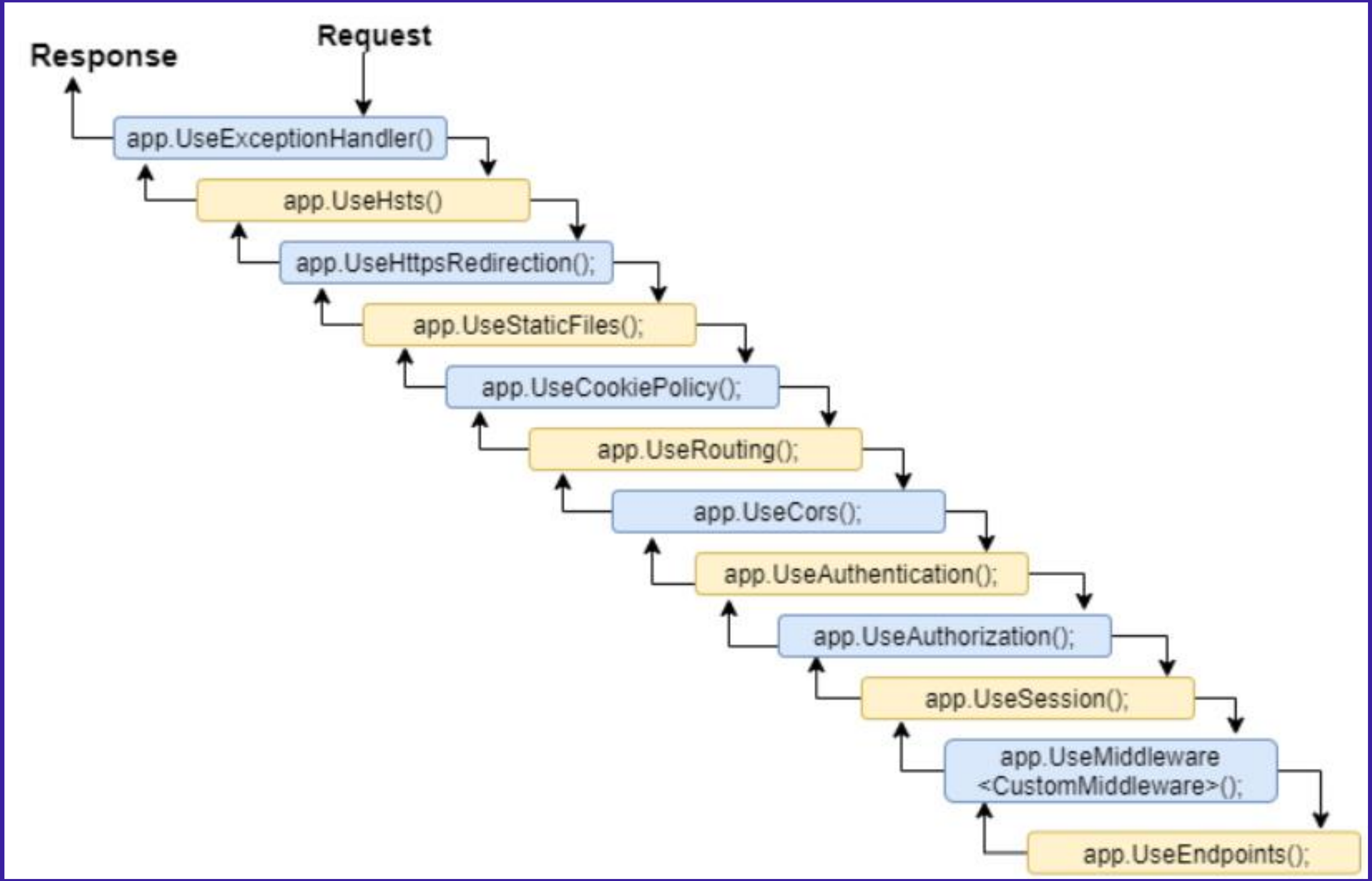
- Each component in the request pipeline is responsible for invoking the next component in the pipeline or short-circuiting the pipeline

- When a middleware short-circuits, it's called a terminal middleware because it prevents further middleware from processing the request.

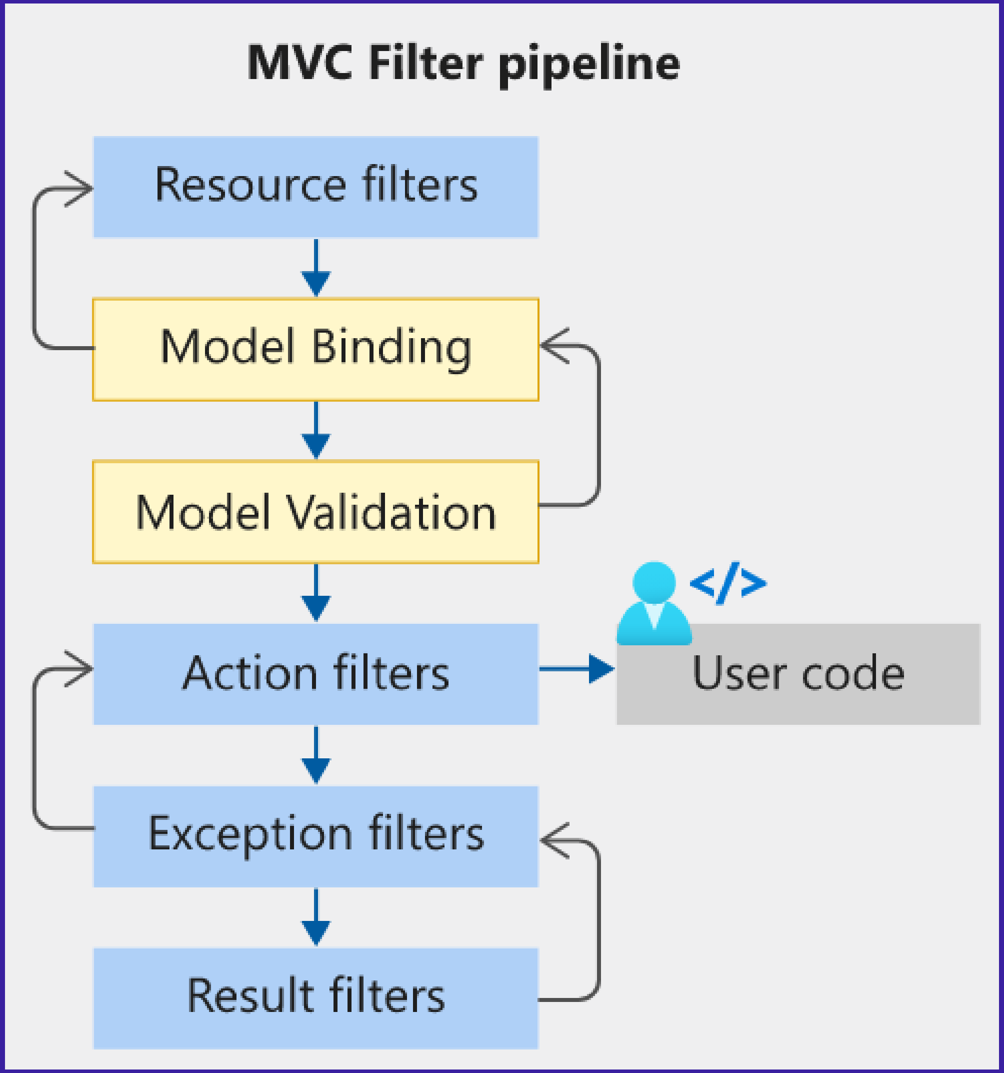
Middleware


Middleware 1
Middleware 2
Middleware 3
Request
// logic
next();
// logic
next();
// logic
// more logic
// more logic
// more logic
Response

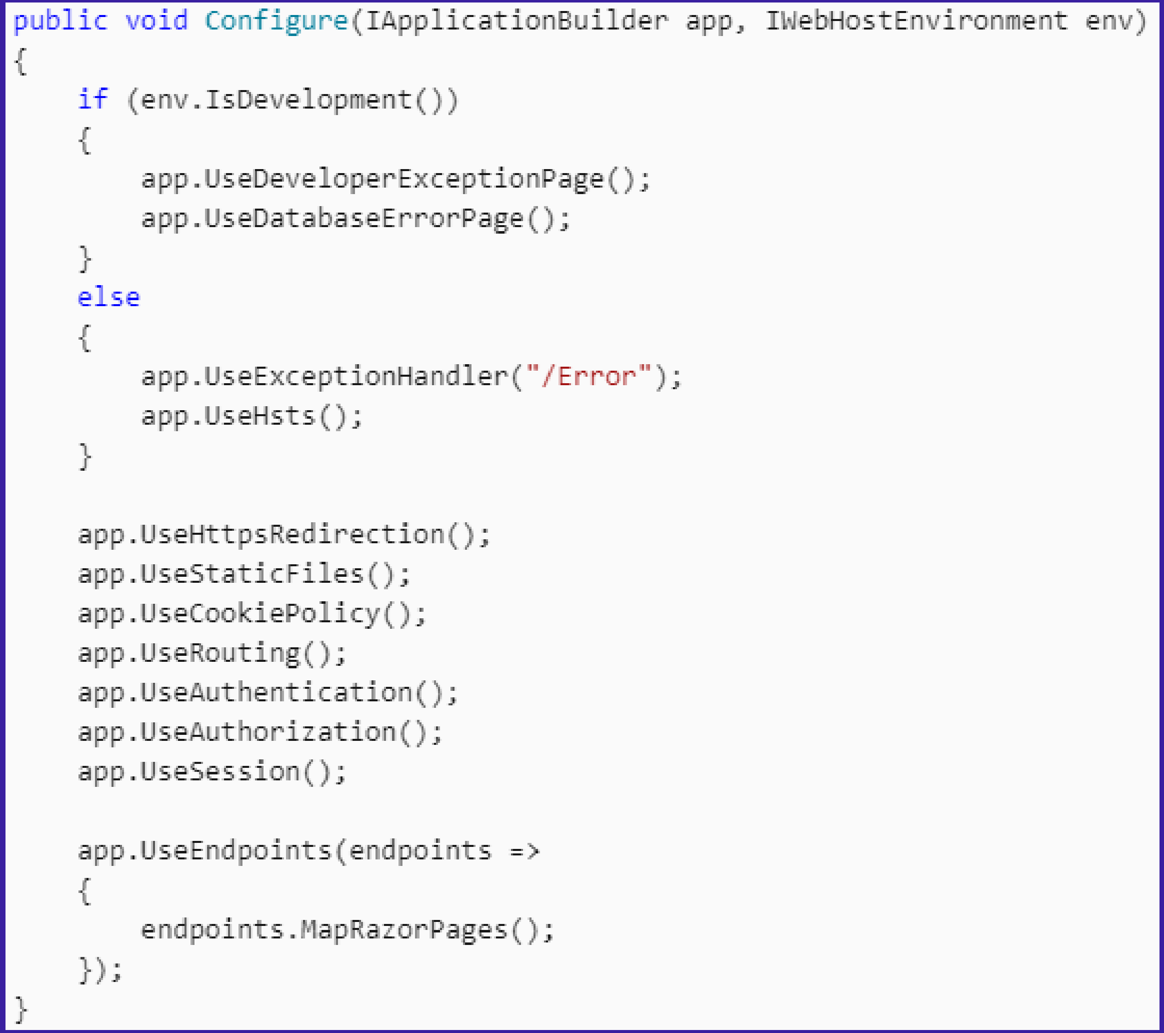
Middleware – Execution Order



Middleware – Execution Order



Middleware - Configuration



```
public void Configure(IApplicationBuilder app, IWebHostEnvironment env)
{
    if (env.IsDevelopment())
    {
        app.UseDeveloperExceptionPage();
        app.UseDatabaseErrorPage();
    }
    else
    {
        app.UseExceptionHandler("/Error");
        app.UseHsts();
    }

    app.UseHttpsRedirection();
    app.UseStaticFiles();
    app.UseCookiePolicy();
    app.UseRouting();
    app.UseAuthentication();
    app.UseAuthorization();
    app.UseSession();

    app.UseEndpoints(endpoints =>
    {
        endpoints.MapRazorPages();
    });
}
```

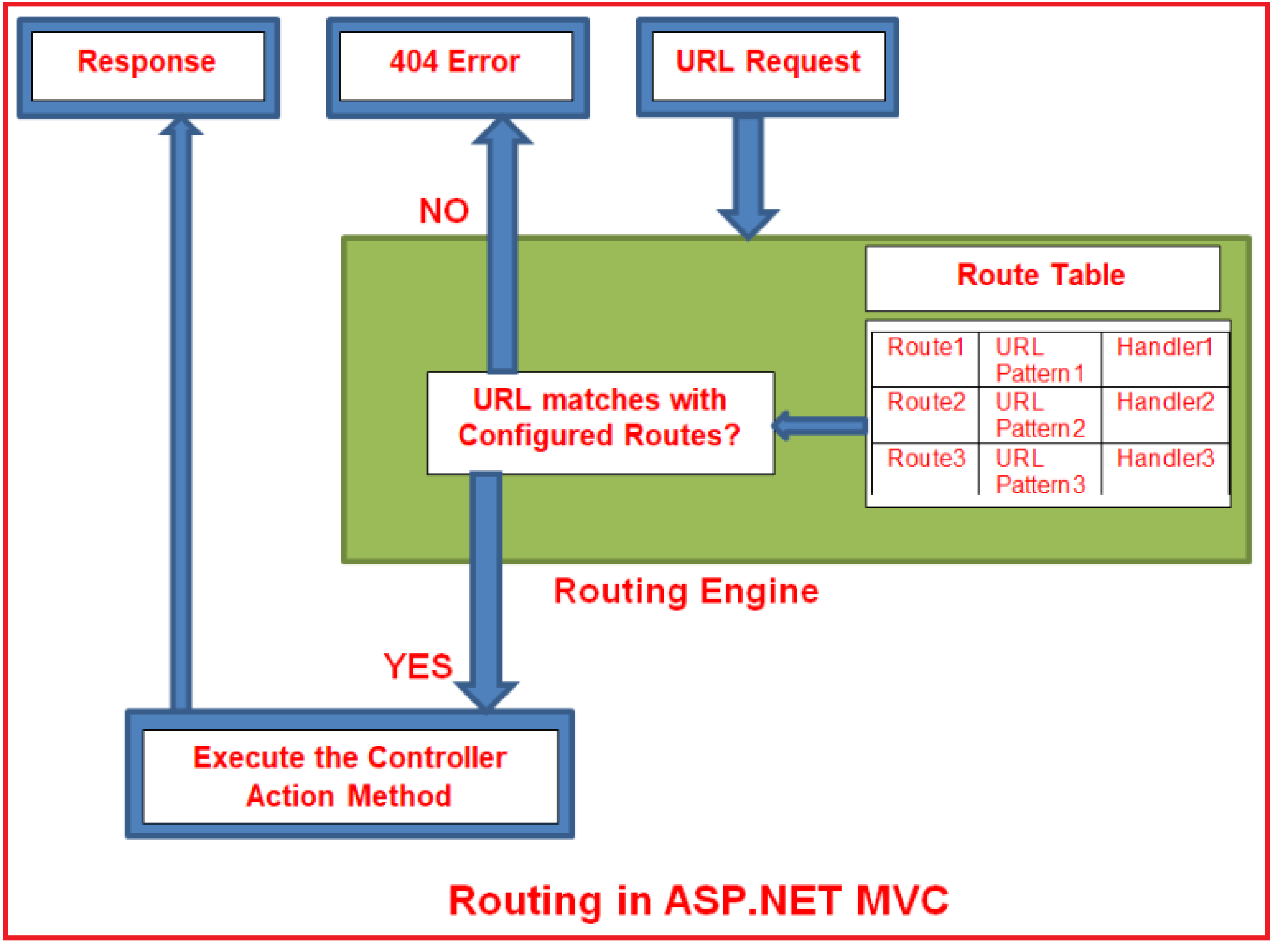
Map, Run & Use methods

- app.Use()
  - used to allow the request delegate to pass the request to the next middleware in the pipeline.
- app.Map()
  - branch the request pipeline with the mentioned URL
- app.Run()
  - will be used to end the pipeline registrations and acts as a final step

Routing

- Routing is responsible for matching incoming HTTP requests to the endpoints
- Endpoints are application’s unit of request handling code
    - Are defined in the app and configured when app starts
    - Matching process extracts values from the URL and provides it for the request processing
- Uses two middlewares, UseRouting & UseEndpoints
    - UseRouting – adds route matching to the middleware pipeline
    - UseEndpoints – adds the endpoint execution to the middleware pipeline

Routing - Flow



Response

404 Error

URL Request

NO

Route Table

Route1	URL Pattern1	Handler1
Route2	URL Pattern2	Handler2
Route3	URL Pattern3	Handler3

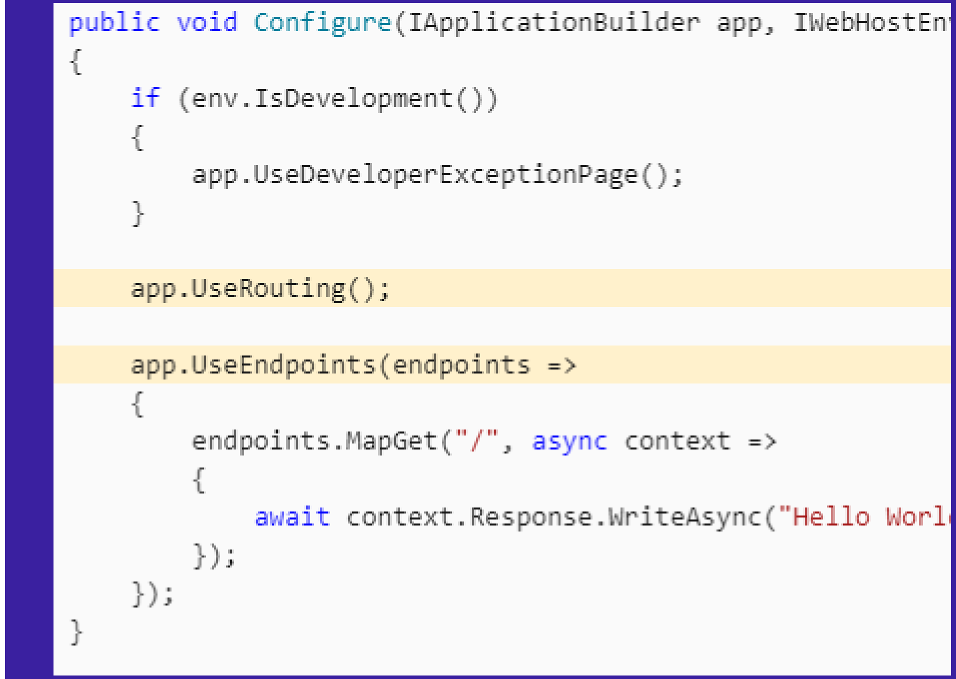
URL matches with Configured Routes?

Routing Engine

YES

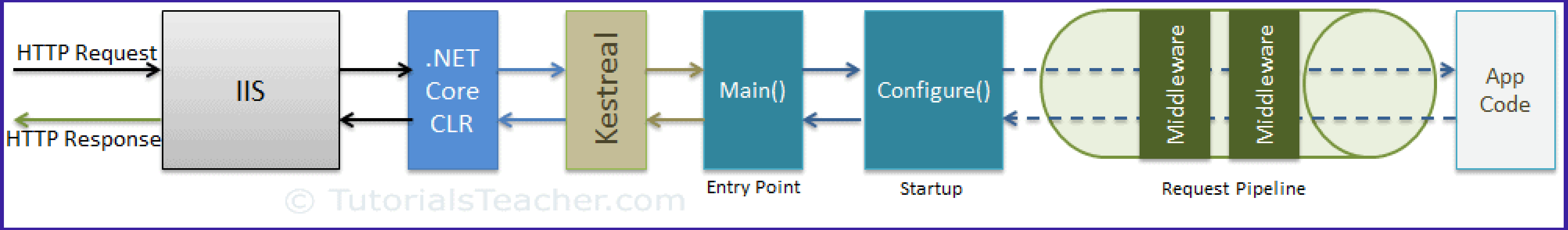
Execute the Controller Action Method

Routing in ASP.NET MVC

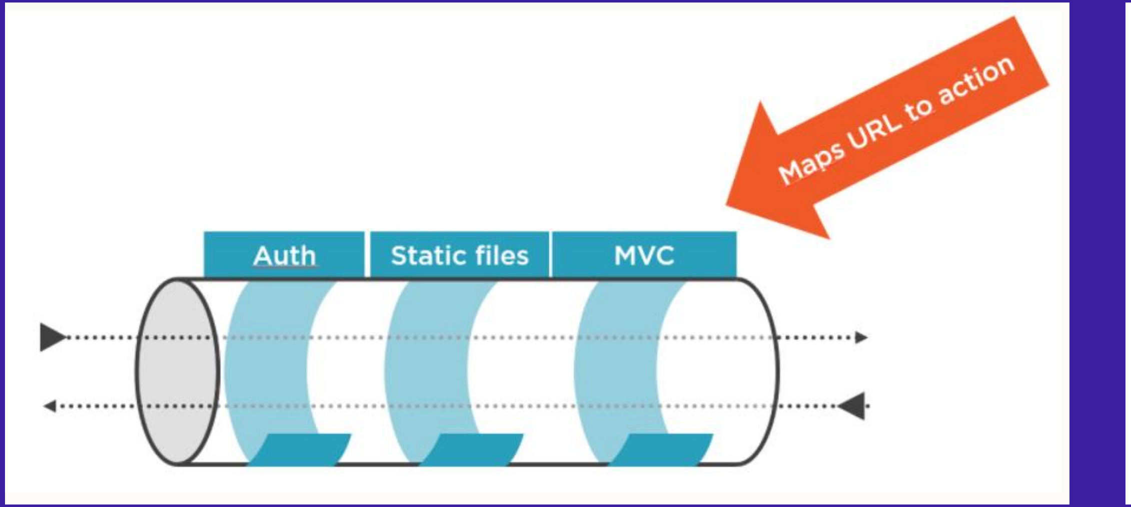


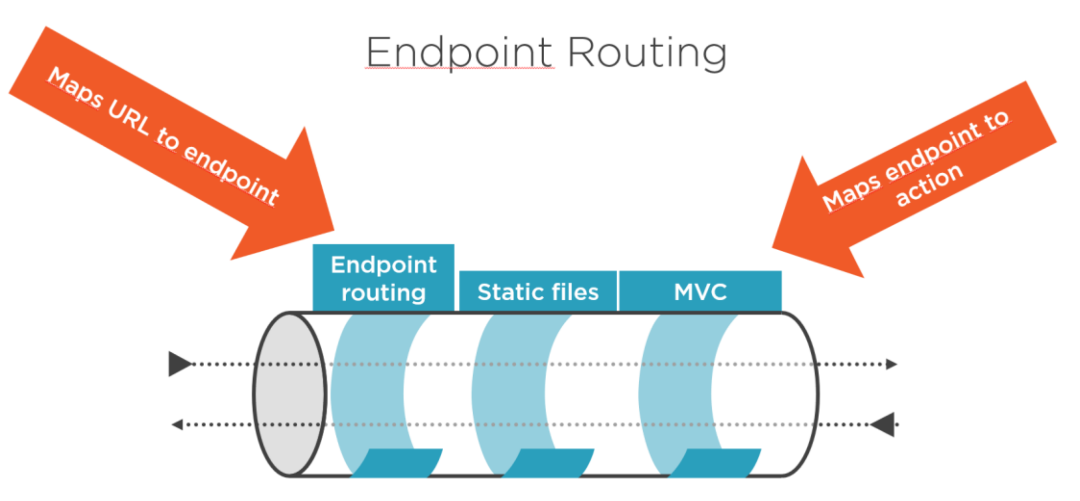
```
public void Configure(IApplicationBuilder app, IWebHostEn
{
    if (env.IsDevelopment())
    {
        app.UseDeveloperExceptionPage();
    }

    app.UseRouting();

    app.UseEndpoints(endpoints =>
    {
        endpoints.MapGet("/", async context =>
        {
            await context.Response.WriteAsync("Hello Worl
        });
    });
}
```

Endpoints



- Before
- After





Thanks for joining!