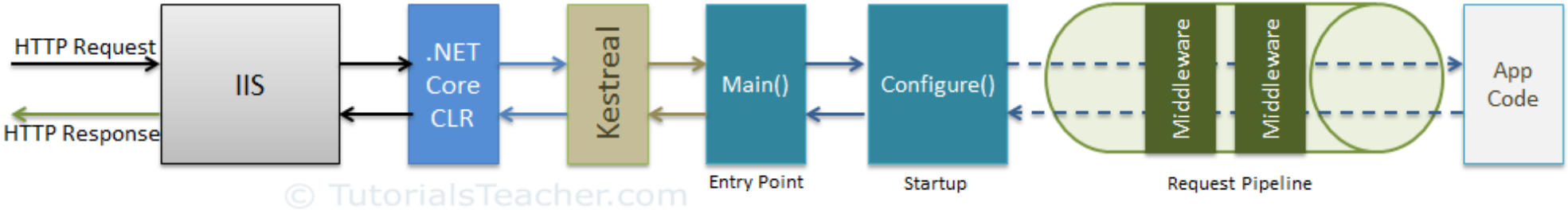
MCSE 541:Web Computing and Mining

ASP.NET Core-Use of Middleware

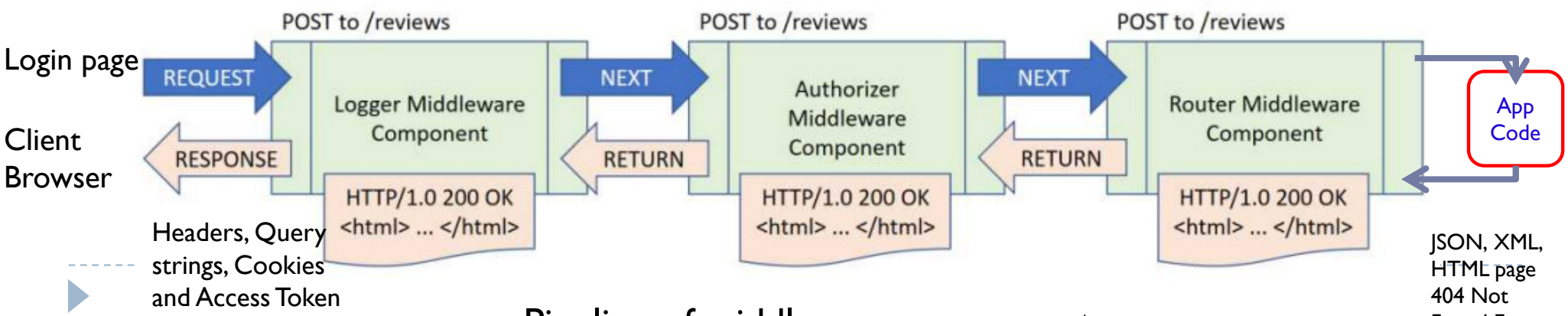
Prof. Dr. Shamim Akhter
shamimakhter@iubat.edu

ASP.NET Core Request Processing



Middleware

- A new concept in ASP.NET Core
- A component (class) which is executed on every http request in ASP.NET Core application.
  - E.g. How the application behaves if there is an error.
- In the classic ASP.NET,
  - HttpHandlers and HttpModules were part of the request pipeline.
- Middleware is similar to HttpHandlers and HttpModules
  - E.g. performs user authentication and authorization.



Pipeline of middleware components

Configure Middleware

- Middleware is configured into the Startup class's Configure method where an IApplicationBuilder interface instance is injected.
- IApplicationBuilder Interface, which is used to defines a class that provides the mechanisms to configure an application's request pipeline.

```
public class Startup {
   public Startup() { }
   public void Configure(IApplicationBuilder app, IHostingEnvironment env,
                         ILoggerFactory loggerFactory)
   { //configure middleware using IApplicationBuilder here..
       app.Run(async (context) => { await context.Response.WriteAsync("Hello World!");
                                   });
          // other code removed for clarity..
    }
}
```

Run() is an **extension method** on IApplicationBuilder instance which adds a terminal middleware to the application's request pipeline.

- The above configured middleware returns a response with a string "Hello World!" for each request.

Extension Method

- Additional method to be injected into a class/interface
  - without modifying, deriving or recompiling the original class.
  - own custom class, .NET framework classes, or third party classes or interfaces can use extension method.

```
int i = 10;
bool result = i.IsGreaterThan(100); //returns false
```

Define an Extension Method

```
namespace ExtensionMethods {
    public static class IntExtensions {
        public static bool IsGreaterThan(this int i, int value) {
            return i > value;
        }
    }
}
```

Binding parameter to bind these methods with int i class.

An extension method can take extra parameters, in addition to an instance of the type that it is extending.

```
using ExtensionMethods;
class Program {
    static void Main(string[] args) {
        int i = 10;
        bool result = i.IsGreaterThan(100);
        Console.WriteLine(result);
    }
}
```

Class Example with Extension Method

```
Using System;
namespace ExtensionMethod {
// Here Geek class contains three methods. Now we want to add two more new methods in it without re-compiling this class
class Geek {
        public void M1()
        {
                Console.WriteLine("Method Name: M1");
         }
         public void M2()
        {
                Console.WriteLine("Method Name: M2");
        }
        public void M3()
        {
                Console.WriteLine("Method Name: M3");
        }
    }
}
```

Defining Extension Methods

```
using System;

namespace ExtensionMethod {

// This class contains M4 and M5 methods. Which we want to add in
//Geek class. NewMethodClass is a static class
static class NewMethodClass {

    public static void M4(this Geek g)
    {
        Console.WriteLine("Method Name: M4");
    }

    public static void M5(this Geek g, string str)
    {
        Console.WriteLine(str);
    }
}

}
```

Call Extension Method

```
using System;

namespace ExtensionMethod {

public class GFG {

    // Main Method
    public static void Main(string[] args)
    {
        Geek g = new Geek();
        g.M1();
        g.M2();
        g.M3();
        g.M4();
        g.M5("Method Name: M5");
    }
}
}
```

Configure Middleware

- Middleware is configured into the Startup class's Configure method where an IApplicationBuilder interface instance is injected.
- IApplicationBUilder Interface, which is used to defines a class that provides the mechanisms to configure an application's request pipeline.

```
public class Startup {
   public Startup() { }
   public void Configure(IApplicationBuilder app, IHostingEnvironment env,
                         ILoggerFactory loggerFactory)
   { //configure middleware using IApplicationBuilder here..
       app.Run(async (context) => { await context.Response.WriteAsync("Hello World!");
                                   });
          // other code removed for clarity..
    }
}
```

Why- async and await?

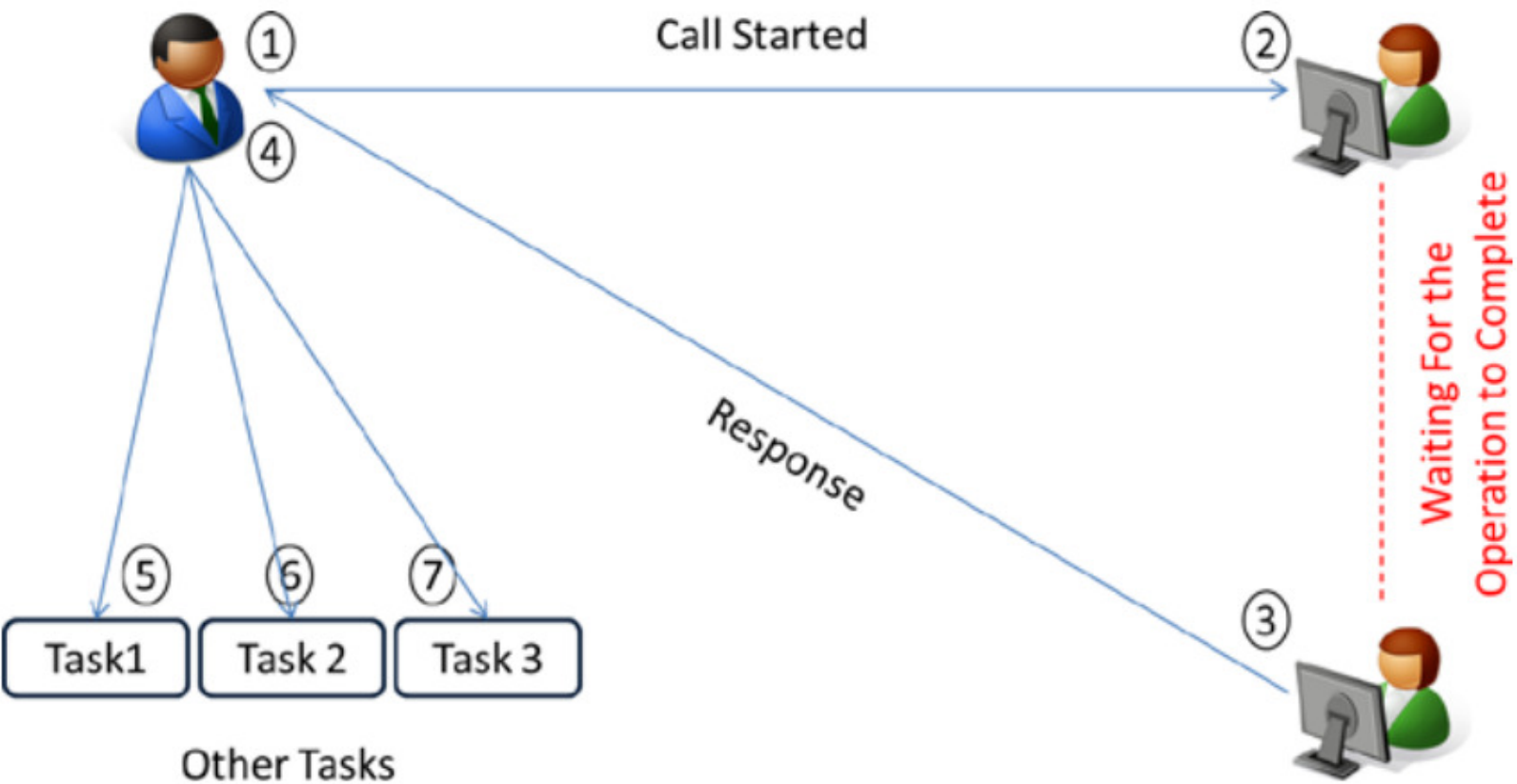
Run() is an extension method on IApplicationBuilder instance which adds a terminal middleware to the application's request pipeline.

The above configured middleware returns a response with a string "Hello World!" for each request.

Synchronous vs Asynchronous

Assume that a few days ago, after I bought a product from Company A, I began having a problem with it. I called the company's support center to explain and sort out the problem. After listening to my explanation, the customer service representative asked me to wait a few minutes. While he tried to solve the problem, I was left hanging on the telephone. The important part here is that I couldn't do anything else until the representative got back to me. Any other tasks I needed to perform were, like me, left on hold while I waited for my call to end.

My situation in this scenario can be related to **synchronous** processing of a long-running operation (Figure 2-1).



Synchronous vs Asynchronous

Let's construct another scenario. This time I bought a product from Company B; again, there was a problem. I called Company B's support center and explained the problem to a customer service representative. This time, however, since the representative said she would call me back as soon as the problem got sorted out, I could hang up the phone. This allowed me to see to other tasks while Company B's people worked on my problem. Later, the representative called me back and informed me of the problem's resolution. This scenario resembles how an **asynchronous** operation works (see Figure 2-2).

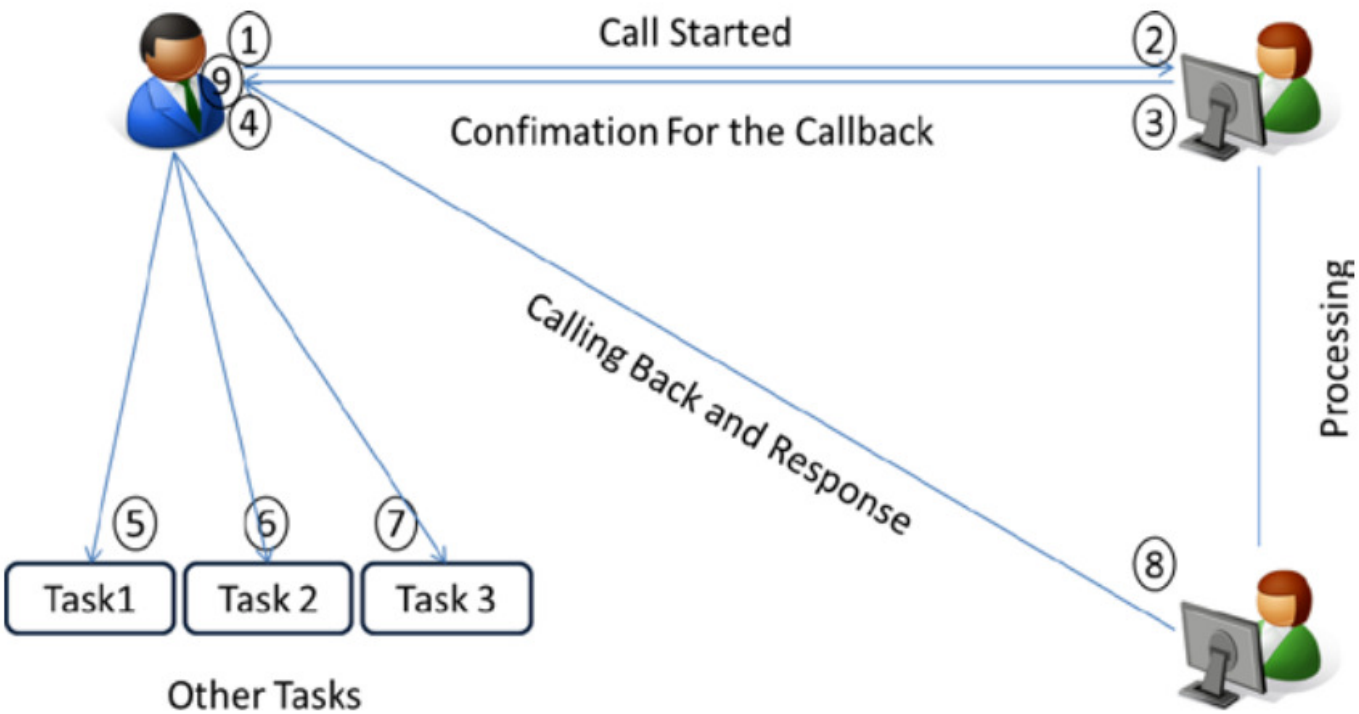


Figure 2-2. *An asynchronous phone call between me and Company B's customer service representative*

Run Method Signature

public static void Run(this IApplicationBuilder app, RequestDelegate handler)

Delegate Signature

Request delegates are used to build the request pipeline.
The request delegates handle each HTTP request.

public delegate Task RequestDelegate(HttpContext context);

```
public class Startup {
    public Startup() { }
    public void Configure(IApplicationBuilder app, IHostingEnvironment env)
    {
        app.Run(MyMiddleware);
    }

    private Task MyMiddleware(HttpContext context)
    {
        return context.Response.WriteAsync("Hello World! ");
    }
}
```

MyMiddleware function is not asynchronous
Thus will block the thread till the time it completes the execution.
So, make it asynchronous by using async and await to improve performance and scalability.

Asynchronous MyMiddleware

```
private async Task MyMiddleware(HttpContext context)
{
    await context.Response.WriteAsync("Hello World! ");
}
```

- The await operator suspends evaluation of the enclosing async method until the asynchronous operation represented by its operand completes.
- When the asynchronous operation completes, the await operator returns the result of the operation, if any.

https://www.c-sharpcorner.com/article/async-and-await-in-c-sharp/

```
using System;
using System.Threading.Tasks;

namespace ASynExample1
{
    class Program
    {
        ***
     }
}
```

```
static async Task Main(string[] args)
 {
     await callMethod();
     Console.ReadKey();
 }

 public static async Task callMethod()
 {
     Method2();
     var count = await Method1();
     Method3(count);
 }
```

```
public static async Task<int> Method1()
{
    int count = 0;
    await Task.Run(() =>
    {
        for (int i = 0; i < 100; i++)
        {
            Console.WriteLine(" Method 1");
            count += 1;
        }
    });
    return count;
}

public static void Method2()
{
    for (int i = 0; i < 25; i++)
    {
        Console.WriteLine(" Method 2");
    }
}

public static void Method3(int count)
{
    Console.WriteLine("Total count is " + count);
}
```

Run method calls with Delegate

```
public class Startup {
    public Startup() { }
    public void Configure(IApplicationBuilder app, IHostingEnvironment env)
    {
        Microsoft.AspNetCore.Http.RequestDelegate Rd = MyMiddleware;
            //public delegate Task RequestDelegate(HttpContext context);
         app.Run(Rd);
    }

    private Task MyMiddleware(HttpContext context)
    {
       return context.Response.WriteAsync("Hello World! ");
    }
}
```

```
Microsoft.AspNetCore.Http.RequestDelegate Rd = (HttpContext context)
=>  context.Response.WriteAsync("Hello World10! ");
App.Run(Rd);
```

```
app.Run((HttpContext context) => context.Response.WriteAsync("Hello World10! "));
```

Configure Multiple Middlewares

There will be multiple middleware components in ASP.NET Core application which will be executed sequentially.

```
public class Startup {
   public Startup() { }
   public void Configure(IApplicationBuilder app, IHostingEnvironment env,
                         ILoggerFactory loggerFactory)
   { //configure middleware using IApplicationBuilder here..
       app.Run(async (context) => { await context.Response.WriteAsync("Hello World1!");
                                   });
       // the following will never be executed
       app.Run(async (context) => { await context.Response.WriteAsync("Hello World2!");
                                   });

    }
}
```

Use() Extension Method

We can use Use() method to configure multiple middleware in the order we like.

```
public class Startup {
   public Startup() { }
   public void Configure(IApplicationBuilder app, IHostingEnvironment env,
                         ILoggerFactory loggerFactory)
   { //configure middleware using IApplicationBuilder here..
      app.Use(async (context, next) => { await context.Response.WriteAsync("Hello World1!");

      await  next();
                                        });

       // the following will never be executed
      app.Run(async (context) => { await context.Response.WriteAsync("Hello World2!");
                                   });

    }
}
```

Add Built-in Middleware Via NuGet

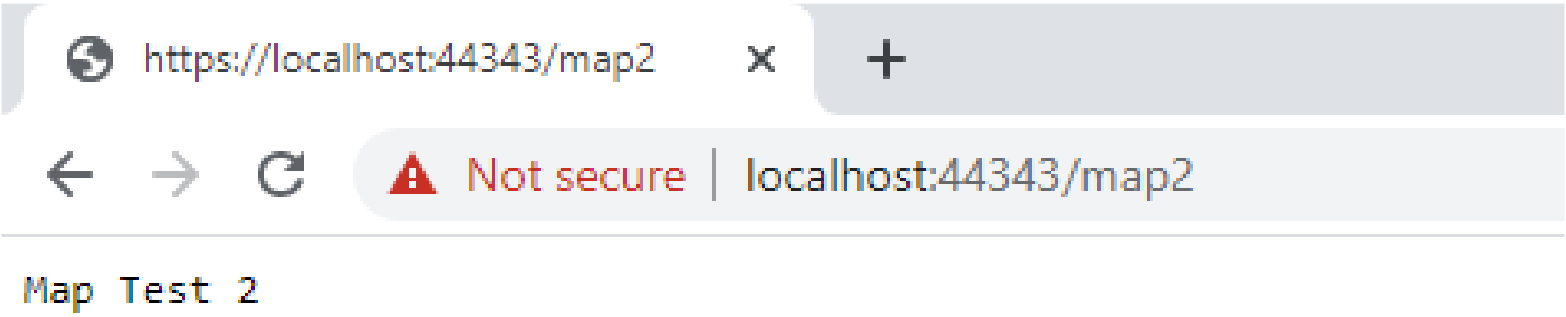
Middleware	Description
Authentication	Adds authentication support.
CORS	Configures Cross-Origin Resource Sharing.
Routing	Adds routing capabilities for MVC or web form
Session	Adds support for user session.
StaticFiles	Adds support for serving static files and directory browsing.
Diagnostics	Adds support for reporting and handling exceptions and errors.

Map Extension Method

- Map extensions are used as a convention for branching the pipeline.
- Map branches the request pipeline based on matches of the given request path.
  - If the request path starts with the given path, the branch is executed.

```
private static void HandleMapTest1(IApplicationBuilder app) {
        app.Run(async context => {
            await context.Response.WriteAsync("Map Test 1"); });
        }
private static void HandleMapTest2(IApplicationBuilder app) {
        app.Run(async context => {
            await context.Response.WriteAsync("Map Test 2"); }); }

public void Configure(IApplicationBuilder app) {
        app.Map("/map1", HandleMapTest1);
        app.Map("/map2", HandleMapTest2);
        app.Run(async context =>
 { await context.Response.WriteAsync("Hello from non-Map delegate. <p>"); }); }
```

Request	Response
localhost:1234	Hello from non-Map delegate.
localhost:1234/map1	Map Test 1
localhost:1234/map2	Map Test 2
localhost:1234/map3	Hello from non-Map delegate.

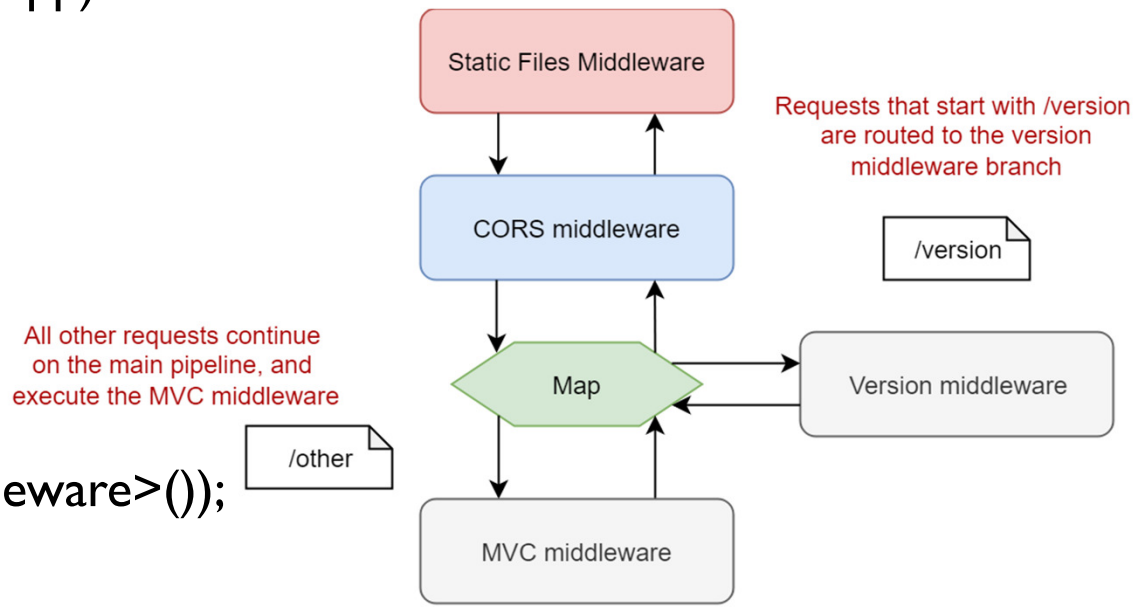



```
Map Test 2
```

Concrete middleware using Map() in ASP.NET Core 2.x

- In ASP.NET Core 2.x, you could include the middleware as a conditional branch in your middleware pipeline, so that it only executes if the app receives a request starting with "/version":

```
public void Configure(IApplicationBuilder app)
{
    app.UseStaticFiles();

    app.UseCors();

    app.Map("/version", versionApp =>

versionApp.UseMiddleware<VersionMiddleware>());

    app.UseMvcWithDefaultRoute();
}
```



- https://andrewlock.net/converting-a-terminal-middleware-to-endpoint-routing-in-aspnetcore-3/

Before ASP.NET Core 3.0

Before ASP.NET Core 3.0 the Route resolution & invoking of the Route were part of the MVC Middleware. We defined the routes while configuring the MVC is using the app.UseMvc as shown below

```
app.UseMvc(routes =>  {
      routes.MapRoute("default",
                        "{controller}/{action}");
});
```

When a new request arrives, the routing middleware parses the incoming URL.

- A matching route is searched in the RouteCollection.
- If a matching route is found, control is passed to the RouteHandler
- If a matching route is not found, the next middleware is invoked.

What is an Endpoint

An Endpoint is an object that contains everything that you need to execute the incoming Request. The Endpoint object contains the following information

- Metadata of the request.
- The delegate (Request handler) that ASP.NET core uses to process the request.

We define the Endpoint at the application startup using the UseEndpoints method.

The ASP.NET Core routing is now not tied up only to the MVC Endpoints. You can define an Endpoint, which can also hit a Razor page, SignalR etc.

The Endpoint Routing has three components

- Defining the Endpoints.
- Route matching & Constructing an Endpoint (UseRouting).
- Endpoint Execution (UseEndpoints).

https://www.tektutorialshub.com/asp-net-core/asp-net-core-endpoint-routing/

```
public void Configure(IApplicationBuilder app, IWebHostEnvironment env)
{
    if (env.IsDevelopment())
    {
        app.UseDeveloperExceptionPage();
    }
    else
    {
        app.UseExceptionHandler("/Home/Error");
        // The default HSTS value is 30 days. You may want to change this for production scenarios, see
https://aka.ms/aspnetcore-hsts.
        app.UseHsts();
    }
    app.UseHttpsRedirection();
    app.UseStaticFiles();

    app.UseRouting();

    app.UseAuthorization();

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

Configure the Endpoint

We configure the Endpoint in the app.UseEndpoints method.

The following method adds the default MVC Conventional route using the MapControllerRoute method.

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

To setup an attribute based Routing use the method MapControllers. We use the attribute based Routing to create a route to Rest API (Web API Controller Action Method). You can also use it to create a route to MVC Controller action method.

```
endpoints.MapControllers();
```

MapControllerRoute & MapControllers methods hides all the complexities of setting up the Endpoint from us. Both sets up the Endpoint to Controller action methods

You can also create an Endpoint to a custom delegate using MapGet method. MapGet accepts two argument. One is Route Pattern (/ in the example) and a request delegate.

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

Converting the middleware to endpoint routing

- In ASP.NET Core 3.0, we use endpoint routing, so the routing step is separated from the invocation of the endpoint. In practical terms that means we have two pieces of middleware:
  - EndpointRoutingMiddleware that does the actual routing i.e. calculating which endpoint will be invoked for a given request URL path.
  - EndpointMiddleware that invokes the endpoint.

```
public void Configure(IApplicationBuilder app)
{
    app.UseStaticFiles();

    // Add the EndpointRoutingMiddleware
    app.UseRouting();

    // All middleware from here onwards know which endpoint will be invoked
    app.UseCors();

    // Execute the endpoint selected by the routing middleware
    app.UseEndpoints(endpoints =>
    {
        endpoints.MapDefaultControllerRoute();
    });
}
```

Map the VersionMiddleware to endpoint routing

```
public void Configure(IApplicationBuilder app)
{
    app.UseStaticFiles();

    app.UseRouting();

    app.UseCors();

    app.UseEndpoints(endpoints =>
    {
        // Add a new endpoint that uses the VersionMiddleware
        endpoints.Map("/version", endpoints.CreateApplicationBuilder()
            .UseMiddleware<VersionMiddleware>()
            .Build())
            .WithDisplayName("Version number");

        endpoints.MapDefaultControllerRoute();
    });
}
```

Map Version Extension Method

```
public static class VersionEndpointRouteBuilderExtensions
{
    public static IEndpointConventionBuilder MapVersion(this IEndpointRouteBuilder endpoints,
string pattern)
    {
        var pipeline = endpoints.CreateApplicationBuilder()
            .UseMiddleware<VersionMiddleware>()
            .Build();

        return endpoints.Map(pattern, pipeline).WithDisplayName("Version number");
    }
}
```

```
public void Configure(IApplicationBuilder app)
{
    app.UseStaticFiles();

    app.UseRouting();

    app.UseCors();

    // Execute the endpoint selected by the routing middleware
    app.UseEndpoints(endpoints =>
    {
        endpoints.MapVersion("/version");
        endpoints.MapDefaultControllerRoute();
    });
}
```

Building Custom Middleware

- So, as you know from the definition of middleware, a RequestDelegate is a method that will handle an HTTP request. All the information about that HTTP request will be inside of the HttpContext passed as a parameter inside of RequestDelegate. Based on this HttpContext, the middleware will be able to inspect the request and produce a response.
- Let's call our middleware component SayHello, which we pass inside of Use() .

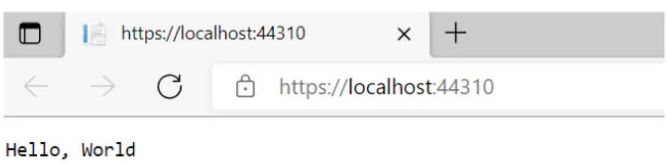
```
app.Use(SayHello);
```

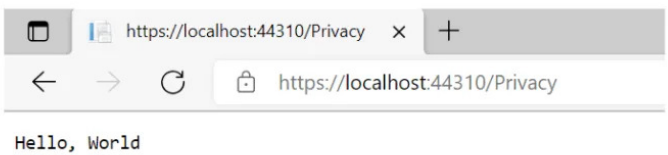
```
private RequestDelegate SayHello(RequestDelegate arg)
{
    return async ctx =>
    {
        await ctx.Response.WriteAsync("Hello, World");
    };
}
```

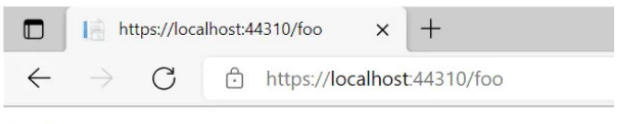
```
public void Configure(IApplicationBuilder app, IWebHostEnvironment env)
{
    if (env.IsDevelopment())
    {
        app.UseDeveloperExceptionPage();
    }
    else
    {
        app.UseExceptionHandler("/Error");
        // The default HSTS value is 30 days. You may want to change thi
        app.UseHsts();
    }
    app.Use(SayHello);

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

    app.UseRouting();

    app.UseAuthorization();

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

Place the app.use at the beginning.

DeveloperExceptionPage is the first component in the middleware pipeline, always pass the HTTP request onto the next component in the pipeline. The next component is our SayHello. From its definition, we know this component will always return a response – the message "Hello, World!". Once SayHello produces a response, it will be passed back to the previous component in the pipeline, DeveloperExceptionPage.

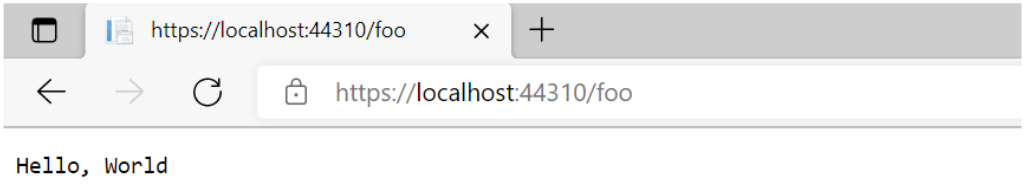






```
public void Configure(IApplicationBuilder app, IWebHostEnvironment env)
{
    if (env.IsDevelopment())
    {
        app.UseDeveloperExceptionPage();
    }
    else
    {
        app.UseExceptionHandler("/Error");
        // The default HSTS value is
        app.UseHsts();
    }


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

    app.UseRouting();

    app.UseAuthorization();

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

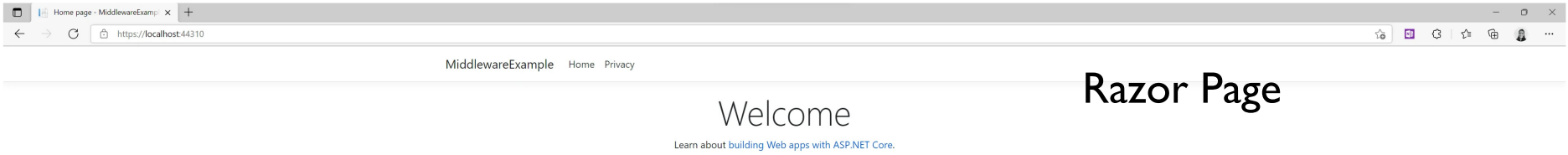
When the HTTP request is valid, meaning that the Endpoints middleware does have something to return, then our web app will work exactly like before we added the new piece of middleware. This is because the HTTP request never reaches the SayHello. If, however, the HTTP request is an invalid URL, we will then hit the end of the pipeline, as none of the earlier components will be able to return anything.



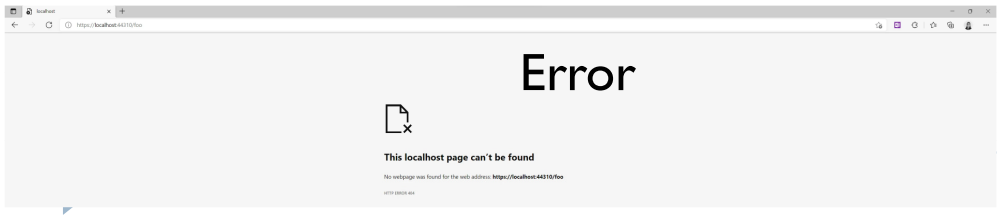
Place the app.use at the last.

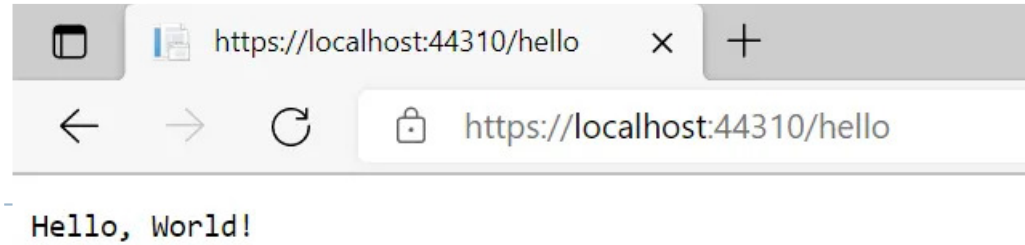
Activate SayHello only when the URL in the request is “/hello”.

```
private RequestDelegate SayHello(RequestDelegate next)
    {
        return async ctx =>
        {
            if (ctx.Request.Path.StartsWithSegments("/hello"))
            {
                await ctx.Response.WriteAsync("Hello, World!");
            }
            else
            {
                        await next(ctx);
            }
        };
    }
```



Razor Page





Using Endpoints

```
app.UseEndpoints(endpoints =>
{
    endpoints.Map("/hello", endpoints.CreateApplicationBuilder()
        .Use(async (context, next) =>
                    { await context.Response.WriteAsync("Hello, World!");
                        await next();
                    })

         .Build());
    endpoints.MapRazorPages();
});
```