**Configure ASP.NET Core request processing pipeline**

As part of the application startup, **Configure()** method sets up the request processing pipeline.

public class Startup

{

    public void ConfigureServices(IServiceCollection services)

    {

    }

    public void Configure(IApplicationBuilder app, IHostingEnvironment env)

    {

        if (env.IsDevelopment())

        {

            app.UseDeveloperExceptionPage();

        }

        app.Run(async (context) =>

        {

            await context.Response.WriteAsync("Hello World!");

        });

    }

}

**At the moment we have 2 middlewares in the pipeline**

* UseDeveloperExceptionPage() and
* another middleware that is registered using the Run() method.

**UseDeveloperExceptionPage Middleware :** As the name implies, this middleware responds with the developer exception page, if there is an exception and if the environment is Development. We will discuss this DeveloperExceptionPage middleware and the  different environments in our upcoming sessions.   
  
The second middleware that is registered using the Run() method, can only write a message to the Response object. At the moment, this is the middleware that responds to every request.   
Doesn’t matter what your request path is. All requests will be handled by this one piece of middleware and the response we get is the string message that the middleware is writing to the Response object. The response is plain text and not html. We can confirm this by inspecting the page source. Notice we do not have any html tags in the source. It’s just, plain text.   
  
Even if you have a file with name **foo.html**and if you include the path to that file in the request, our application will not be able to serve that static file. This is because, at the moment our request processing pipeline does not have the middleware that can serve static files like html files, images, css and JavaScript files. As we progress through this course we will add the required middleware to be able to serve static files.   
  
**Consider the following code in the Configure() method.**

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

**Code Explanation**

* We are using Run() method to add middleware to our application's request processing pipeline
* If you hover the mouse over the Run() method, from the intellisense you can see that this Run() method is implemented as an extension method of IApplicationBuilder interface. This is the reason we are able to invoke this Run() method on IApplicationBuilder object app.
* The parameter that we are passing to the Run() method is a RequestDelegate which we can see from the intellisense.
* RequestDelegate is a delegate that has HttpContext object as a parameter.
* It is through this HttpContext object, the middleware gains access to both the incoming http request and outgoing http response.
* At the moment, we are passing request delegate inline as an anonymous method using a lambda.
* Instead of passing the request delegate inline as an anonymous method, we can define the request delegate in a separate reusable class.
* With this Run() extension method we can only add a terminal middleware to the request pipeline.
* A terminal middleware is a middleware that does not call the next middleware in the pipeline

**Consider the following code**

app.Run(async (context) =>  
{  
    await context.Response.WriteAsync("Hello from 1st Middleware");  
});  
  
app.Run(async (context) =>  
{  
    await context.Response.WriteAsync("Hello from 2nd Middleware");  
});

* We have 2 middlewares registered using the Run() method
* Upon running this project, we only see the response from the first middleware
* We do not see the response from the second middleware
* This is because, a middleware that is registered using the Run() method cannot call the next middleware in the pipeline
* So, the middleware that we register using Run() method is a terminal middleware

If you want your middleware to be able to call the next middleware in the pipeline, then register the middleware using **Use()** method as shown below.

app.Use(async (context, next) =>  
{  
    await context.Response.WriteAsync("Hello from 1st Middleware");  
    await next();  
});  
  
app.Run(async (context) =>  
{  
    await context.Response.WriteAsync("Hello from 2nd Middleware");  
});

Notice, **Use()** method has 2 parameters. The first parameter is the HttpContext context object and the second parameter is the Func type i.e it is a generic delegate that represents the next middleware in the pipeline.   
  
Now, consider the following code

public void Configure(IApplicationBuilder app, IHostingEnvironment env,  
                ILogger<Startup> logger)  
{  
    app.Use(async (context, next) =>  
    {  
        logger.LogInformation("MW1: Incoming Request");  
        await next();  
        logger.LogInformation("MW1: Outgoing Response");  
    });  
  
    app.Use(async (context, next) =>  
    {  
        logger.LogInformation("MW2: Incoming Request");  
        await next();  
        logger.LogInformation("MW2: Outgoing Response");  
    });  
  
    app.Run(async (context) =>  
    {  
        await context.Response.WriteAsync("MW3: Request handled and response produced");  
        logger.LogInformation("MW3: Request handled and response produced");  
    });  
}

* ILogger<Startup> is injected into the Configure() method
* CreateDefaultBuilder() that is called by the Main() method configures logging
* You can verify this on their GitHub page using the link below  
  <https://github.com/aspnet/MetaPackages/blob/release/2.2/src/Microsoft.AspNetCore/WebHost.cs>
* Please check the method ConfigureLogging()
* You will find that, loggers for Console, Debug and EventSource are configured
* We are using the logger instance provided by the Dependency Injection system to log the information
* If you run the project using the .NET Core CLI, you can see the logged information on the Console window
* If you run the project directly from Visual Studio, you can see the logged information in the output window. Select ASP.NET Core Web Server from the dropdownlist in the output window.
* You will see that, the information is logged in the following order
  + **MW1: Incoming Request**
  + **MW2: Incoming Request**
  + **MW3: Request handled and response produced**
  + **MW2: Outgoing Response**
  + **MW1: Outgoing Response**

Now relate the above output, with the following diagram from MSDN to understand what's happening.

* Remember a middleware in asp.net core has access to both the incoming request and the outgoing response
* The request first arrives at Middleware1 which logs (MW1: Incoming Request) so we see this message first.
* Then Middleware1 calls next(). Calling next() invokes Middleware2 in the pipeline.
* Middleware2 logs (MW2: Incoming Request). So we see (MW2: Incoming Request) after (MW1: Incoming Request).
* Then Middleware2 calls next() which invokes Middleware3.
* Middleware3 handles the request and produces a response. Hence, the next message that we see is (MW3: Request handled and response produced)
* So, at this point the pipeline starts to reverse itself.
* The control is then given to Middleware2 and the response produced by Middleware3 is passed to it. Middleware2 then logs (MW2: Outgoing Response) which is what we see next.
* Finally Middleware2 gives control to Middleware1.
* Middleware1 logs (MW1: Outgoing Response) which is what we see finally.

**So here are the 3 very important points to keep in mind regarding the request processing pipeline**

* Everything that happens before the next() method is invoked in each of the middleware components, happen as the request travels from middleware to middleware through the pipeline and this is represented by the **incoming arrow**.
* When a middleware handles the request and produces response, the request processing **pipeline starts to reverse**.
* Everything that happens after the next() method is invoked in a middleware component, happens as the response travels from middleware to middleware through the pipeline and this is represented by the **outgoing arrow**.

I hope you now have a good understanding of the request processing pipeline in asp.ent core.