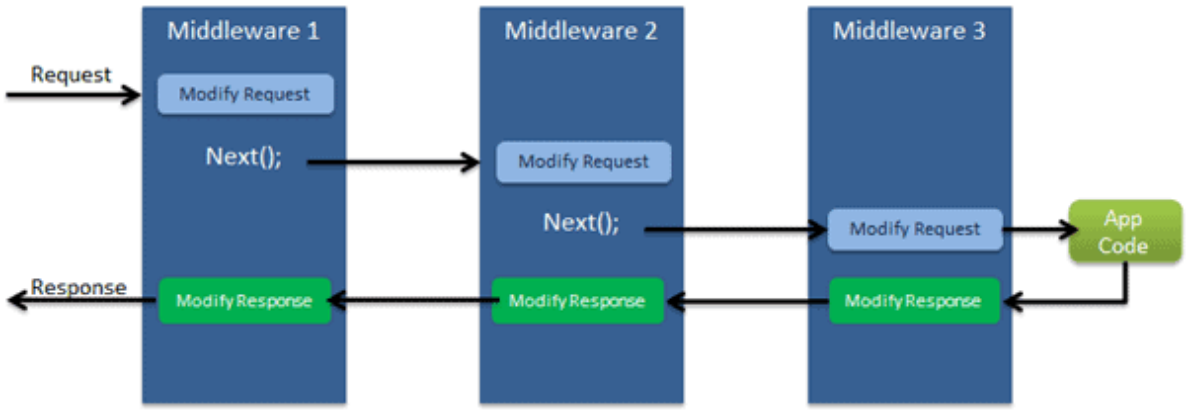
‹ Previous Next ›

ASP.NET Core - Middleware

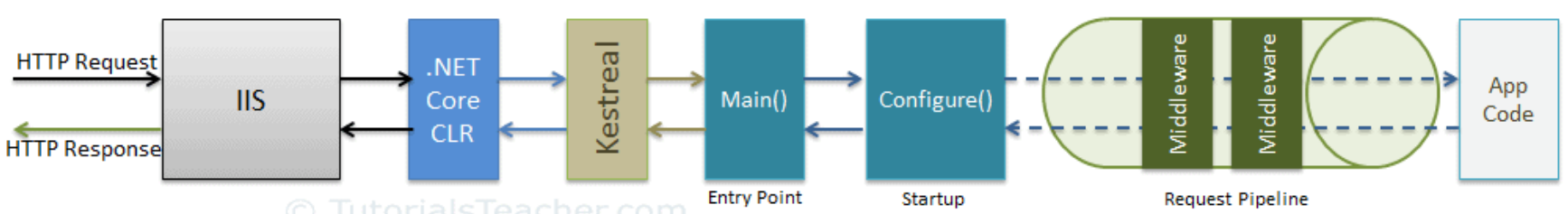
ASP.NET Core introduced a new concept called **Middleware.** A middleware is nothing but a component (class) which is executed on every request in ASP.NET Core application. In the classic ASP.NET, HttpHandlers and HttpModules were part of request pipeline. Middleware is similar to HttpHandlers and HttpModules where both needs to be configured and executed in each request.

Typically, there will be multiple middleware in ASP.NET Core web application. It can be either framework provided middleware, added via NuGet or your own custom middleware. We can set the order of middleware execution in the request pipeline. Each middleware adds or modifies http request and optionally passes control to the next middleware component. The following figure illustrates the execution of middleware components.



ASP.NET Core Middleware

Middlewares build the request pipeline. The following figure illustrates the ASP.NET Core request processing.



ASP.NET Core Request Processing

Configure Middleware

We can configure middleware in the `Configure` method of the Startup class using `IApplicationBuilder` instance. The following example adds a single middleware using Run method which returns a string "Hello World!" on each request.

```
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..
    }
}
```

In the above example, `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.

Understand Run Method

We used Run extension method to add middleware. The following is the signature of the Run method:

Method Signature:

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

The Run method is an extension method on `IApplicationBuilder` and accepts a parameter of `RequestDelegate`. The `RequestDelegate` is a delegate method which handles the request. The following is a RequestDelegate signature.

Method Signature:

```
public delegate Task RequestDelegate(HttpContext context);
```

As you can see above, the `Run` method accepts a method as a parameter whose signature should match with `RequestDelegate`. Therefore, the method should accept the `HttpContext` parameter and return `Task`. So, you can either specify a lambda expression or specify a function in the Run method. The lambda expression specified in the Run method above is similar to the one in the example shown below.

```
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! ");
    }
}
```

The above `MyMiddleware` function is not asynchronous and so 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.

```
// other code removed for clarity

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

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

Thus, the above code snippet is same as the one below.

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

//or

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

So, in this way, we can configure middleware using `Run` method.

Configure Multiple Middleware

Mostly there will be multiple middleware components in ASP.NET Core application which will be executed sequentially. The Run method adds a terminal middleware so it cannot call next middleware as it would be the last middleware in a sequence. The following will always execute the first Run method and will never reach the second Run method.

```
public void Configure(IApplicationBuilder app, IHostingEnvironment env)
{
    app.Run(async (context) =>
    {
        await context.Response.WriteAsync("Hello World From 1st Middleware");
    });

    // the following will never be executed
    app.Run(async (context) =>
    {
        await context.Response.WriteAsync("Hello World From 2nd Middleware");
    });
}
```

To configure multiple middleware, use `Use()` extension method. It is similar to `Run()` method except that it includes next parameter to invoke next middleware in the sequence. Consider the following example.

Example: Use() Copy

```
public void Configure(IApplicationBuilder app, IHostingEnvironment env)
{
    app.Use(async (context, next) =>
    {
        await context.Response.WriteAsync("Hello World From 1st Middleware!");

        await next();
    });

    app.Run(async (context) =>
    {
        await context.Response.WriteAsync("Hello World From 2nd Middleware");
    });
}
```

The above example will display `Hello World From 1st Middleware!Hello World From 2nd Middleware!` in the browser.

Thus, we can use `Use()` method to configure multiple middlewares in the order we like.

Add Built-in Middleware Via NuGet

ASP.NET Core is a modular framework. We can add server side features we need in our application by installing different plug-ins via NuGet. There are many middleware plug-ins available which can be used in our application.

The followings are some built-in middleware:

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.

Let's see how to use Diagnostics middleware.

Diagnostics Middleware

Let's install and use Diagnostics middleware. Diagnostics middleware is used for reporting and handling exceptions and errors in ASP.NET Core, and diagnosing Entity Framework Core migrations errors.

Open project.json and add Microsoft.AspNetCore.Diagnostics dependency if it is not added. Wait for some time till Visual Studio restores the packages.

This package includes following middleware and extension methods for it.

DeveloperExceptionPageMiddleware	UseDeveloperExceptionPage()	Captures synchronous and asynchronous exceptions from the pipeline and generates HTML error responses.
ExceptionHandlerMiddleware	UseExceptionHandler()	Catch exceptions, log them and re-execute in an alternate pipeline.
StatusCodePagesMiddleware	UseStatusCodePages()	Check for responses with status codes between 400 and 599.
WelcomePageMiddleware	UseWelcomePage()	Display Welcome page for the root path.

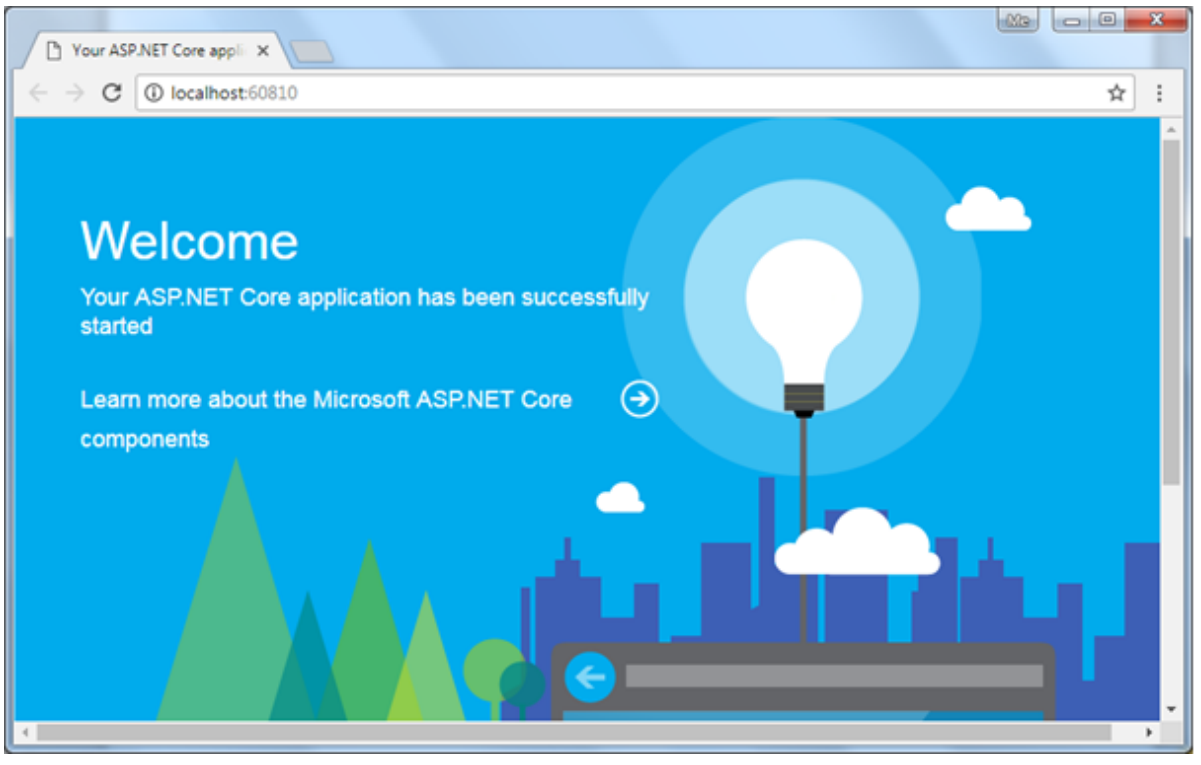
We can call respective Use* extension methods to use the above middleware in the configure method of Startup class.

Let's add welcomePage middleware which will display welcome page for the root path.

Example: Add Diagnostics Middleware — Copy

```
public void Configure(IApplicationBuilder app, IHostingEnvironment env)
{
    app.UseWelcomePage();
    //other code removed for clarity
}
```

The above example will display the following welcome page for each request.



This way we can use different Use* extension methods to include different middleware.

Next, learn how to implement logging functionality in the ASP.NET Core application.

‹ Previous Next ›