

Assignment -5: Middleware

Objectives:

- O[1]. To learn the use of middleware in ASP.NET Core environment.
- O[2]. To understand Run, Use and Map Method.
- O[3]. To learn how to serve static files using `app.UseStaticFiles();`
- O[4]. Apply `app.UseStaticFiles()` Serve static files from different folder than wwwroot folder.
- O[5]. To understand the use of JASON file and services.

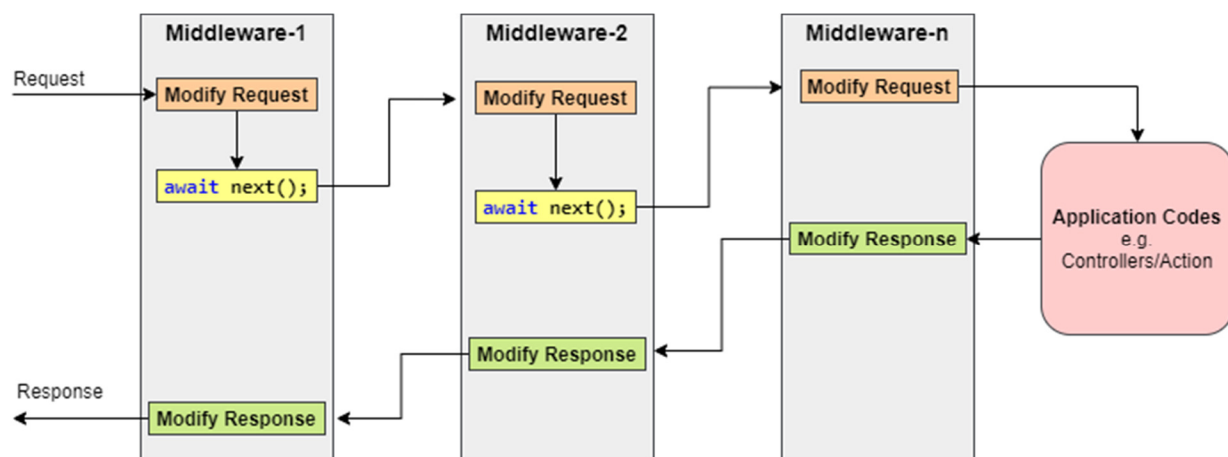
In this works, students are going to understand demonstrates Middleware concepts in ASP.NET Core. In addition, they will learn the use of UseStaticFiles Middleware and apply it to serve statif files. In addition, students students experiment with JASON file and service to learn the techniques of sharing resources.

What is Middleware:

Middleware is a piece of code in an application pipeline used to handle requests and responses. For example, we may have a middleware component to authenticate a user, another piece of middleware to handle errors, and another middleware to serve static files such as JavaScript files, CSS files, images, etc.

Middleware can be built-in as part of the .NET Core framework, added via NuGet packages, or can be custom middleware. These middleware components are configured as part of the application startup class in the configure method. Configure methods set up a request processing pipeline for an ASP.NET Core application. It consists of a sequence of request delegates called one after the other.

The following figure illustrates how a request process through middleware components.



Generally, each middleware may handle the incoming requests and passes execution to the next middleware for further processing.

But a middleware component can decide not to call the next piece of middleware in the pipeline. This is called **short-circuiting or terminate the request pipeline**. Short-circuiting is often desirable because it avoids unnecessary work. For example, **if the request is for a static file like an image CSS file JavaScript file etc., these static files middleware can handle and serve that request and then short-circuit the rest of the pipeline.**

Task1: Choose a WebApp development framework from ASP.NET core environment. ASP.NET core 3.1 supports Razor pages and use endpoints to respond the http request.

- Replace the following UseEndpoints middleware to app.Run middleware

<pre>app.UseEndpoints(endpoints =>{ endpoints.MapRazorPages(); });</pre>	<pre>app.Run(async (context) => { await context.Response.WriteAsync ("Hello World"); });</pre>
--	---

- Run your web application using IIS EXPRESS Web server. Here app is an instance of IApplicationBuilder Interface, which is used to defines a class that provides the mechanisms to configure an application's request pipeline.

app.Run()

This middleware component may expose Run[Middleware] methods that are executed at the end of the pipeline. Generally, this acts as a terminal middleware and is added at the end of the request pipeline, as it cannot call the next middleware.

app.Use()

This is used to configure multiple middleware. Unlike app.Run(), We can include the next parameter into it, which calls the next request delegate in the pipeline. We can also short-circuit (terminate) the pipeline by not calling the next parameter.

app.Map()

These extensions are used as a convention for branching the pipeline. The 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.

Task2: Learn the use of app.Use and app.Run middlewares

- Modify the task1's Startup.cs file and add the following code snippets into the Configure () method.

```
app.Use(async (context, next) =>
```

```

{
    await context.Response.WriteAsync("Before 1stapp.Use()\n");
    await next();
    await context.Response.WriteAsync("After 1stapp.Use()\n");
});

app.Use(async (context, next) =>
{
    await context.Response.WriteAsync("Before 2ndapp.Use()\n");
    await next();
    await context.Response.WriteAsync("After 2ndapp.Use()\n");
});

app.Run(async (context) =>
{
    await context.Response.WriteAsync("Hello 1st app.Run()\n");
});

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

```

- Now run your web application and understand the use of Use() and Run() methods.

Task3: Learn the use of app.Map middlewares

- Remove app.Use() and app.Run() from task2's Startup.cs file and add the following code snippets into the Configure () method.

```

app.Map("/m1", HandleMapOne);
app.Map("/m2", appMap => {
    appMap.Run(async context =>
    {
        await context.Response.WriteAsync("Hello from 2nd app.Map()");
    });
});

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

```

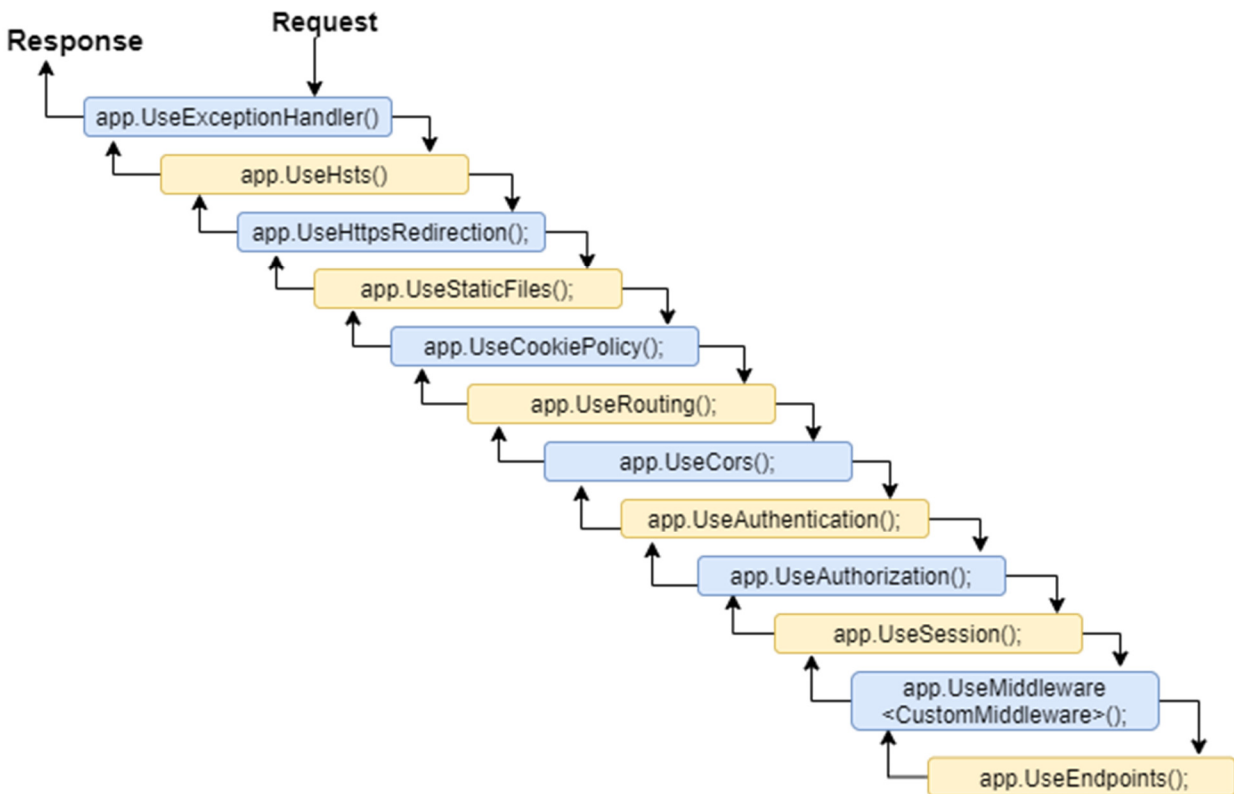
- Now you need to create the following method outside the Configure(). So that the /m1 request can be handled.

```
private static void HandleMapOne(IApplicationBuilder app)
{
    app.Run(async context =>
    {
        await context.Response.WriteAsync("Hello from 1st app.Map()");
    });
}
```

- Now run your web application and understand the use of Map() method.

Middleware Ordering

Middleware components are executed in the order they are added to the pipeline and care should be taken to add the middleware in the right order otherwise the application may not function as expected. This ordering is critical for security, performance, and functionality. The following middleware components are for common app scenarios in the recommended order:



The first configured middleware has received the request, modify it (if required), and passes control to the next middleware. Similarly, the first middleware is executed at the last while processing a response if the echo comes back down the tube. That's why Exception-handling delegates need to be called early in the pipeline, so they can validate the result and displays a possible exception in a browser and client-friendly way.

Serving Static Files

We can configure middleware to serve static files from other folders along with default web root folder wwwroot.

Task4: default.html file

- Create a static html file(default.html) inside wwwroot directory
- Add `app.UseStaticFiles();` middleware
- Execute the web application. However, default.html is not rendered. This localhost page can't be found is shown.
- You can access default.html using url <https://localhost:44314/default.html>
- However, it is not yet been working as default page. To do so we add `app.UseDefaultFiles();` middleware before `app.UseStaticFiles();` middleware
- Now default.html file is presented by default.

Task5: Create htmlpage.html file as default web page

- Create a static html file(htmlpage.html) inside wwwroot directory
- Now write the following middleware codes into the startup.cs file:

```
0 references | 0 exceptions
public void Configure(IApplicationBuilder app, IHostingEnvironment env)
{
    DefaultFilesOptions options = new DefaultFilesOptions();
    options.DefaultFileNames.Clear();
    options.DefaultFileNames.Add("htmlpage.html");

    app.UseDefaultFiles(options);

    app.UseStaticFiles();
}
```

Task6: Create admin.html file in another folder and access it

- Create a new folder into the project. Rename as admin.
- Now write the following code into the Startup.cs file

```
//app.UseFileServer();
app.UseStaticFiles(new StaticFileOptions()
{
    FileProvider = new PhysicalFileProvider(
        Path.Combine(Directory.GetCurrentDirectory(), "admin")),
    RequestPath = new PathString("/Admin")
});
```

Task7: Sharing a variable from JSON file

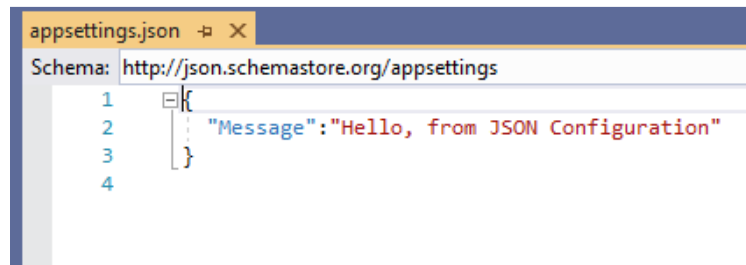
- What is JSON? JSON (JavaScript Object Notation) is a lightweight data-interchange format. It is easy for humans to read and write. It is easy for machines to parse and generate. It is based on a subset of the JavaScript Programming Language Standard ECMA-262 3rd Edition - December 1999. JSON is a text format that is completely language independent but uses conventions that are familiar to programmers of the C-family of languages, including C, C++, C#, Java, JavaScript, Perl, Python, and many others.

JSON is built on two structures:

A collection of name/value pairs. In various languages, this is realized as an object, record, struct, dictionary, hash table, keyed list, or associative array.

An ordered list of values. In most languages, this is realized as an array, vector, list, or sequence.

- There is a shared JSON file named as appsettings.json modify and place a key-value pair as follows:



The screenshot shows a code editor window titled 'appsettings.json'. The schema is set to 'http://json.schemastore.org/appsettings'. The content of the file is a JSON object with a single key-value pair: "Message": "Hello, from JSON Configuration".

```
1 {  
2   "Message": "Hello, from JSON Configuration"  
3 }  
4
```

- Change the following code snippet to Startup.cs file.
Note that version 3.1 does need to build Configuration. It will be injected automatically by the runtime system.



The screenshot shows the Startup.cs file in Visual Studio. The code snippet for the Configure method is highlighted. The code uses the ConfigurationBuilder to load the appsettings.json file and then injects the Configuration into the application. The screenshot also shows the Solution Explorer on the right, which displays the project structure for 'WebApplication7'.

```
4 references | 0 exceptions  
public IConfiguration Configuration { get; set; }  
0 references | 0 exceptions  
public Startup()  
{  
    var builder = new ConfigurationBuilder()  
        .SetBasePath(Directory.GetCurrentDirectory())  
        .AddJsonFile("appsettings.json");  
    Configuration = builder.Build();  
}  
0 references | 0 exceptions  
public void ConfigureServices(IServiceCollection services) { ... }  
0 references | 0 exceptions  
public void Configure(IApplicationBuilder app, IHostingEnvironment env)  
{  
    if (env.IsDevelopment()) { ... }  
  
    app.Run(async (context) =>  
    {  
        var msg = Configuration["Message"];  
        await context.Response.WriteAsync(msg);  
    });  
}
```

-Execute the web application.