**Static Files Middleware in ASP.NET Core Application**

In this article, I will discuss how to **Serve Static Files using Static Files Middleware in an ASP.NET Core** Web Application with Examples. Please read our previous article before proceeding to this one, where we discussed the[**wwwroot folder in the ASP.NET Core**](https://dotnettutorials.net/lesson/wwwroot-folder-asp-net-core/) Application. As part of this article, we will discuss the following pointers in detail.

1. **Where do we need to store the static files in ASP.NET Core?**
2. **What is wwwroot folder in ASP.NET Core?**
3. **How do you configure static files middleware in the ASP.NET Core Web Application?**
4. **How to use your own Webroot folder?**

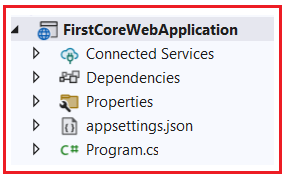
**Static Files Middleware in ASP.NET Core Web Application:**

One of the most important features of almost all web applications should be the ability to serve static files directly from the file system. Static files such as HTML, Images, CSS, and JavaScript are the important assets of a Web Application, and ASP.NET Core can serve these files directly to the clients. However, the important point you need to remember is that ASP.NET Core cannot serve these static files by default. Some configuration is required to enable ASP.NET Core to serve these static files directly. The Static Files Middleware is responsible for handling static files and making them accessible to clients.

In the ASP.NET Core Application, the default directory or location for the static files is the **wwwroot (webroot) folder,**which should be present in the project root directory. By default, this is the only place where the ASP.NET Core application can serve the static files directly. However, we can change this default behavior using the **WebApplicationOptions** Instance and **WebRootPath** property.

**Example to understand Static Files in .NET Application:**

When we create a new ASP.NET Core Application with the Empty project template, by default, you will not find the wwwroot folder. The project structure of the ASP.NET Core Web Application with the Empty Project template is shown below.

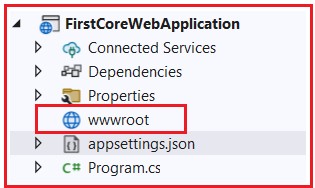


**[×](https://go.ezodn.com/ads/charity/proxy?p_id=c3302fc4-f8da-4c0d-6ea8-3a061d98fb77&d_id=118560&imp_id=2618670734660814&c_id=1084&l_id=10016&url=https%3A%2F%2Fwww.amazonconservation.org%2Ftake-action%2Fdonate%2F&ffid=1&co=IN)**

As you can see in the above image, our application does not have a folder called wwwroot. However, if you create the Project using the MVC (Model-View-Controller) Project Template, Visual Studio will create the wwwroot folder by default.

**Adding the wwwroot (webroot) Folder:**

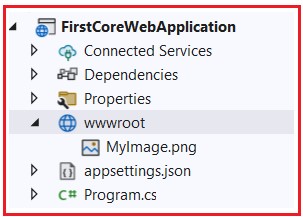
Let us first create the wwwroot Project Folder. To do so, right-click on the project, select the **add => new folder** option from the context menu, and then provide the folder name as wwwroot. Once you create the wwwroot folder, your project structure should be shown below.



Once you have created the **wwwroot** folder, add an Image file. Please download and paste the following image into the wwwroot folder and modify the image name as **MyImage.png**.



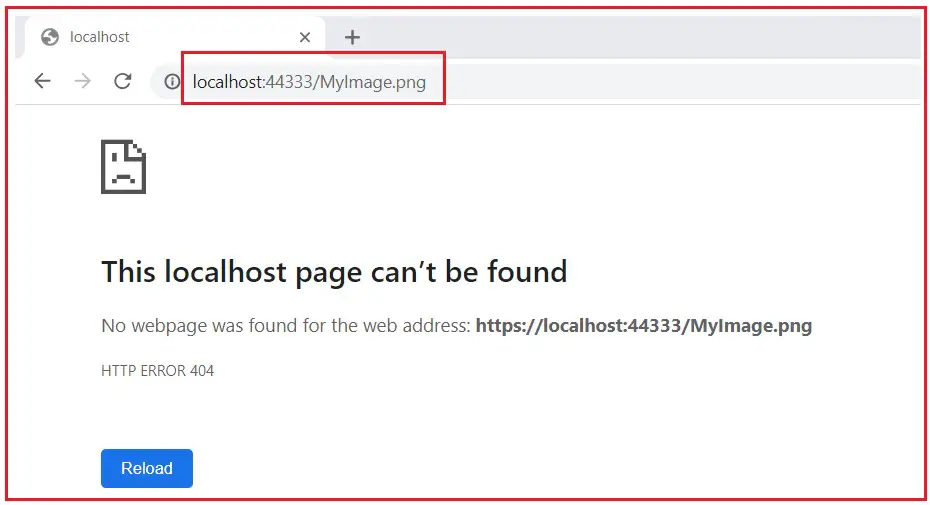
Once you save the above image, your **wwwroot** directory looks as shown below.

Advertisements

Now run the application and navigate to the following URL. You need to replace the port number on which your application is running.

**http://localhost:<portnumber>/MyImage.png**

When you navigate to the above URL, you will not get the output as expected; rather, you will get the following output.



We are not getting the output as expected because we don’t have any middleware that can serve the static files in the request processing pipeline.

**How Do We Configure the Static Files Middleware in ASP.NET Core Application?**

To handle the static resources in the ASP.NET Core Web Application, we need to configure a middleware called **UseStaticFiles()** into the application [**Request Processing Pipeline**](https://dotnettutorials.net/lesson/asp-net-core-request-processing-pipeline/). The **UseStaticFiles()** middleware is an inbuilt Middleware provided by the ASP.NET Core Framework to handle the static files in an ASP.NET Core Web Application.

Let us Modify the Main() Method of the Program class, as shown below, to register the UseStaticFiles() Middleware Component in the application’s Request Processing Pipeline. This method should be placed before any middleware that requires access to the static files (such as MVC middleware if you’re using it).

**namespace** *FirstCoreWebApplication*

**{**

**public** **class** Program

**{**

**public** **static** **void** Main**(string[]** args**)**

**{**

var builder = WebApplication.CreateBuilder**(**args**)**;

var app = builder.Build**()**;

//Adding Static Files Middleware Component to serve the static files

app.UseStaticFiles**()**;

app.MapGet**(**"/", **()** =**>** "Hello World!"**)**;

//This will Run the Application

app.Run**()**;

**}**

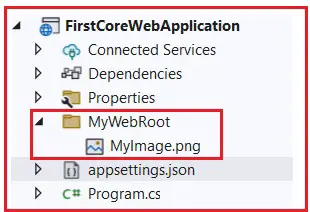
**}**

**}**

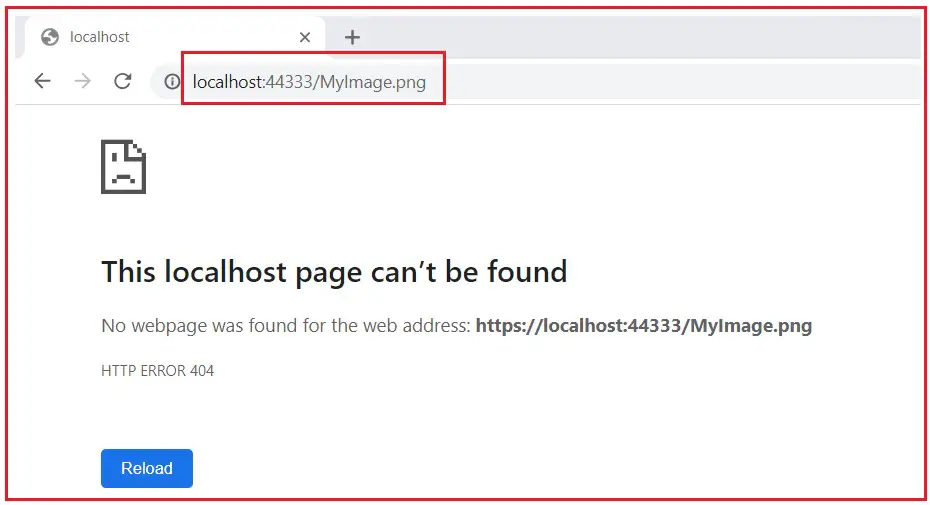
With the above changes in place, now run the application and navigate to the URL: **http://localhost:<portnumber>/MyImage.png,** and you will see the output as expected, as shown in the below image.

**How Do We Create Our Own Webroot Folder in ASP.NET Core?**

Let’s say we don’t want wwwroot as our webroot folder; instead, we want MyWebRoot as the webroot folder for our application. First, modify the wwwroot folder as MyWebRoot, and once you modify it, your project structure should be as shown below.



At this point, if you run the application, then you will not get the output as shown in the below image.



This is because, by default, the static files middleware will look for a folder named wwwroot that is not present in our application. But we don’t want wwwroot. We want the Static files middleware to look at the MyWebRoot folder to serve the static files such as CSS, Images, JS, etc. To do so, we need to tell the ASP.NET Core Framework to use MyWebRoot as the web root path. So, we need to set the WebRootPath property to MyWebRoot while creating the WebApplicationBuilder instance. Please modify the Main method of the Program class as shown below to configure the MyWebRoot folder as the webroot folder for our application.

**namespace** *FirstCoreWebApplication*

**{**

**public** **class** Program

**{**

**public** **static** **void** Main**(string[]** args**)**

**{**

//var builder = WebApplication.CreateBuilder(args);

//Setting Custom Web Root Folder

WebApplicationBuilder builder = WebApplication.CreateBuilder**(**new WebApplicationOptions

**{**

WebRootPath = "MyWebRoot"

**})**;

var app = builder.Build**()**;

//Adding Static Files Middleware Component to serve the static files

app.UseStaticFiles**()**;

app.MapGet**(**"/", **()** =**>** "Hello World!"**)**;

//This will Run the Application

app.Run**()**;

**}**

**}**

**}**

After making the above changes, run the application, and you should get the expected output, as shown in the image below.

In the next article, we will discuss [**Configuring the Default Page in the ASP.NET Core Application**](https://dotnettutorials.net/lesson/configuring-default-page-asp-net-core/)with Examples. In this article, I explain how to serve static files using the static files middleware component in the ASP.NET Core Web application. I would like to have your feedback about this article. Please post your feedback, questions, or comments about this Static Files Middleware Component in the ASP.NET Core Web Application article.

**ASP.NET Core - Static Files**

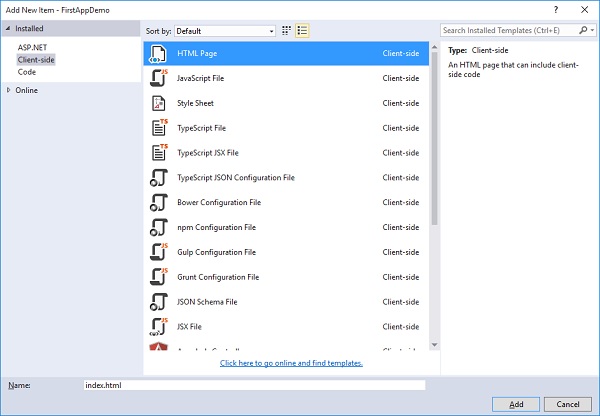
In this chapter, we will learn how to work with files. An important feature nearly every web application needs is the ability to serve up files (static files) from the file system.

* Static files like JavaScript files, images, CSS files that we have on the file system are the assets that ASP.NET Core application can serve directly to clients.
* Static files are typically located in the web root (wwwroot) folder.
* By default, that is the only place where we can serve up files directly from the file system.

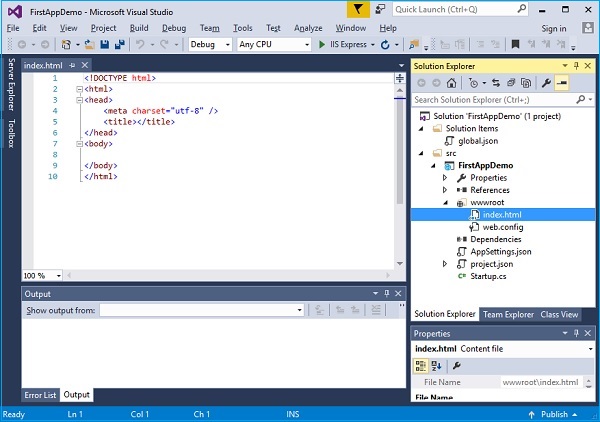
Example

Let us now take a simple example in which we will understand how we can serve those files in our application.

Here, we want to add a simple HTML file to our FirstAppDemo application and this HTML file has to go into the web root (wwwroot) folder. Right-click on wwwroot folder in the Solution Explorer and select **Add → New Item**.



In the middle pane, select the **HTML Page** and call it **index.html** and click the **Add** button.



You will see a simple **index.html**file. Let us add some simple text and title as shown below.

<!DOCTYPE html>

<html>

<head>

<meta charset="utf-8" />

<title>Welcome to ASP.NET Core</title>

</head>

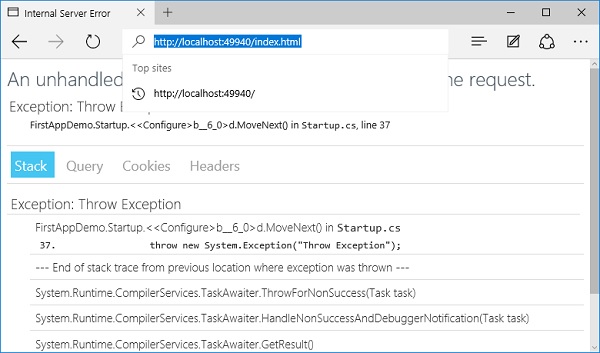
<body>

Hello, Wolrd! this message is from our first static HTML file.

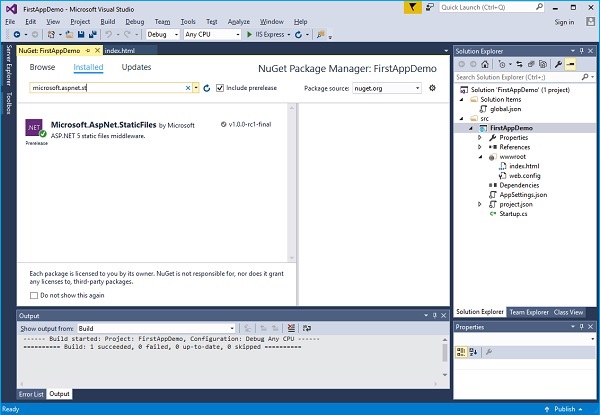
</body>

</html>

When you run your application and go to **index.html** in the browser, you will see that the **app.Run** middleware throws an exception because there is nothing currently in our application.



There is no piece of middleware that will go looking for any file on the file system to serve. To fix this issue, go to the **NuGet packages manager** by right-clicking on your project in Solution Explorer and selecting Manage NuGet Packages.



Search for **Microsoft.AspNet.StaticFiles** which will find the static files middleware. Let us install this nuget package and now we should have additional methods that we can use to register middleware inside the Configure method.

Let us add **UseStaticFiles** middle in Configure method as shown in the following program.

using Microsoft.AspNet.Builder;

using Microsoft.AspNet.Hosting;

using Microsoft.AspNet.Http;

using Microsoft.Extensions.DependencyInjection;

using Microsoft.Extensions.Configuration;

namespace FirstAppDemo {

public class Startup {

public Startup() {

var builder = new ConfigurationBuilder()

.AddJsonFile("AppSettings.json");

Configuration = builder.Build();

}

public IConfiguration Configuration { get; set; }

// This method gets called by the runtime.

// Use this method to add services to the container.

// For more information on how to configure your application,

// visit http://go.microsoft.com/fwlink/?LinkID=398940

public void ConfigureServices(IServiceCollection services) {

}

// This method gets called by the runtime.

// Use this method to configure the HTTP request pipeline.

public void Configure(IApplicationBuilder app) {

app.UseIISPlatformHandler();

app.UseDeveloperExceptionPage(); app.UseRuntimeInfoPage();

app.UseStaticFiles();

app.Run(async (context) => {

throw new System.Exception("Throw Exception");

var msg = Configuration["message"];

await context.Response.WriteAsync(msg);

});

}

// Entry point for the application.

public static void Main(string[] args) => WebApplication.Run<Startup>(args);

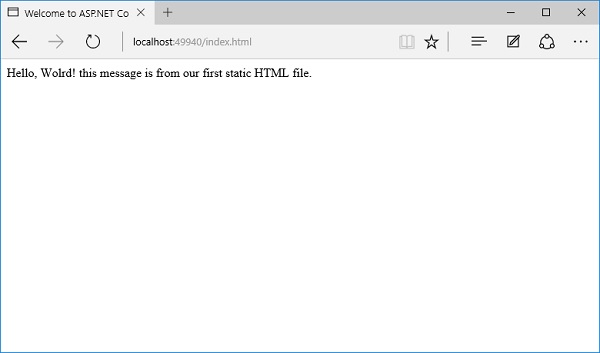
}

}

Unless you override the options and pass in some different configuration parameters, what static files will do is for a given request is to look at the **request path**. This request path is then compared to the file system and what is on the file system.

* If the static file sees a file that it can use, it will serve up that file and not call the next piece of middleware.
* If it doesn't find a matching file, then it will simply continue with the next piece of middleware.

Let us save the **Startup.cs** file and refresh your browser.



You can now see the index.html file. Anything that you put anywhere inside the wwwroot − any JavaScript file or CSS file or HTML file, you will be able to serve them up.

* Now if you want index.html to be your default file, this is a feature that IIS has always had.
* You can always give IIS a list of default files to look for. If someone came to the root of a directory or, in this case, the root of the website and if IIS found something named index.html, it would just automatically serve that file.
* Let us now start by making a few changes. First, we need to remove the forced error and then add another piece of middleware, which is UseDefaultFiles. The following is the implementation of the Configure method.

// This method gets called by the runtime.

// Use this method to configure the HTTP request pipeline.

public void Configure(IApplicationBuilder app) {

app.UseIISPlatformHandler();

app.UseDeveloperExceptionPage();

app.UseRuntimeInfoPage();

app.UseDefaultFiles();

app.UseStaticFiles();

app.Run(async (context) => {

var msg = Configuration["message"];

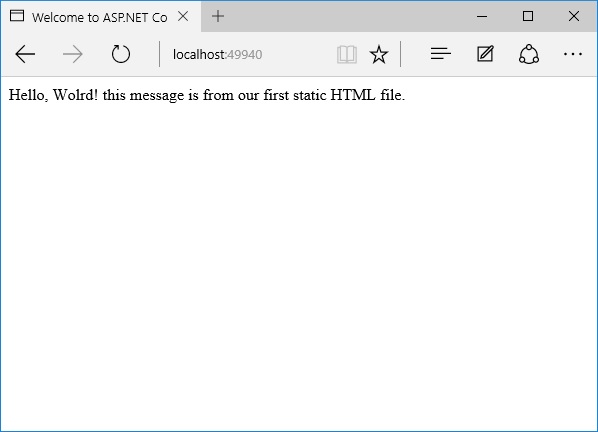
await context.Response.WriteAsync(msg);

});

}

* This piece of middleware will look at an incoming request and see if it is for the root of a directory and if there are any matching default files.
* You can override the options for this piece of middleware to tell it what are the default files to look for, but Index.html is by default one of the default files.

Let us save the **Startup.cs** file and go to the root of the web application in your browser.



You can now see that the index.html is your default file. The order in which you install the middleware is important because if you had UseDefaultFiles after UseStaticFiles, you would not get the same result.

If you are going to use UseDefaultFiles and UseStaticFiles, you might also want another piece of middleware that is inside the Microsoft.aspnet.staticfiles, NuGet package, and that is the **FileServer middleware**. This essentially includes the Default Files and the Static Files in the correct order.

// This method gets called by the runtime.

// Use this method to configure the HTTP request pipeline.

public void Configure(IApplicationBuilder app) {

app.UseIISPlatformHandler();

app.UseDeveloperExceptionPage();

app.UseRuntimeInfoPage();

app. UseFileServer();

app.Run(async (context) => {

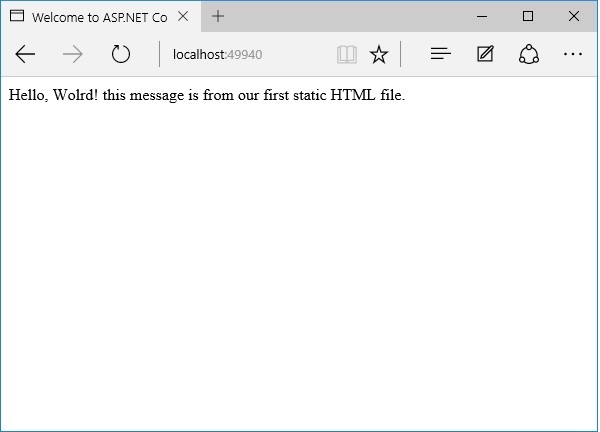
var msg = Configuration["message"];

await context.Response.WriteAsync(msg);

});

}

Let us save the **Startup.cs** file again. Once you refresh the browser, you will see the same result as shown in the following screenshot.



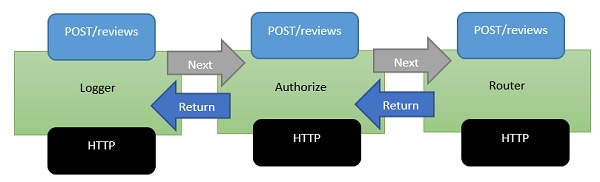
**ASP.NET Core - Middleware**

In this chapter, we will understand how to set up middleware. Middleware in ASP.NET Core controls how our application responds to HTTP requests. It can also control how our application looks when there is an error, and it is a key piece in how we authenticate and authorize a user to perform specific actions.

* Middleware are software components that are assembled into an application pipeline to handle requests and responses.
* Each component chooses whether to pass the request on to the next component in the pipeline, and can perform certain actions before and after the next component is invoked in the pipeline.
* Request delegates are used to build the request pipeline. The request delegates handle each HTTP request.
* Each piece of middleware in ASP.NET Core is an object, and each piece has a very specific, focused, and limited role.
* Ultimately, we need many pieces of middleware for an application to behave appropriately.

Let us now assume that we want to log information about every request into our application.

* In that case, the first piece of middleware that we might install into the application is a logging component.
* This logger can see everything about the incoming request, but chances are a logger is simply going to record some information and then pass along this request to the next piece of middleware.



* Middleware is a series of components present in this processing pipeline.
* The next piece of middleware that we've installed into the application is an authorizer.
* An authorizer might be looking for specific cookie or access tokens in the HTTP headers.
* If the authorizer finds a token, it allows the request to proceed. If not, perhaps the authorizer itself will respond to the request with an HTTP error code or redirect code to send the user to a login page.
* But, otherwise, the authorizer will pass the request to the next piece of middleware which is a router.
* A router looks at the URL and determines your next step of action.
* The router looks over the application for something to respond to and if the router doesn't find anything to respond to, the router itself might return a **404 Not Found error**.

Example

Let us now take a simple example to understand more about middleware. We set up the middleware in ASP.NET using the Configure method of our **Startup class**.

using Microsoft.AspNet.Builder;

using Microsoft.AspNet.Hosting;

using Microsoft.AspNet.Http;

using Microsoft.Extensions.DependencyInjection;

using Microsoft.Extensions.Configuration;

namespace FirstAppDemo {

public class Startup {

public Startup() {

var builder = new ConfigurationBuilder()

.AddJsonFile("AppSettings.json");

Configuration = builder.Build();

}

public IConfiguration Configuration { get; set; }

// This method gets called by the runtime.

// Use this method to add services to the container.

// For more information on how to configure your application,

// visit http://go.microsoft.com/fwlink/?LinkID=398940

public void ConfigureServices(IServiceCollection services) {

}

// This method gets called by the runtime.

// Use this method to configure the HTTP request pipeline.

public void Configure(IApplicationBuilder app) {

app.UseIISPlatformHandler();

app.Run(async (context) => {

var msg = Configuration["message"];

await context.Response.WriteAsync(msg);

});

}

// Entry point for the application.

public static void Main(string[] args) => WebApplication.Run<Startup>(args);

}

}

Inside the **Configure()** method, we will invoke the extension methods on the IApplicationBuilder interface to add middleware.

There are two pieces of middleware in a new empty project by default −

* IISPlatformHandler
* Middleware registered with app.Run

IISPlatformHandler

**IISPlatformHandler** allows us to work with Windows authentication. It will look at every incoming request and see if there is any Windows identity information associated with that request and then it calls the next piece of middleware.

Middleware registered with app.Run

The next piece of middleware in this case is a piece of middleware registered with **app.Run**. The Run method allows us to pass in another method, which we can use to process every single response. Run is not something that you will see very often, it is something that we call a terminal piece of middleware.

Middleware that you register with Run will never have the opportunity to call another piece of middleware, all it does is receive a request, and then it has to produce some sort of response.

You also get access to a Response object and one of the things you can do with a Response object is to write a string.

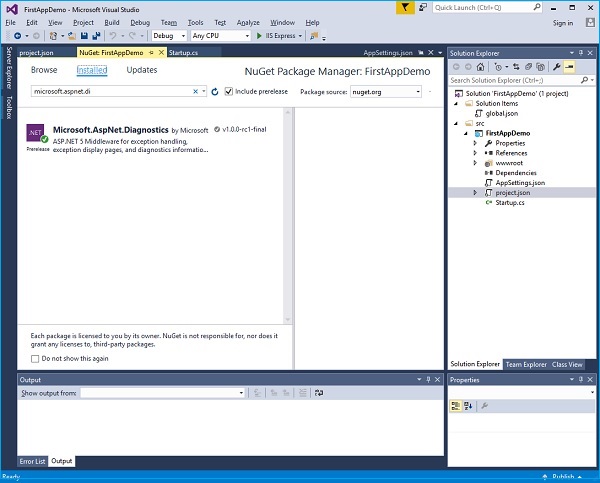
If you want to register another piece of middleware after app.Run, that piece of middleware would never be called because, again, Run is a terminal piece of middleware. It will never call into the next piece of middleware.

How to Add another Middleware

Let us proceed with the following steps to add another middleware −

**Step 1** − To add another middleware, right-click on project and select Manage NuGet Packages.

**Step 2** − Search for **Microsoft.aspnet.diagnostics** that is actually ASP.NET Core middleware for exception handling, exception display pages, and diagnostics information. This particular package contains many different pieces of middleware that we can use.



**Step 3** − Install that package if it is not installed in your project.

**Step 4** − Let us now go to the **Configure()** method and invoke **app.UseWelcomePage** middleware.

// This method gets called by the runtime.

// Use this method to configure the HTTP request pipeline.

public void Configure(IApplicationBuilder app) {

app.UseIISPlatformHandler();

app.UseWelcomePage();

app.Run(async (context) => {

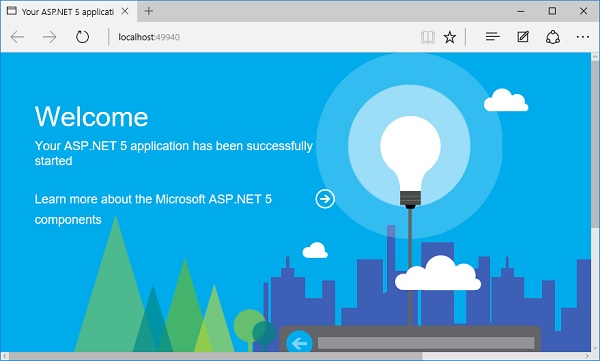
var msg = Configuration["message"];

await context.Response.WriteAsync(msg);

});

}

**Step 5** − Run your application and you will see the following welcome screen.



This Welcome screen might not be as useful.

**Step 6** − Let us try something else that might be a little more useful. Instead of using the Welcome page, we will use the **RuntimeInfoPage**.

// This method gets called by the runtime.

// Use this method to configure the HTTP request pipeline.

public void Configure(IApplicationBuilder app) {

app.UseIISPlatformHandler();

app.UseRuntimeInfoPage();

app.Run(async (context) => {

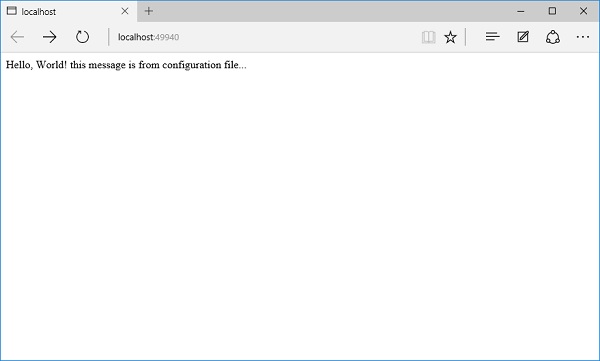
var msg = Configuration["message"];

await context.Response.WriteAsync(msg);

});

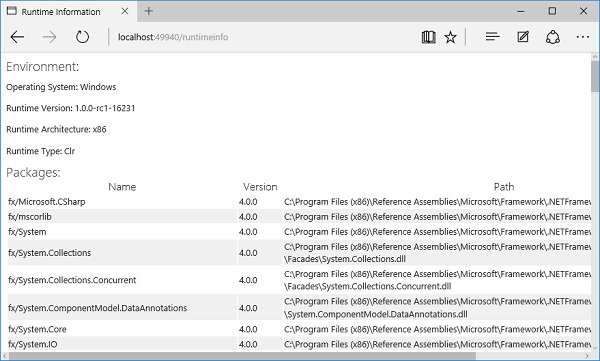
}

**Step 7** − Save your **Startup.cs** page and refresh your browser and you will see the following page.



This **RuntimeInfoPage** is a middleware that will only respond to requests that come in for a specific URL. If the incoming request does not match that URL, this piece of middleware just lets the request pass through to the next piece of middleware. The request will pass through the IISPlatformHandler middleware, then go to the UseRuntimeInfoPage middleware. It is not going to create a response, So it will go to our app.Run and display the string.

**Step 8** − Let us add “/**runtimeinfo**” at the end of your URL. You will now see a page that is produced by that runtime info page middleware.



You will now see a response that gives you some information about your runtime environment such as the Operating System, runtime version, architecture, type and all the packages that you are using etc.