

.NET Core



Mohamed ELshafei

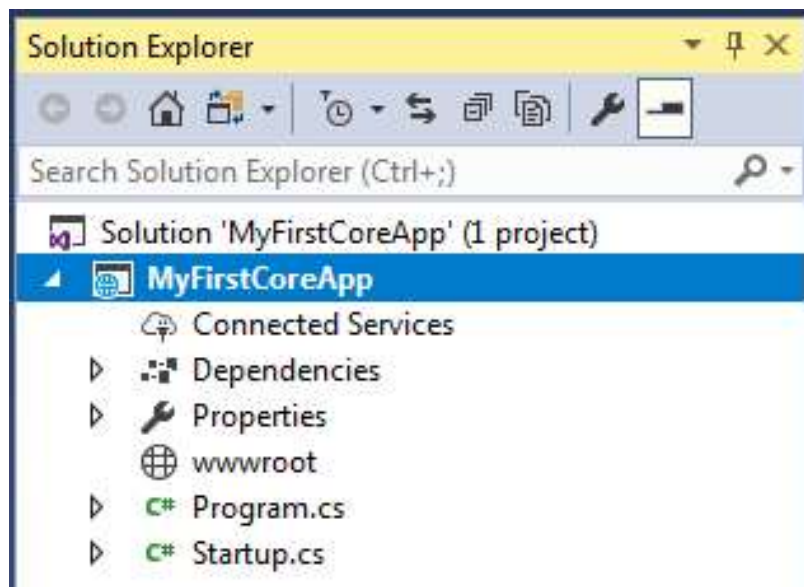
Why ASP.NET Core?

- **Supports Multiple Platforms:** ASP.NET Core applications can run on Windows, Linux, and Mac. So you don't need to build different apps for different platforms using different frameworks.
- **Fast** - 8x Faster than Node; 3x Faster than Go .. ASP.NET Core no longer depends on System.Web.dll for browser-server communication. ASP.NET Core allows us to include packages which we need for our application. This reduces the request pipeline and improves the performance and scalability.
- **IoC Container:** It includes built-in IoC container for automatic dependency injection which makes it maintainable and testable.
- **Open-Source** with Contributions – even the docs

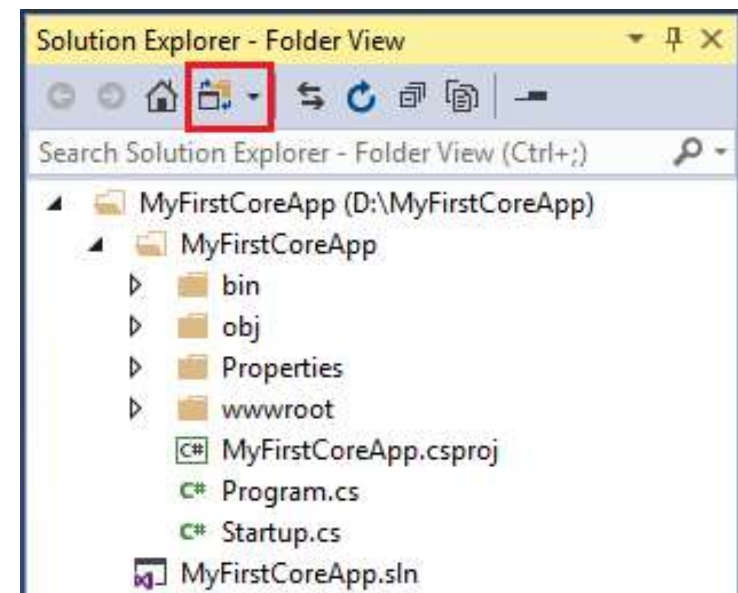
Why ASP.NET Core?

- **Integration with Modern UI Frameworks:** It allows you to use and manage modern UI frameworks such as AngularJS, ReactJS, Umber, Bootstrap etc. using Bower (a package manager for the web).
- **Smaller Deployment Footprint:** ASP.NET Core application runs on .NET Core which is smaller than full .NET Framework. So, the application which uses only a part of .NET CoreFX will have smaller deployment size. This reduces the deployment foot print.
- **Hosting:** ASP.NET Core web application can be hosted on multiple platforms with any web server such as IIS, Apache etc. It is not dependent only on IIS as a standard .NET Framework.
- **Choice of Tools and Editors** – Any Text Editor plus CLI, or Visual Studio

ASP.NET Core Project Structure (Empty Project)



We can change it to folder view by clicking **Solution and Folders** icon.



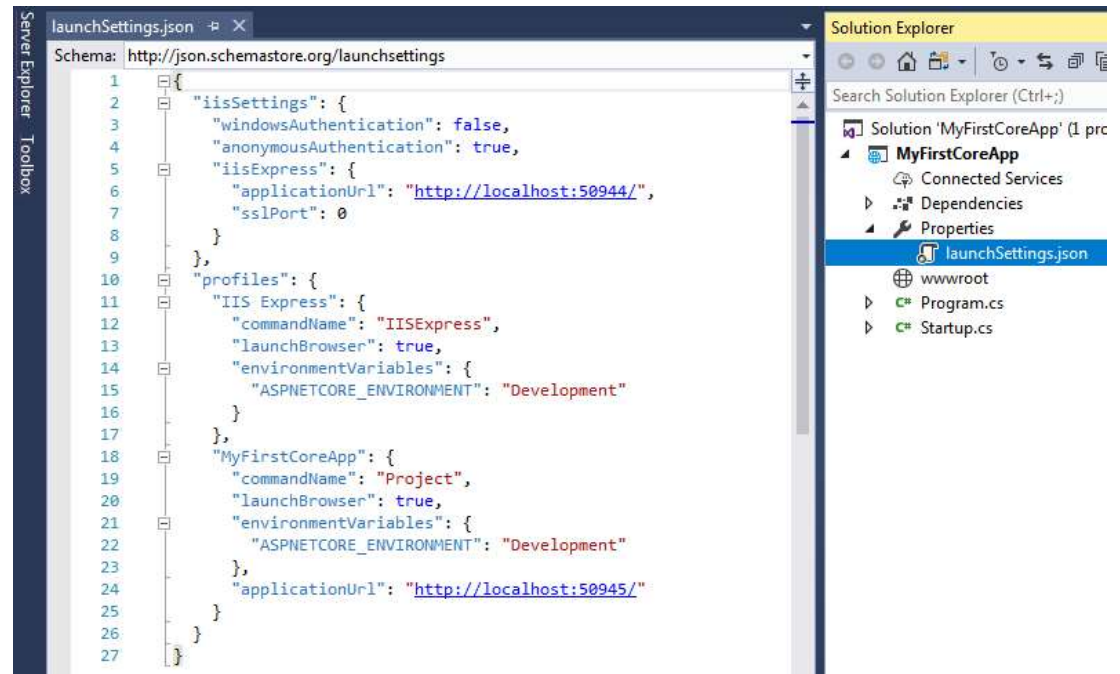
ASP.NET Core project files and folders are synchronized with physical files and folders.

Properties

- The Properties node includes launchSettings.json file which includes Visual Studio profiles of debug settings. The following is a default launchSettings.json file.

Edit debug settings from the debug tab of project properties.

Right click on the project -> select Properties -> click Debug tab.



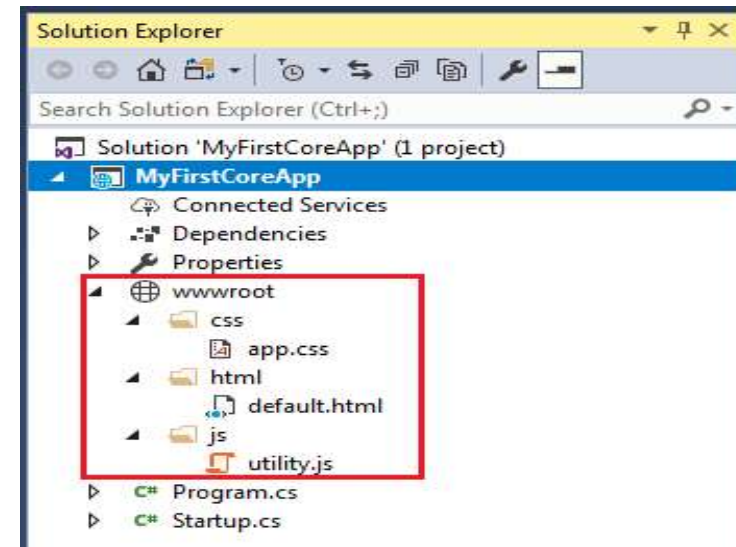
The screenshot shows the Visual Studio interface. On the left, the 'Server Explorer' and 'Toolbox' are visible. The main editor displays the 'launchSettings.json' file with the following content:

```
1 {
2   "iisSettings": {
3     "windowsAuthentication": false,
4     "anonymousAuthentication": true,
5     "iisExpress": {
6       "applicationUrl": "http://localhost:50944/",
7       "sslPort": 0
8     }
9   },
10  "profiles": {
11    "IIS Express": {
12      "commandName": "IISExpress",
13      "launchBrowser": true,
14      "environmentVariables": {
15        "ASPNETCORE_ENVIRONMENT": "Development"
16      }
17    },
18    "MyFirstCoreApp": {
19      "commandName": "Project",
20      "launchBrowser": true,
21      "environmentVariables": {
22        "ASPNETCORE_ENVIRONMENT": "Development"
23      },
24      "applicationUrl": "http://localhost:50945/"
25    }
26  }
27 }
```

On the right, the 'Solution Explorer' shows the project structure for 'MyFirstCoreApp'. The 'Properties' node is expanded, and 'launchSettings.json' is selected.

wwwroot

- **wwwroot** folder is treated as a web root folder. Static files can be stored in any folder under the web root and accessed with a relative path to that root.
- only files that are in wwwroot folder can be served over an http request. All other files are blocked and cannot be served by default.



wwwroot

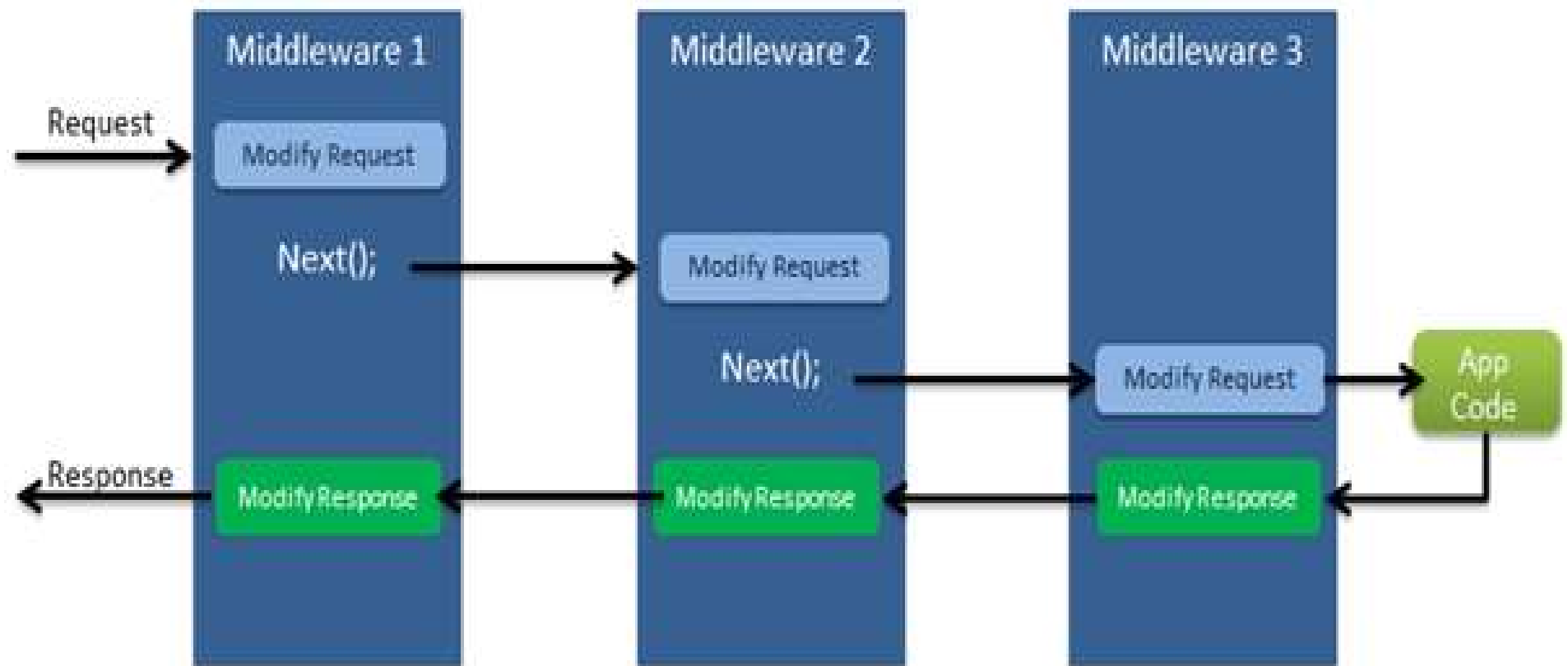
- Generally, there should be separate folders for the different types of static files such as JavaScript, CSS, Images, library scripts etc
- You can access static files with base URL and file name.
For example, we can access site.css file in the css folder by *http://localhost:<port>/css/app.css*.
- you need to include a middleware for serving static files in the Configure method of Startup.cs.

Middleware “Built-in and Custom”

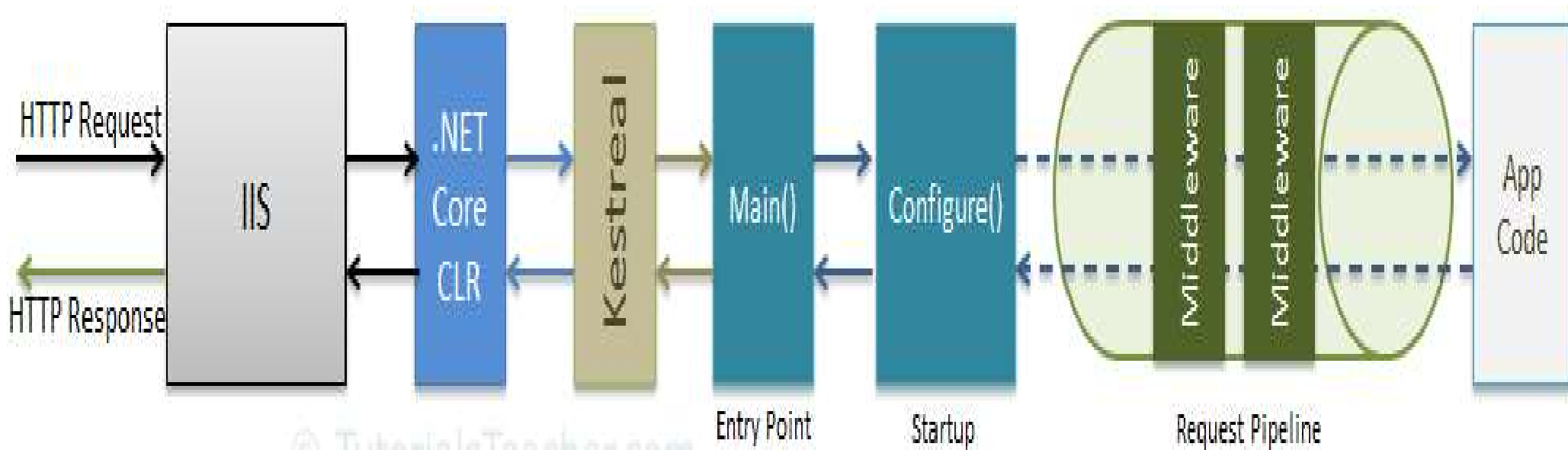
A middleware is nothing but a component (class) which is executed on every request in ASP.NET Core application.

It can be either framework provided middleware, added via NuGet or your own custom middleware.

- Each component in the pipeline is a request delegate.
- Each delegate can invoke the next component in the chain, or short-circuit, returning back up the call chain



ASP.NET Core Request Processing



1.Exception/error handling

- When the app runs in the ***Development*** environment:
 - Developer Exception Page Middleware ([UseDeveloperExceptionPage](#)) reports app runtime errors.
 - Database Error Page Middleware reports database runtime errors.
- When the app runs in the ***Production*** environment:
 - Exception Handler Middleware ([UseExceptionHandler](#)) catches exceptions thrown in the following middlewares.
 - HTTP Strict Transport Security Protocol (HSTS) Middleware ([UseHsts](#)) adds the Strict-Transport-Security header.

2.HTTPS Redirection Middleware ([UseHttpsRedirection](#)) redirects HTTP requests to HTTPS.

3.Static File Middleware ([UseStaticFiles](#)) returns static files and short-circuits further request processing.

4.Cookie Policy Middleware ([UseCookiePolicy](#)) conforms the app to the EU General Data Protection Regulation (GDPR) regulations.

5.Routing Middleware (UseRouting) to route requests.

6.Authentication Middleware ([UseAuthentication](#)) attempts to authenticate the user before they're allowed access to secure resources.

7.Authorization Middleware (UseAuthorization) authorizes a user to access secure resources.

8.Session Middleware ([UseSession](#)) establishes and maintains session state. If the app uses session state, call Session Middleware after Cookie Policy Middleware and before MVC Middleware.

9.Endpoint Routing Middleware (UseEndpoints with MapRazorPages) to add Razor Pages endpoints to the request pipeline

appsettings.json - project.json

- The project.json file includes settings for an application which is used by .NET Core CLR (runtime) to determine how to run an application.



```
project.json  X project.json
Schéma : http://json.schemastore.org/project

{
  "version": "1.0.0-*",
  "buildOptions": {
    "emitEntryPoint": true
  },
  "dependencies": {
  },
  "frameworks": {
    "netcoreapp1.0": {
      "imports": "dnxcore50",
      "dependencies": {
        "Microsoft.NETCore.App": {
          "type": "platform",
          "version": "1.0.1"
        }
      }
    }
  }
}
```

Attribute routing with Http[Verb] attributes.

- Attribute routing can also make use of the *Http[Verb]* attributes such as *HttpPostAttribute*. All of these attributes can accept a route template.

```
[HttpGet("/products")]
public IActionResult ListProducts()
{
    // ...
}

[HttpPost("/products")]
public IActionResult CreateProduct(...)
{
    // ...
}
```

Token replacement in route templates

- For convenience, attribute routes support token replacement by enclosing a token in square-braces ([,]).

- No route Prefix.

```
[Route("[controller]/[action]")]
public class ProductsController : Controller
{
    [HttpGet] // Matches '/Products/List'
    public IActionResult List() {
        // ...
    }

    [HttpGet("{id}")] // Matches '/Products/Edit/{id}'
    public IActionResult Edit(int id) {
        // ...
    }
}
```

Multiple Routes

```
[Route("[controller]")]
public class ProductsController : Controller
{
    [Route("")] // Matches 'Products'
    [Route("Index")] // Matches 'Products/Index'
    public IActionResult Index()
}
```

```
[Route("Store")]
[Route("[controller]")]
public class ProductsController : Controller
{
    [HttpPost("Buy")] // Matches 'Products/Buy' and 'Store/Buy'
    [HttpPost("Checkout")] // Matches 'Products/Checkout' and 'Store/Checkout'
    public IActionResult Buy()
}
```

```
[Route("api/[controller]")]
public class ProductsController : Controller
{
    [HttpPut("Buy")] // Matches PUT 'api/Products/Buy'
    [HttpPost("Checkout")] // Matches POST 'api/Products/Checkout'
    public IActionResult Buy()
}
```


Specifying attribute route optional parameters, default values, and constraints

- {id:int} integar constraint.
- {id:int?} optional Parameter
- {id:int=3} Default Value

```
[HttpPost("product/{id:int}")]  
public IActionResult ShowProduct(int id)  
{  
    // ...  
}
```

Model binding

- By default, model binding gets data in the form of key-value pairs from the following sources in an HTTP request:
 - Form fields
 - The request body (For controllers that have the [ApiController] attribute.)
 - Route data
 - Query string parameters
 - Uploaded files
- For each target parameter or property, the sources are scanned in the order indicated in the preceding list.
- There are a few exceptions: Route data and query string values are used only for simple types.

Binding source parameter inference

A binding source attribute defines the location at which an action parameter's value is found. The following binding source attributes exist:

[FromBody]: Request body

[FromForm]: Form data in the request body

[FromHeader]: Request header

[FromQuery]: Request query string parameter

[FromRoute]: Route data from the current request

Controller action return types

- [Specific type](#)
- [IActionResult](#)
- [ActionResult<T>](#)
- [HttpResults](#)

Controller action return types

- **Specific type**

The most basic action returns a primitive or complex data type, for example, string or a custom object

- **ActionResult**

The ActionResult return types represent various HTTP status codes.

- **ActionResult<T>**

ASP.NET Core includes the ActionResult<T> return type for web API controller actions. It enables returning a type deriving from ActionResult or return a specific type.

- **HttpResults**