**MVC**

**Introduction**  
View Engine renders the HTML to the browser. The view engine templates have a different syntax than the implementation. By default ASP.Net MVC supports ASPX and the Razor View Engine. There are many more third-party view engines, like Spark, Nhaml and so on also available for MVC. We can also write our own view engine.  
  
**ASPX View Engine**  
  
The syntax used for writing a view with the ASPX View Engine is the same as the syntax used in ASP.Net web forms. The file extensions are also the same as for ASP.NET web forms (like .aspx, .ascx, .master). This view engine is the default view engine for MVC 1.0 and MVC 2.0. Implementing the unit testing framework with the ASPX View Engine is very difficult. ASPX uses "<%= %>" or "<%: %>" to render server-side content. We can choose any language with the CodeDom provider. There are on demand or precompiled views supported by an ASPX View Engine. The ASPX View Engine is also known as the Web Form View Engine.  
  
Loop and condition example with ASPX View Engine:

<ul>  
    <%foreach (var item in Products)  
        {  %>  
             <% if (item.IsInStock)  
                  { %>  
                         <p><%=item.ProductName%> is in stock</p>  
                   <% }  
                 else  
                  { %>  
                         <p><%=item.ProductName%> is not in stock</p>  
                  <% } %>  
    <%} %>  
</ul>  
  
**Razor View Engine**  
  
The Razor View Engine is an advanced view engine, available with MVC 3.0 and later versions. Razor uses the "@" character instead of "<% %>" as used by the ASPX View Engine. Razor does not require the code block to be closed, the Razor View Engine parsed itself and it is able to decide during run time that it is a presentation element (content) and that it is a code element. The Razor View Engine is compatible with a unit testing framework. The Razor template does not require the controller or webserver to host it, so views written in Razor are fully testable. The file extension of a Razor view is cshtml (for C#) and vbhtml (for VB.NET). By default all text from the @ expression is HTML encoded. Razor is not a new language. It is easy to learn. The main advantage of Razor, is that there is less transition between HTML and code because Razor provides an optimized syntax to generate HTML using a code focused templating approach.  
  
Loop and condition example with Razor View Engine:  
  
<ul>  
    @foreach (var item in Products)  
    {  
               @if(item.IsinStock)  
               {    
                   @item.ProductName is in stock  
               } else {  
                   @item.ProductName is in stock  
               }  
    }  
</ul>  
  
One of the disadvantages of Razor is, it is not supported by visual editors like Dream Viewer.

**Layout /MasterPage**

* In Razor View Engine we use Layouts.
* In ASPX View Engine we use masterPages.

**PartialPage /WebUserControl**

* In Razor View Engine we use PartialPage.
* In ASPX View Engine we use WebUserControls.

**Extension**

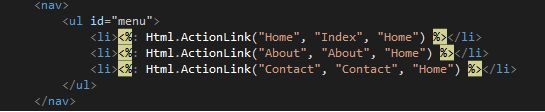
* Razor View Engine has .cshtml (with C#) and .vbhtml (with VB) extension for views, Layout and Partial views.
* ASPX View Engine has a similar extension as in a simple web application like .aspx for the views, .acsx for UserControls and .master for Master Pages.

**Performance**

* Razor Engine is a little slow compared to Aspx Engine.
* Aspx Engine is faster compared to Razor Engine.

**Syntax**

* ‘@’ symbol uses in Razor Engine to write the code. @Html.ActionLink("Login", "LoginView")
* ‘<%:’ delimiters use as starting point and ‘ %>’ use as ending point. You can write the code between them in ASPX Engine.

<%: Html.ActionLink("Login ", " LoginView ") %>   
  


**Cross-Site Scripting Attacks**

* Razor Engine prevents Cross-Site Scripting Attacks, in other words it encodes the script or HTML tags like <,> before rendering to view.
* ASPX Engine does not prevent Cross-Site Scripting Attacks, in other words any script saved in the database will be fired while rendering the page.

**Advantages of Razor View Engine**

* Easy to Learn: Razor is easy to learn. We can also use our existing HTML skills.
* It is Compact, Expressive, and Fluid. Razor helps us to minimize the coding and provide us a fast and fluid coding work flow.
* The parser (available with Razor) is smart enough. It is also able to decide at run time what is a code element and what is a content element.  
    
  For example, in the following code the @ character is also part of an email address, but Razor is smart enough to identify which is code and which is static content.  
    
  <p>  
          please contact to abc@gmail.com to more information  
         Current Date time : @DateTime.Now  
  </p>
* Razor is not a new language but it is markup so that we can also use Razor with any language like C# and VB.
* Razor also supports the concept of layout pages (the same as Master Pages in ASPX View Engine), that allows us to define a common site template, in other words a common look and feel across all the pages within a web site/application.
* Razor does not require any special tool to write markup. We can also write our markup code with any old plain text editor like Notepad.
* The Razor View Engine is designed such that it also supports unit test views without requiring a controller and web server. This can be hosted in any unit project. There is no special application domain required.
* ASP.NET MVC has HTML helpers that are methods that can be invoked within a code block. All existing HTML extension methods can be used with a Razor View Engine without any code changes.
* The code looks clean.
* Powerful built-in validation of markup that helps us to avoid unwanted runtime exceptions due to errors in the view.
* The Razor View Engine has the section concept that is equivalent to content placeholders in the ASPX View Engine and that can be optional.
* The @model directive provides a cleaner and more concise way to define a strongly typed model.  
    
  Example:  
  @model List<MyMVCapplication.EmployeeMaster>  
  or  
  @model MyMVCapplication.EmployeeDetails

**Razor View Engine VS ASPX View Engine**

|  |  |
| --- | --- |
| **Razor View Engine** | **ASPX View Engine (Web form view engine)** |
| The namespace used by the Razor View Engine is System.Web.Razor | The namespace used by the ASPX View Engine is System.Web.Mvc.WebFormViewEngine |
| The file extensions used by the Razor View Engine are different from a web form view engine. It uses cshtml with C# and vbhtml with vb for views, partial view, templates and layout pages. | The file extensions used by the Web Form View Engines are like ASP.Net web forms. It uses the ASPX extension to view the aspc extension for partial views or User Controls or templates and master extensions for layout/master pages. |
| The Razor View Engine is an advanced view engine that was introduced with MVC 3.0. This is not a new language but it is markup. | A web form view engine is the default view engine and available from the beginning of MVC |
| Razor has a syntax that is very compact and helps us to reduce typing. | The web form view engine has syntax that is the same as an ASP.Net forms application. |
| The Razor View Engine uses @ to render server-side content. | The ASPX/web form view engine uses "<%= %>" or "<%: %>" to render server-side content. |
| By default all text from an @ expression is HTML encoded. | There is a different syntax ("<%: %>") to make text HTML encoded. |
| Razor does not require the code block to be closed, the Razor View Engine parses itself and it is able to decide at runtime which is a content element and which is a code element. | A web form view engine requires the code block to be closed properly otherwise it throws a runtime exception. |
| The Razor View Engine prevents Cross Site Scripting (XSS) attacks by encoding the script or HTML tags before rendering to the view. | A web form View engine does not prevent Cross Site Scripting (XSS) attack. |
| The Razor Engine supports Test Driven Development (TDD). | Web Form view engine does not support Test Driven Development (TDD) because it depends on the System.Web.UI.Page class to make the testing complex. |
| Razor uses "@\* … \*@" for multiline comments. | The ASPX View Engine uses "<!--...-->" for markup and "/\* … \*/" for C# code. |
| There is only three transition characters with the Razor View Engine. | There are only three transition characters with the Razor View Engine. |
| The Razor View Engine is a bit slower than the ASPX View Engine. | |

**ViewData vs ViewBag vs TempData vs Session**

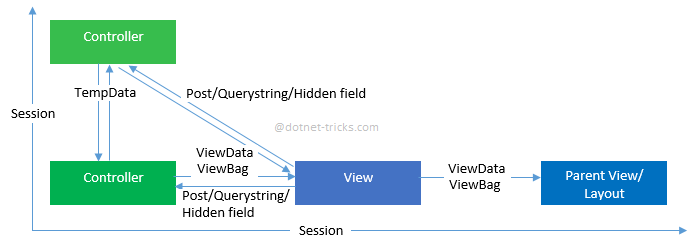
In ASP.NET MVC there are three ways - ViewData, ViewBag and TempData to pass data from controller to view and in next request. Like WebForm, you can also use Session to persist data during a user session. Now question is that when to use ViewData, VieBag, TempData and Session. Each of them has its own importance. In this article, I am trying to explain the differences among these four.

## ViewData

1. ViewData is a dictionary object that is derived from ViewDataDictionary class.
2. public ViewDataDictionary ViewData { get; set; }
3. ViewData is a property of ControllerBase class.
4. ViewData is used to pass data from controller to corresponding view.
5. It’s life lies only during the current request.
6. If redirection occurs then it’s value becomes null.
7. It’s required typecasting for getting data and check for null values to avoid error.

## ViewBag

1. ViewBag is a dynamic property that takes advantage of the new dynamic features in C# 4.0.
2. Basically it is a wrapper around the ViewData and also used to pass data from controller to corresponding view.
3. public Object ViewBag { get; }
4. ViewBag is a property of ControllerBase class.
5. It’s life also lies only during the current request.
6. If redirection occurs then it’s value becomes null.
7. It doesn’t required typecasting for getting data.



## TempData

1. TempData is a dictionary object that is derived from TempDataDictionary class and stored in short lives session.
2. public TempDataDictionary TempData { get; set; }
3. TempData is a property of ControllerBase class.
4. TempData is used to pass data from current request to subsequent request (means redirecting from one page to another).
5. It’s life is very short and lies only till the target view is fully loaded.
6. It’s required typecasting for getting data and check for null values to avoid error.
7. It is used to store only one time messages like error messages, validation messages. To persist data with TempData refer this article: [Persisting Data with TempData](http://www.dotnettricks.com/learn/mvc/persisting-data-with-tempdata)

## Session

1. In ASP.NET MVC, Session is a property of Controller class whose type is HttpSessionStateBase.
2. public HttpSessionStateBase Session { get; }
3. Session is also used to pass data within the ASP.NET MVC application and Unlike TempData, it persists for its expiration time (by default session expiration time is 20 minutes but it can be increased).
4. Session is valid for all requests, not for a single redirect.
5. It’s also required typecasting for getting data and check for null values to avoid error.

##### **Summary**

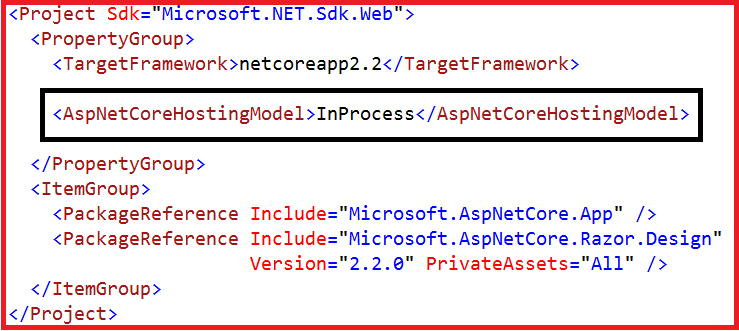
In this article I try to explain the difference between ViewData, ViewBag and TempData. I hope you will refer this article for your need. I would like to have feedback from my blog readers. Please post your feedback, question, or comments about this article.

**Hosting**

##### **ASP.NET Core InProcess Hosting**

Let first have a look on in Process Hosting before proceeding to Out Of Process Hosting

As we already discussed, to configure the InProcess hosting in ASP.NET Core application, you need to add the **<AspNetCoreHostingModel>** element to the application’s project file with a value of InProcess as shown below



In ASP.NET Core, with InProcess Hosting Model our application is going to be hosted in the IIS worker process (i.e. w3wp.exe in case of IIS server or iisexpress.exe if the hosting server is IISExpress). The most important point that you need to remember is we have only one web server i.e. IIS Server in case of InProcess hosting which is going to host our application as shown in the below image.

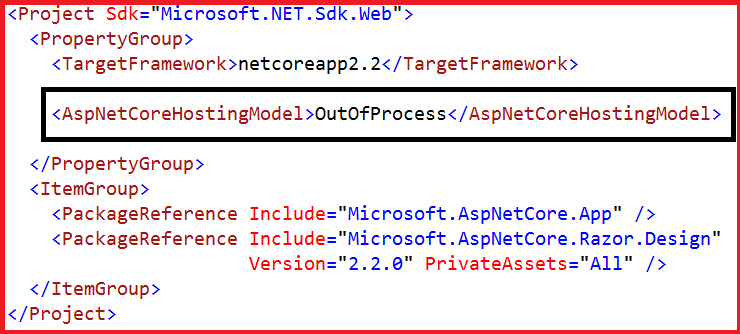
##### **InProcess Hosting in ASP.NET Core**

##### **How to Configure the OutofProcess Hosting in ASP.NET Core Application?**

We can configure the Out Of Process Hosting in two ways in ASP.NET Core.

###### **Way1:**

In this case, you just need to add the **<AspNetCoreHostingModel>** element to the applications project file with a value of **OutOfProcess** as shown below.



###### **Way2:**

The default hosting in ASP.NET Core is OutOfProcess Hosting. That means if you remove the **<AspNetCoreHostingModel>** element from the application’s project file, then by default OutOfProcess hosting will be used.

#### ****What is Out of Process Hosting in ASP.NET Core?****

In the case of the ASP.NET Core OutOfProcess Hosting Model, there are two web servers.

1. **An internal webserver which is the Kestrel web Server**
2. **And an external web server which can be IIS, Apache, and Nginx.**

The most important point that you need to keep in mind is depending on how you are running your application with the OutOfProcess hosting model, the external web server may or may not be used.

## Hosting models

### In-process hosting model

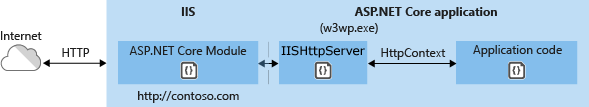
Using in-process hosting, an ASP.NET Core app runs in the same process as its IIS worker process. In-process hosting provides improved performance over out-of-process hosting because requests aren't proxied over the loopback adapter, a network interface that returns outgoing network traffic back to the same machine. IIS handles process management with the [Windows Process Activation Service (WAS)](https://docs.microsoft.com/en-us/iis/manage/provisioning-and-managing-iis/features-of-the-windows-process-activation-service-was).

The [ASP.NET Core Module](https://docs.microsoft.com/en-us/aspnet/core/host-and-deploy/aspnet-core-module?view=aspnetcore-3.1):

* Performs app initialization.
  + Loads the [CoreCLR](https://docs.microsoft.com/en-us/dotnet/standard/glossary" \l "coreclr).
  + Calls Program.Main.
* Handles the lifetime of the IIS native request.

The in-process hosting model isn't supported for ASP.NET Core apps that target the .NET Framework.

The following diagram illustrates the relationship between IIS, the ASP.NET Core Module, and an app hosted in-process:



A request arrives from the web to the kernel-mode HTTP.sys driver. The driver routes the native request to IIS on the website's configured port, usually 80 (HTTP) or 443 (HTTPS). The ASP.NET Core Module receives the native request and passes it to IIS HTTP Server (IISHttpServer). IIS HTTP Server is an in-process server implementation for IIS that converts the request from native to managed.

After the IIS HTTP Server processes the request, the request is pushed into the ASP.NET Core middleware pipeline. The middleware pipeline handles the request and passes it on as an HttpContext instance to the app's logic. The app's response is passed back to IIS through IIS HTTP Server. IIS sends the response to the client that initiated the request.

In-process hosting is opt-in for existing apps, but [dotnet new](https://docs.microsoft.com/en-us/dotnet/core/tools/dotnet-new) templates default to the in-process hosting model for all IIS and IIS Express scenarios.

CreateDefaultBuilder adds an [IServer](https://docs.microsoft.com/en-us/dotnet/api/microsoft.aspnetcore.hosting.server.iserver) instance by calling the [UseIIS](https://docs.microsoft.com/en-us/dotnet/api/microsoft.aspnetcore.hosting.webhostbuilderiisextensions.useiis) method to boot the [CoreCLR](https://docs.microsoft.com/en-us/dotnet/standard/glossary" \l "coreclr) and host the app inside of the IIS worker process (w3wp.exe or iisexpress.exe). Performance tests indicate that hosting a .NET Core app in-process delivers significantly higher request throughput compared to hosting the app out-of-process and proxying requests to [Kestrel](https://docs.microsoft.com/en-us/aspnet/core/fundamentals/servers/kestrel?view=aspnetcore-3.1) server.

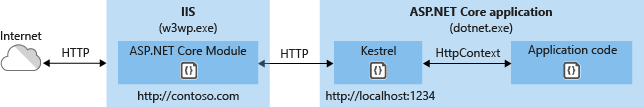
**Note**

Apps published as a single file executable can't be loaded by the in-process hosting model.

### Out-of-process hosting model

Because ASP.NET Core apps run in a process separate from the IIS worker process, the ASP.NET Core Module handles process management. The module starts the process for the ASP.NET Core app when the first request arrives and restarts the app if it shuts down or crashes. This is essentially the same behavior as seen with apps that run in-process that are managed by the [Windows Process Activation Service (WAS)](https://docs.microsoft.com/en-us/iis/manage/provisioning-and-managing-iis/features-of-the-windows-process-activation-service-was).

The following diagram illustrates the relationship between IIS, the ASP.NET Core Module, and an app hosted out-of-process:



Requests arrive from the web to the kernel-mode HTTP.sys driver. The driver routes the requests to IIS on the website's configured port, usually 80 (HTTP) or 443 (HTTPS). The module forwards the requests to Kestrel on a random port for the app, which isn't port 80 or 443.

The module specifies the port via an environment variable at startup, and the [UseIISIntegration](https://docs.microsoft.com/en-us/dotnet/api/microsoft.aspnetcore.hosting.webhostbuilderiisextensions.useiisintegration) extension configures the server to listen on http://localhost:{PORT}. Additional checks are performed, and requests that don't originate from the module are rejected. The module doesn't support HTTPS forwarding, so requests are forwarded over HTTP even if received by IIS over HTTPS.

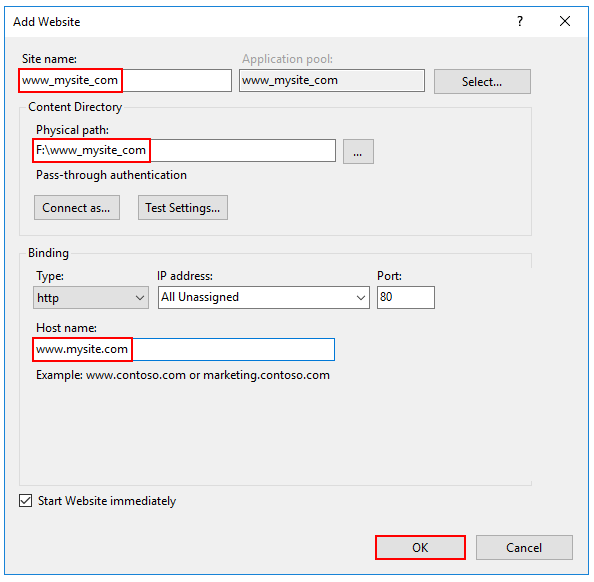
After Kestrel picks up the request from the module, the request is pushed into the ASP.NET Core middleware pipeline. The middleware pipeline handles the request and passes it on as an HttpContext instance to the app's logic. Middleware added by IIS Integration updates the scheme, remote IP, and pathbase to account for forwarding the request to Kestrel. The app's response is passed back to IIS, which pushes it back out to the HTTP client that initiated the request.

For ASP.NET Core Module configuration guidance, see [ASP.NET Core Module](https://docs.microsoft.com/en-us/aspnet/core/host-and-deploy/aspnet-core-module?view=aspnetcore-3.1).

For more information on hosting, see [Host in ASP.NET Core](https://docs.microsoft.com/en-us/aspnet/core/fundamentals/?view=aspnetcore-3.1#host).

## Create the IIS site

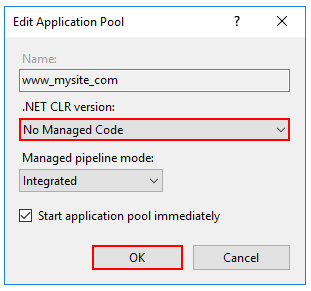
1. On the hosting system, create a folder to contain the app's published folders and files. In a following step, the folder's path is provided to IIS as the physical path to the app. For more information on an app's deployment folder and file layout, see [ASP.NET Core directory structure](https://docs.microsoft.com/en-us/aspnet/core/host-and-deploy/directory-structure?view=aspnetcore-3.1).
2. In IIS Manager, open the server's node in the **Connections** panel. Right-click the **Sites** folder. Select **Add Website** from the contextual menu.
3. Provide a **Site name** and set the **Physical path** to the app's deployment folder. Provide the **Binding** configuration and create the website by selecting **OK**:



**Warning**

Top-level wildcard bindings (http://\*:80/ and http://+:80) should **not** be used. Top-level wildcard bindings can open up your app to security vulnerabilities. This applies to both strong and weak wildcards. Use explicit host names rather than wildcards. Subdomain wildcard binding (for example, \*.mysub.com) doesn't have this security risk if you control the entire parent domain (as opposed to \*.com, which is vulnerable). See [**rfc7230 section-5.4**](https://tools.ietf.org/html/rfc7230#section-5.4) for more information.

1. Under the server's node, select **Application Pools**.
2. Right-click the site's app pool and select **Basic Settings** from the contextual menu.
3. In the **Edit Application Pool** window, set the **.NET CLR version** to **No Managed Code**:



ASP.NET Core runs in a separate process and manages the runtime. ASP.NET Core doesn't rely on loading the desktop CLR (.NET CLR)—the Core Common Language Runtime (CoreCLR) for .NET Core is booted to host the app in the worker process. Setting the **.NET CLR version** to **No Managed Code** is optional but recommended.

1. ASP.NET Core 2.2 or later: For a 64-bit (x64) [self-contained deployment](https://docs.microsoft.com/en-us/dotnet/core/deploying/#self-contained-deployments-scd) that uses the [in-process hosting model](https://docs.microsoft.com/en-us/aspnet/core/host-and-deploy/iis/?view=aspnetcore-3.1#in-process-hosting-model), disable the app pool for 32-bit (x86) processes.

In the **Actions** sidebar of IIS Manager > **Application Pools**, select **Set Application Pool Defaults** or **Advanced Settings**. Locate **Enable 32-Bit Applications** and set the value to False. This setting doesn't affect apps deployed for [out-of-process hosting](https://docs.microsoft.com/en-us/aspnet/core/host-and-deploy/aspnet-core-module?view=aspnetcore-3.1#out-of-process-hosting-model).

1. Confirm the process model identity has the proper permissions.

If the default identity of the app pool (**Process Model** > **Identity**) is changed from **ApplicationPoolIdentity** to another identity, verify that the new identity has the required permissions to access the app's folder, database, and other required resources. For example, the app pool requires read and write access to folders where the app reads and writes files.

**Windows Authentication configuration (Optional)**  
For more information, see [Configure Windows authentication](https://docs.microsoft.com/en-us/aspnet/core/security/authentication/windowsauth?view=aspnetcore-3.1).

## Virtual Directories

[IIS Virtual Directories](https://docs.microsoft.com/en-us/iis/get-started/planning-your-iis-architecture/understanding-sites-applications-and-virtual-directories-on-iis#virtual-directories) aren't supported with ASP.NET Core apps. An app can be hosted as a [sub-application](https://docs.microsoft.com/en-us/aspnet/core/host-and-deploy/iis/?view=aspnetcore-3.1#sub-applications).

## Sub-applications

An ASP.NET Core app can be hosted as an [IIS sub-application (sub-app)](https://docs.microsoft.com/en-us/iis/get-started/planning-your-iis-architecture/understanding-sites-applications-and-virtual-directories-on-iis#applications). The sub-app's path becomes part of the root app's URL.

Static asset links within the sub-app should use tilde-slash (~/) notation. Tilde-slash notation triggers a [Tag Helper](https://docs.microsoft.com/en-us/aspnet/core/mvc/views/tag-helpers/intro?view=aspnetcore-3.1) to prepend the sub-app's pathbase to the rendered relative link. For a sub-app at /subapp\_path, an image linked with src="~/image.png" is rendered as src="/subapp\_path/image.png". The root app's Static File Middleware doesn't process the static file request. The request is processed by the sub-app's Static File Middleware.

If a static asset's src attribute is set to an absolute path (for example, src="/image.png"), the link is rendered without the sub-app's pathbase. The root app's Static File Middleware attempts to serve the asset from the root app's [web root](https://docs.microsoft.com/en-us/aspnet/core/fundamentals/?view=aspnetcore-3.1#web-root), which results in a 404 - Not Found response unless the static asset is available from the root app.

To host an ASP.NET Core app as a sub-app under another ASP.NET Core app:

1. Establish an app pool for the sub-app. Set the **.NET CLR Version** to **No Managed Code** because the Core Common Language Runtime (CoreCLR) for .NET Core is booted to host the app in the worker process, not the desktop CLR (.NET CLR).
2. Add the root site in IIS Manager with the sub-app in a folder under the root site.
3. Right-click the sub-app folder in IIS Manager and select **Convert to Application**.
4. In the **Add Application** dialog, use the **Select** button for the **Application Pool** to assign the app pool that you created for the sub-app. Select **OK**.

The assignment of a separate app pool to the sub-app is a requirement when using the in-process hosting model.

For more information on the in-process hosting model and configuring the ASP.NET Core Module, see [ASP.NET Core Module](https://docs.microsoft.com/en-us/aspnet/core/host-and-deploy/aspnet-core-module?view=aspnetcore-3.1).

# IIS - How To Host A .NET Core Application In 10 Steps

* [](https://www.c-sharpcorner.com/members/fabio-silva-lima)

* [Fabio Silva Lima](https://www.c-sharpcorner.com/members/fabio-silva-lima)

* Updated date Apr 28 2019

* 38.4k

* [6](javascript:void(0);)

* [2](javascript:void(0);)
* [facebook](https://www.facebook.com/sharer.php?u=https://www.c-sharpcorner.com/article/iis-how-to-host-a-net-core-application-in-10-steps/Default.aspx)
* [twitter](https://twitter.com/intent/tweet?&via=CsharpCorner&related=CsharpCorner&text=IIS+-+How+To+Host+A+.NET+Core+Application+In+10+Steps%20https://www.c-sharpcorner.com/article/iis-how-to-host-a-net-core-application-in-10-steps/Default.aspx%20by%20@fabiosilvalima)
* [linkedIn](https://www.linkedin.com/shareArticle?mini=true&url=https://www.c-sharpcorner.com/article/iis-how-to-host-a-net-core-application-in-10-steps/Default.aspx)
* [Reddit](https://www.reddit.com/submit?title=IIS+-+How+To+Host+A+.NET+Core+Application+In+10+Steps&url=https%3a%2f%2fwww.c-sharpcorner.com%2farticle%2fiis-how-to-host-a-net-core-application-in-10-steps%2fdefault.aspx)
* [Expand](javascript:void(0);)

[Download Free .NET & JAVA Files API](https://www.c-sharpcorner.com/AdRedirector.aspx?AdId=133&target=https://www.e-iceblue.com/download/free-products.html?aff_id=100)

[Try Free File Format APIs for Word/Excel/PDF](https://www.c-sharpcorner.com/AdRedirector.aspx?AdId=133&target=https://www.aspose.com/)

If you started developing with .NET Core and you can’t leave your Windows hosting server, then you need to learn how to set up the IIS for it. Follow these steps and you will be good to go.



Developing with .NET Core is not too different from developing with .NET Framework, but setting up the IIS to host a .NET Core application… ow… it needs some tricks.

I’m trying to be as clear as possible. For this article, I am using the source code that I wrote in my [.NET Core for .NET Developers](https://www.c-sharpcorner.com/article/net-core-for-net-developers/) article. Click [here](https://github.com/fabiosilvalima/FSL.NetCoreBasics) to download it.

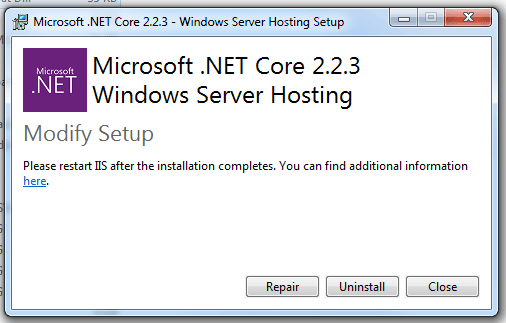
## Installations on Windows Machine

**Step 1**

Before setting up the IIS, it’s required to install .NET Core Runtime. For this article, I’ve used version 2.2. Without that, .NET Core won’t work. Remember, maybe it’s possible to reboot Windows.

**Step 2**

After that, you need to install the .NET Core Windows Server Hosting bundle, and you can download it here. Without that, IIS won’t work with .NET Core. You will need to reboot Windows.



## Changes in the .NET Core web project

**Step 3**

Open the FSL.NetCoreBasics.Mvc.csproj project file using notepad, locate the tag AspNetCoreHostingModel and change it to AspNetCoreHostingModelV2.

1. <AspNetCoreHostingModelV2>InProcess</AspNetCoreHostingModelV2>

After that, you will need to restore NuGet package and compile the application.

## Changes in the C# code

**Step 4**

In Visual Studio, open the project and, then open the Program.cs file, locate a method called Main and make the changes like the code below.

1. **public** **static** **void** Main(**string**[] args)
2. {
3. var host = **new** WebHostBuilder()
4. .UseKestrel()
5. .UseContentRoot(Directory.GetCurrentDirectory())
6. .UseIISIntegration()
7. .UseStartup<Startup>()
8. .Build();
9. host.Run();
10. }

You can delete the CreateWebHostBuilder method because it isn’t necessary anymore.

**Step 05**

Open the Startup.cs file, locate a method called ConfigureServices and add the following commands:

1. services.Configure<IISOptions>(options =>
2. {
3. options.AutomaticAuthentication = **false**;
4. });

## Adding a web.config file

**Step 06**

Yes my friend, it’s required to add a web.config file to our project to host a .NET Core application on IIS. Create that file and copy/paste the following code:

1. <?xml version="1.0" encoding="utf-8"?>
2. <configuration>
3. <system.webServer>
4. <handlers>
5. <add name="aspNetCore"
6. path="\*"
7. verb="\*"
8. modules="AspNetCoreModule"
9. resourceType="Unspecified"/>
10. </handlers>
11. <aspNetCore processPath="%LAUNCHER\_PATH%"
12. arguments="%LAUNCHER\_ARGS%"
13. stdoutLogEnabled="false"
14. stdoutLogFile=".\logs\stdout"
15. forwardWindowsAuthToken="false"/>
16. </system.webServer>
17. </configuration>

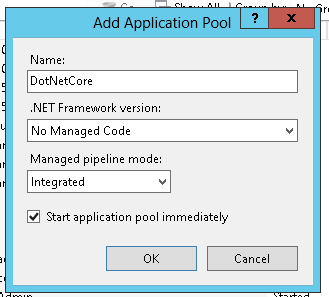
Special thanks to [Macoratti](http://www.macoratti.net/" \t "_blank) for that piece of code.

## Setting up the IIS

The next step is to set up an Application Pool and a virtual directory (or website) in IIS to run a .NET Core application.

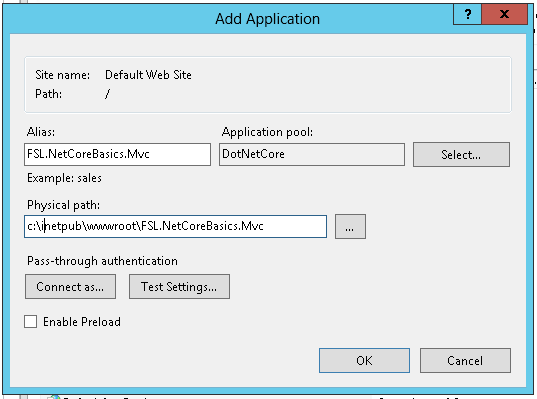
**Step 7**

Open the IIS, add a new Application Pool, then choose No Managed Code for .NET Framework version field and finally, select Integrated for Managed pipeline mode field.



**Step 8**

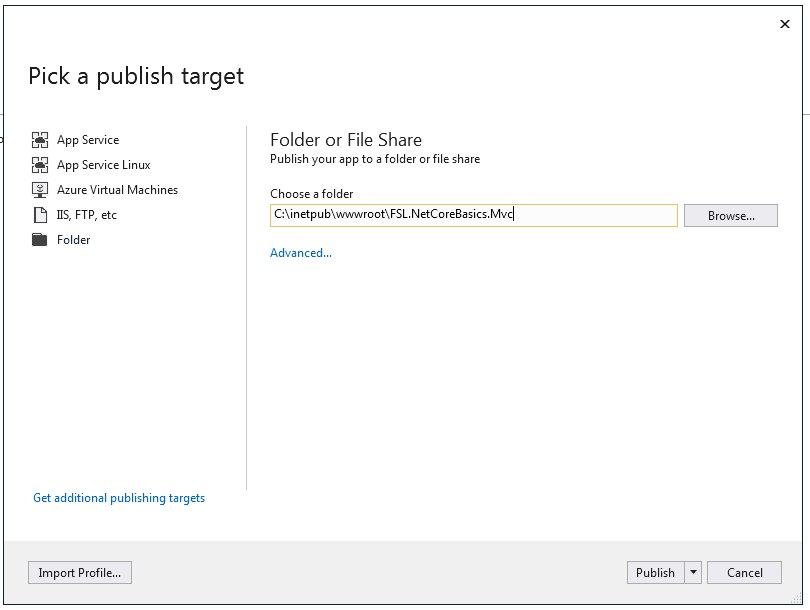
Add a new folder to host your application on IIS and create a virtual directory for it. Don’t forget to choose the Application Pool that you’ve created in Step 7.



## Publishing on IIS

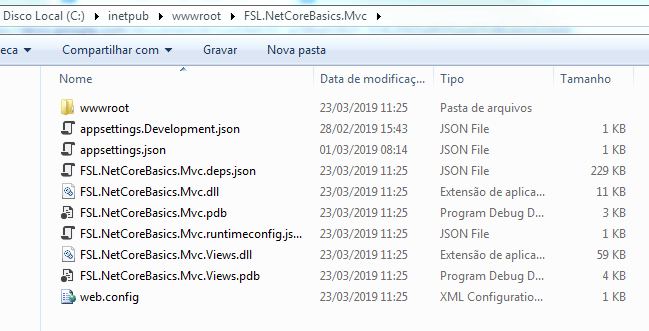
**Step 9**

Open the project in Visual Studio, click with the mouse’s right button on the project and choose the Publish option. Select the Folder tab on the left, fill or select your application folder and click Publish.



If you open Windows Explorer and navigate to your application folder, you will find all DLLs, the web.config file, and a wwwroot folder.

That’s your application published on IIS.



## Testing the application

**Step 10**

Open your favorite browser and type http://localhost/FSL.NetCoreBasics.Mvc. Your .NET Core application will be started, running on IIS.

