# InProcess Hosting

When an ASP.NET core application is executed, the .NET runtime looks for Main() method which is the entry point for the application.

The Main() method then calls CreateDefaultBuilder() static method of the WebHost class.

This CreateDefaultBuilder() method performs several tasks like

1. Setting up the web server
2. Loading the host and application configuration from various configuration sources and
3. Configuring logging

We will discuss the various configuration sources available in asp.net core, Loading the host and application configuration information

let's understand what the CreateDefaultBuilder() method does to configure and set up the web server. An ASP.NET core application can be hosted InProcess or OutOfProcess.

**InProcess hosting in ASP.NET Core**

To configure InProcess hosting, add <AspNetCoreHostingModel> element to the app's project file with a value of InProcess

<AspNetCoreHostingModel>InProcess</AspNetCoreHostingModel>

When we create a new ASP.NET Core project using one of the available project templates, the project defaults to the in-process hosting model for all IIS and IIS Express scenarios.

In case of InProcess hosting, CreateDefaultBuilder() method calls UseIIS() method and host the app inside of the IIS worker process (w3wp.exe or iisexpress.exe).

* From a performance standpoint, InProcess hosting delivers significantly higher request throughput than OutOfProcess hosting
* In the case of IIS, the process name that executes the app is w3wp and in the case of IIS Express it is iisexpress
* To get the process name executing the app, use System.Diagnostics.Process.GetCurrentProcess().ProcessName
* When we are run the project from Visual Studio it uses IISExpress by default.
* IIS Express is a lightweight, self-contained version of IIS, optimized for application development. We do not use it for production. In production we use IIS.

We will discuss deploying ASP.NET Core applications on IIS in our upcoming videos.

**With OutOfProcess hosting :**

* There are 2 web servers - An internal web server and an external web server.
* The internal web server is Kestrel and the external web server can be IIS, Nginx or Apache.
* With InProcess hosting, there is only one web server i.e the IIS that hosts the asp.net core application.
* So, we do not have the performance penalty of proxying requests between internal and external web server.

**What is Kestrel?**

Kestrel is a cross-platform web server for ASP.NET Core. It is supported on all platforms and versions that .NET Core supports.

It is included by default as internal server in ASP.NET Core. Kestrel can be used, by itself as an edge server i.e Internet-facing web server that

can directly process the incoming HTTP requests from the client. In Kestrel, the process used to host the app is dotnet.exe.

When we run a .NET Core application using the .NET Core CLI (Command-Line Interface), the application uses Kestrel as the web server.

The .NET Core CLI is a cross-platform tool for developing .NET core applications. Using the CLI we can

Create a new project, configuration file, or solution based on the specified template

Restore the dependencies and tools required for a .net core project

Build a project and all of its dependencies

Run a project etc...

There are a broad range of things that we can do with the .NET Core CLI. To run our asp.net core application using the .NET Core CLI.

Fire up Windows Command Prompt

Change the directory to the folder that contains your asp.net core project and execute dotnet run command

C:\Projects\EmployeeManagement\EmployeeManagement>dotnet run

After the .NET Core CLI builds and runs the project, it shows the URL using which we can access the application. In my case the application is available at http://localhost:5000

In case of Kestrel, the process used to host and execute the app is dotnet.exe. So when we navigate to http://localhost:5000, we will see the process name dotnet displayed.

# Out Of Process Hosting

There are 2 ways to configure **Out of Process hosting**  
  
**Option 1 :** Add <AspNetCoreHostingModel> element to the app's project file with a value of OutOfProcess

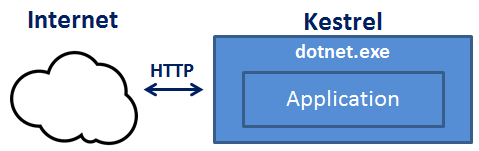
<AspNetCoreHostingModel>OutOfProcess</AspNetCoreHostingModel>

**Option 2 :** The default is OutOfProcess hosting. So if we remove the <AspNetCoreHostingModel> element from the project file, OutOfProcess hosting will be used by default.  
  
With **out of process hosting**

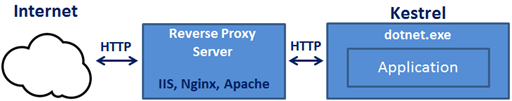
* There are 2 web servers - An an internal web server and an external web server.
  + The internal web server is Kestrel and the external web server can be IIS, Nginx or Apache. We discussed Kestrel in detail in Part 6 of ASP.NET Core Tutorial.

Depending on how you are running the asp.net core application, the external web server may or may not be used.

**Kestrel is a cross-platform web server** that is embedded in your ASP.NET Core application. With Out of Process Hosting model, Kestrel can be used in one of the following 2 ways.  
  
**Kestrel can be used as the internet facing web server** that process the incoming HTTP requests directly. In this model we are not using an external web server. Only Kestrel is used and it is this server that faces the internet, to directly handle the incoming HTTP requests. When we run the asp.net core application using the .NET core CLI, Kestrel is the only web server that is used to handle and process the incoming HTTP request.



**Kestrel can also be used in combination with a reverse proxy server**, such as IIS, Nginx, or Apache.



**If Kestrel can be used by itself as a web server, why do we need a reverse proxy server.**  
With **Out of Process Hosting**, using a reverse proxy server is a good choice as it provides an additional layer of configuration and security. It might integrate better with the existing infrastructure. It can also be used for load balancing.   
  
So, with a reverse proxy server in place, it receives incoming HTTP requests from the network and forwards them to the Kestrel server for processing. Upon processing the request, the Kestrel server sends the response to the reverse proxy server which then ultimately sends the response to the requested client over the network.  
  
We will discuss **Deploying ASP.NET Core application to IIS**and using IIS as a reverse proxy server in our upcoming videos. When we run an asp.net core application directly from Visual Studio it uses by default IIS Express. Since we have configured our application to use Out of Process hosting, IIS Express in this case acts a reverse proxy server.  
  
**IIS Express** takes the incoming HTTP request and forwards it to Kestrel for processing. Kestrel process the request and sends the response to IIS Express. IIS Express, in turn sends that response to the browser.  
  
With **Out of Process Hosting**, whether you use a reverse proxy server or not, it is the Kestrel server that hosts the application and process the request. The reverse proxy server if used, takes the incoming HTTP request and forwards it to the Kestrel server. Similarly it takes the response from the Kestrel server and sends it to the client. So the name of the process that hosts the application is dotnet.exe.  
  
**Use the following code to get the process name**

System.Diagnostics.Process.GetCurrentProcess().ProcessName

When we run the asp.net core project using the .NET Core CLI, by default it ignores the hosting setting we specified in the csproj file. So the AspNetCoreHostingModel element value in the **csproj file**is ignored.   
  
Irrespective of the value you specified (InProcess or OutOfProcess), it always uses OutOfProcess hosting and Kestrel is the web server that hosts the application and handle the http requests.  
  
**One common question :**Can we run an asp.net core application without using the built in kestrel web server.  
The answer is YES. If we use the InProcess hosting model, the application is hosted inside of the IIS worker process (w3wp.exe or iisexpress.exe). Kestrel is not used with InProcess hosting model.

# launchsettings.json File

* You will find this file in the **"Properties"** folder in the project root folder.
* The settings in this file are used when we run this ASP.NET core project either from Visual Studio or by using .NET Core CLI.
* This file is **only used on local development machine**. We do not need it for publishing our asp.net core application.
* If there are certain settings that you want your asp.net core application to use when you publish and deploy your app, store them in **appsettings.json file**. We usually store our application configuration settings in this file.
* We can also have **environment specific appsettings.json files**. For example, appsettings.Staging.json for the staging environment. In ASP.NET Core, in addition to appsettings.json file, we also have other configuration sources like Environment variables, User Secrets, Command Line Arguments and even our own custom configuration source.
* More on these different configuration sources and appsettings.json file in our next video.

At the moment, the following are the settings in **launchSettings.json file**

{

"iisSettings": {

"windowsAuthentication": false,

"anonymousAuthentication": true,

"iisExpress": {

"applicationUrl": "http://localhost:42192",

"sslPort": 0

}

},

"profiles": {

"IIS Express": {

"commandName": "IISExpress",

"launchBrowser": true,

"environmentVariables": {

"ASPNETCORE\_ENVIRONMENT": "Development"

}

},

"EmployeeManagement": {

"commandName": "Project",

"launchBrowser": true,

"applicationUrl": "http://localhost:5000",

"environmentVariables": {

"ASPNETCORE\_ENVIRONMENT": "Development"

}

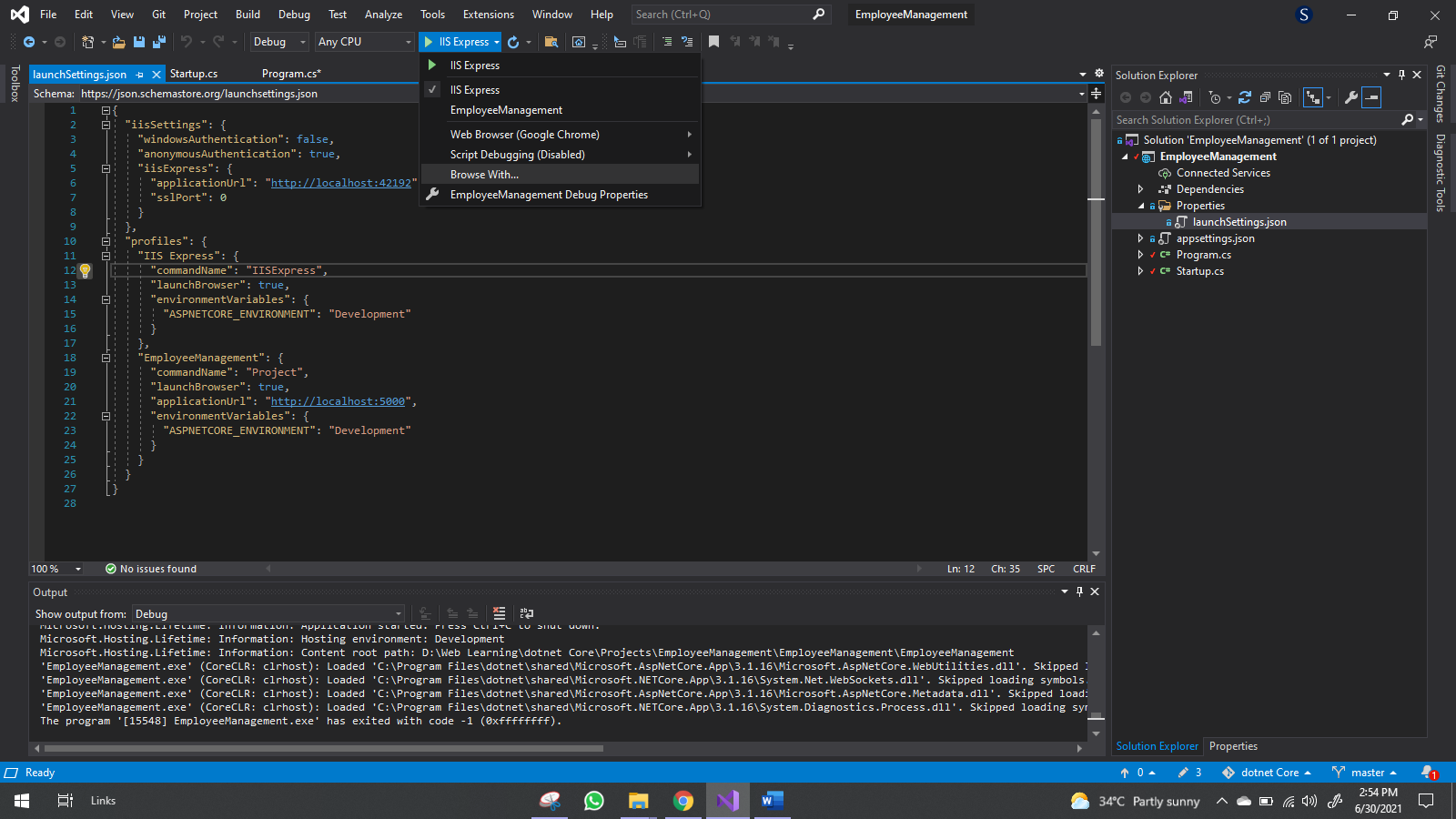
}

}

}

Notice, we have 2 profiles - **IIS Express**and **EmployeeManagement**

When we run the project from Visual Studio by pressing **CTRL + F5** or just **F5**, by default, the profile with "commandName": "IISExpress" is used. On the other hand, if we run the project using .NET Core CLI (dotnet run), the profile with the  "commandName": "Project" is used.   
  
However, we can change which profile to use by clicking on the dropdownlist in Visual Studio



The value of the **commandName**property can be any one of the following. 

* Project
* IISExpress
* IIS

This value along with the value of **AspNetCoreHostingModel**element in the project file, specifies the internal and external web server (reverse proxy server) to launch.

|  |  |  |  |
| --- | --- | --- | --- |
| **commandName** | **AspNetCoreHostingModel** | **Internal Web Server** | **External Web Server** |
| Project | Hosting Setting Ignored | Only one web server is used - Kestrel | |
| IISExpress | InProcess | Only one web server is used - IIS Express | |
| IISExpress | OutOfProcess | Kestrel | IIS Express |
| IIS | InProcess | Only one web server is used - IIS | |
| IIS | OutOfProcess | Kestrel | IIS |

You can change the settings in **launchSettings.json**file by directly editing the file or we can also change the settings using the Graphical User Interface (GUI) provided by Visual Studio.

We can also add new environment Variables. These environment variables are available throughout our asp.net core application and we can include code that conditionally executes depending on the value of these environment variables.

