Agenda: Hosting in Multiple Environments

- Hosting ASP.NET Core Website
- Web server implementations in ASP.NET Core
- Hosting on Windows with IIS
- Hosting as a Windows Service
- Hosting in Microsoft Azure

Hosting ASP.NET Core Website

ASP.NET Core apps configure and launch a *host*. The host is responsible for app startup and lifetime management. At a minimum, the host configures a server and a request processing pipeline.

A typical *Program.cs* calls **CreateDefaultBuilder** to begin setting up a host. **CreateDefaultBuilder** configures **Kestrel** as the web server and enables IIS integration by configuring the base path and port for the ASP.NET Core Module:

```
public class Program
{
   public static void Main(string[] args)
   {
      BuildWebHost(args).Run();
   }
   public static IWebHost BuildWebHost(string[] args) =>
      WebHost.CreateDefaultBuilder(args)
         .UseStartup<Startup>()
         .Build();
}
```

CreateDefaultBuilder performs the following tasks:

1. Configures Kestrel as the web server.
2. Sets the content root to the path returned by Directory.GetCurrentDirectory.(See below)
3. Loads optional configuration from:
   - appsettings.json.
   - appsettings.{Environment}.json.
   - Environment variables.
   - Command-line arguments.
4. Configures logging for console and debug output.

5. When running behind IIS, enables IIS integration. Configures the base path and port the server listens on when using the ASP.NET Core Module. The module creates a reverse proxy between IIS and Kestrel. Also configures the app to capture startup errors.
6. Sets **ServiceProviderOptions**.**ValidateScopes** to true if the app's environment is Development.

Content Root: The *content root* determines where the host searches for content files, such as MVC view files. When the app is started from the project's root folder, the project's root folder is used as the content root.

Environment variable: ASPNETCORE_CONTENTROOT

OR

```
WebHost.CreateDefaultBuilder(args)
  .UseContentRoot("c:\\mywebsite")
```

Environment: Sets the app's environment.

Environment variable: ASPNETCORE_ENVIRONMENT

OR

```
WebHost.CreateDefaultBuilder(args)
  .UseEnvironment("Development")
```

Server URLs: Indicates the IP addresses or host addresses with ports and protocols that the server should listen on for requests.

Default: http://localhost:5000

Environment variable: ASPNETCORE_URLS

OR

```
WebHost.CreateDefaultBuilder(args)
  .UseUrls("http://localhost:5001")
```

Web Root: Sets the relative path to the app's static assets.

Web server implementations in ASP.NET Core

An ASP.NET Core app runs with an in-process HTTP server implementation. The server implementation listens for HTTP requests and surfaces them to the app as sets of request features composed into an HttpContext.

- In ASP.NET Core: IIS, Nginx, and Apache can't be used without Kestrel.
- ASP.NET Core was designed to run in its own process so that it can behave consistently across platforms.

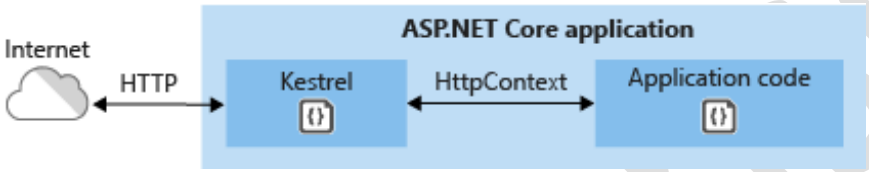
- IIS, Nginx, and Apache dictate their own startup procedure and environment. To use these server technologies directly, ASP.NET Core would need to adapt to the requirements of each server.
- Using a web server implementation, such as Kestrel, ASP.NET Core has control over the startup process and environment when hosted on different server technologies.

ASP.NET Core ships two server implementations:

1. **Kestrel** is a cross-platform HTTP server.
2. **HTTP.sys** is a Windows-only HTTP server (HTTP.sys is called WebListener in ASP.NET Core 1.x.)

About Kestrel

Kestrel is a cross-platform web server for ASP.NET Core. Kestrel is the web server that's included by default in ASP.NET Core project templates.



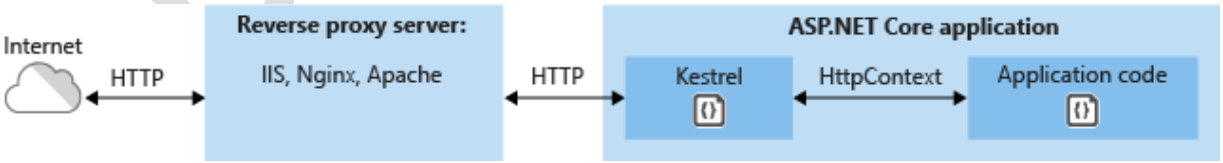
Kestrel supports the following scenarios:

- HTTPS
- WebSockets
- Unix sockets for high performance behind Nginx
- HTTP/2 (except on macOS)

Kestrel with a reverse proxy

A reverse proxy is a common setup for serving dynamic web apps. A reverse proxy terminates the HTTP request and forwards it to the ASP.NET Core app.

Kestrel can be used by itself or with a *reverse proxy server*, such as IIS, Nginx, or Apache.



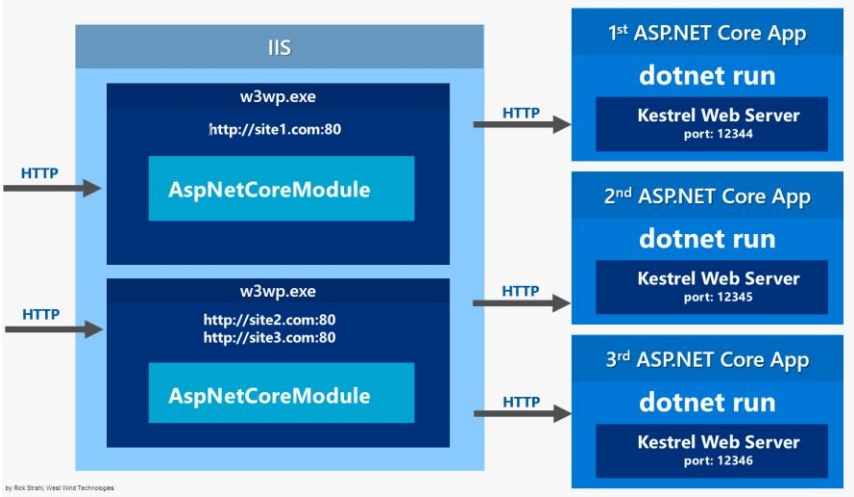
Why use a reverse proxy server?

- Kestrel is great for serving dynamic content from ASP.NET Core. However, the web serving capabilities aren't as feature rich as servers such as IIS, Apache, or Nginx.

- A reverse proxy server can offload work such as serving **static content, caching requests, compressing requests, and SSL termination** from the HTTP server.
- A reverse proxy server may reside on a dedicated machine or may be deployed alongside an HTTP server.

Kestrel used as an edge server without a reverse proxy server **doesn't support sharing the same IP and port** among multiple processes. When Kestrel is configured to listen on a port, Kestrel handles all of the traffic for that port regardless of requests' **Host headers**. A reverse proxy that can share ports has the ability to forward requests to Kestrel on a unique IP and port.

Even if a reverse proxy server isn't required, using a reverse proxy server might be a good choice.



Note: **HTTP.sys server** (formerly called WebListener) doesn't work in a reverse proxy configuration with IIS

How to use Kestrel in ASP.NET Core apps

ASP.NET Core project templates use Kestrel by default. In *Program.cs*, the template code calls **CreateDefaultBuilder**, which calls **UseKestrel** behind the scenes.

```
public static IWebHostBuilder CreateWebHostBuilder(string[] args)
{
  return WebHost.CreateDefaultBuilder(args)
    .UseStartup<Startup>()
    .ConfigureKestrel((context, options) =>
    {
      options.Limits.MaxConcurrentConnections = 100;
      options.Limits.MaxConcurrentUpgradedConnections = 100;
      options.Limits.MaxRequestBodySize = 10 * 1024;
      options.Limits.MinRequestBodyDataRate =
```

```
            new MinDataRate(bytesPerSecond: 100, gracePeriod: TimeSpan.FromSeconds(10));
        options.Limits.MinResponseDataRate =
            new MinDataRate(bytesPerSecond: 100, gracePeriod: TimeSpan.FromSeconds(10));
        options.Listen(IPAddress.Loopback, 5000);
        options.Listen(IPAddress.Loopback, 5001, listenOptions =>
        {
            listenOptions.UseHttps("testCert.pfx", "testPassword");
        });
    });
}
```

Read more about options:

https://docs.microsoft.com/en-us/aspnet/core/fundamentals/servers/kestrel?view=aspnetcore-2.2#kestrel-options

Hosting on Windows with IIS

- Following OS are supported
    - Windows 7 or later
    - Windows Server 2008 R2 or later
- When using **IIS** or **IIS Express** as a reverse proxy for ASP.NET Core, the ASP.NET Core app runs in a process separate from the IIS worker process.
- In the IIS process, the **ASP.NET Core Module** coordinates the reverse proxy relationship.
- The primary functions of the **ASP.NET Core Module** are to **start** the ASP.NET Core app, **restart** the app when it crashes, and **forward** HTTP traffic to the app.

In case required, to configure IIS options, include a service configuration for **IISOptions** in **ConfigureServices**

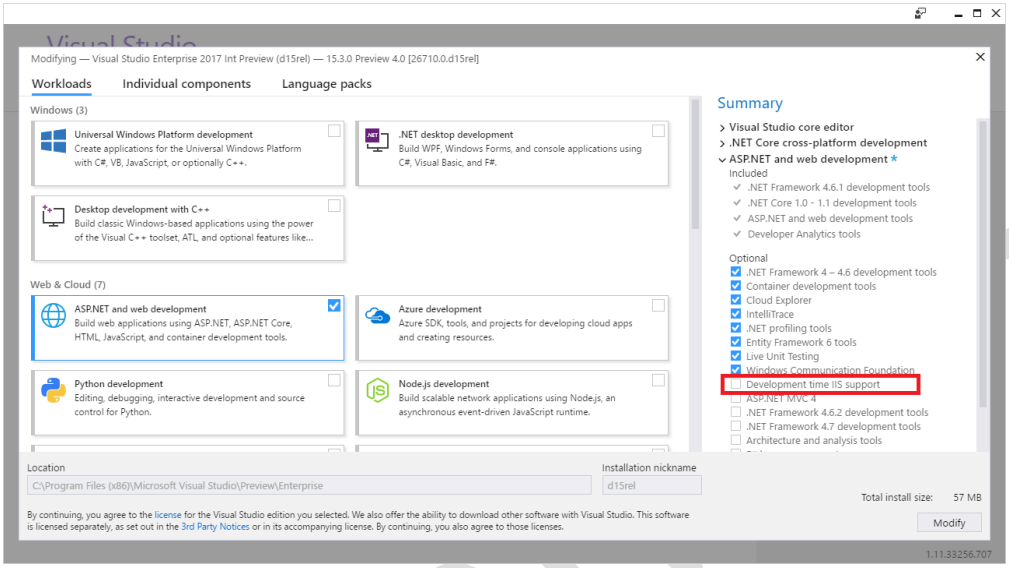
```
services.Configure<IISOptions>(options =>
{
  options.ForwardClientCertificate = false;
  options.AutomaticAuthentication = true;
});
```

1. Enable IIS

Before you can enable *Development time IIS support* in Visual Studio, you will need to enable IIS. You can do this by selecting the ***Internet Information Services*** checkbox in the ***Turn Windows features on or off*** dialog.

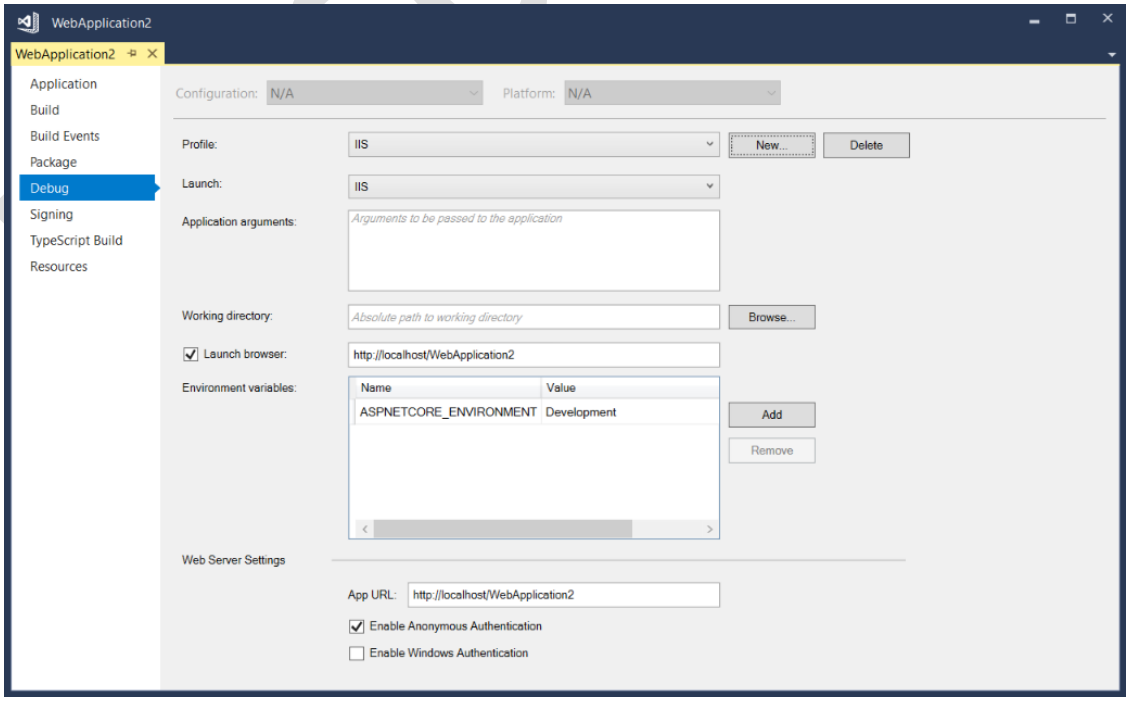
2. Development time IIS support (Optional)

Once you've installed IIS, you can launch the Visual Studio installer to modify your existing Visual Studio installation. In the installer select the ***Development time IIS support*** component which is listed as optional component under the *ASP.NET and web development* workload. This will install the **ASP.NET Core Module** which is a native IIS module required to run ASP.NET Core applications on IIS.



3. Adding support to an existing project

You can now create a new launch profile to add Development time IIS support. Make sure to select *IIS* from the Launch dropdown in the Debug tab of Project properties of your existing ASP.NET Core application



Note that the launchSettings.json file will be updated.

Alternatively:

Install the *.NET Core Windows Server Hosting bundle* on the hosting system.

The bundle installs the **.NET Core Runtime, .NET Core Library, and the ASP.NET Core Module**.

- Navigate to the .NET All Downloads page.
- Select the latest non-preview .NET Core runtime from the list (**.NET Core** > **Runtime** > **.NET Core Runtime x.y.z**).
- On the .NET Core runtime download page under **Windows**, select the **Server Hosting Installer** link to download the *.NET Core Windows Server Hosting bundle*.

Important! If the hosting bundle is installed before IIS, the bundle installation must be repaired. Run the hosting bundle installer again after installing IIS.

Restart the system or execute **net stop was /y** followed by **net start w3svc** from a command prompt. Restarting IIS picks up a change to the system PATH made by the installer.

4. **Deploy the Application:**
   a. **Project → Publish → Create New Profile → Folder → Provide Path → Create Profile → Publish**
   b. **Create the IIS site / Application**
   c. 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. Setting the **.NET CLR version** to **No Managed Code** is optional.

Hosting Models:

1. **In-process Hosting Model**

Add the following to application Project file. IIS HTTP Server (`IISHttpServer`) is used instead of Kestrel server

```
<PropertyGroup>
  <AspNetCoreHostingModel>InProcess</AspNetCoreHostingModel>
</PropertyGroup>
```

2. **Out-of-process hosting model**

```
<PropertyGroup>
  <AspNetCoreHostingModel>OutOfProcess</AspNetCoreHostingModel>
</PropertyGroup>
```

Note: If property isn't present in the file, the default value is `OutOfProcess`

Web.config might need following changes

```
<?xml version="1.0" encoding="utf-8"?>
<configuration>
 <system.webServer>
  <handlers>
   <add name="aspNetCore" path="*" verb="*" modules="AspNetCoreModule" resourceType="Unspecified" />
  </handlers>
  <aspNetCore processPath="dotnet"
        arguments=".\CoreASPNETApp.dll"
        forwardWindowsAuthToken="false"
        startupTimeLimit="3600"
        requestTimeout="23:00:00"
        stdoutLogEnabled="true"
        stdoutLogFile=".\logs\stdout" />
 </system.webServer>
</configuration>
```

To Troubleshoot the problems:

https://docs.microsoft.com/en-us/aspnet/core/host-and-deploy/iis/troubleshoot#aspnet-core-module-stdout-log

About web.config

The *web.config* file configures the ASP.NET Core Module. Creating, transforming, and publishing *web.config* is handled by the .NET Core Web SDK (Microsoft.NET.Sdk.Web)

- ASP.NET Core apps are hosted in a reverse proxy between IIS and the Kestrel server. In order to create the reverse proxy, the *web.config* file must be present at the content root path (typically the app base path) of the deployed app. This is the same location as the website physical path provided to IIS. The *web.config* file is required at the root of the app to enable the publishing of multiple apps using Web Deploy.
- Sensitive files exist on the app's physical path, such as *<assembly>.runtimeconfig.json*, *<assembly>.xml* (XML Documentation comments), and *<assembly>.deps.json*. When the *web.config* file is present and and the site starts normally, IIS doesn't serve these sensitive files if they're requested. If the *web.config* file is missing, incorrectly named, or unable to configure the site for normal startup, IIS may serve sensitive files publicly.

***web.config* file is published for a framework-dependent deployment:**

<aspNetCore processPath="dotnet" arguments=".\MyApp.dll"

```
        stdoutLogEnabled="false"
        stdoutLogFile=".\logs\stdout" />
```

***web.config* is published for a self-contained deployment:**

```
<aspNetCore processPath=".\MyApp.exe"
        stdoutLogEnabled="false"
        stdoutLogFile=".\logs\stdout" />
```

Host ASP.NET Core in a Windows Service

Prerequisite:

When creating a project in Visual Studio, use the **ASP.NET Core Application (.NET Framework)** template.

In the *.csproj* file, specify appropriate values for TargetFramework and RuntimeIdentifier.

```
<PropertyGroup>
 <TargetFramework>net461</TargetFramework>
 <RuntimeIdentifier>win7-x64</RuntimeIdentifier>
</PropertyGroup>
```

If the app receives requests from the Internet (not just from an internal network), it must use the HTTP.sys web server (formerly known as WebListener for ASP.NET Core 1.x apps) rather than Kestrel. IIS is recommended for use as a reverse proxy server with Kestrel for edge deployments.

1. Install the NuGet package **Microsoft.AspNetCore.Hosting.WindowsServices**.
2. Edit Program.cs

```
public static void Main(string[] args)
{
  var pathToExe = Process.GetCurrentProcess().MainModule.FileName;
  var pathToContentRoot = Path.GetDirectoryName(pathToExe);

  var host = WebHost.CreateDefaultBuilder(args)
    .UseContentRoot(pathToContentRoot)
    .UseStartup<Startup>()
    .UseApplicationInsights()
    .Build();

  host.RunAsService();
```

```
}
```

3. Publish the app to a folder. Use dotnet publish or a Visual Studio publish profile that publishes to a folder.
4. Test by creating and starting the service.

```
sc.exe create MyService binPath="c:\svc\aspnetcoreservice.exe"
sc.exe start MyService
```

5. Visit http://localhost:5000

https://docs.microsoft.com/en-us/aspnet/core/host-and-deploy/windows-service?tabs=aspnetcore2x

Host ASP.NET Core on Linux with Nginx

https://docs.microsoft.com/en-us/aspnet/core/host-and-deploy/linux-nginx?tabs=aspnetcore2x