Agenda: Configuring .NET Core Web Applications

- Configuration in ASP.NET Core
- Configuration by environment
- Binding a POCO class
- Access configuration in a Razor Page or MVC View
- ASPNETCORE_ENVIRONMENT Variable

Configuration in ASP.NET Core

Consider the following ***appsettings.json*** file:

```
{
 "Logging": {
  "IncludeScopes": false,
  "LogLevel": {
   "Default": "Warning"
  }
 },
 "option1": "value1_from_json",
 "option2": 2,
 "subsection": {
  "suboption1": "subvalue1_from_json"
 },
 "wizards": [
  {
   "Name": "Gandalf",
   "Age": "1000"
  },
  {
   "Name": "Harry",
   "Age": "17"
  }
 ]
}
```

```
public Startup(IConfiguration configuration)
{
```

```
    Configuration = configuration;
    //Following is used if code is written in any other location.
    /*var builder = new ConfigurationBuilder()
    .SetBasePath(Directory.GetCurrentDirectory())
    .AddJsonFile("appsettings.json");
    Configuration = builder.Build(); */

    Console.WriteLine($"option1 = {Configuration["option1"]}");
    Console.WriteLine($"option2 = {Configuration["option2"]}");
    Console.WriteLine($"suboption1 = {Configuration["subsection:suboption1"]}");
    Console.WriteLine();

    Console.WriteLine("Wizards(Array):");
    Console.Write($"{Configuration["wizards:0:Name"]}, ");
    Console.WriteLine($"age {Configuration["wizards:0:Age"]}");
    Console.Write($"{Configuration["wizards:1:Name"]}, ");
    Console.WriteLine($"age {Configuration["wizards:1:Age"]}");
    Console.WriteLine();
}
```

Configuration by environment

It's typical to have different configuration settings for different environments, for example, development, testing, and production.

The `CreateDefaultBuilder` extension method in an ASP.NET Core 2.x app adds configuration providers for reading JSON files and system configuration sources:

1. *appsettings.json*
2. *appsettings.**<EnvironmentName>**.json*
3. Environment variables (Highest Priority)

Configuration sources are read **in the order that they're specified**. In the preceding code, the environment variables are read last. Any configuration values set through the environment replace those set in the two previous providers.

Binding a to Class

appsettings.json

```
{
```

```
  "Logging": {
    "IncludeScopes": false,
    "LogLevel": {
      "Default": "Warning"
    }
  },
 "ConnectionString": "This is a demo ConnectionString",
"MainWindow": {
    "Left": 10,
    "Top": 20,
    "Width": 30,
    "Height": 40
  }
}
```

Add the following classes to project.

```
public class LogLevelClass
{
    public string Default { get; set; }
}
public class LoggingClass
{
    public string IncludeScopes { get; set; }
    public LogLevelClass LogLevel { get; set; }
}
public class MyWindow
{
    public int Height{ get; set; }
    public int Width { get; set; }
    public int Top { get; set; }
    public int Left { get; set; }
}
```

Write the following code in Startup class

```
public Startup(IConfiguration configuration)
{
```

```
    Configuration = configuration;
    var window = new MyWindow();
     // Bind requires NuGet package Microsoft.Extensions.Configuration.Binder
    Configuration.GetSection("MainWindow").Bind(window);
    Console.WriteLine($"Window.Left = {window.Left}");
    Console.WriteLine($"Window.Top = {window.Top}");
    Console.WriteLine($"Window.Width = {window.Width}");
    Console.WriteLine($"Window.Height = {window.Height}");

    LoggingClass log = new LoggingClass();
    Configuration.GetSection("Logging").Bind(log); //Uses Exisiting Object
    Console.WriteLine($"Logging.IncludeScopes = {log.IncludeScopes}");
    Console.WriteLine($"Logging.LogLevel.Default = {log.LogLevel.Default}");
}
```

```
var cs = Configuration.GetValue<string>("ConnectionString");
```

Using Get<T>

```
MyWindow win = Configuration.GetSection("MainWindow").Get<MyWindow>(); //Creates a New object
Console.WriteLine($"Window.Width = {win.Width}");
```

Injecting Settings as Dependency

```
public class ApplicationSettings
{
    public MyWindow MainWindow { get; set; }
    public string ConnectionString { get; set; }
}
```

Add the Following to **Startup.ConfigureServices**

```
//Registers a Configuration Instance which TOptions will bind against
services.Configure<ApplicationSettings>(Configuration);
```

Inject the IOptions in Controller or any other class

```
public class HomeController : Controller
{
```

```
    string _ConnectionString;
    public HomeController(IOptions<ApplicationSettings> settings)
    {
        _ConnectionString = settings.Value.ConnectionString;
    }
}
```

Note: The **IOptionsSnapshot**<T> is registered as a **scoped dependency** and **IOptions<T>** is registered as **singleton dependency**. So **IOptionsSnapshot**<T> gets the chance to get the current configuration values on every request instead of just once.

Access configuration in a Razor Page or MVC View

To access configuration settings in a Razor Pages page or an MVC view, add a **using directive** for the **Microsoft.Extensions.Configuration** namespace and inject **IConfiguration** into the page or view.

```
@page
@model IndexModel

@using Microsoft.Extensions.Configuration
@inject IConfiguration Configuration

<!DOCTYPE html>
<html lang="en">
<head>
    <title>Index Page</title>
</head>
<body>
    <h1>Access configuration in a Razor Pages page</h1>
    <p>Configuration[&quot;key&quot;]: @Configuration["key"]</p>
</body>
</html>
```

Selecting an Environment using ASPNETCORE_ENVIRONMENT

ASP.NET Core reads the environment variable **ASPNETCORE_ENVIRONMENT** at application startup and stores that value in **IHostingEnvironment**.**EnvironmentName**.

OR

WebHost.CreateDefaultBuilder(args)

```
  .UseEnvironment("Development")
```

ASPNETCORE_ENVIRONMENT can be set to any value, but three values are supported by the framework:

Development, Staging, and Production. If ASPNETCORE_ENVIRONMENT isn't set, it will default to `Production`.

Using the Environment Value:

```
public void Configure(IApplicationBuilder app, IHostingEnvironment env)
{
   app.UseLogUrl();
   if (env.IsDevelopment())
   {
      app.UseDeveloperExceptionPage();
      app.UseBrowserLink();
   }
   else
   {
      app.UseExceptionHandler("/Error");
   }
}
```

The **Environment** Tag Helper uses the value of **IHostingEnvironment.EnvironmentName** to include or exclude markup in the element:

```
@inject Microsoft.AspNetCore.Hosting.IHostingEnvironment hostingEnv
<p> ASPNETCORE_ENVIRONMENT = @hostingEnv.EnvironmentName</p>

<environment include="Development">
   <div>&lt;environment include="Development"&gt;</div>
</environment>
<environment exclude="Development">
   <div>&lt;environment exclude="Development"&gt;</div>
</environment>
<environment include="Staging,Development,Staging_2">
   <div>
      &lt;environment include="Staging,Development,Staging_2"&gt;
   </div>
```

```
</environment>
```

Note: On Windows and macOS, environment variables and values are not case sensitive. Linux environment variables and values are **case sensitive** by default.

Setting the value of EnvironmentName

Windows:

```
C:\> Set ASPNETCORE_ENVIRONMENT=Development
```

Or Use PowerShell

```
PS C:\>$Env:ASPNETCORE_ENVIRONMENT = "Development"
```

These commands take effect only for the current window. When the window is closed, the ASPNETCORE_ENVIRONMENT setting reverts to the default setting or machine value. In order to set the value globally on Windows open the **Control Panel → System → view Advanced system settings** and add or edit the ASPNETCORE_ENVIRONMENT value.

MacOS / Linux

```
export ASPNETCORE_ENVIRONMENT=Development
```

OR

```
ASPNETCORE_ENVIRONMENT=Development
```

Machine level environment variables are set in the ***.bashrc*** or ***.bash_profile*** file. Edit the file using any text editor and add the **export** statement.

For Linux distros, use the `export` command at the command line for session based variable settings and ***bash_profile*** file for machine level environment settings.

/Properties/launchSettings.json File

Environment values set in *launchSettings.json* **override** values set in the system environment.

The following JSON shows three profiles from a *launchSettings.json* file:

```
{
  "iisSettings": {
    "windowsAuthentication": false,
    "anonymousAuthentication": true,
    "iisExpress": {
      "applicationUrl": "http://localhost:59458/",
      "sslPort": 0
    }
```

```
  },
  "profiles": {
    "IIS Express": {
      "commandName": "IISExpress",
      "launchBrowser": true,
      "environmentVariables": {
        "ASPNETCORE_ENVIRONMENT": "Development"
      }
    },
    "WebApplication4": {
      "commandName": "Project",
      "launchBrowser": true,
      "environmentVariables": {
        "ASPNETCORE_ENVIRONMENT": "Development"
      },
      "applicationUrl": "http://localhost:59459/"
    },
    "Kestrel Staging": {
      "commandName": "Project",
      "launchBrowser": true,
      "environmentVariables": {
        "ASPNETCORE_My_Environment": "1",
        "ASPNETCORE_DETAILEDERRORS": "1",
        "ASPNETCORE_ENVIRONMENT": "Staging"
      },
      "applicationUrl": "http://localhost:51997/"
    }
  }
}
```

Note: Changes made to project profiles may not take effect until the web server is restarted. Kestrel must be restarted before it will detect changes made to its environment.

Choosing the Web Server / Environment:

When the application is launched with dotnet run, the first profile with `"commandName": "Project"` will be used.

The value of commandName specifies the web server to launch. commandName can be one of:

- IIS Express
- IIS
- Project (which launches Kestrel)

Note: The Visual Studio Debug tab provides a GUI to edit the *launchSettings.json* file:

Production Recommendations

The production environment should be configured to maximize security, performance, and application robustness.

Some common settings that differ from development include:

- Caching.
- Client-side resources are bundled, minified, and potentially served from a CDN.
- Diagnostic error pages disabled.
- Friendly error pages enabled.
- Production logging and monitoring enabled. For example, Application Insights.