Use multiple environments in ASP.NET Core

Article • 11/28/2023

By Rick Anderson and Kirk Larkin

ASP.NET Core configures app behavior based on the runtime environment using an environment variable.

Environments

To determine the runtime environment, ASP.NET Core reads from the following environment variables:

1. DOTNET_ENVIRONMENT
2. `ASPNETCORE_ENVIRONMENT` when the WebApplication.CreateBuilder method is called. The default ASP.NET Core web app templates call `WebApplication.CreateBuilder`. The `DOTNET_ENVIRONMENT` value overrides `ASPNETCORE_ENVIRONMENT` when `WebApplicationBuilder` is used. For other hosts, such as `ConfigureWebHostDefaults` and `WebHost.CreateDefaultBuilder`, `ASPNETCORE_ENVIRONMENT` has higher precedence.

`IHostEnvironment.EnvironmentName` can be set to any value, but the following values are provided by the framework:

- Development: The launchSettings.json file sets `ASPNETCORE_ENVIRONMENT` to `Development` on the local machine.
- Staging
- Production: The default if `DOTNET_ENVIRONMENT` and `ASPNETCORE_ENVIRONMENT` have not been set.

The following code:

- Is similar to the code generated by the ASP.NET Core templates.
- Enables the Developer Exception Page when `ASPNETCORE_ENVIRONMENT` is set to `Development`. This is done automatically by the WebApplication.CreateBuilder method.
- Calls UseExceptionHandler when the value of `ASPNETCORE_ENVIRONMENT` is anything other than `Development`.

- Provides an IWebHostEnvironment instance in the Environment property of `WebApplication`.

C#

```
var builder = WebApplication.CreateBuilder(args);

// Add services to the container.
builder.Services.AddRazorPages();

var app = builder.Build();

// Configure the HTTP request pipeline.
if (!app.Environment.IsDevelopment())
{
    app.UseExceptionHandler("/Error");
    // The default HSTS value is 30 days. You may want to change this for pro-
duction scenarios, see https://aka.ms/aspnetcore-hsts.
    app.UseHsts();
}

app.UseHttpsRedirection();
app.UseStaticFiles();

app.UseRouting();

app.UseAuthorization();

app.MapRazorPages();

app.Run();
```

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

CSHTML

```
<environment include="Development">
    <div>Environment is Development</div>
</environment>
<environment exclude="Development">
    <div>Environment is NOT Development</div>
</environment>
<environment include="Staging,Development,Staging_2">
    <div>Environment is: Staging, Development or Staging_2</div>
</environment>
```

The About page from the sample code includes the preceding markup and displays the value of `IWebHostEnvironment.EnvironmentName`.

On Windows and macOS, environment variables and values aren't case-sensitive. Linux environment variables and values are case-sensitive by default.

Create EnvironmentsSample

The sample code used in this article is based on a Razor Pages project named *EnvironmentsSample*.

The following .NET CLI commands create and run a web app named *EnvironmentsSample*:

Bash

```
dotnet new webapp -o EnvironmentsSample
cd EnvironmentsSample
dotnet run --verbosity normal
```

When the app runs, it displays output similar to the following:

Bash

```
info: Microsoft.Hosting.Lifetime[14]
      Now listening on: https://localhost:7152
info: Microsoft.Hosting.Lifetime[14]
      Now listening on: http://localhost:5105
info: Microsoft.Hosting.Lifetime[0]
      Application started. Press Ctrl+C to shut down.
info: Microsoft.Hosting.Lifetime[0]
      Hosting environment: Development
info: Microsoft.Hosting.Lifetime[0]
      Content root path: C:\Path\To\EnvironmentsSample
```

Set environment on the command line

Use the `--environment` flag to set the environment. For example:

.NET CLI

```
dotnet run --environment Production
```

The preceding command sets the environment to `Production` and displays output similar to the following in the command window:

Bash

```
info: Microsoft.Hosting.Lifetime[14]
      Now listening on: https://localhost:7262
info: Microsoft.Hosting.Lifetime[14]
      Now listening on: http://localhost:5005
info: Microsoft.Hosting.Lifetime[0]
      Application started. Press Ctrl+C to shut down.
info: Microsoft.Hosting.Lifetime[0]
      Hosting environment: Production
info: Microsoft.Hosting.Lifetime[0]
      Content root path: C:\Path\To\EnvironmentsSample
```

Development and launchSettings.json

The development environment can enable features that shouldn't be exposed in production. For example, the ASP.NET Core project templates enable the Developer Exception Page in the development environment. Because of the performance cost, scope validation and dependency validation only happens in development.

The environment for local machine development can be set in the *Properties\launchSettings.json* file of the project. Environment values set in `launchSettings.json` override values set in the system environment.

The `launchSettings.json` file:

- Is only used on the local development machine.
- Is not deployed.
- Contains profile settings.

The following JSON shows the `launchSettings.json` file for an ASP.NET Core web project named *EnvironmentsSample* created with Visual Studio or `dotnet new`:

JSON

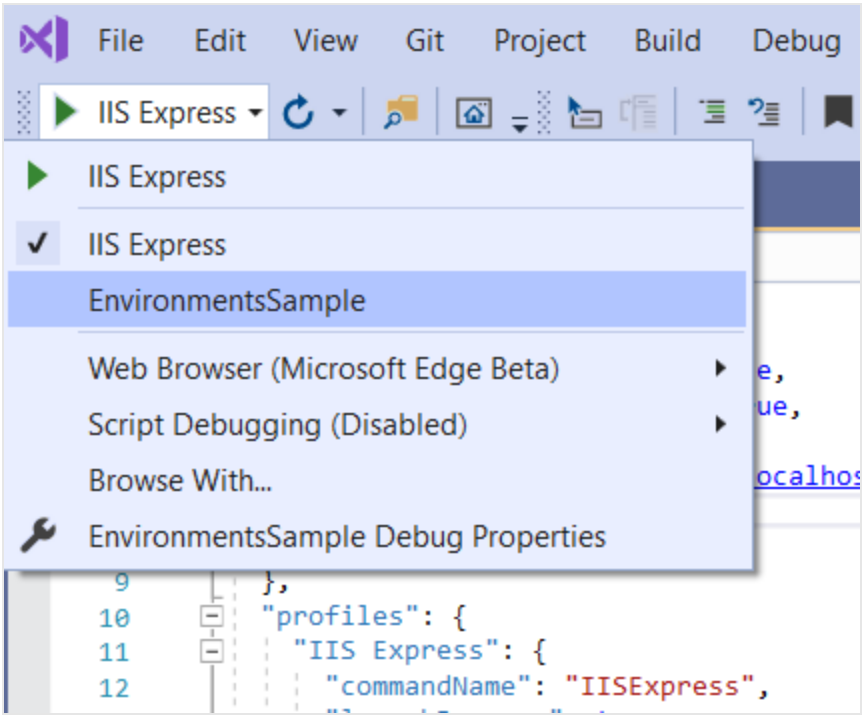
```
{
  "iisSettings": {
    "windowsAuthentication": false,
    "anonymousAuthentication": true,
    "iisExpress": {
```

```
      "applicationUrl": "http://localhost:59481",
      "sslPort": 44308
    }
  },
  "profiles": {
    "EnvironmentsSample": {
      "commandName": "Project",
      "dotnetRunMessages": true,
      "launchBrowser": true,
      "applicationUrl": "https://localhost:7152;http://localhost:5105",
      "environmentVariables": {
        "ASPNETCORE_ENVIRONMENT": "Development"
      }
    },
    "IIS Express": {
      "commandName": "IISExpress",
      "launchBrowser": true,
      "environmentVariables": {
        "ASPNETCORE_ENVIRONMENT": "Development"
      }
    }
  }
}
```

The preceding JSON contains two profiles:

- `EnvironmentsSample`: The profile name is the project name. As the first profile listed, this profile is used by default. The `"commandName"` key has the value `"Project"`, therefore, the Kestrel web server is launched.
- `IIS Express`: The `"commandName"` key has the value `"IISExpress"`, therefore, IISExpress is the web server.

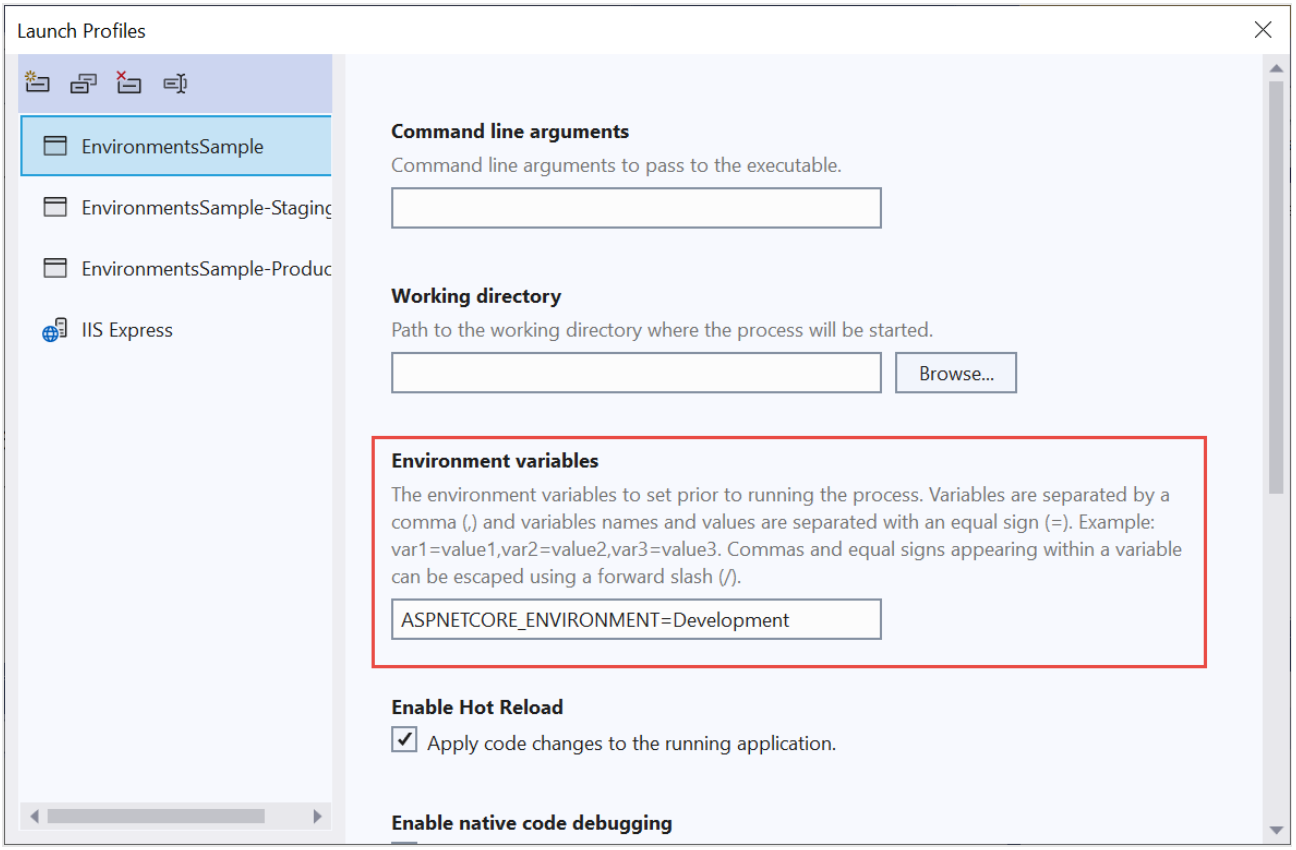
You can set the launch profile to the project or any other profile included in `launchSettings.json`. For example, in the image below, selecting the project name launches the Kestrel web server.



The value of `commandName` can specify the web server to launch. `commandName` can be any one of the following:

- `IISExpress` : Launches IIS Express.
- `IIS` : No web server launched. IIS is expected to be available.
- `Project` : Launches Kestrel.

The Visual Studio 2022 project properties **Debug / General** tab provides an **Open debug launch profiles UI** link. This link opens a **Launch Profiles** dialog that lets you edit the environment variable settings in the `launchSettings.json` file. You can also open the **Launch Profiles** dialog from the **Debug** menu by selecting **<project name> Debug Properties**. Changes made to project profiles may not take effect until the web server is restarted. Kestrel must be restarted before it can detect changes made to its environment.



The following `launchSettings.json` file contains multiple profiles:

JSON

```
{
  "iisSettings": {
    "windowsAuthentication": false,
    "anonymousAuthentication": true,
    "iisExpress": {
      "applicationUrl": "http://localhost:59481",
      "sslPort": 44308
    }
  },
  "profiles": {
    "EnvironmentsSample": {
      "commandName": "Project",
      "dotnetRunMessages": true,
      "launchBrowser": true,
      "applicationUrl": "https://localhost:7152;http://localhost:5105",
      "environmentVariables": {
        "ASPNETCORE_ENVIRONMENT": "Development"
      }
    },
    "EnvironmentsSample-Staging": {
      "commandName": "Project",
      "dotnetRunMessages": true,
      "launchBrowser": true,
```

```
      "applicationUrl": "https://localhost:7152;http://localhost:5105",
      "environmentVariables": {
        "ASPNETCORE_ENVIRONMENT": "Staging",
        "ASPNETCORE_DETAILEDERRORS": "1",
        "ASPNETCORE_SHUTDOWNTIMEOUTSECONDS": "3"
      }
    },
    "EnvironmentsSample-Production": {
      "commandName": "Project",
      "dotnetRunMessages": true,
      "launchBrowser": true,
      "applicationUrl": "https://localhost:7152;http://localhost:5105",
      "environmentVariables": {
        "ASPNETCORE_ENVIRONMENT": "Production"
      }
    },
    "IIS Express": {
      "commandName": "IISExpress",
      "launchBrowser": true,
      "environmentVariables": {
        "ASPNETCORE_ENVIRONMENT": "Development"
      }
    }
  }
}
```

Profiles can be selected:

- From the Visual Studio UI.
- Using the dotnet run CLI command with the `--launch-profile` option set to the profile's name. *This approach only supports Kestrel profiles.*

.NET CLI

```
dotnet run --launch-profile "EnvironmentsSample"
```

> **⚠ Warning**
>
> `launchSettings.json` shouldn't store secrets. The **Secret Manager tool** can be used to store secrets for local development.

When using Visual Studio Code , environment variables can be set in the `.vscode/launch.json` file. The following example sets several environment variables for

Host configuration values:

JSON

```
{
    "version": "0.2.0",
    "configurations": [
        {
            "name": ".NET Core Launch (web)",
            "type": "coreclr",
            // Configuration ommitted for brevity.
            "env": {
                "ASPNETCORE_ENVIRONMENT": "Development",
                "ASPNETCORE_URLS": "https://localhost:5001",
                "ASPNETCORE_DETAILEDERRORS": "1",
                "ASPNETCORE_SHUTDOWNTIMEOUTSECONDS": "3"
            },
            // Configuration ommitted for brevity.
```

The `.vscode/launch.json` file is used only by Visual Studio Code.

Production

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, using Application Insights.

Set the environment by setting an environment variable

It's often useful to set a specific environment for testing with an environment variable or platform setting. If the environment isn't set, it defaults to `Production`, which disables most debugging features. The method for setting the environment depends on the operating system.

When the host is built, the last environment setting read by the app determines the app's environment. The app's environment can't be changed while the app is running.

The About page from the sample code displays the value of `IWebHostEnvironment.EnvironmentName`.

Azure App Service

Production is the default value if `DOTNET_ENVIRONMENT` and `ASPNETCORE_ENVIRONMENT` have not been set. Apps deployed to Azure are `Production` by default.

To set the environment in an Azure App Service app by using the portal:

1. Select the app from the **App Services** page.
2. In the **Settings** group, select **Configuration**.
3. In the **Application settings** tab, select **New application setting**.
4. In the **Add/Edit application setting** window, provide `ASPNETCORE_ENVIRONMENT` for the **Name**. For **Value**, provide the environment (for example, `Staging`).
5. Select the **Deployment slot setting** checkbox if you wish the environment setting to remain with the current slot when deployment slots are swapped. For more information, see Set up staging environments in Azure App Service in the Azure documentation.
6. Select **OK** to close the **Add/Edit application setting** dialog.
7. Select **Save** at the top of the **Configuration** page.

Azure App Service automatically restarts the app after an app setting is added, changed, or deleted in the Azure portal.

Windows - Set environment variable for a process

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

To set the `ASPNETCORE_ENVIRONMENT` for the current session when the app is started using dotnet run, use the following commands at a command prompt or in PowerShell:

Console

```
set ASPNETCORE_ENVIRONMENT=Staging
dotnet run --no-launch-profile
```

PowerShell

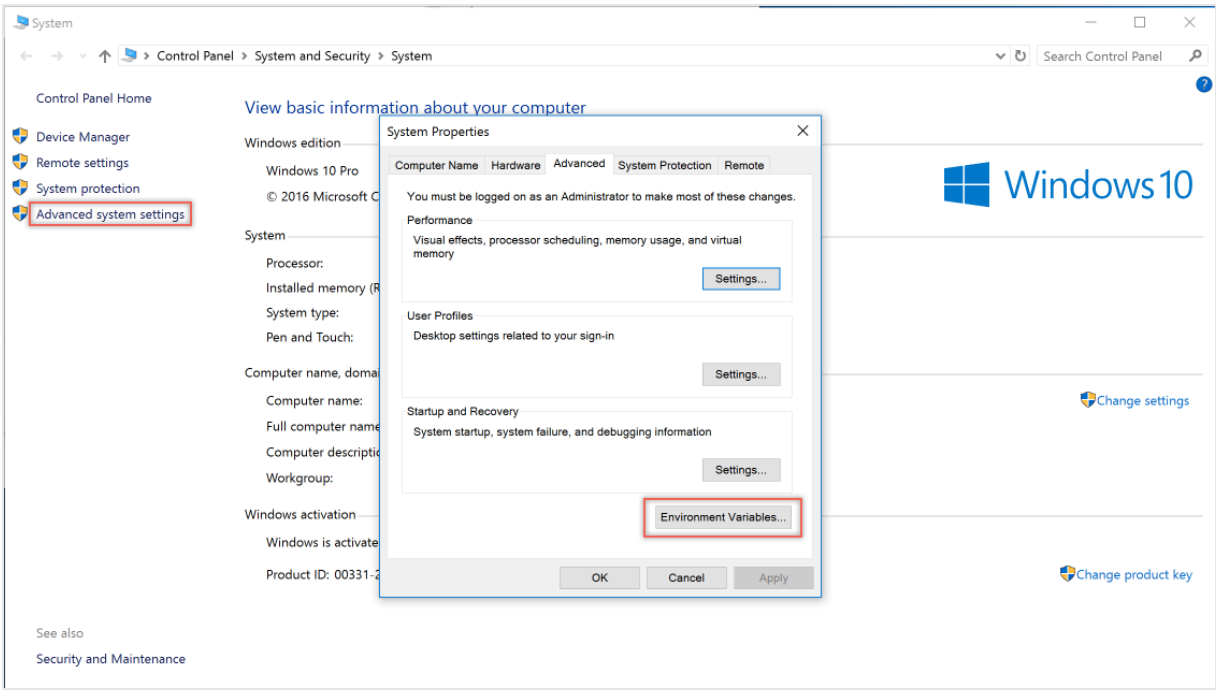
```
$Env:ASPNETCORE_ENVIRONMENT = "Staging"
dotnet run --no-launch-profile
```

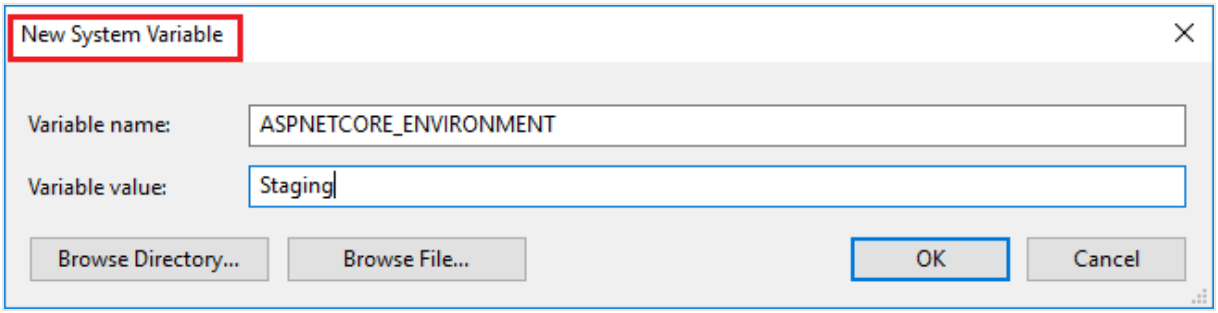
Windows - Set environment variable globally

The preceding commands set `ASPNETCORE_ENVIRONMENT` only for processes launched from that command window.

To set the value globally in Windows, use either of the following approaches:

- Open the **Control Panel** > **System** > **Advanced system settings** and add or edit the `ASPNETCORE_ENVIRONMENT` value:





- Open an administrative command prompt and use the `setx` command or open an administrative PowerShell command prompt and use

`[Environment]::SetEnvironmentVariable`:

- Console

```
setx ASPNETCORE_ENVIRONMENT Staging /M
```

The `/M` switch sets the environment variable at the system level. If the `/M` switch isn't used, the environment variable is set for the user account.

- PowerShell

```
[Environment]::SetEnvironmentVariable("ASPNETCORE_ENVIRONMENT",
"Staging", "Machine")
```

The `Machine` option sets the environment variable at the system level. If the option value is changed to `User`, the environment variable is set for the user account.

When the `ASPNETCORE_ENVIRONMENT` environment variable is set globally, it takes effect for `dotnet run` in any command window opened after the value is set. Environment values in `launchSettings.json` override values set in the system environment.

Windows - Use web.config

To set the `ASPNETCORE_ENVIRONMENT` environment variable with `web.config`, see the *Set environment variables* section of web.config file.

Windows - IIS deployments

Include the `<EnvironmentName>` property in the publish profile (.pubxml) or project file. This approach sets the environment in *web.config* when the project is published:

XML

```
<PropertyGroup>
  <EnvironmentName>Development</EnvironmentName>
</PropertyGroup>
```

To set the `ASPNETCORE_ENVIRONMENT` environment variable for an app running in an isolated Application Pool (supported on IIS 10.0 or later), see the *AppCmd.exe command* section of

Environment Variables <environmentVariables>. When the `ASPNETCORE_ENVIRONMENT` environment variable is set for an app pool, its value overrides a setting at the system level.

When hosting an app in IIS and adding or changing the `ASPNETCORE_ENVIRONMENT` environment variable, use one of the following approaches to have the new value picked up by apps:

- Execute `net stop was /y` followed by `net start w3svc` from a command prompt.
- Restart the server.

macOS

Setting the current environment for macOS can be performed in-line when running the app:

```
Bash

ASPNETCORE_ENVIRONMENT=Staging dotnet run
```

Alternatively, set the environment with `export` prior to running the app:

```
Bash

export ASPNETCORE_ENVIRONMENT=Staging
```

Machine-level environment variables are set in the *.bashrc* or *.bash_profile* file. Edit the file using any text editor. Add the following statement:

```
Bash

export ASPNETCORE_ENVIRONMENT=Staging
```

Linux

For Linux distributions, use the `export` command at a command prompt for session-based variable settings and the *bash_profile* file for machine-level environment settings.

Set the environment in code

To set the environment in code, use WebApplicationOptions.EnvironmentName when creating WebApplicationBuilder, as shown in the following example:

C#

```
var builder = WebApplication.CreateBuilder(new WebApplicationOptions
{
    EnvironmentName = Environments.Staging
});

// Add services to the container.
builder.Services.AddRazorPages();

var app = builder.Build();

// Configure the HTTP request pipeline.
if (!app.Environment.IsDevelopment())
{
    app.UseExceptionHandler("/Error");
    // The default HSTS value is 30 days. You may want to change this for pro-
duction scenarios, see https://aka.ms/aspnetcore-hsts.
    app.UseHsts();
}

app.UseHttpsRedirection();
app.UseStaticFiles();

app.UseRouting();

app.UseAuthorization();

app.MapRazorPages();

app.Run();
```

For more information, see .NET Generic Host in ASP.NET Core.

Configuration by environment

To load configuration by environment, see Configuration in ASP.NET Core.

Configure services and middleware by environment

Use WebApplicationBuilder.Environment or WebApplication.Environment to conditionally add services or middleware depending on the current environment. The project template includes an example of code that adds middleware only when the current environment isn't Development:

C#

```
var builder = WebApplication.CreateBuilder(args);

// Add services to the container.
builder.Services.AddRazorPages();

var app = builder.Build();

// Configure the HTTP request pipeline.
if (!app.Environment.IsDevelopment())
{
    app.UseExceptionHandler("/Error");
    // The default HSTS value is 30 days. You may want to change this for pro-
duction scenarios, see https://aka.ms/aspnetcore-hsts.
    app.UseHsts();
}

app.UseHttpsRedirection();
app.UseStaticFiles();

app.UseRouting();

app.UseAuthorization();

app.MapRazorPages();

app.Run();
```

The highlighted code checks the current environment while building the request pipeline. To check the current environment while configuring services, use `builder.Environment` instead of `app.Environment`.

Additional resources

- View or download sample code (how to download)
- App startup in ASP.NET Core
- Configuration in ASP.NET Core
- ASP.NET Core Blazor environments

Collaborate with us on GitHub

The source for this content can be found on GitHub, where you can also create and review issues and pull requests. For more information, see our contributor guide.

ASP.NET Core feedback

ASP.NET Core is an open source project. Select a link to provide feedback:

Open a documentation issue

Provide product feedback