Performance Diagnostic Tools

Article • 06/17/2024

By Mike Rousos

This article lists tools for diagnosing performance issues in ASP.NET Core.

Visual Studio Diagnostic Tools

The profiling and diagnostic tools built into Visual Studio are a good place to start investigating performance issues. These tools are powerful and convenient to use from the Visual Studio development environment. The tooling allows analysis of CPU usage, memory usage, and performance events in ASP.NET Core apps. Being built-in makes profiling easy at development time.

More information is available in Visual Studio documentation.

Application Insights

Application Insights provides in-depth performance data for your app. Application Insights automatically collects data on response rates, failure rates, dependency response times, and more. Application Insights supports logging custom events and metrics specific to your app.

Azure Application Insights provides multiple ways to give insights on monitored apps:

- Application Map – helps spot performance bottlenecks or failure hot-spots across all components of distributed apps.

- Azure Metrics Explorer is a component of the Microsoft Azure portal that allows plotting charts, visually correlating trends, and investigating spikes and dips in metrics' values.

- Performance blade in Application Insights portal:
  - Shows performance details for different operations in the monitored app.
  - Allows drilling into a single operation to check all parts/dependencies that contribute to a long duration.
  - Profiler can be invoked from here to collect performance traces on-demand.

- Azure Application Insights Profiler allows regular and on-demand profiling of .NET apps. Azure portal shows captured performance traces with call stacks and hot paths. The trace files can also be downloaded for deeper analysis using PerfView.

Application Insights can be used in a variety of environments:

- Optimized to work in Azure.
- Works in production, development, and staging.
- Works locally from Visual Studio or in other hosting environments.

For more information on code-based monitoring, see Application Insights for ASP.NET Core. For more information on codeless monitoring, see Monitor Azure App Service performance.

PerfView

PerfView is a performance analysis tool created by the .NET team specifically for diagnosing .NET performance issues. PerfView allows analysis of CPU usage, memory and GC behavior, performance events, and wall clock time.

For more about PerfView, see the user's guide available in the tool or on GitHub.

Windows Performance Toolkit

Windows Performance Toolkit (WPT) consists of two components: Windows Performance Recorder (WPR) and Windows Performance Analyzer (WPA). The tools produce in-depth performance profiles of Windows operating systems and apps. WPT has richer ways of visualizing data, but its data collecting is less powerful than PerfView's.

PerfCollect

While PerfView is a useful performance analysis tool for .NET scenarios, it only runs on Windows, so you can't use it to collect traces from ASP.NET Core apps running in Linux environments.

PerfCollect is a bash script that uses native Linux profiling tools (Perf and LTTng) to collect traces on Linux that can be analyzed by PerfView. PerfCollect is useful when performance problems show up in Linux environments where PerfView can't be used directly. Instead, PerfCollect can collect traces from .NET Core apps that are then analyzed on a Windows computer using PerfView.

More information about how to install and get started with PerfCollect is available on GitHub.

Other Third-party Performance Tools

The following lists some third-party performance tools that are useful in performance investigation of .NET Core applications.

- MiniProfiler
- dotTrace and dotMemory from JetBrains
- VTune from Intel

ASP.NET Core load/stress testing

Article • 03/07/2024

Load testing and stress testing are important to ensure a web app is performant and scalable. Load and stress testing have different goals even though they often share similar tests.

Load tests: Test whether the app can handle a specified load of users for a certain scenario while still satisfying the response goal. The app is run under normal conditions.

Stress tests: Test app stability when running under extreme conditions, often for a long period of time. The tests place high user load, either spikes or gradually increasing load on the app, or they limit the app's computing resources.

Stress tests determine if an app under stress can recover from failure and gracefully return to expected behavior. Under stress, the app is run at abnormally high stress.

Azure Load Testing is a fully managed load-testing service that enables you to generate high-scale load. The service simulates traffic for apps, regardless of where they're hosted. Azure Load Testing Preview enables you to use existing Apache JMeter scripts to generate high-scale load.

Visual Studio 2019 load testing has been deprecated. The corresponding Azure DevOps cloud-based load testing service has been closed.

Third-party tools

The following list contains third-party web performance tools with various feature sets:

- Apache JMeter
- ApacheBench (ab)
- Gatling
- jmeter-dotnet-dsl
- k6
- Locust
- West Wind WebSurge
- Netling
- Vegeta
- NBomber

Load and stress test with release builds

Load and stress tests should be done in release and production mode and not in debug and development mode. Release configurations are fully optimized with minimal logging. Debug configuration is not optimized. Development mode enables more information logging that can impact performance.

Tutorial: Measure performance using EventCounters in .NET Core

Article • 06/24/2023

This article applies to: ✓ .NET Core 3.0 SDK and later versions

In this tutorial, you'll learn how an EventCounter can be used to measure performance with a high frequency of events. You can use the available counters published by various official .NET Core packages, third-party providers, or create your own metrics for monitoring.

In this tutorial, you will:

✓ Implement an EventSource.
✓ Monitor counters with dotnet-counters.

Prerequisites

The tutorial uses:

- .NET Core 3.1 SDK or a later version.
- dotnet-counters to monitor event counters.
- A sample debug target app to diagnose.

Get the source

The sample application will be used as a basis for monitoring. The sample ASP.NET Core repository is available from the samples browser. You download the zip file, extract it once downloaded, and open it in your favorite IDE. Build and run the application to ensure that it works properly, then stop the application.

Implement an EventSource

For events that happen every few milliseconds, you'll want the overhead per event to be low (less than a millisecond). Otherwise, the impact on performance will be significant. Logging an event means you're going to write something to disk. If the disk is not fast enough, you will lose events. You need a solution other than logging the event itself.

When dealing with a large number of events, knowing the measure per event is not useful either. Most of the time all you need is just some statistics out of it. So you could get the statistics within the process itself and then write an event once in a while to report the statistics, that's what EventCounter will do.

Below is an example of how to implement an System.Diagnostics.Tracing.EventSource. Create a new file named *MinimalEventCounterSource.cs* and use the code snippet as its source:

```C#
using System.Diagnostics.Tracing;

[EventSource(Name = "Sample.EventCounter.Minimal")]
public sealed class MinimalEventCounterSource : EventSource
{
    public static readonly MinimalEventCounterSource Log = new
MinimalEventCounterSource();

    private EventCounter _requestCounter;

    private MinimalEventCounterSource() =>
        _requestCounter = new EventCounter("request-time", this)
        {
            DisplayName = "Request Processing Time",
            DisplayUnits = "ms"
        };

    public void Request(string url, long elapsedMilliseconds)
    {
        WriteEvent(1, url, elapsedMilliseconds);
        _requestCounter?.WriteMetric(elapsedMilliseconds);
    }

    protected override void Dispose(bool disposing)
    {
        _requestCounter?.Dispose();
        _requestCounter = null;

        base.Dispose(disposing);
    }
}
```

The EventSource.WriteEvent line is the EventSource part and is not part of EventCounter, it was written to show that you can log a message together with the event counter.

Add an action filter

The sample source code is an ASP.NET Core project. You can add an action filter globally that will log the total request time. Create a new file named *LogRequestTimeFilterAttribute.cs*, and use the following code:

C#

```
using System.Diagnostics;
using Microsoft.AspNetCore.Http.Extensions;
using Microsoft.AspNetCore.Mvc.Filters;

namespace DiagnosticScenarios
{
    public class LogRequestTimeFilterAttribute : ActionFilterAttribute
    {
        readonly Stopwatch _stopwatch = new Stopwatch();

        public override void OnActionExecuting(ActionExecutingContext
context) => _stopwatch.Start();

        public override void OnActionExecuted(ActionExecutedContext context)
        {
            _stopwatch.Stop();

            MinimalEventCounterSource.Log.Request(
                context.HttpContext.Request.GetDisplayUrl(),
_stopwatch.ElapsedMilliseconds);
        }
    }
}
```

The action filter starts a Stopwatch as the request begins, and stops after it has been completed, capturing the elapsed time. The total milliseconds are logged to the `MinimalEventCounterSource` singleton instance. For this filter to be applied, you need to add it to the filter collection. In the *Startup.cs* file, update the `ConfigureServices` method in include this filter.

C#

```
public void ConfigureServices(IServiceCollection services) =>
    services.AddControllers(options =>
options.Filters.Add<LogRequestTimeFilterAttribute>())
            .AddNewtonsoftJson();
```

Monitor event counter

With the implementation on an EventSource and the custom action filter, build and launch the application. You logged the metric to the EventCounter, but unless you access the statistics from of it, it is not useful. To get the statistics, you need to enable the EventCounter by creating a timer that fires as frequently as you want the events, as well as a listener to capture the events. To do that, you can use dotnet-counters.

Use the dotnet-counters ps command to display a list of .NET processes that can be monitored.

Console

```
dotnet-counters ps
```

Using the process identifier from the output of the `dotnet-counters ps` command, you can start monitoring the event counter with the following `dotnet-counters monitor` command:

Console

```
dotnet-counters monitor --process-id 2196 --counters
Sample.EventCounter.Minimal,Microsoft.AspNetCore.Hosting[total-
requests,requests-per-second],System.Runtime[cpu-usage]
```

While the `dotnet-counters monitor` command is running, hold F5 on the browser to start issuing continuous requests to the `https://localhost:5001/api/values` endpoint. After a few seconds stop by pressing q

Console

```
Press p to pause, r to resume, q to quit.
    Status: Running

[Microsoft.AspNetCore.Hosting]
    Request Rate / 1 sec                               9
    Total Requests                                   134
[System.Runtime]
    CPU Usage (%)                                     13
[Sample.EventCounter.Minimal]
    Request Processing Time (ms)                      34.5
```

The `dotnet-counters monitor` command is great for active monitoring. However, you may want to collect these diagnostic metrics for post processing and analysis. For that, use the `dotnet-counters collect` command. The `collect` switch command is similar to the `monitor` command, but accepts a few additional parameters. You can specify the desired output file name and format. For a JSON file named *diagnostics.json* use the following command:

Console

```
dotnet-counters collect --process-id 2196 --format json -o diagnostics.json
--counters Sample.EventCounter.Minimal,Microsoft.AspNetCore.Hosting[total-
requests,requests-per-second],System.Runtime[cpu-usage]
```

Again, while the command is running, hold F5 on the browser to start issuing continuous requests to the `https://localhost:5001/api/values` endpoint. After a few seconds stop by pressing q. The *diagnostics.json* file is written. The JSON file that is written is not indented, however; for readability it is indented here.

JSON

```json
{
  "TargetProcess": "DiagnosticScenarios",
  "StartTime": "8/5/2020 3:02:45 PM",
  "Events": [
    {
      "timestamp": "2020-08-05 15:02:47Z",
      "provider": "System.Runtime",
      "name": "CPU Usage (%)",
      "counterType": "Metric",
      "value": 0
    },
    {
      "timestamp": "2020-08-05 15:02:47Z",
      "provider": "Microsoft.AspNetCore.Hosting",
      "name": "Request Rate / 1 sec",
      "counterType": "Rate",
      "value": 0
    },
    {
      "timestamp": "2020-08-05 15:02:47Z",
      "provider": "Microsoft.AspNetCore.Hosting",
      "name": "Total Requests",
      "counterType": "Metric",
      "value": 134
    },
    {
      "timestamp": "2020-08-05 15:02:47Z",
      "provider": "Sample.EventCounter.Minimal",
      "name": "Request Processing Time (ms)",
      "counterType": "Metric",
      "value": 0
    },
    {
      "timestamp": "2020-08-05 15:02:47Z",
      "provider": "System.Runtime",
      "name": "CPU Usage (%)",
      "counterType": "Metric",
      "value": 0
    },
    {
      "timestamp": "2020-08-05 15:02:48Z",
      "provider": "Microsoft.AspNetCore.Hosting",
      "name": "Request Rate / 1 sec",
      "counterType": "Rate",
      "value": 0
    },
    {
      "timestamp": "2020-08-05 15:02:48Z",
      "provider": "Microsoft.AspNetCore.Hosting",
      "name": "Total Requests",
      "counterType": "Metric",
      "value": 134
    },
    {
      "timestamp": "2020-08-05 15:02:48Z",
      "provider": "Sample.EventCounter.Minimal",
      "name": "Request Processing Time (ms)",
      "counterType": "Metric",
      "value": 0
    },
    {
      "timestamp": "2020-08-05 15:02:48Z",
      "provider": "System.Runtime",
      "name": "CPU Usage (%)",
      "counterType": "Metric",
      "value": 0
    },
    {
      "timestamp": "2020-08-05 15:02:50Z",
      "provider": "Microsoft.AspNetCore.Hosting",
      "name": "Request Rate / 1 sec",
      "counterType": "Rate",
      "value": 0
    },
    {
      "timestamp": "2020-08-05 15:02:50Z",
      "provider": "Microsoft.AspNetCore.Hosting",
      "name": "Total Requests",
      "counterType": "Metric",
      "value": 134
    },
    {
      "timestamp": "2020-08-05 15:02:50Z",
      "provider": "Sample.EventCounter.Minimal",
      "name": "Request Processing Time (ms)",
      "counterType": "Metric",
      "value": 0
    }
  ]
}
```

Next steps

EventCounters

Globalization and localization in ASP.NET Core

Article • 10/25/2024

Important

This information relates to a pre-release product that may be substantially modified before it's commercially released. Microsoft makes no warranties, express or implied, with respect to the information provided here.

For the current release, see the .NET 9 version of this article.

By Rick Anderson, Damien Bowden, Bart Calixto, Nadeem Afana, and Hisham Bin Ateya

A multilingual website allows a website to reach a wider audience. ASP.NET Core provides services and middleware for localizing into different languages and cultures.

For Blazor localization guidance, which adds to or supersedes the guidance in this article, see ASP.NET Core Blazor globalization and localization.

Terms

- Globalization (G11N): The process of making an app support different languages and regions. The abbreviation comes from the first and last letters and the number of letters between them.
- Localization (L10N): The process of customizing a globalized app for specific languages and regions.
- Internationalization (I18N): Both globalization and localization.
- Culture: A language and, optionally, a region.
- Neutral culture: A culture that has a specified language, but not a region (for example "en", "es").
- Specific culture: A culture that has a specified language and region (for example, "en-US", "en-GB", "es-CL").
- Parent culture: The neutral culture that contains a specific culture (for example, "en" is the parent culture of "en-US" and "en-GB").
- Locale: A locale is the same as a culture.

Language and country/region codes

The RFC 4646 format for the culture name is `<language code>-<country/region code>`, where `<language code>` identifies the language and `<country/region code>` identifies the subculture. For example, `es-CL` for Spanish (Chile), `en-US` for English (United States), and `en-AU` for English (Australia). RFC 4646 is a combination of an ISO 639 two-letter lowercase culture code associated with a language and an ISO 3166 two-letter uppercase subculture code associated with a country or region. For more information, see System.Globalization.CultureInfo.

Tasks to localize an app

Globalizing and localizing an app involves the following tasks:

- Make an ASP.NET Core app's content localizable.
- Provide localized resources for the cultures the app supports
- Implement a strategy to select the culture for each request

View or download sample code (how to download)

Additional resources

- Url culture provider using middleware as filters in ASP.NET Core
- Applying the RouteDataRequest CultureProvider globally with middleware as filters
- IStringLocalizer : Uses the ResourceManager and ResourceReader to provide culture-specific resources at run time. The interface has an indexer and an `IEnumerable` for returning localized strings.
- IHtmlLocalizer: For resources that contain HTML.
- View and DataAnnotations
- Troubleshoot ASP.NET Core localization
- Globalizing and localizing .NET applications
- Resources in .resx Files
- Microsoft Multilingual App Toolkit
- Localization & Generics

Make an ASP.NET Core app's content localizable

Article • 10/25/2024

ⓘ **Important**

This information relates to a pre-release product that may be substantially modified before it's commercially released. Microsoft makes no warranties, express or implied, with respect to the information provided here.

For the current release, see the .NET 9 version of this article.

By Hisham Bin Ateya, Damien Bowden, Bart Calixto and Nadeem Afana

One task for localizing an app is to wrap localizable content with code that facilitates replacing that content for different cultures.

`IStringLocalizer`

IStringLocalizer and IStringLocalizer<T> were architected to improve productivity when developing localized apps. `IStringLocalizer` uses the ResourceManager and ResourceReader to provide culture-specific resources at run time. The interface has an indexer and an `IEnumerable` for returning localized strings. `IStringLocalizer` doesn't require storing the default language strings in a resource file. You can develop an app targeted for localization and not need to create resource files early in development.

The following code example shows how to wrap the string "About Title" for localization.

C#

```csharp
using Microsoft.AspNetCore.Mvc;
using Microsoft.Extensions.Localization;

namespace Localization.Controllers;

[Route("api/[controller]")]
public class AboutController : Controller
{
    private readonly IStringLocalizer<AboutController> _localizer;

    public AboutController(IStringLocalizer<AboutController> localizer)
    {
        _localizer = localizer;
    }

    [HttpGet]
    public string Get()
    {
        return _localizer["About Title"];
    }
}
```

In the preceding code, the `IStringLocalizer<T>` implementation comes from Dependency Injection. If the localized value of "About Title" isn't found, then the indexer key is returned, that is, the string "About Title".

You can leave the default language literal strings in the app and wrap them in the localizer, so that you can focus on developing the app. You develop an app with your default language and prepare it for the localization step without first creating a default resource file.

Alternatively, you can use the traditional approach and provide a key to retrieve the default language string. For many developers, the new workflow of not having a default language *.resx* file and simply wrapping the string literals can reduce the overhead of localizing an app. Other developers prefer the traditional work flow as it can be easier to work with long string literals and easier to update localized strings.

`IHtmlLocalizer`

Use the IHtmlLocalizer<TResource> implementation for resources that contain HTML. IHtmlLocalizer HTML-encodes arguments that are formatted in the resource string, but doesn't HTML-encode the resource string itself. In the following highlighted code, only the value of the `name` parameter is HTML-encoded.

C#

```csharp
using System;
using Microsoft.AspNetCore.Http;
using Microsoft.AspNetCore.Localization;
using Microsoft.AspNetCore.Mvc;
using Microsoft.AspNetCore.Mvc.Localization;

namespace Localization.Controllers;

public class BookController : Controller
{
    private readonly IHtmlLocalizer<BookController> _localizer;

    public BookController(IHtmlLocalizer<BookController> localizer)
    {
        _localizer = localizer;
    }

    public IActionResult Hello(string name)
    {
        ViewData["Message"] = _localizer["<b>Hello</b><i> {0}</i>", name];

        return View();
    }
```

NOTE: Generally, only localize text, not HTML.

`IStringLocalizerFactory`

At the lowest level, IStringLocalizerFactory can be retrieved from of Dependency Injection:

C#

```
public class TestController : Controller
{
    private readonly IStringLocalizer _localizer;
    private readonly IStringLocalizer _localizer2;

    public TestController(IStringLocalizerFactory factory)
    {
        var type = typeof(SharedResource);
        var assemblyName = new
AssemblyName(type.GetTypeInfo().Assembly.FullName);
        _localizer = factory.Create(type);
        _localizer2 = factory.Create("SharedResource", assemblyName.Name);
    }

    public IActionResult About()
    {
        ViewData["Message"] = _localizer["Your application description
page."]
            + " loc 2: " + _localizer2["Your application description
page."];

        return View();
    }
```

The preceding code demonstrates each of the two factory create methods.

Shared resources

You can partition your localized strings by controller or area, or have just one container. In the sample app, a marker class named `SharedResource` is used for shared resources. The marker class is never called:

```C#
// Dummy class to group shared resources

namespace Localization;

public class SharedResource
{
}
```

In the following sample, the `InfoController` and the `SharedResource` localizers are used:

```C#
public class InfoController : Controller
{
    private readonly IStringLocalizer<InfoController> _localizer;
    private readonly IStringLocalizer<SharedResource> _sharedLocalizer;

    public InfoController(IStringLocalizer<InfoController> localizer,
                   IStringLocalizer<SharedResource> sharedLocalizer)
    {
        _localizer = localizer;
        _sharedLocalizer = sharedLocalizer;
    }

    public string TestLoc()
    {
        string msg = "Shared resx: " + _sharedLocalizer["Hello!"] +
                     " Info resx " + _localizer["Hello!"];
        return msg;
    }
```

View localization

The IViewLocalizer service provides localized strings for a view. The `ViewLocalizer` class implements this interface and finds the resource location from the view file path. The following code shows how to use the default implementation of `IViewLocalizer`:

```CSHTML
@using Microsoft.AspNetCore.Mvc.Localization
```

```
@inject IViewLocalizer Localizer

@{
    ViewData["Title"] = Localizer["About"];
}
<h2>@ViewData["Title"].</h2>
<h3>@ViewData["Message"]</h3>

<p>@Localizer["Use this area to provide additional information."]</p>
```

The default implementation of `IViewLocalizer` finds the resource file based on the view's file name. There's no option to use a global shared resource file. `ViewLocalizer` implements the localizer using `IHtmlLocalizer`, so Razor doesn't HTML-encode the localized string. You can parameterize resource strings, and `IViewLocalizer` HTML-encodes the parameters but not the resource string. Consider the following Razor markup:

CSHTML

```
@Localizer["<i>Hello</i> <b>{0}!</b>", UserManager.GetUserName(User)]
```

A French resource file could contain the following values:

Expand table

Key	Value
`<i>Hello</i> <b>{0}!</b>`	`<i>Bonjour</i> <b>{0} !</b>`

The rendered view would contain the HTML markup from the resource file.

Generally, ***only localize text***, not HTML.

To use a shared resource file in a view, inject `IHtmlLocalizer<T>`:

CSHTML

```
@using Microsoft.AspNetCore.Mvc.Localization
@using Localization.Services

@inject IViewLocalizer Localizer
@inject IHtmlLocalizer<SharedResource> SharedLocalizer

@{
    ViewData["Title"] = Localizer["About"];
}
<h2>@ViewData["Title"].</h2>
```

```
<h1>@SharedLocalizer["Hello!"]</h1>
```

DataAnnotations localization

DataAnnotations error messages are localized with `IStringLocalizer<T>`. Using the option `ResourcesPath = "Resources"`, the error messages in `RegisterViewModel` can be stored in either of the following paths:

- *Resources/ViewModels.Account.RegisterViewModel.fr.resx*
- *Resources/ViewModels/Account/RegisterViewModel.fr.resx*

C#

```csharp
using System.ComponentModel.DataAnnotations;

namespace Localization.ViewModels.Account;

public class RegisterViewModel
{
    [Required(ErrorMessage = "The Email field is required.")]
    [EmailAddress(ErrorMessage = "The Email field is not a valid email
address.")]
    [Display(Name = "Email")]
    public string Email { get; set; }

    [Required(ErrorMessage = "The Password field is required.")]
    [StringLength(8, ErrorMessage = "The {0} must be at least {2} characters
long.",

MinimumLength = 6)]
    [DataType(DataType.Password)]
    [Display(Name = "Password")]
    public string Password { get; set; }

    [DataType(DataType.Password)]
    [Display(Name = "Confirm password")]
    [Compare("Password", ErrorMessage =
                            "The password and confirmation password do not
match.")]
    public string ConfirmPassword { get; set; }
}
```

Non-validation attributes are localized.

How to use one resource string for multiple classes

The following code shows how to use one resource string for validation attributes with multiple classes:

```C#
    services.AddMvc()
        .AddDataAnnotationsLocalization(options => {
            options.DataAnnotationLocalizerProvider = (type, factory) =>
                factory.Create(typeof(SharedResource));
        });
```

In the preceding code, `SharedResource` is the class corresponding to the *.resx* file where the validation messages are stored. With this approach, DataAnnotations only uses `SharedResource`, rather than the resource for each class.

Configure localization services

Localization services are configured in `Program.cs`:

```C#
builder.Services.AddLocalization(options => options.ResourcesPath =
"Resources");

builder.Services.AddMvc()
    .AddViewLocalization(LanguageViewLocationExpanderFormat.Suffix)
    .AddDataAnnotationsLocalization();
```

- AddLocalization adds the localization services to the services container, including implementations for `IStringLocalizer<T>` and `IStringLocalizerFactory`. The preceding code also sets the resources path to "Resources".
- AddViewLocalization adds support for localized view files. In this sample, view localization is based on the view file suffix. For example "fr" in the `Index.fr.cshtml` file.
- AddDataAnnotationsLocalization adds support for localized `DataAnnotations` validation messages through `IStringLocalizer` abstractions.

Note

You may not be able to enter decimal commas in decimal fields. To support jQuery validation for non-English locales that use a comma (",") for a decimal point, and non US-English date formats, you must take steps to globalize your app. See this GitHub comment 4076 for instructions on adding decimal comma.

Next steps

Localizing an app also involves the following tasks:

- Provide localized resources for the languages and cultures the app supports
- Implement a strategy to select the language/culture for each request

Additional resources

- Url culture provider using middleware as filters in ASP.NET Core
- Applying the RouteDataRequest CultureProvider globally with middleware as filters
- Globalization and localization in ASP.NET Core
- Provide localized resources for languages and cultures in an ASP.NET Core app
- Strategies for selecting language and culture in a localized ASP.NET Core app
- Troubleshoot ASP.NET Core localization
- Globalizing and localizing .NET applications
- Localization.StarterWeb project used in the article.
- Resources in .resx Files
- Microsoft Multilingual App Toolkit
- Localization & Generics

Provide localized resources for languages and cultures in an ASP.NET Core app

Article • 10/25/2024

ⓘ **Important**

This information relates to a pre-release product that may be substantially modified before it's commercially released. Microsoft makes no warranties, express or implied, with respect to the information provided here.

For the current release, see the .NET 9 version of this article.

By Rick Anderson, Damien Bowden, Bart Calixto, Nadeem Afana, and Hisham Bin Ateya

One task for localizing an app is to provide localized strings in resource files. This article is about working with resource files.

`SupportedCultures` and `SupportedUICultures`

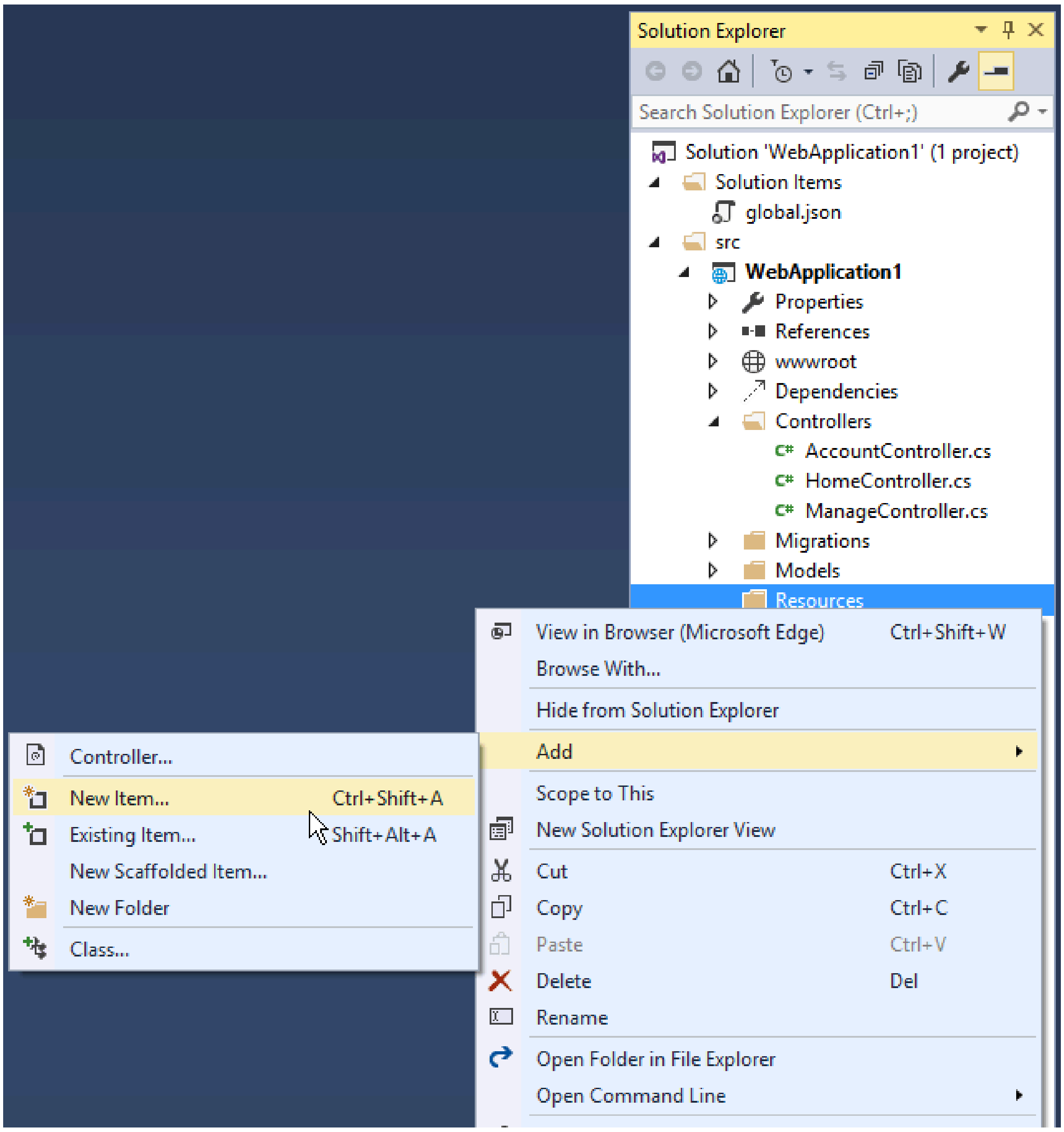
ASP.NET Core has two collections of culture values, `SupportedCultures` and `SupportedUICultures`. The CultureInfo object for `SupportedCultures` determines the results of culture-dependent functions, such as date, time, number, and currency formatting. `SupportedCultures` also determines the sorting order of text, casing conventions, and string comparisons. See StringComparer.CurrentCulture for more info on how the server gets the culture. The `SupportedUICultures` determines which translated strings (from *.resx* files) are looked up by the ResourceManager. The `ResourceManager` simply looks up culture-specific strings that are determined by `CurrentUICulture`. Every thread in .NET has `CurrentCulture` and `CurrentUICulture` objects. ASP.NET Core inspects these values when rendering culture-dependent functions. For example, if the current thread's culture is set to "en-US" (English, United States), `DateTime.Now.ToLongDateString()` displays "Thursday, February 18, 2016", but if `CurrentCulture` is set to "es-ES" (Spanish, Spain) the output will be "jueves, 18 de febrero de 2016".

Resource files

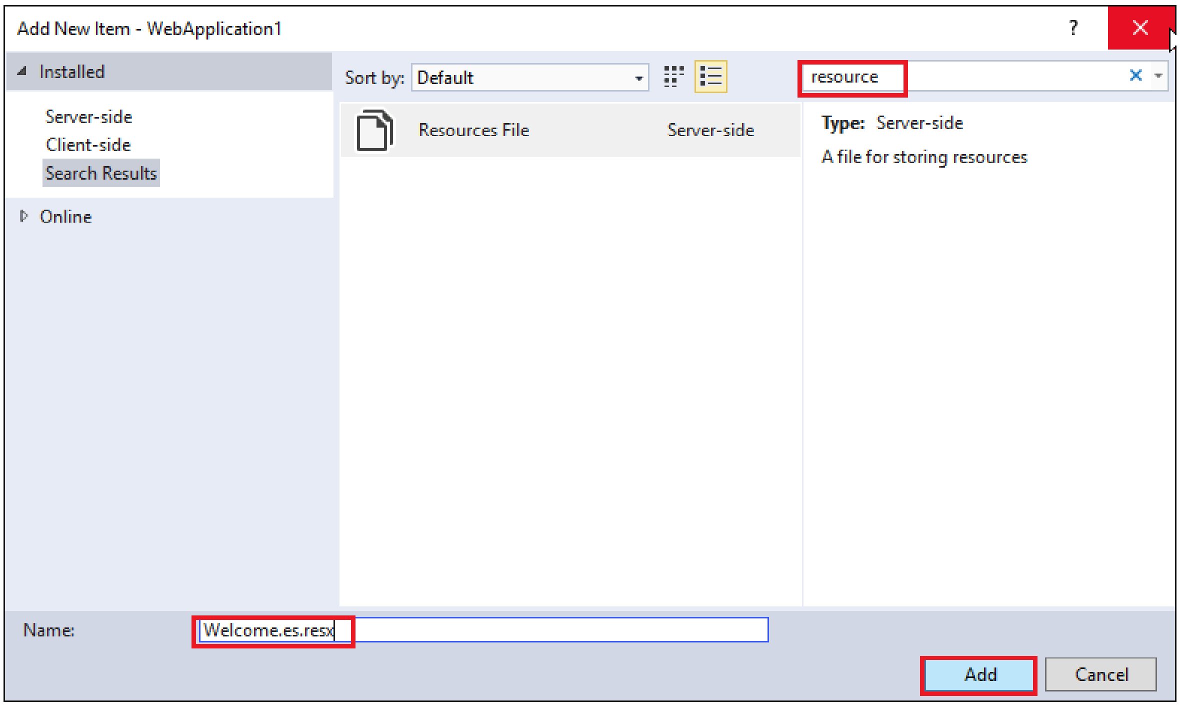
NOTE: ResX Viewer and Editor provides an alternate mechanism to work with resource files using Visual Studio Code.

A resource file is a useful mechanism for separating localizable strings from code. Translated strings for the non-default language are isolated in *.resx* resource files. For example, you might want to create a Spanish resource file named *Welcome.es.resx* containing translated strings. "es" is the language code for Spanish. To create this resource file in Visual Studio:

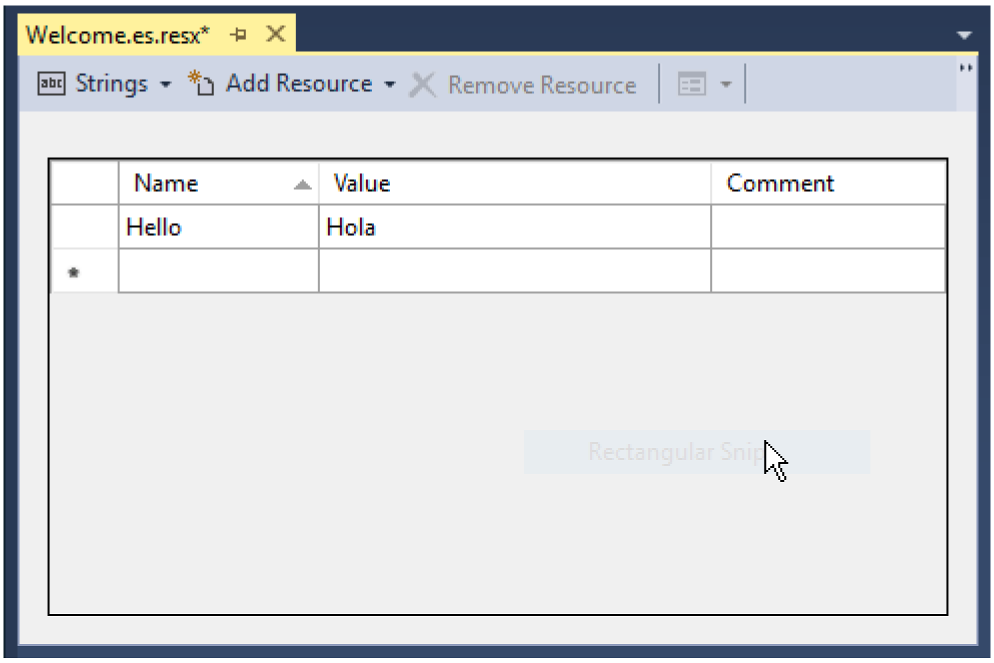
1. In **Solution Explorer**, right click on the folder that will contain the resource file, and then select **Add** > **New Item**.



2. In the **Search installed templates** box, enter "resource" and name the file.

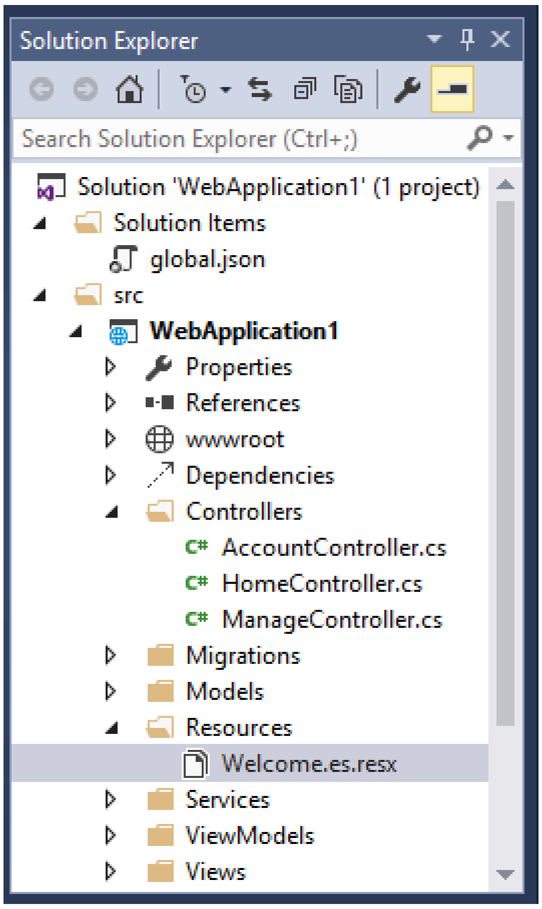


3. Enter the key value (native string) in the **Name** column and the translated string in the **Value** column.



	Name	Value	Comment
	Hello	Hola	
*			

Visual Studio shows the *Welcome.es.resx* file.



Resource file naming

Resources are named for the full type name of their class minus the assembly name. For example, a French resource in a project whose main assembly is `LocalizationWebsite.Web.dll` for the class `LocalizationWebsite.Web.Startup` would be named *Startup.fr.resx*. A resource for the class `LocalizationWebsite.Web.Controllers.HomeController` would be named *Controllers.HomeController.fr.resx*. If your targeted class's namespace isn't the same as the assembly name you will need the full type name. For example, in the sample project a resource for the type `ExtraNamespace.Tools` would be named *ExtraNamespace.Tools.fr.resx*.

In the sample project, the `ConfigureServices` method sets the `ResourcesPath` to "Resources", so the project relative path for the home controller's French resource file is *Resources/Controllers.HomeController.fr.resx*. Alternatively, you can use folders to organize resource files. For the home controller, the path would be *Resources/Controllers/HomeController.fr.resx*. If you don't use the `ResourcesPath` option, the *.resx* file would go in the project base directory. The resource file for `HomeController` would be named *Controllers.HomeController.fr.resx*. The choice of using the dot or path naming convention depends on how you want to organize your resource files.

Expand table

Resource name	Dot or path naming
Resources/Controllers.HomeController.fr.resx	Dot
Resources/Controllers/HomeController.fr.resx	Path

Resource files using `@inject IViewLocalizer` in Razor views follow a similar pattern. The resource file for a view can be named using either dot naming or path naming. Razor view resource files mimic the path of their associated view file. Assuming we set the `ResourcesPath` to "Resources", the French resource file associated with the `Views/Home/About.cshtml` view could be either of the following:

- Resources/Views/Home/About.fr.resx

- Resources/Views.Home.About.fr.resx

If you don't use the `ResourcesPath` option, the *.resx* file for a view would be located in the same folder as the view.

RootNamespaceAttribute

The RootNamespaceAttribute attribute provides the root namespace of an assembly when the root namespace of an assembly is different than the assembly name.

⚠ **Warning**

This can occur when a project's name is not a valid .NET identifier. For instance `my-project-name.csproj` will use the root namespace `my_project_name` and the assembly name `my-project-name` leading to this error.

If the root namespace of an assembly is different than the assembly name:

- Localization does not work by default.
- Localization fails due to the way resources are searched for within the assembly. `RootNamespace` is a build-time value which is not available to the executing process.

If the `RootNamespace` is different from the `AssemblyName`, include the following in `AssemblyInfo.cs` (with parameter values replaced with the actual values):

C#

```
using System.Reflection;
using Microsoft.Extensions.Localization;

[assembly: ResourceLocation("Resource Folder Name")]
[assembly: RootNamespace("App Root Namespace")]
```

The preceding code enables the successful resolution of resx files.

Culture fallback behavior

When searching for a resource, localization engages in "culture fallback". Starting from the requested culture, if not found, it reverts to the parent culture of that culture. As an aside, the CultureInfo.Parent property represents the parent culture. This usually (but not always) means removing the national signifier from the language-and-culture code. For example, the dialect of Spanish spoken in Mexico is "es-MX". It has the parent "es"—Spanish non-specific to any country.

Imagine your site receives a request for a "Welcome" resource using culture "fr-CA". The localization system looks for the following resources, in order, and selects the first match:

- *Welcome.fr-CA.resx*
- *Welcome.fr.resx*
- *Welcome.resx* (if the `NeutralResourcesLanguage` is "fr-CA")

As an example, if you remove the ".fr" culture designator and you have the culture set to French, the default resource file is read and strings are localized. The Resource manager designates a default or fallback resource for when nothing meets your requested culture. If you want to just return the key when missing a resource for the requested culture you must not have a default resource file.

Generate resource files with Visual Studio

If you create a resource file in Visual Studio without a culture in the file name (for example, *Welcome.resx*), Visual Studio will create a C# class with a property for each string. That's usually not what you want with ASP.NET Core. You typically don't have a default *.resx* resource file (a *.resx* file without the culture name). We suggest you create the *.resx* file with a culture name (for example *Welcome.fr.resx*). When you create a *.resx* file with a culture name, Visual Studio won't generate the class file.

Add other cultures

Each language and culture combination (other than the default language) requires a unique resource file. You create resource files for different cultures and locales by creating new resource files in which the language codes are part of the file name (for example, **en-us**, **fr-ca**, and **en-gb**). These codes are placed between the file name and the *.resx* file extension, as in *Welcome.es-MX.resx* (Spanish/Mexico).

Next steps

Localizing an app also involves the following tasks:

- Make the app's content localizable.
- Implement a strategy to select the language/culture for each request

Additional resources

- Url culture provider using middleware as filters in ASP.NET Core
- Applying the RouteDataRequest CultureProvider globally with middleware as filters
- Globalization and localization in ASP.NET Core
- Make an ASP.NET Core app's content localizable
- Strategies for selecting language and culture in a localized ASP.NET Core app
- Troubleshoot ASP.NET Core localization
- Globalizing and localizing .NET applications
- Localization.StarterWeb project used in the article.
- Resources in .resx Files
- Microsoft Multilingual App Toolkit
- Localization & Generics

Implement a strategy to select the language/culture for each request in a localized ASP.NET Core app

Article • 10/25/2024

ⓘ **Important**

This information relates to a pre-release product that may be substantially modified before it's commercially released. Microsoft makes no warranties, express or implied, with respect to the information provided here.

For the current release, see the .NET 9 version of this article.

Hisham Bin Ateya, Damien Bowden, Bart Calixto, Nadeem Afana, and Rick Anderson

One task for localizing an app is to implement a strategy for selecting the appropriate culture for each response the app returns.

Configure Localization middleware

The current culture on a request is set in the localization Middleware. The localization middleware is enabled in `Program.cs`. The localization middleware must be configured before any middleware that might check the request culture (for example, `app.UseMvcWithDefaultRoute()`).

C#

```csharp
builder.Services.Configure<RequestLocalizationOptions>(options =>
{
    var supportedCultures = new[] { "en-US", "fr" };
    options.SetDefaultCulture(supportedCultures[0])
        .AddSupportedCultures(supportedCultures)
        .AddSupportedUICultures(supportedCultures);
});
```

UseRequestLocalization initializes a RequestLocalizationOptions object. On every request the list of RequestCultureProvider in the RequestLocalizationOptions is enumerated and the first provider that can successfully determine the request culture is used. The default providers come from the `RequestLocalizationOptions` class:

1. QueryStringRequestCultureProvider
2. CookieRequestCultureProvider
3. AcceptLanguageHeaderRequestCultureProvider

The default list goes from most specific to least specific. Later in the article you'll see how you can change the order and even add a custom culture provider. If none of the providers can determine the request culture, the DefaultRequestCulture is used.

QueryStringRequestCultureProvider

Some apps will use a query string to set the CultureInfo. For apps that use the cookie or Accept-Language header approach, adding a query string to the URL is useful for debugging and testing code. By default, the QueryStringRequestCultureProvider is registered as the first localization provider in the `RequestCultureProvider` list. You pass the query string parameters `culture` and `ui-culture`. The following example sets the specific culture (language and region) to Spanish/Mexico:

```
http://localhost:5000/?culture=es-MX&ui-culture=es-MX
```

If only `culture` or `ui-culture` is passed in, the query string provider sets both values using the one passed in. For example, setting just the culture will set both the `Culture` and the `UICulture`:

```
http://localhost:5000/?culture=es-MX
```

CookieRequestCultureProvider

Production apps will often provide a mechanism to set the culture with the ASP.NET Core culture cookie. Use the MakeCookieValue method to create a cookie.

The CookieRequestCultureProvider DefaultCookieName returns the default cookie name used to track the user's preferred culture information. The default cookie name is `.AspNetCore.Culture`.

The cookie format is `c=%LANGCODE%|uic=%LANGCODE%`, where `c` is `Culture` and `uic` is `UICulture`, for example:

```
c=en-UK|uic=en-US
```

If only one of culture info or UI culture is provided, the provided culture is used for both culture info and UI culture.

The Accept-Language HTTP header

The Accept-Language header is settable in most browsers and was originally intended to specify the user's language. This setting indicates what the browser has been set to send or has inherited from the underlying operating system. The Accept-Language HTTP header from a browser request isn't an infallible way to detect the user's preferred language (see Setting language preferences in a browser). A production app should include a way for a user to customize their choice of culture.

Set the Accept-Language HTTP header in Edge

1. Search **Settings** for **Preferred languages**.
2. The preferred languages are listed in the **Preferred languages** box.
3. Select **Add languages** to add to the list.
4. Select **More actions …** next to a language to change the order of preference.

The Content-Language HTTP header

The Content-Language entity header:

- Is used to describe the language(s) intended for the audience.
- Allows a user to differentiate according to the users' own preferred language.

Entity headers are used in both HTTP requests and responses.

The `Content-Language` header can be added by setting the property ApplyCurrentCultureToResponseHeaders.

Adding the Content-Language header:

- Allows the RequestLocalizationMiddleware to set the `Content-Language` header with the `CurrentUICulture`.
- Eliminates the need to set the response header `Content-Language` explicitly.

C#

```
app.UseRequestLocalization(new RequestLocalizationOptions
{
    ApplyCurrentCultureToResponseHeaders = true
});
```

Apply the RouteDataRequest CultureProvider

The RouteDataRequestCultureProvider sets the culture based on the value of the `culture` route value. See Url culture provider using middleware as filters for information on:

- Using the middleware as filters feature of ASP.NET Core.
- How to use `RouteDataRequestCultureProvider` to set the culture of an app from the url.

See Applying the RouteDataRequest CultureProvider globally with middleware as filters for information on how to apply the `RouteDataRequestCultureProvider` globally.

Use a custom provider

Suppose you want to let your customers store their language and culture in your databases. You could write a provider to look up these values for the user. The following code shows how to add a custom provider:

C#

```
private const string enUSCulture = "en-US";

services.Configure<RequestLocalizationOptions>(options =>
{
    var supportedCultures = new[]
    {
        new CultureInfo(enUSCulture),
        new CultureInfo("fr")
    };

    options.DefaultRequestCulture = new RequestCulture(culture: enUSCulture,
uiCulture: enUSCulture);
    options.SupportedCultures = supportedCultures;
    options.SupportedUICultures = supportedCultures;

    options.AddInitialRequestCultureProvider(new
CustomRequestCultureProvider(async context =>
    {
        // My custom request culture logic
        return await Task.FromResult(new ProviderCultureResult("en"));
    }));
});
```

Use `RequestLocalizationOptions` to add or remove localization providers.

Change request culture providers order

RequestLocalizationOptions has three default request culture providers: QueryStringRequestCultureProvider, CookieRequestCultureProvider, and AcceptLanguageHeaderRequestCultureProvider. Use RequestLocalizationOptions.RequestCultureProviders property to change the order of these providers as shown in the following below:

```C#
    app.UseRequestLocalization(options =>
    {
        var questStringCultureProvider = options.RequestCultureProviders[0];
        options.RequestCultureProviders.RemoveAt(0);
        options.RequestCultureProviders.Insert(1,
questStringCultureProvider);
    });
```

In the preceding example, the order of `QueryStringRequestCultureProvider` and `CookieRequestCultureProvider` is switched, so the `RequestLocalizationMiddleware` looks for the cultures from the cookies first, then the query string.

As previously mentioned, add a custom provider via AddInitialRequestCultureProvider which sets the order to `0`, so this provider takes the precedence over the others.

User override culture

The RequestLocalizationOptions.CultureInfoUseUserOverride property allows the app to decide whether or not to use non-default Windows settings for the CultureInfo DateTimeFormat and NumberFormat properties. This has ***no*** impact on Linux. This directly corresponds to UseUserOverride.

```C#
    app.UseRequestLocalization(options =>
    {
        options.CultureInfoUseUserOverride = false;
    });
```

Set the culture programmatically

This sample **Localization.StarterWeb** project on GitHub contains UI to set the `Culture`. The `Views/Shared/_SelectLanguagePartial.cshtml` file allows you to select the culture from the list of supported cultures:

CSHTML

```
@using Microsoft.AspNetCore.Builder
@using Microsoft.AspNetCore.Http.Features
@using Microsoft.AspNetCore.Localization
@using Microsoft.AspNetCore.Mvc.Localization
@using Microsoft.Extensions.Options

@inject IViewLocalizer Localizer
@inject IOptions<RequestLocalizationOptions> LocOptions

@{
    var requestCulture = Context.Features.Get<IRequestCultureFeature>();
    var cultureItems = LocOptions.Value.SupportedUICultures
        .Select(c => new SelectListItem { Value = c.Name, Text =
c.DisplayName })
        .ToList();
    var returnUrl = string.IsNullOrEmpty(Context.Request.Path) ? "~/" : $"~
{Context.Request.Path.Value}";
}

<div title="@Localizer["Request culture provider:"]
@requestCulture?.Provider?.GetType().Name">
    <form id="selectLanguage" asp-controller="Home"
          asp-action="SetLanguage" asp-route-returnUrl="@returnUrl"
          method="post" class="form-horizontal" role="form">
        <label asp-
for="@requestCulture.RequestCulture.UICulture.Name">@Localizer["Language:"]
</label> <select name="culture"
          onchange="this.form.submit();"
          asp-for="@requestCulture.RequestCulture.UICulture.Name" asp-
items="cultureItems">
        </select>
    </form>
</div>
```

The `Views/Shared/_SelectLanguagePartial.cshtml` file is added to the `footer` section of the layout file so it will be available to all views:

CSHTML

```
<div class="container body-content" style="margin-top:60px">
    @RenderBody()
    <hr>
    <footer>
        <div class="row">
            <div class="col-md-6">
                <p>&copy; @System.DateTime.Now.Year - Localization</p>
            </div>
            <div class="col-md-6 text-right">
                @await Html.PartialAsync("_SelectLanguagePartial")
            </div>
        </div>
    </footer>
</div>
```

The `SetLanguage` method sets the culture cookie.

```C#
[HttpPost]
public IActionResult SetLanguage(string culture, string returnUrl)
{
    Response.Cookies.Append(
        CookieRequestCultureProvider.DefaultCookieName,
        CookieRequestCultureProvider.MakeCookieValue(new
RequestCulture(culture)),
        new CookieOptions { Expires = DateTimeOffset.UtcNow.AddYears(1) }
    );

    return LocalRedirect(returnUrl);
}
```

You can't plug in the `_SelectLanguagePartial.cshtml` to sample code for this project. The **Localization.StarterWeb** project on GitHub has code to flow the `RequestLocalizationOptions` to a Razor partial through the Dependency Injection container.

Model binding route data and query strings

See Globalization behavior of model binding route data and query strings.

Next steps

Localizing an app also involves the following tasks:

- Make the app's content localizable.

- Provide localized resources for the languages and cultures the app supports

Additional resources

- Url culture provider using middleware as filters in ASP.NET Core
- Applying the RouteDataRequest CultureProvider globally with middleware as filters
- Globalization and localization in ASP.NET Core
- Make an ASP.NET Core app's content localizable
- Provide localized resources for languages and cultures in an ASP.NET Core app
- Troubleshoot ASP.NET Core localization
- Globalizing and localizing .NET applications
- Localization.StarterWeb project used in the article.
- Resources in .resx Files
- Microsoft Multilingual App Toolkit
- Localization & Generics

Configure portable object localization in ASP.NET Core

Article • 08/07/2024

By Hisham Bin Ateya and Sébastien Ros .

This article walks through the steps for using Portable Object (PO) files in an ASP.NET Core application with the Orchard Core framework.

Note: Orchard Core isn't a Microsoft product. Microsoft provides no support for this feature.

View or download sample code (how to download)

What is a PO file?

PO files are distributed as text files containing the translated strings for a given language. Some advantages of using PO files instead of *.resx* files include:

- PO files support pluralization; *.resx* files don't support pluralization.
- PO files aren't compiled like *.resx* files. As such, specialized tooling and build steps aren't required.
- PO files work well with collaborative online editing tools.

Example

The following sample PO file contains the translation for two strings in French, including one with its plural form:

fr.po

```text
#: Pages/Index.cshtml:13
msgid "Hello world!"
msgstr "Bonjour le monde!"

msgid "There is one item."
msgid_plural "There are {0} items."
msgstr[0] "Il y a un élément."
msgstr[1] "Il y a {0} éléments."
```

This example uses the following syntax:

- `#:`: A comment indicating the context of the string to be translated. The same string might be translated differently depending on where it's being used.
- `msgid`: The untranslated string.
- `msgstr`: The translated string.

For pluralization support, more entries can be defined.

- `msgid_plural`: The untranslated plural string.
- `msgstr[0]`: The translated string for the case 0.
- `msgstr[N]`: The translated string for the case N.

The PO file specification can be found here.

Configuring PO file support in ASP.NET Core

This example is based on an ASP.NET Core Web application generated from a Visual Studio 2022 project template.

Referencing the package

Add a reference to the `OrchardCore.Localization.Core` NuGet package.

The `.csproj` file now contains a line similar to the following (version number may vary):

XML

```xml
<PackageReference Include="OrchardCore.Localization.Core" Version="1.5.0" />
```

Registering the service

Add the required services to `Program.cs`:

C#

```csharp
builder.Services.AddPortableObjectLocalization();

builder.Services
    .Configure<RequestLocalizationOptions>(options => options
        .AddSupportedCultures("fr", "cs")
        .AddSupportedUICultures("fr", "cs"));

builder.Services
```

```
      .AddRazorPages()
      .AddViewLocalization();
```

Add the following code to your Razor page of choice. `Index.cshtml` is used in this example.

CSHTML

```cshtml
@page
@using Microsoft.AspNetCore.Mvc.Localization
@inject IViewLocalizer Localizer
@{
    ViewData["Title"] = "Home";
}

<div class="text-center">
    <h1 class="display-4">Welcome</h1>
    <p>Learn about <a href="https://docs.microsoft.com/aspnet/core">building
Web apps with ASP.NET Core</a>.</p>
</div>

<p>@Localizer["Hello world!"]</p>
```

An `IViewLocalizer` instance is injected and used to translate the text "Hello world!".

Creating a PO file

Create a file named *<culture code>.po* in your application root folder. In this example, the file name is *fr.po* because the French language is used:

text

```text
msgid "Hello world!"
msgstr "Bonjour le monde!"
```

This file stores both the string to translate and the French-translated string. Translations revert to their parent culture, if necessary. In this example, the *fr.po* file is used if the requested culture is `fr-FR` or `fr-CA`.

Testing the application

Run your application, the text **Hello world!** is displayed.

Navigate to the URL `/Index?culture=fr-FR`. The text **Bonjour le monde!** is displayed.

Pluralization

PO files support pluralization forms, which is useful when the same string needs to be translated differently based on a cardinality. This task is made complicated by the fact that each language defines custom rules to select which string to use based on the cardinality.

The Orchard Localization package provides an API to invoke these different plural forms automatically.

Creating pluralization PO files

Add the following content to the previously mentioned *fr.po* file:

```text
msgid "There is one item."
msgid_plural "There are {0} items."
msgstr[0] "Il y a un élément."
msgstr[1] "Il y a {0} éléments."
```

See What is a PO file? for an explanation of what each entry in this example represents.

Adding a language using different pluralization forms

English and French strings were used in the previous example. English and French have only two pluralization forms and share the same form rules, which is that a cardinality of one is mapped to the first plural form. Any other cardinality is mapped to the second plural form.

Not all languages share the same rules. This is illustrated with the Czech language, which has three plural forms.

Create the `cs.po` file as follows, and note how the pluralization needs three different translations:

```text
msgid "Hello world!"
msgstr "Ahoj světe!!"

msgid "There is one item."
msgid_plural "There are {0} items."
msgstr[0] "Existuje jedna položka."
```

```
msgstr[1] "Existují {0} položky."
msgstr[2] "Existuje {0} položek."
```

To accept Czech localizations, add `"cs"` to the list of supported cultures in the `Configure` method:

```C#
builder.Services
    .Configure<RequestLocalizationOptions>(options => options
        .AddSupportedCultures("fr", "cs")
        .AddSupportedUICultures("fr", "cs"));
```

Edit the `Pages/Index.cshtml` file to render localized, plural strings for several cardinalities:

```CSHTML
<p>@Localizer.Plural(1, "There is one item.", "There are {0} items.")</p>
<p>@Localizer.Plural(2, "There is one item.", "There are {0} items.")</p>
<p>@Localizer.Plural(5, "There is one item.", "There are {0} items.")</p>
```

Note: In a real world scenario, a variable would be used to represent the count. Here, we repeat the same code with three different values to expose a specific case.

Upon switching cultures, you see the following:

For `/Index`:

```HTML
There is one item.
There are 2 items.
There are 5 items.
```

For `/Index?culture=fr`:

```HTML
Il y a un élément.
Il y a 2 éléments.
Il y a 5 éléments.
```

For `/Index?culture=cs`:

HTML

```
Existuje jedna položka.
Existují 2 položky.
Existuje 5 položek.
```

For the Czech culture, the three translations are different. The French and English cultures share the same construction for the two last translated strings.

Advanced tasks

Using additional arguments

The argument at index zero `{0}` always represents the count value. When invoking the `Plural` method it is possible to add additional arguments and their index will then start at one (`1`).

CSHTML

```
<p>@Localizer.Plural(count, "There is one item with the color {1}.", "There
are {0} items. The main color is {1}.", color)</p>
```

Contextualizing strings

Applications often contain the strings to be translated in several places. The same string may have a different translation in certain locations within an app (Razor views or class files). A PO file supports the notion of a file context, which can be used to categorize the string being represented. Using a file context, a string can be translated differently, depending on the file context (or lack of a file context).

The PO localization services use the name of the full class or the view that's used when translating a string. This is accomplished by setting the value on the `msgctxt` entry.

Consider a minor addition to the previous *fr.po* example. A Razor page located at `Pages/Index.cshtml` can be defined as the file context by setting the reserved `msgctxt` entry's value:

text

```
msgctxt "Views.Home.About"
msgid "Hello world!"
```

```
msgstr "Bonjour le monde!"
```

With the `msgctxt` set as such, text translation occurs when navigating to `/Index?culture=fr-FR`. The translation doesn't occur when navigating to `/Privacy?culture=fr-FR`.

When no specific entry is matched with a given file context, Orchard Core's fallback mechanism looks for an appropriate PO file without a context. Assuming there's no specific file context defined for `Pages/Privacy.cshtml`, navigating to `/Privacy?culture=fr-FR` loads a PO file such as:

text

```
msgid "Hello world!"
msgstr "Bonjour le monde!"
```

Changing the location of PO files

The default location of PO files can be changed in `Programs.cs`:

C#

```csharp
services.AddPortableObjectLocalization(options => options.ResourcesPath =
"Localization");
```

In this example, the PO files are loaded from the *Localization* folder.

Implementing a custom logic for finding localization files

When more complex logic is needed to locate PO files, the `OrchardCore.Localization.PortableObject.ILocalizationFileLocationProvider` interface can be implemented and registered as a service. This is useful when PO files can be stored in varying locations or when the files have to be found within a hierarchy of folders.

Using a different default pluralized language

The package includes a `Plural` extension method that's specific to two plural forms. For languages requiring more plural forms, create an extension method. With an extension method, you won't need to provide any localization file for the default language — the original strings are already available directly in the code.

You can use the more generic `Plural(int count, string[] pluralForms, params object[] arguments)` overload which accepts a string array of translations.

Localization Extensibility

Article • 07/26/2024

> **Important**
>
> This information relates to a pre-release product that may be substantially modified before it's commercially released. Microsoft makes no warranties, express or implied, with respect to the information provided here.
>
> For the current release, see the .NET 9 version of this article.

By Hisham Bin Ateya

This article:

- Lists the extensibility points on the localization APIs.
- Provides instructions on how to extend ASP.NET Core app localization.

Extensible Points in Localization APIs

ASP.NET Core localization APIs are built to be extensible. Extensibility allows developers to customize the localization according to their needs. For instance, OrchardCore has a `POStringLocalizer`. `POStringLocalizer` describes in detail using Portable Object localization to use `PO` files to store localization resources.

This article lists the two main extensibility points that localization APIs provide:

- RequestCultureProvider
- IStringLocalizer

Localization Culture Providers

ASP.NET Core localization APIs have four default providers that can determine the current culture of an executing request:

- QueryStringRequestCultureProvider
- CookieRequestCultureProvider
- AcceptLanguageHeaderRequestCultureProvider
- CustomRequestCultureProvider

The preceding providers are described in detail in the Localization Middleware documentation. If the default providers don't meet your needs, build a custom provider using one of the following approaches:

Use CustomRequestCultureProvider

CustomRequestCultureProvider provides a custom RequestCultureProvider that uses a simple delegate to determine the current localization culture:

C#

```
options.AddInitialRequestCultureProvider(new
CustomRequestCultureProvider(async context =>
{
    var currentCulture = "en";
    var segments = context.Request.Path.Value.Split(new char[] { '/' },
        StringSplitOptions.RemoveEmptyEntries);

    if (segments.Length > 1 && segments[0].Length == 2)
    {
        currentCulture = segments[0];
    }

    var requestCulture = new ProviderCultureResult(currentCulture);

    return Task.FromResult(requestCulture);
}));
```

Use a new implementation of RequestCultureProvider

A new implementation of RequestCultureProvider can be created that determines the request culture information from a custom source. For example, the custom source can be a configuration file or database.

The following example shows `AppSettingsRequestCultureProvider`, which extends the RequestCultureProvider to determine the request culture information from `appsettings.json`:

C#

```
public class AppSettingsRequestCultureProvider : RequestCultureProvider
{
    public string CultureKey { get; set; } = "culture";

    public string UICultureKey { get; set; } = "ui-culture";

    public override Task<ProviderCultureResult>
DetermineProviderCultureResult(HttpContext httpContext)
    {
        if (httpContext == null)
        {
            throw new ArgumentNullException();
        }

        var configuration =
httpContext.RequestServices.GetService<IConfigurationRoot>();
        var culture = configuration[CultureKey];
        var uiCulture = configuration[UICultureKey];

        if (culture == null && uiCulture == null)
        {
            return Task.FromResult((ProviderCultureResult)null);
        }

        if (culture != null && uiCulture == null)
        {
            uiCulture = culture;
        }

        if (culture == null && uiCulture != null)
        {
            culture = uiCulture;
        }

        var providerResultCulture = new ProviderCultureResult(culture,
uiCulture);

        return Task.FromResult(providerResultCulture);
    }
}
```

Localization resources

ASP.NET Core localization provides ResourceManagerStringLocalizer. ResourceManagerStringLocalizer is an implementation of IStringLocalizer that uses `resx` to store localization resources.

You aren't limited to using `resx` files. By implementing `IStringLocalizer`, any data source can be used.

The following example projects implement IStringLocalizer:

- EFStringLocalizer
- JsonStringLocalizer
- SqlLocalizer

Troubleshoot ASP.NET Core localization

Article • 05/03/2024

By Hisham Bin Ateya

This article provides instructions on how to diagnose ASP.NET Core app localization issues.

Localization configuration issues

Localization middleware order

The app may not localize because the localization middleware isn't ordered as expected.

To resolve this issue, ensure that localization middleware is registered before MVC middleware. Otherwise, the localization middleware isn't applied.

```C#
public void ConfigureServices(IServiceCollection services)
{
    services.AddLocalization(options => options.ResourcesPath =
"Resources");

    services.AddMvc();
}
```

Localization resources path not found

Supported Cultures in RequestCultureProvider don't match with registered once

Resource file naming issues

ASP.NET Core has predefined rules and guidelines for localization resources file naming, which are described in Globalization and localization in ASP.NET Core.

Missing resources

Common causes of resources not being found include:

- Resource names are misspelled in either the .NET XML resource file (`.resx`) or the localizer request.

- The resource is missing from the resource file for some languages, but exists in others.
- If you're still having trouble, check the localization log messages (logged at the `Debug` log level) for more details about the missing resources.

> **Tip**
>
> When using CookieRequestCultureProvider, verify single quotes aren't used with the cultures inside the localization cookie value. For example, `c='en-UK'|uic='en-US'` is an invalid cookie value, while `c=en-UK|uic=en-US` is valid.

Resources and class libraries issues

ASP.NET Core by default provides a way to allow the class libraries to find their resource files via ResourceLocationAttribute.

Common issues with class libraries include:

- Missing the ResourceLocationAttribute in a class library prevents ResourceManagerStringLocalizerFactory from discovering the resources.
- Resource file naming. For more information, see the Resource file naming issues section.
- Changing the root namespace of the class library. For more information, see the Root namespace issues section.

`CustomRequestCultureProvider` doesn't work as expected

The RequestLocalizationOptions class has three default providers:

- QueryStringRequestCultureProvider
- CookieRequestCultureProvider
- AcceptLanguageHeaderRequestCultureProvider

The CustomRequestCultureProvider allows you to customize how the localization culture is provided. The CustomRequestCultureProvider is used when the default providers don't meet your requirements.

A common reason for a custom provider not working properly is that it isn't the first provider in the RequestCultureProviders list. To resolve this issue:

- Insert the custom provider at position zero in the RequestCultureProviders list:

```C#
options.AddInitialRequestCultureProvider(
    new CustomRequestCultureProvider(async context =>
    {
        // My custom request culture logic
        return new ProviderCultureResult("en");
    }));
```

- Use the AddInitialRequestCultureProvider extension method to set the custom provider as the initial provider.

Root namespace issues

When the root namespace of an assembly is different than the assembly name, localization doesn't work by default. To avoid this issue use the RootNamespace attribute, which is described in Globalization and localization in ASP.NET Core.

⚠ Warning

A root namespace issue can occur when a project's name isn't a valid .NET identifier. For instance, `my-project-name.csproj` uses the root namespace `my_project_name` and the assembly name `my-project-name`, which results in this error.

Resources and build action

If you use resource files for localization, it's important that they have an appropriate build action. Use **Embedded Resource**; otherwise, the `ResourceStringLocalizer` isn't able to find these resources.

Location override using "Sensors" pane in developer tools

When using the location override using the **Sensors** pane in Google Chrome or Microsoft Edge developer tools, the fallback language is reset after prerendering. Avoid setting the language using the **Sensors** pane when testing. Set the language using the browser's language settings.

For more information, see Blazor Localization does not work with InteractiveServer (dotnet/aspnetcore #53707).

GitHub issues with helpful problem solving tips

- Please add more info about shared files (dotnet/AspNetCore.Docs #28674
- Blazor Localization does not work with InteractiveServer (dotnet/aspnetcore #53707) (Location override using "Sensors" pane)

Model Binding in ASP.NET Core

Article • 09/27/2024

ⓘ **Important**

This information relates to a pre-release product that may be substantially modified before it's commercially released. Microsoft makes no warranties, express or implied, with respect to the information provided here.

For the current release, see the .NET 9 version of this article.

This article explains what model binding is, how it works, and how to customize its behavior.

What is Model binding

Controllers and Razor pages work with data that comes from HTTP requests. For example, route data may provide a record key, and posted form fields may provide values for the properties of the model. Writing code to retrieve each of these values and convert them from strings to .NET types would be tedious and error-prone. Model binding automates this process. The model binding system:

- Retrieves data from various sources such as route data, form fields, and query strings.
- Provides the data to controllers and Razor pages in method parameters and public properties.
- Converts string data to .NET types.
- Updates properties of complex types.

Example

Suppose you have the following action method:

C#

```
[HttpGet("{id}")]
public ActionResult<Pet> GetById(int id, bool dogsOnly)
```

And the app receives a request with this URL:

HTTP

```
https://contoso.com/api/pets/2?DogsOnly=true
```

Model binding goes through the following steps after the routing system selects the action method:

- Finds the first parameter of `GetById`, an integer named `id`.
- Looks through the available sources in the HTTP request and finds `id` = "2" in route data.
- Converts the string "2" into integer 2.
- Finds the next parameter of `GetById`, a boolean named `dogsOnly`.
- Looks through the sources and finds "DogsOnly=true" in the query string. Name matching is not case-sensitive.
- Converts the string "true" into boolean `true`.

The framework then calls the `GetById` method, passing in 2 for the `id` parameter, and `true` for the `dogsOnly` parameter.

In the preceding example, the model binding targets are method parameters that are simple types. Targets may also be the properties of a complex type. After each property is successfully bound, model validation occurs for that property. The record of what data is bound to the model, and any binding or validation errors, is stored in ControllerBase.ModelState or PageModel.ModelState. To find out if this process was successful, the app checks the ModelState.IsValid flag.

Targets

Model binding tries to find values for the following kinds of targets:

- Parameters of the controller action method that a request is routed to.
- Parameters of the Razor Pages handler method that a request is routed to.
- Public properties of a controller or `PageModel` class, if specified by attributes.

[BindProperty] attribute

Can be applied to a public property of a controller or `PageModel` class to cause model binding to target that property:

C#

```
public class EditModel : PageModel
{
    [BindProperty]
    public Instructor? Instructor { get; set; }

    // ...
}
```

[BindProperties] attribute

Can be applied to a controller or `PageModel` class to tell model binding to target all public properties of the class:

```C#
[BindProperties]
public class CreateModel : PageModel
{
    public Instructor? Instructor { get; set; }

    // ...
}
```

Model binding for HTTP GET requests

By default, properties are not bound for HTTP GET requests. Typically, all you need for a GET request is a record ID parameter. The record ID is used to look up the item in the database. Therefore, there is no need to bind a property that holds an instance of the model. In scenarios where you do want properties bound to data from GET requests, set the `SupportsGet` property to `true`:

```C#
[BindProperty(Name = "ai_user", SupportsGet = true)]
public string? ApplicationInsightsCookie { get; set; }
```

Model binding simple and complex types

Model binding uses specific definitions for the types it operates on. A *simple type* is converted from a single string using TypeConverter or a `TryParse` method. A *complex type* is converted from multiple input values. The framework determines the difference based on the existence of a `TypeConverter` or `TryParse`. We recommend creating a type converter or using `TryParse` for a `string` to `SomeType` conversion that doesn't require external resources or multiple inputs.

Sources

By default, model binding gets data in the form of key-value pairs from the following sources in an HTTP request:

1. Form fields
2. The request body (For controllers that have the [ApiController] attribute.)
3. Route data
4. Query string parameters
5. Uploaded files

For each target parameter or property, the sources are scanned in the order indicated in the preceding list. There are a few exceptions:

- Route data and query string values are used only for simple types.
- Uploaded files are bound only to target types that implement `IFormFile` or `IEnumerable<IFormFile>`.

If the default source is not correct, use one of the following attributes to specify the source:

- [FromQuery] - Gets values from the query string.
- [FromRoute] - Gets values from route data.
- [FromForm] - Gets values from posted form fields.
- [FromBody] - Gets values from the request body.
- [FromHeader] - Gets values from HTTP headers.

These attributes:

- Are added to model properties individually and not to the model class, as in the following example:

  C#

  ```
  public class Instructor
  {
      public int Id { get; set; }

      [FromQuery(Name = "Note")]
      public string? NoteFromQueryString { get; set; }
  ```

```
        // ...
    }
```

- Optionally accept a model name value in the constructor. This option is provided in case the property name doesn't match the value in the request. For instance, the value in the request might be a header with a hyphen in its name, as in the following example:

  C#

  ```csharp
  public void OnGet([FromHeader(Name = "Accept-Language")] string
  language)
  ```

[FromBody] attribute

Apply the `[FromBody]` attribute to a parameter to populate its properties from the body of an HTTP request. The ASP.NET Core runtime delegates the responsibility of reading the body to an input formatter. Input formatters are explained later in this article.

When `[FromBody]` is applied to a complex type parameter, any binding source attributes applied to its properties are ignored. For example, the following `Create` action specifies that its `pet` parameter is populated from the body:

C#

```csharp
public ActionResult<Pet> Create([FromBody] Pet pet)
```

The `Pet` class specifies that its `Breed` property is populated from a query string parameter:

C#

```csharp
public class Pet
{
    public string Name { get; set; } = null!;

    [FromQuery] // Attribute is ignored.
    public string Breed { get; set; } = null!;
}
```

In the preceding example:

- The `[FromQuery]` attribute is ignored.

- The `Breed` property is not populated from a query string parameter.

Input formatters read only the body and don't understand binding source attributes. If a suitable value is found in the body, that value is used to populate the `Breed` property.

Don't apply `[FromBody]` to more than one parameter per action method. Once the request stream is read by an input formatter, it's no longer available to be read again for binding other `[FromBody]` parameters.

Additional sources

Source data is provided to the model binding system by *value providers*. You can write and register custom value providers that get data for model binding from other sources. For example, you might want data from cookies or session state. To get data from a new source:

- Create a class that implements `IValueProvider`.
- Create a class that implements `IValueProviderFactory`.
- Register the factory class in `Program.cs`.

The sample includes a value provider and factory example that gets values from cookies. Register custom value provider factories in `Program.cs`:

```C#
builder.Services.AddControllers(options =>
{
    options.ValueProviderFactories.Add(new CookieValueProviderFactory());
});
```

The preceding code puts the custom value provider after all built-in value providers. To make it the first in the list, call `Insert(0, new CookieValueProviderFactory())` instead of `Add`.

No source for a model property

By default, a model state error isn't created if no value is found for a model property. The property is set to null or a default value:

- Nullable simple types are set to `null`.
- Non-nullable value types are set to `default(T)`. For example, a parameter `int id` is set to 0.

- For complex Types, model binding creates an instance by using the default constructor, without setting properties.
- Arrays are set to `Array.Empty<T>()`, except that `byte[]` arrays are set to `null`.

If model state should be invalidated when nothing is found in form fields for a model property, use the [BindRequired] attribute.

Note that this `[BindRequired]` behavior applies to model binding from posted form data, not to JSON or XML data in a request body. Request body data is handled by input formatters.

Type conversion errors

If a source is found but can't be converted into the target type, model state is flagged as invalid. The target parameter or property is set to null or a default value, as noted in the previous section.

In an API controller that has the `[ApiController]` attribute, invalid model state results in an automatic HTTP 400 response.

In a Razor page, redisplay the page with an error message:

C#

```csharp
public IActionResult OnPost()
{
    if (!ModelState.IsValid)
    {
        return Page();
    }

    // ...

    return RedirectToPage("./Index");
}
```

When the page is redisplayed by the preceding code, the invalid input isn't shown in the form field. This is because the model property has been set to null or a default value. The invalid input does appear in an error message. If you want to redisplay the bad data in the form field, consider making the model property a string and doing the data conversion manually.

The same strategy is recommended if you don't want type conversion errors to result in model state errors. In that case, make the model property a string.

Simple types

See Model binding simple and complex types for explanation of simple and complex types.

The simple types that the model binder can convert source strings into include the following:

- Boolean
- Byte, SByte
- Char
- DateOnly
- DateTime
- DateTimeOffset
- Decimal
- Double
- Enum
- Guid
- Int16, Int32, Int64
- Single
- TimeOnly
- TimeSpan
- UInt16, UInt32, UInt64
- Uri
- Version

Bind with `IParsable<T>.TryParse`

The IParsable<TSelf>.TryParse API supports binding controller action parameter values:

C#

```
public static bool TryParse (string? s, IFormatProvider? provider, out TSelf
result);
```

The following `DateRange` class implements IParsable<TSelf> to support binding a date range:

C#

```
public class DateRange : IParsable<DateRange>
{
    public DateOnly? From { get; init; }
    public DateOnly? To { get; init; }

    public static DateRange Parse(string value, IFormatProvider? provider)
    {
        if (!TryParse(value, provider, out var result))
        {
           throw new ArgumentException("Could not parse supplied value.",
nameof(value));
        }

        return result;
    }

    public static bool TryParse(string? value,
                                IFormatProvider? provider, out DateRange
dateRange)
    {
        var segments = value?.Split(',',
StringSplitOptions.RemoveEmptyEntries
                                       | StringSplitOptions.TrimEntries);

        if (segments?.Length == 2
            && DateOnly.TryParse(segments[0], provider, out var fromDate)
            && DateOnly.TryParse(segments[1], provider, out var toDate))
        {
            dateRange = new DateRange { From = fromDate, To = toDate };
            return true;
        }

        dateRange = new DateRange { From = default, To = default };
        return false;
    }
}
```

The preceding code:

- Converts a string representing two dates to a `DateRange` object
- The model binder uses the IParsable<TSelf>.TryParse method to bind the `DateRange`.

The following controller action uses the `DateRange` class to bind a date range:

C#

```
// GET /WeatherForecast/ByRange?range=7/24/2022,07/26/2022
public IActionResult ByRange([FromQuery] DateRange range)
{
    if (!ModelState.IsValid)
        return View("Error", ModelState.Values.SelectMany(v => v.Errors));

    var weatherForecasts = Enumerable
        .Range(1, 5).Select(index => new WeatherForecast
        {
            Date = DateTime.Now.AddDays(index),
            TemperatureC = Random.Shared.Next(-20, 55),
            Summary = Summaries[Random.Shared.Next(Summaries.Length)]
        })
        .Where(wf => DateOnly.FromDateTime(wf.Date) >= range.From
                     && DateOnly.FromDateTime(wf.Date) <= range.To)
        .Select(wf => new WeatherForecastViewModel
        {
            Date = wf.Date.ToString("d"),
            TemperatureC = wf.TemperatureC,
            TemperatureF = 32 + (int)(wf.TemperatureC / 0.5556),
            Summary = wf.Summary
        });

    return View("Index", weatherForecasts);
}
```

The following `Locale` class implements IParsable<TSelf> to support binding to `CultureInfo`:

```C#
public class Locale : CultureInfo, IParsable<Locale>
{
    public Locale(string culture) : base(culture)
    {
    }

    public static Locale Parse(string value, IFormatProvider? provider)
    {
        if (!TryParse(value, provider, out var result))
        {
           throw new ArgumentException("Could not parse supplied value.",
nameof(value));
        }

        return result;
    }

    public static bool TryParse([NotNullWhen(true)] string? value,
                                IFormatProvider? provider, out Locale
locale)
    {
        if (value is null)
        {
            locale = new Locale(CurrentCulture.Name);
            return false;
        }

        try
        {
            locale = new Locale(value);
            return true;
        }
        catch (CultureNotFoundException)
        {
            locale = new Locale(CurrentCulture.Name);
            return false;
        }
    }
}
```

The following controller action uses the `Locale` class to bind a `CultureInfo` string:

C#

```csharp
// GET /en-GB/WeatherForecast
public IActionResult Index([FromRoute] Locale locale)
{
    var weatherForecasts = Enumerable
        .Range(1, 5).Select(index => new WeatherForecast
        {
            Date = DateTime.Now.AddDays(index),
            TemperatureC = Random.Shared.Next(-20, 55),
            Summary = Summaries[Random.Shared.Next(Summaries.Length)]
        })
        .Select(wf => new WeatherForecastViewModel
        {
            Date = wf.Date.ToString("d", locale),
            TemperatureC = wf.TemperatureC,
            TemperatureF = 32 + (int)(wf.TemperatureC / 0.5556),
            Summary = wf.Summary
        });

    return View(weatherForecasts);
}
```

The following controller action uses the `DateRange` and `Locale` classes to bind a date range with `CultureInfo`:

C#

```csharp
// GET /af-ZA/WeatherForecast/RangeByLocale?range=2022-07-24,2022-07-29
public IActionResult RangeByLocale([FromRoute] Locale locale, [FromQuery]
string range)
{
    if (!ModelState.IsValid)
        return View("Error", ModelState.Values.SelectMany(v => v.Errors));

    if (!DateRange.TryParse(range, locale, out DateRange rangeResult))
    {
        ModelState.TryAddModelError(nameof(range),
            $"Invalid date range: {range} for locale {locale.DisplayName}");
        return View("Error", ModelState.Values.SelectMany(v => v.Errors));
    }

    var weatherForecasts = Enumerable
        .Range(1, 5).Select(index => new WeatherForecast
        {
            Date = DateTime.Now.AddDays(index),
            TemperatureC = Random.Shared.Next(-20, 55),
            Summary = Summaries[Random.Shared.Next(Summaries.Length)]
        })
        .Where(wf => DateOnly.FromDateTime(wf.Date) >= rangeResult.From
                     && DateOnly.FromDateTime(wf.Date) <= rangeResult.To)
        .Select(wf => new WeatherForecastViewModel
        {
            Date = wf.Date.ToString("d", locale),
            TemperatureC = wf.TemperatureC,
            TemperatureF = 32 + (int) (wf.TemperatureC / 0.5556),
            Summary = wf.Summary
        });

    return View("Index", weatherForecasts);
}
```

The API sample app on GitHub shows the preceding sample for an API controller.

Bind with `TryParse`

The `TryParse` API supports binding controller action parameter values:

C#

```
public static bool TryParse(string value, T out result);
public static bool TryParse(string value, IFormatProvider provider, T out
result);
```

IParsable<T>.TryParse is the recommended approach for parameter binding because unlike `TryParse`, it doesn't depend on reflection.

The following `DateRangeTP` class implements `TryParse`:

C#

```
public class DateRangeTP
{
    public DateOnly? From { get; }
    public DateOnly? To { get; }

    public DateRangeTP(string from, string to)
    {
        if (string.IsNullOrEmpty(from))
            throw new ArgumentNullException(nameof(from));
        if (string.IsNullOrEmpty(to))
            throw new ArgumentNullException(nameof(to));

        From = DateOnly.Parse(from);
        To = DateOnly.Parse(to);
    }

    public static bool TryParse(string? value, out DateRangeTP? result)
    {
        var range = value?.Split(',', StringSplitOptions.RemoveEmptyEntries
| StringSplitOptions.TrimEntries);
        if (range?.Length != 2)
        {
            result = default;
            return false;
        }

        result = new DateRangeTP(range[0], range[1]);
        return true;
    }
}
```

The following controller action uses the `DateRangeTP` class to bind a date range:

C#

```
// GET /WeatherForecast/ByRangeTP?range=7/24/2022,07/26/2022
public IActionResult ByRangeTP([FromQuery] DateRangeTP range)
{
    if (!ModelState.IsValid)
        return View("Error", ModelState.Values.SelectMany(v => v.Errors));

    var weatherForecasts = Enumerable
        .Range(1, 5).Select(index => new WeatherForecast
        {
            Date = DateTime.Now.AddDays(index),
            TemperatureC = Random.Shared.Next(-20, 55),
            Summary = Summaries[Random.Shared.Next(Summaries.Length)]
        })
        .Where(wf => DateOnly.FromDateTime(wf.Date) >= range.From
                     && DateOnly.FromDateTime(wf.Date) <= range.To)
        .Select(wf => new WeatherForecastViewModel
        {
            Date = wf.Date.ToString("d"),
            TemperatureC = wf.TemperatureC,
            TemperatureF = 32 + (int)(wf.TemperatureC / 0.5556),
            Summary = wf.Summary
        });
    return View("Index", weatherForecasts);
}
```

Complex types

A complex type must have a public default constructor and public writable properties to bind. When model binding occurs, the class is instantiated using the public default constructor.

For each property of the complex type, model binding looks through the sources for the name pattern *prefix.property_name*. If nothing is found, it looks for just *property_name* without the prefix. The decision to use the prefix isn't made per property. For example, with a query containing `?Instructor.Id=100&Name=foo`, bound to method `OnGet(Instructor instructor)`, the resulting object of type `Instructor` contains:

- `Id` set to `100`.
- `Name` set to `null`. Model binding expects `Instructor.Name` because `Instructor.Id` was used in the preceding query parameter.

For binding to a parameter, the prefix is the parameter name. For binding to a `PageModel` public property, the prefix is the public property name. Some attributes have a `Prefix` property that lets you override the default usage of parameter or property name.

For example, suppose the complex type is the following `Instructor` class:

C#

```csharp
public class Instructor
{
    public int ID { get; set; }
    public string LastName { get; set; }
    public string FirstName { get; set; }
}
```

Prefix = parameter name

If the model to be bound is a parameter named `instructorToUpdate`:

C#

```csharp
public IActionResult OnPost(int? id, Instructor instructorToUpdate)
```

Model binding starts by looking through the sources for the key `instructorToUpdate.ID`. If that isn't found, it looks for `ID` without a prefix.

Prefix = property name

If the model to be bound is a property named `Instructor` of the controller or `PageModel` class:

```C#
[BindProperty]
public Instructor Instructor { get; set; }
```

Model binding starts by looking through the sources for the key `Instructor.ID`. If that isn't found, it looks for `ID` without a prefix.

Custom prefix

If the model to be bound is a parameter named `instructorToUpdate` and a `Bind` attribute specifies `Instructor` as the prefix:

```C#
public IActionResult OnPost(
    int? id, [Bind(Prefix = "Instructor")] Instructor instructorToUpdate)
```

Model binding starts by looking through the sources for the key `Instructor.ID`. If that isn't found, it looks for `ID` without a prefix.

Attributes for complex type targets

Several built-in attributes are available for controlling model binding of complex types:

- [Bind]
- [BindRequired]
- [BindNever]

⚠ **Warning**

These attributes affect model binding when posted form data is the source of values. They do ***not*** affect input formatters, which process posted JSON and XML request bodies. Input formatters are explained later in this article.

[Bind] attribute

Can be applied to a class or a method parameter. Specifies which properties of a model should be included in model binding. `[Bind]` does ***not*** affect input formatters.

In the following example, only the specified properties of the `Instructor` model are bound when any handler or action method is called:

```C#
[Bind("LastName,FirstMidName,HireDate")]
public class Instructor
```

In the following example, only the specified properties of the `Instructor` model are bound when the `OnPost` method is called:

```C#
[HttpPost]
public IActionResult OnPost(
    [Bind("LastName,FirstMidName,HireDate")] Instructor instructor)
```

The `[Bind]` attribute can be used to protect against overposting in *create* scenarios. It doesn't work well in edit scenarios because excluded properties are set to null or a default value instead of being left unchanged. For protection against overposting, view models are recommended rather than the `[Bind]` attribute. For more information, see Security note about overposting.

[ModelBinder] attribute

ModelBinderAttribute can be applied to types, properties, or parameters. It allows specifying the type of model binder used to bind the specific instance or type. For example:

```C#
[HttpPost]
public IActionResult OnPost(
    [ModelBinder<MyInstructorModelBinder>] Instructor instructor)
```

The `[ModelBinder]` attribute can also be used to change the name of a property or parameter when it's being model bound:

```C#
public class Instructor
{
    [ModelBinder(Name = "instructor_id")]
    public string Id { get; set; }

    // ...
}
```

[BindRequired] attribute

Causes model binding to add a model state error if binding cannot occur for a model's property. Here's an example:

```C#
public class InstructorBindRequired
{
    // ...

    [BindRequired]
    public DateTime HireDate { get; set; }
}
```

See also the discussion of the `[Required]` attribute in Model validation.

[BindNever] attribute

Can be applied to a property or a type. Prevents model binding from setting a model's property. When applied to a type, the model binding system excludes all properties the type defines. Here's an example:

```C#
public class InstructorBindNever
{
    [BindNever]
    public int Id { get; set; }

    // ...
}
```

Collections

For targets that are collections of simple types, model binding looks for matches to *parameter_name* or *property_name*. If no match is found, it looks for one of the supported formats without the prefix. For example:

- Suppose the parameter to be bound is an array named `selectedCourses`:

  ```C#
  public IActionResult OnPost(int? id, int[] selectedCourses)
  ```

- Form or query string data can be in one of the following formats:

  ```
  selectedCourses=1050&selectedCourses=2000
  ```

  ```
  selectedCourses[0]=1050&selectedCourses[1]=2000
  ```

  ```
  [0]=1050&[1]=2000
  ```

  ```
  selectedCourses[a]=1050&selectedCourses[b]=2000&selectedCourses.index=a
  &selectedCourses.index=b
  ```

  ```
  [a]=1050&[b]=2000&index=a&index=b
  ```

  Avoid binding a parameter or a property named `index` or `Index` if it is adjacent to a collection value. Model binding attempts to use `index` as the index for the collection which might result in incorrect binding. For example, consider the following action:

  C#

```
public IActionResult Post(string index, List<Product> products)
```

In the preceding code, the `index` query string parameter binds to the `index` method parameter and also is used to bind the product collection. Renaming the `index` parameter or using a model binding attribute to configure binding avoids this issue:

```C#
public IActionResult Post(string productIndex, List<Product> products)
```

- The following format is supported only in form data:

  ```
  selectedCourses[]=1050&selectedCourses[]=2000
  ```

- For all of the preceding example formats, model binding passes an array of two items to the `selectedCourses` parameter:
  - selectedCourses[0]=1050
  - selectedCourses[1]=2000

  Data formats that use subscript numbers (... [0] ... [1] ...) must ensure that they are numbered sequentially starting at zero. If there are any gaps in subscript numbering, all items after the gap are ignored. For example, if the subscripts are 0 and 2 instead of 0 and 1, the second item is ignored.

Dictionaries

For `Dictionary` targets, model binding looks for matches to *parameter_name* or *property_name*. If no match is found, it looks for one of the supported formats without the prefix. For example:

- Suppose the target parameter is a `Dictionary<int, string>` named `selectedCourses`:

  ```C#
  public IActionResult OnPost(int? id, Dictionary<int, string>
  selectedCourses)
  ```

- The posted form or query string data can look like one of the following examples:

  ```
  selectedCourses[1050]=Chemistry&selectedCourses[2000]=Economics
  ```

  ```
  [1050]=Chemistry&selectedCourses[2000]=Economics
  ```

  ```
  selectedCourses[0].Key=1050&selectedCourses[0].Value=Chemistry&
  selectedCourses[1].Key=2000&selectedCourses[1].Value=Economics
  ```

  ```
  [0].Key=1050&[0].Value=Chemistry&[1].Key=2000&[1].Value=Economics
  ```

- For all of the preceding example formats, model binding passes a dictionary of two items to the `selectedCourses` parameter:
  - selectedCourses["1050"]="Chemistry"
  - selectedCourses["2000"]="Economics"

Constructor binding and record types

Model binding requires that complex types have a parameterless constructor. Both `System.Text.Json` and `Newtonsoft.Json` based input formatters support deserialization of classes that don't have a parameterless constructor.

Record types are a great way to succinctly represent data over the network. ASP.NET Core supports model binding and validating record types with a single constructor:

```C#
public record Person(
    [Required] string Name, [Range(0, 150)] int Age, [BindNever] int Id);

public class PersonController
{
    public IActionResult Index() => View();

    [HttpPost]
    public IActionResult Index(Person person)
```

```
    {
        // ...
    }
}
```

`Person/Index.cshtml`:

CSHTML

```cshtml
@model Person

<label>Name: <input asp-for="Name" /></label>
<br />
<label>Age: <input asp-for="Age" /></label>
```

When validating record types, the runtime searches for binding and validation metadata specifically on parameters rather than on properties.

The framework allows binding to and validating record types:

C#

```csharp
public record Person([Required] string Name, [Range(0, 100)] int Age);
```

For the preceding to work, the type must:

- Be a record type.
- Have exactly one public constructor.
- Contain parameters that have a property with the same name and type. The names must not differ by case.

POCOs without parameterless constructors

POCOs that do not have parameterless constructors can't be bound.

The following code results in an exception saying that the type must have a parameterless constructor:

C#

```csharp
public class Person(string Name)

public record Person([Required] string Name, [Range(0, 100)] int Age)
{
    public Person(string Name) : this (Name, 0);
}
```

Record types with manually authored constructors

Record types with manually authored constructors that look like primary constructors work

C#

```
public record Person
{
    public Person([Required] string Name, [Range(0, 100)] int Age)
        => (this.Name, this.Age) = (Name, Age);

    public string Name { get; set; }
    public int Age { get; set; }
}
```

Record types, validation and binding metadata

For record types, validation and binding metadata on parameters is used. Any metadata on properties is ignored

C#

```
public record Person (string Name, int Age)
{
   [BindProperty(Name = "SomeName")] // This does not get used
   [Required] // This does not get used
   public string Name { get; init; }
}
```

Validation and metadata

Validation uses metadata on the parameter but uses the property to read the value. In the ordinary case with primary constructors, the two would be identical. However, there are ways to defeat it:

C#

```
public record Person([Required] string Name)
{
    private readonly string _name;
    // The following property is never null.
    // However this object could have been constructed as "new
Person(null)".
    public string Name { get; init => _name = value ?? string.Empty; }
}
```

TryUpdateModel does not update parameters on a record type

```C#
public record Person(string Name)
{
    public int Age { get; set; }
}

var person = new Person("initial-name");
TryUpdateModel(person, ...);
```

In this case, MVC will not attempt to bind `Name` again. However, `Age` is allowed to be updated

Globalization behavior of model binding route data and query strings

The ASP.NET Core route value provider and query string value provider:

- Treat values as invariant culture.
- Expect that URLs are culture-invariant.

In contrast, values coming from form data undergo a culture-sensitive conversion. This is by design so that URLs are shareable across locales.

To make the ASP.NET Core route value provider and query string value provider undergo a culture-sensitive conversion:

- Inherit from IValueProviderFactory
- Copy the code from QueryStringValueProviderFactory or RouteValueValueProviderFactory
- Replace the culture value passed to the value provider constructor with CultureInfo.CurrentCulture
- Replace the default value provider factory in MVC options with your new one:

C#

```
public class CultureQueryStringValueProviderFactory : IValueProviderFactory
{
    public Task CreateValueProviderAsync(ValueProviderFactoryContext
context)
    {
        _ = context ?? throw new ArgumentNullException(nameof(context));

        var query = context.ActionContext.HttpContext.Request.Query;
        if (query?.Count > 0)
        {
            context.ValueProviders.Add(
                new QueryStringValueProvider(
                    BindingSource.Query,
                    query,
                    CultureInfo.CurrentCulture));
        }

        return Task.CompletedTask;
    }
}
```

C#

```
builder.Services.AddControllers(options =>
{
    var index = options.ValueProviderFactories.IndexOf(

options.ValueProviderFactories.OfType<QueryStringValueProviderFactory>()
            .Single());

    options.ValueProviderFactories[index] =
        new CultureQueryStringValueProviderFactory();
});
```

Special data types

There are some special data types that model binding can handle.

IFormFile and IFormFileCollection

An uploaded file included in the HTTP request. Also supported is `IEnumerable<IFormFile>` for multiple files.

CancellationToken

Actions can optionally bind a `CancellationToken` as a parameter. This binds RequestAborted that signals when the connection underlying the HTTP request is aborted. Actions can use this parameter to cancel long running async operations that are executed as part of the controller actions.

FormCollection

Used to retrieve all the values from posted form data.

Input formatters

Data in the request body can be in JSON, XML, or some other format. To parse this data, model binding uses an *input formatter* that is configured to handle a particular content type. By default, ASP.NET Core includes JSON based input formatters for handling JSON data. You can add other formatters for other content types.

ASP.NET Core selects input formatters based on the Consumes attribute. If no attribute is present, it uses the Content-Type header.

To use the built-in XML input formatters:

- In `Program.cs`, call AddXmlSerializerFormatters or AddXmlDataContractSerializerFormatters.

  C#

  ```
  builder.Services.AddControllers()
      .AddXmlSerializerFormatters();
  ```

- Apply the `Consumes` attribute to controller classes or action methods that should expect XML in the request body.

  C#

  ```
  [HttpPost]
  [Consumes("application/xml")]
  public ActionResult<Pet> Create(Pet pet)
  ```

  For more information, see Introducing XML Serialization.

Customize model binding with input formatters

An input formatter takes full responsibility for reading data from the request body. To customize this process, configure the APIs used by the input formatter. This section describes how to customize the `System.Text.Json`-based input formatter to understand a custom type named `ObjectId`.

Consider the following model, which contains a custom `ObjectId` property:

C#

```csharp
public class InstructorObjectId
{
    [Required]
    public ObjectId ObjectId { get; set; } = null!;
}
```

To customize the model binding process when using `System.Text.Json`, create a class derived from JsonConverter<T>:

C#

```csharp
internal class ObjectIdConverter : JsonConverter<ObjectId>
{
    public override ObjectId Read(
        ref Utf8JsonReader reader, Type typeToConvert, JsonSerializerOptions
options)
        => new(JsonSerializer.Deserialize<int>(ref reader, options));

    public override void Write(
        Utf8JsonWriter writer, ObjectId value, JsonSerializerOptions
options)
        => writer.WriteNumberValue(value.Id);
}
```

To use a custom converter, apply the JsonConverterAttribute attribute to the type. In the following example, the `ObjectId` type is configured with `ObjectIdConverter` as its custom converter:

C#

```csharp
[JsonConverter(typeof(ObjectIdConverter))]
public record ObjectId(int Id);
```

For more information, see How to write custom converters.

Exclude specified types from model binding

The model binding and validation systems' behavior is driven by ModelMetadata. You can customize `ModelMetadata` by adding a details provider to MvcOptions.ModelMetadataDetailsProviders. Built-in details providers are available for disabling model binding or validation for specified types.

To disable model binding on all models of a specified type, add an ExcludeBindingMetadataProvider in `Program.cs`. For example, to disable model binding on all models of type `System.Version`:

C#

```csharp
builder.Services.AddRazorPages()
    .AddMvcOptions(options =>
    {
        options.ModelMetadataDetailsProviders.Add(
            new ExcludeBindingMetadataProvider(typeof(Version)));
        options.ModelMetadataDetailsProviders.Add(
            new SuppressChildValidationMetadataProvider(typeof(Guid)));
    });
```

To disable validation on properties of a specified type, add a SuppressChildValidationMetadataProvider in `Program.cs`. For example, to disable validation on properties of type `System.Guid`:

C#

```csharp
builder.Services.AddRazorPages()
    .AddMvcOptions(options =>
    {
        options.ModelMetadataDetailsProviders.Add(
            new ExcludeBindingMetadataProvider(typeof(Version)));
        options.ModelMetadataDetailsProviders.Add(
            new SuppressChildValidationMetadataProvider(typeof(Guid)));
    });
```

Custom model binders

You can extend model binding by writing a custom model binder and using the `[ModelBinder]` attribute to select it for a given target. Learn more about custom model binding.

Manual model binding

Model binding can be invoked manually by using the TryUpdateModelAsync method. The method is defined on both `ControllerBase` and `PageModel` classes. Method overloads let you specify the prefix and value provider to use. The method returns `false` if model binding fails. Here's an example:

```C#
if (await TryUpdateModelAsync(
    newInstructor,
    "Instructor",
    x => x.Name, x => x.HireDate!))
{
    _instructorStore.Add(newInstructor);
    return RedirectToPage("./Index");
}

return Page();
```

TryUpdateModelAsync uses value providers to get data from the form body, query string, and route data. `TryUpdateModelAsync` is typically:

- Used with Razor Pages and MVC apps using controllers and views to prevent over-posting.
- Not used with a web API unless consumed from form data, query strings, and route data. Web API endpoints that consume JSON use Input formatters to deserialize the request body into an object.

For more information, see TryUpdateModelAsync.

[FromServices] attribute

This attribute's name follows the pattern of model binding attributes that specify a data source. But it's not about binding data from a value provider. It gets an instance of a type from the dependency injection container. Its purpose is to provide an alternative to constructor injection for when you need a service only if a particular method is called.

If an instance of the type isn't registered in the dependency injection container, the app throws an exception when attempting to bind the parameter. To make the parameter optional, use one of the following approaches:

- Make the parameter nullable.
- Set a default value for the parameter.

For nullable parameters, ensure that the parameter isn't `null` before accessing it.

Additional resources

- View or download sample code (how to download)
- Model validation in ASP.NET Core MVC
- Custom Model Binding in ASP.NET Core

Custom Model Binding in ASP.NET Core

Article • 04/05/2024

By Kirk Larkin

Model binding allows controller actions to work directly with model types (passed in as method arguments), rather than HTTP requests. Mapping between incoming request data and application models is handled by model binders. Developers can extend the built-in model binding functionality by implementing custom model binders (though typically, you don't need to write your own provider).

View or download sample code (how to download)

Default model binder limitations

The default model binders support most of the common .NET Core data types and should meet most developers' needs. They expect to bind text-based input from the request directly to model types. You might need to transform the input prior to binding it. For example, when you have a key that can be used to look up model data. You can use a custom model binder to fetch data based on the key.

Model binding simple and complex types

Model binding uses specific definitions for the types it operates on. A *simple type* is converted from a single string using TypeConverter or a `TryParse` method. A *complex type* is converted from multiple input values. The framework determines the difference based on the existence of a `TypeConverter` or `TryParse`. We recommend creating a type converter or using `TryParse` for a `string` to `SomeType` conversion that doesn't require external resources or multiple inputs.

See Simple types for a list of types that the model binder can convert from strings.

Before creating your own custom model binder, it's worth reviewing how existing model binders are implemented. Consider the ByteArrayModelBinder which can be used to convert base64-encoded strings into byte arrays. The byte arrays are often stored as files or database BLOB fields.

Working with the ByteArrayModelBinder

Base64-encoded strings can be used to represent binary data. For example, an image can be encoded as a string. The sample includes an image as a base64-encoded string in Base64String.txt.

ASP.NET Core MVC can take a base64-encoded string and use a `ByteArrayModelBinder` to convert it into a byte array. The ByteArrayModelBinderProvider maps `byte[]` arguments to `ByteArrayModelBinder`:

C#

```csharp
public IModelBinder GetBinder(ModelBinderProviderContext context)
{
    if (context == null)
    {
        throw new ArgumentNullException(nameof(context));
    }

    if (context.Metadata.ModelType == typeof(byte[]))
    {
        var loggerFactory =
context.Services.GetRequiredService<ILoggerFactory>();
        return new ByteArrayModelBinder(loggerFactory);
    }

    return null;
}
```

When creating your own custom model binder, you can implement your own `IModelBinderProvider` type, or use the ModelBinderAttribute.

The following example shows how to use `ByteArrayModelBinder` to convert a base64-encoded string to a `byte[]` and save the result to a file:

C#

```csharp
[HttpPost]
public void Post([FromForm] byte[] file, string filename)
{
    // Don't trust the file name sent by the client. Use
    // Path.GetRandomFileName to generate a safe random
    // file name. _targetFilePath receives a value
    // from configuration (the appsettings.json file in
    // the sample app).
    var trustedFileName = Path.GetRandomFileName();
    var filePath = Path.Combine(_targetFilePath, trustedFileName);

    if (System.IO.File.Exists(filePath))
    {
        return;
    }

    System.IO.File.WriteAllBytes(filePath, file);
}
```

If you would like to see code comments translated to languages other than English, let us know in this GitHub discussion issue.

You can POST a base64-encoded string to the previous api method using a tool like curl.

As long as the binder can bind request data to appropriately named properties or arguments, model binding will succeed. The following example shows how to use `ByteArrayModelBinder` with a view model:

```C#
[HttpPost("Profile")]
public void SaveProfile([FromForm] ProfileViewModel model)
{
    // Don't trust the file name sent by the client. Use
    // Path.GetRandomFileName to generate a safe random
    // file name. _targetFilePath receives a value
    // from configuration (the appsettings.json file in
    // the sample app).
    var trustedFileName = Path.GetRandomFileName();
    var filePath = Path.Combine(_targetFilePath, trustedFileName);

    if (System.IO.File.Exists(filePath))
    {
        return;
    }

    System.IO.File.WriteAllBytes(filePath, model.File);
}

public class ProfileViewModel
{
    public byte[] File { get; set; }
    public string FileName { get; set; }
}
```

Custom model binder sample

In this section we'll implement a custom model binder that:

- Converts incoming request data into strongly typed key arguments.
- Uses Entity Framework Core to fetch the associated entity.

- Passes the associated entity as an argument to the action method.

The following sample uses the `ModelBinder` attribute on the `Author` model:

```C#
using CustomModelBindingSample.Binders;
using Microsoft.AspNetCore.Mvc;

namespace CustomModelBindingSample.Data
{
    [ModelBinder(BinderType = typeof(AuthorEntityBinder))]
    public class Author
    {
        public int Id { get; set; }
        public string Name { get; set; }
        public string GitHub { get; set; }
        public string Twitter { get; set; }
        public string BlogUrl { get; set; }
    }
}
```

In the preceding code, the `ModelBinder` attribute specifies the type of `IModelBinder` that should be used to bind `Author` action parameters.

The following `AuthorEntityBinder` class binds an `Author` parameter by fetching the entity from a data source using Entity Framework Core and an `authorId`:

```C#
public class AuthorEntityBinder : IModelBinder
{
    private readonly AuthorContext _context;

    public AuthorEntityBinder(AuthorContext context)
    {
        _context = context;
    }

    public Task BindModelAsync(ModelBindingContext bindingContext)
    {
        if (bindingContext == null)
        {
            throw new ArgumentNullException(nameof(bindingContext));
        }

        var modelName = bindingContext.ModelName;

        // Try to fetch the value of the argument by name
        var valueProviderResult =
bindingContext.ValueProvider.GetValue(modelName);
```

```
        if (valueProviderResult == ValueProviderResult.None)
        {
            return Task.CompletedTask;
        }

        bindingContext.ModelState.SetModelValue(modelName,
valueProviderResult);

        var value = valueProviderResult.FirstValue;

        // Check if the argument value is null or empty
        if (string.IsNullOrEmpty(value))
        {
            return Task.CompletedTask;
        }

        if (!int.TryParse(value, out var id))
        {
            // Non-integer arguments result in model state errors
            bindingContext.ModelState.TryAddModelError(
                modelName, "Author Id must be an integer.");

            return Task.CompletedTask;
        }

        // Model will be null if not found, including for
        // out of range id values (0, -3, etc.)
        var model = _context.Authors.Find(id);
        bindingContext.Result = ModelBindingResult.Success(model);
        return Task.CompletedTask;
    }
}
```

ⓘ **Note**

The preceding `AuthorEntityBinder` class is intended to illustrate a custom model binder. The class isn't intended to illustrate best practices for a lookup scenario. For lookup, bind the `authorId` and query the database in an action method. This approach separates model binding failures from `NotFound` cases.

The following code shows how to use the `AuthorEntityBinder` in an action method:

C#

```
[HttpGet("get/{author}")]
public IActionResult Get(Author author)
{
    if (author == null)
    {
        return NotFound();
    }

    return Ok(author);
}
```

The `ModelBinder` attribute can be used to apply the `AuthorEntityBinder` to parameters that don't use default conventions:

```C#
[HttpGet("{id}")]
public IActionResult GetById([ModelBinder(Name = "id")] Author author)
{
    if (author == null)
    {
        return NotFound();
    }

    return Ok(author);
}
```

In this example, since the name of the argument isn't the default `authorId`, it's specified on the parameter using the `ModelBinder` attribute. Both the controller and action method are simplified compared to looking up the entity in the action method. The logic to fetch the author using Entity Framework Core is moved to the model binder. This can be a considerable simplification when you have several methods that bind to the `Author` model.

You can apply the `ModelBinder` attribute to individual model properties (such as on a viewmodel) or to action method parameters to specify a certain model binder or model name for just that type or action.

Implementing a ModelBinderProvider

Instead of applying an attribute, you can implement `IModelBinderProvider`. This is how the built-in framework binders are implemented. When you specify the type your binder operates on, you specify the type of argument it produces, **not** the input your binder accepts. The following binder provider works with the `AuthorEntityBinder`. When it's added to MVC's collection of providers, you don't need to use the `ModelBinder` attribute on `Author` or `Author`-typed parameters.

C#

```
using CustomModelBindingSample.Data;
using Microsoft.AspNetCore.Mvc.ModelBinding;
using Microsoft.AspNetCore.Mvc.ModelBinding.Binders;
using System;

namespace CustomModelBindingSample.Binders
{
    public class AuthorEntityBinderProvider : IModelBinderProvider
    {
        public IModelBinder GetBinder(ModelBinderProviderContext context)
        {
            if (context == null)
            {
                throw new ArgumentNullException(nameof(context));
            }

            if (context.Metadata.ModelType == typeof(Author))
            {
                return new
BinderTypeModelBinder(typeof(AuthorEntityBinder));
            }

            return null;
        }
    }
}
```

> Note: The preceding code returns a `BinderTypeModelBinder`. `BinderTypeModelBinder` acts as a factory for model binders and provides dependency injection (DI). The `AuthorEntityBinder` requires DI to access EF Core. Use `BinderTypeModelBinder` if your model binder requires services from DI.

To use a custom model binder provider, add it in `ConfigureServices`:

C#

```
public void ConfigureServices(IServiceCollection services)
{
    services.AddDbContext<AuthorContext>(options =>
options.UseInMemoryDatabase("Authors"));

    services.AddControllers(options =>
    {
        options.ModelBinderProviders.Insert(0, new
AuthorEntityBinderProvider());
    });
}
```

When evaluating model binders, the collection of providers is examined in order. The first provider that returns a binder that matches the input model is used. Adding your provider to the end of the collection may thus result in a built-in model binder being called before your custom binder has a chance. In this example, the custom provider is added to the beginning of the collection to ensure it's always used for `Author` action arguments.

Polymorphic model binding

Binding to different models of derived types is known as polymorphic model binding. Polymorphic custom model binding is required when the request value must be bound to the specific derived model type. Polymorphic model binding:

- Isn't typical for a REST API that's designed to interoperate with all languages.
- Makes it difficult to reason about the bound models.

However, if an app requires polymorphic model binding, an implementation might look like the following code:

C#

```csharp
public abstract class Device
{
    public string Kind { get; set; }
}

public class Laptop : Device
{
    public string CPUIndex { get; set; }
}

public class SmartPhone : Device
{
    public string ScreenSize { get; set; }
}

public class DeviceModelBinderProvider : IModelBinderProvider
{
    public IModelBinder GetBinder(ModelBinderProviderContext context)
    {
        if (context.Metadata.ModelType != typeof(Device))
        {
            return null;
        }

        var subclasses = new[] { typeof(Laptop), typeof(SmartPhone), };

        var binders = new Dictionary<Type, (ModelMetadata, IModelBinder)>();
        foreach (var type in subclasses)
        {
            var modelMetadata = 
context.MetadataProvider.GetMetadataForType(type);
            binders[type] = (modelMetadata, 
context.CreateBinder(modelMetadata));
        }

        return new DeviceModelBinder(binders);
    }
}

public class DeviceModelBinder : IModelBinder
{
    private Dictionary<Type, (ModelMetadata, IModelBinder)> binders;

    public DeviceModelBinder(Dictionary<Type, (ModelMetadata, IModelBinder)> 
binders)
    {
        this.binders = binders;
    }

    public async Task BindModelAsync(ModelBindingContext bindingContext)
    {
        var modelKindName = 
ModelNames.CreatePropertyModelName(bindingContext.ModelName, 
nameof(Device.Kind));
        var modelTypeValue = 
bindingContext.ValueProvider.GetValue(modelKindName).FirstValue;

        IModelBinder modelBinder;
        ModelMetadata modelMetadata;
        if (modelTypeValue == "Laptop")
        {
            (modelMetadata, modelBinder) = binders[typeof(Laptop)];
        }
        else if (modelTypeValue == "SmartPhone")
        {
            (modelMetadata, modelBinder) = binders[typeof(SmartPhone)];
        }
        else
        {
            bindingContext.Result = ModelBindingResult.Failed();
            return;
        }

        var newBindingContext = 
DefaultModelBindingContext.CreateBindingContext(
            bindingContext.ActionContext,
            bindingContext.ValueProvider,
            modelMetadata,
            bindingInfo: null,
            bindingContext.ModelName);

        await modelBinder.BindModelAsync(newBindingContext);
        bindingContext.Result = newBindingContext.Result;
        if (newBindingContext.Result.IsModelSet)
        {
            // Setting the ValidationState ensures properties on derived
types are correctly
            bindingContext.ValidationState[newBindingContext.Result.Model] =
new ValidationStateEntry
            {
                Metadata = modelMetadata,
            };
        }
    }
}
```

Recommendations and best practices

Custom model binders:

- Shouldn't attempt to set status codes or return results (for example, 404 Not Found). If model binding fails, an action filter or logic within the action method itself should handle the failure.
- Are most useful for eliminating repetitive code and cross-cutting concerns from action methods.
- Typically shouldn't be used to convert a string into a custom type, a TypeConverter is usually a better option.

Model validation in ASP.NET Core MVC and Razor Pages

Article • 08/30/2024

This article explains how to validate user input in an ASP.NET Core MVC or Razor Pages app.

View or download sample code (how to download).

Model state

Model state represents errors that come from two subsystems: model binding and model validation. Errors that originate from model binding are generally data conversion errors. For example, an "x" is entered in an integer field. Model validation occurs after model binding and reports errors where data doesn't conform to business rules. For example, a 0 is entered in a field that expects a rating between 1 and 5.

Both model binding and model validation occur before the execution of a controller action or a Razor Pages handler method. For web apps, it's the app's responsibility to inspect `ModelState.IsValid` and react appropriately. Web apps typically redisplay the page with an error message, as shown in the following Razor Pages example:

C#

```csharp
public async Task<IActionResult> OnPostAsync()
{
    if (!ModelState.IsValid)
    {
        return Page();
    }

    _context.Movies.Add(Movie);
    await _context.SaveChangesAsync();

    return RedirectToPage("./Index");
}
```

For ASP.NET Core MVC with controllers and views, the following example shows how to check `ModelState.IsValid` inside of a controller action:

C#

```
public async Task<IActionResult> Create(Movie movie)
{
    if (!ModelState.IsValid)
    {
        return View(movie);
    }

    _context.Movies.Add(movie);
    await _context.SaveChangesAsync();

    return RedirectToAction(nameof(Index));
}
```

Web API controllers don't have to check `ModelState.IsValid` if they have the [ApiController] attribute. In that case, an automatic HTTP 400 response containing error details is returned when model state is invalid. For more information, see Automatic HTTP 400 responses.

Rerun validation

Validation is automatic, but you might want to repeat it manually. For example, you might compute a value for a property and want to rerun validation after setting the property to the computed value. To rerun validation, call ModelStateDictionary.ClearValidationState to clear validation specific to the model being validated followed by `TryValidateModel`:

C#

```
public async Task<IActionResult> OnPostTryValidateAsync()
{
    var modifiedReleaseDate = DateTime.Now.Date;
    Movie.ReleaseDate = modifiedReleaseDate;

    ModelState.ClearValidationState(nameof(Movie));
    if (!TryValidateModel(Movie, nameof(Movie)))
    {
        return Page();
    }

    _context.Movies.Add(Movie);
    await _context.SaveChangesAsync();

    return RedirectToPage("./Index");
}
```

Validation attributes

Validation attributes let you specify validation rules for model properties. The following example from the sample app shows a model class that is annotated with validation attributes. The `[ClassicMovie]` attribute is a custom validation attribute and the others are built in. Not shown is `[ClassicMovieWithClientValidator]`, which shows an alternative way to implement a custom attribute.

C#

```
public class Movie
{
    public int Id { get; set; }

    [Required]
    [StringLength(100)]
    public string Title { get; set; } = null!;

    [ClassicMovie(1960)]
    [DataType(DataType.Date)]
    [Display(Name = "Release Date")]
    public DateTime ReleaseDate { get; set; }

    [Required]
    [StringLength(1000)]
    public string Description { get; set; } = null!;

    [Range(0, 999.99)]
    public decimal Price { get; set; }

    public Genre Genre { get; set; }

    public bool Preorder { get; set; }
}
```

Built-in attributes

Here are some of the built-in validation attributes:

- [ValidateNever]: Indicates that a property or parameter should be excluded from validation.
- [CreditCard]: Validates that the property has a credit card format. Requires jQuery Validation Additional Methods.
- [Compare]: Validates that two properties in a model match.
- [EmailAddress]: Validates that the property has an email format.
- [Phone]: Validates that the property has a telephone number format.

- [Range]: Validates that the property value falls within a specified range.
- [RegularExpression]: Validates that the property value matches a specified regular expression.
- [Required]: Validates that the field isn't null. See [Required] attribute for details about this attribute's behavior.
- [StringLength]: Validates that a string property value doesn't exceed a specified length limit.
- [Url]: Validates that the property has a URL format.
- [Remote]: Validates input on the client by calling an action method on the server. See [Remote] attribute for details about this attribute's behavior.

A complete list of validation attributes can be found in the System.ComponentModel.DataAnnotations namespace.

Error messages

Validation attributes let you specify the error message to be displayed for invalid input. For example:

```C#
[StringLength(8, ErrorMessage = "Name length can't be more than 8.")]
```

Internally, the attributes call String.Format with a placeholder for the field name and sometimes additional placeholders. For example:

```C#
[StringLength(8, ErrorMessage = "{0} length must be between {2} and {1}.",
MinimumLength = 6)]
```

When applied to a `Name` property, the error message created by the preceding code would be "Name length must be between 6 and 8.".

To find out which parameters are passed to `String.Format` for a particular attribute's error message, see the DataAnnotations source code.

Use JSON property names in validation errors

By default, when a validation error occurs, model validation produces a ModelStateDictionary with the property name as the error key. Some apps, such as single page apps, benefit from using JSON property names for validation errors generated from Web APIs. The following code configures validation to use the SystemTextJsonValidationMetadataProvider to use JSON property names:

C#

```csharp
using Microsoft.AspNetCore.Mvc.ModelBinding.Metadata;

var builder = WebApplication.CreateBuilder(args);

builder.Services.AddControllers(options =>
{
    options.ModelMetadataDetailsProviders.Add(new
SystemTextJsonValidationMetadataProvider());
});

var app = builder.Build();

app.UseHttpsRedirection();

app.UseAuthorization();

app.MapControllers();

app.Run();
```

The following code configures validation to use the NewtonsoftJsonValidationMetadataProvider to use JSON property name when using Json.NET:

C#

```csharp
using Microsoft.AspNetCore.Mvc.NewtonsoftJson;

var builder = WebApplication.CreateBuilder(args);

builder.Services.AddControllers(options =>
{
    options.ModelMetadataDetailsProviders.Add(new
NewtonsoftJsonValidationMetadataProvider());
}).AddNewtonsoftJson();

var app = builder.Build();

app.UseHttpsRedirection();

app.UseAuthorization();

app.MapControllers();

app.Run();
```

For an example of the policy to use camel-casing, see Program.cs on GitHub .

Non-nullable reference types and [Required] attribute

The validation system treats non-nullable parameters or bound properties as if they had a `[Required(AllowEmptyStrings = true)]` attribute. By enabling Nullable contexts, MVC implicitly starts validating non-nullable properties or parameters as if they had been attributed with the `[Required(AllowEmptyStrings = true)]` attribute. Consider the following code:

C#

```
public class Person
{
    public string Name { get; set; }
}
```

If the app was built with `<Nullable>enable</Nullable>`, a missing value for `Name` in a JSON or form post results in a validation error. This may seem contradictory since the `[Required(AllowEmptyStrings = true)]` attribute is implied, but this is expected behavior because empty strings are converted to null by default. Use a nullable reference type to allow null or missing values to be specified for the `Name` property:

C#

```
public class Person
{
    public string? Name { get; set; }
}
```

This behavior can be disabled by configuring SuppressImplicitRequiredAttributeForNonNullableReferenceTypes in `Program.cs`:

C#

```
builder.Services.AddControllers(
    options =>
options.SuppressImplicitRequiredAttributeForNonNullableReferenceTypes =
true);
```

[Required] validation on the server

On the server, a required value is considered missing if the property is null. A non-nullable field is always valid, and the `[Required]` attribute's error message is never displayed.

However, model binding for a non-nullable property may fail, resulting in an error message such as `The value '' is invalid`. To specify a custom error message for server-side validation of non-nullable types, you have the following options:

- Make the field nullable (for example, `decimal?` instead of `decimal`). Nullable<T> value types are treated like standard nullable types.
- Specify the default error message to be used by model binding, as shown in the following example:

  ```C#
  builder.Services.AddRazorPages()
      .AddMvcOptions(options =>
      {
          options.MaxModelValidationErrors = 50;

  options.ModelBindingMessageProvider.SetValueMustNotBeNullAccessor(
              _ => "The field is required.");
      });

  builder.Services.AddSingleton
      <IValidationAttributeAdapterProvider,
  CustomValidationAttributeAdapterProvider>();
  ```

  For more information about model binding errors that you can set default messages for, see DefaultModelBindingMessageProvider.

[Required] validation on the client

Non-nullable types and strings are handled differently on the client compared to the server. On the client:

- A value is considered present only if input is entered for it. Therefore, client-side validation handles non-nullable types the same as nullable types.
- Whitespace in a string field is considered valid input by the jQuery Validation required method. Server-side validation considers a required string field invalid if only whitespace is entered.

As noted earlier, non-nullable types are treated as though they had a `[Required(AllowEmptyStrings = true)]` attribute. That means you get client-side validation even if you don't apply the `[Required(AllowEmptyStrings = true)]` attribute. But if you don't use the attribute, you get a default error message. To specify a custom error message, use the attribute.

[Remote] attribute

The [Remote] attribute implements client-side validation that requires calling a method on the server to determine whether field input is valid. For example, the app may need to verify whether a user name is already in use.

To implement remote validation:

1. Create an action method for JavaScript to call. The jQuery Validation remote method expects a JSON response:

   - `true` means the input data is valid.
   - `false`, `undefined`, or `null` means the input is invalid. Display the default error message.
   - Any other string means the input is invalid. Display the string as a custom error message.

   Here's an example of an action method that returns a custom error message:

   C#

   ```
   [AcceptVerbs("GET", "POST")]
   public IActionResult VerifyEmail(string email)
   {
       if (!_userService.VerifyEmail(email))
       {
           return Json($"Email {email} is already in use.");
       }

       return Json(true);
   }
   ```

2. In the model class, annotate the property with a `[Remote]` attribute that points to the validation action method, as shown in the following example:

   C#

   ```
   [Remote(action: "VerifyEmail", controller: "Users")]
   public string Email { get; set; } = null!;
   ```

Server side validation also needs to be implemented for clients that have disabled JavaScript.

Additional fields

The AdditionalFields property of the `[Remote]` attribute lets you validate combinations of fields against data on the server. For example, if the `User` model had `FirstName` and `LastName` properties, you might want to verify that no existing users already have that pair of names. The following example shows how to use `AdditionalFields`:

```csharp
[Remote(action: "VerifyName", controller: "Users", AdditionalFields =
nameof(LastName))]
[Display(Name = "First Name")]
public string FirstName { get; set; } = null!;

[Remote(action: "VerifyName", controller: "Users", AdditionalFields =
nameof(FirstName))]
[Display(Name = "Last Name")]
public string LastName { get; set; } = null!;
```

`AdditionalFields` could be set explicitly to the strings "FirstName" and "LastName", but using the nameof operator simplifies later refactoring. The action method for this validation must accept both `firstName` and `lastName` arguments:

```csharp
[AcceptVerbs("GET", "POST")]
public IActionResult VerifyName(string firstName, string lastName)
{
    if (!_userService.VerifyName(firstName, lastName))
    {
        return Json($"A user named {firstName} {lastName} already exists.");
    }

    return Json(true);
}
```

When the user enters a first or last name, JavaScript makes a remote call to see if that pair of names has been taken.

To validate two or more additional fields, provide them as a comma-delimited list. For example, to add a `MiddleName` property to the model, set the `[Remote]` attribute as shown in the following example:

C#

```
[Remote(action: "VerifyName", controller: "Users",
    AdditionalFields = nameof(FirstName) + "," + nameof(LastName))]
public string MiddleName { get; set; }
```

`AdditionalFields`, like all attribute arguments, must be a constant expression. Therefore, don't use an interpolated string or call Join to initialize `AdditionalFields`.

Alternatives to built-in attributes

If you need validation not provided by built-in attributes, you can:

- Create custom attributes.
- Implement IValidatableObject.

Custom attributes

For scenarios that the built-in validation attributes don't handle, you can create custom validation attributes. Create a class that inherits from ValidationAttribute, and override the IsValid method.

The `IsValid` method accepts an object named *value*, which is the input to be validated. An overload also accepts a ValidationContext object, which provides additional information, such as the model instance created by model binding.

The following example validates that the release date for a movie in the *Classic* genre isn't later than a specified year. The `[ClassicMovie]` attribute:

- Is only run on the server.
- For Classic movies, validates the release date:

C#

```
public class ClassicMovieAttribute : ValidationAttribute
{
    public ClassicMovieAttribute(int year)
        => Year = year;

    public int Year { get; }

    public string GetErrorMessage() =>
        $"Classic movies must have a release year no later than {Year}.";

    protected override ValidationResult? IsValid(
        object? value, ValidationContext validationContext)
    {
        var movie = (Movie)validationContext.ObjectInstance;
        var releaseYear = ((DateTime)value!).Year;

        if (movie.Genre == Genre.Classic && releaseYear > Year)
        {
            return new ValidationResult(GetErrorMessage());
        }

        return ValidationResult.Success;
    }
}
```

The `movie` variable in the preceding example represents a `Movie` object that contains the data from the form submission. When validation fails, a ValidationResult with an error message is returned.

IValidatableObject

The preceding example works only with `Movie` types. Another option for class-level validation is to implement IValidatableObject in the model class, as shown in the following example:

C#

```csharp
public class ValidatableMovie : IValidatableObject
{
    private const int _classicYear = 1960;

    public int Id { get; set; }

    [Required]
    [StringLength(100)]
    public string Title { get; set; } = null!;

    [DataType(DataType.Date)]
    [Display(Name = "Release Date")]
    public DateTime ReleaseDate { get; set; }

    [Required]
    [StringLength(1000)]
    public string Description { get; set; } = null!;

    [Range(0, 999.99)]
    public decimal Price { get; set; }

    public Genre Genre { get; set; }

    public bool Preorder { get; set; }
    public IEnumerable<ValidationResult> Validate(ValidationContext
validationContext)
    {
        if (Genre == Genre.Classic && ReleaseDate.Year > _classicYear)
        {
            yield return new ValidationResult(
                $"Classic movies must have a release year no later than
{_classicYear}.",
                new[] { nameof(ReleaseDate) });
        }
    }
}
```

Custom validation

The following code shows how to add a model error after examining the model:

C#

```
if (Contact.Name == Contact.ShortName)
{
    ModelState.AddModelError("Contact.ShortName",
                             "Short name can't be the same as Name.");
}
```

The following code implements the validation test in a controller:

C#

```
if (contact.Name == contact.ShortName)
{
    ModelState.AddModelError(nameof(contact.ShortName),
                             "Short name can't be the same as Name.");
}
```

The following code verifies the phone number and email are unique:

C#

```
public async Task<IActionResult> OnPostAsync()
{
    // Attach Validation Error Message to the Model on validation failure.

    if (Contact.Name == Contact.ShortName)
    {
        ModelState.AddModelError("Contact.ShortName",
                                 "Short name can't be the same as Name.");
    }

    if (_context.Contact.Any(i => i.PhoneNumber == Contact.PhoneNumber))
    {
        ModelState.AddModelError("Contact.PhoneNumber",
                                  "The Phone number is already in use.");
    }
    if (_context.Contact.Any(i => i.Email == Contact.Email))
    {
        ModelState.AddModelError("Contact.Email", "The Email is already in
use.");
    }

    if (!ModelState.IsValid || _context.Contact == null || Contact == null)
    {
        // if model is invalid, return the page with the model state errors.
        return Page();
    }
    _context.Contact.Add(Contact);
    await _context.SaveChangesAsync();

    return RedirectToPage("./Index");
}
```

The following code implements the validation test in a controller:

C#

```
[HttpPost]
[ValidateAntiForgeryToken]
public async Task<IActionResult>
Create([Bind("Id,Name,ShortName,Email,PhoneNumber")] Contact contact)
{
    // Attach Validation Error Message to the Model on validation failure.
    if (contact.Name == contact.ShortName)
    {
        ModelState.AddModelError(nameof(contact.ShortName),
                                 "Short name can't be the same as Name.");
    }

    if (_context.Contact.Any(i => i.PhoneNumber == contact.PhoneNumber))
    {
        ModelState.AddModelError(nameof(contact.PhoneNumber),
                                  "The Phone number is already in use.");
    }
    if (_context.Contact.Any(i => i.Email == contact.Email))
    {
        ModelState.AddModelError(nameof(contact.Email), "The Email is
already in use.");
    }

    if (ModelState.IsValid)
    {
        _context.Add(contact);
        await _context.SaveChangesAsync();
        return RedirectToAction(nameof(Index));
    }
    return View(contact);
}
```

Checking for a unique phone number or email is typically also done with remote validation.

ValidationResult

Consider the following custom `ValidateNameAttribute`:

C#

```
public class ValidateNameAttribute : ValidationAttribute
{
    public ValidateNameAttribute()
    {
        const string defaultErrorMessage = "Error with Name";
        ErrorMessage ??= defaultErrorMessage;
    }

    protected override ValidationResult? IsValid(object? value,
                                         ValidationContext
validationContext)
    {
        if (value == null || string.IsNullOrWhiteSpace(value.ToString()))
        {
            return new ValidationResult("Name is required.");
        }

        if (value.ToString()!.ToLower().Contains("zz"))
        {

            return new ValidationResult(
                        FormatErrorMessage(validationContext.DisplayName));
        }

        return ValidationResult.Success;
    }
}
```

In the following code, the custom `[ValidateName]` attribute is applied:

C#

```
public class Contact
{
    public Guid Id { get; set; }

    [ValidateName(ErrorMessage = "Name must not contain `zz`")]
    public string? Name { get; set; }
    public string? Email { get; set; }
    public string? PhoneNumber { get; set; }
}
```

When the model contains `zz`, a new ValidationResult is returned.

Top-level node validation

Top-level nodes include:

- Action parameters
- Controller properties
- Page handler parameters
- Page model properties

Model-bound top-level nodes are validated in addition to validating model properties. In the following example from the sample app, the `VerifyPhone` method uses the RegularExpressionAttribute to validate the `phone` action parameter:

```C#
[AcceptVerbs("GET", "POST")]
public IActionResult VerifyPhone(
    [RegularExpression(@"^\d{3}-\d{3}-\d{4}$")] string phone)
{
    if (!ModelState.IsValid)
    {
        return Json($"Phone {phone} has an invalid format. Format: ###-###-
####");
    }

    return Json(true);
}
```

Top-level nodes can use BindRequiredAttribute with validation attributes. In the following example from the sample app, the `CheckAge` method specifies that the `age` parameter must be bound from the query string when the form is submitted:

C#

```
[HttpPost]
public IActionResult CheckAge([BindRequired, FromQuery] int age)
{
```

In the Check Age page (`CheckAge.cshtml`), there are two forms. The first form submits an `Age` value of `99` as a query string parameter: `https://localhost:5001/Users/CheckAge?Age=99`.

When a properly formatted `age` parameter from the query string is submitted, the form validates.

The second form on the Check Age page submits the `Age` value in the body of the request, and validation fails. Binding fails because the `age` parameter must come from a query string.

Maximum errors

Validation stops when the maximum number of errors is reached (200 by default). You can configure this number with the following code in `Program.cs`:

C#

```csharp
builder.Services.AddRazorPages()
    .AddMvcOptions(options =>
    {
        options.MaxModelValidationErrors = 50;
        options.ModelBindingMessageProvider.SetValueMustNotBeNullAccessor(
            _ => "The field is required.");
    });

builder.Services.AddSingleton
    <IValidationAttributeAdapterProvider,
CustomValidationAttributeAdapterProvider>();
```

Maximum recursion

ValidationVisitor traverses the object graph of the model being validated. For models that are deep or are infinitely recursive, validation may result in stack overflow. MvcOptions.MaxValidationDepth provides a way to stop validation early if the visitor recursion exceeds a configured depth. The default value of `MvcOptions.MaxValidationDepth` is 32.

Automatic short-circuit

Validation is automatically short-circuited (skipped) if the model graph doesn't require validation. Objects that the runtime skips validation for include collections of primitives (such as `byte[]`, `string[]`, `Dictionary<string, string>`) and complex object graphs that don't have any validators.

Client-side validation

Client-side validation prevents submission until the form is valid. The Submit button runs JavaScript that either submits the form or displays error messages.

Client-side validation avoids an unnecessary round trip to the server when there are input errors on a form. The following script references in `_Layout.cshtml` and `_ValidationScriptsPartial.cshtml` support client-side validation:

CSHTML

```cshtml
<script src="https://cdnjs.cloudflare.com/ajax/libs/jquery/3.6.0/jquery.js">
</script>
```

CSHTML

```cshtml
<script src="https://cdnjs.cloudflare.com/ajax/libs/jquery-
validate/1.19.3/jquery.validate.js"></script>
<script src="https://cdnjs.cloudflare.com/ajax/libs/jquery-validation-
unobtrusive/3.2.12/jquery.validate.unobtrusive.js"></script>
```

The jQuery Unobtrusive Validation script is a custom Microsoft front-end library that builds on the popular jQuery Validation plugin. Without jQuery Unobtrusive Validation, you would have to code the same validation logic in two places: once in the server-side validation attributes on model properties, and then again in client-side scripts. Instead, Tag Helpers and HTML helpers use the validation attributes and type metadata from model properties to render HTML 5 `data-` attributes for the form elements that need validation. jQuery Unobtrusive Validation parses the `data-` attributes and passes the logic to jQuery Validation, effectively "copying" the server-side validation logic to the client. You can display validation errors on the client using tag helpers as shown here:

CSHTML

```
<div class="form-group">
    <label asp-for="Movie.ReleaseDate" class="control-label"></label>
    <input asp-for="Movie.ReleaseDate" class="form-control" />
    <span asp-validation-for="Movie.ReleaseDate" class="text-danger"></span>
</div>
```

The preceding tag helpers render the following HTML:

```html
<div class="form-group">
    <label class="control-label" for="Movie_ReleaseDate">Release
Date</label>
    <input class="form-control" type="date" data-val="true"
        data-val-required="The Release Date field is required."
        id="Movie_ReleaseDate" name="Movie.ReleaseDate" value="">
    <span class="text-danger field-validation-valid"
        data-valmsg-for="Movie.ReleaseDate" data-valmsg-replace="true">
</span>
</div>
```

Notice that the `data-` attributes in the HTML output correspond to the validation attributes for the `Movie.ReleaseDate` property. The `data-val-required` attribute contains an error message to display if the user doesn't fill in the release date field. jQuery Unobtrusive Validation passes this value to the jQuery Validation required() method, which then displays that message in the accompanying **<span>** element.

Data type validation is based on the .NET type of a property, unless that is overridden by a [DataType] attribute. Browsers have their own default error messages, but the jQuery Validation Unobtrusive Validation package can override those messages. `[DataType]` attributes and subclasses such as [EmailAddress] let you specify the error message.

Unobtrusive validation

For information on unobtrusive validation, see this GitHub issue.

Add Validation to Dynamic Forms

jQuery Unobtrusive Validation passes validation logic and parameters to jQuery Validation when the page first loads. Therefore, validation doesn't work automatically on dynamically generated forms. To enable validation, tell jQuery Unobtrusive Validation to parse the dynamic form immediately after you create it. For example, the following code sets up client-side validation on a form added via AJAX.

JavaScript

```
$.get({
    url: "https://url/that/returns/a/form",
    dataType: "html",
    error: function(jqXHR, textStatus, errorThrown) {
        alert(textStatus + ": Couldn't add form. " + errorThrown);
    },
    success: function(newFormHTML) {
        var container = document.getElementById("form-container");
        container.insertAdjacentHTML("beforeend", newFormHTML);
        var forms = container.getElementsByTagName("form");
        var newForm = forms[forms.length - 1];
        $.validator.unobtrusive.parse(newForm);
    }
})
```

The `$.validator.unobtrusive.parse()` method accepts a jQuery selector for its one argument. This method tells jQuery Unobtrusive Validation to parse the `data-` attributes of forms within that selector. The values of those attributes are then passed to the jQuery Validation plugin.

Add Validation to Dynamic Controls

The `$.validator.unobtrusive.parse()` method works on an entire form, not on individual dynamically generated controls, such as `<input>` and `<select/>`. To reparse the form, remove the validation data that was added when the form was parsed earlier, as shown in the following example:

JavaScript

```
$.get({
    url: "https://url/that/returns/a/control",
    dataType: "html",
    error: function(jqXHR, textStatus, errorThrown) {
        alert(textStatus + ": Couldn't add control. " + errorThrown);
    },
    success: function(newInputHTML) {
        var form = document.getElementById("my-form");
        form.insertAdjacentHTML("beforeend", newInputHTML);
        $(form).removeData("validator")    // Added by jQuery Validation
               .removeData("unobtrusiveValidation");   // Added by jQuery
Unobtrusive Validation
        $.validator.unobtrusive.parse(form);
    }
})
```

Custom client-side validation

Custom client-side validation is done by generating `data-` HTML attributes that work with a custom jQuery Validation adapter. The following sample adapter code was written for the `[ClassicMovie]` and `[ClassicMovieWithClientValidator]` attributes that were introduced earlier in this article:

JavaScript

```javascript
$.validator.addMethod('classicmovie', function (value, element, params) {
    var genre = $(params[0]).val(), year = params[1], date = new
Date(value);

    // The Classic genre has a value of '0'.
    if (genre && genre.length > 0 && genre[0] === '0') {
        // The release date for a Classic is valid if it's no greater than
the given year.
        return date.getUTCFullYear() <= year;
    }

    return true;
});

$.validator.unobtrusive.adapters.add('classicmovie', ['year'], function
(options) {
    var element = $(options.form).find('select#Movie_Genre')[0];

    options.rules['classicmovie'] = [element,
parseInt(options.params['year'])];
    options.messages['classicmovie'] = options.message;
});
```

For information about how to write adapters, see the jQuery Validation documentation.

The use of an adapter for a given field is triggered by `data-` attributes that:

- Flag the field as being subject to validation (`data-val="true"`).
- Identify a validation rule name and error message text (for example, `data-val-rulename="Error message."`).
- Provide any additional parameters the validator needs (for example, `data-val-rulename-param1="value"`).

The following example shows the `data-` attributes for the sample app's `ClassicMovie` attribute:

HTML

```
<input class="form-control" type="date"
    data-val="true"
    data-val-classicmovie="Classic movies must have a release year no later
than 1960."
    data-val-classicmovie-year="1960"
    data-val-required="The Release Date field is required."
    id="Movie_ReleaseDate" name="Movie.ReleaseDate" value="">
```

As noted earlier, Tag Helpers and HTML helpers use information from validation attributes to render `data-` attributes. There are two options for writing code that results in the creation of custom `data-` HTML attributes:

- Create a class that derives from AttributeAdapterBase<TAttribute> and a class that implements IValidationAttributeAdapterProvider, and register your attribute and its adapter in DI. This method follows the single responsibility principle in that server-related and client-related validation code is in separate classes. The adapter also has the advantage that since it's registered in DI, other services in DI are available to it if needed.
- Implement IClientModelValidator in your ValidationAttribute class. This method might be appropriate if the attribute doesn't do any server-side validation and doesn't need any services from DI.

AttributeAdapter for client-side validation

This method of rendering `data-` attributes in HTML is used by the `ClassicMovie` attribute in the sample app. To add client validation by using this method:

1. Create an attribute adapter class for the custom validation attribute. Derive the class from AttributeAdapterBase<TAttribute>. Create an `AddValidation` method that adds `data-` attributes to the rendered output, as shown in this example:

```C#
public class ClassicMovieAttributeAdapter :
AttributeAdapterBase<ClassicMovieAttribute>
{
    public ClassicMovieAttributeAdapter(
        ClassicMovieAttribute attribute, IStringLocalizer?
stringLocalizer)
        : base(attribute, stringLocalizer)
    {

    }

    public override void AddValidation(ClientModelValidationContext
context)
    {
        MergeAttribute(context.Attributes, "data-val", "true");
        MergeAttribute(context.Attributes, "data-val-classicmovie",
GetErrorMessage(context));

        var year =
Attribute.Year.ToString(CultureInfo.InvariantCulture);
        MergeAttribute(context.Attributes, "data-val-classicmovie-
year", year);
    }

    public override string GetErrorMessage(ModelValidationContextBase
validationContext)
        => Attribute.GetErrorMessage();
}
```

2. Create an adapter provider class that implements IValidationAttributeAdapterProvider. In the GetAttributeAdapter method pass in the custom attribute to the adapter's constructor, as shown in this example:

C#

```
public class CustomValidationAttributeAdapterProvider :
IValidationAttributeAdapterProvider
{
    private readonly IValidationAttributeAdapterProvider baseProvider =
        new ValidationAttributeAdapterProvider();

    public IAttributeAdapter? GetAttributeAdapter(
        ValidationAttribute attribute, IStringLocalizer?
stringLocalizer)
    {
        if (attribute is ClassicMovieAttribute classicMovieAttribute)
        {
            return new
ClassicMovieAttributeAdapter(classicMovieAttribute, stringLocalizer);
        }

        return baseProvider.GetAttributeAdapter(attribute,
stringLocalizer);
    }
}
```

3. Register the adapter provider for DI in `Program.cs`:

C#

```
builder.Services.AddRazorPages()
    .AddMvcOptions(options =>
    {
        options.MaxModelValidationErrors = 50;

options.ModelBindingMessageProvider.SetValueMustNotBeNullAccessor(
            _ => "The field is required.");
    });

builder.Services.AddSingleton
    <IValidationAttributeAdapterProvider,
CustomValidationAttributeAdapterProvider>();
```

IClientModelValidator for client-side validation

This method of rendering `data-` attributes in HTML is used by the `ClassicMovieWithClientValidator` attribute in the sample app. To add client validation by using this method:

- In the custom validation attribute, implement the IClientModelValidator interface and create an AddValidation method. In the `AddValidation` method, add `data-` attributes for validation, as shown in the following example:

```C#
public class ClassicMovieWithClientValidatorAttribute :
    ValidationAttribute, IClientModelValidator
{
    public ClassicMovieWithClientValidatorAttribute(int year)
        => Year = year;

    public int Year { get; }

    public void AddValidation(ClientModelValidationContext context)
    {
        MergeAttribute(context.Attributes, "data-val", "true");
        MergeAttribute(context.Attributes, "data-val-classicmovie",
GetErrorMessage());

        var year = Year.ToString(CultureInfo.InvariantCulture);
        MergeAttribute(context.Attributes, "data-val-classicmovie-
year", year);
    }

    public string GetErrorMessage() =>
        $"Classic movies must have a release year no later than
{Year}.";

    protected override ValidationResult? IsValid(
        object? value, ValidationContext validationContext)
    {
        var movie = (Movie)validationContext.ObjectInstance;
        var releaseYear = ((DateTime)value!).Year;
        if (movie.Genre == Genre.Classic && releaseYear > Year)
        {
            return new ValidationResult(GetErrorMessage());
        }

        return ValidationResult.Success;
    }

    private static bool MergeAttribute(IDictionary<string, string>
attributes, string key, string value)
    {
        if (attributes.ContainsKey(key))
        {
            return false;
        }

        attributes.Add(key, value);
        return true;
    }
}
```

Disable client-side validation

The following code disables client validation in Razor Pages:

C#

```
builder.Services.AddRazorPages()
    .AddViewOptions(options =>
    {
        options.HtmlHelperOptions.ClientValidationEnabled = false;
    });
```

Other options to disable client-side validation:

- Comment out the reference to `_ValidationScriptsPartial` in all the `.cshtml` files.
- Remove the contents of the *Pages\Shared_ValidationScriptsPartial.cshtml* file.

The preceding approach won't prevent client-side validation of ASP.NET Core Identity Razor class library. For more information, see Scaffold Identity in ASP.NET Core projects.

Problem details

Problem Details are not the only response format to describe an HTTP API error, however, they are commonly used to report errors for HTTP APIs.

The problem details service implements the IProblemDetailsService interface, which supports creating problem details in ASP.NET Core. The AddProblemDetails(IServiceCollection) extension method on IServiceCollection registers the default `IProblemDetailsService` implementation.

In ASP.NET Core apps, the following middleware generates problem details HTTP responses when `AddProblemDetails` is called, except when the Accept request HTTP header doesn't include one of the content types supported by the registered IProblemDetailsWriter (default: `application/json`):

- ExceptionHandlerMiddleware: Generates a problem details response when a custom handler is not defined.
- StatusCodePagesMiddleware: Generates a problem details response by default.
- DeveloperExceptionPageMiddleware: Generates a problem details response in development when the `Accept` request HTTP header doesn't include `text/html`.

Additional resources

- System.ComponentModel.DataAnnotations
- Model Binding

Compatibility version for ASP.NET Core MVC

Article • 06/03/2022

By Rick Anderson

The SetCompatibilityVersion method is a no-op for ASP.NET Core 3.0 apps. That is, calling `SetCompatibilityVersion` with any value of CompatibilityVersion has no impact on the application.

- The next minor version of ASP.NET Core may provide a new `CompatibilityVersion` value.
- `CompatibilityVersion` values `Version_2_0` through `Version_2_2` are marked `[Obsolete(...)]`.
- See Breaking API changes in Antiforgery, CORS, Diagnostics, Mvc, and Routing. This list includes breaking changes for compatibility switches.

To see how `SetCompatibilityVersion` works with ASP.NET Core 2.x apps, select the ASP.NET Core 2.2 version of this article.

Write custom ASP.NET Core middleware

Article • 07/26/2024

Important

This information relates to a pre-release product that may be substantially modified before it's commercially released. Microsoft makes no warranties, express or implied, with respect to the information provided here.

For the current release, see the .NET 9 version of this article.

By Fiyaz Hasan, Rick Anderson, and Steve Smith

Middleware is software that's assembled into an app pipeline to handle requests and responses. ASP.NET Core provides a rich set of built-in middleware components, but in some scenarios you might want to write a custom middleware.

This topic describes how to write *convention-based* middleware. For an approach that uses strong typing and per-request activation, see Factory-based middleware activation in ASP.NET Core.

Middleware class

Middleware is generally encapsulated in a class and exposed with an extension method. Consider the following inline middleware, which sets the culture for the current request from a query string:

C#

```
using System.Globalization;

var builder = WebApplication.CreateBuilder(args);
var app = builder.Build();

app.UseHttpsRedirection();
app.Use(async (context, next) =>
{
    var cultureQuery = context.Request.Query["culture"];
    if (!string.IsNullOrWhiteSpace(cultureQuery))
    {
        var culture = new CultureInfo(cultureQuery);

        CultureInfo.CurrentCulture = culture;
        CultureInfo.CurrentUICulture = culture;
    }

    // Call the next delegate/middleware in the pipeline.
    await next(context);
});

app.Run(async (context) =>
{
    await context.Response.WriteAsync(
        $"CurrentCulture.DisplayName:
{CultureInfo.CurrentCulture.DisplayName}");
});

app.Run();
```

The preceding highlighted inline middleware is used to demonstrate creating a middleware component by calling Microsoft.AspNetCore.Builder.UseExtensions.Use. The preceding `Use` extension method adds a middleware delegate defined in-line to the application's request pipeline.

There are two overloads available for the `Use` extension:

- One takes a HttpContext and a `Func<Task>`. Invoke the `Func<Task>` without any parameters.
- The other takes a `HttpContext` and a RequestDelegate. Invoke the `RequestDelegate` by passing the `HttpContext`.

Prefer using the later overload as it saves two internal per-request allocations that are required when using the other overload.

Test the middleware by passing in the culture. For example, request `https://localhost:5001/?culture=es-es`.

For ASP.NET Core's built-in localization support, see Globalization and localization in ASP.NET Core.

The following code moves the middleware delegate to a class:

C#

```
using System.Globalization;

namespace Middleware.Example;

public class RequestCultureMiddleware
{
    private readonly RequestDelegate _next;

    public RequestCultureMiddleware(RequestDelegate next)
    {
        _next = next;
    }

    public async Task InvokeAsync(HttpContext context)
    {
        var cultureQuery = context.Request.Query["culture"];
        if (!string.IsNullOrWhiteSpace(cultureQuery))
        {
            var culture = new CultureInfo(cultureQuery);

            CultureInfo.CurrentCulture = culture;
            CultureInfo.CurrentUICulture = culture;
        }

        // Call the next delegate/middleware in the pipeline.
        await _next(context);
    }
}
```

The middleware class must include:

- A public constructor with a parameter of type RequestDelegate.
- A public method named `Invoke` or `InvokeAsync`. This method must:
  - Return a `Task`.
  - Accept a first parameter of type HttpContext.

Additional parameters for the constructor and `Invoke`/`InvokeAsync` are populated by dependency injection (DI).

Typically, an extension method is created to expose the middleware through IApplicationBuilder:

C#

```
using System.Globalization;

namespace Middleware.Example;

public class RequestCultureMiddleware
{
    private readonly RequestDelegate _next;

    public RequestCultureMiddleware(RequestDelegate next)
    {
        _next = next;
    }

    public async Task InvokeAsync(HttpContext context)
    {
        var cultureQuery = context.Request.Query["culture"];
        if (!string.IsNullOrWhiteSpace(cultureQuery))
        {
            var culture = new CultureInfo(cultureQuery);

            CultureInfo.CurrentCulture = culture;
            CultureInfo.CurrentUICulture = culture;
        }

        // Call the next delegate/middleware in the pipeline.
        await _next(context);
    }
}

public static class RequestCultureMiddlewareExtensions
{
    public static IApplicationBuilder UseRequestCulture(
        this IApplicationBuilder builder)
    {
        return builder.UseMiddleware<RequestCultureMiddleware>();
    }
}
```

The following code calls the middleware from `Program.cs`:

C#

```
using Middleware.Example;
using System.Globalization;

var builder = WebApplication.CreateBuilder(args);
var app = builder.Build();

app.UseHttpsRedirection();

app.UseRequestCulture();

app.Run(async (context) =>
{
    await context.Response.WriteAsync(
        $"CurrentCulture.DisplayName:
{CultureInfo.CurrentCulture.DisplayName}");
});
```

```
app.Run();
```

Middleware dependencies

Middleware should follow the Explicit Dependencies Principle by exposing its dependencies in its constructor. Middleware is constructed once per *application lifetime*.

Middleware components can resolve their dependencies from dependency injection (DI) through constructor parameters. UseMiddleware can also accept additional parameters directly.

Per-request middleware dependencies

Middleware is constructed at app startup and therefore has application life time. Scoped lifetime services used by middleware constructors aren't shared with other dependency-injected types during each request. To share a *scoped* service between middleware and other types, add these services to the `InvokeAsync` method's signature. The `InvokeAsync` method can accept additional parameters that are populated by DI:

```C#
namespace Middleware.Example;

public class MyCustomMiddleware
{
    private readonly RequestDelegate _next;

    public MyCustomMiddleware(RequestDelegate next)
    {
        _next = next;
    }

    // IMessageWriter is injected into InvokeAsync
    public async Task InvokeAsync(HttpContext httpContext, IMessageWriter 
svc)
    {
        svc.Write(DateTime.Now.Ticks.ToString());
        await _next(httpContext);
    }
}

public static class MyCustomMiddlewareExtensions
{
    public static IApplicationBuilder UseMyCustomMiddleware(
        this IApplicationBuilder builder)
    {
        return builder.UseMiddleware<MyCustomMiddleware>();
    }
}
```

Lifetime and registration options contains a complete sample of middleware with *scoped* lifetime services.

The following code is used to test the preceding middleware:

```C#
using Middleware.Example;
var builder = WebApplication.CreateBuilder(args);

builder.Services.AddScoped<IMessageWriter, LoggingMessageWriter>();

var app = builder.Build();

app.UseHttpsRedirection();

app.UseMyCustomMiddleware();

app.MapGet("/", () => "Hello World!");

app.Run();
```

The `IMessageWriter` interface and implementation:

```C#
namespace Middleware.Example;

public interface IMessageWriter
{
    void Write(string message);
}

public class LoggingMessageWriter : IMessageWriter
{

    private readonly ILogger<LoggingMessageWriter> _logger;

    public LoggingMessageWriter(ILogger<LoggingMessageWriter> logger) =>
        _logger = logger;

    public void Write(string message) =>
        _logger.LogInformation(message);
}
```

Additional resources

- Sample code used in this article
- UseExtensions source on GitHub
- Lifetime and registration options contains a complete sample of middleware with *scoped*, *transient*, and *singleton* lifetime services.
- DEEP DIVE: HOW IS THE ASP.NET CORE MIDDLEWARE PIPELINE BUILT
- ASP.NET Core Middleware
- Test ASP.NET Core middleware
- Migrate HTTP handlers and modules to ASP.NET Core middleware
- App startup in ASP.NET Core
- Request Features in ASP.NET Core
- Factory-based middleware activation in ASP.NET Core
- Middleware activation with a third-party container in ASP.NET Core

Request and response operations in ASP.NET Core

Article • 07/26/2024

> **Important**
>
> This information relates to a pre-release product that may be substantially modified before it's commercially released. Microsoft makes no warranties, express or implied, with respect to the information provided here.
>
> For the current release, see the .NET 9 version of this article.

By Justin Kotalik

This article explains how to read from the request body and write to the response body. Code for these operations might be required when writing middleware. Outside of writing middleware, custom code isn't generally required because the operations are handled by MVC and Razor Pages.

There are two abstractions for the request and response bodies: Stream and Pipe. For request reading, HttpRequest.Body is a Stream, and `HttpRequest.BodyReader` is a PipeReader. For response writing, HttpResponse.Body is a Stream, and `HttpResponse.BodyWriter` is a PipeWriter.

Pipelines are recommended over streams. Streams can be easier to use for some simple operations, but pipelines have a performance advantage and are easier to use in most scenarios. ASP.NET Core is starting to use pipelines instead of streams internally. Examples include:

- `FormReader`
- `TextReader`
- `TextWriter`
- `HttpResponse.WriteAsync`

Streams aren't being removed from the framework. Streams continue to be used throughout .NET, and many stream types don't have pipe equivalents, such as `FileStreams` and `ResponseCompression`.

Stream examples

Suppose the goal is to create a middleware that reads the entire request body as a list of strings, splitting on new lines. A simple stream implementation might look like the following example:

⚠ **Warning**

The following code:

- Is used to demonstrate the problems with not using a pipe to read the request body.
- Is not intended to be used in production apps.

C#

```csharp
private async Task<List<string>> GetListOfStringsFromStream(Stream
requestBody)
{
    // Build up the request body in a string builder.
    StringBuilder builder = new StringBuilder();

    // Rent a shared buffer to write the request body into.
    byte[] buffer = ArrayPool<byte>.Shared.Rent(4096);

    while (true)
    {
        var bytesRemaining = await requestBody.ReadAsync(buffer, offset: 0,
buffer.Length);
        if (bytesRemaining == 0)
        {
            break;
        }

        // Append the encoded string into the string builder.
        var encodedString = Encoding.UTF8.GetString(buffer, 0,
bytesRemaining);
        builder.Append(encodedString);
    }

    ArrayPool<byte>.Shared.Return(buffer);

    var entireRequestBody = builder.ToString();

    // Split on \n in the string.
    return new List<string>(entireRequestBody.Split("\n"));
}
```

If you would like to see code comments translated to languages other than English, let us know in this GitHub discussion issue.

This code works, but there are some issues:

- Before appending to the `StringBuilder`, the example creates another string (`encodedString`) that is thrown away immediately. This process occurs for all bytes in the stream, so the result is extra memory allocation the size of the entire request body.
- The example reads the entire string before splitting on new lines. It's more efficient to check for new lines in the byte array.

Here's an example that fixes some of the preceding issues:

⚠ **Warning**

The following code:

- Is used to demonstrate the solutions to some problems in the preceding code while not solving all the problems.
- Is not intended to be used in production apps.

C#

```csharp
private async Task<List<string>>
GetListOfStringsFromStreamMoreEfficient(Stream requestBody)
{
    StringBuilder builder = new StringBuilder();
    byte[] buffer = ArrayPool<byte>.Shared.Rent(4096);
    List<string> results = new List<string>();

    while (true)
    {
        var bytesRemaining = await requestBody.ReadAsync(buffer, offset: 0,
buffer.Length);

        if (bytesRemaining == 0)
        {
            results.Add(builder.ToString());
            break;
        }

        // Instead of adding the entire buffer into the StringBuilder
        // only add the remainder after the last \n in the array.
        var prevIndex = 0;
        int index;
        while (true)
        {
            index = Array.IndexOf(buffer, (byte)'\n', prevIndex);
            if (index == -1)
            {
                break;
            }

            var encodedString = Encoding.UTF8.GetString(buffer, prevIndex,
index - prevIndex);

            if (builder.Length > 0)
            {
                // If there was a remainder in the string buffer, include it
in the next string.
                results.Add(builder.Append(encodedString).ToString());
                builder.Clear();
            }
            else
            {
                results.Add(encodedString);
            }

            // Skip past last \n
            prevIndex = index + 1;
        }

        var remainingString = Encoding.UTF8.GetString(buffer, prevIndex,
bytesRemaining - prevIndex);
        builder.Append(remainingString);
    }

    ArrayPool<byte>.Shared.Return(buffer);

    return results;
}
```

This preceding example:

- Doesn't buffer the entire request body in a `StringBuilder` unless there aren't any newline characters.
- Doesn't call `Split` on the string.

However, there are still a few issues:

- If newline characters are sparse, much of the request body is buffered in the string.
- The code continues to create strings (`remainingString`) and adds them to the string buffer, which results in an extra allocation.

These issues are fixable, but the code is becoming progressively more complicated with little improvement. Pipelines provide a way to solve these problems with minimal code complexity.

Pipelines

The following example shows how the same scenario can be handled using a PipeReader:

C#

```
private async Task<List<string>> GetListOfStringFromPipe(PipeReader reader)
{
    List<string> results = new List<string>();

    while (true)
    {
        ReadResult readResult = await reader.ReadAsync();
        var buffer = readResult.Buffer;

        SequencePosition? position = null;

        do
        {
            // Look for a EOL in the buffer
            position = buffer.PositionOf((byte)'\n');

            if (position != null)
            {
                var readOnlySequence = buffer.Slice(0, position.Value);
                AddStringToList(results, in readOnlySequence);

                // Skip the line + the \n character (basically position)
                buffer = buffer.Slice(buffer.GetPosition(1,
position.Value));
            }
        }
        while (position != null);


        if (readResult.IsCompleted && buffer.Length > 0)
        {
            AddStringToList(results, in buffer);
        }

        reader.AdvanceTo(buffer.Start, buffer.End);

        // At this point, buffer will be updated to point one byte after the
last
        // \n character.
        if (readResult.IsCompleted)
        {
            break;
        }
    }

    return results;
}

private static void AddStringToList(List<string> results, in
ReadOnlySequence<byte> readOnlySequence)
{
    // Separate method because Span/ReadOnlySpan cannot be used in async
methods
    ReadOnlySpan<byte> span = readOnlySequence.IsSingleSegment ?
readOnlySequence.First.Span : readOnlySequence.ToArray().AsSpan();
    results.Add(Encoding.UTF8.GetString(span));
}
```

This example fixes many issues that the streams implementations had:

- There's no need for a string buffer because the `PipeReader` handles bytes that haven't been used.
- Encoded strings are directly added to the list of returned strings.
- Other than the `ToArray` call, and the memory used by the string, string creation is allocation free.

Adapters

The `Body`, `BodyReader`, and `BodyWriter` properties are available for `HttpRequest` and `HttpResponse`. When you set `Body` to a different stream, a new set of adapters automatically adapt each type to the other. If you set `HttpRequest.Body` to a new stream, `HttpRequest.BodyReader` is automatically set to a new `PipeReader` that wraps `HttpRequest.Body`.

StartAsync

`HttpResponse.StartAsync` is used to indicate that headers are unmodifiable and to run `OnStarting` callbacks. When using Kestrel as a server, calling `StartAsync` before using the `PipeReader` guarantees that memory returned by `GetMemory` belongs to Kestrel's internal Pipe rather than an external buffer.

Additional resources

- System.IO.Pipelines in .NET
- Write custom ASP.NET Core middleware

Request decompression in ASP.NET Core

Article • 09/27/2024

Important

This information relates to a pre-release product that may be substantially modified before it's commercially released. Microsoft makes no warranties, express or implied, with respect to the information provided here.

For the current release, see the .NET 9 version of this article.

By David Acker

Request decompression middleware:

- Enables API endpoints to accept requests with compressed content.
- Uses the Content-Encoding HTTP header to automatically identify and decompress requests which contain compressed content.
- Eliminates the need to write code to handle compressed requests.

When the `Content-Encoding` header value on a request matches one of the available decompression providers, the middleware:

- Uses the matching provider to wrap the HttpRequest.Body in an appropriate decompression stream.
- Removes the `Content-Encoding` header, indicating that the request body is no longer compressed.

Requests that don't include a `Content-Encoding` header are ignored by the request decompression middleware.

Decompression:

- Occurs when the body of the request is read. That is, decompression occurs at the endpoint on model binding. The request body isn't decompressed eagerly.
- When attempting to read the decompressed request body with invalid compressed data for the specified `Content-Encoding`, an exception is thrown. Brotli can throw System.InvalidOperationException: Decoder ran into invalid data. Deflate and GZip can throw System.IO.InvalidDataException: The archive entry was compressed using an unsupported compression method.

If the middleware encounters a request with compressed content but is unable to decompress it, the request is passed to the next delegate in the pipeline. For example, a request with an unsupported `Content-Encoding` header value or multiple `Content-Encoding` header values is passed to the next delegate in the pipeline.

Configuration

The following code uses AddRequestDecompression(IServiceCollection) and UseRequestDecompression to enable request decompression for the default `Content-Encoding` types:

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

builder.Services.AddRequestDecompression();

var app = builder.Build();

app.UseRequestDecompression();

app.MapPost("/", (HttpRequest request) => Results.Stream(request.Body));

app.Run();
```

Default decompression providers

The `Content-Encoding` header values that the request decompression middleware supports by default are listed in the following table:

Expand table

Content-Encoding header values	Description
`br`	Brotli compressed data format
`deflate`	DEFLATE compressed data format
`gzip`	Gzip file format

Custom decompression providers

Support for custom encodings can be added by creating custom decompression provider classes that implement IDecompressionProvider:

```C#
public class CustomDecompressionProvider : IDecompressionProvider
{
    public Stream GetDecompressionStream(Stream stream)
    {
        // Perform custom decompression logic here
        return stream;
    }
}
```

Custom decompression providers are registered with RequestDecompressionOptions along with their corresponding `Content-Encoding` header values:

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

builder.Services.AddRequestDecompression(options =>
{
    options.DecompressionProviders.Add("custom", new
CustomDecompressionProvider());
});

var app = builder.Build();

app.UseRequestDecompression();

app.MapPost("/", (HttpRequest request) => Results.Stream(request.Body));

app.Run();
```

Request size limits

In order to protect against zip bombs or decompression bombs:

- The maximum size of the decompressed request body is limited to the request body size limit enforced by the endpoint or server.
- If the number of bytes read from the decompressed request body stream exceeds the limit, an InvalidOperationException is thrown to prevent additional bytes from being read from the stream.

In order of precedence, the maximum request size for an endpoint is set by:

1. IRequestSizeLimitMetadata.MaxRequestBodySize, such as RequestSizeLimitAttribute or DisableRequestSizeLimitAttribute for MVC endpoints.
2. The global server size limit IHttpMaxRequestBodySizeFeature.MaxRequestBodySize. `MaxRequestBodySize` can be overridden per request with IHttpMaxRequestBodySizeFeature.MaxRequestBodySize, but defaults to the limit configured for the web server implementation.

Expand table

Web server implementation	`MaxRequestBodySize` configuration
HTTP.sys	HttpSysOptions.MaxRequestBodySize
IIS	IISServerOptions.MaxRequestBodySize
Kestrel	KestrelServerLimits.MaxRequestBodySize

⚠ **Warning**

Disabling the request body size limit poses a security risk in regards to uncontrolled resource consumption, particularly if the request body is being buffered. Ensure that safeguards are in place to mitigate the risk of denial-of-service (DoS) attacks.

Additional Resources

- ASP.NET Core Middleware
- Mozilla Developer Network: Content-Encoding
- Brotli Compressed Data Format
- DEFLATE Compressed Data Format Specification version 1.3
- GZIP file format specification version 4.3

URL Rewriting Middleware in ASP.NET Core

Article • 07/26/2024

ⓘ **Important**

This information relates to a pre-release product that may be substantially modified before it's commercially released. Microsoft makes no warranties, express or implied, with respect to the information provided here.

For the current release, see the .NET 9 version of this article.

By Kirk Larkin and Rick Anderson

This article introduces URL rewriting with instructions on how to use URL Rewriting Middleware in ASP.NET Core apps.

URL rewriting is the act of modifying request URLs based on one or more predefined rules. URL rewriting creates an abstraction between resource locations and their addresses so that the locations and addresses aren't tightly linked. URL rewriting is valuable in several scenarios to:

- Move or replace server resources temporarily or permanently and maintain stable locators for those resources.
- Split request processing across different apps or across areas of one app.
- Remove, add, or reorganize URL segments on incoming requests.
- Optimize public URLs for Search Engine Optimization (SEO).
- Permit the use of friendly public URLs to help visitors predict the content returned by requesting a resource.
- Redirect insecure requests to secure endpoints.
- Prevent hotlinking, where an external site uses a hosted static asset on another site by linking the asset into its own content.

URL rewriting can reduce the performance of an app. Limit the number and complexity of rules.

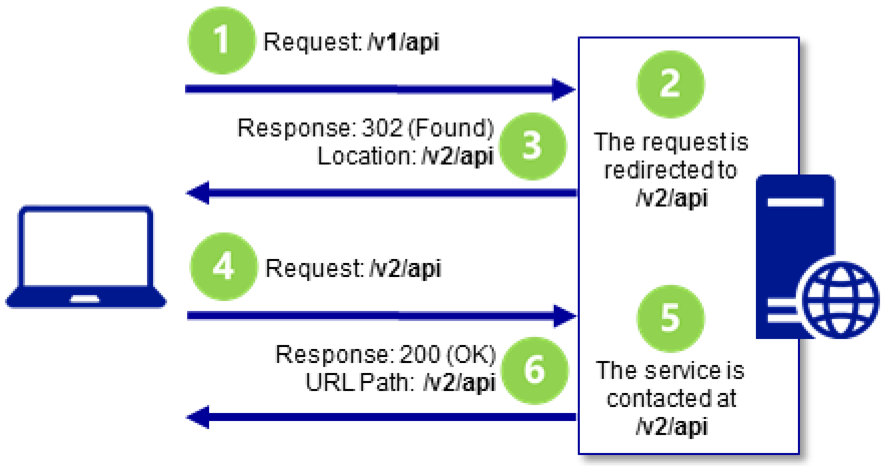
URL redirect and URL rewrite

The difference in wording between *URL redirect* and *URL rewrite* is subtle but has important implications for providing resources to clients. ASP.NET Core's URL Rewriting

Middleware is capable of meeting the need for both.

A *URL redirect* involves a client-side operation, where the client is instructed to access a resource at a different address than the client originally requested. This requires a round trip to the server. The redirect URL returned to the client appears in the browser's address bar when the client makes a new request for the resource.

If `/resource` is *redirected* to `/different-resource`, the server responds that the client should obtain the resource at `/different-resource` with a status code indicating that the redirect is either temporary or permanent.



When redirecting requests to a different URL, indicate whether the redirect is permanent or temporary by specifying the status code with the response:

- The 301 - Moved Permanently status code is used where the resource has a new, permanent URL and that all future requests for the resource should use the new URL. *The client may cache and reuse the response when a 301 status code is received.*

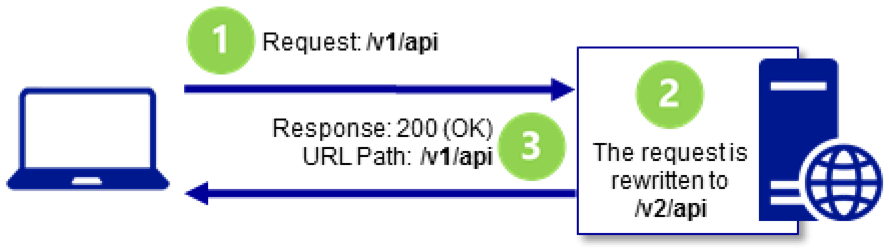
- The 302 - Found status code is used where the redirection is temporary or generally subject to change. The 302 status code indicates to the client not to store the URL and use it in the future.

For more information on status codes, see RFC 9110: Status Code Definitions.

A *URL rewrite* is a server-side operation that provides a resource from a different resource address than the client requested. Rewriting a URL doesn't require a round trip to the server. The rewritten URL isn't returned to the client and doesn't appear in the browser's address bar.

If `/resource` is *rewritten* to `/different-resource`, the server *internally* fetches and returns the resource at `/different-resource`.

Although the client might be able to retrieve the resource at the rewritten URL, the client isn't informed that the resource exists at the rewritten URL when it makes its request and receives the response.



URL rewriting sample app

Explore the features of the URL Rewriting Middleware with the sample app. The app applies redirect and rewrite rules and shows the redirected or rewritten URL for several scenarios.

When to use URL rewriting middleware

Use URL Rewriting Middleware when the following approaches aren't satisfactory:

- URL Rewrite module with IIS on Windows Server
- Apache mod_rewrite module on Apache Server
- URL rewriting on Nginx

Use the URL rewriting middleware when the app is hosted on HTTP.sys server.

The main reasons to use the server-based URL rewriting technologies in IIS, Apache, and Nginx are:

- The middleware doesn't support the full features of these modules.

  Some of the features of the server modules don't work with ASP.NET Core projects, such as the `IsFile` and `IsDirectory` constraints of the IIS Rewrite module. In these scenarios, use the middleware instead.

- The performance of the middleware probably doesn't match that of the modules.

  Benchmarking is the only way to know with certainty which approach degrades performance the most or if degraded performance is negligible.

Extension and options

Establish URL rewrite and redirect rules by creating an instance of the RewriteOptions class with extension methods for each of the rewrite rules. Chain multiple rules ***in the order that they should be processed***. The `RewriteOptions` are passed into the URL Rewriting Middleware as it's added to the request pipeline with UseRewriter:

C#

```
using Microsoft.AspNetCore.Rewrite;
using RewriteRules;

var builder = WebApplication.CreateBuilder(args);
var app = builder.Build();

using (StreamReader apacheModRewriteStreamReader =
    File.OpenText("ApacheModRewrite.txt"))
using (StreamReader iisUrlRewriteStreamReader =
    File.OpenText("IISUrlRewrite.xml"))
{
    var options = new RewriteOptions()
        .AddRedirect("redirect-rule/(.*)", "redirected/$1")
        .AddRewrite(@"^rewrite-rule/(\d+)/(\d+)", "rewritten?
var1=$1&var2=$2",
            skipRemainingRules: true)
        .AddApacheModRewrite(apacheModRewriteStreamReader)
        .AddIISUrlRewrite(iisUrlRewriteStreamReader)
        .Add(MethodRules.RedirectXmlFileRequests)
        .Add(MethodRules.RewriteTextFileRequests)
        .Add(new RedirectImageRequests(".png", "/png-images"))
        .Add(new RedirectImageRequests(".jpg", "/jpg-images"));

    app.UseRewriter(options);
}

app.UseStaticFiles();

app.Run(context => context.Response.WriteAsync(
    $"Rewritten or Redirected Url: " +
    $"{context.Request.Path + context.Request.QueryString}"));

app.Run();
```

In the preceding code, MethodRules is a user defined class. See RewriteRules.cs in this article for more information.

Redirect non-www to www

Three options permit the app to redirect non-`www` requests to `www`:

- AddRedirectToWwwPermanent: Permanently redirect the request to the `www` subdomain if the request is non-`www`. Redirects with a Status308PermanentRedirect status code.
- AddRedirectToWww: Redirect the request to the `www` subdomain if the incoming request is non-`www`. Redirects with a Status307TemporaryRedirect status code. An overload permits providing the status code for the response. Use a field of the StatusCodes class for a status code assignment.

URL redirect

Use AddRedirect to redirect requests. The first parameter contains the .NET regular expression (Regex) for matching on the path of the incoming URL. The second parameter is the replacement string. The third parameter, if present, specifies the status code. If the status code isn't specified, the status code defaults to 302 - Found, which indicates that the resource is temporarily moved or replaced.

```C#
using Microsoft.AspNetCore.Rewrite;
using RewriteRules;

var builder = WebApplication.CreateBuilder(args);
var app = builder.Build();

using (StreamReader apacheModRewriteStreamReader =
    File.OpenText("ApacheModRewrite.txt"))
using (StreamReader iisUrlRewriteStreamReader =
    File.OpenText("IISUrlRewrite.xml"))
{
    var options = new RewriteOptions()
        .AddRedirect("redirect-rule/(.*)", "redirected/$1")
        .AddRewrite(@"^rewrite-rule/(\d+)/(\d+)", "rewritten?
var1=$1&var2=$2",
            skipRemainingRules: true)
        .AddApacheModRewrite(apacheModRewriteStreamReader)
        .AddIISUrlRewrite(iisUrlRewriteStreamReader)
        .Add(MethodRules.RedirectXmlFileRequests)
        .Add(MethodRules.RewriteTextFileRequests)
        .Add(new RedirectImageRequests(".png", "/png-images"))
        .Add(new RedirectImageRequests(".jpg", "/jpg-images"));

    app.UseRewriter(options);
}

app.UseStaticFiles();

app.Run(context => context.Response.WriteAsync(
    $"Rewritten or Redirected Url: " +
    $"{context.Request.Path + context.Request.QueryString}"));

app.Run();
```

In a browser with developer tools enabled, make a request to the sample app with the path `/redirect-rule/1234/5678`. The regular expression matches the request path on `redirect-rule/(.*)`, and the path is replaced with `/redirected/1234/5678`. The redirect URL is sent back to the client with a *302 - Found* status code. The browser makes a new request at the redirect URL, which appears in the browser's address bar. Since no rules in the sample app match on the redirect URL:

- The second request receives a *200 - OK* response from the app.
- The body of the response shows the redirect URL.

A round trip is made to the server when a URL is *redirected*.

> ⚠ **Warning**
>
> Be cautious when establishing redirect rules. Redirect rules are evaluated on every request to the app, including after a redirect. It's easy to accidentally create a loop of ***infinite*** redirects.

The part of the expression contained within parentheses is called a capture group. The dot (`.`) of the expression means *match any character*. The asterisk (`*`) indicates *match the preceding character zero or more times*. Therefore, the last two path segments of the URL, `1234/5678`, are captured by capture group `(.*)`. Any value provided in the request URL after `redirect-rule/` is captured by this single capture group.

In the replacement string, captured groups are injected into the string with the dollar sign (`$`) followed by the sequence number of the capture. The first capture group value is obtained with `$1`, the second with `$2`, and they continue in sequence for the capture groups in the regular expression. There's only one captured group in the redirect rule regular expression in `redirect-rule/(.*)`, so there's only one injected group in the replacement string, which is `$1`. When the rule is applied, the URL becomes `/redirected/1234/5678`.

Try `/redirect-rule/1234/5678` with the browser tools on the network tab.

URL redirect to a secure endpoint

Use AddRedirectToHttps to redirect HTTP requests to the same host and path using the HTTPS protocol. If the status code isn't supplied, the middleware defaults to *302 - Found*. If the port isn't supplied:

- The middleware defaults to `null`.
- The scheme changes to `https` (HTTPS protocol), and the client accesses the resource on port 443.

The following example shows how to set the status code to `301 - Moved Permanently` and change the port to the HTTPS port used by Kestrel on localhost. In production, the HTTPS port is set to null:

```C#
using Microsoft.AspNetCore.Rewrite;
using RewriteRules;

var builder = WebApplication.CreateBuilder(args);
var app = builder.Build();

int? localhostHTTPSport = null;
if (app.Environment.IsDevelopment())
{
    localhostHTTPSport = Int32.Parse(Environment.GetEnvironmentVariable(
                   "ASPNETCORE_URLS")!.Split(new Char[] { ':', ';' })[2]);
}

using (StreamReader apacheModRewriteStreamReader =
    File.OpenText("ApacheModRewrite.txt"))
using (StreamReader iisUrlRewriteStreamReader =
    File.OpenText("IISUrlRewrite.xml"))
{
    var options = new RewriteOptions()
        // localhostHTTPport not needed for production, used only with
localhost.
        .AddRedirectToHttps(301, localhostHTTPSport)
        .AddRedirect("redirect-rule/(.*)", "redirected/$1")
        .AddRewrite(@"^rewrite-rule/(\d+)/(\d+)", "rewritten?
var1=$1&var2=$2",
            skipRemainingRules: true)
        .AddApacheModRewrite(apacheModRewriteStreamReader)
        .AddIISUrlRewrite(iisUrlRewriteStreamReader)
        .Add(MethodRules.RedirectXmlFileRequests)
        .Add(MethodRules.RewriteTextFileRequests)
        .Add(new RedirectImageRequests(".png", "/png-images"))
        .Add(new RedirectImageRequests(".jpg", "/jpg-images"));

    app.UseRewriter(options);
}

app.UseStaticFiles();
app.Run(context => context.Response.WriteAsync(
    $"Rewritten or Redirected Url: " +
    $"{context.Request.Path + context.Request.QueryString}"));

app.Run();
```

Use AddRedirectToHttpsPermanent to redirect insecure requests to the same host and path with secure HTTPS protocol on port 443. The middleware sets the status code to `301 - Moved Permanently`.

C#

```
using Microsoft.AspNetCore.Rewrite;
using RewriteRules;

var builder = WebApplication.CreateBuilder(args);
var app = builder.Build();

using (StreamReader apacheModRewriteStreamReader =
    File.OpenText("ApacheModRewrite.txt"))
using (StreamReader iisUrlRewriteStreamReader =
    File.OpenText("IISUrlRewrite.xml"))
{
    var options = new RewriteOptions()
        .AddRedirectToHttpsPermanent()
        .AddRedirect("redirect-rule/(.*)", "redirected/$1")
        .AddRewrite(@"^rewrite-rule/(\d+)/(\d+)", "rewritten?
var1=$1&var2=$2",
            skipRemainingRules: true)
        .AddApacheModRewrite(apacheModRewriteStreamReader)
        .AddIISUrlRewrite(iisUrlRewriteStreamReader)
        .Add(MethodRules.RedirectXmlFileRequests)
        .Add(MethodRules.RewriteTextFileRequests)
        .Add(new RedirectImageRequests(".png", "/png-images"))
        .Add(new RedirectImageRequests(".jpg", "/jpg-images"));

    app.UseRewriter(options);
}

app.UseStaticFiles();

app.Run(context => context.Response.WriteAsync(
    $"Rewritten or Redirected Url: " +
    $"{context.Request.Path + context.Request.QueryString}"));

app.Run();
```

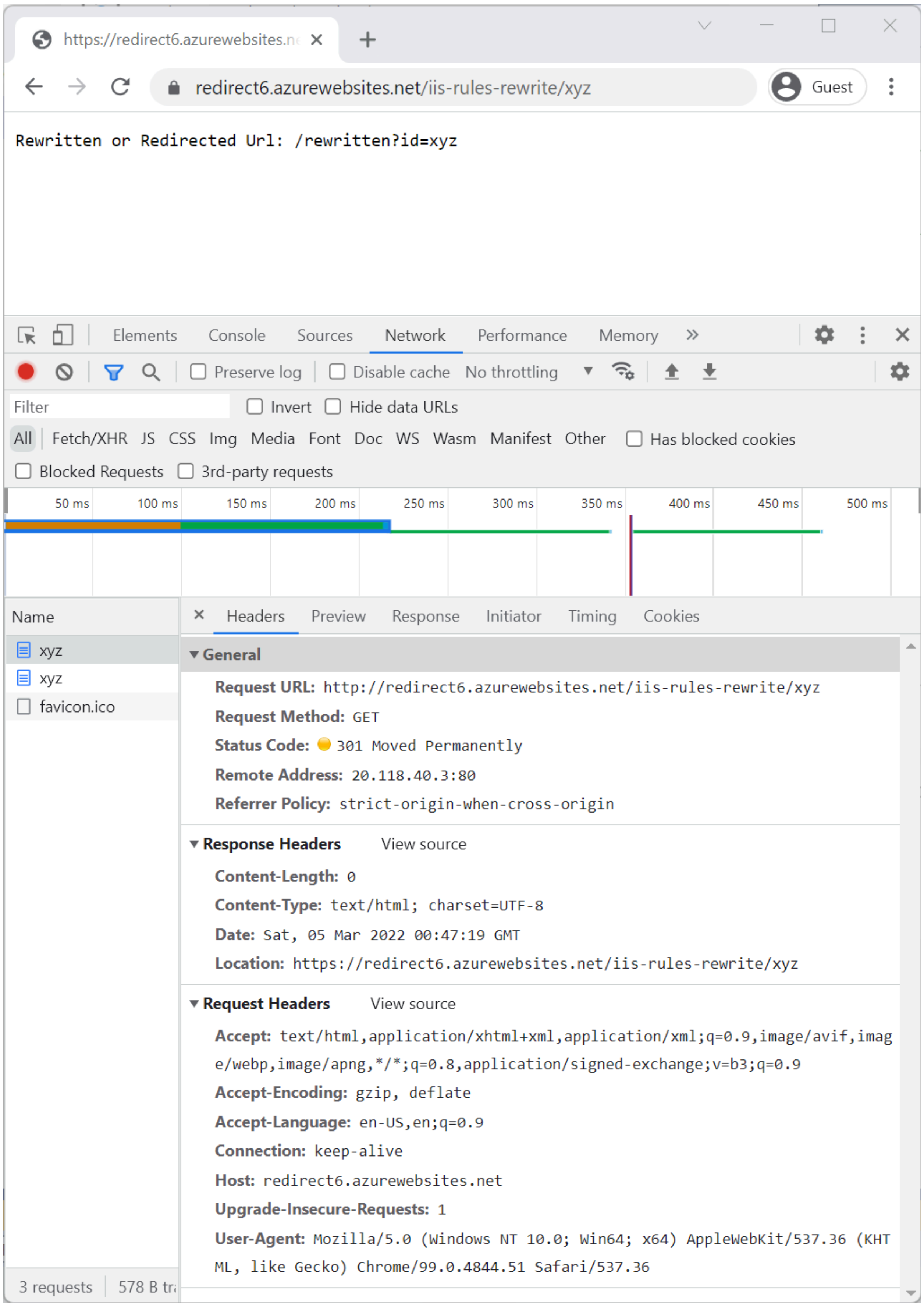
ⓘ **Note**

When redirecting to a secure endpoint without the requirement for additional redirect rules, we recommend using HTTPS Redirection Middleware. For more information, see **Enforce HTTPS**.

The sample app demonstrates how to use `AddRedirectToHttps` or `AddRedirectToHttpsPermanent`. Make an insecure HTTP request to the app at `http://redirect6.azurewebsites.net/iis-rules-rewrite/xyz`. When testing HTTP to HTTPS redirection with localhost:

- Use the HTTP URL, which has a different port than the HTTPS URL. The HTTP URL is in the `Properties/launchSettings.json` file.
- Removing the `s` from `https://localhost/{port}` fails because localhost doesn't respond on HTTP to the HTTPS port.

The following image shows the F12 browser tools image of a request to `http://redirect6.azurewebsites.net/iis-rules-rewrite/xyz` using the preceding code:



```
Rewritten or Redirected Url: /rewritten?id=xyz
```

General

Request URL: `http://redirect6.azurewebsites.net/iis-rules-rewrite/xyz`
Request Method: `GET`
Status Code: `301 Moved Permanently`
Remote Address: `20.118.40.3:80`
Referrer Policy: `strict-origin-when-cross-origin`

Response Headers View source

Content-Length: `0`
Content-Type: `text/html; charset=UTF-8`
Date: `Sat, 05 Mar 2022 00:47:19 GMT`
Location: `https://redirect6.azurewebsites.net/iis-rules-rewrite/xyz`

Request Headers View source

Accept: `text/html,application/xhtml+xml,application/xml;q=0.9,image/avif,image/webp,image/apng,*/*;q=0.8,application/signed-exchange;v=b3;q=0.9`
Accept-Encoding: `gzip, deflate`
Accept-Language: `en-US,en;q=0.9`
Connection: `keep-alive`
Host: `redirect6.azurewebsites.net`
Upgrade-Insecure-Requests: `1`
User-Agent: `Mozilla/5.0 (Windows NT 10.0; Win64; x64) AppleWebKit/537.36 (KHTML, like Gecko) Chrome/99.0.4844.51 Safari/537.36`

URL rewrite

Use AddRewrite to create a rule for rewriting URLs. The first parameter contains the regular expression for matching on the incoming URL path. The second parameter is the replacement string. The third parameter, `skipRemainingRules: {true|false}`, indicates to the middleware whether or not to skip additional rewrite rules if the current rule is applied.

```C#
using Microsoft.AspNetCore.Rewrite;
using RewriteRules;

var builder = WebApplication.CreateBuilder(args);
var app = builder.Build();

using (StreamReader apacheModRewriteStreamReader =
    File.OpenText("ApacheModRewrite.txt"))
using (StreamReader iisUrlRewriteStreamReader =
    File.OpenText("IISUrlRewrite.xml"))
{
    var options = new RewriteOptions()
        .AddRedirectToHttpsPermanent()
        .AddRedirect("redirect-rule/(.*)", "redirected/$1")
        .AddRewrite(@"^rewrite-rule/(\d+)/(\d+)", "rewritten?
var1=$1&var2=$2",
            skipRemainingRules: true)
        .AddApacheModRewrite(apacheModRewriteStreamReader)
        .AddIISUrlRewrite(iisUrlRewriteStreamReader)
        .Add(MethodRules.RedirectXmlFileRequests)
        .Add(MethodRules.RewriteTextFileRequests)
        .Add(new RedirectImageRequests(".png", "/png-images"))
        .Add(new RedirectImageRequests(".jpg", "/jpg-images"));

    app.UseRewriter(options);
}

app.UseStaticFiles();

app.Run(context => context.Response.WriteAsync(
    $"Rewritten or Redirected Url: " +
    $"{context.Request.Path + context.Request.QueryString}"));

app.Run();
```

Try the request to `https://redirect6.azurewebsites.net/rewrite-rule/1234/5678`

The caret (`^`) at the beginning of the expression means that matching starts at the beginning of the URL path.

In the earlier example with the redirect rule, `redirect-rule/(.*)`, there's no caret (`^`) at the start of the regular expression. Therefore, any characters may precede `redirect-rule/` in the path for a successful match.

Expand table

Path	Match
`/redirect-rule/1234/5678`	Yes
`/my-cool-redirect-rule/1234/5678`	Yes
`/anotherredirect-rule/1234/5678`	Yes

The rewrite rule, `^rewrite-rule/(\d+)/(\d+)`, only matches paths if they start with `rewrite-rule/`. In the following table, note the difference in matching.

Expand table

Path	Match
`/rewrite-rule/1234/5678`	Yes
`/my-cool-rewrite-rule/1234/5678`	No
`/anotherrewrite-rule/1234/5678`	No

Following the `^rewrite-rule/` portion of the expression, there are two capture groups, `(\d+)/(\d+)`. The `\d` signifies *match a digit (number)*. The plus sign (`+`) means *match one or more of the preceding character*. Therefore, the URL must contain a number followed by a forward-slash followed by another number. These capture groups are injected into the rewritten URL as `$1` and `$2`. The rewrite rule replacement string places the captured groups into the query string. The requested path `/rewrite-rule/1234/5678` is rewritten to return the resource at `/rewritten?var1=1234&var2=5678`. If a query string is present on the original request, it's preserved when the URL is rewritten.

There's no round trip to the server to return the resource. If the resource exists, it's fetched and returned to the client with a *200 - OK* status code. Because the client isn't redirected, the URL in the browser's address bar doesn't change. Clients can't detect that a URL rewrite operation occurred on the server.

Performance tips for URL rewrite and redirect

For the fastest response:

- Order rewrite rules from the most frequently matched rule to the least frequently matched rule.

- Use `skipRemainingRules: true` whenever possible because matching rules is computationally expensive and increases app response time. Skip the processing of the remaining rules when a match occurs and no additional rule processing is required.

> ⚠ **Warning**
>
> A malicious user can provide expensive to process input to `RegularExpressions` causing a Denial-of-Service attack. ASP.NET Core framework APIs that use `RegularExpressions` pass a timeout. For example, the RedirectRule and RewriteRule classes both pass in a one second timeout.

Apache mod_rewrite

Apply Apache mod_rewrite rules with AddApacheModRewrite. Make sure that the rules file is deployed with the app. For more information and examples of mod_rewrite rules, see Apache mod_rewrite.

A StreamReader is used to read the rules from the *ApacheModRewrite.txt* rules file:

C#

```csharp
using Microsoft.AspNetCore.Rewrite;
using RewriteRules;

var builder = WebApplication.CreateBuilder(args);
var app = builder.Build();

using (StreamReader apacheModRewriteStreamReader =
    File.OpenText("ApacheModRewrite.txt"))
using (StreamReader iisUrlRewriteStreamReader =
    File.OpenText("IISUrlRewrite.xml"))
{
    var options = new RewriteOptions()
        .AddRedirectToHttpsPermanent()
        .AddRedirect("redirect-rule/(.*)", "redirected/$1")
        .AddRewrite(@"^rewrite-rule/(\d+)/(\d+)", "rewritten?
var1=$1&var2=$2",
            skipRemainingRules: true)
        .AddApacheModRewrite(apacheModRewriteStreamReader)
        .AddIISUrlRewrite(iisUrlRewriteStreamReader)
        .Add(MethodRules.RedirectXmlFileRequests)
        .Add(MethodRules.RewriteTextFileRequests)
        .Add(new RedirectImageRequests(".png", "/png-images"))
        .Add(new RedirectImageRequests(".jpg", "/jpg-images"));

    app.UseRewriter(options);
}

app.UseStaticFiles();

app.Run(context => context.Response.WriteAsync(
    $"Rewritten or Redirected Url: " +
    $"{context.Request.Path + context.Request.QueryString}"));

app.Run();
```

The sample app redirects requests from `/apache-mod-rules-redirect/(.\*)` to `/redirected?id=$1`. The response status code is *302 - Found*.

```
# Rewrite path with additional sub directory
RewriteRule ^/apache-mod-rules-redirect/(.*) /redirected?id=$1 [L,R=302]
```

Try the request to `https://redirect6.azurewebsites.net/apache-mod-rules-redirect/1234`

The Apache middleware supports the following Apache mod_rewrite server variables:

- CONN_REMOTE_ADDR
- HTTP_ACCEPT
- HTTP_CONNECTION
- HTTP_COOKIE
- HTTP_FORWARDED
- HTTP_HOST
- HTTP_REFERER
- HTTP_USER_AGENT
- HTTPS
- IPV6
- QUERY_STRING
- REMOTE_ADDR
- REMOTE_PORT
- REQUEST_FILENAME
- REQUEST_METHOD
- REQUEST_SCHEME
- REQUEST_URI
- SCRIPT_FILENAME
- SERVER_ADDR
- SERVER_PORT
- SERVER_PROTOCOL

- TIME
- TIME_DAY
- TIME_HOUR
- TIME_MIN
- TIME_MON
- TIME_SEC
- TIME_WDAY
- TIME_YEAR

IIS URL Rewrite Module rules

To use the same rule set that applies to the IIS URL Rewrite Module, use AddIISUrlRewrite. Make sure that the rules file is deployed with the app. Don't direct the middleware to use the app's *web.config* file when running on Windows Server IIS. With IIS, these rules should be stored outside of the app's *web.config* file in order to avoid conflicts with the IIS Rewrite module. For more information and examples of IIS URL Rewrite Module rules, see Using Url Rewrite Module 2.0 and URL Rewrite Module Configuration Reference.

A StreamReader is used to read the rules from the `IISUrlRewrite.xml` rules file:

```csharp
using Microsoft.AspNetCore.Rewrite;
using RewriteRules;

var builder = WebApplication.CreateBuilder(args);
var app = builder.Build();

using (StreamReader apacheModRewriteStreamReader =
    File.OpenText("ApacheModRewrite.txt"))
using (StreamReader iisUrlRewriteStreamReader =
    File.OpenText("IISUrlRewrite.xml"))
{
    var options = new RewriteOptions()
        .AddRedirectToHttpsPermanent()
        .AddRedirect("redirect-rule/(.*)", "redirected/$1")
        .AddRewrite(@"^rewrite-rule/(\d+)/(\d+)", "rewritten?
var1=$1&var2=$2",
            skipRemainingRules: true)
        .AddApacheModRewrite(apacheModRewriteStreamReader)
        .AddIISUrlRewrite(iisUrlRewriteStreamReader)
        .Add(MethodRules.RedirectXmlFileRequests)
        .Add(MethodRules.RewriteTextFileRequests)
        .Add(new RedirectImageRequests(".png", "/png-images"))
        .Add(new RedirectImageRequests(".jpg", "/jpg-images"));

    app.UseRewriter(options);
}

app.UseStaticFiles();

app.Run(context => context.Response.WriteAsync(
    $"Rewritten or Redirected Url: " +
    $"{context.Request.Path + context.Request.QueryString}"));

app.Run();
```

The sample app rewrites requests from `/iis-rules-rewrite/(.*)` to `/rewritten?id=$1`. The response is sent to the client with a *200 - OK* status code.

XML

```xml
<rewrite>
  <rules>
    <rule name="Rewrite segment to id querystring" stopProcessing="true">
      <match url="^iis-rules-rewrite/(.*)$" />
      <action type="Rewrite" url="rewritten?id={R:1}"
appendQueryString="false"/>
    </rule>
  </rules>
</rewrite>
```

Try the request to `https://redirect6.azurewebsites.net/iis-rules-rewrite/xyz`

Apps that have an active IIS Rewrite Module with server-level rules configured that impacts the app in undesirable ways:

- Consider disabling the IIS Rewrite Module for the app.
- For more information, see Disabling IIS modules.

Unsupported features

The middleware doesn't support the following IIS URL Rewrite Module features:

- Outbound Rules
- Custom Server Variables
- Wildcards
- LogRewrittenUrl

Supported server variables

The middleware supports the following IIS URL Rewrite Module server variables:

- CONTENT_LENGTH
- CONTENT_TYPE
- HTTP_ACCEPT
- HTTP_CONNECTION
- HTTP_COOKIE
- HTTP_HOST
- HTTP_REFERER
- HTTP_URL
- HTTP_USER_AGENT
- HTTPS
- LOCAL_ADDR
- QUERY_STRING
- REMOTE_ADDR
- REMOTE_PORT
- REQUEST_FILENAME
- REQUEST_URI

IFileProvider can be obtained via a PhysicalFileProvider. This approach may provide greater flexibility for the location of rewrite rules files. Make sure that the rewrite rules files are deployed to the server at the path provided.

```C#
var fileProvider = new
PhysicalFileProvider(Directory.GetCurrentDirectory());
```

Method-based rule

Use Add to implement custom rule logic in a method. `Add` exposes the RewriteContext, which makes available the HttpContext for use in redirect methods. The RewriteContext.Result property determines how additional pipeline processing is handled. Set the value to one of the RuleResult fields described in the following table.

Expand table

Rewrite context result	**Action**
`RuleResult.ContinueRules` (default)	Continue applying rules.
`RuleResult.EndResponse`	Stop applying rules and send the response.

Rewrite context result	Action
`RuleResult.SkipRemainingRules`	Stop applying rules and send the context to the next middleware.

```C#
using Microsoft.AspNetCore.Rewrite;
using RewriteRules;

var builder = WebApplication.CreateBuilder(args);
var app = builder.Build();

using (StreamReader apacheModRewriteStreamReader =
    File.OpenText("ApacheModRewrite.txt"))
using (StreamReader iisUrlRewriteStreamReader =
    File.OpenText("IISUrlRewrite.xml"))
{
    var options = new RewriteOptions()
        .AddRedirectToHttpsPermanent()
        .AddRedirect("redirect-rule/(.*)", "redirected/$1")
        .AddRewrite(@"^rewrite-rule/(\d+)/(\d+)", "rewritten?
var1=$1&var2=$2",
            skipRemainingRules: true)
        .AddApacheModRewrite(apacheModRewriteStreamReader)
        .AddIISUrlRewrite(iisUrlRewriteStreamReader)
        .Add(MethodRules.RedirectXmlFileRequests)
        .Add(MethodRules.RewriteTextFileRequests)
        .Add(new RedirectImageRequests(".png", "/png-images"))
        .Add(new RedirectImageRequests(".jpg", "/jpg-images"));

    app.UseRewriter(options);
}

app.UseStaticFiles();

app.Run(context => context.Response.WriteAsync(
    $"Rewritten or Redirected Url: " +
    $"{context.Request.Path + context.Request.QueryString}"));

app.Run();
```

The sample app demonstrates a method that redirects requests for paths that end with `.xml`. When a request is made for `/file.xml`:

- The request is redirected to `/xmlfiles/file.xml`
- The status code is set to `301 - Moved Permanently`. When the browser makes a new request for `/xmlfiles/file.xml`, Static File Middleware serves the file to the client from the *wwwroot/xmlfiles* folder. For a redirect, explicitly set the status code of

the response. Otherwise, a *200 - OK* status code is returned, and the redirect doesn't occur on the client.

`RewriteRules.cs`:

```C#
public static void RedirectXmlFileRequests(RewriteContext context)
{
    var request = context.HttpContext.Request;

    // Because the client is redirecting back to the same app, stop
    // processing if the request has already been redirected.
    if (request.Path.StartsWithSegments(new PathString("/xmlfiles")) ||
        request.Path.Value==null)
    {
        return;
    }

    if (request.Path.Value.EndsWith(".xml",
StringComparison.OrdinalIgnoreCase))
    {
        var response = context.HttpContext.Response;
        response.StatusCode = (int) HttpStatusCode.MovedPermanently;
        context.Result = RuleResult.EndResponse;
        response.Headers[HeaderNames.Location] =
            "/xmlfiles" + request.Path + request.QueryString;
    }
}
```

This approach can also rewrite requests. The sample app demonstrates rewriting the path for any text file request to serve the *file.txt* text file from the *wwwroot* folder. Static File Middleware serves the file based on the updated request path:

```C#
using Microsoft.AspNetCore.Rewrite;
using RewriteRules;

var builder = WebApplication.CreateBuilder(args);
var app = builder.Build();

using (StreamReader apacheModRewriteStreamReader =
    File.OpenText("ApacheModRewrite.txt"))
using (StreamReader iisUrlRewriteStreamReader =
    File.OpenText("IISUrlRewrite.xml"))
{
    var options = new RewriteOptions()
        .AddRedirectToHttpsPermanent()
        .AddRedirect("redirect-rule/(.*)", "redirected/$1")
        .AddRewrite(@"^rewrite-rule/(\d+)/(\d+)", "rewritten?
var1=$1&var2=$2",
            skipRemainingRules: true)
        .AddApacheModRewrite(apacheModRewriteStreamReader)
        .AddIISUrlRewrite(iisUrlRewriteStreamReader)
        .Add(MethodRules.RedirectXmlFileRequests)
        .Add(MethodRules.RewriteTextFileRequests)
        .Add(new RedirectImageRequests(".png", "/png-images"))
        .Add(new RedirectImageRequests(".jpg", "/jpg-images"));

    app.UseRewriter(options);
}

app.UseStaticFiles();

app.Run(context => context.Response.WriteAsync(
    $"Rewritten or Redirected Url: " +
    $"{context.Request.Path + context.Request.QueryString}"));

app.Run();
```

`RewriteRules.cs`:

```C#
public static void RewriteTextFileRequests(RewriteContext context)
{
    var request = context.HttpContext.Request;

    if (request.Path.Value != null &&
        request.Path.Value.EndsWith(".txt",
StringComparison.OrdinalIgnoreCase))
    {
        context.Result = RuleResult.SkipRemainingRules;
        request.Path = "/file.txt";
    }
}
```

IRule-based rule

Use Add to use rule logic in a class that implements the IRule interface. `IRule` provides greater flexibility over using the method-based rule approach. The implementation class may include a constructor that allows passing in parameters for the ApplyRule method.

```C#
using Microsoft.AspNetCore.Rewrite;
using RewriteRules;

var builder = WebApplication.CreateBuilder(args);
var app = builder.Build();

using (StreamReader apacheModRewriteStreamReader =
    File.OpenText("ApacheModRewrite.txt"))
using (StreamReader iisUrlRewriteStreamReader =
    File.OpenText("IISUrlRewrite.xml"))
{
    var options = new RewriteOptions()
        .AddRedirectToHttpsPermanent()
        .AddRedirect("redirect-rule/(.*)", "redirected/$1")
        .AddRewrite(@"^rewrite-rule/(\d+)/(\d+)", "rewritten?
var1=$1&var2=$2",
            skipRemainingRules: true)
        .AddApacheModRewrite(apacheModRewriteStreamReader)
        .AddIISUrlRewrite(iisUrlRewriteStreamReader)
        .Add(MethodRules.RedirectXmlFileRequests)
        .Add(MethodRules.RewriteTextFileRequests)
        .Add(new RedirectImageRequests(".png", "/png-images"))
        .Add(new RedirectImageRequests(".jpg", "/jpg-images"));

    app.UseRewriter(options);
}

app.UseStaticFiles();

app.Run(context => context.Response.WriteAsync(
    $"Rewritten or Redirected Url: " +
    $"{context.Request.Path + context.Request.QueryString}"));

app.Run();
```

The values of the parameters in the sample app for the `extension` and the `newPath` are checked to meet several conditions. The `extension` must contain a value, and the value must be `.png`, `.jpg`, or `.gif`. If the `newPath` isn't valid, an ArgumentException is thrown. If a request is made for `image.png`, the request is redirected to `/png-images/image.png`. If a request is made for `image.jpg`, the request is redirected to `/jpg-images/image.jpg`. The status code is set to `301 - Moved Permanently`, and the `context.Result` is set to stop processing rules and send the response.

```C#
public class RedirectImageRequests : IRule
{
    private readonly string _extension;
    private readonly PathString _newPath;

    public RedirectImageRequests(string extension, string newPath)
    {
        if (string.IsNullOrEmpty(extension))
        {
            throw new ArgumentException(nameof(extension));
        }

        if (!Regex.IsMatch(extension, @"^\.(png|jpg|gif)$"))
        {
            throw new ArgumentException("Invalid extension",
nameof(extension));
        }

        if (!Regex.IsMatch(newPath, @"(/[A-Za-z0-9]+)+?"))
        {
            throw new ArgumentException("Invalid path", nameof(newPath));
        }

        _extension = extension;
        _newPath = new PathString(newPath);
    }

    public void ApplyRule(RewriteContext context)
    {
        var request = context.HttpContext.Request;

        // Because we're redirecting back to the same app, stop
        // processing if the request has already been redirected
        if (request.Path.StartsWithSegments(new PathString(_newPath)) ||
            request.Path.Value == null)
        {
            return;
        }

        if (request.Path.Value.EndsWith(_extension,
StringComparison.OrdinalIgnoreCase))
        {
            var response = context.HttpContext.Response;
            response.StatusCode = (int) HttpStatusCode.MovedPermanently;
            context.Result = RuleResult.EndResponse;
            response.Headers[HeaderNames.Location] =
                _newPath + request.Path + request.QueryString;
        }
    }
}
```

Try:

- PNG request: `https://redirect6.azurewebsites.net/image.png`
- JPG request: `https://redirect6.azurewebsites.net/image.jpg`

Regex examples

Expand table

Goal	Regex String & Match Example	Replacement String & Output Example
Rewrite path into querystring	`^path/(.*)/(.*)` `/path/abc/123`	`path?var1=$1&var2=$2` `/path?var1=abc&var2=123`
Strip trailing slash	`^path2/(.*)/$` `/path2/xyz/`	`$1` `/path2/xyz`
Enforce trailing slash	`^path3/(.*[^/])$` `/path3/xyz`	`$1/` `/path3/xyz/`
Avoid rewriting specific requests	`^(.*)(?<!\.axd)$` or `^(?!.*\.axd$)(.*)$` Yes: `/path4/resource.htm` No: `/path4/resource.axd`	`rewritten/$1` `/rewritten/resource.htm` `/resource.axd`
Rearrange URL segments	`path5/(.*)/(.*)/(.*)` `path5/1/2/3`	`path5/$3/$2/$1` `path5/3/2/1`
Replace a URL segment	`^path6/(.*)/segment2/(.*)` `^path6/segment1/segment2/segment3`	`path6/$1/replaced/$2` `/path6/segment1/replaced/segment3`

The links in the preceding table use the following code deployed to Azure:

```csharp
using Microsoft.AspNetCore.Rewrite;
using RewriteRules;

var builder = WebApplication.CreateBuilder(args);
var app = builder.Build();

using (StreamReader apacheModRewriteStreamReader =
    File.OpenText("ApacheModRewrite.txt"))
using (StreamReader iisUrlRewriteStreamReader =
    File.OpenText("IISUrlRewrite.xml"))
{
    var options = new RewriteOptions()
        .AddRedirectToHttpsPermanent()
        .AddRedirect("redirect-rule/(.*)", "redirected/$1")
        .AddRewrite(@"^rewrite-rule/(\d+)/(\d+)", "rewritten?
var1=$1&var2=$2",
            skipRemainingRules: true)
        // Rewrite path to QS.
        .AddRewrite(@"^path/(.*)/(.*)", "path?var1=$1&var2=$2",
            skipRemainingRules: true)
        // Skip trailing slash.
        .AddRewrite(@"^path2/(.*)/$", "path2/$1",
            skipRemainingRules: true)
         // Enforce trailing slash.
         .AddRewrite(@"^path3/(.*[^/])$", "path3/$1/",
            skipRemainingRules: true)
         // Avoid rewriting specific requests.
         .AddRewrite(@"^path4/(.*)(?<!\.axd)$", "rewritten/$1",
            skipRemainingRules: true)
         // Rearrange URL segments
         .AddRewrite(@"^path5/(.*)/(.*)/(.*)", "path5/$3/$2/$1",
            skipRemainingRules: true)
          // Replace a URL segment
          .AddRewrite(@"^path6/(.*)/segment2/(.*)", "path6/$1/replaced/$2",
            skipRemainingRules: true)

        .AddApacheModRewrite(apacheModRewriteStreamReader)
        .AddIISUrlRewrite(iisUrlRewriteStreamReader)
        .Add(MethodRules.RedirectXmlFileRequests)
        .Add(MethodRules.RewriteTextFileRequests)
        .Add(new RedirectImageRequests(".png", "/png-images"))
        .Add(new RedirectImageRequests(".jpg", "/jpg-images"));

    app.UseRewriter(options);
}

app.UseStaticFiles();

app.Run(context => context.Response.WriteAsync(
    $"Rewritten or Redirected Url: " +
    $"{context.Request.Path + context.Request.QueryString}"));

app.Run();
```

In most of the preceding regular expression samples, the literal `path` is used to make unique testable rewrite rules for the deployed sample. Typically the regular expression wouldn't include `path`. For example, see these regular expression examples table.

Additional resources

- View or download sample code (how to download)
- RewriteMiddleware source on GitHub
- App startup in ASP.NET Core
- ASP.NET Core Middleware
- Regular expressions in .NET
- Regular expression language - quick reference

- Apache mod_rewrite
- Using Url Rewrite Module 2.0 (for IIS)
- URL Rewrite Module Configuration Reference
- Keep a simple URL structure
- 10 URL Rewriting Tips and Tricks
- To slash or not to slash

File Providers in ASP.NET Core

Article • 07/26/2024

Important

This information relates to a pre-release product that may be substantially modified before it's commercially released. Microsoft makes no warranties, express or implied, with respect to the information provided here.

For the current release, see the .NET 9 version of this article.

By Steve Smith

ASP.NET Core abstracts file system access through the use of File Providers. File Providers are used throughout the ASP.NET Core framework. For example:

- IWebHostEnvironment exposes the app's content root and web root as `IFileProvider` types.
- Static File Middleware uses File Providers to locate static files.
- Razor uses File Providers to locate pages and views.
- .NET Core tooling uses File Providers and glob patterns to specify which files should be published.

View or download sample code (how to download)

File Provider interfaces

The primary interface is IFileProvider. `IFileProvider` exposes methods to:

- Obtain file information (IFileInfo).
- Obtain directory information (IDirectoryContents).
- Set up change notifications (using an IChangeToken).

`IFileInfo` provides methods and properties for working with files:

- Exists
- IsDirectory
- Name
- Length (in bytes)
- LastModified date

You can read from the file using the IFileInfo.CreateReadStream method.

The `FileProviderSample` sample app demonstrates how to configure a File Provider in `Startup.ConfigureServices` for use throughout the app via dependency injection.

File Provider implementations

The following table lists implementations of `IFileProvider`.

Expand table

Implementation	Description
Composite File Provider	Used to provide combined access to files and directories from one or more other providers.
Manifest Embedded File Provider	Used to access files embedded in assemblies.
Physical File Provider	Used to access the system's physical files.

Physical File Provider

The PhysicalFileProvider provides access to the physical file system. `PhysicalFileProvider` uses the System.IO.File type (for the physical provider) and scopes all paths to a directory and its children. This scoping prevents access to the file system outside of the specified directory and its children. The most common scenario for creating and using a `PhysicalFileProvider` is to request an `IFileProvider` in a constructor through dependency injection.

When instantiating this provider directly, an absolute directory path is required and serves as the base path for all requests made using the provider. Glob patterns aren't supported in the directory path.

The following code shows how to use `PhysicalFileProvider` to obtain directory contents and file information:

C#

```csharp
var provider = new PhysicalFileProvider(applicationRoot);
var contents = provider.GetDirectoryContents(string.Empty);
var filePath = Path.Combine("wwwroot", "js", "site.js");
var fileInfo = provider.GetFileInfo(filePath);
```

Types in the preceding example:

- `provider` is an `IFileProvider`.
- `contents` is an `IDirectoryContents`.
- `fileInfo` is an `IFileInfo`.

The File Provider can be used to iterate through the directory specified by `applicationRoot` or call `GetFileInfo` to obtain a file's information. Glob patterns can't be passed to the `GetFileInfo` method. The File Provider has no access outside of the `applicationRoot` directory.

The `FileProviderSample` sample app creates the provider in the `Startup.ConfigureServices` method using IHostEnvironment.ContentRootFileProvider:

```C#
var physicalProvider = _env.ContentRootFileProvider;
```

Manifest Embedded File Provider

The ManifestEmbeddedFileProvider is used to access files embedded within assemblies. The `ManifestEmbeddedFileProvider` uses a manifest compiled into the assembly to reconstruct the original paths of the embedded files.

To generate a manifest of the embedded files:

1. Add the Microsoft.Extensions.FileProviders.Embedded NuGet package to your project.

2. Set the `<GenerateEmbeddedFilesManifest>` property to `true`. Specify the files to embed with <EmbeddedResource>:

```XML
<Project Sdk="Microsoft.NET.Sdk.Web">

  <PropertyGroup>
    <TargetFramework>netcoreapp3.1</TargetFramework>
    <GenerateEmbeddedFilesManifest>true</GenerateEmbeddedFilesManifest>
  </PropertyGroup>

  <ItemGroup>
    <PackageReference
Include="Microsoft.Extensions.FileProviders.Embedded" Version="3.1.0"
/>
```

```
  </ItemGroup>

  <ItemGroup>
    <EmbeddedResource Include="Resource.txt" />
  </ItemGroup>

</Project>
```

Use glob patterns to specify one or more files to embed into the assembly.

The `FileProviderSample` sample app creates an `ManifestEmbeddedFileProvider` and passes the currently executing assembly to its constructor.

`Startup.cs`:

```C#
var manifestEmbeddedProvider =
    new ManifestEmbeddedFileProvider(typeof(Program).Assembly);
```

Additional overloads allow you to:

- Specify a relative file path.
- Scope files to a last modified date.
- Name the embedded resource containing the embedded file manifest.

Expand table

Overload	Description
`ManifestEmbeddedFileProvider(Assembly, String)`	Accepts an optional `root` relative path parameter. Specify the `root` to scope calls to GetDirectoryContents to those resources under the provided path.
`ManifestEmbeddedFileProvider(Assembly, String, DateTimeOffset)`	Accepts an optional `root` relative path parameter and a `lastModified` date (DateTimeOffset) parameter. The `lastModified` date scopes the last modification date for the IFileInfo instances returned by the IFileProvider.
`ManifestEmbeddedFileProvider(Assembly, String, String, DateTimeOffset)`	Accepts an optional `root` relative path, `lastModified` date, and `manifestName` parameters. The `manifestName` represents the name of the embedded resource containing the manifest.

Composite File Provider

The CompositeFileProvider combines `IFileProvider` instances, exposing a single interface for working with files from multiple providers. When creating the `CompositeFileProvider`, pass one or more `IFileProvider` instances to its constructor.

In the `FileProviderSample` sample app, a `PhysicalFileProvider` and a `ManifestEmbeddedFileProvider` provide files to a `CompositeFileProvider` registered in the app's service container. The following code is found in the project's `Startup.ConfigureServices` method:

C#

```csharp
var physicalProvider = _env.ContentRootFileProvider;
var manifestEmbeddedProvider =
    new ManifestEmbeddedFileProvider(typeof(Program).Assembly);
var compositeProvider =
    new CompositeFileProvider(physicalProvider, manifestEmbeddedProvider);

services.AddSingleton<IFileProvider>(compositeProvider);
```

Watch for changes

The IFileProvider.Watch method provides a scenario to watch one or more files or directories for changes. The `Watch` method:

- Accepts a file path string, which can use glob patterns to specify multiple files.
- Returns an IChangeToken.

The resulting change token exposes:

- HasChanged: A property that can be inspected to determine if a change has occurred.
- RegisterChangeCallback: Called when changes are detected to the specified path string. Each change token only calls its associated callback in response to a single change. To enable constant monitoring, use a TaskCompletionSource<TResult> (shown below) or recreate `IChangeToken` instances in response to changes.

The `WatchConsole` sample app writes a message whenever a `.txt` file in the `TextFiles` directory is modified:

C#

```
private static readonly string _fileFilter = Path.Combine("TextFiles",
"*.txt");

public static void Main(string[] args)
{
    Console.WriteLine($"Monitoring for changes with filter '{_fileFilter}'
(Ctrl + C to quit)...");

    while (true)
    {
        MainAsync().GetAwaiter().GetResult();
    }
}

private static async Task MainAsync()
{
    var fileProvider = new
PhysicalFileProvider(Directory.GetCurrentDirectory());
    IChangeToken token = fileProvider.Watch(_fileFilter);
    var tcs = new TaskCompletionSource<object>();

    token.RegisterChangeCallback(state =>
        ((TaskCompletionSource<object>)state).TrySetResult(null), tcs);

    await tcs.Task.ConfigureAwait(false);

    Console.WriteLine("file changed");
}
```

Some file systems, such as Docker containers and network shares, may not reliably send change notifications. Set the `DOTNET_USE_POLLING_FILE_WATCHER` environment variable to `1` or `true` to poll the file system for changes every four seconds (not configurable).

Glob patterns

File system paths use wildcard patterns called *glob (or globbing) patterns*. Specify groups of files with these patterns. The two wildcard characters are `*` and `**`:

`*`

Matches anything at the current folder level, any filename, or any file extension. Matches are terminated by `/` and `.` characters in the file path.

`**`

Matches anything across multiple directory levels. Can be used to recursively match many files within a directory hierarchy.

The following table provides common examples of glob patterns.

Expand table

Pattern	Description
`directory/file.txt`	Matches a specific file in a specific directory.
`directory/*.txt`	Matches all files with `.txt` extension in a specific directory.
`directory/*/appsettings.json`	Matches all `appsettings.json` files in directories exactly one level below the `directory` folder.
`directory/**/*.txt`	Matches all files with a `.txt` extension found anywhere under the `directory` folder.

Request Features in ASP.NET Core

Article • 09/17/2024

By Steve Smith

The `HttpContext` API that applications and middleware use to process requests has an abstraction layer underneath it called *feature interfaces*. Each feature interface provides a granular subset of the functionality exposed by `HttpContext`. These interfaces can be added, modified, wrapped, replaced, or even removed by the server or middleware as the request is processed without having to re-implement the entire `HttpContext`. They can also be used to mock functionality when testing.

Feature collections

The Features property of `HttpContext` provides access to the collection of feature interfaces for the current request. Since the feature collection is mutable even within the context of a request, middleware can be used to modify the collection and add support for additional features. Some advanced features are only available by accessing the associated interface through the feature collection.

Feature interfaces

ASP.NET Core defines a number of common HTTP feature interfaces in Microsoft.AspNetCore.Http.Features, which are shared by various servers and middleware to identify the features that they support. Servers and middleware may also provide their own interfaces with additional functionality.

Most feature interfaces provide optional, light-up functionality, and their associated `HttpContext` APIs provide defaults if the feature isn't present. A few interfaces are indicated in the following content as required because they provide core request and response functionality and must be implemented in order to process the request.

The following feature interfaces are from Microsoft.AspNetCore.Http.Features:

IHttpRequestFeature: Defines the structure of an HTTP request, including the protocol, path, query string, headers, and body. This feature is required in order to process requests.

IHttpResponseFeature: Defines the structure of an HTTP response, including the status code, headers, and body of the response. This feature is required in order to process requests.

IHttpResponseBodyFeature: Defines different ways of writing out the response body, using either a `Stream`, a `PipeWriter`, or a file. This feature is required in order to process requests. This replaces `IHttpResponseFeature.Body` and `IHttpSendFileFeature`.

IHttpAuthenticationFeature: Holds the ClaimsPrincipal currently associated with the request.

IFormFeature: Used to parse and cache incoming HTTP and multipart form submissions.

IHttpBodyControlFeature: Used to control if synchronous IO operations are allowed for the request or response bodies.

`IHttpActivityFeature`: Used to add `Activity` information for diagnostic listeners.

IHttpConnectionFeature: Defines properties for the connection id and local and remote addresses and ports.

IHttpMaxRequestBodySizeFeature: Controls the maximum allowed request body size for the current request.

`IHttpRequestBodyDetectionFeature`: Indicates if the request can have a body.

IHttpRequestIdentifierFeature: Adds a property that can be implemented to uniquely identify requests.

IHttpRequestLifetimeFeature: Defines support for aborting connections or detecting if a request has been terminated prematurely, such as by a client disconnect.

IHttpRequestTrailersFeature: Provides access to the request trailer headers, if any.

IHttpResetFeature: Used to send reset messages for protocols that support them such as HTTP/2 or HTTP/3.

IHttpResponseTrailersFeature: Enables the application to provide response trailer headers if supported.

IHttpUpgradeFeature: Defines support for HTTP Upgrades, which allow the client to specify which additional protocols it would like to use if the server wishes to switch protocols.

IHttpWebSocketFeature: Defines an API for supporting WebSockets.

IHttpsCompressionFeature: Controls if response compression should be used over HTTPS connections.

IItemsFeature: Stores the Items collection for per request application state.

IQueryFeature: Parses and caches the query string.

IRequestBodyPipeFeature: Represents the request body as a PipeReader.

IRequestCookiesFeature: Parses and caches the request `Cookie` header values.

IResponseCookiesFeature: Controls how response cookies are applied to the `Set-Cookie` header.

IServerVariablesFeature: This feature provides access to request server variables such as those provided by IIS.

IServiceProvidersFeature: Provides access to an IServiceProvider with scoped request services.

ISessionFeature: Defines `ISessionFactory` and ISession abstractions for supporting user sessions. `ISessionFeature` is implemented by the SessionMiddleware (see Session in ASP.NET Core).

ITlsConnectionFeature: Defines an API for retrieving client certificates.

ITlsTokenBindingFeature: Defines methods for working with TLS token binding parameters.

ITrackingConsentFeature: Used to query, grant, and withdraw user consent regarding the storage of user information related to site activity and functionality.

Additional resources

- Web server implementations in ASP.NET Core
- ASP.NET Core Middleware

Access `HttpContext` in ASP.NET Core

Article • 07/26/2024

ⓘ **Important**

This information relates to a pre-release product that may be substantially modified before it's commercially released. Microsoft makes no warranties, express or implied, with respect to the information provided here.

For the current release, see the .NET 9 version of this article.

HttpContext encapsulates all information about an individual HTTP request and response. An `HttpContext` instance is initialized when an HTTP request is received. The `HttpContext` instance is accessible by middleware and app frameworks such as Web API controllers, Razor Pages, SignalR, gRPC, and more.

For information about using `HttpContext` with a HTTP request and response, see Use HttpContext in ASP.NET Core.

Access `HttpContext` from Razor Pages

The Razor Pages PageModel exposes the PageModel.HttpContext property:

C#

```csharp
public class IndexModel : PageModel
{
    public void OnGet()
    {
        var message = HttpContext.Request.PathBase;

        // ...
    }
}
```

The same property can be used in the corresponding Razor Page View:

CSHTML

```cshtml
@page
@model IndexModel

@{
    var message = HttpContext.Request.PathBase;

    // ...
}
```

Access `HttpContext` from a Razor view in MVC

Razor views in the MVC pattern expose the `HttpContext` via the RazorPage.Context property on the view. The following example retrieves the current username in an intranet app using Windows Authentication:

CSHTML

```cshtml
@{
    var username = Context.User.Identity.Name;

    // ...
}
```

Access `HttpContext` from a controller

Controllers expose the ControllerBase.HttpContext property:

C#

```csharp
public class HomeController : Controller
{
    public IActionResult About()
    {
        var pathBase = HttpContext.Request.PathBase;

        // ...

        return View();
    }
}
```

Access `HttpContext` from minimal APIs

To use `HttpContext` from minimal APIs, add a `HttpContext` parameter:

C#

```
app.MapGet("/", (HttpContext context) => context.Response.WriteAsync("Hello
World"));
```

Access `HttpContext` from middleware

To use `HttpContext` from custom middleware components, use the `HttpContext` parameter passed into the `Invoke` or `InvokeAsync` method:

```C#
public class MyCustomMiddleware
{
    // ...

    public async Task InvokeAsync(HttpContext context)
    {
        // ...
    }
}
```

Access `HttpContext` from SignalR

To use `HttpContext` from SignalR, call the GetHttpContext method on Hub.Context:

```C#
public class MyHub : Hub
{
    public async Task SendMessage()
    {
        var httpContext = Context.GetHttpContext();

        // ...
    }
}
```

Access `HttpContext` from gRPC methods

To use `HttpContext` from gRPC methods, see Resolve HttpContext in gRPC methods.

Access `HttpContext` from custom components

For other framework and custom components that require access to `HttpContext`, the recommended approach is to register a dependency using the built-in Dependency Injection (DI) container. The DI container supplies the `IHttpContextAccessor` to any classes that declare it as a dependency in their constructors:

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

builder.Services.AddControllersWithViews();
builder.Services.AddHttpContextAccessor();
builder.Services.AddTransient<IUserRepository, UserRepository>();
```

In the following example:

- `UserRepository` declares its dependency on `IHttpContextAccessor`.
- The dependency is supplied when DI resolves the dependency chain and creates an instance of `UserRepository`.

```csharp
public class UserRepository : IUserRepository
{
    private readonly IHttpContextAccessor _httpContextAccessor;

    public UserRepository(IHttpContextAccessor httpContextAccessor) =>
        _httpContextAccessor = httpContextAccessor;

    public void LogCurrentUser()
    {
        var username = _httpContextAccessor.HttpContext.User.Identity.Name;

        // ...
    }
}
```

`HttpContext` access from a background thread

`HttpContext` isn't thread-safe. Reading or writing properties of the `HttpContext` outside of processing a request can result in a NullReferenceException.

ⓘ **Note**

If your app generates sporadic `NullReferenceException` errors, review parts of the code that start background processing or that continue processing after a request completes. Look for mistakes, such as defining a controller method as `async void`.

To safely do background work with `HttpContext` data:

- Copy the required data during request processing.
- Pass the copied data to a background task.
- Do ***not*** reference `HttpContext` data in parallel tasks. Extract the data needed from the context before starting the parallel tasks.

To avoid unsafe code, never pass `HttpContext` into a method that does background work. Pass the required data instead. In the following example, `SendEmail` calls `SendEmailCoreAsync` to start sending an email. The value of the `X-Correlation-Id` header is passed to `SendEmailCoreAsync` instead of the `HttpContext`. Code execution doesn't wait for `SendEmailCoreAsync` to complete:

```C#
public class EmailController : Controller
{
    public IActionResult SendEmail(string email)
    {
        var correlationId = HttpContext.Request.Headers["X-Correlation-
Id"].ToString();

        _ = SendEmailCoreAsync(correlationId);

        return View();
    }

    private async Task SendEmailCoreAsync(string correlationId)
    {
        // ...
    }
}
```

`IHttpContextAccessor`/`HttpContext` in Razor components (Blazor)

IHttpContextAccessor must be avoided with interactive rendering because there isn't a valid `HttpContext` available.

IHttpContextAccessor can be used for components that are statically rendered on the server. **However, we recommend avoiding it if possible.**

HttpContext can be used as a cascading parameter only in *statically-rendered root components* for general tasks, such as inspecting and modifying headers or other properties in the `App` component (`Components/App.razor`). The value is always `null` for interactive rendering.

```C#
[CascadingParameter]
public HttpContext? HttpContext { get; set; }
```

For scenarios where the HttpContext is required in interactive components, we recommend flowing the data via persistent component state from the server. For more information, see Server-side ASP.NET Core Blazor additional security scenarios.

Detect changes with change tokens in ASP.NET Core

Article • 07/26/2024

Important

This information relates to a pre-release product that may be substantially modified before it's commercially released. Microsoft makes no warranties, express or implied, with respect to the information provided here.

For the current release, see the .NET 9 version of this article.

A *change token* is a general-purpose, low-level building block used to track state changes.

View or download sample code (how to download)

IChangeToken interface

IChangeToken propagates notifications that a change has occurred. `IChangeToken` resides in the Microsoft.Extensions.Primitives namespace. The Microsoft.Extensions.Primitives NuGet package is implicitly provided to the ASP.NET Core apps.

`IChangeToken` has two properties:

- ActiveChangeCallbacks indicate if the token proactively raises callbacks. If `ActiveChangedCallbacks` is set to `false`, a callback is never called, and the app must poll `HasChanged` for changes. It's also possible for a token to never be cancelled if no changes occur or the underlying change listener is disposed or disabled.
- HasChanged receives a value that indicates if a change has occurred.

The `IChangeToken` interface includes the RegisterChangeCallback(Action<Object>, Object) method, which registers a callback that's invoked when the token has changed. `HasChanged` must be set before the callback is invoked.

ChangeToken class

ChangeToken is a static class used to propagate notifications that a change has occurred. `ChangeToken` resides in the Microsoft.Extensions.Primitives namespace. The Microsoft.Extensions.Primitives NuGet package is implicitly provided to the ASP.NET Core apps.

The ChangeToken.OnChange(Func<IChangeToken>, Action) method registers an `Action` to call whenever the token changes:

- `Func<IChangeToken>` produces the token.
- `Action` is called when the token changes.

The ChangeToken.OnChange<TState>(Func<IChangeToken>, Action<TState>, TState) overload takes an additional `TState` parameter that's passed into the token consumer `Action`.

`OnChange` returns an IDisposable. Calling Dispose stops the token from listening for further changes and releases the token's resources.

Example uses of change tokens in ASP.NET Core

Change tokens are used in prominent areas of ASP.NET Core to monitor for changes to objects:

- For monitoring changes to files, IFileProvider's Watch method creates an `IChangeToken` for the specified files or folder to watch.
- `IChangeToken` tokens can be added to cache entries to trigger cache evictions on change.
- For `TOptions` changes, the default OptionsMonitor<TOptions> implementation of IOptionsMonitor<TOptions> has an overload that accepts one or more IOptionsChangeTokenSource<TOptions> instances. Each instance returns an `IChangeToken` to register a change notification callback for tracking options changes.

Monitor for configuration changes

By default, ASP.NET Core templates use JSON configuration files (`appsettings.json`, `appsettings.Development.json`, and `appsettings.Production.json`) to load app configuration settings.

These files are configured using the AddJsonFile(IConfigurationBuilder, String, Boolean, Boolean) extension method on ConfigurationBuilder that accepts a `reloadOnChange` parameter. `reloadOnChange` indicates if configuration should be reloaded on file changes. This setting appears in the Host convenience method CreateDefaultBuilder:

```
C#
config.AddJsonFile("appsettings.json", optional: true, reloadOnChange: true)
      .AddJsonFile($"appsettings.{env.EnvironmentName}.json", optional:
true,
          reloadOnChange: true);
```

File-based configuration is represented by FileConfigurationSource. `FileConfigurationSource` uses IFileProvider to monitor files.

By default, the `IFileMonitor` is provided by a PhysicalFileProvider, which uses FileSystemWatcher to monitor for configuration file changes.

The sample app demonstrates two implementations for monitoring configuration changes. If any of the `appsettings` files change, both of the file monitoring implementations execute custom code—the sample app writes a message to the console.

A configuration file's `FileSystemWatcher` can trigger multiple token callbacks for a single configuration file change. To ensure that the custom code is only run once when multiple token callbacks are triggered, the sample's implementation checks file hashes. The sample uses SHA1 file hashing. A retry is implemented with an exponential back-off.

`Utilities/Utilities.cs`:

```
C#
public static byte[] ComputeHash(string filePath)
{
    var runCount = 1;

    while(runCount < 4)
    {
        try
        {
            if (File.Exists(filePath))
            {
                using (var fs = File.OpenRead(filePath))
                {
                    return System.Security.Cryptography.SHA1
                        .Create().ComputeHash(fs);
                }
            }
            else
            {
                throw new FileNotFoundException();
            }
        }
        catch (IOException ex)
        {
            if (runCount == 3)
            {
                throw;
            }

            Thread.Sleep(TimeSpan.FromSeconds(Math.Pow(2, runCount)));
            runCount++;
        }
    }

    return new byte[20];
}
```

Simple startup change token

Register a token consumer `Action` callback for change notifications to the configuration reload token.

In `Startup.Configure`:

C#

```
ChangeToken.OnChange(
    () => config.GetReloadToken(),
    (state) => InvokeChanged(state),
    env);
```

`config.GetReloadToken()` provides the token. The callback is the `InvokeChanged` method:

C#

```
private void InvokeChanged(IWebHostEnvironment env)
{
    byte[] appsettingsHash = ComputeHash("appSettings.json");
    byte[] appsettingsEnvHash =
        ComputeHash($"appSettings.{env.EnvironmentName}.json");

    if (!_appsettingsHash.SequenceEqual(appsettingsHash) ||
        !_appsettingsEnvHash.SequenceEqual(appsettingsEnvHash))
    {
        _appsettingsHash = appsettingsHash;
        _appsettingsEnvHash = appsettingsEnvHash;

        WriteConsole("Configuration changed (Simple Startup Change Token)");
    }
}
```

The `state` of the callback is used to pass in the `IWebHostEnvironment`, which is useful for specifying the correct `appsettings` configuration file to monitor (for example, `appsettings.Development.json` when in the Development environment). File hashes are used to prevent the `WriteConsole` statement from running multiple times due to multiple token callbacks when the configuration file has only changed once.

This system runs as long as the app is running and can't be disabled by the user.

Monitor configuration changes as a service

The sample implements:

- Basic startup token monitoring.
- Monitoring as a service.
- A mechanism to enable and disable monitoring.

The sample establishes an `IConfigurationMonitor` interface.

`Extensions/ConfigurationMonitor.cs`:

```C#
public interface IConfigurationMonitor
{
    bool MonitoringEnabled { get; set; }
    string CurrentState { get; set; }
}
```

The constructor of the implemented class, `ConfigurationMonitor`, registers a callback for change notifications:

```C#
public ConfigurationMonitor(IConfiguration config, IWebHostEnvironment env)
{
    _env = env;

    ChangeToken.OnChange<IConfigurationMonitor>(
        () => config.GetReloadToken(),
        InvokeChanged,
        this);
}
```

```
public bool MonitoringEnabled { get; set; } = false;
public string CurrentState { get; set; } = "Not monitoring";
```

`config.GetReloadToken()` supplies the token. `InvokeChanged` is the callback method. The `state` in this instance is a reference to the `IConfigurationMonitor` instance that's used to access the monitoring state. Two properties are used:

- `MonitoringEnabled`: Indicates if the callback should run its custom code.
- `CurrentState`: Describes the current monitoring state for use in the UI.

The `InvokeChanged` method is similar to the earlier approach, except that it:

- Doesn't run its code unless `MonitoringEnabled` is `true`.
- Outputs the current `state` in its `WriteConsole` output.

C#

```csharp
private void InvokeChanged(IConfigurationMonitor state)
{
    if (MonitoringEnabled)
    {
        byte[] appsettingsHash = ComputeHash("appSettings.json");
        byte[] appsettingsEnvHash =
            ComputeHash($"appSettings.{_env.EnvironmentName}.json");

        if (!_appsettingsHash.SequenceEqual(appsettingsHash) ||
            !_appsettingsEnvHash.SequenceEqual(appsettingsEnvHash))
        {
            string message = $"State updated at {DateTime.Now}";


            _appsettingsHash = appsettingsHash;
            _appsettingsEnvHash = appsettingsEnvHash;

            WriteConsole("Configuration changed (ConfigurationMonitor Class)
" +
                $"{message}, state:{state.CurrentState}");
        }
    }
}
```

An instance `ConfigurationMonitor` is registered as a service in `Startup.ConfigureServices`:

C#

```csharp
services.AddSingleton<IConfigurationMonitor, ConfigurationMonitor>();
```

The Index page offers the user control over configuration monitoring. The instance of `IConfigurationMonitor` is injected into the `IndexModel`.

`Pages/Index.cshtml.cs`:

C#

```csharp
public IndexModel(
    IConfiguration config,
    IConfigurationMonitor monitor,
    FileService fileService)
{
    _config = config;
    _monitor = monitor;
    _fileService = fileService;
}
```

The configuration monitor (`_monitor`) is used to enable or disable monitoring and set the current state for UI feedback:

C#

```csharp
public IActionResult OnPostStartMonitoring()
{
    _monitor.MonitoringEnabled = true;
    _monitor.CurrentState = "Monitoring!";

    return RedirectToPage();
}

public IActionResult OnPostStopMonitoring()
{
    _monitor.MonitoringEnabled = false;
    _monitor.CurrentState = "Not monitoring";

    return RedirectToPage();
}
```

When `OnPostStartMonitoring` is triggered, monitoring is enabled, and the current state is cleared. When `OnPostStopMonitoring` is triggered, monitoring is disabled, and the state is set to reflect that monitoring isn't occurring.

Buttons in the UI enable and disable monitoring.

`Pages/Index.cshtml`:

CSHTML

```
<button class="btn btn-success" asp-page-handler="StartMonitoring">
    Start Monitoring
</button>

<button class="btn btn-danger" asp-page-handler="StopMonitoring">
    Stop Monitoring
</button>
```

Monitor cached file changes

File content can be cached in-memory using IMemoryCache. In-memory caching is described in the Cache in-memory topic. Without taking additional steps, such as the implementation described below, *stale* (outdated) data is returned from a cache if the source data changes.

For example, not taking into account the status of a cached source file when renewing a sliding expiration period leads to stale cached file data. Each request for the data renews the sliding expiration period, but the file is never reloaded into the cache. Any app features that use the file's cached content are subject to possibly receiving stale content.

Using change tokens in a file caching scenario prevents the presence of stale file content in the cache. The sample app demonstrates an implementation of the approach.

The sample uses `GetFileContent` to:

- Return file content.
- Implement a retry algorithm with exponential back-off to cover cases where a file access problem temporarily delays reading the file's content.

`Utilities/Utilities.cs`:

C#

```csharp
public async static Task<string> GetFileContent(string filePath)
{
    var runCount = 1;

    while(runCount < 4)
    {
        try
        {
            if (File.Exists(filePath))
            {
                using (var fileStreamReader = File.OpenText(filePath))
                {
                    return await fileStreamReader.ReadToEndAsync();
                }
            }
            else
            {
                throw new FileNotFoundException();
            }
        }
        catch (IOException ex)
        {
            if (runCount == 3)
            {
                throw;
            }

            Thread.Sleep(TimeSpan.FromSeconds(Math.Pow(2, runCount)));
            runCount++;
        }
    }

    return null;
}
```

A `FileService` is created to handle cached file lookups. The `GetFileContent` method call of the service attempts to obtain file content from the in-memory cache and return it to the caller (`Services/FileService.cs`).

If cached content isn't found using the cache key, the following actions are taken:

1. The file content is obtained using `GetFileContent`.
2. A change token is obtained from the file provider with IFileProviders.Watch. The token's callback is triggered when the file is modified.
3. The file content is cached with a sliding expiration period. The change token is attached with MemoryCacheEntryExtensions.AddExpirationToken to evict the cache entry if the file changes while it's cached.

In the following example, files are stored in the app's content root. `IWebHostEnvironment.ContentRootFileProvider` is used to obtain an IFileProvider pointing at the app's `IWebHostEnvironment.ContentRootPath`. The `filePath` is obtained with IFileInfo.PhysicalPath.

C#

```csharp
public class FileService
{
    private readonly IMemoryCache _cache;
    private readonly IFileProvider _fileProvider;
    private List<string> _tokens = new List<string>();

    public FileService(IMemoryCache cache, IWebHostEnvironment env)
    {
        _cache = cache;
        _fileProvider = env.ContentRootFileProvider;
    }

    public async Task<string> GetFileContents(string fileName)
    {
        var filePath = _fileProvider.GetFileInfo(fileName).PhysicalPath;
        string fileContent;

        // Try to obtain the file contents from the cache.
        if (_cache.TryGetValue(filePath, out fileContent))
        {
            return fileContent;
        }

        // The cache doesn't have the entry, so obtain the file
        // contents from the file itself.
        fileContent = await GetFileContent(filePath);

        if (fileContent != null)
        {
            // Obtain a change token from the file provider whose
            // callback is triggered when the file is modified.
            var changeToken = _fileProvider.Watch(fileName);

            // Configure the cache entry options for a five minute
            // sliding expiration and use the change token to
            // expire the file in the cache if the file is
            // modified.
            var cacheEntryOptions = new MemoryCacheEntryOptions()
                .SetSlidingExpiration(TimeSpan.FromMinutes(5))
                .AddExpirationToken(changeToken);

            // Put the file contents into the cache.
            _cache.Set(filePath, fileContent, cacheEntryOptions);

            return fileContent;
        }

        return string.Empty;
    }
}
```

The `FileService` is registered in the service container along with the memory caching service.

In `Startup.ConfigureServices`:

C#

```
services.AddMemoryCache();
services.AddSingleton<FileService>();
```

The page model loads the file's content using the service.

In the Index page's `OnGet` method (`Pages/Index.cshtml.cs`):

```C#
var fileContent = await _fileService.GetFileContents("poem.txt");
```

CompositeChangeToken class

For representing one or more `IChangeToken` instances in a single object, use the CompositeChangeToken class.

```C#
var firstCancellationTokenSource = new CancellationTokenSource();
var secondCancellationTokenSource = new CancellationTokenSource();

var firstCancellationToken = firstCancellationTokenSource.Token;
var secondCancellationToken = secondCancellationTokenSource.Token;

var firstCancellationChangeToken = new
CancellationChangeToken(firstCancellationToken);
var secondCancellationChangeToken = new
CancellationChangeToken(secondCancellationToken);

var compositeChangeToken =
    new CompositeChangeToken(
        new List<IChangeToken>
        {
            firstCancellationChangeToken,
            secondCancellationChangeToken
        });
```

`HasChanged` on the composite token reports `true` if any represented token `HasChanged` is `true`. `ActiveChangeCallbacks` on the composite token reports `true` if any represented token `ActiveChangeCallbacks` is `true`. If multiple concurrent change events occur, the composite change callback is invoked one time.

Additional resources

- Cache in-memory in ASP.NET Core
- Distributed caching in ASP.NET Core
- Response caching in ASP.NET Core
- Response Caching Middleware in ASP.NET Core
- Cache Tag Helper in ASP.NET Core MVC
- Distributed Cache Tag Helper in ASP.NET Core

Open Web Interface for .NET (OWIN) with ASP.NET Core

Article • 12/02/2024

By Steve Smith and Rick Anderson

ASP.NET Core:

- Supports the Open Web Interface for .NET (OWIN).
- Has .NET Core compatible replacements for the `Microsoft.Owin.*` (Katana) libraries.

OWIN allows web apps to be decoupled from web servers. It defines a standard way for middleware to be used in a pipeline to handle requests and associated responses. ASP.NET Core applications and middleware can interoperate with OWIN-based applications, servers, and middleware.

OWIN provides a decoupling layer that allows two frameworks with disparate object models to be used together. The `Microsoft.AspNetCore.Owin` package provides two adapter implementations:

- ASP.NET Core to OWIN
- OWIN to ASP.NET Core

This allows ASP.NET Core to be hosted on top of an OWIN compatible server/host or for other OWIN compatible components to be run on top of ASP.NET Core.

Note

Using these adapters comes with a performance cost. Apps using only ASP.NET Core components shouldn't use the `Microsoft.AspNetCore.Owin` package or adapters.

View or download sample code (how to download)

Running OWIN middleware in the ASP.NET Core pipeline

ASP.NET Core's OWIN support is deployed as part of the `Microsoft.AspNetCore.Owin` package. You can import OWIN support into your project by installing this package.

OWIN middleware conforms to the OWIN specification, which requires a `Func<IDictionary<string, object>, Task>` interface, and specific keys be set (such as `owin.ResponseBody`). The following simple OWIN middleware displays "Hello World":

C#

```csharp
public Task OwinHello(IDictionary<string, object> environment)
{
    string responseText = "Hello World via OWIN";
    byte[] responseBytes = Encoding.UTF8.GetBytes(responseText);

    // OWIN Environment Keys: https://owin.org/spec/spec/owin-1.0.0.html
    var responseStream = (Stream)environment["owin.ResponseBody"];
    var responseHeaders = (IDictionary<string,
string[]>)environment["owin.ResponseHeaders"];

    responseHeaders["Content-Length"] = new string[] {
responseBytes.Length.ToString(CultureInfo.InvariantCulture) };
    responseHeaders["Content-Type"] = new string[] { "text/plain" };

    return responseStream.WriteAsync(responseBytes, 0,
responseBytes.Length);
}
```

The sample signature returns a `Task` and accepts an `IDictionary<string, object>` as required by OWIN.

The following code shows how to add the `OwinHello` middleware (shown above) to the ASP.NET Core pipeline with the `UseOwin` extension method.

C#

```csharp
public void Configure(IApplicationBuilder app)
{
    app.UseOwin(pipeline =>
    {
        pipeline(next => OwinHello);
    });
}
```

You can configure other actions to take place within the OWIN pipeline.

Note

Response headers should only be modified prior to the first write to the response stream.

ⓘ Note

Multiple calls to `UseOwin` is discouraged for performance reasons. OWIN components will operate best if grouped together.

C#

```
app.UseOwin(pipeline =>
{
    pipeline(next =>
    {
        return async environment =>
        {
            // Do something before.
            await next(environment);
            // Do something after.
        };
    });
});
```

OWIN environment

You can construct an OWIN environment using the `HttpContext`.

C#

```

   var environment = new OwinEnvironment(HttpContext);
   var features = new OwinFeatureCollection(environment);
```

OWIN keys

OWIN depends on an `IDictionary<string,object>` object to communicate information throughout an HTTP Request/Response exchange. ASP.NET Core implements the keys listed below. See the primary specification, extensions, and OWIN Key Guidelines and Common Keys.

Request data (OWIN v1.0.0)

Expand table

Key	Value (type)	Description
owin.RequestScheme	`String`	
owin.RequestMethod	`String`	
owin.RequestPathBase	`String`	
owin.RequestPath	`String`	
owin.RequestQueryString	`String`	
owin.RequestProtocol	`String`	
owin.RequestHeaders	`IDictionary<string,string[]>`	
owin.RequestBody	`Stream`	

Request data (OWIN v1.1.0)

Expand table

Key	Value (type)	Description
owin.RequestId	`String`	Optional

Response data (OWIN v1.0.0)

Expand table

Key	Value (type)	Description
owin.ResponseStatusCode	`int`	Optional
owin.ResponseReasonPhrase	`String`	Optional
owin.ResponseHeaders	`IDictionary<string,string[]>`	
owin.ResponseBody	`Stream`	

Other data (OWIN v1.0.0)

Expand table

Key	Value (type)	Description
owin.CallCancelled	`CancellationToken`	
owin.Version	`String`	

Common keys

Expand table

Key	Value (type)	Description
ssl.ClientCertificate	`X509Certificate`	
ssl.LoadClientCertAsync	`Func<Task>`	
server.RemoteIpAddress	`String`	
server.RemotePort	`String`	
server.LocalIpAddress	`String`	
server.LocalPort	`String`	
server.OnSendingHeaders	`Action<Action<object>,object>`	

SendFiles v0.3.0

Expand table

Key	Value (type)	Description
sendfile.SendAsync	See delegate signature	Per Request

Opaque v0.3.0

Expand table

Key	Value (type)	Description
opaque.Version	`String`	
opaque.Upgrade	`OpaqueUpgrade`	See delegate signature
opaque.Stream	`Stream`	

Key	Value (type)	Description
opaque.CallCancelled	`CancellationToken`	

WebSocket v0.3.0

Expand table

Key	Value (type)	Description
websocket.Version	`String`	
websocket.Accept	`WebSocketAccept`	See delegate signature
websocket.AcceptAlt		Non-spec
websocket.SubProtocol	`String`	See RFC6455 Section 4.2.2 Step 5.5
websocket.SendAsync	`WebSocketSendAsync`	See delegate signature
websocket.ReceiveAsync	`WebSocketReceiveAsync`	See delegate signature
websocket.CloseAsync	`WebSocketCloseAsync`	See delegate signature
websocket.CallCancelled	`CancellationToken`	
websocket.ClientCloseStatus	`int`	Optional
websocket.ClientCloseDescription	`String`	Optional

Additional resources

- See the source on GitHub for OWIN keys supported in the translation layer.
- Middleware
- Servers

Background tasks with hosted services in ASP.NET Core

Article • 05/31/2024

By Jeow Li Huan

> **Important**
>
> This information relates to a pre-release product that may be substantially modified before it's commercially released. Microsoft makes no warranties, express or implied, with respect to the information provided here.
>
> For the current release, see the .NET 9 version of this article.

In ASP.NET Core, background tasks can be implemented as *hosted services*. A hosted service is a class with background task logic that implements the IHostedService interface. This article provides three hosted service examples:

- Background task that runs on a timer.
- Hosted service that activates a scoped service. The scoped service can use dependency injection (DI).
- Queued background tasks that run sequentially.

Worker Service template

The ASP.NET Core Worker Service template provides a starting point for writing long running service apps. An app created from the Worker Service template specifies the Worker SDK in its project file:

XML

```xml
<Project Sdk="Microsoft.NET.Sdk.Worker">
```

To use the template as a basis for a hosted services app:

Visual Studio

1. Create a new project.
2. Select **Worker Service**. Select **Next**.

3. Provide a project name in the **Project name** field or accept the default project name. Select **Next**.
4. In the **Additional information** dialog, Choose a **Framework**. Select **Create**.

Package

An app based on the Worker Service template uses the `Microsoft.NET.Sdk.Worker` SDK and has an explicit package reference to the Microsoft.Extensions.Hosting package. For example, see the sample app's project file (`BackgroundTasksSample.csproj`).

For web apps that use the `Microsoft.NET.Sdk.Web` SDK, the Microsoft.Extensions.Hosting package is referenced implicitly from the shared framework. An explicit package reference in the app's project file isn't required.

IHostedService interface

The IHostedService interface defines two methods for objects that are managed by the host:

- StartAsync(CancellationToken)
- StopAsync(CancellationToken)

`StartAsync`

StartAsync(CancellationToken) contains the logic to start the background task. `StartAsync` is called *before*:

- The app's request processing pipeline is configured.
- The server is started and IApplicationLifetime.ApplicationStarted is triggered.

`StartAsync` should be limited to short running tasks because hosted services are run sequentially, and no further services are started until `StartAsync` runs to completion.

`StopAsync`

- StopAsync(CancellationToken) is triggered when the host is performing a graceful shutdown. `StopAsync` contains the logic to end the background task. Implement IDisposable and finalizers (destructors) to dispose of any unmanaged resources.

The cancellation token has a default 30 second timeout to indicate that the shutdown process should no longer be graceful. When cancellation is requested on the token:

- Any remaining background operations that the app is performing should be aborted.
- Any methods called in `StopAsync` should return promptly.

However, tasks aren't abandoned after cancellation is requested—the caller awaits all tasks to complete.

If the app shuts down unexpectedly (for example, the app's process fails), `StopAsync` might not be called. Therefore, any methods called or operations conducted in `StopAsync` might not occur.

To extend the default 30 second shutdown timeout, set:

- ShutdownTimeout when using Generic Host. For more information, see .NET Generic Host in ASP.NET Core.
- Shutdown timeout host configuration setting when using Web Host. For more information, see ASP.NET Core Web Host.

The hosted service is activated once at app startup and gracefully shut down at app shutdown. If an error is thrown during background task execution, `Dispose` should be called even if `StopAsync` isn't called.

BackgroundService base class

BackgroundService is a base class for implementing a long running IHostedService.

ExecuteAsync(CancellationToken) is called to run the background service. The implementation returns a Task that represents the entire lifetime of the background service. No further services are started until ExecuteAsync becomes asynchronous, such as by calling `await`. Avoid performing long, blocking initialization work in `ExecuteAsync`. The host blocks in StopAsync(CancellationToken) waiting for `ExecuteAsync` to complete.

The cancellation token is triggered when IHostedService.StopAsync is called. Your implementation of `ExecuteAsync` should finish promptly when the cancellation token is fired in order to gracefully shut down the service. Otherwise, the service ungracefully shuts down at the shutdown timeout. For more information, see the IHostedService interface section.

For more information, see the BackgroundService source code.

Timed background tasks

A timed background task makes use of the System.Threading.Timer class. The timer triggers the task's `DoWork` method. The timer is disabled on `StopAsync` and disposed when the service container is disposed on `Dispose`:

C#

```csharp
public class TimedHostedService : IHostedService, IDisposable
{
    private int executionCount = 0;
    private readonly ILogger<TimedHostedService> _logger;
    private Timer? _timer = null;

    public TimedHostedService(ILogger<TimedHostedService> logger)
    {
        _logger = logger;
    }

    public Task StartAsync(CancellationToken stoppingToken)
    {
        _logger.LogInformation("Timed Hosted Service running.");

        _timer = new Timer(DoWork, null, TimeSpan.Zero,
            TimeSpan.FromSeconds(5));

        return Task.CompletedTask;
    }

    private void DoWork(object? state)
    {
        var count = Interlocked.Increment(ref executionCount);

        _logger.LogInformation(
            "Timed Hosted Service is working. Count: {Count}", count);
    }

    public Task StopAsync(CancellationToken stoppingToken)
    {
        _logger.LogInformation("Timed Hosted Service is stopping.");

        _timer?.Change(Timeout.Infinite, 0);

        return Task.CompletedTask;
    }

    public void Dispose()
    {
        _timer?.Dispose();
    }
}
```

The Timer doesn't wait for previous executions of `DoWork` to finish, so the approach shown might not be suitable for every scenario. Interlocked.Increment is used to increment the execution counter as an atomic operation, which ensures that multiple threads don't update `executionCount` concurrently.

The service is registered in `IHostBuilder.ConfigureServices` (`Program.cs`) with the `AddHostedService` extension method:

```C#
services.AddHostedService<TimedHostedService>();
```

Consuming a scoped service in a background task

To use scoped services within a BackgroundService, create a scope. No scope is created for a hosted service by default.

The scoped background task service contains the background task's logic. In the following example:

- The service is asynchronous. The `DoWork` method returns a `Task`. For demonstration purposes, a delay of ten seconds is awaited in the `DoWork` method.
- An ILogger is injected into the service.

```C#
internal interface IScopedProcessingService
{
    Task DoWork(CancellationToken stoppingToken);
}

internal class ScopedProcessingService : IScopedProcessingService
{
    private int executionCount = 0;
    private readonly ILogger _logger;

    public ScopedProcessingService(ILogger<ScopedProcessingService> logger)
    {
        _logger = logger;
    }

    public async Task DoWork(CancellationToken stoppingToken)
    {
        while (!stoppingToken.IsCancellationRequested)
        {
            executionCount++;

            _logger.LogInformation(
                "Scoped Processing Service is working. Count: {Count}",
executionCount);

            await Task.Delay(10000, stoppingToken);
        }
    }
}
```

The hosted service creates a scope to resolve the scoped background task service to call its `DoWork` method. `DoWork` returns a `Task`, which is awaited in `ExecuteAsync`:

C#

```csharp
public class ConsumeScopedServiceHostedService : BackgroundService
{
    private readonly ILogger<ConsumeScopedServiceHostedService> _logger;

    public ConsumeScopedServiceHostedService(IServiceProvider services,
        ILogger<ConsumeScopedServiceHostedService> logger)
    {
        Services = services;
        _logger = logger;
    }

    public IServiceProvider Services { get; }

    protected override async Task ExecuteAsync(CancellationToken
stoppingToken)
    {
        _logger.LogInformation(
            "Consume Scoped Service Hosted Service running.");

        await DoWork(stoppingToken);
    }

    private async Task DoWork(CancellationToken stoppingToken)
    {
        _logger.LogInformation(
            "Consume Scoped Service Hosted Service is working.");

        using (var scope = Services.CreateScope())
        {
            var scopedProcessingService =
                scope.ServiceProvider
                    .GetRequiredService<IScopedProcessingService>();

            await scopedProcessingService.DoWork(stoppingToken);
        }
    }
    public override async Task StopAsync(CancellationToken stoppingToken)
    {
        _logger.LogInformation(
            "Consume Scoped Service Hosted Service is stopping.");

        await base.StopAsync(stoppingToken);
    }
}
```

The services are registered in `IHostBuilder.ConfigureServices` (`Program.cs`). The hosted service is registered with the `AddHostedService` extension method:

C#

```
services.AddHostedService<ConsumeScopedServiceHostedService>();
services.AddScoped<IScopedProcessingService, ScopedProcessingService>();
```

Queued background tasks

A background task queue is based on the .NET 4.x QueueBackgroundWorkItem:

C#

```
public interface IBackgroundTaskQueue
{
    ValueTask QueueBackgroundWorkItemAsync(Func<CancellationToken,
ValueTask> workItem);

    ValueTask<Func<CancellationToken, ValueTask>> DequeueAsync(
        CancellationToken cancellationToken);
}

public class BackgroundTaskQueue : IBackgroundTaskQueue
{
    private readonly Channel<Func<CancellationToken, ValueTask>> _queue;

    public BackgroundTaskQueue(int capacity)
    {
        // Capacity should be set based on the expected application load and
        // number of concurrent threads accessing the queue.
        // BoundedChannelFullMode.Wait will cause calls to WriteAsync() to
return a task,
        // which completes only when space became available. This leads to
backpressure,
        // in case too many publishers/calls start accumulating.
        var options = new BoundedChannelOptions(capacity)
        {
            FullMode = BoundedChannelFullMode.Wait
        };
        _queue = Channel.CreateBounded<Func<CancellationToken, ValueTask>>
(options);
    }

    public async ValueTask QueueBackgroundWorkItemAsync(
        Func<CancellationToken, ValueTask> workItem)
    {
        if (workItem == null)
        {
            throw new ArgumentNullException(nameof(workItem));
        }

        await _queue.Writer.WriteAsync(workItem);
    }

    public async ValueTask<Func<CancellationToken, ValueTask>> DequeueAsync(
        CancellationToken cancellationToken)
    {
        var workItem = await _queue.Reader.ReadAsync(cancellationToken);

        return workItem;
    }
}
```

In the following `QueueHostedService` example:

- The `BackgroundProcessing` method returns a `Task`, which is awaited in `ExecuteAsync`.
- Background tasks in the queue are dequeued and executed in `BackgroundProcessing`.
- Work items are awaited before the service stops in `StopAsync`.

C#

```
public class QueuedHostedService : BackgroundService
{
    private readonly ILogger<QueuedHostedService> _logger;

    public QueuedHostedService(IBackgroundTaskQueue taskQueue,
        ILogger<QueuedHostedService> logger)
    {
        TaskQueue = taskQueue;
        _logger = logger;
    }

    public IBackgroundTaskQueue TaskQueue { get; }

    protected override async Task ExecuteAsync(CancellationToken
stoppingToken)
    {
        _logger.LogInformation(
```