# Globalization and localization in ASP.NET Core

11/30/2019 • 34 minutes to read •      +18

### In this article



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

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

Internationalization involves Globalization and Localization. Globalization is the process of designing apps that support different cultures. Globalization adds support for input, display, and output of a defined set of language scripts that relate to specific geographic areas.

Localization is the process of adapting a globalized app, which you have already processed for localizability, to a particular culture/locale. For more information see **Globalization and localization terms** near the end of this document.

App localization involves the following:

1. Make the app's content localizable
2. Provide localized resources for the languages and cultures you support
3. Implement a strategy to select the language/culture for each request

View or download sample code (how to download)

# Make the app's content localizable

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 code below shows how to wrap the string "About Title" for localization.

C# Copy

```
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 your 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 will prefer the traditional work flow as it can make it easier to work with longer string literals and make it easier to update localized strings.

Use the `IHtmlLocalizer<T>` 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 sample highlighted below, only the value of `name` parameter is HTML encoded.

C# Copy

```
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.

At the lowest level, you can get `IStringLocalizerFactory` out of Dependency Injection:

C# Copy

```
{
    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."];
```

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

You can partition your localized strings by controller, area, or have just one container. In the sample app, a dummy class named `SharedResource` is used for shared resources.

C# Copy

```
// Dummy class to group shared resources

namespace Localization
{
    public class SharedResource
    {
    }
}
```

Some developers use the `Startup` class to contain global or shared strings. In the sample below, the `InfoController` and the `SharedResource` localizers are used:

C# Copy

```
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 Copy

```
@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` will HTML encode the parameters, but not the resource string. Consider the following Razor markup:

CSHTML Copy

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

A French resource file could contain the following:

| 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.

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

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

CSHTML Copy

```
@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# Copy

```
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; }
}
```

In ASP.NET Core MVC 1.1.0 and higher, non-validation attributes are localized. ASP.NET Core MVC 1.0 does **not** look up localized strings for non-validation attributes.

## Using one resource string for multiple classes

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

C# Copy

```
public void ConfigureServices(IServiceCollection services)
{
    services.AddMvc()
        .AddDataAnnotationsLocalization(options => {
            options.DataAnnotationLocalizerProvider = (type, factory) =>
                factory.Create(typeof(SharedResource));
        });
}
```

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

# Provide localized resources for the languages and cultures you support

## SupportedCultures and SupportedUICultures

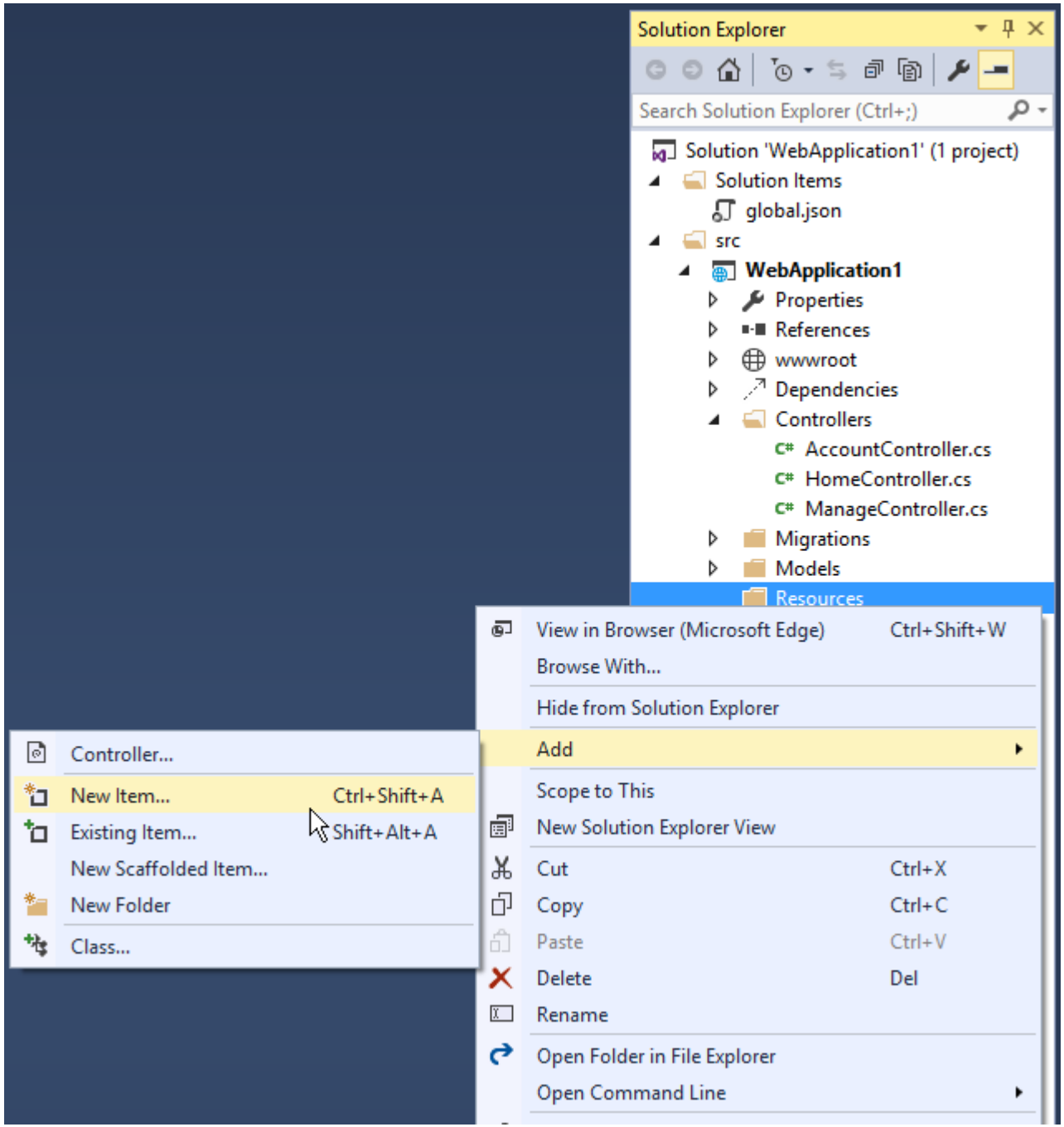
ASP.NET Core allows you to specify two culture values, `SupportedCultures` and `SupportedUICultures`. The CultureInfo object for `SupportedCultures` determines the results of culture-dependent functions, such as date, time, number, and currency formatting. `SupportedCultures` also determines the sorting order of text, casing conventions, and string comparisons. See CultureInfo.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's 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".

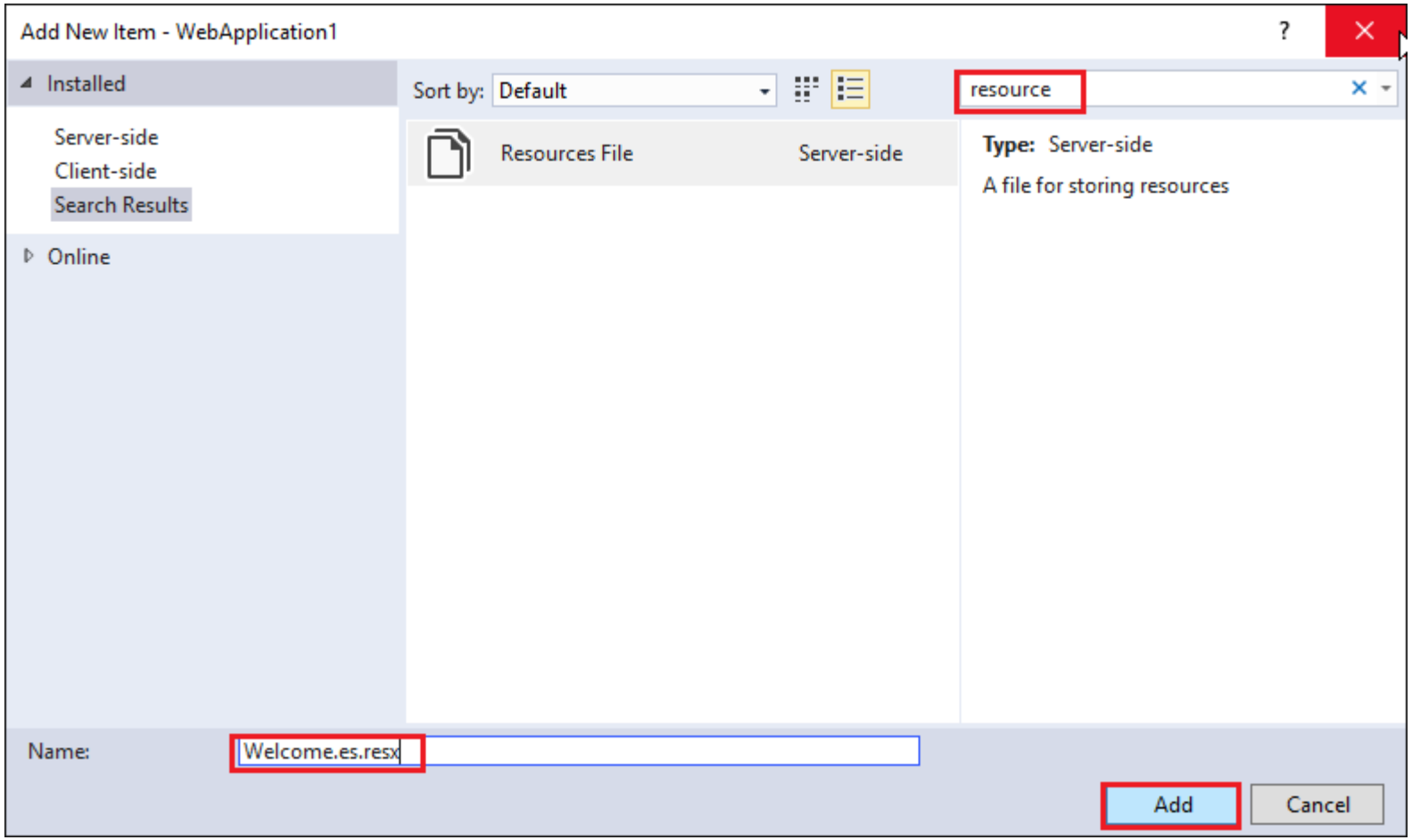
# Resource files

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 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:

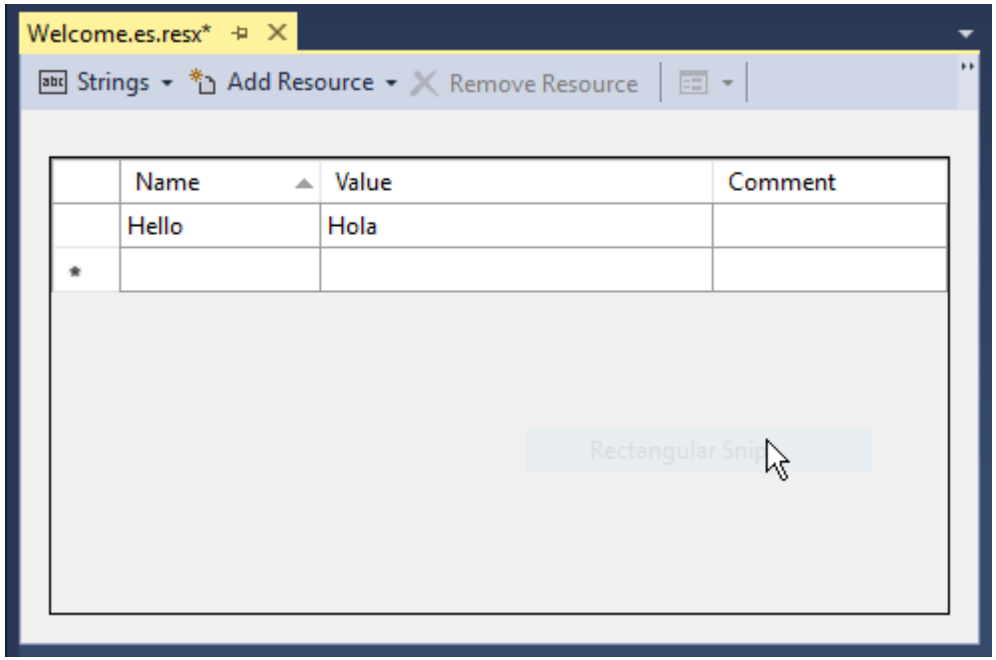
1. In **Solution Explorer**, right click on the folder which will contain the resource file > **Add** > **New Item**.



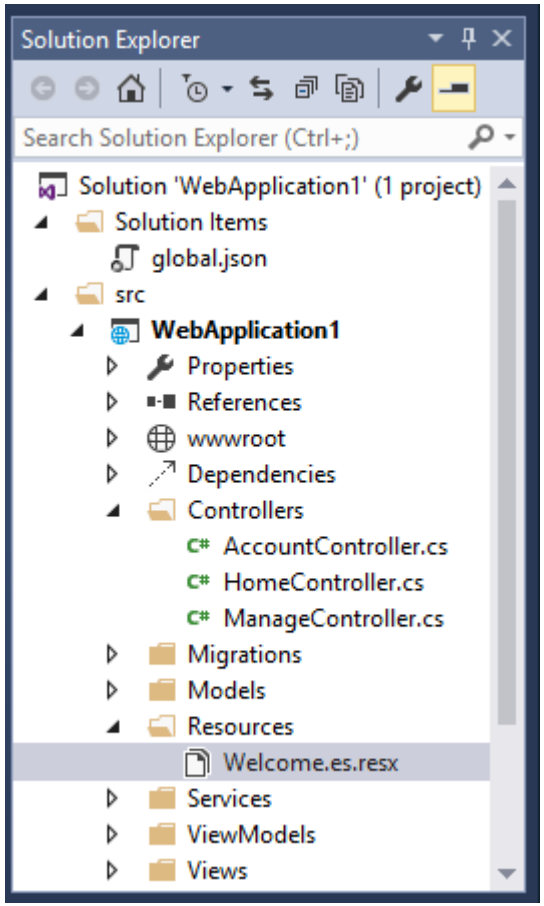

2. In the **Search installed templates** box, enter "resource" and name the file.




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




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.

| 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# Copy

```
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 ISO. 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 ISO language codes are part of the file name (for example, **en-us**, **fr-ca**, and **en-gb**). These ISO codes are placed between the file name and the *.resx* file extension, as in *Welcome.es-MX.resx* (Spanish/Mexico).

# Implement a strategy to select the language/culture for each request

# Configure localization

Localization is configured in the `Startup.ConfigureServices` method:

C# Copy

```
services.AddLocalization(options => options.ResourcesPath = "Resources");

services.AddMvc()
    .AddViewLocalization(LanguageViewLocationExpanderFormat.Suffix)
    .AddDataAnnotationsLocalization();
```

- `AddLocalization` adds the localization services to the services container. The code above 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.

# Localization middleware

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

C# Copy

```
var supportedCultures = new[] { "en-US", "fr" };
var localizationOptions = new RequestLocalizationOptions().SetDefaultCulture(supportedCultures[0])
    .AddSupportedCultures(supportedCultures)
```

```
    .AddSupportedUICultures(supportedCultures);

app.UseRequestLocalization(localizationOptions);
```

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

`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 we'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:

Copy

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

If you only pass in one of the two (`culture` or `ui-culture`), the query string provider will set both values using the one you 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 you only specify one of culture info and UI culture, the specified culture will be used for both culture info and UI culture.

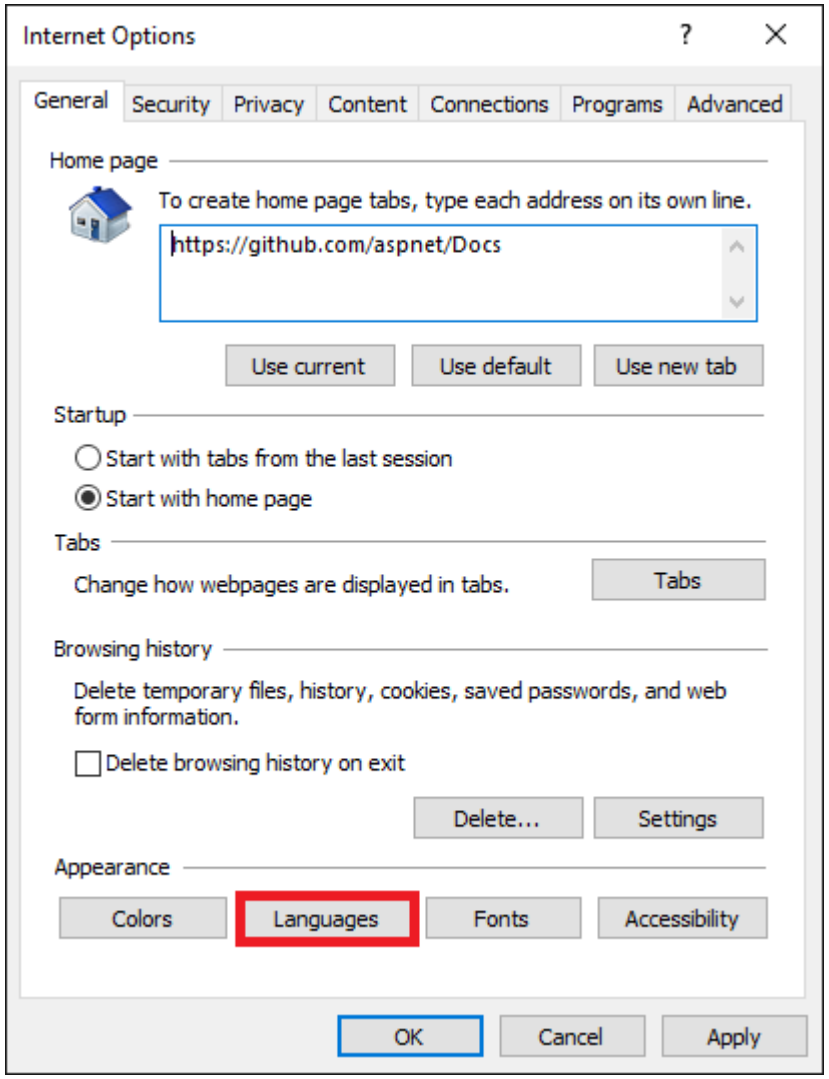
## The Accept-Language HTTP header

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 IE

1. From the gear icon, tap **Internet Options**.

2. Tap **Languages**.




3. Tap **Set Language Preferences**.

4. Tap **Add a language**.

5. Add the language.

6. Tap the language, then tap **Move Up**.

## 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# Copy

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

## 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# Copy

```
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 new ProviderCultureResult("en");
    }));
});
```

Use `RequestLocalizationOptions` to add or remove localization providers.

## 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 Copy

```
@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 Copy

```
<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# Copy

```
[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.

# Globalization and localization terms

The process of localizing your app also requires a basic understanding of relevant character sets commonly used in modern software development and an understanding of the issues associated with them. Although all computers store text as numbers (codes), different systems store the same text using different numbers. The localization process refers to translating the app user interface (UI) for a specific culture/locale.

Localizability is an intermediate process for verifying that a globalized app is ready for localization.

The RFC 4646 format for the culture name is `<languagecode2>-<country/regioncode2>`, where `<languagecode2>` is the language code and `<country/regioncode2>` is the subculture code. 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.

Internationalization is often abbreviated to "I18N". The abbreviation takes the first and last letters and the number of letters between them, so 18 stands for the number of letters between the first "I" and the last "N". The same applies to Globalization (G11N), and Localization (L10N).

Terms:

- Globalization (G11N): The process of making an app support different languages and regions.
- Localization (L10N): The process of customizing an app for a given language and region.
- Internationalization (I18N): Describes both globalization and localization.
- Culture: It's 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.

> **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 issue 4076** for instructions on adding decimal comma.

> **Note**
>
> Prior to ASP.NET Core 3.0 web apps write one log of type `LogLevel.Warning` per request if the requested culture is unsupported. Logging one `LogLevel.Warning` per request is can make large log files with redundant information. This behavior has been changed in ASP.NET 3.0. The `RequestLocalizationMiddleware` writes a log of type `LogLevel.Debug`, which reduces the size of production logs.

# Additional resources

- Troubleshoot ASP.NET Core Localization
- Localization.StarterWeb project used in the article.
- Globalizing and localizing .NET applications
- Resources in .resx Files
- Microsoft Multilingual App Toolkit
- Localization & Generics

## Is this page helpful?

Yes No

# Recommended content

### Use LibMan with ASP.NET Core in Visual Studio

Learn how to use LibMan in an ASP.NET Core project with Visual Studio.

### Bundle and minify static assets in ASP.NET Core

Learn how to optimize static resources in an ASP.NET Core web application by applying bundling and minification techniques.

### Troubleshoot ASP.NET Core Localization

Learn how to diagnose problems with localization in ASP.NET Core apps.

### Razor file compilation in ASP.NET Core

Learn how compilation of Razor files occurs in an ASP.NET Core app.

Show more