Tutorials

ASP.NET Core Localization

Dasun Nirmitha, March 24, 2021 · 23 min read

As ASP.NET Core web developers, what are our main concerns when developing our catchy web applications? *UI/UX, performance, scalability, etc.* would come to mind, but would the squeaky clean sites we spent sleepless hours crafting matter if our sites look like just random sets of

Related posts

Insights · Localization

Design-stage localization: 3 reasons why it's the solution for fast-growing, agile companies (Part 1)

symbols to our site viewers? Hence surfaces the much-needed requirement of *localization*.

ASP.NET opened its hands beyond Microsoft Windows as a cross-platform web framework starting from ASP.NET Core. But instead of stopping there, Microsoft chose to provide a diverse list of localization and internationalization functions to help our ASP.NET Core applications reach broader ranges of locales and cultures. Let's put them to use, shall we? So, in this article, let us take a look at how l10n/i18n works on the ASP.NET Core framework.

We will cover the following topics in this tutorial:

- ASP.NET Core i18n/l10n (internationalization/localization).
- Step-by-step guide on basic ASP.NET Core MVC Web application.
- Adding language resources and conventions followed.
- Localizing with the help of `ResourceManager`.
- Automatically change app culture using `UseRequestLocalization` middleware.

Tutorials

Vue i18n: Building a multi-lingual app

Tutorials

Laravel localization: A step-by-step guide

Tutorials

React i18n: A step-by-step guide to React-intl

Tutorials

Angular i18n: Performing translations with built-in module

Tutorials

Spring Boot internationalization: Step-by-step

- Localize controllers using `IStringLocalizer`, `IHtmlLocalizer`, and views using `IViewLocalizer`.
- Identify user's culture using `IRequestCultureProvider` implementations.
- Date and time format localization.
- Usage of placeholders.

Assumptions

Basic knowledge of:

- Microsoft ASP.NET
- C#

- MVC

Prerequisites

Local environment set up with:

- ASP.NET Core 3.1+ (latest LTS release at the time of writing)
- Visual Studio 2019 IDE
- Any API Client (e.g.: Postman)

Environment

I will be using the following environment for my development purposes:

- Visual Studio Community 2019 16.9.1
- .NET Framework 4.8.04084

- Postman 8.0.7

The source code is available on GitHub.

Basic ASP.NET Core project awaiting localization

Before anything else, let's go ahead and set up a simple ASP.NET Core project which we can later transform into an internationalized web application.

Let's open up Visual Studio and create an empty project with the following configuration:

```
Template:          ASP.NET Core Empty
Name:              ASPNETCoreL10n
Target Framework:  .NET Core 3.1
```

Note: *Let's tick the "Place solution and project in the same directory" option since we are not planning to join multiple solutions within this project.*

MVC paradigm

Time to make our `ASPNETCoreL10n` project follow the MVC design model.

Firstly, let's open up the `Startup.cs` and place the following inside its `ConfigureServices` method:

```
services.AddControllersWithViews();   //1
services.AddRazorPages();             //2
```

1. Adds services related to MVC controllers and views to the Dependency Injection container of the project.
2. Adds services related to Razor pages to the Dependency Injection container.

Secondly, let's add the MVC middleware to the application request processing pipeline.

Let's head over to the `Configure` method within the `Startup.cs` class. Now, let us replace the current `endpoints.MapGet` endpoint inside the `app.UseEndpoints` middleware as follows:

```
 public void Configure(IApplicationBuilder app,
IWebHostEnvironment env)
 {
 .
     app.UseRouting();  //1
     app.UseEndpoints(endpoints =>
     {
         endpoints.MapControllerRoute(  //2
             name: "default",
             pattern: "
{controller=home}/{action=index}/{id?}");
     });
 }
```

1. Add EndpointRoutingApplicationBuilderExtensions.UseRouting middleware that performs request-to-endpoint route matching.
2. Use ControllerEndpointRouteBuilderExtensions.MapControllerRoute middleware to add a Controller endpoint route. This middleware specifies a route named “default” that looks for an “Index” action within a Controller that has a basename of “home”.

Important Note: *Make sure to always put* `MapControllerRoute` *middleware after* `UseRouting` *middleware in the request processing pipeline. This is so that* `UseRouting` *would have already matched the request to an endpoint by the time the execution call reaches the* `MapControllerRoute`*.*

At the moment, if we run our application it would simply give us a *'Page Not Found (404)'* error.

Therefore, let's add a simple controller to our project to match with what our "default" `MapControllerRoute` is looking for:

1. Create a `Controllers` directory within the root of the `ASPNETCoreL10n` project.
2. Add an empty MVC Controller named `HomeController.cs` within it.

Visual Studio now creates a `HomeController` class with an auto-generated `Index` action method inside it. Alright! Our "default" `MapControllerRoute` is happy now that it's got a controller endpoint to match with. But now our

`HomeController` is complaining it's got no view to return. Let's fix it, shall we?

We're going to add a Razor view page to our project. Create a `Views` directory within the root of our `ASPNETCoreL10n` project and also a `Home` directory inside it. Now, let's add a new empty Razor view `Index.cshtml` inside the `Home` folder and fill it like this:

```
<html xmlns="http://www.w3.org/1999/xhtml">
<head>
    <title>ASPNETCoreL10n</title>
</head>
<body>
    <p>Ready to get localized!</p>
</body>
</html>
```

Little note on view discovery

You might have wondered...

What's with all the new directories inside new directories when creating a simple view? Can't I just place the view

anywhere I want?

This requirement simply boils down to a process called View Discovery performed by ASP.NET Core. By default, this `View Discovery` procedure looks in the `Views/[ControllerName]` folder for a particular view.

That's all for making our basic ASP.NET Core project. Let's run the app and we'll be able to observe a browser page open showing the paragraph we added to our `Index.cshtml` view. This marks that our `ASPNETCoreL10n` project successfully matched our "default" `MapControllerRoute` with the `Index` action inside `HomeController`, and the `Index` action discovered an `Index.cshtml` view within `Views/Home` folder and returned it.

Add some resources

Before touching on localization logic, let's prepare the ground and add several language resources into our ASP.NET Core project.

In ASP.NET Core, string resources for each targeted class or view we plan to localize, are stored inside resource files having a .resx extension.

ASP.NET Core localized resource file organizing

Alright, RESX files for resources. But how should I name them? And where do I place them? Don't we need separate resource files for each language we plan to support?

Let me clear these questions for you, one by one.

Naming resource files

First question, naming resources. Resources for ASP.NET Core are named following these simple rules:

- If the namespace of the target class is equal to the current project's assembly name:

```
Resource file name = Fully qualified type name of target class – Assembly name
```

For example:

```
 Target class's fully qualified type name:
ASPNETCoreL10n.HomeController
 Current project's assembly name:
ASPNETCoreL10n
 Resource file name:
HomeController.resx
```

- If the namespace of the target class is *not* equal to the current project's assembly name:

`Resource file name` = `Fully qualified type name of target class`

For example:

```
 Target class's fully qualified type name:
ASPNETCoreUtils.StringFormatter
 Current project's assembly name:
ASPNETCoreL10n
 Resource file name:
ASPNETCoreUtils.StringFormatter.resx
```

Where to place the resources

Second question, placing the resources. We can simply place our resource files right next to the target classes or views.

Note: *You can take a look at the resource file naming section in ASP.NET Core official documentation on localization for an alternative resource organizing method based on resource path.*

Resources for multiple locales

For the third question, the answer is, yes, we need to place an isolated resource file for each language we plan to localize to.

But there's a catch! When naming these additional languages we have to strictly follow the undermentioned syntax when naming them.

For neutral culture (only the language specified) resources:

```
<resource-file-name>.<language>.resx
```

For example: `HomeController.en.resx`.

For specific culture (language and region specified) resources:

```
<resource-file-name>.<language>-<region>.resx
```

For example: `HomeController.fr-FR.resx`.

According to the terms specified on ASP.NET Core official documentation on localization, the aforementioned syntax follow RFC 4646 format consisting of an ISO 639 language code and ISO 3166 two-letter uppercase subculture code.

In other words, values valid for language and region when naming our resource files would have to take this form:

```
<ISO 639 language code>
```

or

```
<ISO 639 language code>-<ISO 3166 region code>
```

With the resource naming and placing conventions cleared up, let us add several language resources to our `ASPNETCoreL10n` project.

Let's go ahead and create a `HomeController.en-US.resx` file inside the `Controllers` directory of our project, and fill it as follows:

```
Name:  welcome
Value: Welcome!
```

Note: *`HomeController.en-US.resx` will contain localization values for the English language in the US region.*

For our localization purposes, let's add another `HomeController.fr-FR.resx` file inside the `Controllers` directory of our `ASPNETCoreL10n` project:

```
Name:  welcome
Value: Bienvenue!
```

Note: *`HomeController.fr-FR.resx` resource file will hold localization values for the French language in the France region.*

Time to touch ASP.NET Core localization

Okay, we got our ASP.NET Core project set up with MVC, and fed multiple language resources to it. Hence, we are now ready to internationalize our ASP.NET Core project to support localization on multiple locales and cultures. Let's see how!

.NET ResourceManager for ASP.NET Core localization

Let's just say using ResourceManager is the oldest and the *been-there-for-decades* way for localization in the ASP.NET

framework. Shall we find out how we can use the `ResourceManager` inside our ASP.NET Core project for localization purposes?

Firstly, let's head over to the `ASPNETCoreL10n/Startup.cs` file and add the following inside its `ConfigureServices` method:

```
 string baseName =
"ASPNETCoreL10n.Controllers.HomeController";  //1
 services.AddSingleton(new
ResourceManager(baseName,
Assembly.GetExecutingAssembly()));  //2
```

1. `baseName` string variable holds the root name `ResourceManager` should scan for resources in.
2. A `ResourceManager` instance is created passing `baseName` created in step 1 and another argument holding a reference to the currently executing assembly. Then, this `ResourceManager` instance is passed over as an argument to ServiceCollectionServiceExtensions.AddSingleton method to add a `ResourceManager` singleton service to the DI container of our project.

Secondly, let us visit the `ASPNETCoreL10n/HomeController` file and make these changes:

```
 public class HomeController : Controller
 {
     private readonly ResourceManager
_resourceManager;  //1
     public HomeController(ResourceManager
resourceManager)  //2
     {
         _resourceManager = resourceManager;
     }

     public IActionResult Index()
     {
         ViewData["greeting"] =
_resourceManager.GetString("welcome");  //3

         return View();  //4
     }
 }
```

1. Create a private read-only `_resourceManager` field to hold a `ResourceManager` instance.

2. `resourcemanager` parameter added to the constructor to let ASP.NET Core framework dependency inject (DI) a `ResourceManager` service to it.

3. `_resourceManager` service scans the `baseName` path we set in the previous section and retrieves a resource with a key “welcome”. The retrieved resource value is saved as a *loosely typed* ViewData with a key of “greeting”.

4. `Index` action method asks a View on the default route to render a response View passing the `ViewData` along with it. Once the response View is received, the `Index` action method returns a ViewResult holding this rendered response View.

Note: *According to ASP.NET Core View Discovery that we also discussed a while ago, the default route for our* `Index` *method inside* `HomeController` *should either be* `Views/Home/Index.cshtml` *or* `Views/Shared/Index.cshtml`*. So, our* `ASPNETCoreL10n` *project’s* `Index.cshtml` *view we created inside* `Views/Home` *directory should aptly receive this call.*

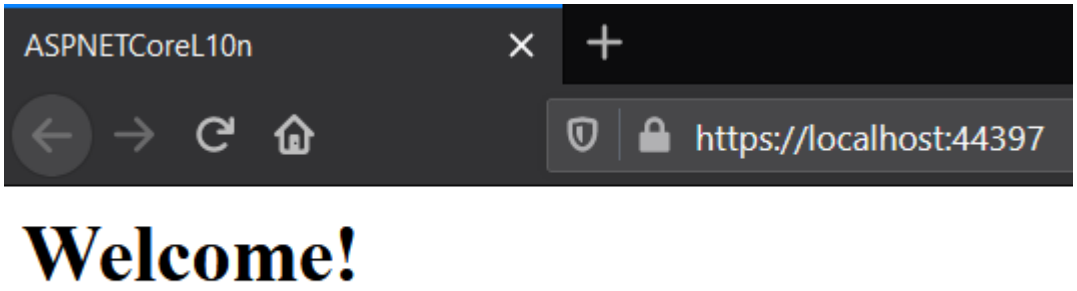
Thirdly, let’s grab our “greeting” `ViewData` sent over by the relevant vew’s `Index` action. Let us go ahead and open the

`Views/Home/Index.cshtml` view, and change its HTML body content as follows:

```
<body>
    <h1>@ViewData["greeting"]</h1>
</body>
```

Is that it?

Alright, the MVC changes related to the `ResourceManager` localization are complete. So, if we run our project now, we'll be able to notice a welcoming message appearing in our default `en-US` language:



But there's an issue! Even if we switch our browser language to `fr-FR` culture, we would still be shown the *same* `en-US`

message in the *same* `en-US` language. Let's see what's happening here.

Setting supported cultures

ASP.NET Core gets the help of SupportedCultures and SupportedUICultures properties to hold culture-related localization specifications of the application.

In particular, `SupportedCultures` property holds cultures our web app localizes to regarding culture-specific functions. These range from matters like date and time formatting to text sorting orders, likewise. On the other hand, `SupportedUICultures` simply keeps the cultures our ASP.NET Core application's UI (Razor Views) localizes to.

Hence, without setting these values within our `ASPNETCoreL10n` project, the ASP.NET framework wouldn't know which languages the application localizes to.

Get help of UseRequestLocalization

Now we know the importance of placing the supported cultures in our `ASPNETCoreL10n` project. But, simply setting the

cultures we support would not let the application know when to use each of those. To rephrase it, let's say you're reaching the `ASPNETCoreL10n` web app from a French locale; I'm reaching it from an English locale. And, thousands if not millions more are reaching our web app from various locales, at the same time. So, at the moment, can we expect our `ASPNETCoreL10n` application to serve a preferred language to each user? I believe not.

Here comes the need for our project to get the assistance of UseRequestLocalization. This middleware makes sure to automatically change the application's culture, per request.

Let's head over to the `Startup.cs` file within our `ASPNETCoreL10n` and add the following code inside the `Configure()` method:

```
 var supportedCultures = new[] {new CultureInfo("en-US"), new CultureInfo("fr-FR")};  //1
 var requestLocalizationOptions = new RequestLocalizationOptions  //2
 {
     SupportedCultures = supportedCultures,
     SupportedUICultures = supportedCultures
 };
```

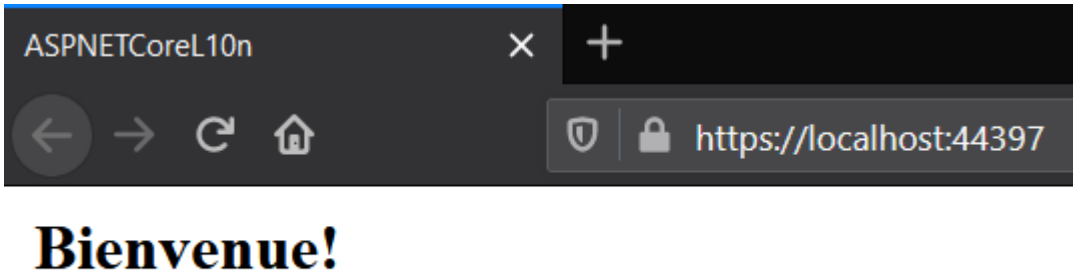
```
 app.UseRequestLocalization(requestLocalizationOption
;  //3
 .
```

1. Create a `supportedCultures` variable holding a list of two `CultureInfo` objects indicating `en-US` and `fr-FR` as supported cultures.
2. Make a RequestLocalizationOptions object mentioning both `SupportedCultures` and `SupportedUICultures` for our application. The `supportedCultures` variable we created in step 1 is passed as values for both `SupportedCultures` and `SupportedUICultures`.
3. Add the `UseRequestLocalization` middleware to the `ASPNETCoreL10n` project's request processing pipeline.

Important Note: *Make sure to place UseRequestLocalization middleware before all other middleware in the request processing pipeline. This is just to make sure any middleware that could require the request's localized culture has it already set by the time the pipeline reaches it.*

Test it out

Those are all the changes `RequestManager` localization asks for. Let's run our `ASPNETCoreL10n` application and see how it works. Now, we'll be able to notice the welcome message swiftly localizes between English and French languages as we expected:



Using IStringLocalizer<T> interface

ASP.NET Core introduced IStringLocalizer to make localization a little bit easier than with `ResourceManager`. Let's take a look at how!

Firstly, let us head over to the `Startup.cs` file within our `ASPNETCoreL10n` project. Now, let's go ahead and add this line within its `ConfigureServices` method:

```
services.AddLocalization();
```

This simple line brings all the services related to localization into our project, together with the `IStringLocalizer` service that we need.

So secondly, let's open the `HomeController` of our `ASPNETCoreL10n` project and add some lines to it as follows:

```
public class HomeController : Controller
{
    .
    private readonly IStringLocalizer _stringLocaliz
//1

    public HomeController(..,

IStringLocalizer<HomeController> stringLocalizer)  /2
    {
        .
        _stringLocalizer = stringLocalizer;
    }

    public IActionResult UsingIStringLocalizer()  //
    {
        ViewData["localized"] =
```

```
_stringLocalizer["localizedUsingIStringLocalizer"].Va
//4

            return View();
        }
    }
```

1. Create a private read-only `_stringLocalizer` field to hold an `IStringLocalizer` instance. Notice that compared to `ResourceManager`, we didn't have to hard-code the basenames and manually inject singletons to the `HomeController`.

2. `stringLocalizer` parameter added to the constructor to let ASP.NET Core framework dependency inject (DI) an `IStringLocalizer<HomeController>` service to it. Passing `HomeController` type to `IStringLocalizer` informed `IStringLocalizer` to specifically browse resources for and provide strings for `HomeController`.

3. Create a new action method `UsingIStringLocalizer` inside the `HomeController`.

4. `_stringLocalizer` asked to retrieve the localized resource value holding a `localizedUsingIStringLocalizer`

key. Afterward, the retrieved resource value is passed over to a `ViewData` with a `localized` key.

Thirdly, let's open up the `HomeController.en-US.resx` resource file inside the `Controller` folder and add a `localizedUsingIStringLocalizer` resource to it:

```
 Name:  localizedUsingIStringLocalizer
 Value: This sentence was localized using
IStringLocalizer.
```

Let's not forget our French resource! So similarly, open up the `HomeController.fr-FR.resx` file and put the following resource inside it:

```
 Name:  localizedUsingIStringLocalizer
 Value: Cette phrase a été localisée à l'aide
d'IStringLocalizer.
```

Finally, let's create a view to respond to the template request from our `UsingIStringLocalizer` action. For this, I believe we

should create a `UsingIStringLocalizer.cshtml` Razor View file inside our project's `Views/Home/` directory, and fill it like this:

```
<html xmlns="http://www.w3.org/1999/xhtml">
<head>
    <title>UsingIStringLocalizer</title>
</head>
<body>
    <h1>@ViewData["localized"]</h1>
</body>
</html>
```

As mentioned on the highlighted line, our `UsingIStringLocalizer.cshtml` View will retrieve a `ViewData` key of `localized` and display it inside an `<h1>` header.

Let's see how it shows

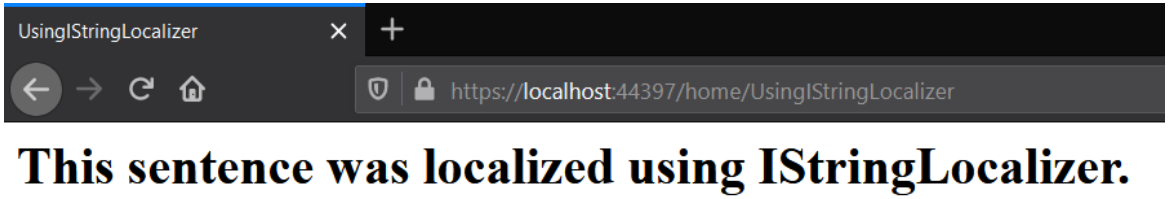
Time to run our `ASPNETCoreL10n` application and head over to the URL endpoint that matches with our `UsingIStringLocalizer` action:

```
 https://localhost:
<port>/home/UsingIStringLocalizer
```

Note: *Make sure to replace* ***<port>*** *with the port number your local webserver runs on.*

If all goes well the browser should show as follows for each browser locale:

In en-US locale



In fr-FR locale

Using IHtmlLocalizer<T> interface

Imagine for a second our `ASPNETCoreL10n` web application hosted HTML lessons for students. And we thought of internationalizing our web app to attract more students to our course. So, we got our `ASPNETCoreL10n` application localized to multiple locales using localized resource files (`.resx`) holding HTML lines and examples. Unsurprisingly loads of students from all over the world signed up for the class. But, on the very first day, pretty much all of them complained saying our `ASPNETCoreL10n` app's content only showed a bunch of HTML tags. They couldn't see what those tags actually *did. What's going on here?*

What's happening?

This scenario happens because `IStringLocalizer` isn't *HTML-aware*. `IStringLocalizer` sees all resources it accesses–even HTML resources–as *strings* and hence, lets them get HTML encoded. So if the resource held HTML elements, they would not get processed.

IHtmlLocalizer was introduced to overcome this. Let's see how we can use it on our `ASPNETCoreL10n` project.

Firstly, let us open the `Startup.cs` file within our `ASPNETCoreL10n` project. Here, let's chain the AddViewLocalization service right with the previously set call to put the `AddRazorPages` service to the DI container:

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

Note: *`AddViewLocalization` view introduces MVC View localization-related services to the project including the IHtmlLocalizer service.*

Secondly, let's open the `ASPNETCoreL10n/HomeController` file and add some code like this:

```
public class HomeController : Controller
{
    .
    private readonly IHtmlLocalizer
_htmlLocalizer;  //1
    public HomeController(..,

IHtmlLocalizer<HomeController> htmlLocalizer)  //2
    {
```

```
            .
            _htmlLocalizer = htmlLocalizer;
        }

        public IActionResult UsingIHtmlLocalizer()
//3
        {
            ViewData["localizedPreservingHtml"] =
_htmlLocalizer["notHtmlEncoded"];  //4

            return View();  //5
        }
    }
```

1. Create a private read-only `_htmlLocalizer` field to hold an `IHtmlLocalizer` instance.
2. `htmlLocalizer` parameter added to the constructor to let ASP.NET Core framework dependency inject (DI) an `IHtmlLocalizer<HomeController>` service to it.
3. Create a new action method `UsingIHtmlLocalizer` inside `HomeController`.
4. `_htmlLocalizer` asked to retrieve the HTML-aware localized resource value holding a “notHtmlEncoded” key as a non-encoded value. Afterward, the retrieved

resource value is passed over to a `ViewData` with a "localizedPreservingHtml" key.

5. `UsingIHtmlLocalizer` action method asks a View on the default route to render a response View passing the `ViewData` along with it. Note that this time, the resource value grabbed by the View would consist of its non-encoded HTML properties.

Thirdly, let us add a record inside our resource files for the "notHtmlEncoded" key.

Let's open up the `Controller/HomeController.en-US.resx` resource file and add the following resource to it:

```
 Name:  notHtmlEncoded
 Value: <b>This resource value was not HTML
encoded.</b>
```

Same way, let's add the localized value inside the `HomeController.fr-FR.resx` file:

```
 Name:  notHtmlEncoded
```

```
 Value: <b>Cette valeur de ressource n'a pas été
codée en HTML.</b>
```

Fourthly and finally, let's make a View to grab the template request from `UsingIHtmlLocalizer` action. Let's create a `UsingIHtmlLocalizer.cshtml` Razor View file inside our project's `Views/Home/` directory, and fill it as follows:

```
<html xmlns="http://www.w3.org/1999/xhtml">
<head>
    <title>UsingIHtmlLocalizer</title>
</head>
<body>
    <p>@ViewData["localizedPreservingHtml"]</p>
</body>
</html>
```

As we can see on the highlighted line, our `UsingIHtmlLocalizer.cshtml` View will retrieve a `ViewData` key of "localizedPreservingHtml" and place it inside an `<p>` tag.

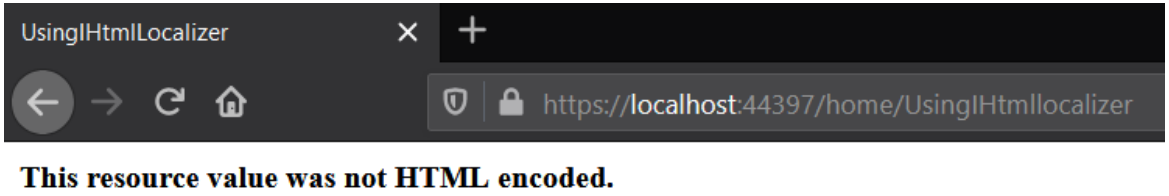
Let's run it and see

Let us run our `ASPNETCoreL10n` application and let the browser point to the URL endpoint that matches with our `UsingIHtmlLocalizer` action:

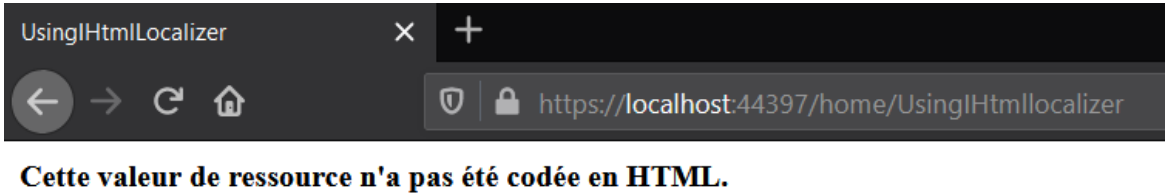
```
https://localhost:<port>/home/UsingIHtmlLocalizer
```

Now, we should be able to observe the browser showing localized paragraph values properly **bolded** as we set in our "notHtmlEncoded" resource value:

In en-US locale



In fr-FR locale



Localizing Views

Alright, hence we know how to localize our ASP.NET Core MVC project passing localized resource values from the controller to the view. But, what if a somewhat *stubborn* view thought...

I don't want to rely on the Controller to send me the localized resources. I'll fetch them myself!

In fact, ASP.NET Core makes this possible making our localization jobs a little bit easier. So, let us find out how we can localize content inside an ASP.NET Core app's MVC view itself.

To begin with, we have already added the AddViewLocalization service to our `ASPNETCoreL10n` project in the previous section using `IHtmlLocalizer`. So, for this section, we'll be using another service within `AddViewLocalization`, namely the IViewLocalizer service.

First of all, let us add a simple `UsingIViewLocalizer` action method inside our `ASPNETCoreL10n` project's `HomeController`:

```
public IActionResult UsingIViewLocalizer()
{
    return View();
}
```

Observe this time we had neither a dependency injection of a service to `HomeController` nor a data passing from `HomeController` over to the view.

Secondly, let's put a "localizedUsingIViewLocalizer" record inside our resource files for a View to receive it later on.

Let's navigate to our `ASPNETCoreL10n` project's `Views/Home` directory. Let us create a `UsingIViewLocalizer.en-US.resx` resource file inside it and fill it with a resource having a key of "localizedUsingIViewLocalizer":

```
 Name:  localizedUsingIViewLocalizer
 Value: This sentence was localized using
IViewLocalizer.
```

Similarly, let us add the "localizedUsingIViewLocalizer" key's `fr-FR` localized value inside a new `UsingIViewLocalizer.fr-`

`FR.resx` resource file:

```
 Name:  localizedUsingIViewLocalizer
 Value: Cette phrase a été localisée à l'aide de
IViewLocalizer.
```

As the last step, it's time to create a Razor view to do both localized resource retrieval and displaying them. Let's create a `UsingIViewLocalizer.cshtml` Razor View file inside our project's `Views/Home/` directory, and fill it like this:

```
 @using Microsoft.AspNetCore.Mvc.Localization  //1
 @inject IViewLocalizer ViewLocalizer  //2

 <html xmlns="http://www.w3.org/1999/xhtml">
     <head>
         <title>UsingIViewLocalizer</title>
     </head>
     <body>

<h1>@ViewLocalizer["localizedUsingIViewLocalizer"]</h
//3
     </body>
 </html>
```

1. @using Razor syntax used to import Microsoft.AspNetCore.Mvc.Localization namespace to `UsingIViewLocalizer.cshtml` Razor View.
2. @inject Razor syntax used to inject IViewLocalizer service from DI service container into a `ViewLocalizer` variable in `UsingIViewLocalizer.cshtml` Razor View.
3. `IViewLocalizer` service used to retrieve localized resource with a key “localizedUsingIViewLocalizer”. The retrieved value is set inside an H1 header tag.

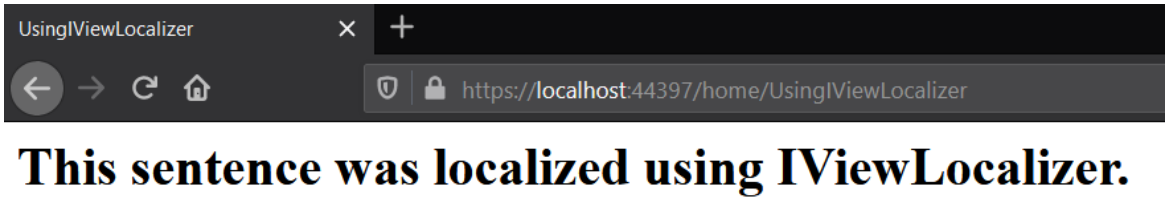
Time to run it

Let’s run our `ASPNETCoreL10n` application and let the browser point to the URL endpoint that matches with our `UsingIViewLocalizer` action:

```
https://localhost:<port>/home/UsingIViewLocalizer
```

If all went well, we would be able to see our browsers showing values localized plainly using Razor Views:

In en-US locale



In fr-FR locale

ASP.NET Core localized resource sharing

At times our ASP.NET Core web app might encounter repetitive resources that we still require to be displayed in a localized manner. Maybe it's a welcome message displayed on each view? Or an "Ok" button label displayed on each

and every view? Let us discover how we can manage such a situation in an ASP.NET Core web application.

Firstly, we need a dummy class at the root of our application which would simply act as an *anchor* to associate our shared resources with.

Let's head on over to our `ASPNETCoreL10n` project and create an empty `SharedResource` class at the root of it.

Since our `SharedResource` class is at the root of the project, it would be accessible by any controller or view within the project. Hence, we would be able to declare an `IStringLocalizer<SharedResource>` instance anywhere in the project and access its resources.

Note: *Make sure the namespace of `SharedResource` class is equal to the assembly name of the project, which is "ASPNETCoreL10n".*

Secondly, let's create a `SharedResource.en-US.resx` resource file in the root of our `ASPNETCoreL10n` project and feed it with a

simple resource:

```
 Name:  localizedUsingSharedResources
 Value: This localization is shared across all
Controllers and Views.
```

In the same way, let's add a `SharedResource.fr-FR.resx` in the project root and put the relevant `fr-FR` localization value inside it:

```
 Name:  localizedUsingSharedResources
 Value: Cette localisation est partagée entre tous
les Controllers et Views.
```

Thirdly, let's visit the `HomeController` in our `ASPNETCoreL10n` project and add some code as follows:

```
 public class HomeController : Controller
 {
     .
     private readonly IStringLocalizer
_sharedStringLocalizer;  //1
     public HomeController(..,
```

```
IStringLocalizer<SharedResource> sharedStringLocalize
//2
     {
          .
          _sharedStringLocalizer = sharedStringLocaliz
     }
     public IActionResult UsingSharedResource()  //3
     {
          ViewData["sharedResourceSentFromController"]
_sharedStringLocalizer["localizedUsingSharedResources
//4

          return View();  //5
     }
 }
```

1. Create a private read-only `_sharedStringLocalizer` field to hold an `IStringLocalizer` instance.
2. `sharedStringLocalizer` parameter added to the constructor to let ASP.NET Core framework dependency inject (DI) an `IStringLocalizer<SharedResource>` service to it.

3. Create a new action method `UsingSharedResource` inside `HomeController`.

4. `_sharedStringLocalizer` asked to retrieve shared resource key “localizedUsingSharedResources”. Afterward, the retrieved resource value is passed over to a `ViewData` with a “sharedResourceSentFromController” key.

Fourthly, let’s create a `UsingSharedResource.cshtml` Razor View file inside our project’s `Views/Home/` directory, and fill it accordingly:

```
 @using Microsoft.Extensions.Localization  //1
 @inject
IStringLocalizer<ASPNETCoreL10n.SharedResource>
SharedLocalizer  //2

 <html xmlns="http://www.w3.org/1999/xhtml">
 <head>
     <title>UsingSharedResource</title>
 </head>
 <body>
     <p>Shared resource sent from Controller:
@ViewData["sharedResourceSentFromController"]</p>
//3
```

```
    <p>Shared resource received by this View:
@SharedLocalizer["localizedUsingSharedResources"]</p>
//4
 </body>
 </html>
```

1. @using Razor syntax used to import Microsoft.Extensions.Localization namespace to `UsingSharedResource.cshtml` Razor View.
2. @inject Razor syntax used to inject IStringLocalizer service from DI service container into a `SharedLocalizer` variable in `UsingSharedResource.cshtml` Razor View.
3. HTML paragraph element containing "sharedResourceSentFromController" `ViewData` resource value retrieved from the Controller.
4. HTML paragraph element containing "localizedUsingSharedResources" shared resource value directly retrieved by `UsingSharedResource.cshtml` Razor View.

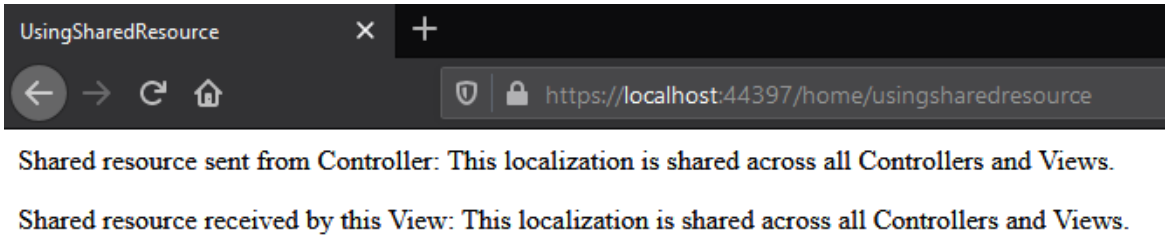
Let us run it

Let's run our `ASPNETCoreL10n` application and let the browser point to the URL endpoint that matches with our `UsingSharedResource` action:

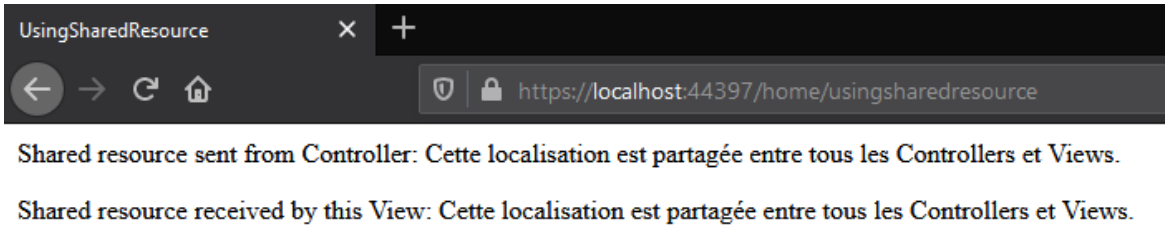
```
https://localhost:<port>/home/UsingSharedResource
```

We should now be able to see the browser showing shared resources obtained from both `HomeController` and `UsingSharedResource.cshtml`:

In en-US locale



In fr-FR locale



Identify the user's culture

Okay, we talked about ways to localize our ASP.NET Core web application to multiple cultures, We learned the *HOW*. But, in what manner would our app know which culture to localize to? So, time to learn how ASP.NET Core web apps decide their localization language, let's learn the *WHAT*.

Say hello to RequestLocalizationOptions

Remember when we previously got assistance from `UseRequestLocalization` middleware to switch app localization per each request? This time, we'll explore RequestLocalizationOptions which is initialized by `UseRequestLocalization`.

`RequestLocalizationOptions` holds an IRequestCultureProvider list which provides `UseRequestLocalization` a list of options when discovering the locale. ASP.NET Core conveniently ships with the following implementations of `IRequestCultureProvider` to help us with our localization duties:

- QueryStringRequestCultureProvider
- AcceptLanguageHeaderRequestCultureProvider
- CookieRequestCultureProvider

Let's have a look at how each of these works, shall we?

Using QueryStringRequestCultureProvider

As the name hints, QueryStringRequestCultureProvider helps the ASP.NET Core web app user specify the culture he or she needs through a "culture" query string. Let's see how!

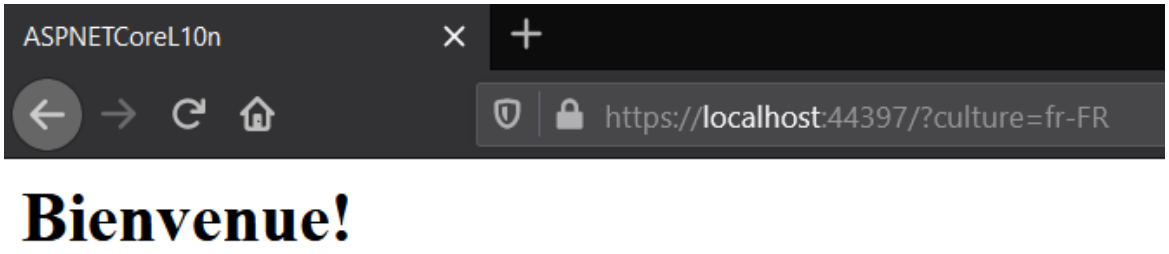
First up, we need to put a `UseRequestLocalization` middleware to our `ASPNETCoreL10n` project, passing in a `RequestLocalizationOptions` parameter. We can safely skip this step because we already placed this middleware inside the

`Startup.cs` file when we were setting up `UseRequestLocalization` middleware earlier.

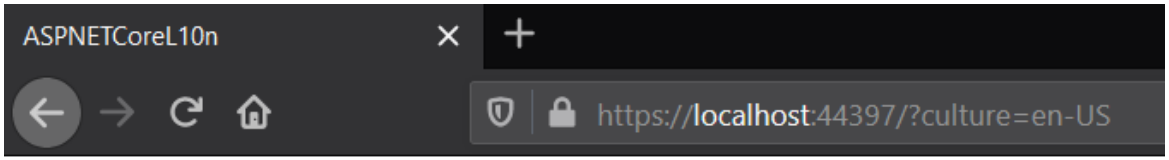
Then, let's simply call our `ASPNETCoreL10n` project's `Index` URL with a "culture" query string with a value of `fr-FR`:

```
https://localhost:<port>/?culture=fr-FR
```

We should be able to see the page swiftly localized to `fr-FR` culture:



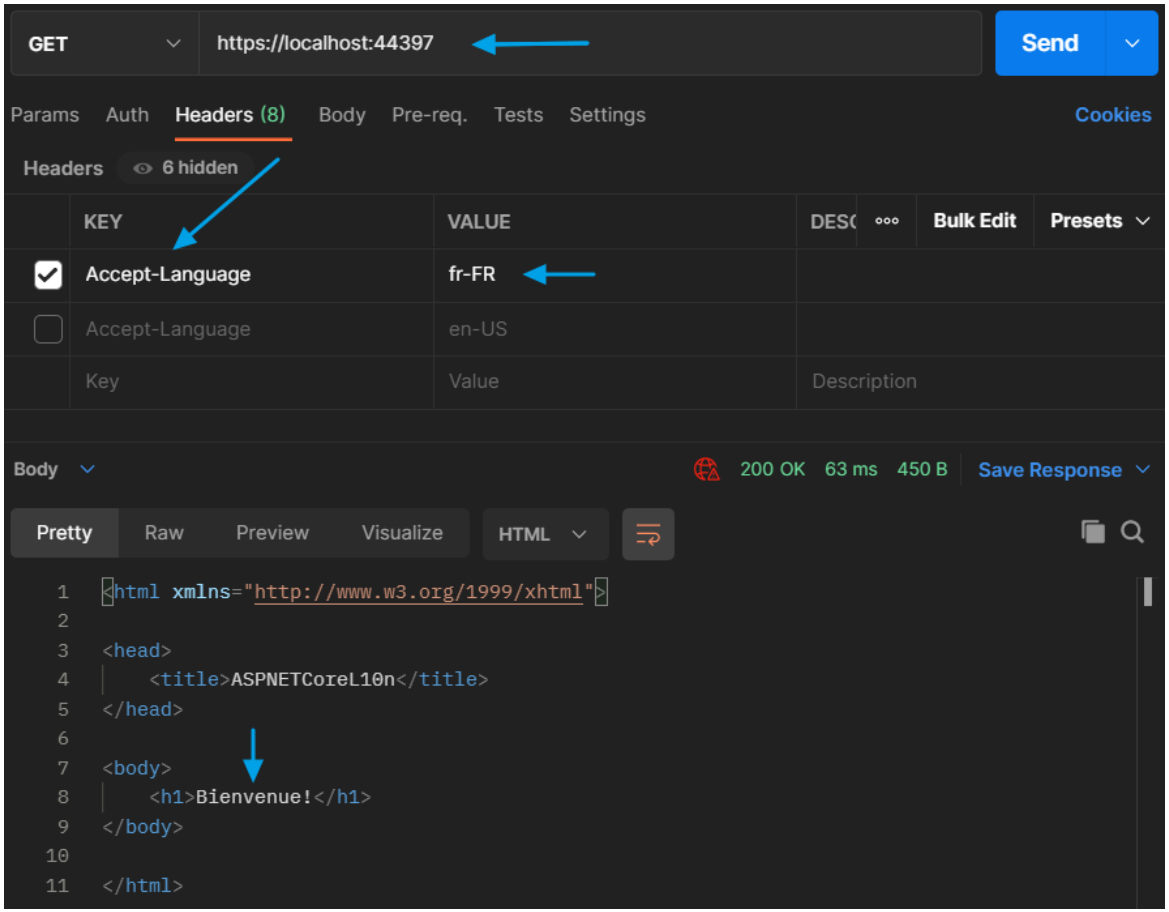
Equally, providing an `en-US` value for the "culture" query string should show the page localized to `en-US` culture:



Using AcceptLanguageHeaderRequestCultureProvider

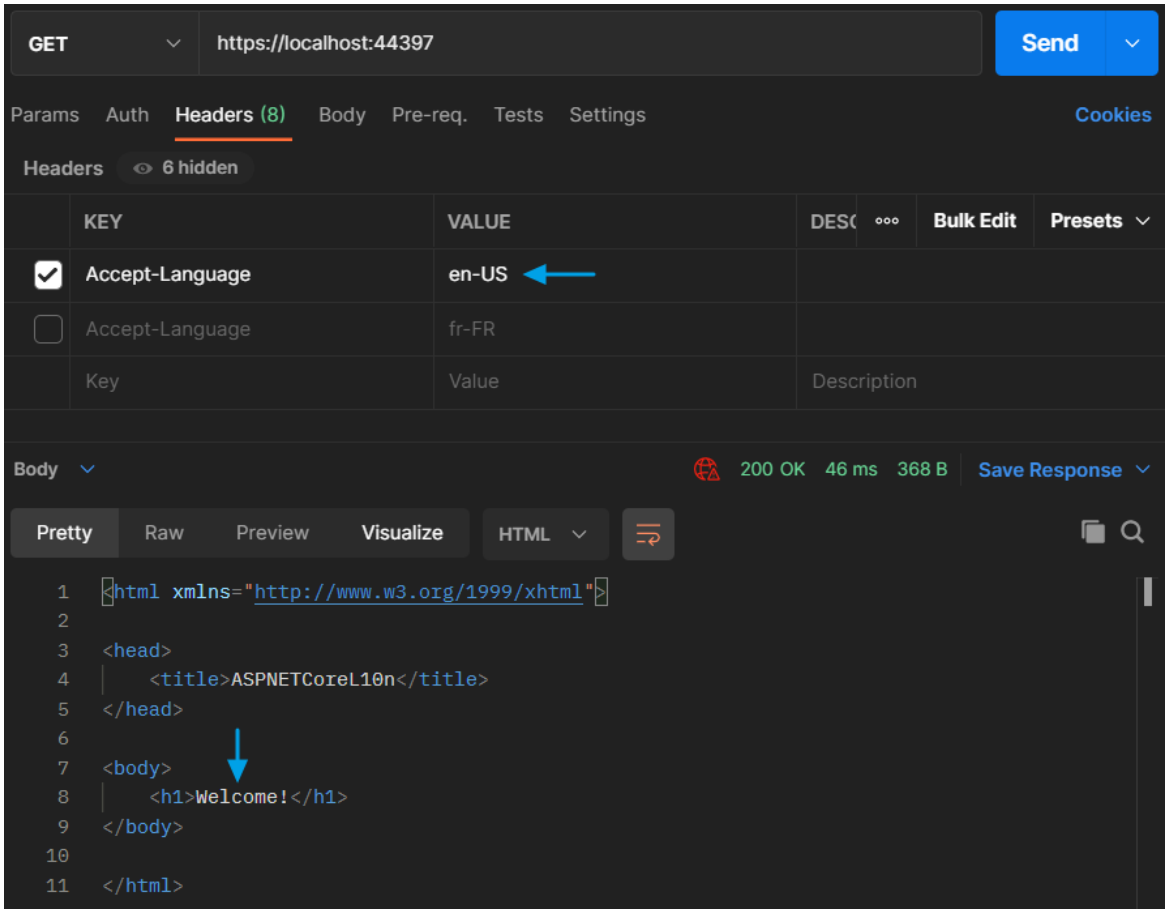
Another way we can make our ASP.NET Core web app perform localization to a certain culture is to provide an `Accept-Language` header.

Let us open up our API client app and call our `ASPNETCoreL10n` project's `Index` URL with an `Accept-Language` header holding a value of "fr-FR":



As we can see from the result, calling `localhost`–this time with no query strings added–has still given us a properly localized page.

Similarly, calling `Index` URL with an `Accept-Language` value set to “en-US” should give us a page localized to the relevant culture:



Using CookieRequestCultureProvider

Both the `RequestCultureProvider` instances we talked about earlier rely on the user to provide the culture each time a request is made to the server. Hence, the lifetime of each localization is just for that particular request. Let's see how

to overcome this limitation with a good old cookie provided to the user!

Let's head over to the `HomeController` on our `ASPNETCoreL10n` project and add a new action to it as follows:

```
 [Route("Home/UsingCookieRequestCultureProvider/{cult
]  //1
 public string UsingCookieRequestCultureProvider(stri
culture)  //2
 {
     Response.Cookies.Append(  //3
         CookieRequestCultureProvider.DefaultCookieNa
//4
         CookieRequestCultureProvider.MakeCookieValue
RequestCulture(culture)),  //5
         new CookieOptions { Expires =
DateTimeOffset.UtcNow.AddYears(1) }  //6
     );

     return "Cookie updated to this culture: " + cult
//7
 }
```

1. Attribute routing that accepts a "culture" parameter.

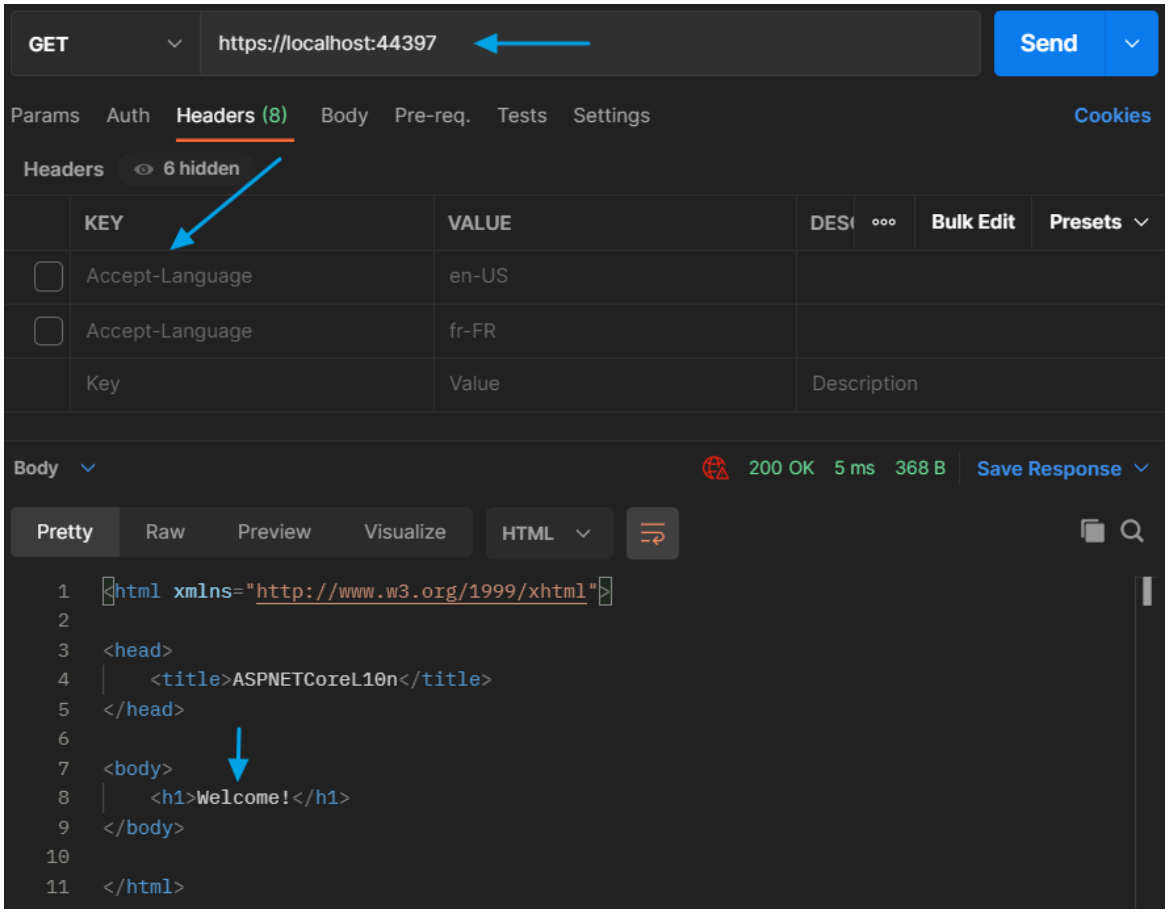
2. Create a new action method `UsingCookieRequestCultureProvider` with a string parameter “culture”. This parameter will hold the culture our app user prefers.
3. Append a cookie to the HttpResponse.
4. Default cookie name for culture “.AspNetCore.Culture” given as the cookie name.
5. MakeCookieValue is used to create a cookie passing in a RequestCulture object. This `RequestCulture` holds the culture requested by the user.
6. CookieOptions setting for the cookie to expire in one year.
7. String value returned from the action noting its successful execution.

Test it out

That’s all! let’s see if it works.

Before we start testing `CookieRequestCultureProvider`, let’s run our `ASPNETCoreL10n` project and open its `Index` URL in our `ASPNETCoreL10n` project with no query strings or headers

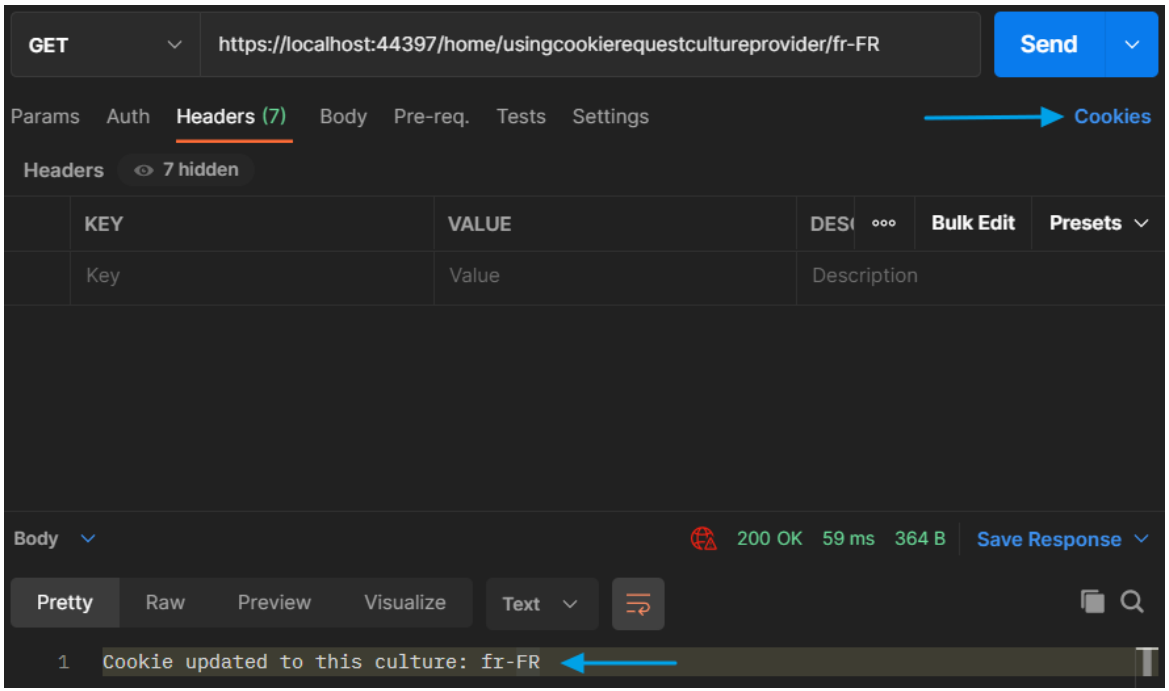
added. This should give us a simple welcome message in `en-US` culture:



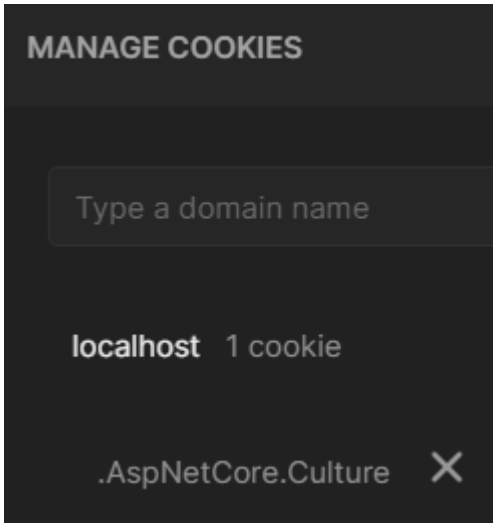
Now, let us open up our API client app and call our `CookieRequestCultureProvider` URL passing in an `fr-FR` culture as its parameter:

```
https://localhost:
<port>/home/usingcookierequestcultureprovider/fr-FR
```

As we can see, it gives us a 200 OK response informing us the cookie was updated to `fr-FR` culture:

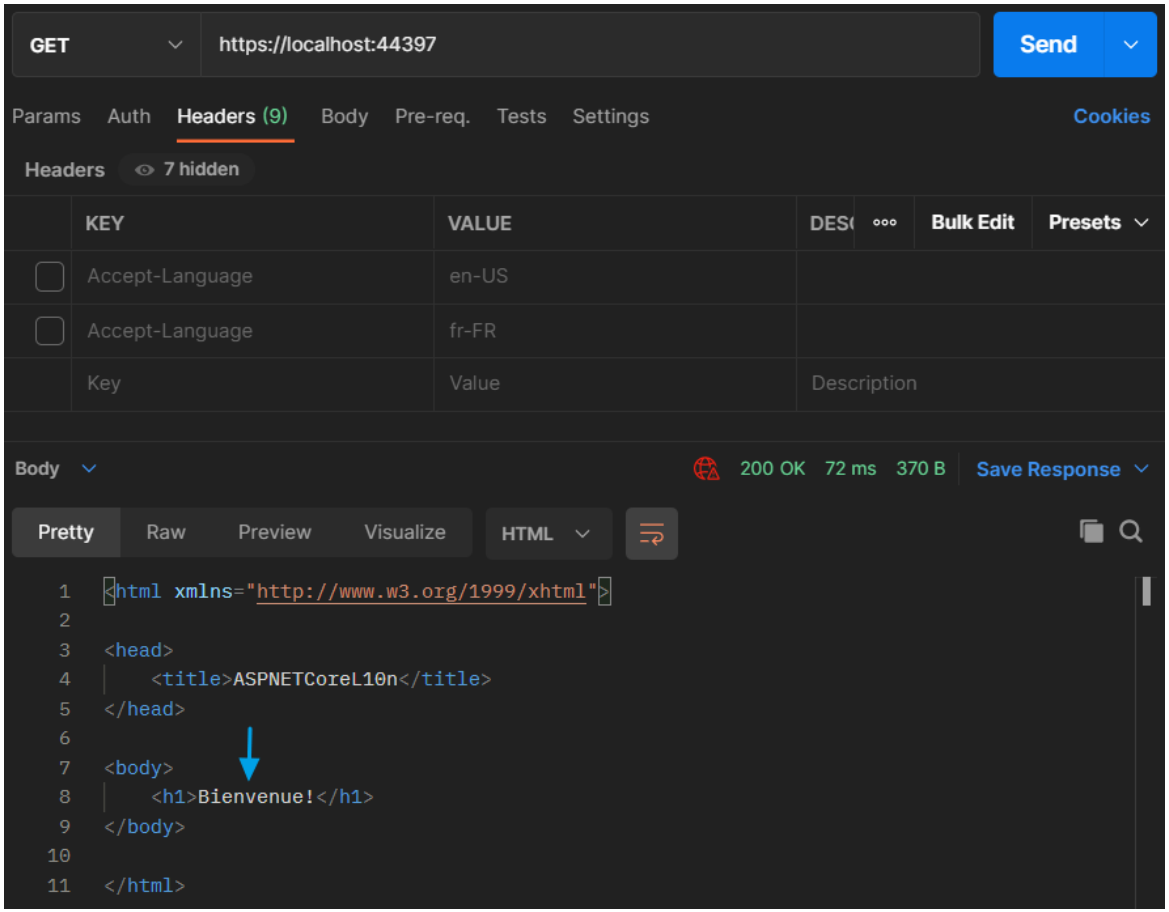


Additionally, we can check cookies on our API client to see that the cookie has been added:



But, let's not take the word of that HTTP response! Let us verify if the culture actually has been updated.

Simply head over back to the `Index` URL in our `ASPNETCoreL10n` project:



We'll be able to notice the welcome message is now shown on an `fr-FR` culture. Even if we restart our API client the result would be the same, until the cookie is expired or manually deleted!

Little note on RequestCultureProvider enumeration

Once the `RequestCultureProvider` middleware retrieves the `RequestCultureProvider` list from `RequestLocalizationOptions`, `RequestCultureProvider` sequentially enumerates the list until one provider successfully determines the request culture. If none of them could determine the culture, the default culture will be used.

Using CustomRequestCultureProvider

We talked about the default methods provided by ASP.NET Core as implementations of `IRequestCultureProvider`. Instead, ASP.NET Core also allows us to use a CustomRequestCultureProvider where we can code our own logic to determine the culture of our application. Let's make a sample custom implementation taking use of `CustomRequestCultureProvider`:

Let's open up the `Startup.cs` file in our `ASPNETCoreL10n` project, and add the following snippet inside the `Configure` method:

```
 requestLocalizationOptions.AddInitialRequestCultureP
w CustomRequestCultureProvider(async context =>  //1
 {
     var currentCulture = "en-US";  //2
```

```
        var segments = context.Request.Path.Value.Split(
{ '/' },
            StringSplitOptions.RemoveEmptyEntries);

        if (segments.Length > 0)
        {
            var lastElement = segments[segments.Length -
            if (lastElement.Length == 2 || lastElement.L
5)
            {
                currentCulture = lastElement;
            }
        }

        return new ProviderCultureResult(currentCulture)
    }));
```

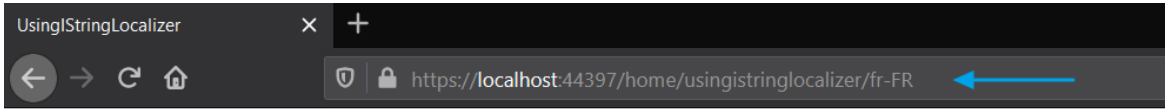
1. Use AddInitialRequestCultureProvider method to add our custom `IRequestCultureProvider` implementation as the first option on the `RequestCultureProvider` list passed over by the `RequestLocalizationOptions`. Hence, our custom request culture provider will be the first one used by `UseRequestLocalization` middleware in its `RequestCultureProvider` enumeration.

2. Simple custom logic to decide the request culture based on the last element on the request URL.

3. Return ProviderCultureResult containing the culture determined by our custom request culture provider.

Important Note: *make sure to place this* `requestLocalizationOptions.AddInitialRequestCultureProvider` *call before you call* `app.UseRequestLocalization`.

Let us test it out by calling one of our `ASPNETCoreL10n` project's URLs postfixed with a culture:



Cette phrase a été localisée à l'aide d'IStringLocalizer.

Setting defaults

Imagine for a moment our `ASPNETCoreL10n` web application was a music streaming service. It planned to reach all across the world but only users from `en-US` locale seemed to be able

to sign up for the service. Neither Gimhani from Sri Lanka nor Min-ho from South Korea could register for our service– while both of them being well capable of understanding the English language in the `en-US` locale. *What's going on?*

This happened because our app was localized solely to the `en-US` culture. Localization is all fun and games until our user sees a `404 - Page not found` page simply because he or she's from a locale our app isn't currently localized to. This is why we should always set default languages and resource values for our ASP.NET Core web application when implementing localization. This way our app will always have a culture to fall back to. Let's find out how we can do this!

StringLocalizer behavior for missing resources

`StringLocalizer` brings a useful solution for dealing with missing resources. If `StringLocalizer` couldn't find a resource value to a key we provided, it would simply return the resource key we provided as its result. Let's see how this works, shall we?

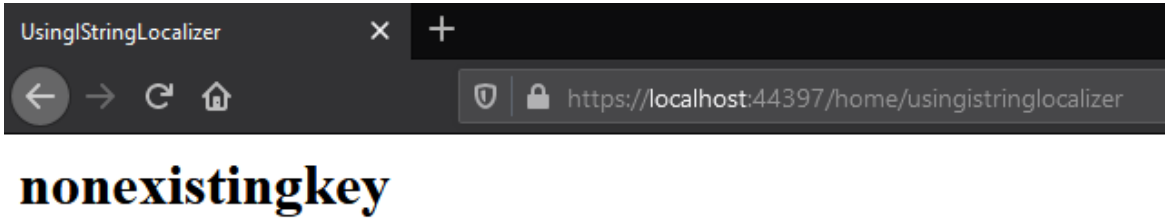
Let's head back over to the `UsingIStringLocalizer` action we created in the `HomeController` of our `ASPNETCoreL10n` project.

This time, let us change the action to ask the `StringLocalizer` to retrieve a key that does not exist on the project resources:

```
public IActionResult UsingIStringLocalizer()
{
    ViewData["localized"] =
_stringLocalizer["nonexistingkey"].Value;

    return View();
}
```

Running our `ASPNETCoreL10n` application and calling `UsingIStringLocalizer` action on `https://localhost:<port>/home/UsingIStringLocalizer` should provide us back the key we provided:



Set default culture for smoother ASP.NET Core localization

When setting the `RequestLocalizationOptions` passed over to the `UseRequestLocalization` middleware, `RequestLocalizationOptions` allows you to pass a DefaultRequestCulture property.

Let's head over to our `ASPNETCoreL10n` project's `Startup.cs` file and place this line inside its `Configure` method:

```
 requestLocalizationOptions.DefaultRequestCulture =
new RequestCulture("en-US");
```

Now, if our ASP.NET Core web app user doesn't specify a culture he/she prefers, `en-US` will be used. Equally, if another person from a culture our web app currently doesn't support happens to access our web application, he or she will be offered our app in `en-US` culture.

Congratulations! we completed learning the essentials on ASP.NET Core localization. Henceforward you'd know how to

localize an ASP.NET Core application to any and all the languages you fancy.

Some ASP.NET Core localization extras

Let's take a look at a few more extra features you're pretty sure to stumble upon on your ASP.NET Core localization journey.

ASP.NET Core date and time format localization

Setting SupportedCultures inside your project makes our project automatically display its dates and times formatted for the current localization.

Firstly, let us head over to our `ASPNETCoreL10n` project's `Startup.cs` file and make sure `SupportedCultures` have already been added to `RequestLocalizationOptions`:

```
var supportedCultures = new[] {new
```

```
CultureInfo("en-US"), new CultureInfo("fr-FR")};
//1
 var requestLocalizationOptions = new
RequestLocalizationOptions
 {
     SupportedCultures = supportedCultures,  //2
     .
 };
```

Note: *If you followed through the ResourceManager-related l10n section in this tutorial, you must already have this set up in your application.*

1. A list containing `en-US`, `fr-FR` cultures added to `supportedCultures` variable.
2. `supportedCultures` passed over to `RequestLocalizationOptions`.

Secondly, open the project's `HomeController` and add a new action as follows:

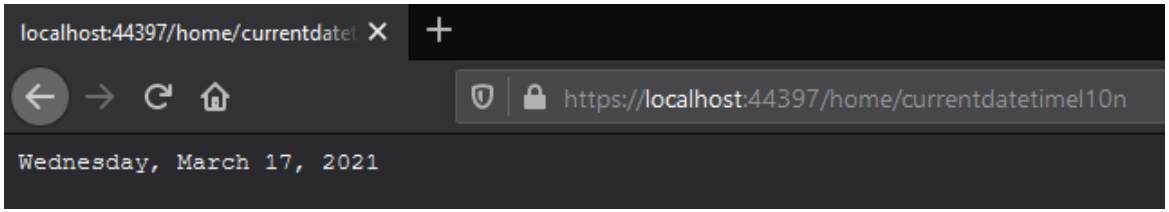
```
public string CurrentDateTimeL10n()  //1
{
    return DateTime.Now.ToLongDateString();  //2
```

```
}
```

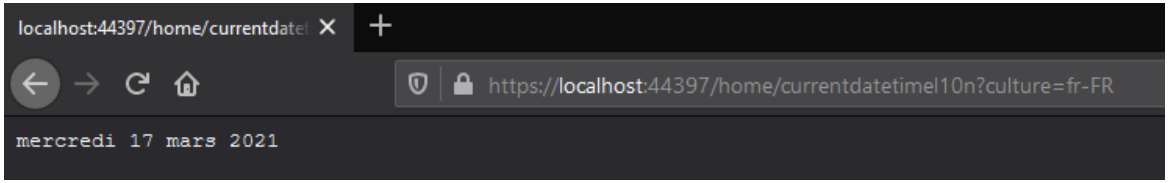
1. Create new `CurrentDateTimeL10n` action method.
2. Return current date and time.

Running our app for each localization should show the appropriate date and time value localized to the current culture:

In en-US locale



In fr-FR locale



Placeholder usage in ASP.NET Core localization

There can be times our ASP.NET Core application needs to receive a parameter from the user and display it inside of a localized message. Let's see how we can do this using `IStringLocalizer`.

Firstly, let us add a new resource inside our `Controllers/HomeController.en-US.resx` file:

```
Name:  welcomeWithName
Value: Welcome {0}!
```

Add the `fr-FR` inside `HomeController.fr-FR.resx` file as well:

```
Name:  welcomeWithName
Value: Bienvenue {0}!
```

As you can see, we added a `welcomeWithName` resource which has a placeholder within its value.

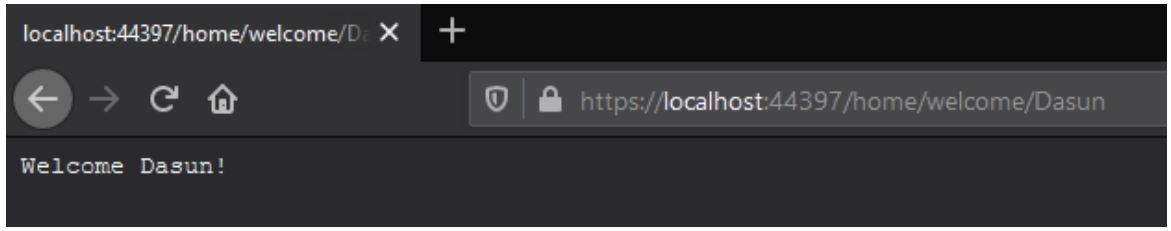
Secondly, let us open the `HomeController` file inside our`ASPNETCoreL10n` project and add a new action method as follows:

```
 [Route("Home/Welcome/{name}")]  //1
 public string Welcome(string name)  //2
 {
     return _stringLocalizer["welcomeWithName",
name];  //3
 }
```

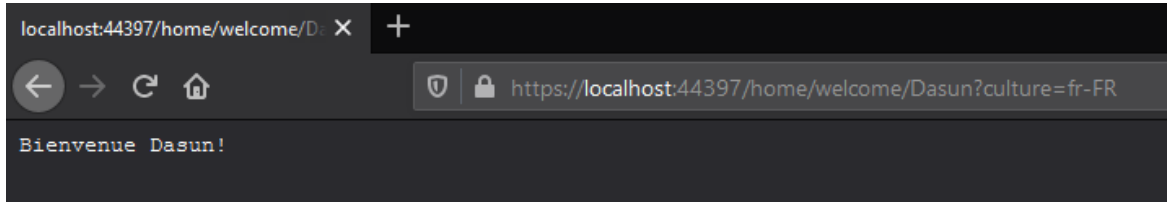
1. Attribute routing that accepts a “home” parameter.
2. Create a new action method `UsingCookieRequestCultureProvider` with a string parameter “name”. This parameter will hold the name our app user provides as a parameter.
3. Ask `_stringLocalizer` to find a resource with a key “welcomeWithName”. The `name` parameter acquired from the request is passed over as the 2nd index.

Let’s run it and see if the placeholders have been correctly set on the localized welcome messages:

In en-US locale



In fr-FR locale



Let Lokalise do the localizing

Wonder if you read my article from its start to the finish. If you're working from home, you might have missed out on loads of chores. If you're at your office, you could have postponed a good deal of tasks squeezing in time allocations to internationalize your ASP.NET application.

What if I told you there's a much clearer, 1000x faster, and a million times favorable way to handle all the ins and outs of your ASP.NET project's localization and internationalization?

Meet Lokalise, the translation management system that takes care of all your ASP.NET application internationalization needs. With features like:

- Easy integration with various other services
- Collaborative translations
- Quality assurance tools for translations
- Easy management of your translations through a central dashboard
- Plus, loads of others

Lokalise will make your life a whole lot easier by letting you expand your ASP.NET Core app to all the locales you'll ever plan to reach.

Start with Lokalise in just a few steps:

- Sign up for a free trial (no credit card information required).
- Log in to your account.
- Create a new project under any name you like.

- Upload your translation files and edit them as required.

That's all it takes! You have already completed the baby steps toward Lokalise*-ing* your ASP.NET Core application. See the Getting Started section for a collection of articles that will provide all the help you'll need to kick-start the Lokalise journey. Also, refer to Lokalise API Documentation for a complete list of REST commands you can call on your Lokalise translation project.

Conclusion

In this tutorial, we explored how we can localize an ASP.NET Core application to multiple locales. We examined how to add language resources to our app and organize them. Further, we looked at localizing with the help of .NET `ResourceManager`, through various other ASP.NET Core localization-related classes like `IStringLocalizer`, `IHtmlLocalizer`, and `IViewLocalizer`. We also checked out on setting up `UseRequestLocalization` middleware. We found out how we could place repetitive resources in a common resource file and reviewed ways to Identify the user's culture

using various implementations of `IRequestCultureProvider`. Finally, we wrapped the main section of our article inspecting how we can set default values in the project.

Additionally, we looked at how date and time format localization takes place in an ASP.NET Core application and addressed placeholder usage.

So, with that, it's time for me to wrap up. Till we meet again, have a great day with lesser bugs and even lesser viruses, in your workstations and your real lives!

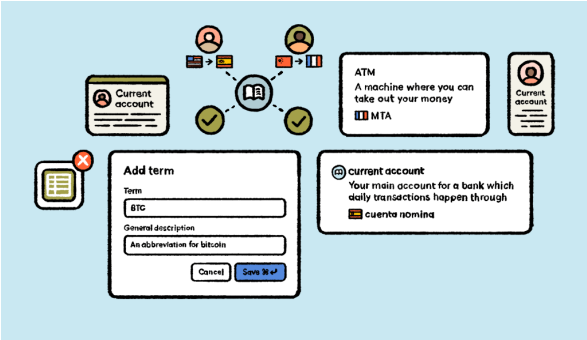
Tutorials

Author

Dasun Nirmitha

Dasun is a Technical Writer-slash-Backend Developer that—after a midlife crisis—realized his quite perfectionist self belongs in the writer's realm. Here's a guy that shifted from 9 to 5 jobs to freelancing full-time, just to follow his passion-the art of coding-the way he always planned to do so.

Read also



Guides · Insights · Localization

Localization and glossaries: everything you need to know

Each industry has its own jargon and terminology. And each company has brand-specific phrases, non-translatable terms, and abbreviations. When you work with translators who are from outside your field, a glossary...

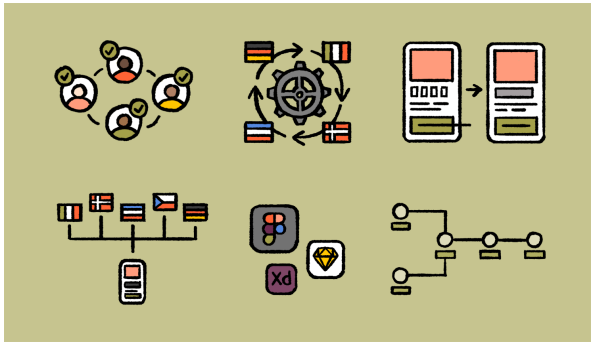
September 17, 2021 · Jess Evans

Inside Lokalise

Meet Gita Timofejeva, Account Manager

Welcome to Humans of Lokalise. We're building a culture here at Lokalise where amazing people (like you) can do their best work. Stay tuned to get to know our team,...

September 16, 2021 · Stefanos Bournias



Guides · Localization

Design-stage localization: The top challenges for localization teams (Part 2)

In Part 1, we covered why design-stage localization is a game-changing solution for agile, multilingual product development. If you read it, you've likely thought about what it would take to...

September 10, 2021 · Niklas Hisinger

Case Studies

Specialized: Riding towards global success with a 100% teammate enthusiasm

Read more

Localization workflow for your web and mobile apps, games and digital content.