My Experience of localizing of ASP.NET MVC Core application

Table of Contents

Introduction

Being student at online course of ASP.Net Core 3.1 at Udemy online university, I asked the question, how it possible to globalize ASP.NET core application. It was when we finished the CRUD views, so I wanted to localize them.

There are lot of articles about basics of localizations, but I found that there are mostly adapted for small amount of object or resources.

In this article I shall use the example Leave Management, which contains entities of Employee, Leave Type, and leave allocations. And in the part of localization, I will explore only CRUD operations.

What parts of applications needs to be localized.

Parts to localize in Model-View-Controller application

Like says MVC, there are three parts to localize. It is Model, View and Controller. In the case of ASP.NET Core MVC Application Model means ViewModel. The applied object often different of its presentation and can have multiple presentations. One for editing, other for list, and another for detailed view, for example.

What localize in ViewModel

Localization of model means localizing its Display Name, Description, Prompt and Validation messages. All these things, which we can put in Data Annotation.

What localize in controller.

In controller, we can localize the messages about exceptions or advanced model validation issues, which comes mostly for business logic, rather that input format. Often, these issues can be found of server side.

What localize in views

In view, it can be different static texts, except annotations for the ViewModel. They will be localized by Data Annotations of the ViewModel. In the view, we can localize static texts and images.

What localize in all these parts?

Also, one global resource file can be useful. Here is good place to put the localization of common phrases end commands, such “Create New”, “Save”, “Undo”, etc. Things, which repeated from one view to other.

What not covered

For the moment, I not covered the localization of eventual javascript components.

Default rules for placing and naming of resx files and resources naming in application

Thera are two conventions for ranking resources files in the project. It is Dot convention and Folder convention. I compared the both in example:

Full type name	Dot convention	Path convention
`LocalizationWebsite.Controllers.BookController`	`Resources/Controllers.BookController.{Culture}.resx`	`Resources/Controllers/HomeController.{Culture}.resx`

LocalizationWebsite.Models.BookDetailsViewModel	Resources/Models.BookDetails.fr.resx	Resources/Models/BookDetails.fr.resx
And for the view		
View path	Dot convention	Path convention
Views/Home/ShowPerson.cshtml	Resources/Views.Home.ShowPerson{.Culture}.resx	Resources/Views/Home/ShowPerson{.Culture}.resx

In the case, when assembly name is different from Default namespace, folder convention ceases to work for ViewModel and Controller. But still works for the views.

Full type name	Dot convention	Path convention
LocalizationWebsite.Controllers.BookController	Resources/Controllers.BookController.{Culture}.resx	N/A
LocalizationWebsite.Models.BookDetailsViewModel	Resources/Models.BookDetails.fr.resx	N/A
And for the view		
View path	Dot convention	Path convention
Views/Home/ShowPerson.cshtml	Resources/Views.Home.ShowPerson{.Culture}.resx	Resources/Views/Home/ShowPerson{.Culture}.resx

If you vant to know exact resource name in your assembly, you can execute this code in Immediate Window of your Visual Studio. But you need to have the resource file for invariant culture. For example, Resources/ Views.Home.ShowPerson.resx

```
typeof(Any-type-from-yours-assembly).Assembly.GetManifestResourcesNames()
```

I recommend to read the section https://docs.microsoft.com/fr-fr/aspnet/core/fundamentals/localization?view=aspnetcore-3.1#resource-file-naming about this subject.

Requirements to resources files locations and files tree

The problem, that I found for conventional localization approach, is or it generates lot of resource files, which must be maintained and translated in one folder, or one huge global resources file. In my case AssemlyName was different of Default Namespace, which made resource name very large.

Requirements to the resource files of the views.

Files must be located in individual folder, which corresponds to controller and action of this view. For exemple, for the controller LeaveTypesController, which contains CRUD (Create, Read, Update, Delete) actions for Leave Type entity, I expect such tree:

```
Resources
    Views
      LeaveTypes
       Create
         Create.resx
         Create.fr.resx
         Create.ru.resx
       Update
         Update.resx
         Update.fr.resx
         Update.ru.resx
       Delete
         Delete.resx
```

Requirements for resource files for the ViewModel

As was mentioned, one business object can be presented by multiples ViewModels. Most of them will have the same properties. To avoid the redundant translations, better way me seems is the pointing to business object, razer ViewModesl. So, in my example ViewModels have such organization:

```
ViewModels
    LeaveType
      LeaveTypeEditionViewModel.cs
      LeaveTypeDetailsViewModel.cs
      LeaveTypeIndexViewModel.cs
      LeaveAllocation
    LeaveAllocation
      LeaveAllocationEditionViewModel
      .cs
```

and so on. So, for the resources structure I decided to organize them next way.

```
Resources
    ViewModels
      LeaveType
         LeaveType.resx
         LeaveType.fr.resx
         LeaveType.ru.resx
      LeaveAllocation
         LeaveAllocation.resx
         LeaveAllocation.fr.resx
         LeaveAllocation.ru.resx
```

Requirements for resource files for the controllers

For the controller, the approach is the same, as for ViewModel. One resource file for controller. Of course, controller must also have access for global resources file.

```
Resources
    Controllers
      LeaveTypesController
       LeaveTypesController.resx
       LeaveTypesController.fr.resx
       LeaveTypesController.ru.resx
```

About the code section, it looks in similar way.

```
Controllers
    LeaveTypesController.cs
    LeaveAllocationsCopntroller.cs
```

Localizing application

ASP.Net Core provides mechanics to localize all these parts. With the help of IStringLocalizer and IHtmlLocalizer it is possible to accomplish these tasks.

Optional changes in AssemblyInfo.cs – check!

If your application has Defaul namespace, which is different from assembly name, you must change/create file AssemblyInfo.cs and add attribute `[assembly: RootNamespace("YOUR-ROUTE-NAMESPACE")]`

Enabling localization in Startup.cs

To add the possibilities of localization, you must include localization services during start of application. For this, you must customize supported cultures in Startup.cs

For externalizing all actions related to Globalization, which repeats the signatures of methods from the class `Startup`, and incapsulates all settings, related to localization. Then, I call corresponded methods from `GlobalizationStartup` in my `Startup` class.

```
    public static class GlobalizationStartup {

        #region Startup methods
        public static void ConfigureServices(IServiceCollection services) {
            services.AddLocalization(options => {
                options.ResourcesPath = "Resources";
            });
            services.AddScoped< ILeaveManagementCustomLocalizerFactory,
LeaveManagementCustomLocalizerFactory>();
            services.AddMvc()
                .AddViewLocalization(LanguageViewLocationExpanderFormat.SubFolder)
                .AddDataAnnotationsLocalization(options => {
                    options.DataAnnotationLocalizerProvider = (type, factory) => {
                        using (var serviceProvider = services.BuildServiceProvider()) {
                            var localizingService =
serviceProvider.GetRequiredService<ILeaveManagementCustomLocalizerFactory>();
                            return localizingService.CreateStringLocalizer(type);
                        }
                    };
                });
        }

        public static void Configure(IApplicationBuilder app, IWebHostEnvironment env) {
            CultureInfo[] supportedCultures = new CultureInfo[] {
                new CultureInfo("en"),
                new CultureInfo("en-US"),
                new CultureInfo("ru"),
                new CultureInfo("fr")
            };


            IRequestCultureProvider[] cultureProviders = new IRequestCultureProvider[] {
                new QueryStringRequestCultureProvider(),
                new CookieRequestCultureProvider(),
                new AcceptLanguageHeaderRequestCultureProvider()
            };

            app.UseRequestLocalization(new RequestLocalizationOptions {
                DefaultRequestCulture = new RequestCulture("en-US"),
                RequestCultureProviders = cultureProviders.ToList(),
                // Formatting numbers, dates, etc.
                SupportedCultures = supportedCultures,
                // UI strings that we have localized.
                SupportedUICultures = supportedCultures
            }); ;

        }
        #endregion
    }
```

In method Configure,

Configuring localization of ViewModel. DataAnnotations localizations.

By default, to localize your validation messages, descriptions and prompts, you must call `AddDataAnnotationsLocalization` method in your `Startup`s' `ConfigureServices` method

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

services.AddMvc()

.AddViewLocalization(Microsoft.AspNetCore.Mvc.Razor.LanguageViewLocationExpanderFormat.SubFolder)

.AddDataAnnotationsLocalization();

If you want, you can redefine your way of localizing DataAnnotation. In my case, it was retrieving my localization service and using it in DataAnnotationLocalizerProvider

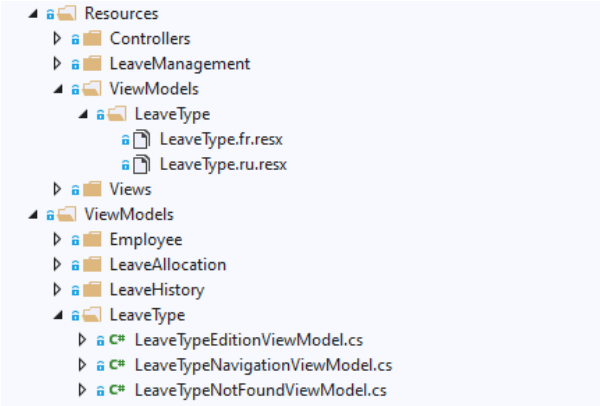
```
            services.AddScoped< ILeaveManagementCustomLocalizerFactory,
LeaveManagementCustomLocalizerFactory>();
            services.AddMvc()
                .AddViewLocalization(LanguageViewLocationExpanderFormat.SubFolder)
                .AddDataAnnotationsLocalization(options => {
                    options.DataAnnotationLocalizerProvider = (type, factory) => {
                        using (var serviceProvider = services.BuildServiceProvider()) {
                            var localizingService =
serviceProvider.GetRequiredService<ILeaveManagementCustomLocalizerFactory>();
                            return localizingService.CreateStringLocalizer(type);
                        }
                    };
                });
```

It creates string localizer for given type of view model, it corresponds to convention. You can add your own convention.

Now, you can tag the properties of your ViewModel class with attributes `Display`, `Required`, `DataType`.

```
        [Required(AllowEmptyStrings = false, ErrorMessage = "Leave Type - name required")]
        [DataType(DataType.Text, ErrorMessage = "Expected value: text")]
        [Display(Name = "LeaveTypeName", Description = "LeaveTypeName_Description", Prompt
= "Insert leave type name, please")]
        public string LeaveTypeName { get; set; }
```

The rules about placing the translation parts were explained before



Configuring localization for Views.

For view localization, both conventions works, even when Default namespace is different from assembly name. So it is possible to use default mechanics by injecting the IViewLocalizer to corresponded view.

In this case it is `IViewLocalizer Localizer.` To share some other translations, which can be used in different parts of application, I also injected my localization factory, which search data in appropriate resources sections.

```
@using LeaveManagement
@using LeaveManagement.Models
@using  LeaveManagement.Code.CustomLocalization;
@using Microsoft.Extensions.Localization;
@using Microsoft.AspNetCore.Mvc.Localization;
@addTagHelper *, Microsoft.AspNetCore.Mvc.TagHelpers

@inject IViewLocalizer Localizer;
@inject ILeaveManagementCustomLocalizerFactory LeaveManagementLocalizerFactory;
```

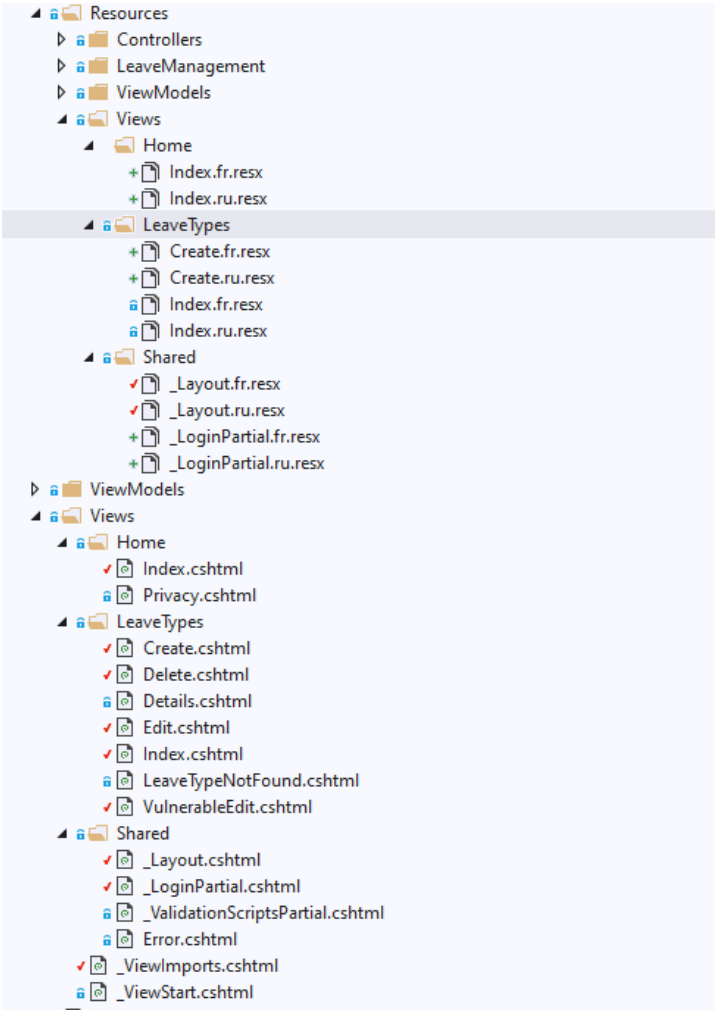
Example of using is:

```
@{
    ViewData["Title"] = Localizer["Index"];
}

<h1>@Localizer["Index"]</h1>

<p>
    <a class="btn btn-primary" asp-
action="Create">@LeaveManagementLocalizerFactory.CommandsLocalizer["Create New"]</a>
</p>
```

And resources ranking:



Configuring localization for Controller

In controller, I use dependency injection of my localization factory in constructor. The goal for the factory is obtain appropriates resources for the type of constructor.

```
private readonly IStringLocalizer _Localizer;

public LeaveTypesController(Contracts.ILeaveTypeRepository repository,
            AutoMapper.IMapper mapper,
            ILogger<LeaveTypesController> logger,
            ILeaveManagementCustomLocalizerFactory localizerFactory)
{
    _Repository = repository;
    _Mapper = mapper;
    _Logger = logger;
    _Localizer = localizerFactory.CreateStringLocalizer(typeof(LeaveTypesController));
}
```

Using is quite simple

```
ModelState.AddModelError("StateInvalid", _Localizer["Model state invalid"]);
                    ViewBag.Message = _Localizer["Model state invalid"];
```

Adding global resource file and using it anywhere.

Thanks to dependency injection, it is possible to use this factory just injecting it anywhere you want.

Implementation and using of my Localization factory

Implementation

Project specific localization factory contains of two parts: Interface and resource-based implementation. It has two files:

`ILeaveManagementCustomLocalizerFactory.cs`

```
namespace LeaveManagement.Code.CustomLocalization {
    public interface ILeaveManagementCustomLocalizerFactory {
        IStringLocalizer CreateStringLocalizer(Type type);

        IStringLocalizer CommandsLocalizer { get; }

        IStringLocalizer MenuLocalizer { get;  }

        IStringLocalizer MiscelanousLocalizer { get; }

    }
}
```

And implementation for resources, which you can found in file
`ILeaveManagementCustomLocalizerFactory.cs`

In this class I use the Responsibility Chain of mappers. Chain stores in

```
        /// <summary>
        /// List of mappers from type to corresponded resource name.
        /// </summary>
        public ConcurrentBag<Func<Type, Tuple<string, string>>>
ConventionalResourceMappers { get; private set; }
```

In this example there are two default mappers, which reacts to convention about DataAnotation and controllers. It gets the full name of classes, and if it corresponds to Convention about controllers or view models, then returns Tuple with conventional resource name and assembly name. But it is possible to add your own.

Class for further refactoring, I have not understood the dependency injection on the degree, to make this service extensible from anywhere.

Listing of ILeaveManagementCustomLocalizerFactory.cs

```
using Microsoft.AspNetCore.Hosting;
using Microsoft.AspNetCore.Mvc.Localization;
using Microsoft.Extensions.Localization;
using System;
```

```
using System.Collections.Concurrent;
using System.Linq;
using System.Threading.Tasks;

namespace LeaveManagement.Code.CustomLocalization {
    public class LeaveManagementCustomLocalizerFactory :
ILeaveManagementCustomLocalizerFactory {

        #region Root namespaces for different parts of applications
        public const string ViewModelsRootNamespace = "ViewModels";
        public const string ControllersRootNameSpace = "Controllers";

        #endregion


        #region Private fields and constructor
        private readonly IStringLocalizerFactory StringLocalizerFactory;

        private readonly IHtmlLocalizerFactory HtmlLocalizerFactory; //For possible future
ViewLocalizer

        private readonly IWebHostEnvironment WebHostEnvironment; //For possible future
ViewLocalizer

        public LeaveManagementCustomLocalizerFactory(IStringLocalizerFactory
stringLocalizerFactory,
        IHtmlLocalizerFactory htmlLocalizerFactory,
        IWebHostEnvironment hostEnvironement = null) {
            StringLocalizerFactory = stringLocalizerFactory;
            HtmlLocalizerFactory = htmlLocalizerFactory;
            WebHostEnvironment = hostEnvironement;
            InitMappers();
        }
        #endregion

        #region Code for initialize rules


        /// <summary>
        /// List of mappers from type to corresponded resource name.
        /// </summary>
        public ConcurrentBag<Func<Type, Tuple<string, string>>>
ConventionalResourceMappers { get; private set; }

        /// <summary>
        /// Gets the IStringLocalizer for given type, looking also in custom
ClassRessourceMappers
        /// </summary>
        /// <param name="factory"></param>
        /// <param name="type"></param>
        /// <returns></returns>
        public IStringLocalizer MapRessourceToType(IStringLocalizerFactory factory, Type
type) {
            IStringLocalizer result = null;
            Tuple<string, string> knownMapping = null;
            Parallel.For(0, ConventionalResourceMappers.Count, (element, state) => {
                try {
```

```
                    var mapping =
ConventionalResourceMappers.ElementAt(element).Invoke(type);
                    if (mapping != null) {
                        knownMapping = mapping;
                        state.Break();
                    }

                }
                catch {
                    ;
                }
            });
            if (knownMapping != null)
                result = factory.Create(knownMapping.Item1, knownMapping.Item2);
            else
                result = factory.Create(type);
            return result;
        }


        private void InitMappers() {
            ConventionalResourceMappers = new ConcurrentBag<Func<Type, Tuple<string,
string>>>();

ConventionalResourceMappers.Add(LeaveManagementViewModelDataAnnotationsMapper);
            ConventionalResourceMappers.Add(LeaveManagementControllerMapper);

        }
        /// <summary>
        /// Gets resource type and assembly name for DataAnnoration translation for type,
if type follows the convention of
        /// ViewModels resources and type is from this assembly.
        /// </summary>
        /// <param name="expectedType">Type to get its resource name</param>
        /// <remarks>
        /// Convention for localizing DataAnnotations for ViewModels of this project is
next:
        /// ViewModels' full type name conform to template
LeaveManagement.ViewModels.Model.SpecificViewModel.
        /// for exemple, for ViewModel of type LeaveTypeNavigationViewModel, namespace
will be next: LeaveManagement.ViewModels.LeaveType;
        /// ViewModels' ressources must be located in path
[ResourcesFolder]/ViewModels/Model/ and have the name Model-Name.[Culture].resx.
        /// for exemple, for DataAnnotations for same
LeaveManagement.ViewModels.LeaveType.LeaveTypeNavigationViewModel path
        /// for resources must be:
[ResourcesFolder]\ViewModels\LeaveType\LeaveType.??.resx
        /// </remarks>
        /// <returns>If type is one of conventional ViewModels type, returns
Tuple(resourceName, assembmyName). Otherwise, null</returns>
        private Tuple<string, string> LeaveManagementViewModelDataAnnotationsMapper(Type
expectedType) {
            string[] nameSpacePath = expectedType.Namespace?.Split(new char[] { '.' });
            if(nameSpacePath.Length > 2 &&
nameSpacePath.ElementAt(1).Equals(ViewModelsRootNamespace)) {
                string resourceName = String.Concat(string.Join('.',
nameSpacePath.TakeLast(nameSpacePath.Length - 1)), ".", nameSpacePath.Last());
                return Tuple.Create(resourceName, expectedType.Assembly.FullName);
```

```
            }
            else
                return null;
        }

        /// <summary>
        /// Gets resource type and assembly for localizing controller messages.
        /// </summary>
        /// <param name="expectedType"></param>
        /// <returns></returns>
        private Tuple<string, string> LeaveManagementControllerMapper(Type expectedType) {
            string[] nameSpacePath = expectedType.FullName?.Split(new char[] { '.' });
            if(nameSpacePath.Length > 2 &&
nameSpacePath.ElementAt(1).Equals(ControllersRootNameSpace)) {
                string resourceName = String.Concat(string.Join('.',
nameSpacePath.TakeLast(nameSpacePath.Length - 1)), ".", nameSpacePath.Last());
                return Tuple.Create(resourceName, expectedType.Assembly.FullName);
            }
            else
                return null;
        }

        #endregion



        public IStringLocalizer CreateStringLocalizer(Type type) =>
MapRessourceToType(StringLocalizerFactory, type);

        private IStringLocalizer CommandsLocalizerField = null;

        public IStringLocalizer CommandsLocalizer {
            get {
                if (CommandsLocalizerField == null)
                    CommandsLocalizerField =
StringLocalizerFactory.Create("LeaveManagement.CommandsLocalizer.CommandLocalization",
this.GetType().Assembly.FullName);
                return CommandsLocalizerField;
            }
        }


        private IStringLocalizer MenuLocalizerField = null;
        public IStringLocalizer MenuLocalizer {
            get {
                if(MenuLocalizerField == null)
                    MenuLocalizerField =
StringLocalizerFactory.Create("LeaveManagement.MenuLocalizer.MenuLocalizer",
this.GetType().Assembly.FullName);
                return MenuLocalizerField;
            }
        }

        private IStringLocalizer MiscelanousLocalizerField = null;

        public IStringLocalizer MiscelanousLocalizer{
            get {
                if(MiscelanousLocalizerField == null)
```

```
                    MiscelanousLocalizerField =
StringLocalizerFactory.Create("LeaveManagement.MiscLocalizer.Miscellaneous",
this.GetType().Assembly.FullName);
                return MiscelanousLocalizerField;
            }
        }
    }
}
```

References

https://www.udemy.com/course/complete-aspnet-core-31-and-entity-framework-development/

https://docs.microsoft.com/fr-fr/aspnet/core/fundamentals/localization?view=aspnetcore-3.1

https://joonasw.net/view/aspnet-core-localization-deep-dive