

[articles](#) [quick answers](#) [discussions](#) [features](#) [community](#) [help](#)

Search for articles, questions, tips

[Articles](#) » [Web Development](#) » [ASP.NET](#) » [General](#)

Get insight to build your first Multi-Language ASP.NET MVC 5 Web Application

**jamuro77**13 Dec 2016 [CPOL](#)Rate this:  4.97 (29 votes)

This article explains how to create a simple Multi-Language ASP.NET MVC 5 Web Application. I'll show how to translate texts, localize images or entire views as well as how to deal with URL routing to support several languages.

[Download demo - 3.9 MB](#)

Introduction

This article explains how to create a simple Multi-Language ASP.NET MVC 5 Web application. The application will be able to deal with *English (United States)*, *Spanish* and *French* languages. *English* will be the *default* language. Of course, it will be very easy to extend the solution for including new languages.

To begin with, it's assumed that readers have a basic knowledge on ASP.NET MVC 5 framework. Other related technologies such as *jQuery* will be slightly commented throughout the article owing to they have been used to add new functionality. I will focus my explanations on MVC framework components. Anyway, I will try to explain how each component works or, at least, providing you with links to get more detailed information. It's not the goal of this article to explain each line of code for each technology or repeat explanations that are very well documented in other articles. My goal will be to explain the main features of our demo application at the same time as remember key issues to get better insight and understanding.

That being said, our demo application will have a **HomeController** with three basic typical actions such as **Index**, **About** and **Contact**. Besides, we will derive this controller from a **BaseController**, as we will see later, to provide every controller with common functionality. So, content rendered in **Views** called from **HomeController** will

be localized according to default or user selected language.

Background

From my view, when talking about Multi-Language ASP.NET MVC applications it would be necessary to take into account at least the following key issues:

- Consider components to apply *Globalization* and *Localization*.
- Configure *URL Routing* for Multi-Language purposes, especially bearing in mind SEO perspective. Regarding this, the most important issue is keeping distinct content on different URLs, never serving distinct content from the same URL.

Globalization and Localization

We must be able to set up the proper culture for each request being processed on current *Thread* running on controllers. So, we will create a **CultureInfo** object and set the **CurrentCulture** and **CurrentUICulture** properties on the **Thread** (see more about [here](#)) to serve content based on appropriate culture. To do this, we will extract culture information from *Url Routing*.

CurrentCulture property makes reference to *Globalization* or how dates, currency or numbers are managed. Instead, **CurrentUICulture** governs *Localization* or how content and resources are translated or localized. In this article I'm going to focus on *Localization*.

CultureInfo class is instantiated based on a unique name for each specific culture. This code uses the pattern "xx-XX" where the first two letter stands for language and the second one for sublanguage, country or region (See more about [here](#)). In this demo application, *en-US*, *es-ES* and *fr-FR* codes represent supporting languages *English* for United States, *Spanish* for Spain and *French* for France.

Having said that, here is a list of elements to be localized according to culture:

- Plain Texts.
 - We will translate texts by using **Resource Files**. In short, these files allow us to save content resources, mainly texts and images, based on a dictionary of key/value pairs. We will just employ these files to store texts, not images. Read more about at [Microsoft Walkthrough](#).
- Images.
 - We will localize images by extending **UrlHelper** class, contained in **System.Web.Mvc.dll** assembly. By means of extension methods inserted into this class, we will look for images within a previously-created structure of folders according to supported languages. Briefly explained, **UrlHelper** class contains methods to deal with URL addresses within a MVC application. In particular, we can obtain a reference to a **UrlHelper** class within a **Razor View** by making use of **Url** built-in property from **WebViewPage** class. See more about [here](#).
- Validation Messages from Client and Server code.
 - For translating Server Side Validation Messages we will employ **Resource Files**.
 - For translating Client Side Validation Messages we will override default messages. As we will make use of *jQuery Validation Plugin 1.11.1* to apply client validation, we'll have to override messages from this plugin. Localized messages will be saved in separate files based on supported languages. So, to gain access to localized script files we will extend again **UrlHelper** class.
- Localizing entire **Views** might be necessary according to requirements of our application. So, we'll consider this issue.

- In this demo, *English (United States)* and *Spanish* languages will not make use of this option but, with demo purposes, *French* language will. So, we will create a new derived **ViewEngine** from default **RazorViewEngine** to achieve this goal. This new view engine will look for **Views** through a previously-created folder tree.
- Other Script and CSS files.
 - For large applications, perhaps it would be necessary to consider localized scripts and CSS files. The same strategy chosen with image files might be used. We will not dive into this issue, simply take into account.
- Localized content from back-end storage components such as databases.
 - We'll not work with databases in this demo application. The article would be too long. Instead, we'll assume that, if necessary, information about current culture set on **Thread** will be provided to database from *Data Access Layer*. This way, corresponding translated texts or localized images should be returned accordingly. At least, bear in mind this if you're planning use localized content from databases.

Let's see some screenshot about our demo application:

Home page *English (United States)* version:



Home page *Spanish* version:



It's a very simple application, but enough to get insight about multi-language key issues.

- *Home View page* contains localized texts and images.
- *About View page* just includes localized texts.
- *Contact View page* contains also localized texts but it also includes a *partial view* with a form to post data and apply client and server validation over corresponding underlying model.
- Shared *Error View page* will be translated as well.
- A list of selecting flags are provided from layout view page.

URL Routing

First of all, we must accomplish with the fact of not serving different -language-based- content from the same URL. Configure appropriate *URL Routing* is mandatory to serve language-based content in accordance with different URLs. So, we will configure routing for including Multi-Language support by extracting specific culture from URL routing.

Our URLs addresses, on debug mode, will look like as it is shown below. I'm assuming that our demo application is being served on *localhost* with XXXXX port.

- *English (United States) language*:
 - `http://localhost:XXXXX/Home/Index` or `http://localhost:XXXXX/en-US/Home/Index`
 - `http://localhost:XXXXX/Home/About` or `http://localhost:XXXXX/en-US/Home/About`
 - `http://localhost:XXXXX/Home/Contact` or `http://localhost:XXXXX/en-US/Home/Contact`
- *Spanish language*:

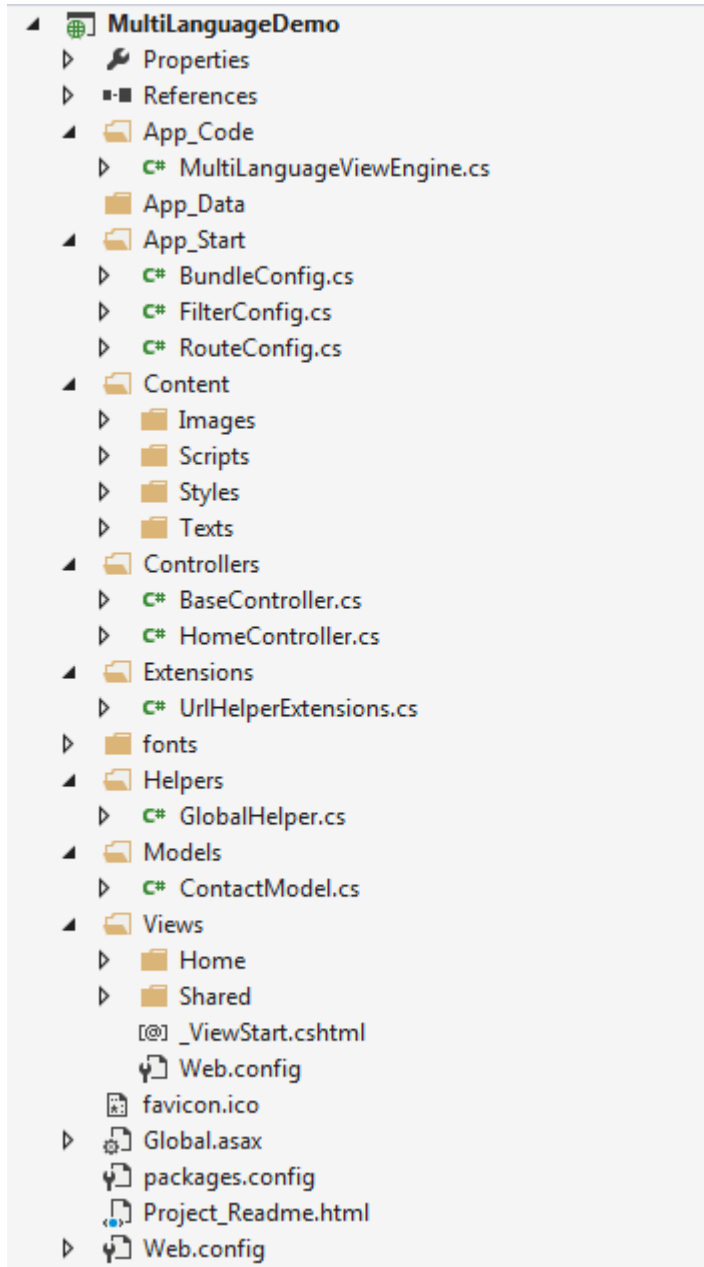
- <http://localhost:XXXXX/es-ES/Home/Index>
- <http://localhost:XXXXX/es-ES/Home/About>
- <http://localhost:XXXXX/es-ES/Home/Contact>
- French language:
 - <http://localhost:XXXXX/fr-FR/Home/Index>
 - <http://localhost:XXXXX/fr-FR/Home/About>
 - <http://localhost:XXXXX/fr-FR/Home/Contact>

Furthermore, we will provide the user with a list of supporting languages in the *Layout View Page*. So, users always can get to the desired language by clicking on one of them. When users select a different language, we will use a *Cookie* to save this manual choice. The use of a *Cookie* might generate controversy. So, to use it or not is up to you. It's not a key point in the article. We will use it taking into account that we will never create *Cookies* from server side based on content of URL routing. So, if a given user never changes language manually, he will navigate in the language that he entered our website. Next time users get into our website, if the cookie exists, they will be redirected to the appropriate URL according to their last language selection. Anyway, remember again, never think of using only *Cookies*, *Session State*, *Browser's Client* user settings, etc. to serve different content from the same URL.

Using the code

First steps to create our Multi-Language Application

I have taken the simple MVC 5 template given by Microsoft Visual Studio 2013 for starting to work, changing authentication options to *No Authentication*. As you can see below, the name of my demo application is *MultiLanguageDemo*. Then, I have rearranged folders as is shown below:



- Notice folders under *Content* directory. Personally, I like to have this structure for storing *Images*, *Scripts*, *Styles* and *Texts*. Each time you create a new directory and add classes to it, a new namespace is added by default with the folder's name. Take it into account. I have modified **web.config** in Views folder to include these new namespaces. Doing this, you can gain direct access to classes in these namespaces from Razor view code nuggets.

```

<system.web.webPages.razor>
  <host factoryType="System.Web.Mvc.MvcWebRazorHostFactory, System.Web.Mvc, Version=5.2.3.0, Culture=neutral, PublicKeyToken=31bf3856ad364e35">
  <pages pageBaseType="System.Web.Mvc.WebViewPage">
    <namespaces>
      <add namespace="System.Web.Mvc" />
      <add namespace="System.Web.Mvc.Ajax" />
      <add namespace="System.Web.Mvc.Html" />
      <add namespace="System.Web.Optimization" />
      <add namespace="System.Web.Routing" />
      <add namespace="MultiLanguageDemo" />
      <add namespace="MultiLanguageDemo.Models" />
      <add namespace="MultiLanguageDemo.Helpers" />
      <add namespace="MultiLanguageDemo.Extensions" />
      <add namespace="MultiLanguageDemo.Content.Texts" />
    </namespaces>
  </pages>
</system.web.webPages.razor>

```

- As *en-US* will be the default culture, it's necessary to configure **web.config** in accordance with:

```

<system.web>
  <compilation debug="true" targetFramework="4.5"/>
  <httpRuntime targetFramework="4.5"/>
  <globalization culture="en-US" uiCulture="en-US"/>
</system.web>

```

- We will use a custom **GlobalHelper** class to include global common functionality such as reading current culture on **Thread** or default culture in **web.config**. Here is the code:

[Hide](#) [Copy Code](#)

```

public class GlobalHelper
{
    public static string CurrentCulture
    {
        get
        {
            return Thread.CurrentThread.CurrentUICulture.Name;
        }
    }

    public static string DefaultCulture
    {
        get
        {
            Configuration config = WebConfigurationManager.OpenWebConfiguration("/");
            GlobalizationSection section = (GlobalizationSection)config.GetSection("system.web/globalization");

```

```
        return section.UICulture;
    }
}
```

Setting-up URL Routing

We'll have two routes, *LocalizedDefault* and *Default*. We'll use **lang** placeholder to manage culture. Here is the code within **RouteConfig** class in **RouteConfig.cs** file ([see more about URL Routing](#)):

[Hide](#) [Copy Code](#)

```
public class RouteConfig
{
    public static void RegisterRoutes(RouteCollection routes)
    {
        routes.IgnoreRoute("{resource}.axd/{*pathInfo}");

        routes.MapRoute(
            name: "LocalizedDefault",
            url: "{lang}/{controller}/{action}",
            defaults: new { controller = "Home", action = "Index" },
            constraints: new { lang="es-ES|fr-FR|en-US" }
        );

        routes.MapRoute(
            name: "Default",
            url: "{controller}/{action}",
            defaults: new { controller = "Home", action = "Index", lang = en-US }
        );
    }
}
```

On one hand, *Default* route will be used to match URLs without specifying explicit culture. Therefore, we'll configure it for using default culture. Notice how **lang** is set to *en-US* culture in **defaults** param.

On the other hand, *LocalizedDefault* route is configured to use specific culture on URLs. Besides, **lang** param is restricted to be included in supporting languages *es-ES*, *fr-FR* or *en-US*. Notice how this is configured by setting **constraints** param in **MapRoute** method. This way we'll cover all previously-established routes.

Configuring Controllers to serve proper based-language content

As I said before, to switch culture is necessary to create a **CultureInfo** object to set the **CurrentCulture** and **CurrentUICulture** properties on the **Thread** that processes each http request sent to controllers. Using MVC 5, there are several ways of achieving this. In this case, I will create an abstract **BaseController** class from which the

rest of controllers will be derived . The **BaseController** will contain common functionality and will override **OnActionExecuting** method from **System.Web.Mvc.Controller** class. The key point about **OnActionExecuting** method is to be aware of it is always called before a controller method is invoked.

At last, simply saying that another way of getting this would be by means of *Global Action Filters* instead of using a base class. It's not considered in this example, but bearing it in mind if you like more.

Let's have a look at our **BaseController** class code:

[Hide](#) [Shrink ▲](#) [Copy Code](#)

```
public abstract class BaseController : Controller
{
    private static string _cookieLangName = "LangForMultiLanguageDemo";

    protected override void OnActionExecuting(ActionExecutingContext filterContext)
    {
        string cultureOnCookie = GetCultureOnCookie(filterContext.HttpContext.Request);
        string cultureOnURL = filterContext.RouteData.Values.ContainsKey("lang")
            ? filterContext.RouteData.Values["lang"].ToString()
            : GlobalHelper.DefaultCulture;
        string culture = (cultureOnCookie == string.Empty)
            ? (filterContext.RouteData.Values["lang"].ToString())
            : cultureOnCookie;

        if (cultureOnURL != culture)
        {
            filterContext.HttpContext.Response.RedirectToRoute("LocalizedDefault",
                new { lang=culture,
                    controller = filterContext.RouteData.Values["controller"],
                    action = filterContext.RouteData.Values["action"]
                });
            return;
        }

        SetCurrentCultureOnThread(culture);

        if (culture != MultiLanguageViewEngine.CurrentCulture)
        {
            (ViewEngines.Engines[0] as MultiLanguageViewEngine).SetCurrentCulture(culture);
        }

        base.OnActionExecuting(filterContext);
    }

    private static void SetCurrentCultureOnThread(string lang)
    {
        if (string.IsNullOrEmpty(lang))
            lang = GlobalHelper.DefaultCulture;
        var cultureInfo = new System.Globalization.CultureInfo(lang);
    }
}
```

```
        System.Threading.Thread.CurrentThread.CurrentUICulture = cultureInfo;
        System.Threading.Thread.CurrentThread.CurrentCulture = cultureInfo;
    }

    public static String GetCultureOnCookie(HttpRequestBase request)
    {
        var cookie = request.Cookies[_cookieLangName];
        string culture = string.Empty;
        if (cookie != null)
        {
            culture= cookie.Value;
        }
        return culture;
    }
}
```

BaseController class overrides **OnActionExecuting** method. Then, we get information about specific culture from URL Routing and Cookies. If there's no cookie, culture on **Thread** will be set from Url Routing. Otherwise, if a final user has selected manually a language and then a cookie exists, the http response will be redirected to the corresponding route containing language stored in cookie.

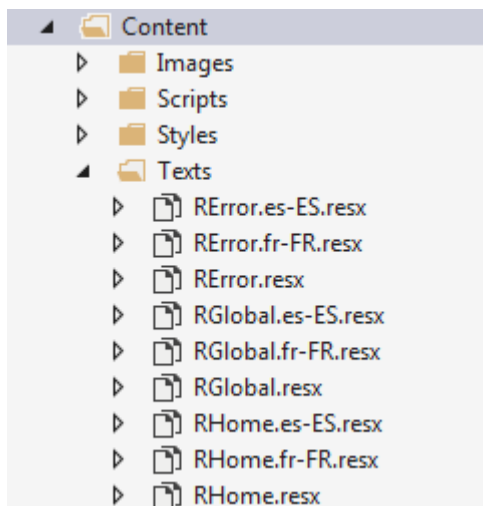
Additionally, to set current culture on Thread, **BaseController** use **SetCurrentCultureOnThread** private function. First, a new **CultureInfo** class is created based on specific culture passed as param. Finally, **CurrentUICulture** and **CultureInfo** properties from current **Thread** are assigned with previously created **CultureInfo** object.

Dealing with Plain Texts

To translate plain texts, we will use **Resource Files**. These are a great way of storing texts to be translated. The storage is based on a dictionary of key/value pairs, where *key* is a string identifying a given resource and *value* is the translated *text* or localized *image*. Internally, all this information is saved in XML format and compiled dynamically by Visual Studio Designer.

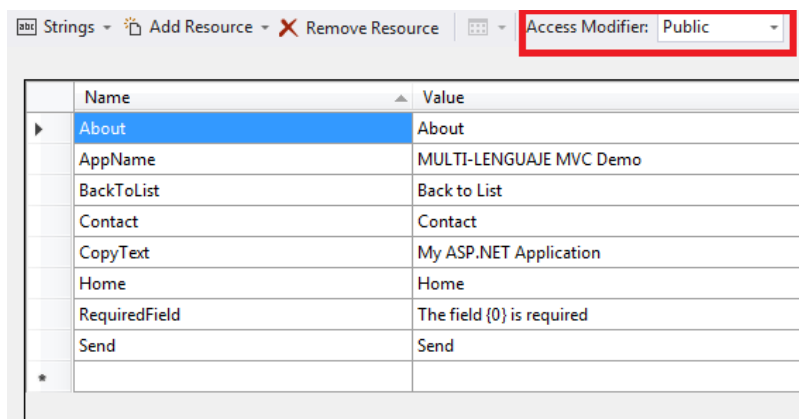
Resource Files have a **RESX** extension. So, in this demo we will create three different **Resource Files** for the default culture. One for storing global texts, **RGlobal.resx**, another for general error messages, **RError.resx**, and the last for storing messages related to Home Controller, **RHome.resx**. I like to create this structure of resource files in my projects, normally including one resource file for each controller, but you can choose another way if you prefer.

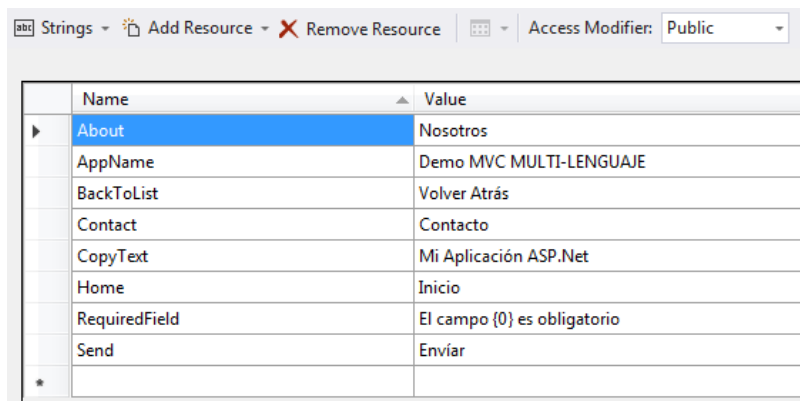
For other supporting languages we will create resource files with names **RGlobal.es-ES.resx**, **RError.es-ES.resx**, **RHome.es-ES.resx** (Spanish) and **RGlobal.fr-FR.resx**, **RError.fr-FR.resx** and **RHome.fr-FR.resx** (French). Note the cultural code for each name. Here is our **Resource Files** tree:



The most important points to know about are:

- When you create a resource file for the default culture such as **RGlobal.resx** file, an **internal** class called **RGlobal** is auto-generated by Visual Studio. Using the designer, you should change the *Access Modifier* to **public** for using it in the solution. Let's have a look at our **RGlobal** files for *English* and *Spanish* languages:



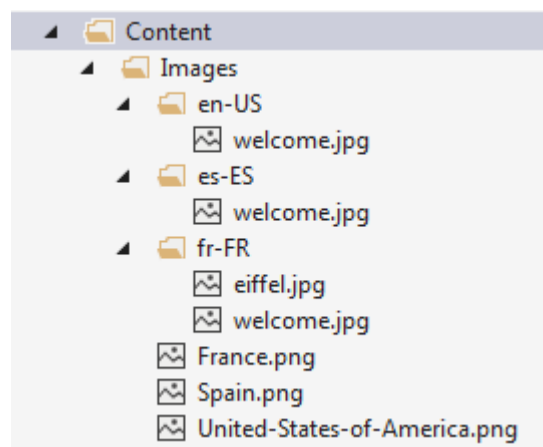


Name	Value
About	Nosotros
AppName	Demo MVC MULTI-LENGUAJE
BackToList	Volver Atrás
Contact	Contacto
CopyText	Mi Aplicación ASP.Net
Home	Inicio
RequiredField	El campo {0} es obligatorio
Send	Envíar
*	

- Resources for each specific culture are compiled in separate assemblies, saved in different subdirectories according to culture and named as *AssemblyName.resources.dll*. In our case names will be **MultiLanguageDemo.resources.dll**
- Once specific culture is set on **Thread**, the runtime will choose the assembly accordingly.
- Individual resources can be consumed for controllers or other classes by concatenating the resource file name with the keyword. For instance **RGlobal.About**, **RGlobal.AppName**, etc.
- To use individual resources inside views with Razor syntax you just have to add code such as **@RHome.Title**, **@RHome.Subtitle** or **@RHome.Content**.

Dealing with Images

As I said before, we will just store texts in **Resource Files**, although images might be saved too. Personally, I prefer to save images in another way. Let's have a look at our *Images* folder under *Content* directory.



As you can see, a specific folder has been created for each culture. Images not requiring localization will be saved directly in *Images* folder as *France.jpg* or *Spain.jpg*. These files just contain flags for showing and selecting languages and therefore they don't require localization. The rest of images requiring localization will be stored separately. For instance, *welcome.jpg* file, under *en-US* subdirectory contains a picture with text "Welcome", instead *welcome.jpg* file under *es-ES* subdirectory contains a drawing with text "Bienvenido".

Having said that, let's go on with our **GetImage** method extension in **UrlHelper** class for selecting localized images. This static method will be contained in **UrlHelperExtensions** static class inside a file called **UrlHelperExtensions.cs** under *Extensions* folder. Here is the code:

[Hide](#) [Shrink ▲](#) [Copy Code](#)

```
public static class UrlHelperExtensions
{
    public static string GetImage(this UrlHelper helper,
        string imageFileName,
        bool localizable=true)
    {
        string strUrlPath, strFilePath = string.Empty;
        if (localizable)
        {
            /* Look for current culture */
            strUrlPath = string.Format("/Content/Images/{0}/{1}",
                GlobalHelper.CurrentCulture,
                imageFileName);
            strFilePath = HttpContext.Current.Server.MapPath(strUrlPath);
            if (!File.Exists(strFilePath))
            {
                /* Look for default culture */
                strUrlPath = string.Format("/Content/{0}/Images/{1}",
                    GlobalHelper.DefaultCulture,
                    imageFileName);
            }
            return strUrlPath;
        }

        strUrlPath = string.Format("/Content/Images/{0}", imageFileName);
        strFilePath = HttpContext.Current.Server.MapPath(strUrlPath);
        if (File.Exists(strFilePath))
        {
            /* Look for resources in general folder as last option */
            return strUrlPath;
        }

        return strUrlPath;
    }
}
```

We'll extend **UrlHelper** by adding a new **GetImage** method. This method will allow us to look for localized images under *Images* directory. We just need to call the method by passing to it the proper image filenames. There's another boolean param to set whether image is localized. If so, method will look for results inside the corresponding subdirectory based on current culture and if not encountered will try with default culture and general folder in that order. Anyway, first search should be enough if everything is well-configured.

A typical call within a template **View** would be:

[Hide](#) [Copy Code](#)

```

```

`Url` is a property of `System.Web.Mvc.WebViewPage` class, from which all Razor Views are derived. This property returns a `UrlHelper` instance. This way, we can gain access to our `GetImage` method.

Dealing with Validation Messages

We'll consider both server and client validation. To apply localization to server side validation we'll use `Resource Files` whereas to client validation we'll create a structure of directories similar to what we did with images. Then, we'll create new script files to override default messages according to supporting languages and we'll extend `UrlHelper` class to gain access to these new files.

Server Validation

Server validation is usually executed on controllers over models. If validation is not correct, model state dictionary object `ModelState` that contains the state of the model will be set as incorrect. In code, this is equal to set `IsValid` property of `ModelState` to `false`. Consequently, `ModelState` dictionary will be filled up with validation messages according to input fields, global validations, etc. these messages should be translated.

In this example I'm going to show how translate validation messages originated from *Data Annotations*. In MVC projects is very common to configure server validations by using classes contained in `System.ComponentModel.DataAnnotations`. Let's see an example.

This is the code related to `Contact Model` to be applied to `Contact View`:

[Hide](#) [Shrink](#) ▲ [Copy Code](#)

```
namespace MultiLanguageDemo.Models
{
    [MetadataType(typeof(ContactModelMetaData))]
    public partial class ContactModel
    {
        public string ContactName { get; set; }
        public string ContactEmail { get; set; }
        public string Message { get; set; }
    }

    public partial class ContactModelMetaData
    {
        [Required(ErrorMessageResourceName = "RequiredField",
            ErrorMessageResourceType = typeof(RGlobal))]
        [Display(Name = "ContactName", ResourceType = typeof(RHome))]
        public string ContactName { get; set; }

        [Required(ErrorMessageResourceName = "RequiredField",
            ErrorMessageResourceType = typeof(RGlobal))]
        [Display(Name = "ContactEmail", ResourceType = typeof(RHome))]
        [DataType(DataType.EmailAddress)]
        public string ContactEmail { get; set; }

        [Required(ErrorMessageResourceName = "RequiredField",
```

```

        ErrorMessageResourceType = typeof(RGlobal))]
        [Display(Name = "Message", ResourceType = typeof(RHome))]
        public string Message { get; set; }
    }
}

```

On one hand, we have a **ContactModel** class with three simple properties. On the other hand, we have a **ContactModelMetaData** class used to apply validations over **ContactModel** and to set further functionality or metadata in order to show labels related to fields, data types, etc.

Regarding validation we are configuring all model fields as **Required**. So, to enforce localization, it's necessary to reference the auto-generated class associated with a Resource File. It is done by means of **ErrorMessageResourceType** property. We also have to configure, the keyword name related to the corresponding validation message that we want to show. It is done by using **ErrorMessageResourceName** property. This way, messages from Resource Files -being selected automatically based on culture- will be returned accordingly.

Client Validation

By using Client Validation is possible to execute validation in clients avoiding unnecessary requests to controllers. We'll make use of this feature by means of *jQuery Validation Plugin 1.11.1* and *jQuery Validation Unobtrusive Plugin*. References to these files are auto-generated when you start a new MVC 5 project by using Microsoft Visual Studio MVC 5 template project. You can enable Client Validation in **web.config** file as is shown in figure below:

```

<?xml version="1.0" encoding="utf-8"?>
<configuration>
  <appSettings>
    <add key="webpages:Version" value="3.0.0.0"/>
    <add key="webpages:Enabled" value="false"/>
    <add key="ClientValidationEnabled" value="true"/>
    <add key="UnobtrusiveJavaScriptEnabled" value="true"/>
  </appSettings>
  <system.web>
    <compilation debug="true" targetFramework="4.5"/>
    <httpRuntime targetFramework="4.5"/>
    <globalization culture="en-US" uiCulture="en-US"/>
  </system.web>
  <runtime>...</runtime>
</configuration>

```

You can also enable/disable Client Validation directly from **Views** by means of inherited **Html** property from **System.Web.Mvc.WebViewPage** class. As it is shown at figure below, **Html** property within a **View** returns a **HtmlHelper** object that contains **EnableClientValidation** and **EnableUnobtrusiveJavaScript** methods. Once Client Validation is enabled, **HtmlHelper** class is allowed to write client validation code automatically.

```

<div class="row">
  <div class="col-md-12 text-center">
    <p class="text-center">
      
    </p>
    <p>@RHome.Content</p>
  </div>
</div>

```

AttributeEncode
 EnableClientValidation
 EnableUnobtrusiveJavaScript
 Encode
 EndForm
 EnumDropDownListFor<>

void HtmlHelper.EnableClientValidation()
Enables or disables client validation.

In our demo application we're employing *jQuery Validation Plugins* to perform validation. So, default messages are shown in *English* but we need to supply translated messages for all supporting languages. To achieve this, we will extend the plugin. First, we'll create a directory tree as it's shown at picture below.

 Scripts Directory Tree for Client Validation

Then, for each supported language we'll create a *javascript* file to override default messages according to culture running on current **Thread**. Here is the code related to *Spanish* language:

Hide Copy Code

```

jQuery.extend(jQuery.validator.messages, {
  required: "Este campo es obligatorio.",
  remote: "Por favor, rellena este campo.",
  email: "Por favor, escribe una dirección de correo válida",
  url: "Por favor, escribe una URL válida.",
  date: "Por favor, escribe una fecha válida.",
  dateISO: "Por favor, escribe una fecha (ISO) válida.",
  number: "Por favor, escribe un número entero válido.",
  digits: "Por favor, escribe sólo dígitos.",
  creditcard: "Por favor, escribe un número de tarjeta válido.",
  equalTo: "Por favor, escribe el mismo valor de nuevo.",
  accept: "Por favor, escribe un valor con una extensión aceptada.",
  maxlength: jQuery.validator.format("Por favor, no escribas más de {0} caracteres."),
  minlength: jQuery.validator.format("Por favor, no escribas menos de {0} caracteres."),
  rangelength: jQuery.validator.format("Por favor, escribe un valor entre {0} y {1} caracteres."),
  range: jQuery.validator.format("Por favor, escribe un valor entre {0} y {1}."),
  max: jQuery.validator.format("Por favor, escribe un valor menor o igual a {0}."),
  min: jQuery.validator.format("Por favor, escribe un valor mayor o igual a {0}."),
});

```


I'm taken for granted that jQuery and **jQuery Validation Plugin** are loaded before these files are. Anyway, for referencing these files from a view that require client validation, we have to use the following code:

[Hide](#) [Copy Code](#)

```
@Scripts.Render("~/bundles/jqueryval")
@if (this.Culture != GlobalHelper.DefaultCulture)
{
    <script src="@Url.GetScript("jquery.validate.extension.js")" defer></script>
}
```

As I did before, I have extended **UrlHelper** class to add a new method **GetScript** for searching localized script files. Then, I make use of it by referencing **jquery.validate.extension.js** file right after loading **jQuery Validation plugin**, but only if current culture is different from default one.

As a consequence of all previously-mentioned, when we try to send our **Contact View** without filling up any required field, we obtain the following validation messages according to *English* and *Spanish* languages.

Validation messages for **English** language:

Contact

- The field Contact Name is required
- The field Contact Email is required
- The field Contact Message is required

Contact Name

The field Contact Name is required

Contact Email

The field Contact Email is required

Contact Message

The field Contact Message is required

Send

Validation messages for **Spanish** languages:

Contacto

- El campo Nombre de Contacto es obligatorio
- El campo Email de Contacto es obligatorio
- El campo Mensaje es obligatorio

Nombre de Contacto

El campo Nombre de Contacto es obligatorio

Email de Contacto

El campo Email de Contacto es obligatorio

Mensaje

El campo Mensaje es obligatorio

Envíar

At last, here is a code snippet from `_Contact` partial view in `_Contact.cshtml` file:

```

<div class="form-horizontal">
    <h4>@RHome.Contact</h4>
    <hr />

    @if (ViewBag.ContactResult != null)
    {
        var classFromResult = ViewBag.ContactResult ? "text-success" : "text-danger";
        <p class="@classFromResult"><strong>@ViewBag.ContactResultMessage</strong></p>
    }
    else
    {
        @Html.ValidationSummary(false, "", new { @class = "text-danger" })
        <div class="form-group">
            @Html.LabelFor(model => model.ContactName,
                htmlAttributes: new { @class = "control-label col-md-2" })
            <div class="col-md-10">
                @Html.EditorFor(model => model.ContactName,
                    new { htmlAttributes = new { @class = "form-control" } })
                @Html.ValidationMessageFor(model => model.ContactName, "",
                    new { @class = "text-danger" })
            </div>
        </div>

        <div class="form-group">...</div>

        <div class="form-group">...</div>

        <div class="form-group">...</div>
    }
</div>

```

This partial view contains a simple form to post data. If this partial view is rendered from a *Get* method on http request, a form will be shown. Instead, if it's rendered after a *Post* request sending data, result from *Post* will be displayed. Nothing more to say, except if you want to dive into source code I'm using a Post-Redirect-Get pattern to do this ([see more about](#)).

Focusing on validation, I'd like to point out that when client validation is activated, some methods from **HtmlHelper** class such as **ValidationMessageFor** (see picture above), are enabled to write html code to manage validation for each input field according to annotations in model metadata classes. For simple validations you don't need to do anything else.

Dealing with Localizing Entire Views

So far, we have achieved pretty much everything regarding localization for not very complex large applications. These might demand new features such as localizing Entire Views. That is, **Views** must be very different for each culture. Therefore, we need to add new features to our application. I'm going to apply this case to the *French* culture. Views for this

language will be different. What do we need to reach this goal? To begin with, we need to create new specific **Views** for this language. Secondly we must be able to reference these **Views** when *French* culture is selected. At last, all previously explained should work well too. Let's see how we can accomplish all of this.

First, we'll create a directory tree under Views directory as is shown below:

Notice *fr-FR* subdirectory under *Home* directory. It will contain specific Views for *French* culture. Views directly under Home directory will be used for *Default* and *Spanish* culture. If there were more controllers than **Home Controller**, the same strategy should be taken.

At this point, we have to supply a way of selecting template **Views** based on culture. For this, we will create a custom **ViewEngine** derived from **RazorViewEngine** ([more about here](#)). We'll call this engine **MultiLanguageViewEngine**. Briefly explained, view engines are responsible for searching, retrieving and rendering views, partial views and layouts. By default there are two view engines pre-loaded when you run a MVC 5 Web application: Razor and ASPX View Engine. However, we can remove them or add new custom view engines, normally in **Application_Start** method in **Global.asax** file. In this case, we'll unload pre-existing default view engines to add our **MultiLanguageViewEngine**. It will do the same as **RazorViewEngine** but additionally and according to culture will look up for specific subdirectories containing localized entire view templates. Let's have a look at code stored in **MultiLanguageViewEngine.cs** file under *App_code* folder:

[Hide](#) [Shrink](#) ▲ [Copy Code](#)

```
namespace MultiLanguageDemo
{
    public class MultiLanguageViewEngine : RazorViewEngine
    {
        private static string _currentCulture = GlobalHelper.CurrentCulture;

        public MultiLanguageViewEngine()
            : this(GlobalHelper.CurrentCulture){
        }

        public MultiLanguageViewEngine(string lang)
        {
            SetCurrentCulture(lang);
        }

        public void SetCurrentCulture(string lang)
        {
            _currentCulture = lang;
            ICollection<string> arViewLocationFormats =
                new string[] { "~/Views/{1}/" + lang + "{0}.cshtml" };
            ICollection<string> arBaseViewLocationFormats = new string[] {
                @"~/Views/{1}/{0}.cshtml",
                @"~/Views/Shared/{0}.cshtml"};
            this.ViewLocationFormats = arViewLocationFormats.Concat(arBaseViewLocationFormats).ToArray();
        }

        public static string CurrentCulture
        {
            get { return _currentCulture; }
        }
    }
}
```

```

    }
}
</string></string></string>

```

To begin with, notice how **MultiLanguageViewEngine** inherits from **RazorViewEngine**. Then, I have added a constructor for getting supporting languages. This constructor will set new locations where looking for localized entire views by making use of new **SetCurrentCulture** method. This method set a new location to look for views based on **lang** param. This new path is inserting at first position in the array of locations to search. and the array of strings is saved in **ViewLocationFormats** property. Besides, **MultiLanguageViewEngine** will return the specific culture used for setting this property.

That being said, how to deal with **MultiLanguageViewEngine**? First, we'll create a new instance of this view engine in **Application_Start** method in **Global.asax** file. Secondly, we'll switch current culture for the custom view engine right after setting culture on **Thread**. More in detail, we'll override **OnActionExecuting** method on our **BaseController** class. I remind you this method is always called before any method on controller is invoked.

Let's see **Application_Start** method in **Global.asax** file:

[Hide](#) [Copy Code](#)

```

protected void Application_Start()
{
    AreaRegistration.RegisterAllAreas();
    FilterConfig.RegisterGlobalFilters(GlobalFilters.Filters);
    RouteConfig.RegisterRoutes(RouteTable.Routes);
    BundleConfig.RegisterBundles(BundleTable.Bundles);

    ViewEngines.Engines.Clear();
    ViewEngines.Engines.Add(new MultiLanguageViewEngine());
}

```

Bolded code shows how to unload collection of pre-load view engines and how to load our new **MultiLanguageViewEngine**.

Now, let's see again **OnActionExecuting** method in **BaseController** class focusing on this:

[Hide](#) [Shrink](#) [Copy Code](#)

```

protected override void OnActionExecuting(ActionExecutingContext filterContext)
{
    string cultureOnCookie = GetCultureOnCookie(filterContext.HttpContext.Request);
    string cultureOnURL = filterContext.RouteData.Values.ContainsKey("lang")
        ? filterContext.RouteData.Values["lang"].ToString()
        : GlobalHelper.DefaultCulture;
    string culture = (cultureOnCookie == string.Empty)
        ? (filterContext.RouteData.Values["lang"].ToString())
        : cultureOnCookie;

    if (cultureOnURL != culture)
    {
        filterContext.HttpContext.Response.RedirectToRoute("LocalizedDefault",

```

```

        new { lang=culture,
              controller = filterContext.RouteData.Values["controller"],
              action = filterContext.RouteData.Values["action"]
        });
    return;
}

SetCurrentCultureOnThread(culture);

if (culture != MultiLanguageViewEngine.CurrentCulture)
{
    (ViewEngines.Engines[0] as MultiLanguageViewEngine).SetCurrentCulture(culture);
}

base.OnActionExecuting(filterContext);
}

```

Bolded code above shows how to update our custom view engine. If current culture on **Thread**, stored in **culture** variable, is different from current culture in **MultiLanguageViewEngine**, our new engine is updated to be synchronized with **Thread**. We gain access to **MultiLanguageViewEngine** through **Engines** collection property of **ViewEngines** class with zero index. Take into account that we unloaded pre-loaded view engines in **global.asax** file to add only **MultiLanguageViewEngine**. So, it is in the first place.

Switching languages from User Interface

As it is shown in previous screenshots, our demo application will have a list of flags to switch language, placed in the lower right corner. So, this functionality will be included in **_Layout.cshtml** file. This file will contain the layout for every view in the project.

On one hand, here is a excerpt of html code to render flags. It is a simple option list to show flags representing supporting languages. Once a language is set, the selected flag will be highlighted with a solid green border.

To handle user selections we'll include javascript code. To begin with, I have created a javascript file **multiLanguageDemo.js** to include common functionality to the application. Basically, this file contains functions to read and write cookies. It is based on "*namespace pattern*" ([see more about here](#)) Needless, this file is contained in *Scripts* folder.

Once a user clicks an option, a cookie with the selected language will be created. After this, the page will be reloaded to navigate to the corresponding URL based on specified language. Here is the jQuery code to get this:

You must notice use of **MultiLanguageDemo.Cookies.getCookie** to read cookie value and **MultiLanguageDemo.Cookies.SetCookie** to set value on cookie. Besides, when some flag is clicked, *javascript* code sets an **active-lang** class to the selected flag, captures language from **data-lang** attribute and reload view page.

Points of Interest

I have had a great time building this little demo application. Anyway, it has been hard to try to explain in detail somethings not in my mother tongue. Sorry about.

Environment

This demo has been developped using *Microsoft Visual Studio 2013 for the Web* with *.Net Framework 4.5* and *MVC 5*. Other main components used have been *jQuery JavaScript Library v1.10.2*, *jQuery Validation Plugin 1.11.1*, *jQuery Validation Unobtrusive Plugin* and *Bootstrap v3.0.0*.

History

It's the first version of the article. In later reviews I would like to add some improvements about providing the application with some examples using both globalization issues and localized content from databases objects such as tables, stored procedures, etc.

License

This article, along with any associated source code and files, is licensed under [The Code Project Open License \(CPOl\)](#)


Share

About the Author

jamuro77



Architect

Spain 



Telecommunication Engineer by University of Zaragoza, Spain.

Passionate Software Architect, MCPD, MCAD, MCP. Over 10 years on the way building software.

Mountain Lover, Popular Runner, 3km 8'46, 10km 31'29, Half Marathon 1h08'50, Marathon 2h27'02. I wish I could be able to improve, but It's difficult by now

Comments and Discussions

You must [Sign In](#) to use this message board.

Search Comments

Spacing

Relaxed

 Layout

Normal

 Per page

25

Update

			First	Prev	Next
	RenderPartial	Member 10774546	28-Aug-19 6:15		
	Solution AJAX	rudolf cruz	6-Jun-19 9:35		
	JesonResult	rudolf cruz	6-Jun-19 9:00		
	how to add custom font with resources	ko ko maung	18-Mar-19 5:32		
	Ajax seems will redirect to xx-XX/controller/action which lead to parameter missing	siew peng	29-Nov-18 16:39		

The pages are loaded twice and the parameter values are lost	zZMaxZz	13-Nov-18 7:07
Re: The pages are loaded twice and the parameter values are lost	Member 7844417	17-Nov-18 4:12
Re: The pages are loaded twice and the parameter values are lost	siew peng	29-Nov-18 16:45
Missing Script tree client validation Image	Member 12190015	4-Jul-18 5:58
Attribute Routes	Member 2712404	16-Apr-18 3:04
Localized views are not displayed	Member 10103365	15-Apr-18 8:35
Re: Localized views are not displayed	Member 13398733	2-May-18 4:25
how to change dir for arabic	Member 13051527	13-Apr-18 21:12
Related with Model	Member 13050076	6-Apr-18 2:27
MultiLanguageViewEngine NullPointerException	Alessandro Bellone	11-Jan-18 7:14
Wrong controller handler	squizzy	9-Jan-18 21:28
Localized views are not displayed	roman-f	4-Jan-18 1:11
Re: Localized views are not displayed	Member 10435319	17-Feb-18 3:29
Re: Localized views are not displayed	Member 13398733	2-May-18 4:26
Re: Localized views are not displayed	Member 13590369	7-Jun-18 9:08
Re: Localized views are not displayed	Jan Šotola - Rebex	26-Jun-18 3:41
Multi language with db in mvc	Elxan Quliyev	18-Nov-17 13:47
MultiLanguag Site with Partial Views	Member 12772959	14-Nov-17 21:31
Outstanding!	Thiago Daher	6-Nov-17 18:12
Re: Outstanding!	jamuro77	7-Nov-17 12:20

Last Visit: 3-Mar-20 6:47 Last Update: 7-Mar-20 14:51

[Refresh](#)[1](#) [2](#) [Next »](#)[General](#) [News](#) [Suggestion](#) [Question](#) [Bug](#) [Answer](#) [Joke](#) [Praise](#) [Rant](#) [Admin](#)

Use Ctrl+Left/Right to switch messages, Ctrl+Up/Down to switch threads, Ctrl+Shift+Left/Right to switch pages.

[Permalink](#)[Advertise](#)[Privacy](#)[Cookies](#)[Terms of Use](#)Layout: [fixed](#) | [fluid](#)Article Copyright 2016 by jamuro77
Everything else Copyright © [CodeProject](#), 1999-2020

Web04 2.8.200307.1