This blog post is intended for software development engineers building advanced solutions for Service Manager and have a need to localize their solutions.

Before reading this post, please check out this blog post on [localizing content in management packs](http://blogs.technet.com/servicemanager/archive/2009/07/24/localizing-management-pack-content.aspx).

In most places in the Service Manager console, when you provide the display strings in management packs the right string just shows up depending on the context the user is in and what display language the user is running the OS as.

When you are creating custom forms though you need to do a bit of extra work to localize the form string content. There are a couple of ways to do this:

1. The typical approach of using the Resource class in .Net and resource files. You can read more about this approach here: <http://www.codeproject.com/KB/dotnet/Localization.aspx> This is just standard development stuff so I’m not going to go into it more here.
2. Leveraging DisplayStrings defined in management packs.

As a best practice we recommend that you localize your string content for forms using the DisplayString approach. There are a few reasons for this:

* It is consistent with the way the rest of your management pack localization is done.
* When you need to hand off files to your localization team/vendor, all you need to do is handoff the management pack XML file(s) instead of digging around for various resource files in your code base.
* You can easily add languages in the future without having to release updated binaries. All you need to do is release an updated management pack file or even just a dependent management pack file that just contains additional languages.
* Your customers can add their own languages if they need simply using dependent management packs without impacting your code.
* You can leverage the display strings for the property labels and other things so there is guaranteed consistency in terminology. If you call a property ‘X’ in one place it will always be called ‘X’ everywhere. If you need to change it to ‘Y’ it can be updated everywhere by changing it in one place.

In this blog post, I’ll show you how to localize the string content in a custom form. For this example, we’ll take the Service Request form introduced earlier in this [blog post](http://blogs.technet.com/servicemanager/archive/2010/01/08/creating-custom-forms-for-service-manager-using-custom-controls-service-request-example.aspx).

In that first example we defined the labels like this:

<Label Grid.Row="0" Grid.Column="0" Margin="8,8,85,0" Height="28" VerticalAlignment="Top" Name="lblTitle" Grid.ColumnSpan="2">Title:</Label>

Notice how I hard coded the label name in the XAML. Bad developer! I should be fired. : )

Instead what we want to do is define our labels so that the string is retrieved depending on the language being shown.

There are really two main scenarios here:

1. Label:Control Pairs for Properties
2. Non-Data-Model-Specific Strings on the Form

Just a quick **important note** before we begin. You’ll need to add these two namespace declarations to your XAML in order to do what I’m going to describe below.

xmlns:scwpf=http://schemas.microsoft.com/SystemCenter/Common/UI/Wpf

xmlns:views="clr-namespace:Microsoft.EnterpriseManagement.UI.FormsInfra;assembly=Microsoft.EnterpriseManagement.UI.FormsInfra"

You’ll also need to add a Reference to Microsoft.EnterpriseManagement.UI.FormsInfra.dll in your Visual Studio/Blend project. You can find this .dll in the installation directory of Service Manager.

# Label:Control Pairs for Properties

Typically, when you put a label next to a control you want the label for the control to be the same as the Property name of the property that the control is bound to. For example, if I have a textbox bound to the Title property on the ServiceRequest class in this example, it makes a lot of sense to have the label for that control be the same as the Property name – in this case ‘Title’. So – here, I want to bind my label content to the localized Property display name.



Let’s think about what this means in more detail. Remember that the Service Request class derives from the System.WorkItem class. This is what the XML looks like for defining the Service Request class:

<ClassType ID="Microsoft.Demo.WorkItem.ServiceRequest"

Accessibility="Public"

Abstract="false"

Base="WorkItem!System.WorkItem"

Hosted="false"

Singleton="false"

Extension="false">

<Property ID="Classification"

Type="enum"

Key="false"

Required="false" EnumType="ServiceRequestClassificationEnum" />

</ClassType>

The work item class defines the following properties (these are the IDs):

* ActualEndDate
* ActualStartDate
* ContactMethod
* CreatedDate
* Description
* Id
* ScheduledEndDate
* ScheduledStartDate
* Title

So you can see that the Title (and others) property is an inherited property. The Classification property (see above) is a property that is defined on the Service Request class itself.

Remember that each of these Properties has a Display Name associated with it *for each language* that is defined in the Language Pack section of the management pack. For example the Classification property has this DisplayString in the ENU Language Pack:

<DisplayString ElementID="Microsoft.Demo.WorkItem.ServiceRequest" SubElementID="Classification">

<Name>Service Request Classification</Name>

</DisplayString>

OK, so, now what we want to do is instead of hard coding the label for the Title control like this…

<Label Grid.Row="0" Grid.Column="0" Margin="8,8,85,0" Height="28" VerticalAlignment="Top" Name="lblTitle" Grid.ColumnSpan="2">Title:</Label>

… we want to bind to the Property’s DisplayString value. The way we do that is like this:

<Label Grid.Row="0" Grid.Column="0" Margin="8,8,85,0" Height="28" VerticalAlignment="Top" Name="lblTitle" Grid.ColumnSpan="2”>

<Label.Content>

<Binding

RelativeSource="

{RelativeSource FindAncestor, AncestorType={x:Type views:FormView}}"

Path="Strings[Title].Value"

FallbackValue="[Title]:">

<Binding.Converter>

<scwpf:FormatterConverter>{0}:</scwpf:FormatterConverter>

</Binding.Converter>

</Binding>

</Label.Content>

</Label>

The key to this approach is to use the XAML method of *Binding* an attribute value to something. The part in blue highlight is what binds the label content to Property’s DisplayString value. Everything except for the part in blue there remains constant for each label on your form. All you have to do is put the Property ID (see magenta highlight above) in the square brackets.

You can optionally define a FallbackValue. This is the value that is shown if the form host cannot resolve the string to display from the Binding configuration. Remember that the form host try the following in order:

1. Try to find the display string for the Property in the same language as the user’s current language setting in the OS.
2. Try to find the display string for the Property in the default language (as defined in the Management Pack) if different than #1.
3. Use the FallbackValue defined in the <Binding> above.

It is just our convention to put the Property ID in square brackets followed by a colon for the FallbackValue. You can choose to follow this convention or do something else that might make more sense in your particular case. It’s just a string.

The last thing I’ll point out here is the BindingConverter. The BindingConverter simply takes the string that is looked up and appends a colon after it. This is the normal user interface convention for control labels. You can use this or something different or not at all as appropriate.

Note: this approach works the same for inherited properties (like the Title Property example above) and for properties on the class that you are creating the form for (like Classification in this example).

# Non-Data-Model-Specific Strings on the Form

There are also cases where you may have some strings on the form – maybe some instructions in a label or a label that describes a group of controls or an error message. In these cases we need a way to define a new display string that is not associated with a particular control/Property.

For this example, consider the label over this ListView control:



Since this is a list of related objects we can’t simply bind to a Property display name as above. We need a way to define a new string and have it be localizable. To declare a new string you need to do the following three things in the Management Pack:

1. Create a FormString element inside the Form element

<Form ID="Form.ServiceRequest" …>

<FormStrings>

<FormString ID="Label\_AffectedCIs">

$MPElement[Name="Microsoft.Demo.ServiceRequest.Form.Label\_AffectedCIs"]$

</FormString>

</FormStrings>

</Form>

1. Create a StringResource element

<StringResource ID="Microsoft.Demo.ServiceRequest.Form.Label\_AffectedCIs"/>

The yellow highlighted items need to match between the MPElement reference in the FormString element and in the StringResource element ID. If you need to you can refernce StringResources in other sealed MPs by referencing them using a MP alias in front of the MPElement Name.

1. Create the DisplayString for each language as usual

<DisplayString ElementID="Microsoft.Demo.ServiceRequest.Form.Label\_AffectedCIs">

<Name>Affected Configuration Items</Name>

</DisplayString>

The ElementID atribute value needs to match the StringResource ID attribute value.

Now, to reference this new FormString in your XAML you need to do essentially the same thing as for Property display strings, but instead of putting in the Property ID, you put in the FormString ID.

<Label Grid.Row="6" Grid.Column="0" Margin="8,8,0,0" Height="28" VerticalAlignment="Top" HorizontalAlignment="Left" Name="lblAffectedCIs" Grid.ColumnSpan="2">

<Binding

RelativeSource="

{RelativeSource FindAncestor, AncestorType={x:Type views:FormView}}" Path="Strings[Label\_AffectedCIs].Value" FallbackValue="[AffectedCIs]:">

<Binding.Converter>

<scwpf:FormatterConverter>{0}:</scwpf:FormatterConverter>

</Binding.Converter>

</Binding>

</Label>

You can see other examples of how I have created strings for things like column headers and button labels in the new Service Request form/management pack. You can also see other examples of how to use the FormatterConverter in different situations like a button label that should always be followed by a ‘…’ string like ‘Add…’.

The completed Service Request solution with all the string elements properly localized using Management Pack DisplayStrings is available here.

Anybody want to contribute to a community project to localize the Service Request management pack content? If you do, send me the updated version of the MP with the additional Language Pack [twright at microsoft dot com] and I’ll add it in and credit you in the comments. : )

Happy Localizing!