Creating New Forms, Using Custom Controls : Service Request Example

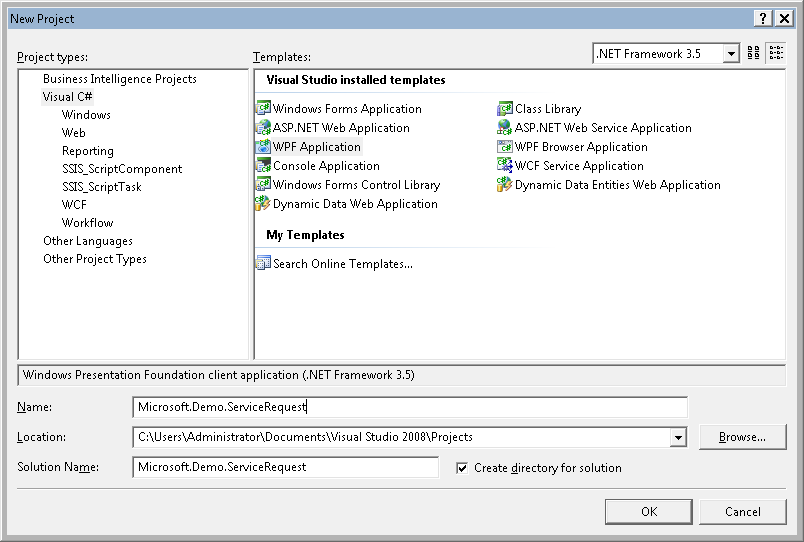
I have been getting some requests for how to create new forms and especially for how to use the custom controls/forms that Service Manager provides:

* User Picker
* List (aka “Enumeration value”) Picker
* Date Picker
* Instance (aka “Object”) Picker

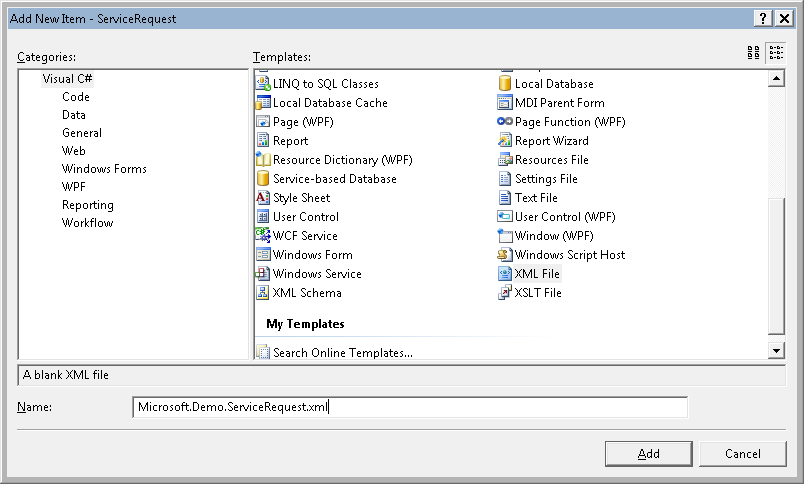
Along the way, I want to show you more about Type Projections. We might as well do something useful while we are learning. Service Manager doesn’t provide a Service Request work item class out of the box in Service Manager 2010. For now, we recommend using Change Requests for complex service requests involving approval steps, routing work to multiple people, or automated activities. You can use Incidents for more basic kinds of service requests. For this example, we’ll create a basic Service Request work item class which will give us an opportunity to learn about creating new forms, using the custom controls, and other aspects of building a new solution from the ground up.

# Getting Started

First start by creating a new ServiceRequest Visual Studio Project based on the WPF Application template:



Next, delete the two .xaml files that come with the project by default. We won’t need those. Then, add a XML file to the project by right clicking on the project, choosing Add -> New Item and selecting XML file from the options. Name the file Microsoft.Demo.ServiceRequest.



# Management Pack

Now, in the management pack let’s add a couple of MP References, Define our new Work Item class, and fill out the first part of the Language Pack section. Our basic MP looks like this:

<ManagementPack ContentReadable="true" SchemaVersion="1.1" OriginalSchemaVersion="1.1" xmlns:xsd="http://www.w3.org/2001/XMLSchema" xmlns:xsl="http://www.w3.org/1999/XSL/Transform">

<Manifest>

<Identity>

<ID>Microsoft.Demo.ServiceRequest</ID>

<Version>1.0.0.0</Version>

</Identity>

<Name>Microsoft Demonstration Connector MP</Name>

<References>

<Reference Alias="System">

<ID>System.Library</ID>

<Version>7.0.5244.0</Version>

<PublicKeyToken>31bf3856ad364e35</PublicKeyToken>

</Reference>

<Reference Alias="WorkItem.Library">

<ID>System.WorkItem.Library</ID>

<Version>7.0.5244.0</Version>

<PublicKeyToken>31bf3856ad364e35</PublicKeyToken>

</Reference>

</References>

</Manifest>

<TypeDefinitions>

<EntityTypes>

<ClassTypes>

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

Accessibility="Public"

Abstract="false" Base="WorkItem.Library!System.WorkItem"

Hosted="false"

Singleton="false"

Extension="false"/>

</ClassTypes>

</EntityTypes>

</TypeDefinitions>

<LanguagePacks>

<LanguagePack ID="ENU" IsDefault="true">

<DisplayStrings>

<DisplayString ElementID="Microsoft.Demo.ServiceRequest">

<Name>Service Request</Name>

</DisplayString>

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

<Name>Service Request</Name>

</DisplayString>

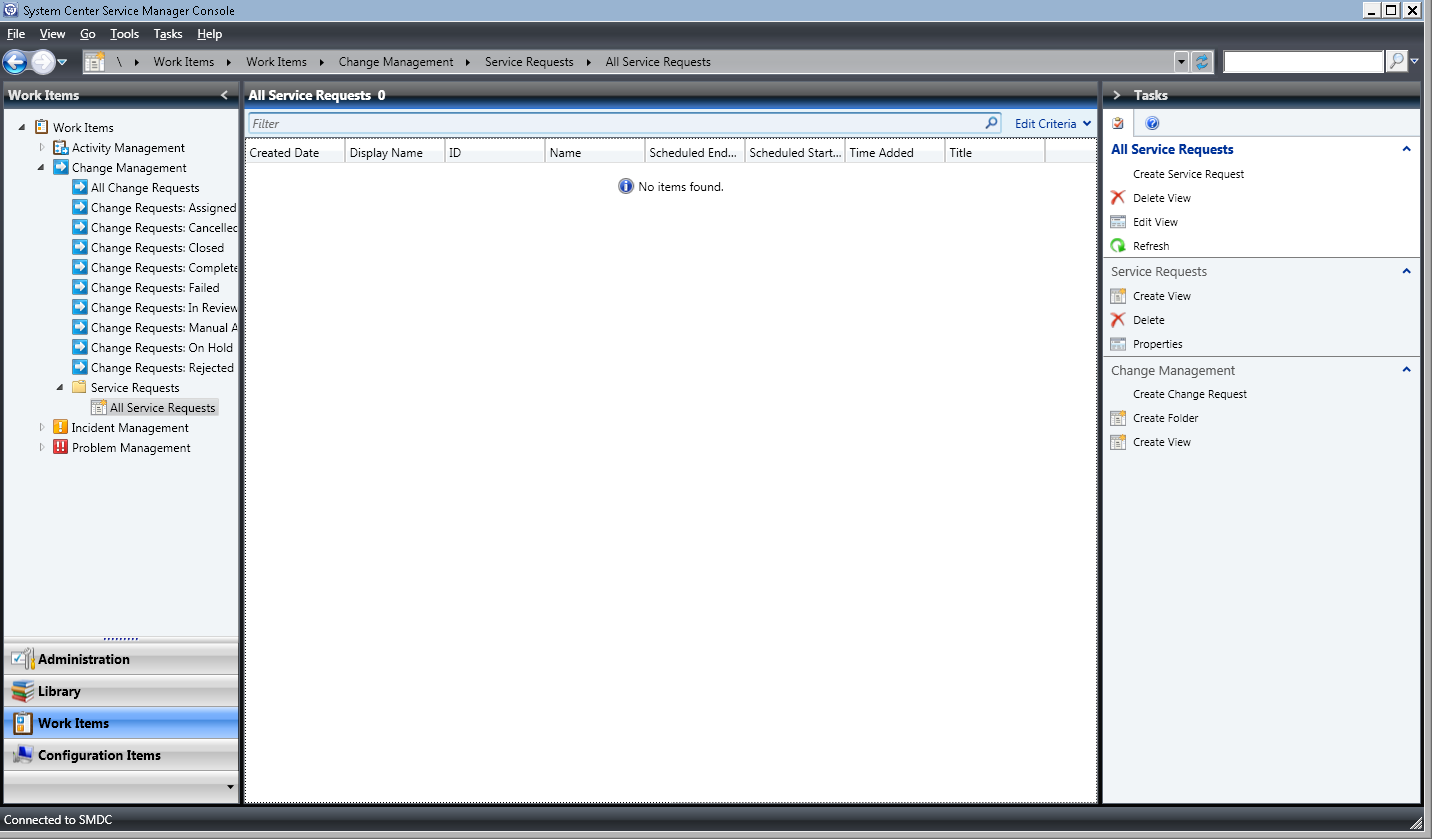
</DisplayStrings>

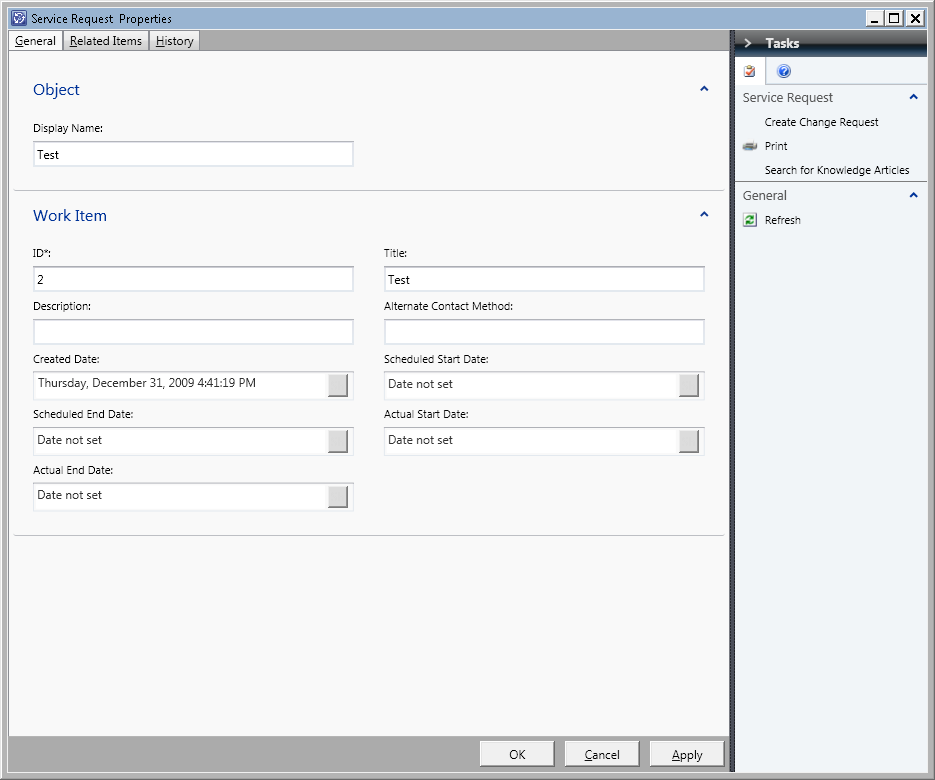
</LanguagePack>

</LanguagePacks>

</ManagementPack>

At this point, we could import the MP into Service Manager and actually start creating views of Service Requests and creating the Service Requests themselves using the generic form.





Let’s get a little bit fancier though… The generic form doesn’t provide a great user experience in this case for a few reasons:

1. The Display Name and Title properties are both exposed which could lead to confusion.
2. The ID property is read/write which is not good since it is the key property
3. The datetime fields don’t line up very well with each other – for example the Scheduled Start/End Date fields don’t line up.
4. The generic form doesn’t expose relationship types – so although our new class has inherited the WorkItemAssignedToUser relationship type there is no way to set that using the generic form.
5. We didn’t inherit any Enumeration data type properties from System.WorkItem so if we want to demonstrate how to use the Enumeration Value Picker control we’ll need to add an Enumeration data type property to our Service Request class.

Let’s go back to our management pack…

First off, let’s add a Classification property that is an Enumeration data type property.

<EntityTypes>

<ClassTypes>

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

Accessibility="Public"

Abstract="false"

Base="WorkItem.Library!System.WorkItem"

Hosted="false"

Singleton="false"

Extension="false">

<Property ID="Classification"

Type="enum"

Key="false"

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

</ClassType>

</ClassTypes>

<EnumerationTypes>

<EnumerationValue ID="ServiceRequestClassificationEnum"

Accessibility="Public" />

<EnumerationValue ID="ServiceRequestClassificationEnum.AddUserToGroup"

Accessibility="Public"

Parent="ServiceRequestClassificationEnum"

Ordinal="0" />

<EnumerationValue ID="ServiceRequestClassificationEnum.CreateUser"

Accessibility="Public"

Parent="ServiceRequestClassificationEnum"

Ordinal="5" />

<EnumerationValue ID="ServiceRequestClassificationEnum.DisableUser"

Accessibility="Public"

Parent="ServiceRequestClassificationEnum"

Ordinal="10" />

</EnumerationTypes>

</EntityTypes>

Notice how the EnumType attribute of the Classification property points to the root EnumerationValue - ServiceRequestClassificationEnum. The other EnumerationValue elements point to ServiceRequestClassificationEnum as the parent. This way – AddUserToGroup, CreateUser, and DisableUser will be shown in the drop down in the form. We could create a deeper hiearchy by just creating new EnumerationValue elements which point to the second level EnumerationValue elements as parents and so on. The Ordinal attribute controls the display order in the drop down control with the lowest numbers appearing first.

Don’t forget to add the LanguagePack content for these…

<DisplayString ElementID="Microsoft.Demo.WorkItem.ServiceRequest"

SubElementID="Classification">

<Name>Service Request Classification</Name>

</DisplayString>

<DisplayString ElementID="ServiceRequestClassificationEnum.AddUserToGroup">

<Name>Add User to Group</Name>

</DisplayString>

<DisplayString ElementID="ServiceRequestClassificationEnum.CreateUser">

<Name>Create User</Name>

</DisplayString>

<DisplayString ElementID="ServiceRequestClassificationEnum.DisableUser">

<Name>Disable User</Name>

</DisplayString>

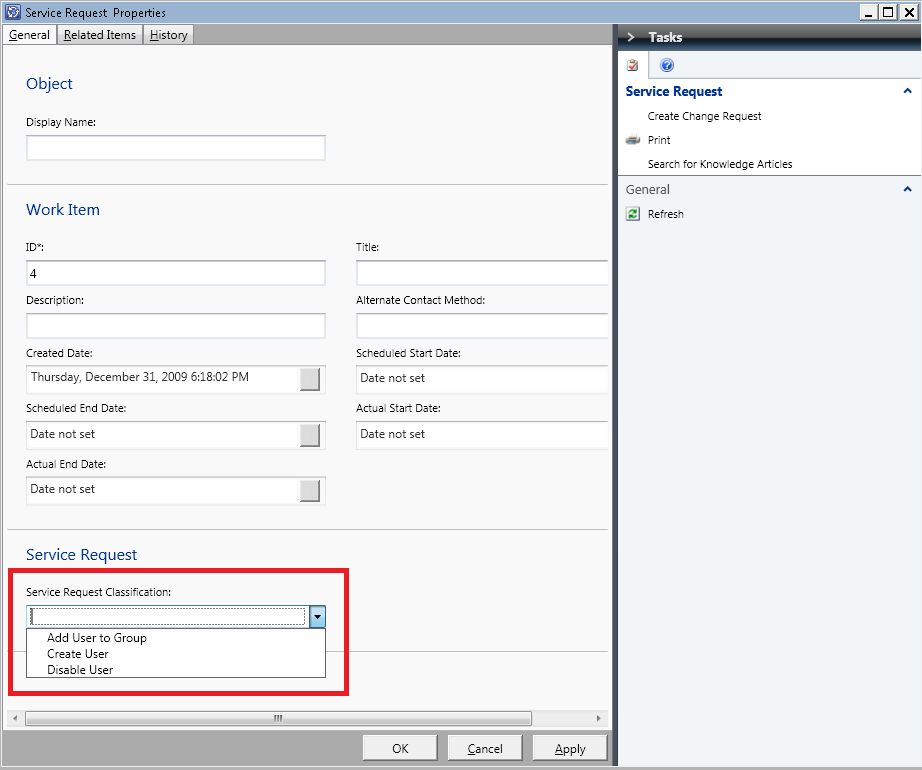
Let’s also export out the MP that we imported and copy the Folder and View out of that MP into our MP…

After cleaning up the auto-generated IDs, adding references to the following MPs:

* Microsoft.EnterpriseManagement.ServiceManager.UI.Console
* ServiceManager.WorkItem.Library
* Microsoft.EnterpriseManagement.ServiceManager.UI.Authoring

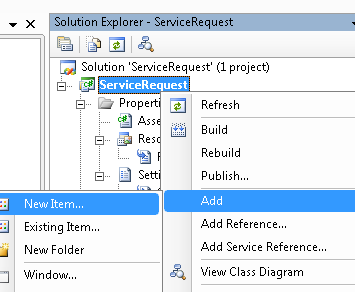
… and changing the parent folder from Change Requests to the Work Items root we have a top level folder and a view in our MP.

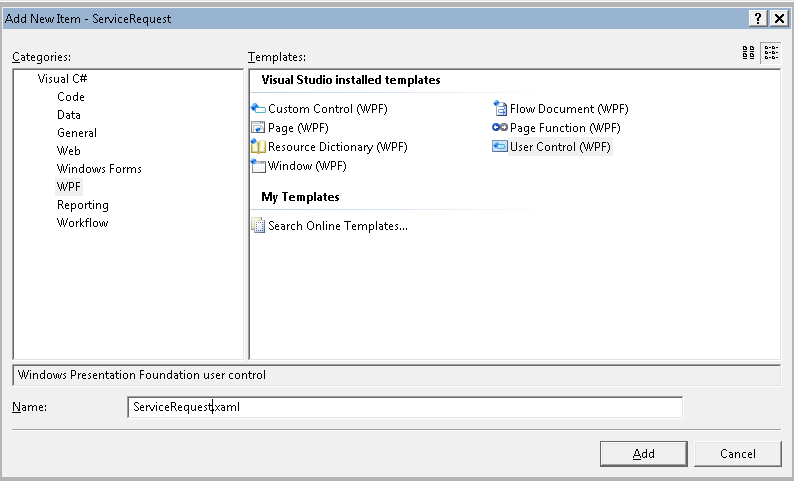
Now our Generic form display s our enum property using the Enum Value Picker control:



# Creating the Custom Form

To create the Custom Form we need to first add a WPF Form to our Project. Do this by right clicking on the project in the Project Explorer and select Add -> New Item… and then choose WPF User Control from the dialog:





So – now if we think about the properties that are available to us from the model:

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| Class | Property | Primary Key | Data Type | Is Auto Increment |
| System.Entity | DisplayName | 0 | String | 0 |
| System.WorkItem | ScheduledStartDate | 0 | DateTime | 0 |
| System.WorkItem | Title | 0 | String | 0 |
| System.WorkItem | ContactMethod | 0 | String | 0 |
| System.WorkItem | Id | 1 | String | 1 |
| System.WorkItem | ActualEndDate | 0 | DateTime | 0 |
| System.WorkItem | ScheduledEndDate | 0 | DateTime | 0 |
| System.WorkItem | ActualStartDate | 0 | DateTime | 0 |
| System.WorkItem | Description | 0 | String | 0 |
| System.WorkItem | CreatedDate | 0 | DateTime | 0 |
| Microsoft.Demo.WorkItem.ServiceRequest | Classification | 0 | Enum | 0 |

And the relationships that are inherited:

|  |  |  |
| --- | --- | --- |
| Source Class | Relationship Type Name | Target Class |
| System.Entity | System.Containment | System.Entity |
| System.Entity | System.Hosting | System.Entity |
| System.Entity | System.Membership | System.Entity |
| System.Entity | System.Reference | System.Entity |
| System.Entity | System.WatchedBy | System.Perspective |
| System.WorkItem | System.WorkItemAboutConfigItem | System.ConfigItem |
| System.WorkItem | System.WorkItemAffectedUser | System.User |
| System.WorkItem | System.WorkItemAssignedToUser | System.User |
| System.WorkItem | System.WorkItemCreatedByUser | System.User |
| System.WorkItem | System.WorkItemDependsOnWorkItem | System.WorkItem |
| System.WorkItemGroup | System.WorkItemGroupContainsWorkItems | System.WorkItem |
| System.WorkItem | System.WorkItemHasBillableTime | System.WorkItem.BillableTime |
| System.WorkItem | System.WorkItemHasChildWorkItem | System.WorkItem |
| System.WorkItem | System.WorkItemHasFileAttachment | System.FileAttachment |
| System.WorkItem | System.WorkItemImpactsService | System.Service |
| System.WorkItem | System.WorkItemRelatesToConfigItem | System.ConfigItem |
| System.WorkItem | System.WorkItemRelatesToWorkItem | System.WorkItem |
| System.WorkItem | System.WorkItemResolvesWorkItem | System.WorkItem |
| Microsoft.SystemCenter.HealthService | Microsoft.SystemCenter.HealthServiceManagesEntity | System.Entity |
| Microsoft.SystemCenter.HealthService | Microsoft.SystemCenter.HealthServiceShouldManageEntity | System.Entity |
| Microsoft.SystemCenter.Site | Microsoft.SystemCenter.SiteMembership | System.Entity |
| Microsoft.SystemCenter.InstanceGroup | Microsoft.SystemCenter.InstanceGroupContainsEntities | System.Entity |
| System.WorkItem | System.WorkItemContainsActivity | System.WorkItem.Activity |
| System.Entity | System.EntityLinksToKnowledgeDocument | System.Knowledge.Article |

So that we can bring all of this together we first need to create a TypeProjection in our MP like this:

<TypeProjections>

<TypeProjection ID="TypeProjection.ServiceRequest" Accessibility="Public" Type="Microsoft.Demo.WorkItem.ServiceRequest">

<Component Path="$Target/Path[Relationship='WorkItem!System.WorkItemAssignedToUser']$" Alias="AssignedTo" />

<Component Path="$Target/Path[Relationship='WorkItem!System.WorkItemAffectedUser']$" Alias="AffectedUser" />

<Component Path="$Target/Path[Relationship='WorkItem!System.WorkItemCreatedByUser']$" Alias="CreatedBy" />

<Component Path="$Target/Path[Relationship='WorkItem!System.WorkItemRelatesToConfigItem']$" Alias="RelatesToCI" />

</TypeProjection>

</TypeProjections>

This type projection has relationship types in it from the System.WorkItem.Library MP which are aliased as ‘WorkItem’. So, we’ll need to add a reference to this MP in the References section of the MP like this:

<Reference Alias="WorkItem">

<ID>System.WorkItem.Library</ID>

<Version>7.0.5244.0</Version>

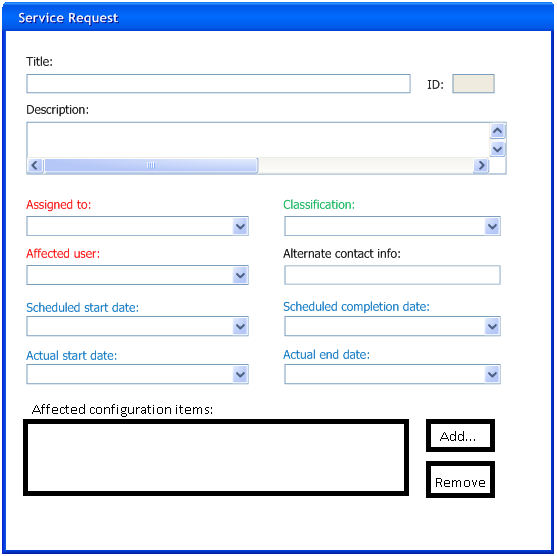
<PublicKeyToken>31bf3856ad364e35</PublicKeyToken>

</Reference>

So – in summary we have the need for the following controls:

* 4 DatePicker controls
  + ScheduledStartDate
  + ActualStartDate
  + ScheduledEndDate
  + ActualEndDate
* 3 TextBox controls
  + Title
  + ID (read only)
  + Description
* 1 ListPicker
  + Classification
* 2 UserPicker
  + Assigned to user (System.WorkItemAffectedUser)
  + Affected user (System.WorkItemAssignedToUser)
* ListView + InstancePicker
  + Work Item Relates to Config Item (System.WorkItemRelatesToConfigItem)
* Other Notes
  + We will use code-behind in our form to combine the ID and Title property values into the Display Name
  + The Created Date will be set using code behind logic as well

Here is a quick mockup of what we are going to try and do:



The controls with the black labels will be textboxes. The controls with red labels will be user pickers. The controls with blue labels will be datetime controls. The control with the green label will be an enum value picker.

When you are designing WPF forms you have to think in terms of grids. The above layout can be done pretty easily using a 2 column x 6 row grid. The Title/ID and Description controls will be in rows that span both columns.

Let’s start by changing the default size from 300x300 to 600x600:

<UserControl x:Class="Microsoft.Demo.ServiceRequest.ServiceRequest"

xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"

xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"

Height="600" Width="600">

And then creating our grid in the XAML editor part of the designer. I like to turn on ShowGridLines while I am creating the form and then turn it off before the final delivery. If we add the following XAML:

<Grid.RowDefinitions>

<RowDefinition />

<RowDefinition />

<RowDefinition />

<RowDefinition />

<RowDefinition />

<RowDefinition />

</Grid.RowDefinitions>

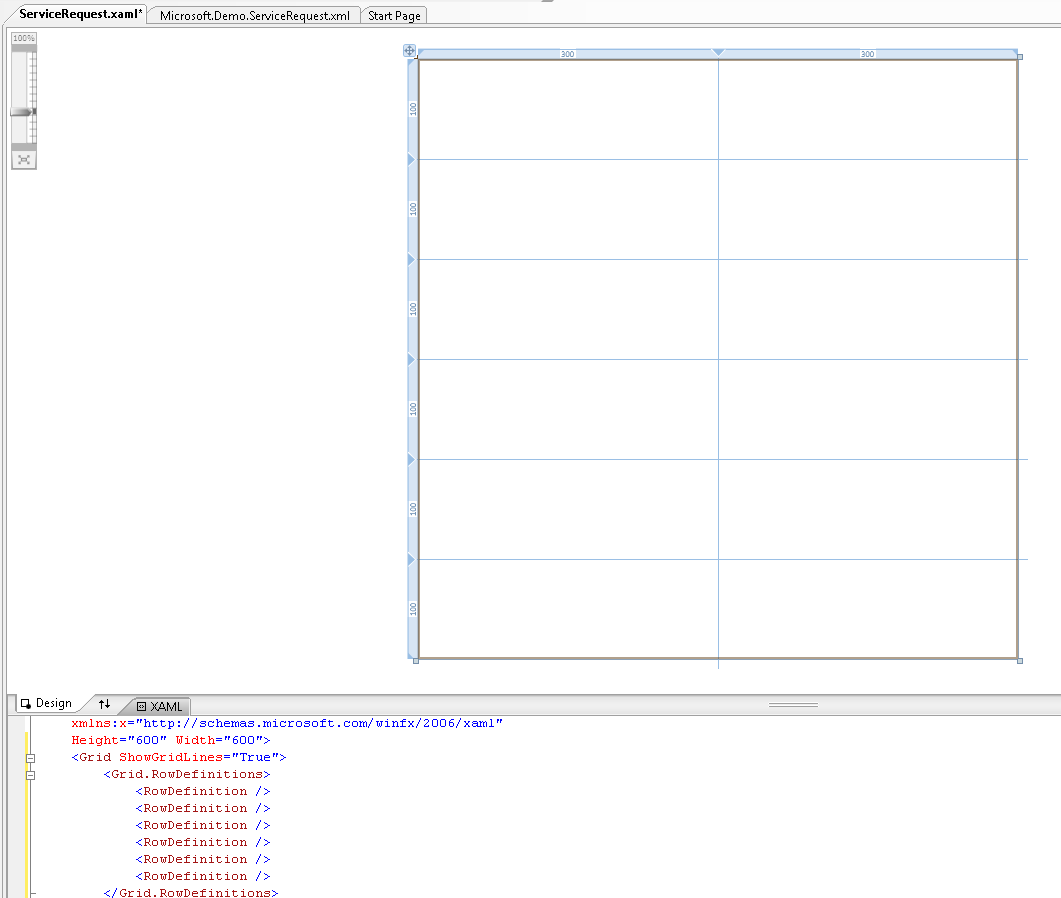
<Grid.ColumnDefinitions>

<ColumnDefinition />

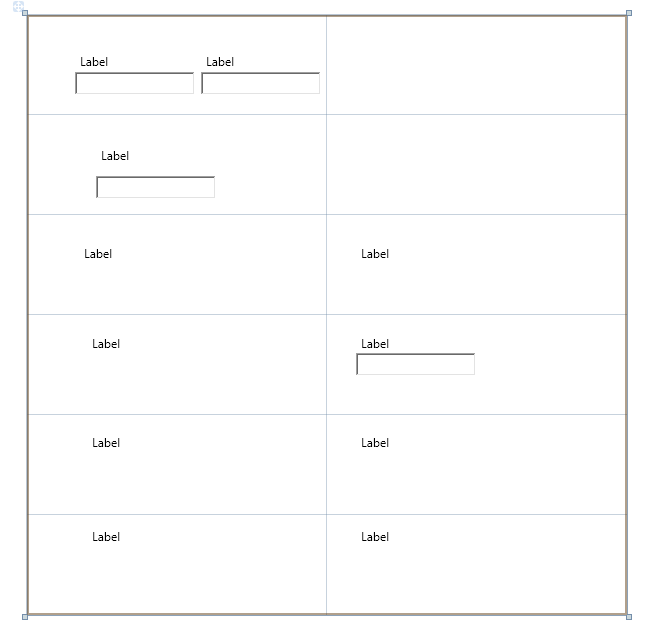
<ColumnDefinition />

</Grid.ColumnDefinitions>

we get a designer that looks like this:



Now, let’s drag and drop the textboxes and labels onto the form from the toolbox so that we have something like this:



Notice that in the XAML editor part of the designer the XAML is being added for you. Now let’s clean up the XAML a bit and provide labels by changing the Label element content. For example from

<Label Grid.Row="2" Margin="52,25,128,47" Name="label4">Label</Label>

To:

<Label Grid.Row="2" Margin="52,25,128,47" Name="label4">Assigned to:</Label>

Just to keep things nice and clean let’s also change the Name attribute to something more meaningful and change the Margin values so that everything lines up nicely. In the end you should end up with something like this: In the end you should end up with something like this:

After cleaning up the XAML and getting everything positioned just right you should have something 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>

<TextBox Grid.Row="0" Grid.Column="0" Margin="8,30,85,0" Height="28" VerticalAlignment="Top" Name="txtTitle" Grid.ColumnSpan="2" />

<Label Grid.Row="0" Grid.Column="0" Margin="0,8,8,0" Height="28" VerticalAlignment="Top" HorizontalAlignment="Right" Name="lblID" Grid.ColumnSpan="2" Width="72">ID:</Label>

<TextBox Grid.Row="0" Grid.Column="0" Margin="0,30,8,0" Height="28" VerticalAlignment="Top" HorizontalAlignment="Right" Name="txtID" Grid.ColumnSpan="2" Width="70" />

<Label Grid.Row="1" Grid.Column="0" Margin="8,8,0,0" Height="28" VerticalAlignment="Top" Name="lblDescription">Description:</Label>

<TextBox Grid.Row="1" Grid.Column="0" Margin="8,30,8,0" VerticalAlignment="Top" Name="txtDescription" Grid.ColumnSpan="2" TextWrapping="Wrap" MinLines="4" />

<Label Grid.Row="2" Grid.Column="0" Margin="8,8,0,0" Height="28" VerticalAlignment="Top" HorizontalAlignment="Left" Name="lblAssgignedToUser">Assigned to:</Label>

<Label Grid.Row="2" Grid.Column="1" Margin="8,8,0,0" Height="28" VerticalAlignment="Top" HorizontalAlignment="Left" Name="lblClassiication">Classification:</Label>

<Label Grid.Row="3" Grid.Column="0" Margin="8,8,0,0" Height="28" VerticalAlignment="Top" HorizontalAlignment="Left" Name="lblAffectedUser">Affected user:</Label>

<Label Grid.Row="3" Grid.Column="1" Margin="8,8,0,0" Height="28" VerticalAlignment="Top" HorizontalAlignment="Left" Name="lblAlternateContactInfo">Alternate contact info:</Label>

<TextBox Grid.Row="3" Grid.Column="1" Margin="8,30,8,0" Height="28" VerticalAlignment="Top" Name="txtAlternateContactInfo"/>

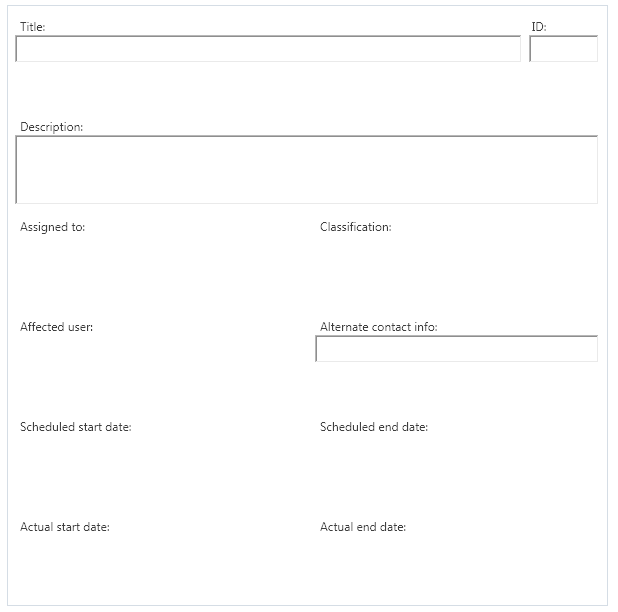
<Label Grid.Row="4" Grid.Column="0" Margin="8,8,0,0" Height="28" VerticalAlignment="Top" HorizontalAlignment="Left" Name="lblScheduledStartDate">Scheduled start date:</Label>

<Label Grid.Row="4" Grid.Column="1" Margin="8,8,0,0" Height="28" VerticalAlignment="Top" HorizontalAlignment="Left" Name="lblScheduledEndDate">Scheduled end date:</Label>

<Label Grid.Row="5" Grid.Column="0" Margin="8,8,0,0" Height="28" VerticalAlignment="Top" HorizontalAlignment="Left" Name="lblActualStartDate">Actual start date:</Label>

<Label Grid.Row="5" Grid.Column="1" Margin="8,8,0,0" Height="28" VerticalAlignment="Top" HorizontalAlignment="Left" Name="lblActualEndDate">Actual end date:</Label>

Which looks like this:



Now, let’s add the binding to each of the TextBox controls. It’s really easy – you just add an attribute to each textbox like this:

<TextBox Grid.Row="0" Grid.Column="0" Margin="8,30,85,0" Height="28" VerticalAlignment="Top" Name="txtTitle" Grid.ColumnSpan="2" Text="{Binding Title}"/>

Repeat for the other three TextBoxes changing Title for the appropriate property name – Id, Description or ContactMethod.

<TextBox Grid.Row="0" Grid.Column="0" Margin="0,30,8,0" Height="28" VerticalAlignment="Top" HorizontalAlignment="Right" Name="txtID" Grid.ColumnSpan="2" Width="70" Text="{Binding Id}"/>

<TextBox Grid.Row="1" Grid.Column="0" Margin="8,30,8,0" VerticalAlignment="Top" Name="txtDescription" Grid.ColumnSpan="2" TextWrapping="Wrap" MinLines="4" Text="{Binding Description}"/>

<TextBox Grid.Row="3" Grid.Column="1" Margin="8,30,8,0" Height="28" VerticalAlignment="Top" Name="txtAlternateContactInfo" Text="{Binding ContactMethod}"/>

Now we are ready to add in the custom controls. Let’s start with the DateTime controls. First, add a Reference to the WPFToolkit.dll to your project. The DLL is located in the directory %ProgramFiles%\Microsoft System Center\Service Manager 2010. Then add the namespace declaration at the top of the XAML:

<UserControl x:Class="Microsoft.Demo.ServiceRequest.ServiceRequest"

xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"

xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"

xmlns:wpfToolKit="clr-namespace:Microsoft.Windows.Controls;assembly=wpfToolKit"

Height="600" Width="600">

Then add the control in the right spots like this:

<wpfToolKit:DatePicker Grid.Row="4" Grid.Column="0" Margin="8,30,0,0" Height="28" VerticalAlignment="Top" Name="dtScheduledStartDate" SelectedDateFormat="FullDateTime" SelectedDate="{Binding ScheduledStartDate, Mode=TwoWay}"/>

Do this for each of the four DatePicker controls changing the Grid.Row, Grid.Column, Name, and Binding information as appropriate to ge the positioning and binding correct.

Now we can add the ListPicker custom control. Add a reference to Microsoft.EnterpriseManagement.UI.SMControls and Microsoft.EnterpriseManagement.UI.Foundation by pointing to the DLLs located in the directory %ProgramFiles%\Microsoft System Center\Service Manager 2010.

<UserControl x:Class="Microsoft.Demo.ServiceRequest.ServiceRequest"

xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"

xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"

xmlns:wpfToolKit="clr-namespace:Microsoft.Windows.Controls;assembly=wpfToolKit"

xmlns:smcontrols="clr-namespace:Microsoft.EnterpriseManagement.UI.WpfControls;assembly=Microsoft.EnterpriseManagement.UI.SmControls"

Height="600" Width="600">

Add the ListPicker control in the appropriate place like this:

1. First add a new namespace called ‘local’

<UserControl x:Class="Microsoft.Demo.ServiceRequest.ServiceRequest"

xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"

xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"

xmlns:wpfToolKit="clr-namespace:Microsoft.Windows.Controls;assembly=wpfToolKit"

xmlns:smcontrols="clr-namespace:Microsoft.EnterpriseManagement.UI.WpfControls;assembly=Microsoft.EnterpriseManagement.UI.SmControls"

xmlns:local="clr-namespace:Microsoft.Demo.ServiceRequest"

Height="600" Width="600">

1. Create a new Guid member constant of the ServiceRequest class in the C# code behind

public partial class ServiceRequest : UserControl

{

public static Guid guidClassificationEnumRoot = new Guid("4A7368B0-10F9-E93E-F399-0E250E05F108");

public ServiceRequest()

{

InitializeComponent();

}

}

Where did I get the Guid? I queried the database table EnumType and got the root enum ID off the EnumTypeId column for the ServiceRequestClassificationEnum.

1. Add the XAML in the right spot. Notice the binding to our new public static Guid.

<smcontrols:ListPicker Grid.Row="2" Grid.Column="1" Margin="8,30,8,0" Name="lpClassification" VerticalAlignment="Top" ParentCategoryId="{Binding Source={x:Static local:Microsoft.Demo.ServiceRequest.guidClassificationEnumRoot}, Mode=OneWay}" SelectedItem="{Binding Path=Classification, UpdateSourceTrigger=PropertyChanged, Mode=TwoWay}"/>

Next we need to add the two user picker controls. **Notice how the Binding Path maps to the Alias of the Component defined in the TypeProjection above in the MP.**

<smcontrols:UserPicker Grid.Row="2" Grid.Column="0" Margin="8,30,8,0" User="{Binding Path=AssignedTo, Mode=TwoWay}" VerticalAlignment="Top" Name="upAssignedTo"/>

<smcontrols:UserPicker Grid.Row="2" Grid.Column="0" Margin="8,30,8,0" User="{Binding Path=AffectedUser, Mode=TwoWay}" VerticalAlignment="Top" Name="upAffectedUser"/>

Now let’s add the list box control for the Affected CIs to the bottom grid row:

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

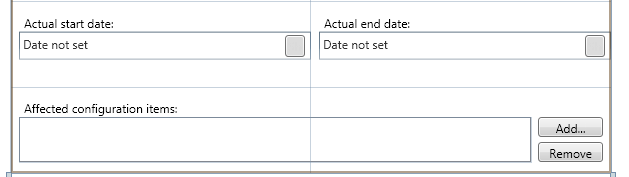
<ListView Grid.Row="6" Grid.Column="0" Margin="8,30,80,8" VerticalAlignment="Top" Name="lvAffectedCIs" Grid.ColumnSpan="2" MinHeight="45">

</ListView>

<Button Grid.Row="6" Grid.Column="0" Margin="0,30,8,0" VerticalAlignment="Top" HorizontalAlignment="Right" Name="btnAdd" Grid.ColumnSpan="2" Height="20" Width="65">Add...</Button>

<Button Grid.Row="6" Grid.Column="0" Margin="0,55,8,0" VerticalAlignment="Top" HorizontalAlignment="Right" Name="btnRemove" Grid.ColumnSpan="2" Height="20" Width="65">Remove</Button>

So – it looks like this now:



Now let’s add the Binding and Column information. The binding information is highlighted. Notice how the ItemsSource is bound to the same name as the Alias for the TypeProjection\Component – “RelatesToCI”. And then the column binds to the property of the related CI – DisplayName.

<ListView Grid.Row="6" Grid.Column="0" Margin="8,30,80,8" VerticalAlignment="Top" Name="lvAffectedCIs" Grid.ColumnSpan="2" MinHeight="45" ItemsSource="{Binding Path=RelatesToCI, UpdateSourceTrigger=PropertyChanged, Mode=TwoWay}">

<ListView.View>

<GridView>

<GridViewColumn Header="Display Name"

DisplayMemberBinding="{Binding DisplayName}"

Width="Auto" />

</GridView>

</ListView.View>

</ListView>

Now, let’s add the button behavior.

using System.Collections.ObjectModel;

using System.Collections;

Double click on each of the buttons to create the Click event handler in the code behind .cs file.

private void btnAdd\_Click(object sender, RoutedEventArgs e)

{

}

private void btnRemove\_Click(object sender, RoutedEventArgs e)

{

}

For the Add event handler we need some code like this:

private void btnAdd\_Click(object sender, RoutedEventArgs e)

{

AddItemToListView(this.lvAffectedCIs, new Guid("62F0BE9F-ECEA-E73C-F00D-3DD78A7422FC"));

}

For the Remove event handler we need some code like this:

private void btnRemove\_Click(object sender, RoutedEventArgs e)

{

RemoveItemFromWorkItemListView(this.lvAffectedCIs);

}

Then we need to actually add these functions to our code as follows:

internal static void AddItemToListView(ListView listView, Guid classId)

{

if (listView != null && listView.Items != null)

{

/\* NOTE: The use of the IDataItem and InstancePickerDialog interfaces here is not supported/documented.

\* This interface may change in the future and no migration path is guaranteed by Microsoft.

\*/

InstancePickerDialog ciPicker = new InstancePickerDialog();

ciPicker.ClassId = classId;

ciPicker.SelectionMode = SelectionMode.Multiple;

if (listView.Items.Count > 0)

{

ciPicker.SetPickedInstances((Collection<IDataItem>)listView.ItemsSource);

}

bool? result = ciPicker.ShowDialog();

if (result != null && result == true)

{

Collection<IDataItem> items = listView.ItemsSource as Collection<IDataItem>;

foreach (IDataItem item in ciPicker.RemovedInstances)

items.Remove(item);

foreach (IDataItem item in ciPicker.PickedInstances)

if (!items.Contains(item))

items.Add(item);

}

}

}

internal static void RemoveItemFromWorkItemListView(ListView listView)

{

if (listView.ItemsSource == null ||

listView.SelectedItems == null ||

listView.SelectedItems.Count == 0)

{

return;

}

ItemCollection items = listView.ItemsSource as ItemCollection;

if (items != null)

{

foreach (object obj in new ArrayList(listView.SelectedItems))

{

/\* NOTE: The use of the IDataItem interface here is not supported/documented.

\* This interface may change in the future and no migration path is guaranteed

by Microsoft.

\*/

items.Remove(obj as IDataItem);

}

}

}

The last thing we need to do is put in a bit of code that takes the service request title and ID and combines them together to populate the DisplayName property. The DisplayName property is the name of the object that is shown all over the place in the user interface like in ‘Related Items’ lists in forms, in search results, etc. The reason we want to combine the ID and the Title into the DisplayName property is so that somebody can run a simple search for either the ID or a keyword in the title.

To do this, first we need to add an Event Handler for the \_Loaded event like this:

<UserControl x:Class="Microsoft.Demo.ServiceRequest.ServiceRequest"

xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"

xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"

xmlns:wpfToolKit="clr-namespace:Microsoft.Windows.Controls;assembly=wpfToolKit"

xmlns:smcontrols="clr-namespace:Microsoft.EnterpriseManagement.UI.WpfControls;assembly=Microsoft.EnterpriseManagement.UI.SmControls"

xmlns:local="clr-namespace:Microsoft.Demo.ServiceRequest"

Height="600" Width="600"

Background="White"

Loaded="UserControl\_Loaded"

>

And then in the C# code behind we add the handler like this:

private void UserControl\_Loaded(object sender, RoutedEventArgs e)

{

this.AddHandler(FormEvents.PreviewSubmitEvent, new EventHandler<PreviewFormCommandEventArgs>(this.OnPreviewSubmit));

}

What this handler does is add another handler for the PreviewSubmitEvent which is a custom event from the form host that is triggered with the user clicks the OK button. It gives you a chance to make any changes to the data or do any validation prior to the data being submitted to the Data Access Service. Here is the code for the OnPreviewSubmit handler:

private void OnPreviewSubmit(object sender, PreviewFormCommandEventArgs e)

{

/\* NOTE: The use of the IDataItem interface here is not supported/documented.

\* This interface may change in the future and no migration path is guaranteed by Microsoft.

\*/

IDataItem itemServiceRequest = this.DataContext as IDataItem;

itemServiceRequest["DisplayName"] = itemServiceRequest["Id"] + " - " + itemServiceRequest["Title"];

}

That’s it! Now build your form.

Then we need to add the Form to the management pack using a Form element and a Resource element like this:

<Presentation>

<Forms>

<Form ID="Form.ServiceRequest"

Accessibility="Public"

Target="TypeProjection.ServiceRequest"

Assembly="Assembly.ServiceRequest"

TypeName="Microsoft.Demo.ServiceRequest.ServiceRequest">

<Category>Form</Category>

</Form>

</Forms>

</Presentation>

…

<Resources>

<Assembly ID="Assembly.ServiceRequest"

Accessibility="Public"

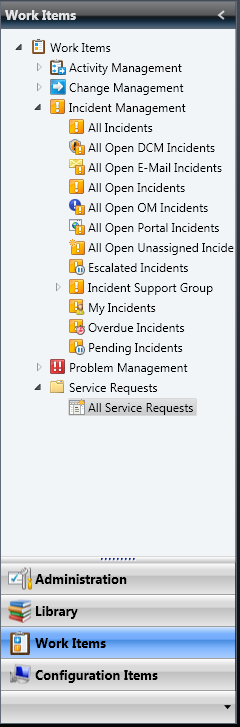
FileName="Microsoft.Demo.ServiceRequest.dll"

HasNullStream="false"   
QualifiedName="Microsoft.Demo.ServiceRequest, Version=1.0.0.0" />

</Resources>

And then we need to bundle the MP .xml file and form .dll together into a [Management Pack Bundle](http://blogs.technet.com/servicemanager/archive/2009/09/04/introducing-management-pack-bundles.aspx).

Import the management pack bundle and you should be able to see a new folder and view in the Work Items workspace.



Our final form looks like this:

