Creating the CSV Connector

# Legal Disclaimer

Copyright(c) Microsoft. All rights reserved.

This code is licensed under the Microsoft Public License.

http://www.microsoft.com/opensource/licenses.mspx

THIS CODE IS PROVIDED \*AS IS\* WITHOUT WARRANTY OF ANY KIND,

EITHER EXPRESSED OR IMPLIED, INCLUDING ANY IMPLIED WARRANTIES

OF FITNESS FOR A PARITCULAR PURPOSE, MERCHANTABILITY, OR

NON-INFRINGEMENT.

**This project and document is provided as an *example* of how to implement a connector. While it has been tested somewhat, it is not a supported component of the Service Manager product.**

# Introduction

This document describes the different components of the implementation of a CSV Connector including how to add the CSV Connector administration user experience into the Connectors view in the Service Manager console. This example shows how to create, edit, delete, and enable/disable connectors using task handlers and the wizard/property sheet framework.

The end result is the added ability to create CSV connectors that periodically import data contained in a .csv file into Service Manager to create new objects in the CMDB or update them.

As this is an example, the code is not highly optimized or optimally factored. It is written in a way that is intended to be easy to understand and reuse.

# Management Pack

## Manifest

The manifest defines a new MP – Microsoft.Demo.Connectors. For now it contains only the CSV Connector. I’ll add other examples over time to this MP. The manifest also includes references to some other MPs that this MP will depend on.

<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.Connectors</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="LinkingFramework.Library">

<ID>ServiceManager.LinkingFramework.Library</ID>

<Version>7.0.5244.0</Version>

<PublicKeyToken>31bf3856ad364e35</PublicKeyToken>

</Reference>

<Reference Alias="ServiceManager.UI.Administration">

<ID>Microsoft.EnterpriseManagement.ServiceManager.UI.Administration</ID>

<Version>7.0.5244.0</Version>

<PublicKeyToken>31bf3856ad364e35</PublicKeyToken>

</Reference>

<Reference Alias="Console">

<ID>Microsoft.EnterpriseManagement.ServiceManager.UI.Console</ID>

<Version>7.0.5244.0</Version>

<PublicKeyToken>31bf3856ad364e35</PublicKeyToken>

</Reference>

<Reference Alias="SystemCenter">

<ID>Microsoft.SystemCenter.Library</ID>

<Version>7.0.5244.0</Version>

<PublicKeyToken>31bf3856ad364e35</PublicKeyToken>

</Reference>

<Reference Alias="Windows">

<ID>Microsoft.Windows.Library</ID>

<Version>7.0.5244.0</Version>

<PublicKeyToken>31bf3856ad364e35</PublicKeyToken>

</Reference>

</References>

</Manifest>

## CSV Connector Class

The CSV Connector class defined in this management pack has the following derivation:

**System.Entity** [System.Library MP]

.DisplayName

**Microsoft.SystemCenter.Connector** [SystemCenter.Library MP]

.Id

.Description

.Name

.DiscoveryDataIsManaged

.DiscoveryDataIsShared

**System.LinkingFramework.DataSource** [ServiceManager.LinkingFramework.Library]

.DataProviderName

.DataProviderDisplayName

.Enabled

.SyncTime

.SolutionName

.ReaderProfileName

.Reserved

.ImpersonationEnabled

.SyncType

.SyncInterval

.SyncNow

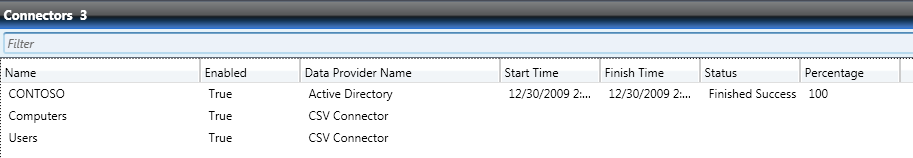
**Microsoft.Demo.Connectors.CSVConnector** [Microsoft.Demo.Connectors]

.DataFilePath

.MappingFilePath

.NumberMinutes

If you look at the Connectors view in the UI, the properties map to the columns as follows:



* Name = **System.Entity.DisplayName**
* Enabled = **System.LinkingFramework.DataSource.Enabled**
* Data Provider Name = **System.LinkingFramework.DataSource.DataProviderDisplayName**

We are only going to actually use the highlighted properties in our example. The other properties are either intended to be used primarily used for the AD and SCCM connectors.

Before we go on, I want to provide just a quick note on the two properties - DiscoveryDataIsManaged and DiscoveryDataIsShared. Even though we are not going to use these for the CSV connector, tthese two properties have significant meaning when building different types of connectors so we’ll explain them here briefly.

**DiscoveryDataIsManaged**

In System Center Operations Manager, everything except objects that derive from System.Device (e.g. Windows.Computer) are automatically managed (aka monitored), and can be used as a target of a workflow. It is the discovery source used to discover (i.e. insert) the System.Device instance which determines whether it is managed.

There are two system discovery sources which allow System.Device instances to be managed, along with any Connector which has DiscoveryDataIsManaged = True.

In both SCOM and SCSM, this is determined by the Management Configuration Service Data Access Layer. SCSM and SCOM have slightly different DALs and therefore slightly different behavior). In SCSM, we have an [explicit list of classes which can be targets](http://blogs.technet.com/servicemanager/archive/2009/08/21/targeting-workflows-in-service-manager.aspx), so DiscoveryDataIsManaged has no use in SCSM.

Default is DiscoveryDataIsManaged = NULL (or False).

**DiscoveryDataIsShared**

In SCSM, we need the ability to allow one DiscoverySource/Connector to delete an object regardless of whether any other DiscoverySource/Connector had already inserted or updated that object (i.e. the object’s reference count was > 1).  It is essentially sharing a single reference count with all other DiscoverySources/Connectors.

A Connector with DiscoveryDataIsShared = True can delete an object regardless of whether or not other discovery sources/connectors have updated it.

Default is DiscoveryDataIsShared = NULL (or False).

The new class – Microsoft.Demo.Connector.CSVConnector is defined as:

<ClassType ID="Microsoft.Demo.Connectors.CSVConnector"

Accessibility="Public"

Abstract="false"

Base="LinkingFramework.Library!System.LinkingFramework.DataSource"

Hosted="false"

Singleton="false"

Extension="false">

<Property ID="DataFilePath"

Type="string"

AutoIncrement="false"

Key="false"

CaseSensitive="false"

MaxLength="256"   
MinLength="0"

Required="false" />

<Property ID="MappingFilePath"

Type="string"

AutoIncrement="false"

Key="false"

CaseSensitive="false"

MaxLength="256"

MinLength="0"

Required="false" />

<Property ID="NumberMinutes"

Type="string"

AutoIncrement="false"

Key="false"

CaseSensitive="false"

MaxLength="256"

MinLength="0"

Required="false" />

</ClassType>

The class is Singleton="false" and Abstract="false" so that we can create multiple objects of this class, each one representing a “connector”. Each object of this class will show up as a row in the Connectors view and can be independently managed – enable/disable, delete, change properties.

The <Property ID="DataFilePath" will store the path to the .csv data file that we are going to import.

The <Property ID="MappingFilePath"will store the path to the .xml mapping file that we will use to format the data for insertion.

These paths can be a local path on the management server or UNC path to a remote server.

The <Property ID="NumberMinutes" will store how frequently the connector should run in number of minutes.

## Module Type

The module type declared in the MP was generated by using the Authoring Console. I created a very simple workflow with just one PowerShell activity in it.

<ModuleTypes>

<WriteActionModuleType ID="CSVConnectorWorkflow.MT" Accessibility="Public" Batching="false" RunAs="SystemCenter!Microsoft.SystemCenter.DatabaseWriteActionAccount">

<Configuration>

<IncludeSchemaTypes>

<SchemaType>Windows!Microsoft.Windows.PowerShellSchema</SchemaType>

</IncludeSchemaTypes>

<xsd:element name="DataFilePath" type="xsd:string" xmlns:xsd="http://www.w3.org/2001/XMLSchema" />

<xsd:element name="FormatFilePath" type="xsd:string" xmlns:xsd="http://www.w3.org/2001/XMLSchema" />

</Configuration>

<OverrideableParameters>

<OverrideableParameter ID="DataFilePath" Selector="$Config/DataFilePath$" ParameterType="string" />

<OverrideableParameter ID="FormatFilePath" Selector="$Config/FormatFilePath$" ParameterType="string" />

</OverrideableParameters>

<ModuleImplementation Isolation="Any">

<Composite>

<MemberModules>

<WriteAction ID="CSVConnectorWorkflow.PSWA" TypeID="Windows!Microsoft.Windows.PowerShellWriteAction">

<ScriptName>DoWork.ps1</ScriptName>

<ScriptBody>

param ( [string]$DataFilePath,[string]$FormatFilePath )

Add-PSSnapin SMCmdletSnapin ; Import-SCSMInstance -DataFileName $DataFilePath -FormatFileName $FormatFilePath

</ScriptBody>

<SnapIns></SnapIns>

<Parameters>

<Parameter>

<Name>DataFilePath</Name>

<Value>$Config/DataFilePath$</Value>

</Parameter>

<Parameter>

<Name>FormatFilePath</Name>

<Value>$Config/FormatFilePath$</Value>

</Parameter>

</Parameters>

<TimeoutSeconds>300</TimeoutSeconds>

<StrictErrorHandling>true</StrictErrorHandling>

<SerializationDepth>3</SerializationDepth>

</WriteAction>

</MemberModules>

<Composition>

<Node ID="CSVConnectorWorkflow.PSWA" />

</Composition>

</Composite>

</ModuleImplementation>

<InputType>System!System.BaseData</InputType>

</WriteActionModuleType>

</ModuleTypes>

The PowerShell script simply takes the input of the mapping and data file paths for the connector and passes those to the Import-SCSMObject cmdlet provided out of the box with Service Manager.

## Categories

We need to add just a couple of categories to this MP:

<Categories>

<Category

ID="Category.DoubleClickEditCSVConnector"

Target="Task.EditCSVConnector"

Value="Console!Microsoft.EnterpriseManagement.ServiceManager.UI.Console.DoubleClickTask" />

<Category

ID="Category.SCSM.MP"

Value="Console!Microsoft.EnterpriseManagement.ServiceManager.ManagementPack">

<ManagementPackName>Microsoft.Demo.Connectors</ManagementPackName>

<ManagementPackVersion>1.0.0.0</ManagementPackVersion>

</Category>

</Categories>

The first category tells Service Manager that the Edit task is going to be the task that is run on a double click event in the Connectors view when a CSV Connector is double clicked on.

The second category is used to categorize this management pack as a management pack intended for use in Service Manager. This will ensure that the consle tasks defined in this MP will be displayed in the Service Manager console. Some MPs designed for Operations Manager originally also have Console Tasks in them which may not make sense to display in Service Manager. In that case, because those MPs do not have this special category, the tasks in those MPs will not be displayed in the Service Manager console.

## Runtime Task

The task defined in the management pack is the one that actually runs the PowerShell script. It was auto-generated by the authoring console so there isn’t really much to look at here.

<Task ID="CSVConnectorWorkflow.PowerShellScript.ae4f230f\_b48e\_40d1\_987e\_10b21964932e"

Accessibility="Public"

Enabled="true"

Target="Windows!Microsoft.Windows.Computer"

Timeout="300"

Remotable="true">

<Category>Notification</Category>

<WriteAction ID="CSVConnectorWorkflow.WA" TypeID="CSVConnectorWorkflow.MT">

<DataFilePath></DataFilePath>

<FormatFilePath></FormatFilePath>

</WriteAction>

</Task>

## Console Tasks

The first console task is the Create task:

<ConsoleTask ID="Task.CreateCSVConnector"

Accessibility="Public"

Enabled="true"

Target="ServiceManager.UI.Administration!Microsoft.EnterpriseManagement.ServiceManager.UI.Administration.Task.CreateConnector"

RequireOutput="false">

<Assembly>Console!SdkDataAccessAssembly</Assembly>

<Handler>Microsoft.EnterpriseManagement.UI.SdkDataAccess.ConsoleTaskHandler</Handler>

<Parameters>

<Argument Name="Assembly">Microsoft.Demo.Connectors</Argument>

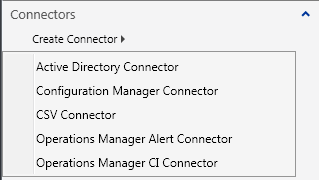
<Argument Name="Type">Microsoft.Demo.Connectors.CSV.CSVConnector</Argument>

<Argument>Create</Argument>

</Parameters>

</ConsoleTask>

A few important things to point out about this console task. Typically a Console Task will target a ClassType element. In this case, it targets a ConsoleTask element in the Administration MP. This is how cascading menus are built in Service Manager. In this case the Create task will now appear on the Create > flyout menu in the Connectors view because it targets the CreateConnector console task in the Administration MP.



The next important thing to point out is the Task Handler. For Service Manager 2010 we only support this one method of using a Task Handler which should provide plenty of flexibility. Basically, all you need to do is provide the Assembly and Class that implements your handler as arguments as above.

We’ll get into this more later, but I just want to point this out now quickly. The handler I’ve created for this console task is in an assembly called Microsoft.Demo.Connectors.dll. There is a class defined in it that looks like this:

namespace Microsoft.Demo.Connectors.CSV

{

public class CSVConnector : ConsoleCommand

{

public CSVConnector()

{

}

public override void ExecuteCommand(IList<NavigationModelNodeBase> nodes, NavigationModelNodeTask task, ICollection<string> parameters)

{

if(parameters.Contains("Create"))

{

... some code ...

You can see that the “Type” attribute points to the namespace + class of the code that I want to call. My class ‘CSVConnector’ derives from the ConsoleCommand class which is located in the Microsoft.EnterpriseManagement.UI.SdkDataAccess assembly and requires a using statement like this to use:

using Microsoft.EnterpriseManagement.UI.SdkDataAccess;

Then I had to implement an override method for ExecuteCommand. ExecuteCommand will give me the selected node (which row is selected), which console task was clicked, and the parameters collection.

I could have implemented separate classes for each of the different console tasks that we are going to use, but instead I just defined one class and use the passed in parameters to control the flow. For example, above you can see that there is an additional <Argument>Create</Argument> that is passed to the ConsoleCommand in the parameters list. I can then use that information to control code flow like this:

if(parameters.Contains("Create"))

{

... some code ...

The rest of the console tasks are similar. They all call the same task handler and just pass different strings for the <Argument>. The only other difference between the create task and the rest of the tasks is that the rest of the tasks target the Microsoft.Demo.Connectors.CSVConnector class so that they are only displayed when a particular CSV Connector object is selected in the UI.

<ConsoleTask ID="Task.EditCSVConnector"

Accessibility="Public"

Enabled="true"

Target="Microsoft.Demo.Connectors.CSVConnector"

RequireOutput="false">

<Assembly>Console!SdkDataAccessAssembly</Assembly>

<Handler>Microsoft.EnterpriseManagement.UI.SdkDataAccess.ConsoleTaskHandler</Handler>

<Parameters>

<Argument Name="Assembly">Microsoft.Demo.Connectors</Argument>

<Argument Name="Type">Microsoft.Demo.Connectors.CSV.CSVConnector</Argument>

<Argument>Edit</Argument>

</Parameters>

</ConsoleTask>

<ConsoleTask ID="Task.DeleteCSVConnector"

Accessibility="Public"

Enabled="true"

Target="Microsoft.Demo.Connectors.CSVConnector"

RequireOutput="false">

<Assembly>Console!SdkDataAccessAssembly</Assembly>

<Handler>Microsoft.EnterpriseManagement.UI.SdkDataAccess.ConsoleTaskHandler</Handler>

<Parameters>

<Argument Name="Assembly">Microsoft.Demo.Connectors</Argument>

<Argument Name="Type">Microsoft.Demo.Connectors.CSV.CSVConnector</Argument>

<Argument>Delete</Argument>

</Parameters>

</ConsoleTask>

<ConsoleTask ID="Task.EnableCSVConnector"

Accessibility="Public"

Enabled="true"

Target="Microsoft.Demo.Connectors.CSVConnector"

RequireOutput="false">

<Assembly>Console!SdkDataAccessAssembly</Assembly>

<Handler>Microsoft.EnterpriseManagement.UI.SdkDataAccess.ConsoleTaskHandler</Handler>

<Parameters>

<Argument Name="Assembly">Microsoft.Demo.Connectors</Argument>

<Argument Name="Type">Microsoft.Demo.Connectors.CSV.CSVConnector</Argument>

<Argument>Enable</Argument>

</Parameters>

</ConsoleTask>

<ConsoleTask ID="Task.DisableCSVConnector"

Accessibility="Public"

Enabled="true"

Target="Microsoft.Demo.Connectors.CSVConnector"

RequireOutput="false">

<Assembly>Console!SdkDataAccessAssembly</Assembly>

<Handler>Microsoft.EnterpriseManagement.UI.SdkDataAccess.ConsoleTaskHandler</Handler>

<Parameters>

<Argument Name="Assembly">Microsoft.Demo.Connectors</Argument>

<Argument Name="Type">Microsoft.Demo.Connectors.CSV.CSVConnector</Argument>

<Argument>Disable</Argument>

</Parameters>

</ConsoleTask>

</ConsoleTasks>

And then in the task handler I just handle the different cases like this:

if(parameters.Contains("Create"))

{

...some code...

}

else if (parameters.Contains("Edit"))

{

...some code...

}

else if (parameters.Contains("Delete") || parameters.Contains("Disable") || parameters.Contains("Enable"))

{

...some code...

}

## Image References

The image references make it easy to have a consistent look and feel for our console tasks. By pointing to the same images used by the Service Manager product itself the implementation of our connector looks seamless.

<ImageReferences>

<ImageReference ElementID="Task.EditCSVConnector" ImageID="Console!Microsoft.EnterpriseManagement.ServiceManager.UI.Console.Image.Properties" />

<ImageReference ElementID="Task.DeleteCSVConnector" ImageID="Console!Microsoft.EnterpriseManagement.ServiceManager.UI.Console.Image.Remove" />

<ImageReference ElementID="Task.EnableCSVConnector" ImageID="Console!Microsoft.EnterpriseManagement.ServiceManager.UI.Console.Image.Enable" />

<ImageReference ElementID="Task.DisableCSVConnector" ImageID="Console!Microsoft.EnterpriseManagement.ServiceManager.UI.Console.Image.Disable" />

</ImageReferences>

Of course if we wanted to we could provide our own images for our console tasks, but for these kinds of standard tasks – edit, disable/enable, delete it really is better to reference the standard images. Also – that way if Microsoft updates those images in the future those updates will apply to this connector as well.

## Language Pack

Lastly, we get to the language pack section of the management pack. For more information on this, please see the blog post [Localizing Management Pack Content in Service Manager](http://blogs.technet.com/servicemanager/archive/2009/07/24/localizing-management-pack-content.aspx).

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

<DisplayStrings>

<DisplayString ElementID="Microsoft.Demo.Connectors.CSVConnector">

<Name>CSV Connector</Name>

</DisplayString>

<DisplayString ElementID="Microsoft.Demo.Connectors.CSVConnector" SubElementID="DataFilePath">

<Name>Data File Path</Name>

</DisplayString>

<DisplayString ElementID="Microsoft.Demo.Connectors.CSVConnector" SubElementID="MappingFilePath">

<Name>Mapping File Path</Name>

</DisplayString>

<DisplayString ElementID="Task.CreateCSVConnector">

<Name>CSV Connector</Name>

</DisplayString>

<DisplayString ElementID="Task.EditCSVConnector">

<Name>Properties</Name>

</DisplayString>

<DisplayString ElementID="Task.DeleteCSVConnector">

<Name>Delete</Name>

</DisplayString>

<DisplayString ElementID="Task.EnableCSVConnector">

<Name>Enable</Name>

</DisplayString>

<DisplayString ElementID="Task.DisableCSVConnector">

<Name>Disable</Name>

</DisplayString>

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

<Name>Microsoft.Demo.Connectors</Name>

<Description></Description>

</DisplayString>

</DisplayStrings>

</LanguagePack>

# The Task Handler & Wizard Framework

We’ve already discussed a bit how the ConsoleTasks defined in the management pack connect up to the code behind in the Task Handler. Let’s take a look at the code itself more deeply.

## The Create Task

First we’ll take a look at the task handler code for the Create task. At a high level what this task will do is create two things:

1. An object of the class Microsoft.Demo.Connectors.CSVConnector in the ServiceManager database. This object will store the DisplayName, ID, DataFilePath, MappingFilePath, and NumberMinutes properties. The DisplayName, DataFilePath, MappingFilePath, and NumberMinutes will be provided by the user using the wizard to input the data. The ID will be automatically generated.
2. A rule in the Microsoft.Demos.Connectors management pack in the ServiceManager database. The rule ID will be set to the same as the connector object ID to maintain a identity mapping between the connector object and the rule. The NumberMinutes value will be embedded into the Scheduler DataSourceModule configuration to tell the rule how often to run. The DataFilePath and MappingFilePath values will be embedded into the WriteActionModule configuration. For more information on rules/modules, please see these blog posts:

* [The System Center Platform in Service Manager Part 4: The System Center Management Service](http://blogs.technet.com/servicemanager/archive/2009/08/09/the-system-center-platform-in-service-manager-part-4-the-system-center-management-service.aspx)
* [The System Center Platform in Service Manager Part 4: The Management Service – Try It!](file:///C:\Users\Administrator\Documents\The%20System%20Center%20Platform%20in%20Service%20Manager%20Part%204:%20The%20Management%20Service%20–%20Try%20It!)

The object is stored in the database so that the connector can be represented in the Connectors view. The Connectors view shows all objects of all classes that derive from the **System.LinkingFramework.DataSource** class. The rule needs to be created because that is what makes the connector actually do something – namely run a Windows Workflow Foundation task which then submits a PowerShell script task.

Let’s take a look at the code that creates the object and the rule for the Create console task.

if(parameters.Contains("Create"))

{

WizardStory wizard = new WizardStory();

//Set the wizard icon and title bar

ResourceManager rm = new ResourceManager("Microsoft.Demo.Connectors.Resources",

typeof(Resources).Assembly);

Bitmap bitmap = (Bitmap)rm.GetObject("CSVConnector");

IntPtr ptr = bitmap.GetHbitmap();

BitmapSource bitmapsource = System.Windows.Interop.Imaging.CreateBitmapSourceFromHBitmap(ptr, IntPtr.Zero,

Int32Rect.Empty,BitmapSizeOptions.FromEmptyOptions());

wizard.StoryImage = bitmapsource;

wizard.WizardWindowTitle = "Create CSV Connector";

//Create the wizard data

WizardData data = new CSVConnectorWizardData();

wizard.WizardData = data;

//Add the wizard pages

wizard.AddLast(new WizardStep("Welcome", typeof(CSVConnectorWelcomePage),wizard.WizardData));

wizard.AddLast(new WizardStep("Configuration", typeof(CSVConnectorConfigurationPage), wizard.WizardData));

wizard.AddLast(new WizardStep("Summary", typeof(CSVConnectorSummaryPage), wizard.WizardData));

wizard.AddLast(new WizardStep("Results", typeof(CSVConnectorResultPage), wizard.WizardData));

//Create a wizard window and show it

WizardWindow wizardwindow = new WizardWindow(wizard);

// this is needed so that WinForms will pass messages on to the hosted WPF control

System.Windows.Forms.Integration.ElementHost.EnableModelessKeyboardInterop(wizardwindow);

wizardwindow.ShowDialog();

//Update the view when done with the wizard so that the new connector shows

if (data.WizardResult == WizardResult.Success)

{

RequestViewRefresh();

}

}

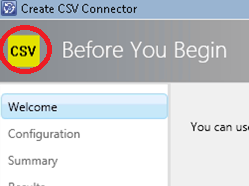
The first part of this code shows us how to instantiate a wizard:

WizardStory wizard = new WizardStory();

The WizardStory class requires a reference to Microsoft.EnterpriseManagement.UI.WpfWizardFramework and a using statement like this:

using Microsoft.EnterpriseManagement.UI.WpfWizardFramework;

Wizards can have an icon in the upper left corner like this:



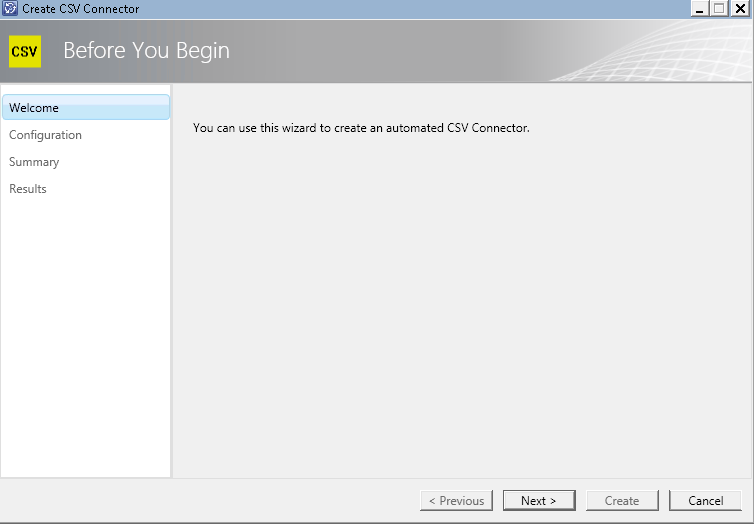
The icon size is 32x32 pixels. You can see an example of how I get the Bitmap image out of an embedded resource in the example code above. You can also see in this screenshot how you can set the wizard title bar using:

wizard.WizardWindowTitle = "Create CSV Connector";

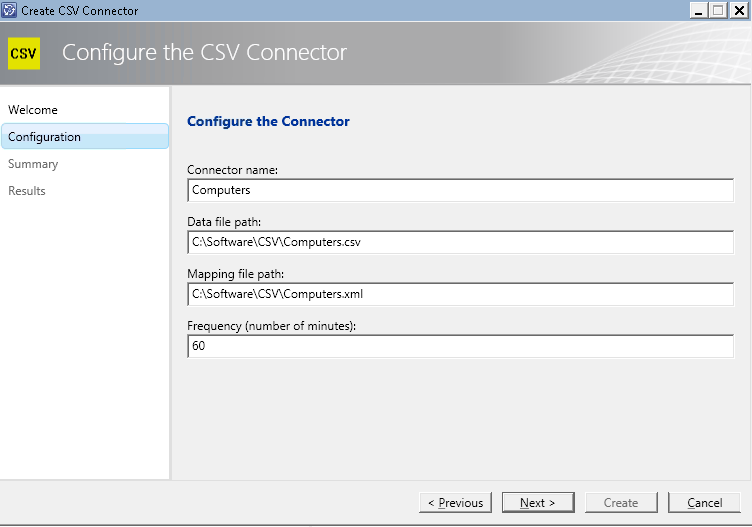
Next, we create the WizardData object and assign it to the WizardStory. That’s where the real magic happens actually. We’ll cover that a bit more in just a minute.

Next we add each of the WizardStep classes (i.e. “pages”)to the wizard, starting with the first page we want to show and ending in the last page because we are always using AddLast(). In this case, I am adding four wizard pages that look like this:

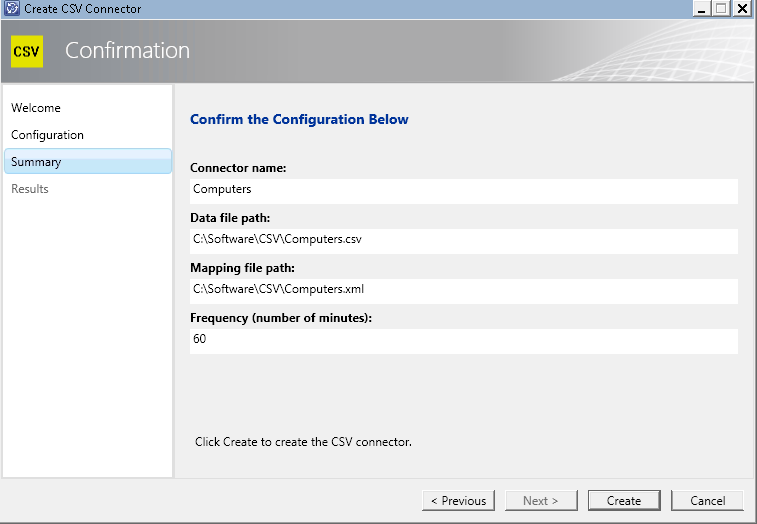
A Welcome page:



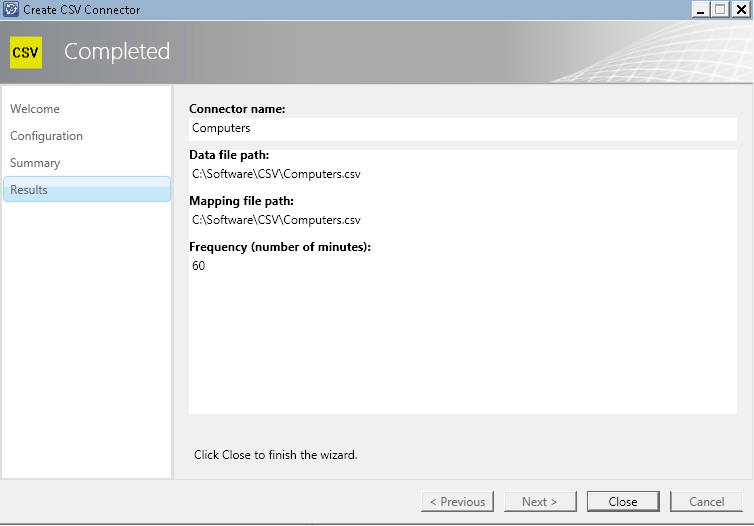
A Configuration page to get user input:



A Confirmation page to confirm the user’s input and initiate the actual creation by clicking on the Create button:



A Completion page:



The second parameter to AddLast() takes a typeof(WizardPage) argument. We’ll talk more about wizard pages [below](#_WizardPages).

Notice how the left hand navigation panel display text is set by the first string parameter passed in AddLast(). For example:

wizard.AddLast(new WizardStep("Welcome", typeof(CSVConnectorWelcomePage),wizard.WizardData));

The labels in the header area – “Before You Begin”, “Configure the CSV Connector”, “Confirmation”, “Completed” – are provided for you by the base classes that these WizardPage derive from. The behavior of the button bar (e.g. Back is disabled on the welcome page) is also controlled to a certain extent by the base WizardPage classes that you derive from. We’ll see that more in just a minute.

Now that we have created the connector WizardStory, we create a WizardWindow from that and display the wizard to the user. If the user is successful running the wizard we can call the RefreshView() method to update the view and show the new Connector to the user.

//Create a wizard window and show it

WizardWindow wizardwindow = new WizardWindow(wizard);

// this is needed so that WinForms will pass messages on to the hosted WPF control

System.Windows.Forms.Integration.ElementHost.EnableModelessKeyboardInterop(wizardwindow);

wizardwindow.ShowDialog();

//Update the view when done with the wizard so that the new connector shows

if (data.WizardResult == WizardResult.Success)

{

RequestViewRefresh();

}

Now let’s take a closer look at a couple of things – the WizardData class and the WizardPage classes.

## The CSVConnectorWizardData class

The class CSVConnectorWizardData derives from the WizardData class. It inherits a couple of key methods that we’ll see in just a minute.

class CSVConnectorWizardData: WizardData, INotifyPropertyChanged

First, we’ll define a few public properties on this class:

private String strDisplayName = String.Empty;

private String strDataFilePath = String.Empty;

private String strMappingFilePath = String.Empty;

private String strNumberMinutes = String.Empty;

private String strConnectorID = String.Empty;

private Guid guidEnterpriseManagementObjectID = Guid.Empty;

private String strErrorMessage = String.Empty;

public String DisplayName

{

get

{

return this.strDisplayName;

}

set

{

if (this.strDisplayName != value)

{

this.strDisplayName = value;

this.NotifyPropertyChanged("DisplayName");

}

}

}

... and so on ...

This class has two constructors – one used in the case where we are creating a new WizardData object from scratch with nothing in it and another one where we are going to fill up the properties based on the selected object in the UI (this is used for the Edit console task we will get to later).

internal CSVConnectorWizardData()

{

}

internal CSVConnectorWizardData(EnterpriseManagementObject emoCSVConnector)

{

... some code ...

}

Now, here is the first method we need to override – the AcceptChanges method.

public override void AcceptChanges(WizardMode wizardMode)

{

if (wizardMode == WizardMode.PropertySheet)

{

this.UpdateConnectorInstance();

}

else

{

try

{

this.CreateConnectorInstance();

this.WizardResult = WizardResult.Success;

}

catch (Exception ex)

{

this.WizardResult = WizardResult.Failed;

this.ErrorMessage = ex.ToString();

}

}

}

AcceptChanges() is called whenever the user clicks on the ‘Create’ button in the wizard or the ‘OK’ button when the wizard is being used in property page mode. The mode is passed as a parameter to AcceptChanges() so you know what context you are dealing with.

In this case when the user clicks on the Create button in the wizard it will call AcceptChanges() and since it is in wizard mode not PropertySheet mode, the method CreateConnectorInstance() will be called.

Now we get to the magical part! I’ve commented the code pretty well and it’s pretty self-explanatory so I’m not going to dissect this.

Especially look out for where the WizardData info is passed into the Connector object values so it can be stored in the ServiceManager database and where the WizardData info is passed into the Rule DataSourceModule and WriteActionModule configuration XML. I’ve highlighted those areas below.

The other thing to notice is how the connector ID is generated and then stored as part of the Connector object and set as the ID of the rule to create an association between them. I’ve highlighted that part too.

private void CreateConnectorInstance()

{

try

{

//Get the server name to connect to

String strServerName = Registry.GetValue("HKEY\_CURRENT\_USER\\Software\\Microsoft\\System

Center\\2010\\Service Manager\\Console\\User Settings", "SDKServiceMachine", "localhost").ToString();

//Conneect to the server

EnterpriseManagementGroup emg = new EnterpriseManagementGroup(strServerName);

//Get the System MP so we can get the system key token and version so we can get other MPs using that info

ManagementPack mpSystem =

emg.ManagementPacks.GetManagementPack(SystemManagementPack.System);

Version verSystemVersion = mpSystem.Version;

string strSystemKeyToken = mpSystem.KeyToken;

//Also get the System Center, Subscription, and Connector Demo MPs - we'll need things from those MPs later

ManagementPack mpSystemCenter =

emg.ManagementPacks.GetManagementPack(SystemManagementPack.SystemCenter);

ManagementPack mpSubscriptions =

emg.GetManagementPack("Microsoft.SystemCenter.Subscriptions", strSystemKeyToken, verSystemVersion);

ManagementPack mpConnectors = emg.GetManagementPack("Microsoft.Demo.Connectors", null,

new Version("1.0.0.0"));

//Get the CSVConnector class in the Connectors MP

ManagementPackClass classCSVConnector =

mpConnectors.GetClass("Microsoft.Demo.Connectors.CSVConnector");

//Create a new CreatableEnterpriseManagementObject. We'll set the properties on this and then post it to the database.

EnterpriseManagementObject cemoCSVConnector =

new CreatableEnterpriseManagementObject(emg, classCSVConnector);

//Set the property values...

//Sytem.Entity properties

cemoCSVConnector[classCSVConnector, "DisplayName"].Value = this.DisplayName;

//Microsoft.SystemCenter.Connector properties

//This is just a tricky way to create a unique ID which conforms to the syntax rules for MP element ID attribute values.

String strConnectorID = String.Format(CultureInfo.InvariantCulture, "{0}.{1}",

"CSVConnector", Guid.NewGuid().ToString("N"));

cemoCSVConnector[classCSVConnector, "Id"].Value = strConnectorID;

//System.LinkingFramework.DataSource properties

cemoCSVConnector[classCSVConnector, "DataProviderDisplayName"].Value = "CSV Connector";

cemoCSVConnector[classCSVConnector, "Enabled"].Value = true;

//Microsoft.Demo.Connectors.CSVConnector properties

cemoCSVConnector[classCSVConnector, "DataFilePath"].Value = this.DataFilePath;

cemoCSVConnector[classCSVConnector, "MappingFilePath"].Value = this.MappingFilePath;

cemoCSVConnector[classCSVConnector, "NumberMinutes"].Value = this.NumberMinutes;

//Create Connector instance

cemoCSVConnector.Commit();

//Now we need to create the CSV Connector rule...

//Get the Scheduler data source module type from the System MP and the Windows Workflow Task Write Action Module Type from the Subscription MP

ManagementPackDataSourceModuleType dsmtScheduler =

(ManagementPackDataSourceModuleType)mpSystem.GetModuleType("System.Scheduler");

ManagementPackWriteActionModuleType wamtWindowsWorkflowTaskWriteAction =

(ManagementPackWriteActionModuleType)mpSubscriptions.GetModuleType("Microsoft.EnterpriseManagement.SystemCenter.Subscription.WindowsWorkflowTaskWriteAction");

//Create a new rule for the CSV Connector in the Connectors MP. Set the name of this rule to be the same as the connector instance ID so there is a pairing between them

ManagementPackRule ruleCSVConnector = new ManagementPackRule(mpConnectors, strConnectorID);

//Set the target and other properties of the rule

ruleCSVConnector.Target =

mpSystemCenter.GetClass("Microsoft.SystemCenter.SubscriptionWorkflowTarget");

//Create a new Data Source Module in the new CSV Connector rule

ManagementPackDataSourceModule dsmSchedule = new

ManagementPackDataSourceModule(ruleCSVConnector, "DS1");

//Set the configuration of the data source rule. Pass in the frequency (number of minutes)

dsmSchedule.Configuration =

"<Scheduler>" +

"<SimpleReccuringSchedule>" +

"<Interval Unit=\"Minutes\">" + this.NumberMinutes + "</Interval>" +

"</SimpleReccuringSchedule>" +

"<ExcludeDates />" +

"</Scheduler>";

//Set the Schedule Data Source Module Type to the Simple Schedule Module Type from the System MP

dsmSchedule.TypeID = dsmtScheduler;

//Add the Scheduler Data Source to the Rule

ruleCSVConnector.DataSourceCollection.Add(dsmSchedule);

//Now repeat essentially the same process for the Write Action module...

//Create a new Write Action Module in the CSV Connector rule

ManagementPackWriteActionModule wamCSVConnector = new

ManagementPackWriteActionModule(ruleCSVConnector, "WA1");

//Set the Configuration XML

wamCSVConnector.Configuration =

"<Subscription>" +

"<WindowsWorkflowConfiguration>" +

//Specify the Windows Workflow Foundation workflow Assembly name here

"<AssemblyName>CSVConnectorWorkflow</AssemblyName>" +

//Specify the type name of the workflow to call in the assembly here:

"<WorkflowTypeName>WorkflowAuthoring.CSVConnectorWorkflow</WorkflowTypeName>" +

"<WorkflowParameters>" +

//Pass in the parameters here. In this case the two parameters are the data file path and the mapping file path

"<WorkflowParameter Name=\"DataFilePath\" Type=\"string\">" +

this.DataFilePath +

"</WorkflowParameter>" +

"<WorkflowParameter Name=\"FormatFilePath\" Type=\"string\">" +

this.MappingFilePath +

"</WorkflowParameter>" +

"</WorkflowParameters>" +

"<RetryExceptions />" +

"<RetryDelaySeconds>60</RetryDelaySeconds>" +

"<MaximumRunningTimeSeconds>300</MaximumRunningTimeSeconds>" +

"</WindowsWorkflowConfiguration>" +

"</Subscription>";

//Set the module type of the module to be the Windows Workflow Task Write Action Module Type from the Subscriptions MP.

wamCSVConnector.TypeID = wamtWindowsWorkflowTaskWriteAction;

//Add the Write Action Module to the rule

ruleCSVConnector.WriteActionCollection.Add(wamCSVConnector);

//Mark the rule as pending update

ruleCSVConnector.Status = ManagementPackElementStatus.PendingAdd; ;

//Accept the rule changes which updates the database

mpConnectors.AcceptChanges();

}

catch (Exception e)

{

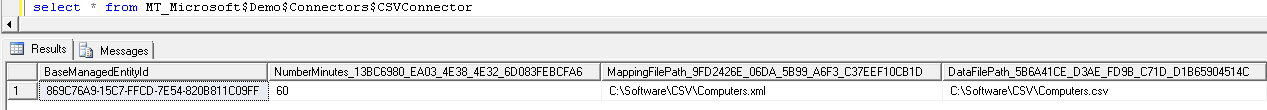
MessageBox.Show(e.Message + e.InnerException.Message);

}

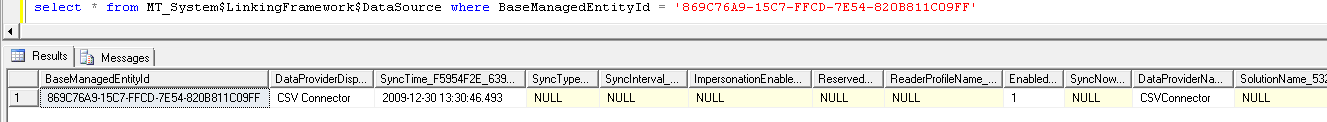
}

When the user is done going through the wizard he ends up with a Connector object in the database and a new rule in the management pack.

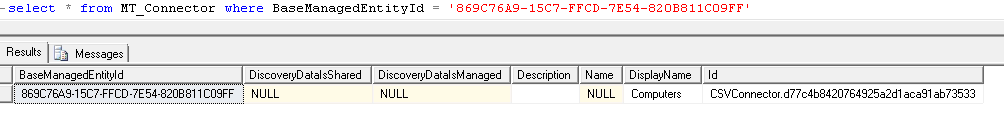
The CSVConnector properties are stored on the MT\_Microsoft$Demo$Connectors$CSVConnector table:



The System.LinkingFramework.DataSource properties are stored on the MT\_System$LinkingFramework$DataSource table:



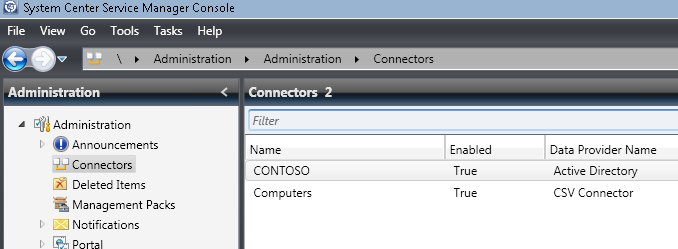
The System.Connector properties are stored on the MT\_Connector table:



The rule that is added to the Microsoft.Demo.Connectors MP looks like this:



And there will be a new Connector listed in the Connectors view:



Now, the update MP will be downloaded to the System Center Management Service and will run according to the Configuration of the Scheduler DataSourceModule. Whatever data exists in the .csv file will be imported/updated in Service Manager.

## The Edit Task

The Edit task is similar to the Create task with a couple of important differences:

1. We need to get the context of the object the user has selected in the UI and pass that information to the WizardData class.
2. We will be using the PropertySheet WizardMode and only need to show the Configuration WizardPage.
3. We need to *update* the properties of the existing Connector object and *update* the Rule DataSourceModule and WriteActionModule configuration XML.

Here is the code for the task handler for the Edit task:

else if (parameters.Contains("Edit"))

{

//Get the server name to connect to and connect to the server

String strServerName = Registry.GetValue("HKEY\_CURRENT\_USER\\Software\\Microsoft\\System Center\\2010\\Service Manager\\Console\\User Settings", "SDKServiceMachine", "localhost").ToString();

EnterpriseManagementGroup emg = new EnterpriseManagementGroup(strServerName);

//Get the object using the selected node ID

String strID = String.Empty;

foreach (NavigationModelNodeBase node in nodes)

{

strID = node["$Id$"].ToString();

}

EnterpriseManagementObject emoCSVConnector = emg.EntityObjects.GetObject<EnterpriseManagementObject>(new Guid(strID), ObjectQueryOptions.Default);

//Create a new "wizard" (also used for property dialogs as in this case), set the title bar, create the data, and add the pages

WizardStory wizard = new WizardStory();

wizard.WizardWindowTitle = "Edit CSV Connector";

WizardData data = new CSVConnectorWizardData(emoCSVConnector);

wizard.WizardData = data;

wizard.AddLast(new WizardStep("Configuration", typeof(CSVConnectorConfigurationPage), wizard.WizardData));

//Show the property page

PropertySheetDialog wizardWindow = new PropertySheetDialog(wizard);

//Update the view when done so the new values are shown

bool? dialogResult = wizardWindow.ShowDialog();

if (dialogResult.HasValue && dialogResult.Value)

{

RequestViewRefresh();

}

}

As you can see, it pretty much follows the same pattern as the Create task except that we have to get the object ID from the selected Node, call the SDK and get the EnterpriseManagementObject and then pass that to the CSVConnectorWizardData constructor that accepts an EnterpriseManagementObject.

Then we only add the Configuration WizardPage and this time we create a PropertySheeDialog class instead of a WizardWindow class.

Now, let’s take a look at how the CSVConnectorWizardData class is different in the Edit case.

internal CSVConnectorWizardData(EnterpriseManagementObject emoCSVConnector)

{

//Get the server name to connect to

String strServerName = Registry.GetValue("HKEY\_CURRENT\_USER\\Software\\Microsoft\\System

Center\\2010\\Service Manager\\Console\\User Settings", "SDKServiceMachine", "localhost").ToString();

//Conneect to the server

EnterpriseManagementGroup emg = new EnterpriseManagementGroup(strServerName);

ManagementPack mpConnectors = emg.GetManagementPack("Microsoft.Demo.Connectors", null,

new Version("1.0.0.0"));

ManagementPackClass classCSVConnector =

mpConnectors.GetClass("Microsoft.Demo.Connectors.CSVConnector");

this.EnterpriseManagementObjectID = emoCSVConnector.Id;

this.DisplayName = emoCSVConnector.DisplayName;

this.DataFilePath = emoCSVConnector[classCSVConnector,"DataFilePath"].ToString();

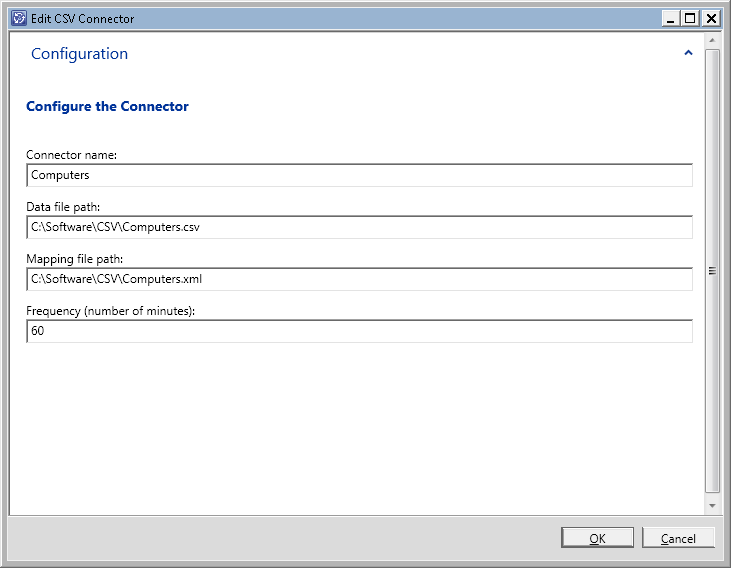
this.MappingFilePath = emoCSVConnector[classCSVConnector, "MappingFilePath"].ToString();

this.NumberMinutes = emoCSVConnector[classCSVConnector, "NumberMinutes"].ToString();

this.ConnectorID = emoCSVConnector[classCSVConnector, "Id"].ToString();

}

The constructor simply takes the EnterpriseManagementObject that is passed in and assigns the current property values to the CSVConnectorWizardData properties. Now they will show up in the property sheet so the user can see the current configuration and change the values if he wants to:



Now, when the user clicks the OK button it will call the same AcceptChanges() method we saw earlier except this time because we are in PropertySheet WizardMode the UpdateConnectorInstance() method is going to be called.

public override void AcceptChanges(WizardMode wizardMode)

{

if (wizardMode == WizardMode.PropertySheet)

{

this.UpdateConnectorInstance();

}

else

{

try

{

this.CreateConnectorInstance();

this.WizardResult = WizardResult.Success;

}

catch (Exception ex)

{

this.WizardResult = WizardResult.Failed;

this.ErrorMessage = ex.ToString();

}

}

}

UpdateConnectorInstance() is where all the magic happens for the Edit task. This is very similar to the CreateConnectorInstance() method except that we are updating an existing object. The only interesting thing to note here really is how we use the connector ID to get the Rule by ID (remember that we created them with the same ID at creation time so that we could update them together later! ☺). See highlight.

private void UpdateConnectorInstance()

{

//Get the server name to connect to and connect

String strServerName =

Registry.GetValue("HKEY\_CURRENT\_USER\\Software\\Microsoft\\System

Center\\2010\\Service Manager\\Console\\User Settings", "SDKServiceMachine", "localhost").ToString();

EnterpriseManagementGroup emg = new EnterpriseManagementGroup(strServerName);

//Get the Connectors MP and CSV Connector Class

ManagementPack mpConnectors = emg.GetManagementPack("Microsoft.Demo.Connectors", null,

new Version("1.0.0.0"));

ManagementPackClass classCSVConnector =

mpConnectors.GetClass("Microsoft.Demo.Connectors.CSVConnector");

//Get the Connector object using the object ID

EnterpriseManagementObject emoCSVConnector =

emg.EntityObjects.GetObject<EnterpriseManagementObject>(this.EnterpriseManagementObjectID, ObjectQueryOptions.Default);

//Set the property values to the new values

emoCSVConnector[classCSVConnector, "DisplayName"].Value = this.DisplayName;

emoCSVConnector[classCSVConnector, "DataFilePath"].Value = this.DataFilePath;

emoCSVConnector[classCSVConnector, "DataFilePath"].Value = this.DataFilePath;

emoCSVConnector[classCSVConnector, "MappingFilePath"].Value = this.MappingFilePath;

emoCSVConnector[classCSVConnector, "NumberMinutes"].Value = this.NumberMinutes;

//Update Connector instance

emoCSVConnector.Commit();

//Get the rule using the Connector ID and then update the data source and write action module configuration

ManagementPackRule ruleConnector = mpConnectors.GetRule(this.ConnectorID);

ruleConnector.DataSourceCollection[0].Configuration =

"<Subscription>" +

"<WindowsWorkflowConfiguration>" +

"<AssemblyName>CSVConnectorWorkflow</AssemblyName>" +

"<WorkflowTypeName>WorkflowAuthoring.CSVConnectorWorkflow</WorkflowTypeName>" +

"<WorkflowParameters>" +

"<WorkflowParameter Name=\"DataFilePath\" Type=\"string\">" +

this.DataFilePath +

"</WorkflowParameter>" +

"<WorkflowParameter Name=\"FormatFilePath\" Type=\"string\">" +

this.MappingFilePath +

"</WorkflowParameter>" +

"</WorkflowParameters>" +

"<RetryExceptions />" +

"<RetryDelaySeconds>60</RetryDelaySeconds>" +

"<MaximumRunningTimeSeconds>300</MaximumRunningTimeSeconds>" +

"</WindowsWorkflowConfiguration>" +

"</Subscription>";

ruleConnector.Status = ManagementPackElementStatus.PendingUpdate;

mpConnectors.AcceptChanges();

}

## The Delete, Disable, Enable Tasks

The Delete, Disable, and Enable tasks are pretty similar, so we will handle them all in one branch of the else statement. This is all pretty self explanatory and well commented.

else if (parameters.Contains("Delete") || parameters.Contains("Disable") || parameters.Contains("Enable"))

{

//Get the server name to connect to and create a connection

String strServerName =

Registry.GetValue("HKEY\_CURRENT\_USER\\Software\\Microsoft\\System Center\\2010\\Service Manager\\Console\\User Settings", "SDKServiceMachine", "localhost").ToString();

EnterpriseManagementGroup emg = new EnterpriseManagementGroup(strServerName);

//Get the object using the selected node ID

String strID = String.Empty;

foreach (NavigationModelNodeBase node in nodes)

{

strID = node["$Id$"].ToString();

}

EnterpriseManagementObject emoCSVConnector =

emg.EntityObjects.GetObject<EnterpriseManagementObject>(new Guid(strID), ObjectQueryOptions.Default);

if(parameters.Contains("Delete"))

{

//Remove the object from the database

IncrementalDiscoveryData idd = new IncrementalDiscoveryData();

idd.Remove(emoCSVConnector);

idd.Commit(emg);

}

//Get the rule using the connector ID

ManagementPack mpConnectors = emg.GetManagementPack("Microsoft.Demo.Connectors",

null, new Version("1.0.0.0"));

ManagementPackClass classCSVConnector =

mpConnectors.GetClass("Microsoft.Demo.Connectors.CSVConnector");

String strConnectorID = emoCSVConnector[classCSVConnector, "Id"].ToString();

ManagementPackRule ruleConnector = mpConnectors.GetRule(strConnectorID);

//Update the Enabled property or delete as appropriate

if(parameters.Contains("Delete"))

{

ruleConnector.Status = ManagementPackElementStatus.PendingDelete;

}

else if (parameters.Contains("Disable"))

{

emoCSVConnector[classCSVConnector, "Enabled"].Value = false;

ruleConnector.Enabled = ManagementPackMonitoringLevel.@false;

ruleConnector.Status = ManagementPackElementStatus.PendingUpdate;

}

else if (parameters.Contains("Enable"))

{

emoCSVConnector[classCSVConnector, "Enabled"].Value = true;

ruleConnector.Enabled = ManagementPackMonitoringLevel.@true;

ruleConnector.Status = ManagementPackElementStatus.PendingUpdate;

}

//Commit the changes to the connector object and rule

emoCSVConnector.Commit();

mpConnectors.AcceptChanges();

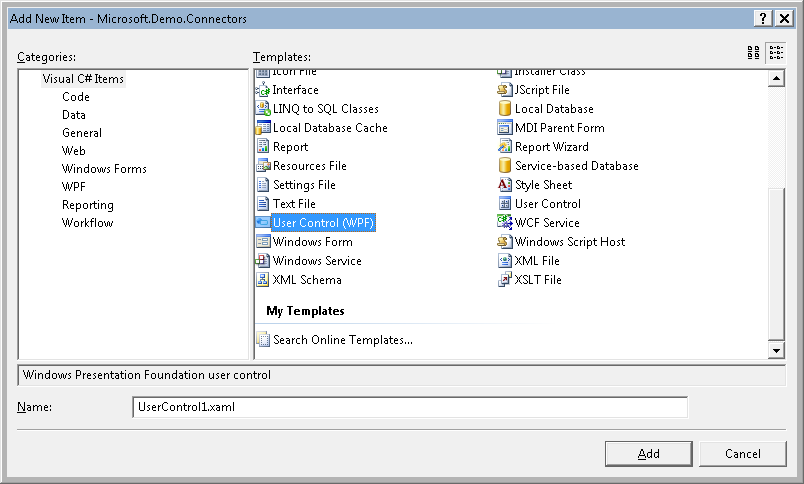
//Update the view when done so the item is either removed or the updated Enabled value shows

RequestViewRefresh();

}

## WizardPages

In this example there are four wizard pages in the wizard and one of the pages (CSVConnectorConfigurationPage) is reused in the PropertySheet mode. Each WizardPage is a XAML UserControl. To add a WizardPage to your project just right click on your project in the Solution Explorer and choose Add 🡪 New Item and then choose User Control (WPF):



Then in the XAML, you’ll want to change a few things:

* Change the base class from UserControl to derive from one of the three Wizard Framework WizardPage base classes:
  + wpfwiz:WizardWelcomePageBase – use this for the first page in the wizard – typically a welcome page
  + wpfwiz:WizardRegularPageBase – use this for any page in the middle of the wizard including a confirmation page
  + wpfwiz:WizardResultPageBase – use this for the last page in the wizard – typically a completion page
* add the wpfwiz namespace
  + xmlns:wpfwiz="clr-namespace:Microsoft.EnterpriseManagement.UI.WpfWizardFramework;assembly=Microsoft.EnterpriseManagement.UI.WpfWizardFramework"

The complete XAML for the Configuration page is below. Notice the binding to the CSVConnectorWizardData class properties. That’s all you have to do to bind the XAML form controls to WizardData classes! When this form is used in edit mode the exact same page is used and the binding still works!

<wpfwiz:WizardRegularPageBase x:Class="Microsoft.Demo.Connectors.CSV.CSVConnectorConfigurationPage"

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

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

xmlns:local="clr-namespace:Microsoft.Demo.Connectors.CSV"

xmlns:wpfwiz="clr-namespace:Microsoft.EnterpriseManagement.UI.WpfWizardFramework;assembly=Microsoft.EnterpriseManagement.UI.WpfWizardFramework">

<Grid Name="ConfigurationGrid" Margin="15,25,15,25">

<TextBlock Height="28" Margin="0,0,0,0" Name="titleLabel" FontWeight="Bold" Foreground="#FF003399" FontSize="14" VerticalAlignment="Top" Text="Configure the Connector"/>

<ScrollViewer Margin="0,50,0,50" Name="scrollViewer" CanContentScroll="True" VerticalScrollBarVisibility="Auto">

<StackPanel Name="stackPanel" Orientation="Vertical">

<Label Height="25" Padding="0" Margin="0,0,0,0" Name="displayamelabel" Content="Connector name:"/>

<TextBox Height="25" Margin="0,-8,0,10" Name="displaynameTextBlock">

<TextBox.Text>

<Binding Path="DisplayName" Mode="TwoWay"/>

</TextBox.Text>

</TextBox>

<Label Height="25" Padding="0" Margin="0,0,0,0" Name="datafilepathLabel" Content="Data file path:"/>

<TextBox Height="25" Margin="0,-8,0,10" Name="domainTextBlock">

<TextBox.Text>

<Binding Path="DataFilePath" Mode="TwoWay"/>

</TextBox.Text>

</TextBox>

<Label Height="25" Padding="0" Margin="0,0,0,0" Name="mappingfilepathLabel" Content="Mapping file path:"/>

<TextBox Height="25" Margin="0,-8,0,10" Name="credentialTextBlock">

<TextBox.Text>

<Binding Path="MappingFilePath" Mode="TwoWay"/>

</TextBox.Text>

</TextBox>

<Label Height="25" Padding="0" Margin="0,0,0,0" Name="numberminutesLabel" Content="Frequency (number of minutes):"/>

<TextBox Height="25" Margin="0,-8,0,10" Name="numberminutesTextBlock">

<TextBox.Text>

<Binding Path="NumberMinutes" Mode="TwoWay" FallbackValue="60"/>

</TextBox.Text>

</TextBox>

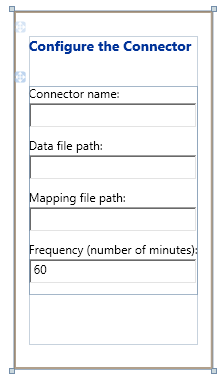
</StackPanel>

</ScrollViewer>

</Grid>

</wpfwiz:WizardRegularPageBase>

The Page looks like this in design mode:



If you want to you can put some code behind logic in the .cs file that corresponds to the .xaml file. You’ll at least need to provide a constructor for the WizardPage class that accepts a WizardData object as a parameter and binds it to the WizardPage DataContext property. You may also want to set the FinishButtonText, Title, and maybe other properties of the WizardPage class.

public CSVConnectorConfigurationPage(WizardData wizardData)

{

if (wizardData == null)

{

throw new ArgumentNullException("wizardData");

}

InitializeComponent();

this.DataContext = wizardData;

this.csvConnectorWizardData = this.DataContext as CSVConnectorWizardData;

this.Title = "Configure the CSV Connector";

this.FinishButtonText = "Create";

}

# Deployment

In order to fully deploy this management pack the following actions must be taken:

1. The Microsoft.Demo.Connectors.xml management pack must be imported into Service Manager.
2. The Microsoft.Demo.Connectors.dll located in the \bin\debug directory must be deployed to the %ProgramFiles%\Microsoft System Center\Service Manager 2010 directory on all computers where the users will be interacting with the CSV connectors wizard/property sheet. This is only accessible to Service Manager administrators so it doesn’t necessarily need to be deployed broadly to all Service Manager consoles.
3. The CSVConnectorWorkflow.dll file must be deployed to all Service Manager Management Servers that run workflows (if you have more than one for redundancy/failover). The file must be placed in the %ProgramFiles%\Microsoft System Center\Service Manager 2010 directory.

# Testing

Two pairs of .csv data and .xml mapping files are provided with the example package – one for users and another for computers. You can use these for testing by putting them someplace on the management server or on a file share accessible by the Operational System Account.

* Users.csv
* Users.xml
* Computers.csv
* Computers.xml