PNMsoft Knowledge Base

Sequence User Guides

Defining a Custom Activity

Nov. 2013
Product Version 7.0 and above

PNMSOFT
Business Processes in Motion™

PNMsoft UK 38 Clarendon Road Watford Hertfordshire WD17 1JJ

Tel: +44(0)192 381 3420 • Email: info@pnmsoft.com • Website: www.pnmsoft.com

Microsoft Partner
Gold Application Development

PNMSOFT
Business Processes in Motion™

TABLE OF CONTENTS

Introduction

This document describes how to add a Custom Activity to the Sequence App Studio.

The App Studio already contains many useful activities, but sometimes, you may want to design a custom activity which performs actions or enables integration which is specific to your organization or its requirements.

By creating a Custom Activity, you are extending Sequence, providing your developers with additional building blocks that they can use to better create workflow applications that fit your organisation's needs.

For example, you may want to create an activity which communicates with a proprietary system, or an activity which retrieves data from one source and updates another.

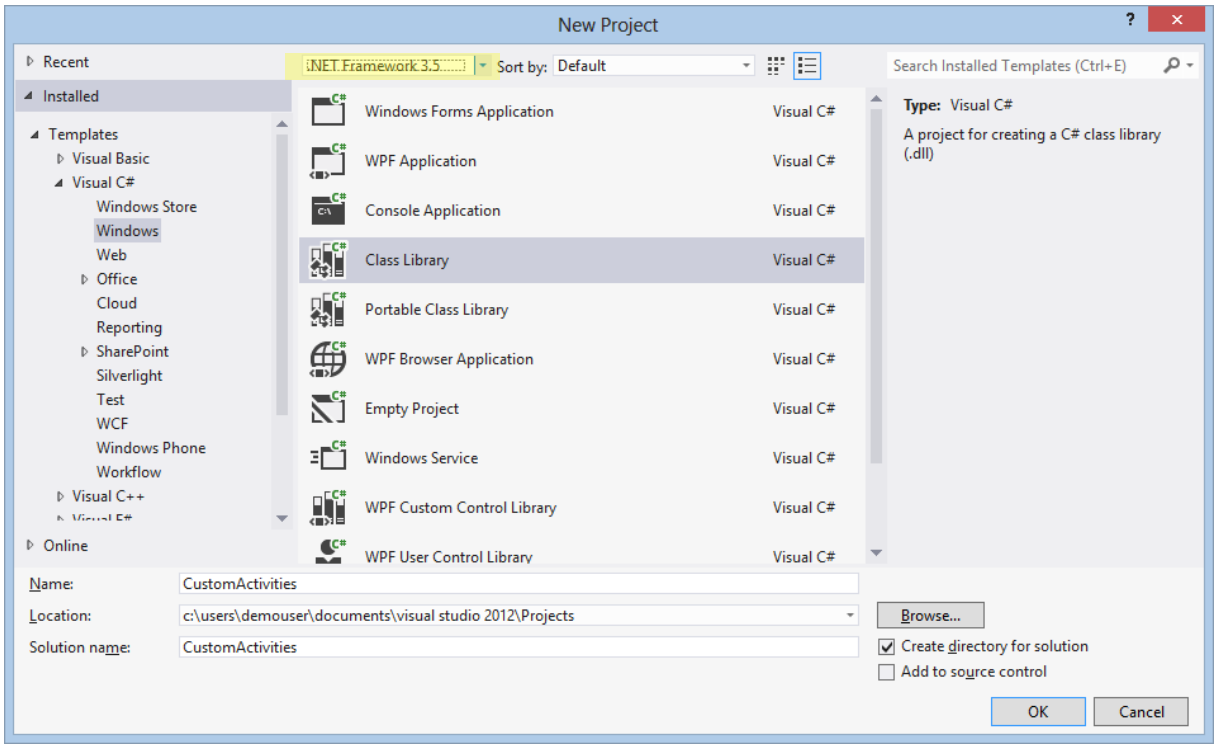
The possibilities are limitless. This document simply provides the general steps you need to take in each case to set up your Custom Activity.

Activity Development

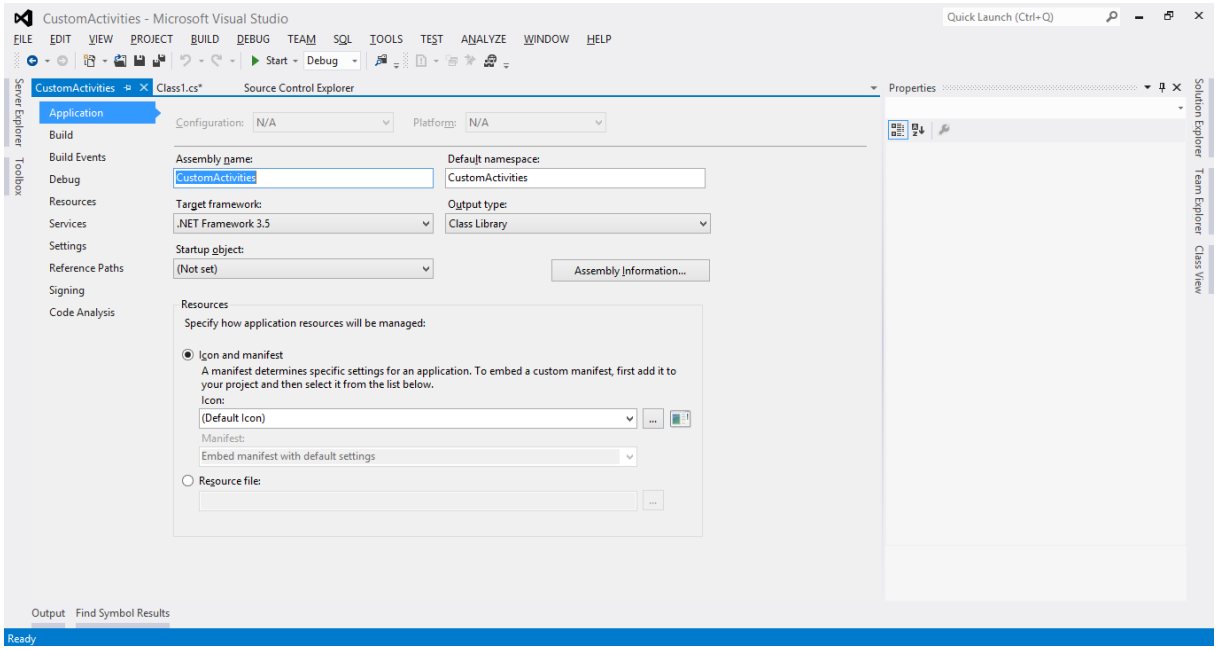
Setting up the project

1. In Visual Studio, add a new project: of type C# Class Library
   If you are using Visual Studio 2012 or later change the .NET Framework to 3.5.

   Note: Use .NET 3.5 if your solution uses SharePoint is 2010 or below.
   Use .NET 4.0 if your solution uses SharePoint 2013.



Add New Project



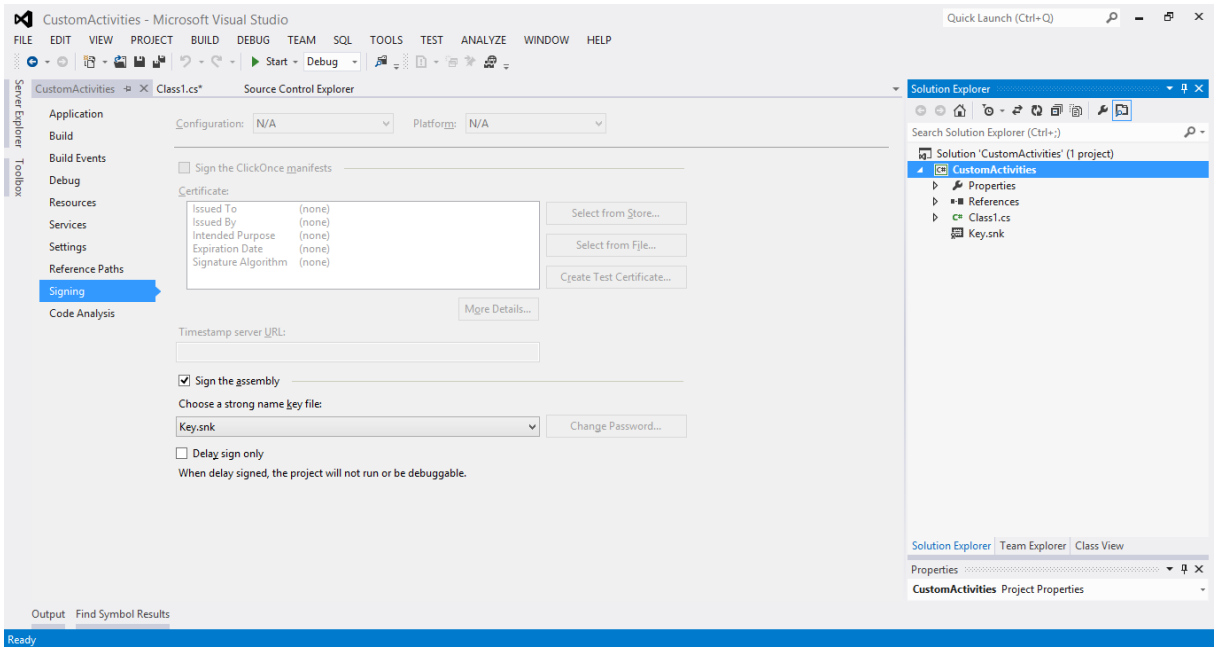
New Project

1. Sign the assembly.

   Assembly signing (also called strong-name signing) gives an application or component a unique identity that other software can use to identify and refer explicitly to it. A strong name consists of its simple text name, version number, culture information (if provided), plus a public/private key pair.

To sign the assembly:

1. Open **Project Properties**, and select **Signing** from the menu options.
2. Select **Sign the assembly**.
3. In the Choose a strong name key file combo box, select **New**.
4. In the pop-up box, enter a name e.g. key.
5. Deselect the Protect my key file with a password check box.
6. Select **Save**.



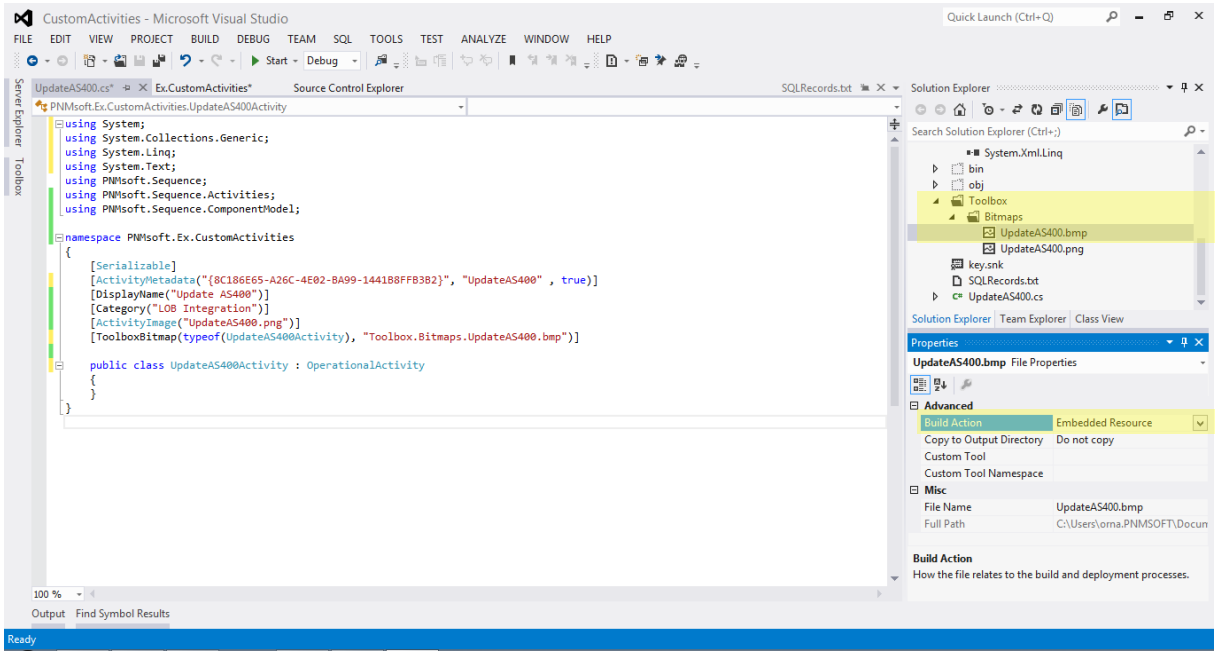
Sign the Assembly

2. Add an image for the Custom Activity to be displayed in the App Studio toolbox.

Here you will add a .bmp file as an embedded resource to the project.

To add a toolbox image for the Custom Activity:

a. Create a 32 x 32 pixel .bmp image to represent the activity.
b. Place the .bmp file in the project folder.
c. Select the .bmp file in the Solution Explorer.
d. In the Properties section, set the *Build Action* property to **Embedded Resource**.



Adding an Image

7. Add references to these dlls:

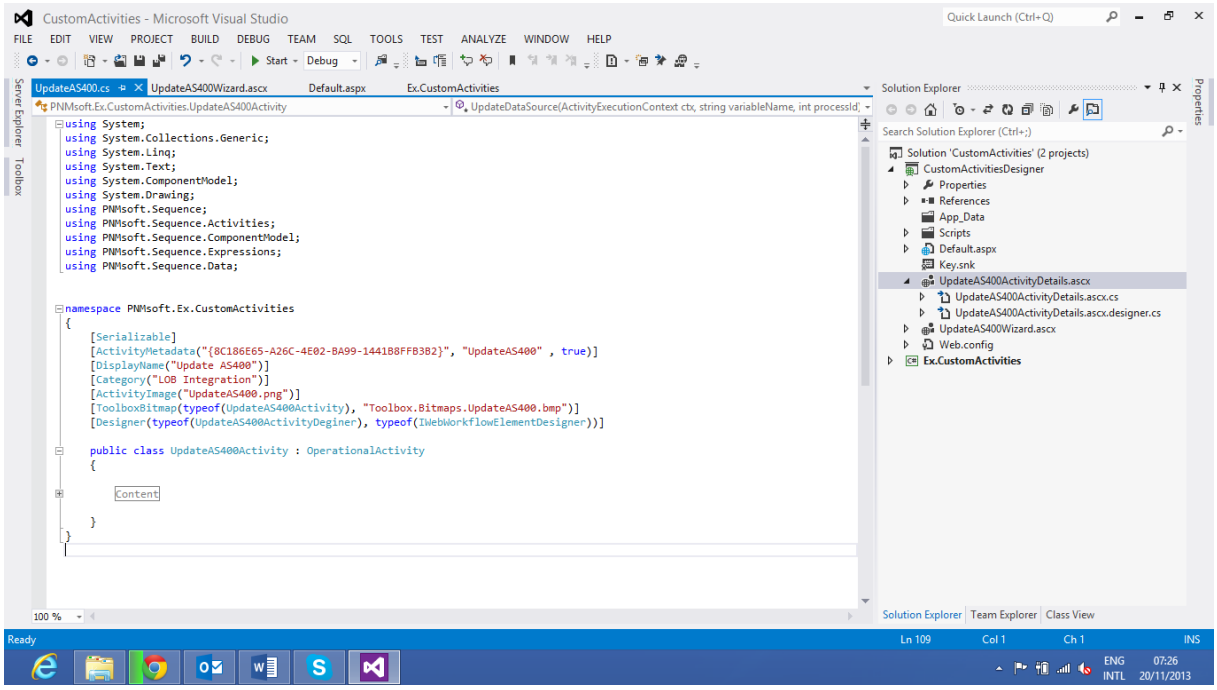
   ```
   PNMsoft.Sequence
   PNMsoft.Sequence.Activities
   PNMsoft.Sequence.Security
   PNMsoft.Sequence.Design
   ```

Note: Other references might be needed based on your implementation.

Implementing the Activity Logic

1. Add a class item that will be used for the Custom Activity code. The class should inherit from the activity you wish to extend. If you are unsure, use `OperationalActivity`.
2. Add the following using statements

   ```
   using PNMsoft.Sequence;
   using PNMsoft.Sequence.Activities;
   using PNMsoft.Sequence.ComponentModel;
   using  PNMsoft.Sequence.Expressions;
   using PNMsoft.Sequence.Data;
   ```



Add Class for Custom Activity

3. Define the following attributes to the Custom Activity class:

   - `ActivityMetadata`: create a GUID for the custom activity and a default name to be used when a user adds a new activity of this type.
     Include this in the format: GUID,<default name>
     (e.g. 4E0782F3-B3B6-48B5-B9E1-222C9C0F516D,CustomAct).

     Use false if this activity prevents workflows from been initiated manually; use true otherwise.

   - `DisplayName` – the name of the activity in the toolbox.

   - `ActivityImage` – the image of the activity to be displayed on the workflow canvas. Place the image in: *C:\Program Files\PNMsoft\Shared Resources\images\Diagram*.
     The image should be 32 x 32 pixels.

   - `Category` – the category of the toolbox where the activity will appear (you can create a new category).

   - `Serializable` – Add this decoration to your class to make sure your activity is applying this behavior. As the workflow definition is saved to the database this is a mandatory property.

     Serialization is the process of converting an object into a stream of bytes in order to store the object or transmit it to memory, a database, or a file. Its main purpose is to save the state of an object in order to be able to recreate it when needed. The reverse process is called deserialization.

   - `ToolboxBitmap` – the name of the image you added to the project. This image will be displayed on the activities toolbar.

   - `Designer` – the designer class type. Enter the designer class name which you will create in the next steps.

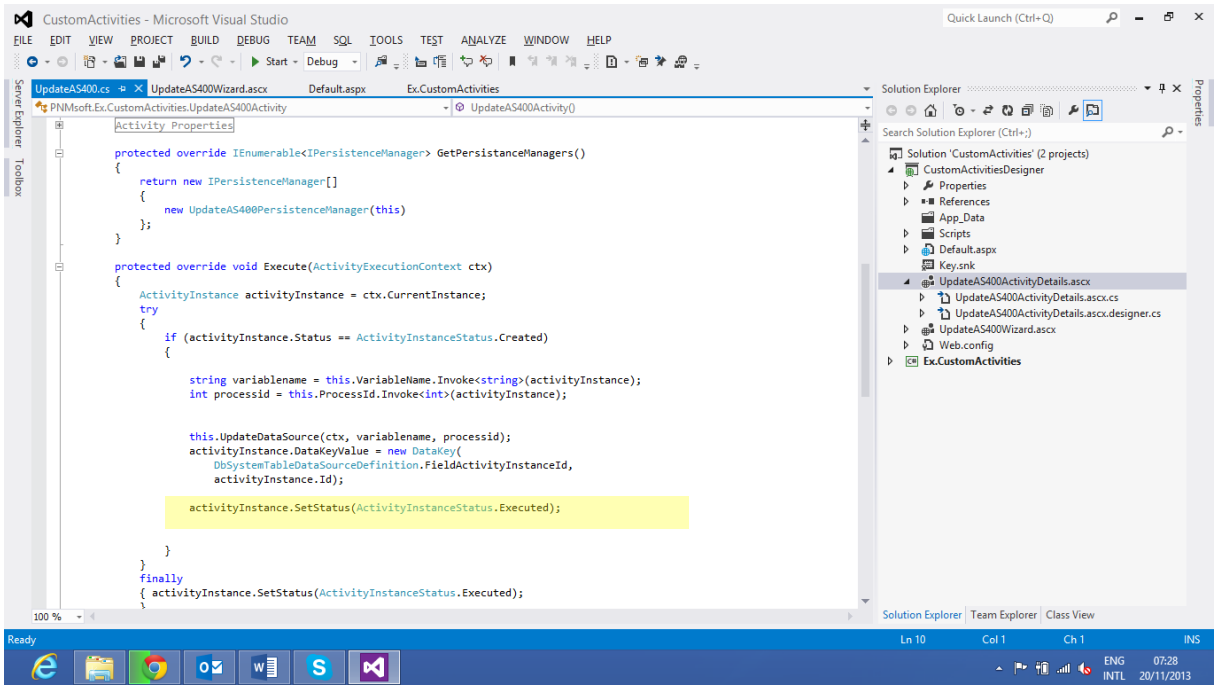
4. In your custom activity class, add an empty constructor.

5. Add your activity properties.
6. Override the `GetPersistanceManagers()` method and return an instance of the class that you will create in the next section.
7. Override the Execute method. Verify that you have set the activity status correctly.

   Use this line to achieve this:

   activityInstance.SetStatus(ActivityInstanceStatus.Executed);

Note: You should add try and finally in your code to make sure the activity does not stay in execute status.



Add Execute Method

8. Add the relevant code to your project. Save data to the activity table by using `IDbTableDataSourceExecutor`
9. Below is an example which writes two values to the database.

   In this example, the UpdateDataSource method inserts the data to the database, and it is called from the execute method. The properties used are expressions, invoke is used to return the results.

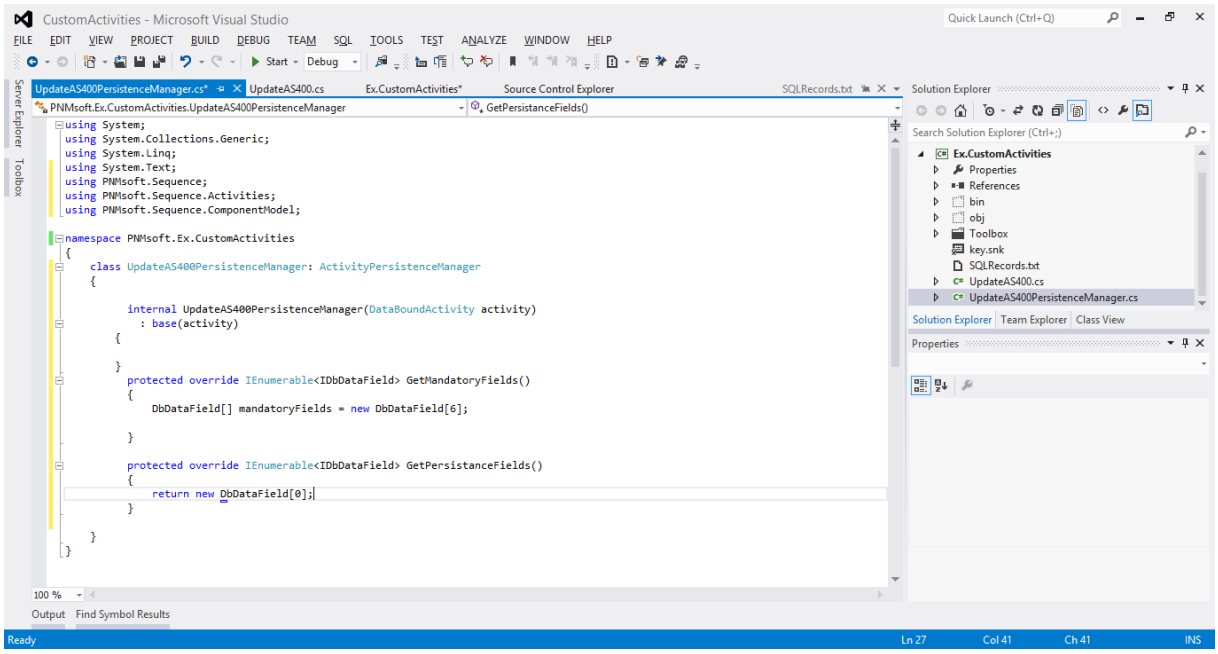
```
using System;
using System.Collections.Generic;
using System.Linq;
using System.Text;
using System.ComponentModel;
using System.Drawing;
using PNMsoft.Sequence;
using PNMsoft.Sequence.Activities;
using PNMsoft.Sequence.ComponentModel;
using PNMsoft.Sequence.Expressions;
using PNMsoft.Sequence.Data;
```

```
namespace PNMsoft.Ex.CustomActivities
{
    [Serializable]
    [ActivityMetadata("{8C186E65-A26C-4E02-BA99-1441B8FFB3B2}", "UpdateAS400" , true)]
    [DisplayName("Update AS400")]
    [Category("LOB Integration")]
    [ActivityImage("UpdateAS400.png")]
    [ToolboxBitmap(typeof(UpdateAS400Activity), "Toolbox.Bitmaps.UpdateAS400.bmp")]
    [Designer(typeof(UpdateAS400ActivityDeginer),
typeof(IWebWorkflowElementDesigner))]

    public class UpdateAS400Activity : OperationalActivity
    {

        #region Content
        public UpdateAS400Activity()
        {}

        #region  Activity Properties

         [ExpressionValidation(ExpectedType = typeof(int), IsRequired = true)]
        public SqExpression ProcessId
        {
            get
            {
                return base.Get<SqExpression>("ProcessId");
            }
            set
            {
                base.Set<SqExpression>("ProcessId", value);
            }
        }

        [ExpressionValidation(ExpectedType = typeof(string), IsRequired = true)]
        public SqExpression VariableName
        {
            get
            {
                return base.Get<SqExpression>("VariableName");
            }
            set
            {
                base.Set<SqExpression>("VariableName", value);
            }
        }
        #endregion

        protected override IEnumerable<IPersistenceManager> GetPersistanceManagers()
        {
            return new IPersistenceManager[]
            {
                new UpdateAS400PersistenceManager(this)
            };
        }

        protected override void Execute(ActivityExecutionContext ctx)
        {
            ActivityInstance activityInstance = ctx.CurrentInstance;
            try
            {
```

```
                if (activityInstance.Status == ActivityInstanceStatus.Created)
                {

                    string variablename =
this.VariableName.Invoke<string>(activityInstance);
                    int processid = this.ProcessId.Invoke<int>(activityInstance);


                    this.UpdateDataSource(ctx, variablename, processid);
                    activityInstance.DataKeyValue = new DataKey(
                        DbSystemTableDataSourceDefinition.FieldActivityInstanceId,
                        activityInstance.Id);

                    activityInstance.SetStatus(ActivityInstanceStatus.Executed);


                }
            }
            finally
            { activityInstance.SetStatus(ActivityInstanceStatus.Executed);
            }
        }

        protected void UpdateDataSource(ActivityExecutionContext ctx, string
variableName, int processId)
        {
                Dictionary<string, object> tableData = new Dictionary<string,
object>();

tableData.Add(DbSystemTableDataSourceDefinition.FieldWorkflowInstanceId,
ctx.WorkflowInstance.Id);

tableData.Add(DbSystemTableDataSourceDefinition.FieldActivityInstanceId,
ctx.CurrentInstance.Id);
                tableData.Add("VariableName", variableName);
                tableData.Add("ProcessId", processId);
                IDbTableDataSourceExecutor executor =
(IDbTableDataSourceExecutor)this.DataSource.GetExecutor();
                executor.ExecuteInsert(tableData);
        }
        #endregion

    }
}
```

Implementing Persistance Manager (optional)

1. If your activity captures data during its execution, add a persistence manager class that inherits from `ActivityPersistenceManager`.



Add Persistence Manager Class

2. Add an internal constructor and override the GetMandatoryFields() method and GetPersistenceFileds().
   - GetMandatoryFields () defines the fields that will be created in the database table.
3. Below is an example of a persistence manager that creates a table with a primary key, system keys and two more fields (one is string and one is int).

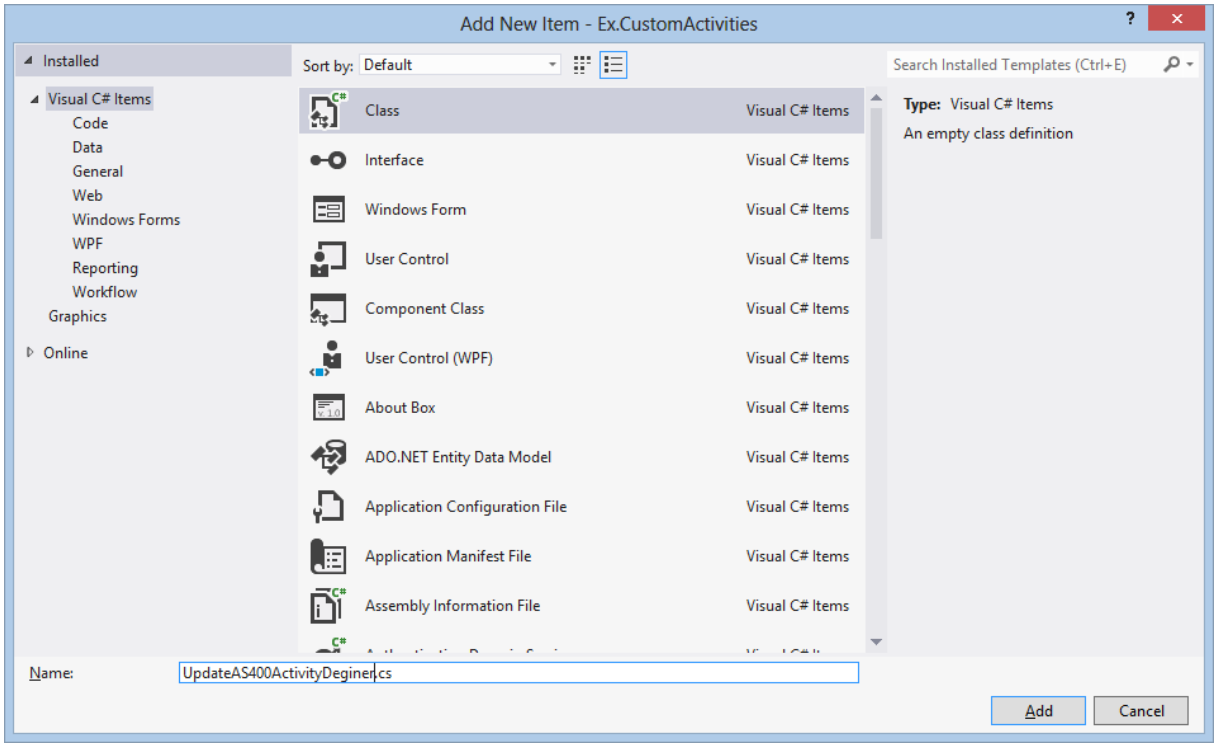
```
using System;
using System.Collections.Generic;
using System.Linq;
using System.Text;
using System.Data;
using PNMsoft.Sequence;
using PNMsoft.Sequence.Activities;
using PNMsoft.Sequence.ComponentModel;
using PNMsoft.Sequence.Data;

namespace PNMsoft.Ex.CustomActivities
{
    class UpdateAS400PersistenceManager: ActivityPersistenceManager
    {

          internal UpdateAS400PersistenceManager(DataBoundActivity activity)
            : base(activity)
        {

        }
          protected override IEnumerable<IDbDataField> GetMandatoryFields()
          {

              DbDataField[] mandatoryFields = new DbDataField[6];

              mandatoryFields[0] = new DbDataField(
                  DbSystemTableDataSourceDefinition.FieldDefaultKey,
                  DbType.Int32,
                  4, false, DbKeyType.PrimaryKey, true, true, 1, 1);
```

```
            mandatoryFields[1] = new DbDataField(
                DbSystemTableDataSourceDefinition.FieldWorkflowInstanceId,
                DbType.Int32,
                4, true);


            mandatoryFields[2] = new DbDataField(
                DbSystemTableDataSourceDefinition.FieldActivityInstanceId,
                DbType.Int32,
                4, true);

            mandatoryFields[3] = new DbDataField(
                "VariableName",
                DbType.String,
                null, false);
            mandatoryFields[4] = new DbDataField(
                "ProcessId",
                DbType.Int32,
                null, false);
        }
    }
}
```

Persistence Manager class

Developing The Activity Designer (optional)

1. Add a class to your project for the activity designer. Ensure that the name matches the custom activity attribute you have added in earlier stages.



Add New Item

2. Add these using statements to your class:

   using System.Web;

   using PNMsoft.Sequence;

   using PNMsoft.Sequence.Design.Web;

   using PNMsoft.Sequence.Activities;

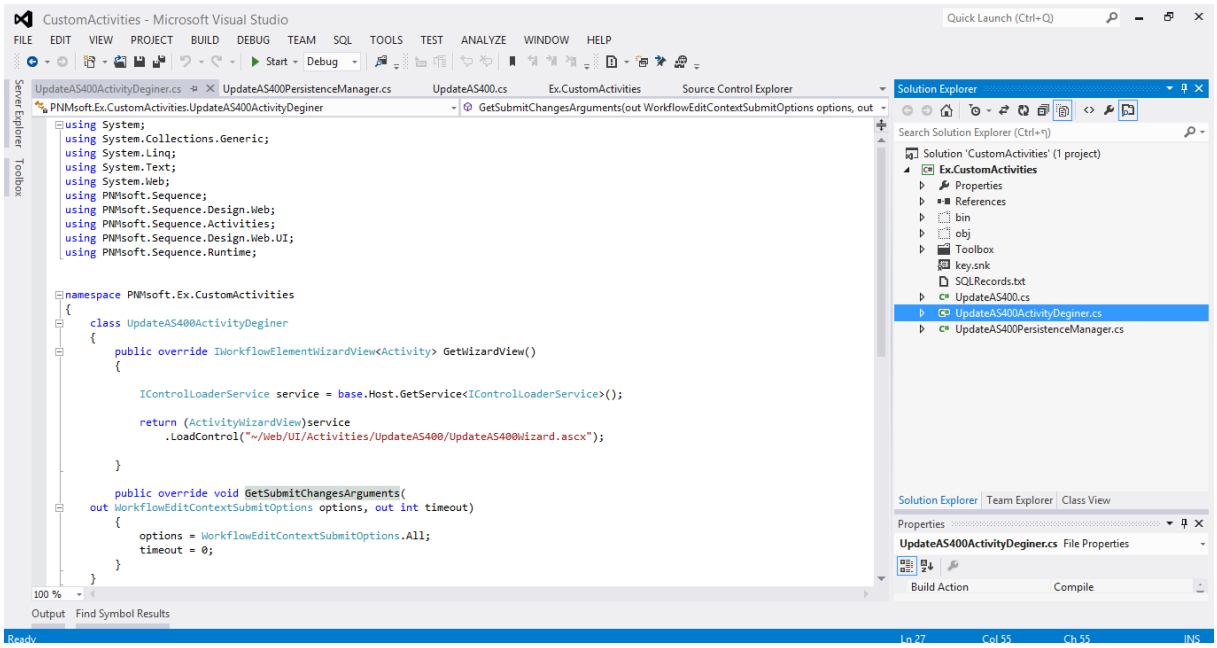
   using PNMsoft.Sequence.Design.Web.UI;

   using PNMsoft.Sequence.Runtime;

3. Make sure your class inherits from `ActivityDesigner`.
4. Override the `GetWizardView` method with code that loads your user control (you will create the user control in future steps).
5. Override GetSubmitChangesArguments.
6. Below is an example of an Activity Designer class:

```
using System.Collections.Generic;
using System.Linq;
using System.Text;
using System.Web;
using PNMsoft.Sequence;
using PNMsoft.Sequence.Design.Web;
using PNMsoft.Sequence.Activities;
using PNMsoft.Sequence.Design.Web.UI;
using PNMsoft.Sequence.Runtime;


namespace PNMsoft.Ex.CustomActivities
{
    class UpdateAS400ActivityDeginer: ActivityDesigner
    {
        public override IWorkflowElementWizardView<Activity> GetWizardView()
        {

            IControlLoaderService service =
base.Host.GetService<IControlLoaderService>();

            return (ActivityWizardView)service

.LoadControl("~/Web/UI/Activities/UpdateAS400/UpdateAS400Wizard.ascx");

        }

        public override void GetSubmitChangesArguments(
    out WorkflowEditContextSubmitOptions options, out int timeout)
        {
            options = WorkflowEditContextSubmitOptions.All;
            timeout = 0;
        }
    }
}
```

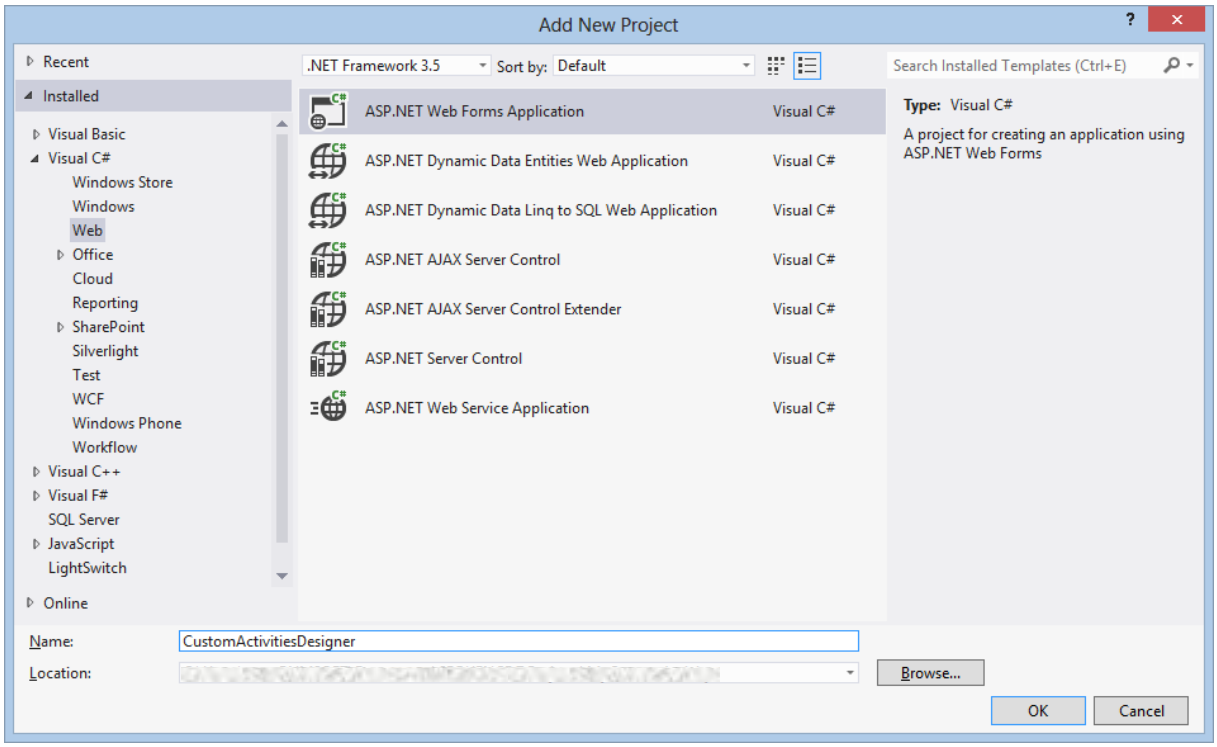
Note: the ascx file and path will be added to the solution later.



Add Content to Activity Designer Class

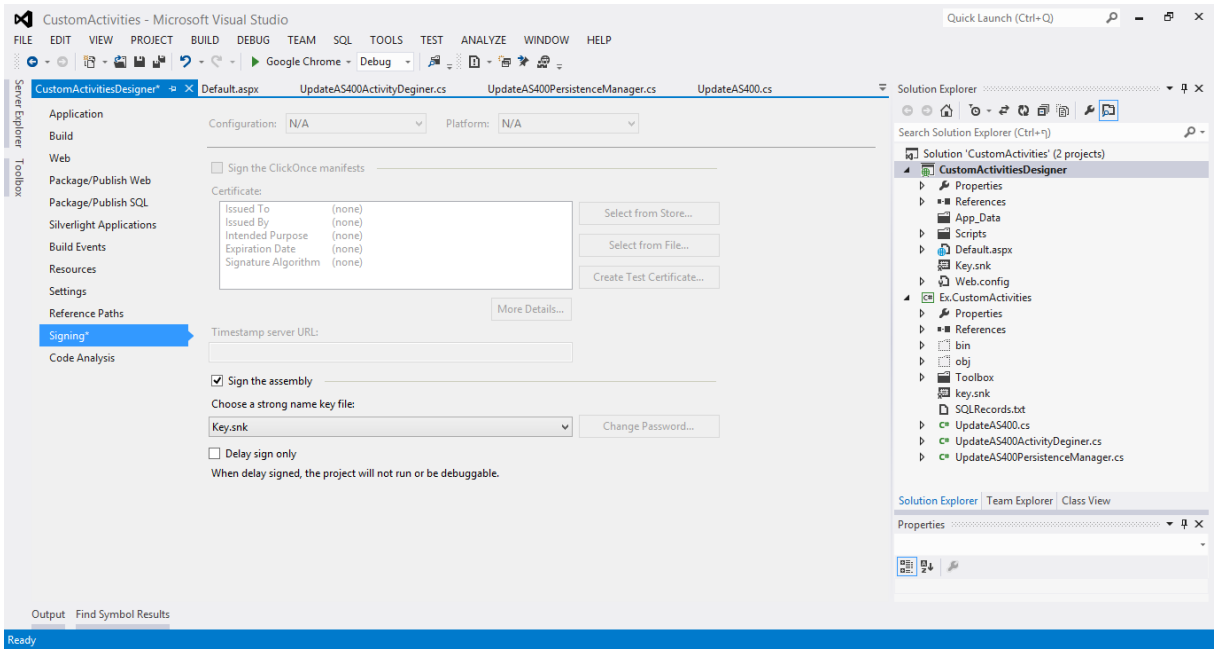
Developing the Activity Wizard

1. Add a web application project to the solution. This project will be the wizard UI.



Add Web Application to the Project

2. Sign the project (see step 1 above).

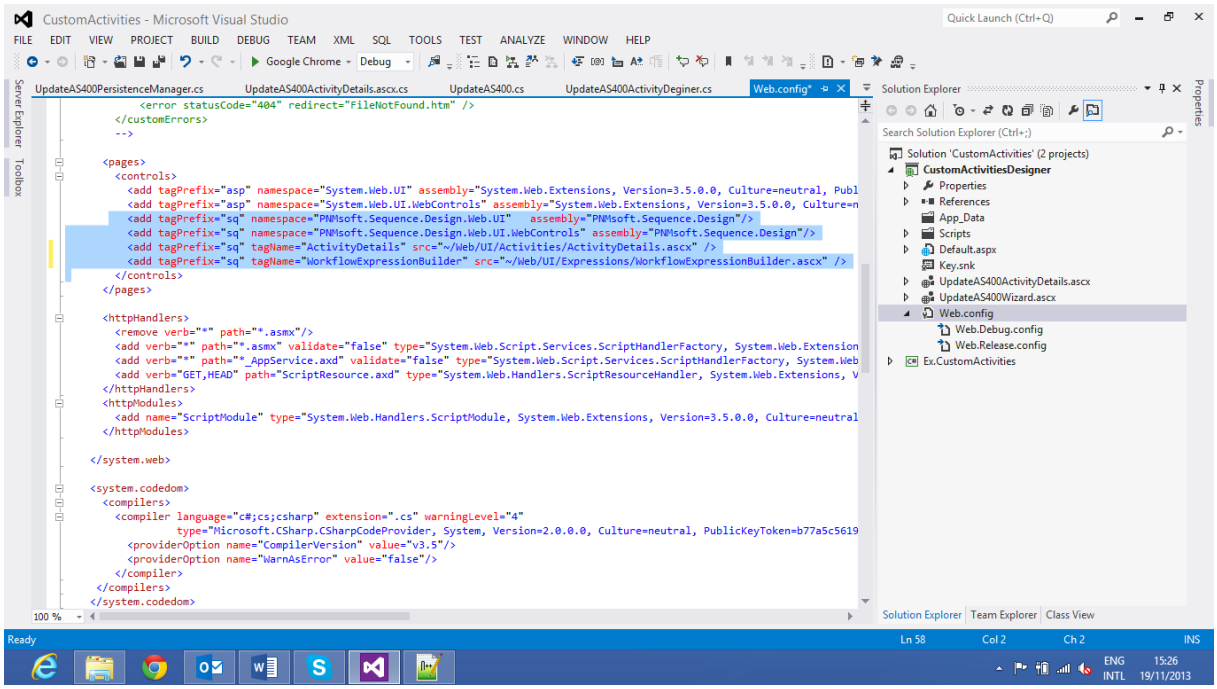


Sign the Project

3. In the project, add references to
   `PNMsoft.Sequence`
   `PNMsoft.Sequence.Design`
   and your activity project, e.g. `Ex.CustomActivities`

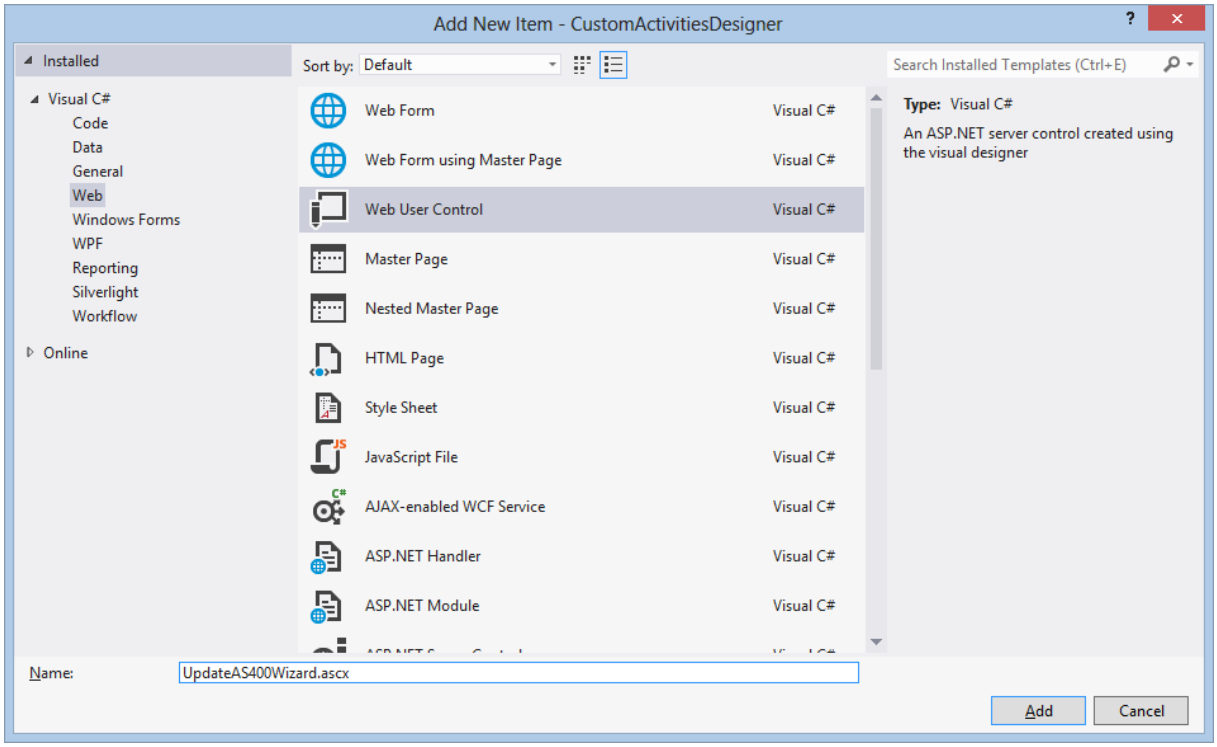
4. Add the following two control tags in the web.config, in the <pages><controls> element.

```
   <add tagPrefix="sq" namespace="PNMsoft.Sequence.Design.Web.UI"
assembly="PNMsoft.Sequence.Design"/>
   <add tagPrefix="sq" namespace="PNMsoft.Sequence.Design.Web.UI.WebControls"
assembly="PNMsoft.Sequence.Design"/>
   <add tagPrefix="sq" tagName="ActivityDetails"
src="~/Web/UI/Activities/ActivityDetails.ascx" />
   <add tagPrefix="sq" tagName="WorkflowExpressionBuilder"
src="~/Web/UI/Expressions/WorkflowExpressionBuilder.ascx" />
```



Add Control Tags in web.config

5. Add a Web User Control (ascx file) named *UpdateAS400Wizard.ascx* to your project. Edit the source location if necessary (Src). This file will be the designer shell. You will add the content in the following steps.



Add Web User Control

6. Below is an example for the wizard shell which contains two pages, one for the activity details (name and alias) and the other for the AS400 setting.

```
<%@ Control Language="C#" AutoEventWireup="true"
CodeBehind="UpdateAS400Wizard.ascx.cs"
Inherits="CustomActivitiesDesigner.UpdateAS400Wizard" %>

<%@ Register Src="~/Web/UI/Activities/UpdateAS400/UpdateAS400ActivityDetails.ascx"
TagPrefix="ca" TagName="CustomActivityDetails" %>

<sq:WizardForm ID="WizardForm" runat="server">
    <wizardpages>

       <sq:WizardTemplatedPage runat="server" ID="ActivityDetailsPage" Title="Update
AS400 Activity Details">
            <ContentTemplate>

                <sq:ActivityDetails runat="server" ID="ActivityDetails" />

            </ContentTemplate>
            <FooterTemplate>

                <sq:WizardButtonGroup runat="server" Buttons="Next" />
                <sq:WizardButtonGroup runat="server" Buttons="Cancel" />

            </FooterTemplate>
        </sq:WizardTemplatedPage>

        <sq:WizardTemplatedPage runat="server" ID="SourceDatabaseSettingsPage"
Title="AS400 Connection Settings">
            <ContentTemplate>

                <ca:CustomActivityDetails runat="server"></ca:CustomActivityDetails>

            </ContentTemplate>
            <FooterTemplate>
                <sq:WizardButtonGroup runat="server" Buttons="Back, Finish" />
                <sq:WizardButtonGroup runat="server" Buttons="Cancel" />
            </FooterTemplate>
        </sq:WizardTemplatedPage>

    </wizardpages>
</sq:WizardForm>
```

7. Add another ascx file named UpdateAS400ActivityDetails.ascx. Here you will build a GUI interface to enable editing the Custom Activity's properties. If a property is an sqExpression, you need to add a button. See more in the example below.

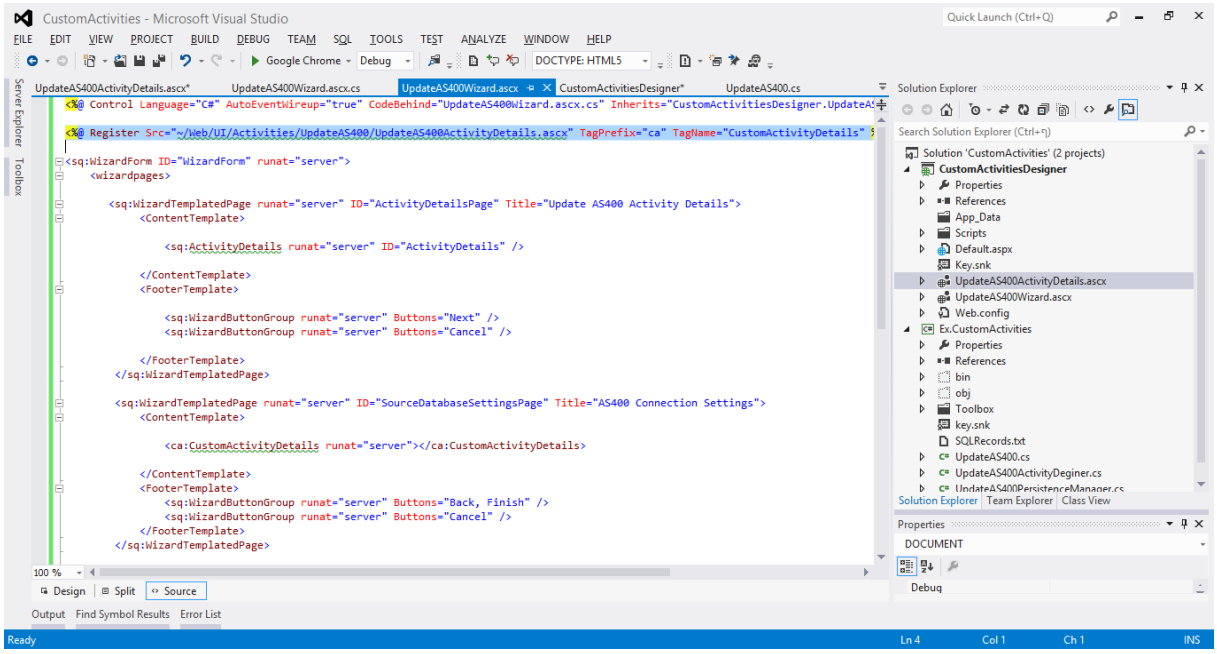
Note: Sequence has built-in skins for different wizard elements. It is recommended to use the system skins to create wizards that blend in with the Sequence environment.

```
<%@ Control Language="C#" AutoEventWireup="true"
CodeBehind="UpdateAS400ActivityDetails.ascx.cs"
Inherits="CustomActivitiesDesigner.UpdateAS400ActivityDetails" %>

<table>
    <tr>
        <td>
            <asp:Label ID="Label3" runat="server" AssociatedControlID="processId"
SkinID="MainInstruction">Process Id</asp:Label>
```

```
            <asp:TextBox runat="server" ID="ProcessId"
SkinID="TextBoxHalfSize"></asp:TextBox>
            <asp:Button runat="server" ID="processIdButton" SkinID="3DotsButton" />
            <asp:RequiredFieldValidator ID="RequiredFieldValidator3" runat="server"
ControlToValidate="processId" ErrorMessage="The process Id is required."
SetFocusOnError="true"></asp:RequiredFieldValidator>
            <br />
        </td>
    </tr>

    <tr>
        <td>
            <asp:Label ID="Label5" runat="server" AssociatedControlID="VariableName"
SkinID="MainInstruction">Variable Name</asp:Label>
            <asp:TextBox runat="server" ID="VariableName"
SkinID="TextBoxHalfSize"></asp:TextBox>
            <asp:Button runat="server" ID="VariableNameButton" SkinID="3DotsButton" />
            <asp:RequiredFieldValidator ID="RequiredFieldValidator5" runat="server"
ControlToValidate="VariableName" ErrorMessage="The Variable Name is required."
SetFocusOnError="true"></asp:RequiredFieldValidator>
        </td>
    </tr>
</table>
```

8. Go back to `UpdateAS400Wizard.ascx` and add a 'Register' tag that points to the `UpdateAS400ActivityDetails.ascx` web control which you have just created.

   <%@ Register
   Src="~/Web/UI/Activities/UpdateAS400/UpdateAS400ActivityDetails.ascx"
   TagPrefix="ca" TagName="CustomActivityDetails" %>



Add Details.ascx

9. In the code behind of the `UpdateAS400ActivityDetails.ascx` , bind the activity properties to the textbox value or any other control value. This code should be placed in the `OnPreRender` method.

   See the example below:

```
protected override void OnPreRender(EventArgs e)
{

        IWorkflowElementDesignerHost host
                =
((PNMsoft.Sequence.Runtime.IServiceProviderEx)this.Page).GetService<IWorkflowElementDe
signerHost>();
        UpdateAS400Activity activity = (UpdateAS400Activity)host.WorkflowElement;

        if (!base.Initialized)
        {

            if (activity.VariableName != null)
            {
                this.VariableName.Text = activity.VariableName.ToString();
            }
            if (activity.ProcessId != null)
            {
                this.ProcessId.Text = activity.processId.ToString();
  }
 base.Initialized = true;
}
 base.OnPreRender(e);
}
```

10. To allow the expression editor to be displayed when the user clicks  , extend the OnClientClick to include show a modal dialog with WorkflowExpressionWizardOptions object.

    See the example below:

```
WorkflowExpressionWizardOptions exprOptionsItem = new WorkflowExpressionWizardOptions(
                this, (Workflow)activity.Parent, activity);
                exprOptionsItem.PageTitle = "Select Process ID";
                exprOptionsItem.WizardTitle = "Process ID";
                exprOptionsItem.ClientValueControl = this.ProcessId.ClientID; //return
value to
                exprOptionsItem.ExpressionFormat = SqExpressionFormat.Object;

                this.processIdButton.OnClientClick =
this.Page.ClientScript.GetShowModalDialogReference(exprOptionsItem);
```

11. Save the values from the page control to the activity properties by overriding the `SaveTo` method.

```
public override void SaveTo(Activity workflowElement)
     {
        UpdateAS400Activity customActivity = (UpdateAS400Activity)workflowElement;

        customActivity.VariableName = VariableName.Text;
        customActivity.ProcessId = ProcessId.Text;
}
```

12. Here is an example of the complete designer code:

```
using PNMsoft.Ex.CustomActivities;
using PNMsoft.Sequence;
using PNMsoft.Sequence.Design.Web;
using PNMsoft.Sequence.Design.Web.UI;
using PNMsoft.Sequence.Design.Web.UI.Expressions;
using PNMsoft.Sequence.Expressions;
using System;
using System.Collections.Generic;
```

```
using System.Linq;
using System.Web;
using System.Web.UI;
using System.Web.UI.WebControls;

namespace CustomActivitiesDesigner
{
    public partial class UpdateAS400ActivityDetails : ActivityDesignerView
    {


     protected override void OnPreRender(EventArgs e)
     {

        IWorkflowElementDesignerHost host
                =
((PNMsoft.Sequence.Runtime.IServiceProviderEx)this.Page).GetService<IWorkflowElementDe
signerHost>();
        UpdateAS400Activity activity = (UpdateAS400Activity)host.WorkflowElement;

        if (!base.Initialized)
        {

            if (activity.VariableName != null)
            {
                this.VariableName.Text = activity.VariableName.ToString();
            }

            if (activity.ProcessId != null)
            {
                this.ProcessId.Text = activity.ProcessId.ToString();
            }

            WorkflowExpressionWizardOptions exprOptionsItem = new
WorkflowExpressionWizardOptions(
            this, (Workflow)activity.Parent, activity);
            exprOptionsItem.PageTitle = "Select Process ID";
            exprOptionsItem.WizardTitle = "Process ID";
            exprOptionsItem.ClientValueControl = this.ProcessId.ClientID; //return
value to
            exprOptionsItem.ExpressionFormat = SqExpressionFormat.Object;

            this.processIdButton.OnClientClick =
this.Page.ClientScript.GetShowModalDialogReference(exprOptionsItem);

            WorkflowExpressionWizardOptions exprOptionsItem2 = new
WorkflowExpressionWizardOptions(
            this, (Workflow)activity.Parent, activity);
            exprOptionsItem2.PageTitle = "Select Variable Name";
            exprOptionsItem2.WizardTitle = "Variable Name";
            exprOptionsItem2.ClientValueControl = this.VariableName.ClientID; //return
value to
            exprOptionsItem2.ExpressionFormat = SqExpressionFormat.Object;
            this.VariableNameButton.OnClientClick =
this.Page.ClientScript.GetShowModalDialogReference(exprOptionsItem2);

            base.Initialized = true;
          }

            base.OnPreRender(e);
        }
```

```
    public  void SaveTo(Activity workflowElement)
    {
       UpdateAS400Activity customActivity = (UpdateAS400Activity)workflowElement;

       customActivity.VariableName = VariableName.Text;
       customActivity.ProcessId = ProcessId.Text;
    }
   }
}
```

Deployment

1. Apply the steps below on the target server(s).
2. Copy the activity assembly (DLL) into the GAC (Global Assembly Cache).
3. Copy the two web controls (ascx files) under the administration directory.
4. Create a folder with your activity name under web\ui\activities and place the files there. For example: *C:\inetpub\wwwroot\Administartion\Web\UI\Activities\UpdateAS400*
5. Copy the image used in the canvas (.png file) into: *C:\Program Files\PNMsoft\Shared Resources\images\Diagram*
6. Copy the designer assembly into the bin of the administration site.
7. Add a line to tblActivityTypes table that matches the new activity information.

   fldGuid – should match the data in the ActivityMetaData attribute.

   fldType – custom activity strong name.

   Here is an SQL example:

```
 if not exists(select * from [tblActivityTypes] where [fldGuid] =
 '8C186E65-A26C-4E02-BA99-1441B8FFB3B2')

  begin
  insert into
[tblActivityTypes]([fldGuid],[fldType],[fldCanStartWorkflowManually])
  values('8C186E65-A26C-4E02-BA99-1441B8FFB3B2',
          'PNMsoft.Ex.CustomActivities.UpdateAS400Activity,
        PNMsoft.Ex.CustomActivities, Version=1.0.0.0, Culture=neutral,
        PublicKeyToken=b89fc47cd0f734c9',
          1)
  End
```

8. Run IISreset to flush the application cache.

   Well done. You have completed your first custom activity!