Dev Talk: Create a Custom Action for Nintex Workflow 2013

Posted by Matt Briggs Oct 3, 2016

Did you know you can create a custom action in Nintex Workflow 2013 using Visual Studio and the Nintex 2013 Platform SDK. With a custom action, you can:

- insert your function into a Nintex workflow
- pass data from the workflow into your function
- return information processed by the function to the workflow where it can be used by workflow actions

The process of building out the custom action, deploying, and testing it takes some time. My record time so far is about 10 hours (while plugged into headphones, drinking a steady flow of black coffee and Coke 0, and not talking to anyone.) So settle in when you're ready to tackle this approach and brew up your favorite coding beverage to help you along.

Although I provide an example Visual Studio solution of a custom action, this topic will walk you through the process I used in building a custom action using the Nintex Platform SDK. In creating your own custom action you will need to read the comments in the code templates to get all of the details. This post provides some of the higher level way-points in creating a project.
This is what might be called a **beta** topic. I'm looking for feedback on how to make this subject clearer and more useful to you. Please feel free to send me your feedback or include your feedback as comments to this post. Thanks, Matt

Table of contents

Downloads:

- To download a copy of the project, see "Install the BasicAction Sample."
- To install the Nintex Platform SDK with the custom action adapter and custom action activity templates, see "Installing and Using the Nintex Workflow 2013 SDK."

The operation of the finished action

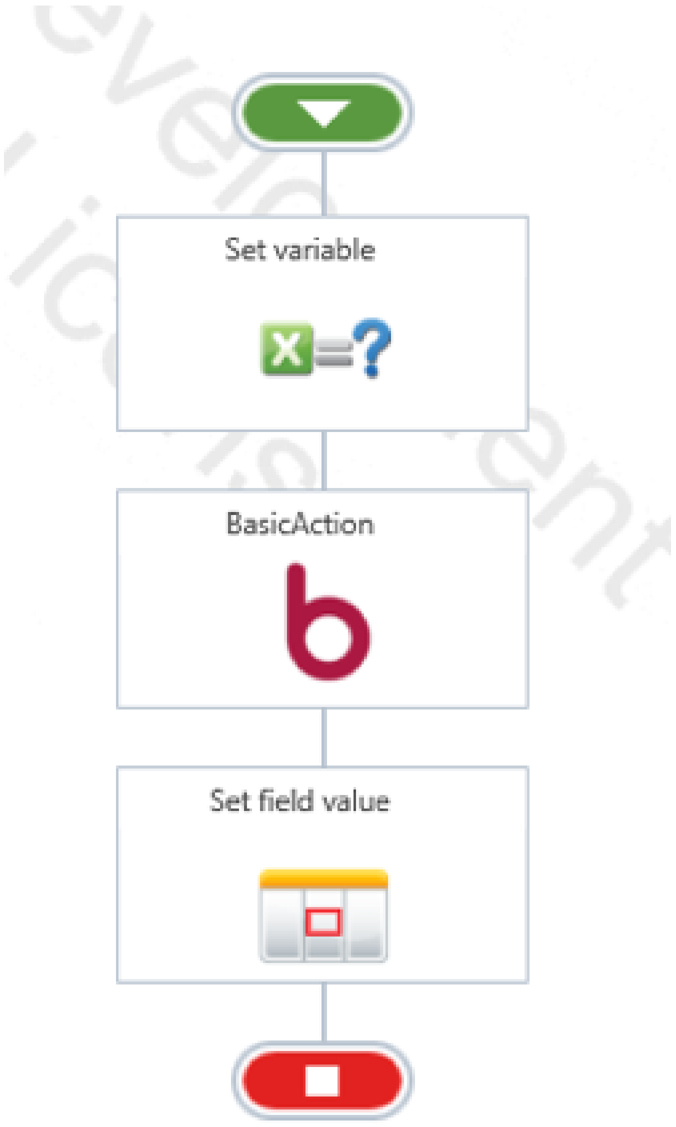
In the example, I created a RESTful call to the Merriam-Webster Dictionary API as a method in a web call class called **WebCallAction**. The example takes three parameters set in the action configuration: a word, definition, and API Key. The action then looks up the word in the API, retrieves the XML definition, and uses XPATH to find the first definition. It then passes the return definition into the Nintex workflow.

You can leverage this sample to integrate your own web service, third-party service, or your own functionality into a Nintex Workflow. As you will see below, the the core functionality of the action is to create an HTTP client, send the call, and then process the call's return XML using the .Net's XML library. To process a JSON return, you might use a library such as Newtonsoft's JSON.Net.

You can also can create other functionality contained in its own class and add it to your own project. You can pass a parameter or set of parameters from the workflow to your method, perform your operation, and then return the product as a parameter that is placed back into the workflow context. This method would allow you to leverage your favorite existing ASP.NET libraries or if you are using a RESTful call, your favorite APIs. For example, I also wrote a JSON to XML conversion utility. In the configuration page, a user specifies a variable that is a multi-line text value that contains JSON which might be the return from the REST action. The adapter class picks up the value, and passes it to activity class that contains the Execute() method. The method instantiates the class that contains my conversion utility, passes it the JSON, and returns the XML translation, that gets handed back to the workflow and passed into a multi-line text variable. I can then use this XML in the workflow using the Nintex Workflow XML tools.

You can add nearly any functionality to the Nintex Workflow as long as you can trigger the action and pass input and retrieve the product.

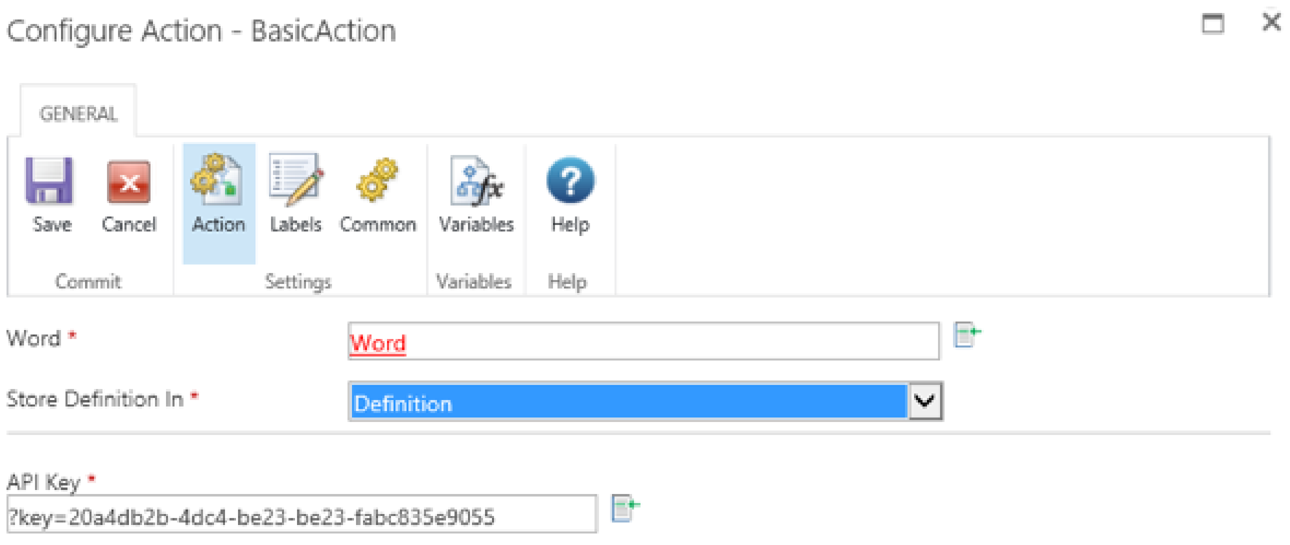
Before we get into the steps of building the BasicAction, lets look at the final result. The following workflow shows the BasicAction added to a simple Nintex workflow. The workflow has two variables defined: **word** and **definition**. The first workflow action, *Set variable*, loads a word from a SharePoint list. And then the *BasicAction* action looks the word up in the Merriam-Webster Dictionary API, and adds the definition to the definition variable. And then the third action, *Set field value*, inserts the definition text into the definition column in the SharePoint list. In building the workflow, drag the BasicAction from the Custom Action group in the toolbar. And then enable the workflow to start from a list item.



When I built and configured the workflow in the Workflow designer, I added the variables defined in the workflow in the configuration page. I also added the API key requested from Merriam-Webster.
I used the following parameters:

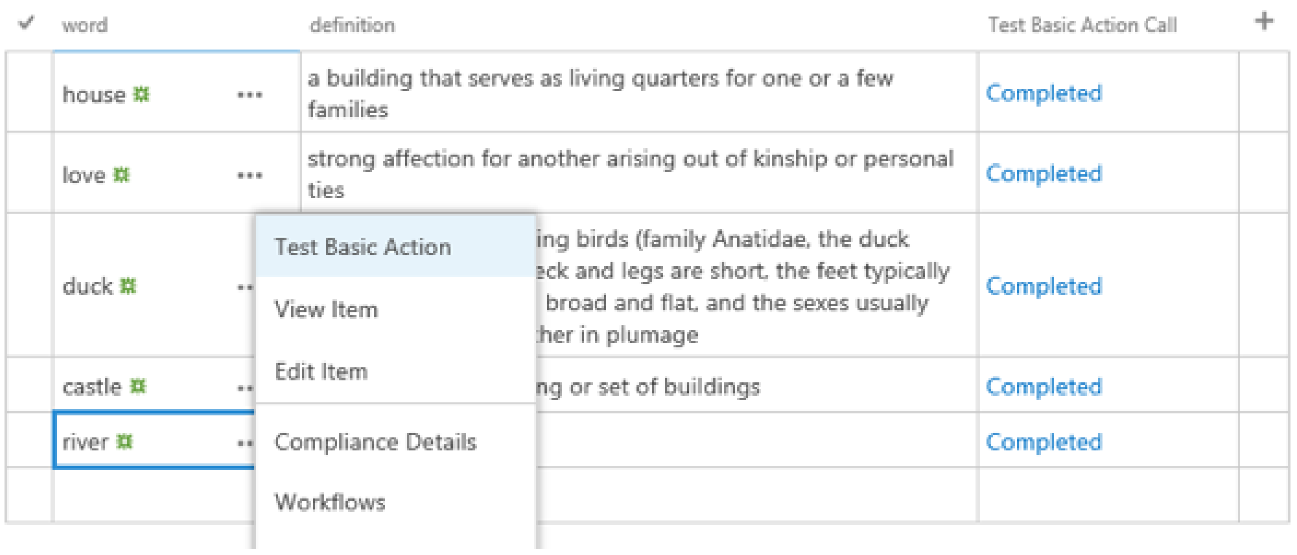
- **CallWord** (string)
- **WordDef** (string)
- **ApiKey** (string)

The parameters are defined in the configuration page. The CallWord and WordDef parameters are held in workflow variables. The API-Key is a string typed in a text box in the action configuration window.



Word *	Word
Store Definition In *	Definition
API Key *	?key=20a4db2b-4dc4-be23-be23-fabc835e9055

To see the action in the context of the SharePoint list, I typed a word in the word column, and then right-clicked the item and select the workflow, titled, **Test Basic Action**. The workflow then executed, and placed the definition in the definition column.



word	definition	Test Basic Action Call
house	a building that serves as living quarters for one or a few families	Completed
love	strong affection for another arising out of kinship or personal ties	Completed
duck	ing birds (family Anatidae, the duck eck and legs are short, the feet typically broad and flat, and the sexes usually her in plumage	Completed
castle	ng or set of buildings	Completed
river		Completed

Now let's turn to the steps of how I built the action.

Design the Action

In designing the action, I first created a console application to build and verify the functionality at the core of the workflow action. I created the functionality in its own class, **Web Call Action.** Once I had clearly defined the input parameters and return and made sure it worked as expected, I added the Web Call Action Class to a custom action project built from the Nintex Platform SDK templates in Visual Studio.
Here is the simple HTTP client (the HttpWebRequest Class in the System.Net library) that makes the web call to the API (**WebCallAction.cs**):

```
using System;
using System.IO;
using System.Net;
using System.Xml;

namespace ConsoleTest
{
    internal class WebCallAction
    {
        public static string RootUrl = "http://www.dictionaryapi.com/api/v1/references/collegiate/xml/";

        public string MakeCall(string CallWord, string ApiKey)
        {
            try
            {
                var WordDef = MakeRequest(CallWord, ApiKey);
                return WordDef;
            }
            catch (Exception e)
            {
                return e.Message;
            }
        }

        public static string MakeRequest(string inWord, string ApiKey)
        {
            try
            {
                var request = WebRequest.Create(RootUrl + inWord + ApiKey) as HttpWebRequest;
                var response = request.GetResponse() as HttpWebResponse;

                var xmlDoc = new XmlDocument();
                xmlDoc.Load(response.GetResponseStream());
                var returnword = xmlDoc.SelectSingleNode("//entry[1]/def/dt[1]").InnerText;

                return returnword;
            }
            catch (Exception e)
            {
                var xmlDoc = new XmlDocument();
                var elem = xmlDoc.CreateElement("error");
                elem.InnerText = e.Message;
                var errorword = XmltoString(xmlDoc);

                return errorword;
            }
        }

        public static string XmltoString(XmlDocument input)
        {
            var stringWriter = new StringWriter();
            var xmlTextWriter = new XmlTextWriter(stringWriter);
            input.WriteTo(xmlTextWriter);
            var output = stringWriter.ToString();

            return output;
        }
    }
}
```

I created a console application to make sure this class worked as expected. Here is the code for the console app (Program.cs):

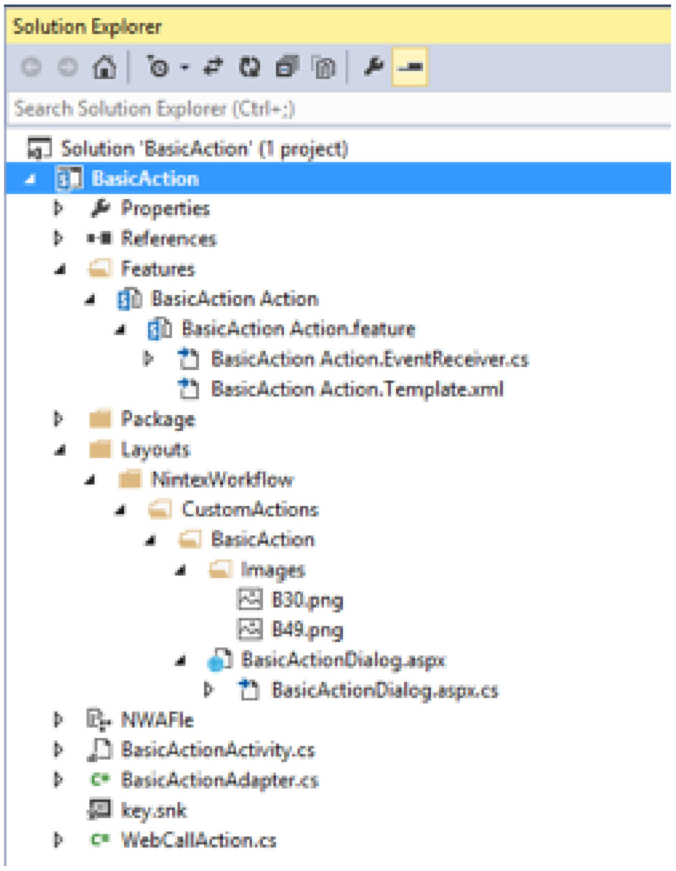
```
using System;
using System.Collections.Generic;
using System.Linq;
using System.Text;

namespace ConsoleTest
{
    class Program
    {
        static void Main(string[] args)
        {
            string resolvedNewWord = "lake";
            string resolvedAPIKey = "?key=<yourkey>";
            string resolvedOutDef = "";
            WebCallAction webster = new WebCallAction();
            string defword = webster.MakeCall(resolvedNewWord, resolvedAPIKey);
            resolvedOutDef = defword;
            Console.Write(resolvedOutDef);
        }
    }
}
```

As you can see from this short sample, the core functionality of my action is relatively simple. The complexity of this project comes from adding the functionality and all of the corresponding code so that SharePoint, Nintex Workflow, and users using the action in the Workflow designer can interact with this functionality. As you can see in the example below of the solution file, the key functionality of the action is only one item among six components.

Build the Custom Action project in Visual Studio

I found that creating the custom action project in Visual Studio was very tricky since the custom action requires five parts, or six if you isolate your functionality in its own class. Before you build your own part, you may want to understand what the project will look like when it is done. I consulted the Nintex Workflow SDK custom action sample a number of times to make sure I had everything in place.

Parts of the project



```
Solution 'BasicAction' (1 project)
  BasicAction
    Properties
    References
    Features
      BasicAction Action
        BasicAction Action.feature
          BasicAction Action.EventReceiver.cs
          BasicAction Action.Template.xml
    Package
    Layouts
      NintexWorkflow
        CustomActions
          BasicAction
            Images
              B30.png
              B49.png
            BasicActionDialog.aspx
              BasicActionDialog.aspx.cs
    NWAFIe
    BasicActionActivity.cs
    BasicActionAdapter.cs
    key.snk
    WebCallAction.cs
```

These are the parts and why you need them:

- **Feature (Event Receiver)** The event receiver registers your action with SharePoint and Nintex Workflow. The feature is added to the SharePoint solution project containing the workflow action adapter, and an event receiver is added to the feature to interact with Nintex Workflow 2013 object model.
- **Layouts (Configuration page)**The configuration page provides a GUI for the user to set up your custom action. Each workflow action adapter has a corresponding configuration page, an ASP.NET (.aspx) web page displayed by the Workflow designer, so that the user can interactively configure the workflow action.
- **BasicActionActivity.cs (Nintex Workflow activity) This class is the core class of the action and contains the Execute() method for the action. A workflow activity is the fundamental building block of SharePoint 2013 workflows. In Nintex Workflow 2013, a workflow action adapts a workflow activity for use in Nintex workflows.**
- **Action definition file (NWA)**An action definition (.nwa) file is a text file that contains an XML fragment which represents the metadata needed by Nintex Workflow 2013 export (.nwp) file to associate a workflow activity to a workflow action adapter for a custom workflow action, and to display information about the resulting custom workflow action.
- **BasicActionAdapter.cs (Nintex Workflow adapter)**

  A workflow action adapter provides the interaction layer between a workflow activity and Nintex Workflow 2013, as well as supporting Nintex-specific features for workflow actions..

- WebCallAction.cs (class that contains the web call method)In this solution the functionality of the action, making the call to the API and finding the definition in the return definition, is isolated in its own class. The Execute() method in the activity class instantiates this class, and then access the methods that perform the core functions of the custom action.

If you would like more information, you can find a more detailed overview of the parts of the BasicAction in the SDK.

Build the empty project

To build the project, I relied on the templates in the Nintex Platform 2013 SDK. I had already installed the SDK, but if you haven't done so, you will need to install the Visual Studio Templates. I then created the workflow adapter. And then in the same project, I created the workflow activity, moved the activity class file into the adapter project, and then removed the activity files. For a step-by step-process, you may want to refer to the "How to create a custom action" in the SDK.

Code the action

With an empty custom action project built from the adapter template and with the empty activity class file added to the adapter project, I added the class that contained my functionality, WebCallAction.cs. To use my web call, I knew I had to pass two variables to the class from the workflow context, the word (CallWord), and the API Key (ApiKey), and in turn the definition (WordDef) would returned by the WebCallAction would be inserted back into a workflow. I planned to use a workflow variable.

The SDK templates provides inline instructions as comments in the code. I found that that it was helpful to have two copies of the project open. One was an existing custom action project, and the second the empty project I had just built. And I also referred to the Workflow SDK. I also drafted a topic, "Code the custom action," to help guide you through the process of creating each of the elements.
The steps here may seem cryptic if you are not using the templates in the Nintex SDK. The templates contain inline comments that will help guide you. The first time through I found myelf lost in the trees, and the sequence in this post is to provide you a bit a forest-level view. To install the Nintex Platform SDK with the custom action adapter and custom action activity templates, see "Installing and Using the Nintex Workflow 2013 SDK."

Code the activity class

I started by adding the parameters as properties in the Activity class.

```
public static DependencyProperty NewWordProperty = DependencyProperty.Register("NewWord",
typeof(string),  typeof(BasicActionActivity));

public static DependencyProperty OutDefProperty = DependencyProperty.Register("OutDef",  typeof(string),
typeof(BasicActionActivity));

public static DependencyProperty APIKeyProperty = DependencyProperty.Register("APIKey",  typeof(string),
typeof(BasicActionActivity));
```

And then I added the accessors for the properties:

```
public string NewWord {
     get { return (string)base.GetValue(BasicActionActivity.NewWordProperty); }
     set { base.SetValue(BasicActionActivity.NewWordProperty, value); }
    }
public string OutDef {
    get { return (string)base.GetValue(BasicActionActivity.OutDefProperty); }
    set { base.SetValue(OutDefProperty, value); }
    }
public string APIKey {
    get { return (string)base.GetValue(BasicActionActivity.APIKeyProperty); }
    set { base.SetValue(BasicActionActivity.APIKeyProperty, value); }
    }
```

Code the adapter class

In the adapter class, I continued to thread the parameters through the project by adding the fields:

```
private const string NewWordPropertyName = "NewWord";
private const string OutDefPropertyName = "OutDef";
private const string APIKeyPropertyName = "APIKey";
```

I retrieve the default configuration for the workflow action, and then instantiate the **NWActionConfig** object that represents the default configuration for the workflow action, and then I created an array that contains the parameters.

```
c.Parameters[0] = new ActivityParameter();
c.Parameters[0].Name = NewWordPropertyName;
c.Parameters[0].PrimitiveValue = new PrimitiveValue();
c.Parameters[0].PrimitiveValue.Value = string.Empty;
c.Parameters[0].PrimitiveValue.ValueType = SPFieldType.Text.ToString();

// Note using NWWorkflowVariable is necessary for the API return to be stored in a Nintex Workflow variable set in the
action configuration screen.
c.Parameters[1] = new ActivityParameter();
c.Parameters[1].Name = OutDefPropertyName;
c.Parameters[1].Variable = new NWWorkflowVariable();
c.Parameters[2] = new ActivityParameter();
c.Parameters[2].Name = APIKeyPropertyName;
c.Parameters[2].PrimitiveValue = new PrimitiveValue();
c.Parameters[2].PrimitiveValue.Value = string.Empty;
c.Parameters[2].PrimitiveValue.ValueType = SPFieldType.Text.ToString();
```

I assigned the activity parameter values to the workflow activity.

```
parameters[NewWordPropertyName].AssignTo(activity, BasicActionActivity.NewWordProperty, context);

parameters[OutDefPropertyName].AssignTo(activity, BasicActionActivity.OutDefProperty, context);

parameters[APIKeyPropertyName].AssignTo(activity, BasicActionActivity.APIKeyProperty, context);
```

For each property, I retrieved and updated the values in the configuration.

```
parameters[NewWordPropertyName].RetrieveValue(context.Activity, BasicActionActivity.NewWordProperty, context);

parameters[OutDefPropertyName].RetrieveValue(context.Activity, BasicActionActivity.OutDefProperty, context);

parameters[APIKeyPropertyName].RetrieveValue(context.Activity, BasicActionActivity.APIKeyProperty, context);
```

Code the configuration page

And then I add the data entry fields in the Configuration page. I open the ASPX page.

I add to the **TPARetrieveConfig()** function:

```
setRTEValue('<%=NewWordProperty.ClientID%>', configXml.selectSingleNode("/NWActionConfig/Parameters/Parameter[@Name='NewWord']/PrimitiveValue/@Value").text);

var drpOutDef = document.getElementById("<%= OutDefProperty.ClientID %>");

drpOutDef.value = configXml.selectSingleNode("/NWActionConfig/Parameters/Parameter[@Name='OutDef']/Variable/@Name").text;

setRTEValue('<%=APIKeyProperty.ClientID%>', configXml.selectSingleNode("/NWActionConfig/Parameters/Parameter[@Name='APIKey']/PrimitiveValue/@Value").text);
```

I added the **TPAWriteConfig()** function:

```
configXml.selectSingleNode("/NWActionConfig/Parameters/Parameter[@Name='NewWord']/PrimitiveValue/@Value").text = getRTEValue('<%=NewWordProperty.ClientID%>');

var drpOutDef = document.getElementById("<%= OutDefProperty.ClientID %>");

configXml.selectSingleNode("/NWActionConfig/Parameters/Parameter[@Name='OutDef']/Variable/@Name").text = drpOutDef.value;

configXml.selectSingleNode("/NWActionConfig/Parameters/Parameter[@Name='APIKey']/PrimitiveValue/@Value").text = getRTEValue('<%=APIKeyProperty.ClientID%>');
```

Then I added to the **contentBody** section:

```
<Nintex:ConfigurationProperty runat="server" FieldTitle="Word" RequiredField="True">

<TemplateControlArea>

<Nintex:SingleLineInput clearFieldOnInsert="true" runat="server" id="NewWordProperty"></Nintex:SingleLineInput>

</TemplateControlArea>

</Nintex:ConfigurationProperty>

<Nintex:ConfigurationProperty runat="server" FieldTitle="Store Definition In" RequiredField="True">
```

```
<TemplateControlArea>

<Nintex:VariableSelector ID="OutDefProperty" runat="server"
IncludeTextVars="True"></Nintex:VariableSelector>

</TemplateControlArea>

</Nintex:ConfigurationProperty>

</TemplateRowsArea>

</Nintex:ConfigurationPropertySection>

<Nintex:ConfigurationProperty runat="server" FieldTitle="API Key" RequiredField="True">

<TemplateControlArea>

<Nintex:SingleLineInput clearFieldOnInsert="true" runat="server"
id="APIKeyProperty"></Nintex:SingleLineInput>

</TemplateControlArea>
```

Add the graphics to the layouts folder

The template project places two images in the Layouts folder. One image appears in the Workflow designer toolbox and the other appears on the workflow designer surface. The toolbox icon is 49 pixels. The other is a 30 pixel square. I created my own icons and placed them in the solution file, replacing the existing images. Later, I also update the NWA file with the paths of the updated images. You can find a source of vector icons at: SmartIcons - Smart SVG icon system.

Add the event listener feature

And then I added the event listener feature by following the steps at the "Deploying workflow actions with SharePoint features" topic in the SDK. This topic first had me adding the action definition (.nwa) file, which I will talk about about after I build the event listener feature. First, I followed the instruction about adding the NWA file, and then I followed the instructions for adding the event listener feature.
The event receiver for the SharePoint feature performs the following tasks when the feature is activated in SharePoint:

- Registers the custom workflow action with Nintex Workflow 2013 export (.nwp) file, using the action definition file.
- Enables the custom workflow action for use with the Workflow designer, within the scope of the feature.
- Adds necessary entries in the web.config configuration file for the web application, to authorize the custom workflow activity included with the custom workflow action for use in SharePoint declarative workflows.

When you follow these steps yourself, you will need to add the event receiver following the SDK instructions since it will place a GUID into the code that will help activate the feature. Follow the steps precisely placing the code fragments in the positions indicated in the instructions. When you are done you are going need to insert the full name of the project assembly. This will require that you build your project, and then identify the four-part assembly name from the DLL file. The name may resemble the following string:

```
<ActivityAssembly>MyCompany.WorkflowActivities, Version=1.0.0.0, Culture=neutral,
PublicKeyToken=e5daa473dcec272b</ActivityAssembly>
```

I wrote a short topic that has a tiny console app that uses the **Assembly Name** class and **GetAssemblyName()** method from the **Reflection** library to help identify the assembly name. You can also find a discussion on MSDN about "Understanding and Using Assemblies and Namespaces in .NET."

Add and update the NWA file

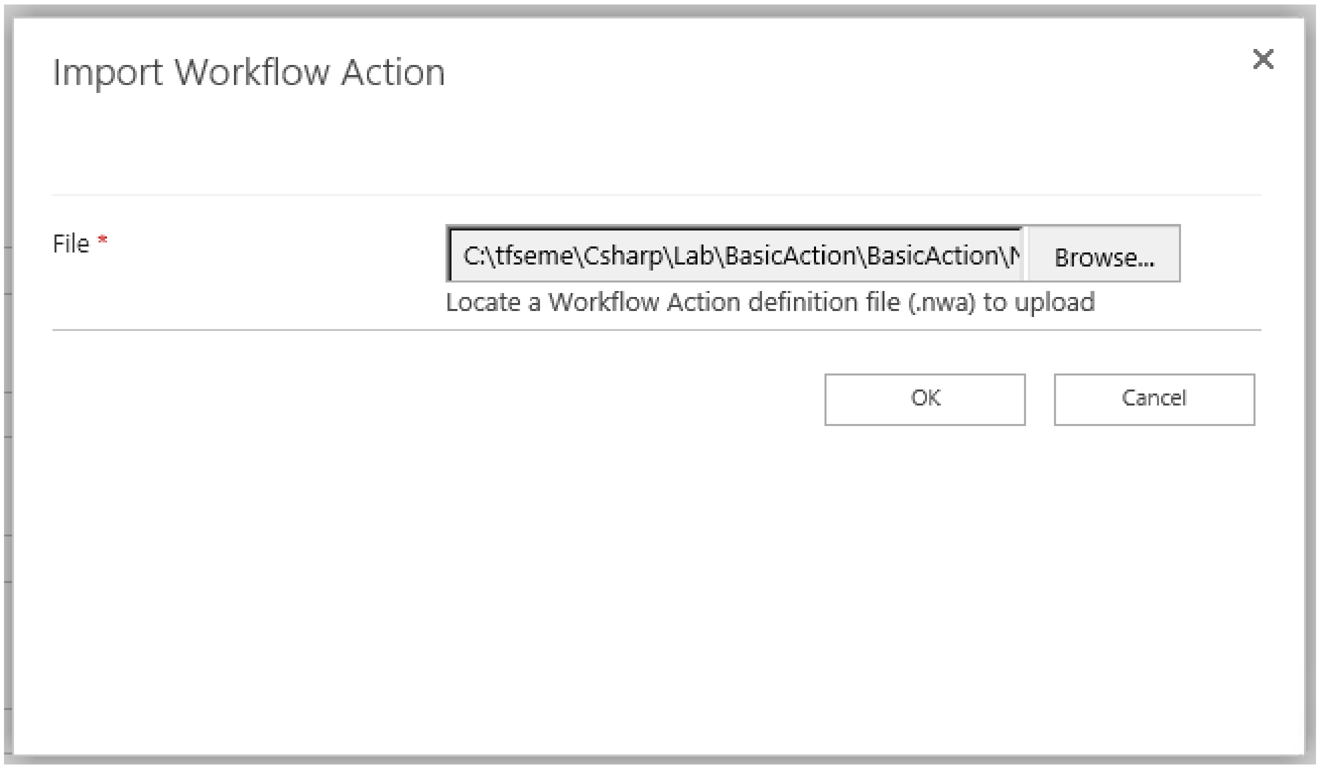
I updated the action definition (.nwa) file with the values from the project. This file will be used when you import the action into the custom action manager. Here is the complete action definition in the new file for the basic action:

```
<NintexWorkflowActivity>
    <Name>BasicAction</Name>
    <Category>Custom Actions</Category>
    <Description>BasicAction</Description>
    <ActivityType>BasicAction.BasicActionActivity</ActivityType>
    <ActivityAssembly>BasicAction, Version=1.0.0.0, Culture=neutral,
PublicKeyToken=f4377886582ac619</ActivityAssembly>
    <AdapterType>BasicAction.BasicActionAdapter</AdapterType>
    <AdapterAssembly>BasicAction, Version=1.0.0.0, Culture=neutral,
PublicKeyToken=f4377886582ac619</AdapterAssembly>
    <HandlerUrl>ActivityServer.ashx</HandlerUrl>
    <Icon>/_layouts/NintexWorkflow/CustomActions/BasicAction/Images/B49.png</Icon>
    <ToolboxIcon>/_layouts/NintexWorkflow/CustomActions/BasicAction/Images/B30.png</
ToolboxIcon>
    <ConfigurationDialogUrl>CustomActions/BasicAction/BasicActionDialog.aspx</ConfigurationDialogUrl>
    <ShowInCommonActions>yes</ShowInCommonActions>
    <DocumentLibrariesOnly>no</DocumentLibrariesOnly>
</NintexWorkflowActivity>
```

Build, deploy, and import to Nintex Workflow Management

I built and deployed the solution to my SharePoint instance. To import the action, I opened SharePoint Central Administration, and navigated to the Nintex Workflow Management group. I clicked **Manage allowed**

actions, then clicked **Add a custom action**, and then clicked **Import workflow action**. I navigated to the action definition file in my custom action solution and clicked **OK**.


Import Workflow Action
File *
C:\tfseme\Csharp\Lab\BasicAction\BasicAction\N
Browse...
Locate a Workflow Action definition file (.nwa) to upload
OK
Cancel

Verify the action works

And then I created a Nintex workflow in SharePoint that uses the action and verify that it works, as you can see at the beginning of this post.
You can also find an additional custom action in the Nintex Platform SDK, "DeployWorkflowActionWithFeature." The sample represents a SharePoint 2013 solution package, in which a single feature, scoped to a web application, performs the following tasks:

- Registers the custom workflow action with Nintex Workflow 2013.
- Enables the custom workflow action for use with the Workflow designer, within the SharePoint farm.
- Adds necessary entries in the web.config configuration file for the web application, to authorize the custom workflow activity included with the custom workflow action for use in SharePoint declarative workflows.

I found the very first action to difficult to get together. I'm glad I have two monitors, since I consulted an existing action project repeatedly and like any complex project found I'd left this or that part out. However once it was built, the second time was much easier. Good luck, and let me know how it goes.
2148 Views

Erwin Bakels
Dec 19, 2016 7:14 AM
Hi Matt,

#3 was automaticly fixed after the modifications from #1 and #2.
(Activating the Feature adds the assembly to the web.config)

Erwin

Matt Briggs *in response to* *Erwin Bakels* *on page 15*
Dec 16, 2016 10:56 PM
Hi Erwin,
Thanks for passing these bugs along to me. I updated the first two in the Event Receiver and found a few other things to update. I was able to deploy with no issue. I'm not sure of the cause of the cause for #3 but will do some research. Thanks again for letting know about these issues. Matt

Erwin Bakels *in response to* *Matt Briggs* *on page 16*
Dec 16, 2016 9:12 AM
Hi Matt,

Thanx for the feedback.

I was digging through some ULS logging / code and found 3 things:

1. The feature could not activate because the "BasicAction.nwa" file could not be found. It was looking for it in the NWAFle subdirectory. I changed the path in the projectfile: SharePointProjectItem.spdata (1 bug fixed)

2. There was a typo in the EventeReceiver:
const string adapterType = "BasicAction.BasicActionAdapte";   (Missing 'r')

3. For some reason the BasicAction was not added to the Authorized Assemblies in the web.config

```
<System.Workflow.ComponentModel.WorkflowCompiler>
  <authorizedTypes>
    <targetFx version="v4.0">
      <authorizedType Assembly="BasicAction, Version=1.0.0.0, Culture=neutral,
PublicKeyToken=f4377886582ac619" Namespace="BasicAction" TypeName="BasicActionActivity"
Authorized="True" />
```

After these modifications, i could publish the CustomAction in the workflow.

Erwin

PS: Environment: Server 2012R2, SharePoint 2013, VS2015

Matt Briggs *in response to Erwin Bakels on page 16*
Dec 15, 2016 6:11 PM
Hi Erwin,
Thanks. I find getting my copy of Visual Studio to register properly with the my SharePoint Farm to be a bit of a hassle. You will want to be using an account that has rights to both the SharePoint site and also your database. And if worse comes to worse I will open VS with admin rights. I've run into the error as well, and I can't recall the specific steps I used to resolve it, but yeah, makings sure the DLL is in the GAC is one of the steps.
In terms of the terms of the deprecated warnings, those are with the current implementing of the Nintex Workflow 2013 base-classes used for the activity. I haven't begun work yet on looking at Nintex Workflow 2016 -- but I suspect that those classes will still trigger the warning. There is also some new work coming up that may make the workflow extensibility features a lot easier to use, I hope.
Thanks for your feedback and questions.
Cheers!
Matt

Erwin Bakels
Dec 15, 2016 11:01 AM
Hi Matt,
Great post. Only 2 questions:

1. I get deprecated warnings:
Warning CS0618 'CompositeActivity' is obsolete: 'The System.Workflow.* types are deprecated. Instead, please use the new types from System.Activities.*'
Warning CS0618 'ActivityBind' is obsolete: 'The System.Workflow.* types are deprecated. Instead, please use the new types from System.Activities.*'

Will there be a newer sample application?

2. I can add the Custom Action to the workflow, save it. But cannot publish it:
Failed to publish workflow: The type or namespace name 'BasicAction' could not be found (are you missing a using directive or an assembly reference?)

I already tried to register the DLL to the GAC (again) *gacutil.exe /i BasicAction.dll* and restarting the server, but still no succes.

Any idea what could be wrong?

Mike Matsako
Oct 7, 2016 3:58 PM
Love this type of content. Hope to see more of it!

Matt Briggs *in response to burkslm on page 17*
Oct 4, 2016 3:44 PM
Thanks. Let me know how your custom action project goes!

burkslm
Oct 4, 2016 1:34 PM
This is great! I've passed it on to my fellow developers!!!