Using Open Xml Power Tools from Workflow Foundation (Applying a corporate style to Open Xml Documents Part III)

Antonio Zamora
Staff DotNet
http://blogs.staffdotnet.com/antoniozamora

Download this sample code here (Requirements: Visual Studio 2008, .NET Framework 3.5, Windows PowerShell 1.0, Open Xml Power Tools) Please read readme.txt file provided with the code sample.

Hi people! So far, we have learned how to create a Power Shell script in order to apply a corporate style to a document and how to fire off that Power Shell script from a .NET Application. In this third post of the “Applying a corporate style to Open Xml Documents” series, I will show you how to combine the Power Shell and Workflow Foundation (WF) capabilities in order to process Open Xml Documents transformations. Basically we are going to implement a simple sequential workflow in WF calling two Power Shell scripts in order to apply a corporate style (surprise ☺) to a set of Open Xml Documents and to apply some security features to the same set of documents.

Before going on with this post I would recommend reading the first two blog post of this series (if you haven’t done that yet). It is also recommendable to have some basic knowledge about Workflow Foundation. (Don’t worry, this blog post is a good place to start learning WF if you haven’t done that yet).

Stop talking, start programming

Ok. Let’s go to the action.

Open Visual Studio and create a new empty solution called **WFCmdletExecution.** Add the following C# projects to the solution:

PowerShellHelperLibrary (class library)
- This project will contain all the code required to communicate with our Power Shell scripts and cmdlets.

OXPTWorkFlowLibrary (Sequential Workflow Library)
- This project will contain the sequential workflow to process the Open Xml Documents.

OXPTCommandUI (Windows Desktop Application)
- User Interface to collect the value for the scripts parameters.

OXPTWorkFlowLibraryActivitiesLibrary (Workflow Activity Library)
- Set of Activities to be used at the sequential workflow.

Calling the PowerShell script from a .NET application

Ok, now in the **PowerShellHelperLibrary project** add a reference to the System.Management.Automation dll file located at the Power Shell home directory (usually **C:\Program Files\Reference Assemblies\Microsoft\WindowsPowerShell\v1.0**). Delete the auto-generated Class1.cs file and add a new class called **PowerShellInvoker.** Change the class scope to **Public** and import the System.Collections.ObjectModel, System.Management.Automation and System.Management.Automation.Runspaces namespaces at the beginning of the class file.

Now, add the following method to the PowerShellInvoker class:

```
      public static string RunScript(string script)
      {
            //Runspace for executing PowerShell cmdlets and scripts from an
external application
            Runspace runspace;

            //Configuration class for the runspace
            RunspaceConfiguration rsConfig;

            //Pipeline used to invoke powershell commands
            Pipeline pipeline;

            //Creating the runspace configuration class
            rsConfig = RunspaceConfiguration.Create();

            //Registering the PowerTools for Open Xml snapin
            //in order to call Power Tools for Open Xml cmdlets
            PSSnapInException exception;
            rsConfig.AddPSSnapIn(psSnapIn, out exception);

            //Creating the runspace
            runspace = RunspaceFactory.CreateRunspace(rsConfig);

            //Opening the runspace
            runspace.Open();

            //Creating the pipeline (in order to include the cmdlets and
scripts to execute)
            pipeline = runspace.CreatePipeline();

            //Adding cmdlets and scripts to the pipeline
            foreach (string script in scriptsToExecute)
            {
                pipeline.Commands.AddScript(script);
            }

            //Invoking the pipeline and saving the results
            Collection<PSObject> output = pipeline.Invoke();
            runspace.Close();

            //Displaying the obtained results
            StringBuilder strBuilder = new StringBuilder();
            foreach (PSObject psObj in output)
            {
            strBuilder.AppendLine(psObj.ToString());
            }

            return strBuilder.ToString();
        }
```

Build the **PowerShellHelperLibrary** project.

Creating the Workflow Activities

Go to **OXPTWorkFlowLibraryActivitiesLibrary** project and add a new activity for our Workflow by choosing *Add->Activity* option (which appears after right clicking on the project name at the Solution Explorer**)**. Name the activity as **PSScriptCallActivity** (this is the Activity class we are going to use to call our PowerShell scripts). Delete the auto-generate Activity1.cs file. Right click anywhere in the activity designer and choose the **View code** option.

Change the parent class for **PSScriptCallActivity** from **SequenceActivity** to **Activity** (see following code):

Change this:

```
public partial class PSScriptCallActivity: SequenceActivity
```

To this:

```
public partial class PSScriptCallActivity: Activity
```

Add the following code to the **PSScriptCallActivity** class to define the activity parameters:

```
//Property for setting the Power Shell script path to be invoked
[Browsable(true)]
public string ScriptPath { get; set; }
//Array of string to receive the values for the Power Shell script to be
invoked
[Browsable(true)]
public string[] ScriptParams { get; set; }
```

*Notice the **Browsable** attribute which allows the property to appear on the Properties window.*

Add a reference to the **PowerShellHelperLibrary** project at the **OXPTWorkFlowLibraryActivitiesLibrary** .

Import the **PowerShellHelperLibrary** namespace at the beginning of the **PSScriptCallActivity** code file and add the following method to the class**:**

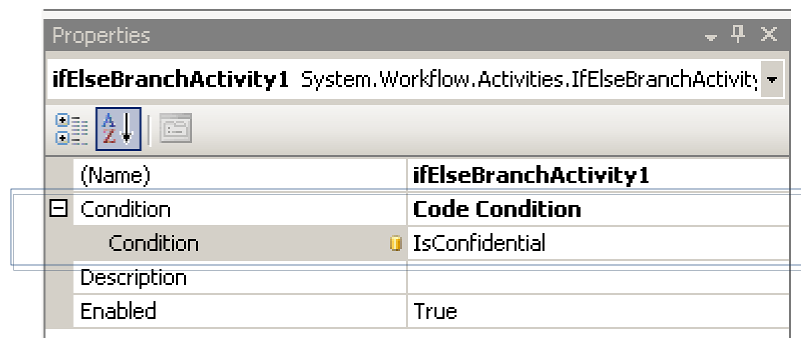
```
protected override ActivityExecutionStatus Execute(ActivityExecutionContext
executionContext)
{
//Using the Array.Aggregate function and Lambda functions to append the
script parameters values to the script path
//CODE BIT taken from:
http://blogs.staffdotnet.com/codebits/blog/default.aspx?id=6&t=Using-
StringAggregate-and-Lambda-functi
string scriptWithParameters = ScriptParams.Aggregate(new
StringBuilder(ScriptPath),
(s, i) => s.Append(" '" + i + "'"), s => s.ToString());

//Using the PowerShellInvoker helper class to invoke the Power Shell script
located at ScriptPath
PowerShellInvoker.RunScript("OpenXml.PowerTools",new string[]
{scriptWithParameters});
return base.Execute(executionContext);
}
```

Now. Build the solution.

Creating the workflow

So far we have defined the class to invoke Power Shell scripts and the workflow activity to call that class . It is time to start working with on the workflow. Go to the **OXPTWorkFlowLibrary** project. Delete the Workflow1.cs file and add a new sequential workflow (with code separation) called **CmdletWF**. Open the toolbox and notice that our **PSScriptCallActivity** is already there. Add a new PSScriptCallActivity to the beginning of the workflow and name this activity as **ApplyCorporateStyleActivity**. Go to the activity properties windows and set the *scriptPath* property to the path of the ApplyCorporateStyle script (created in a previous post as mentioned at the beginning of this post and provided along with the code for this tutorial-). For this tutorial purposes we recommend copying this script in an easy to remember location like c:\.

Now add a **IfElse** Activity below the activity we have just added. Select the **ifElse** activity left branch an open its Properties Window, then go to the *Condition* property and select the “Code Condition” option. Expand the *Condition* property and write the name for the method you want to invoke when reaching the **ifElseActivity**. For this example name this method **IsConfidential** as shown in the following image:



Pres the Enter key and add the following code to the *IsConfidential* method (please notice that constant definition goes outside the method body):

```
//Constant member for defining the index for watermark text parameter at
ApplyCorporateStyleActivity parameter list
const int WATERMARK_PARAMETER_POSITION = 2;
```

```
private void IsConfidential(object sender, ConditionalEventArgs e)
{
string watermarkText =
ApplyCorporateStyleActivity.ScriptParams[WATERMARK_PARAMETER_POSITION];
//We decide if a document is confidential or not by looking at the watermark
text to be applied value
            if (watermarkText.ToLower().Equals("confidential"))
            {
                e.Result = true;
            }
            else
            {
                e.Result = false;
            }
}
```

In this conditional method we are returning true if the watermark text to be applied to the documents is equal to “confidential” and false if not.

Let's go again to the workflow designer and add a new **PSScriptActivity** to the **ifElse** Activity left branch. Name it **SecurityActivity.** Go to Properties windows and set the scriptPath property to the path for the Power Shell *apply-security* script (provided along with the code for this sample) The same as the apply-corporate-style script we recommend copying this script to an easy to remember path. Add a **Terminate** Activity to the **ifElse** Activity right branch to indicate that nothing is going to happen if watermark text for the documents is not equal to "confidential" (we could use a custom property instead of the watermak text to define if we want to secure a document or not).

Defining the workflow parameters

Remember that this workflow is going to be use to call Power Shell scripts which receive parameters In WF, workflow parameters are define by using class Properties. Add the following code to the **CmdletWF** workflow class in order to define the required parameters for the workflow (and so for the PowerShell scripts):

```
//Path to the template document the get the corporate style from
public string TemplateDocumentPath { get; set; }
//Path to the documents to apply the corporate style and security options
public string DocumentsToModifyPath { get; set;}
//Watermark text to apply to the documents
public string WatermarkText { get; set; }
//Path to the digital certificate to sign the documents
public string DigitalCertificatePath { get; set; }
```

Passing the workflow parameter values to the workflow activities

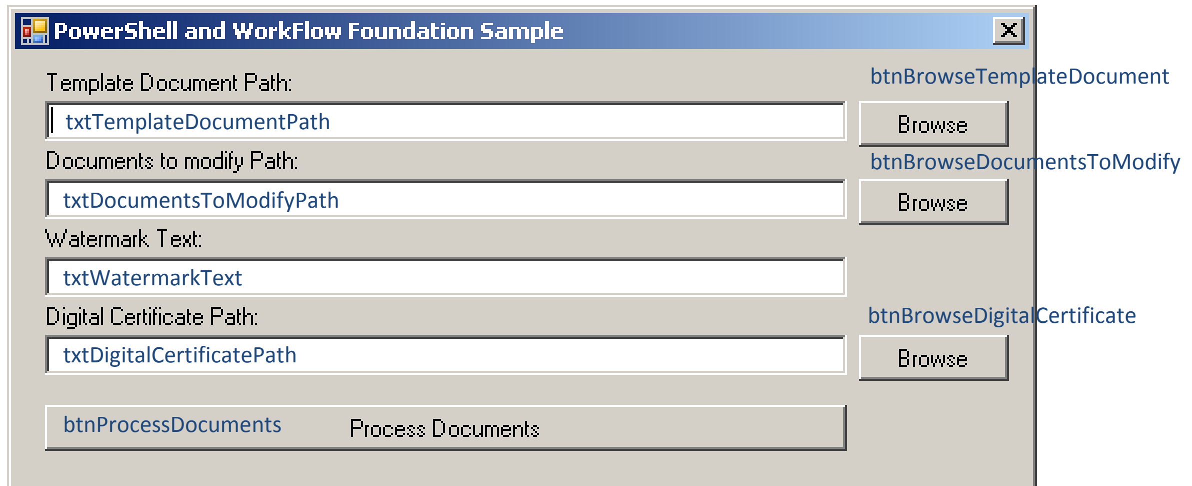
We need to pass the workflow parameter values to the PowerShell invoking activies. In order to do that add a **Code** activity (from the toolbox) before **ApplycorporateStyleActivity**. Name it **InitActivity**.

Double click the **InitActivity** activity and add the following code to the *InitActivity_ExecuteCode* method body:

```
private void InitActivity_ExecuteCode(object sender, EventArgs e)
{
//Setting the Apply Corporate Style Activity params from the corresponding
Workflows properties
string[] applyCorporateStyleActivityParams =
{TemplateDocumentPath,DocumentsToModifyPath,WatermarkText};
ApplyCorporateStyleActivity.ScriptParams = applyCorporateStyleActivityParams;
//Setting the Secure Documents Activity params from the corresponding
Workflows properties
string[] secureDocumentsActivityParams = { DigitalCertificatePath,
DocumentsToModifyPath };
SecurityActivity.ScriptParams = secureDocumentsActivityParams;
}
```

Getting the Workflow (Power Shell scripts) parameters values from the user

Go to the **OXPTCommandUI** project. Delete the form created automatically by the IDE and add a new one named frmCommandUI. Add and arrange the necessary controls to make your form look like the following picture (Control names in blue):.



Go to the Main method at Program.cs file, look for the following line of code:

```
Application.Run(new Form1());
```

And change it to:

```
Application.Run(new frmCommandUI());
```

Now add a new OpenFileDialog control to the form and name it browseDocumentsDialog. Double click on btnBrowseTemplateDocument button and add the following code the the Click delegate handler method:

```
private void btnBrowseTemplateDocument_Click(object sender, EventArgs e)
{/// Browsing for the Template Document Path
browseDocumentsDialog.FileName = string.Empty;
browseDocumentsDialog.ShowDialog();
if (browseDocumentsDialog.FileName.Trim().Length > 0)
{
txtTemplateDocumentPath.Text = browseDocumentsDialog.FileName;
}
}
```

Double click on btnBrowseDocumentsToModify button and add the following code the the Click delegate handler method:

```
private void btnBrowseDocumentsToModify_Click(object sender, EventArgs e)
{/// Browsing for the Documents to Modify Path
browseDocumentsDialog.FileName = string.Empty;
browseDocumentsDialog.ShowDialog();
if (browseDocumentsDialog.FileName.Trim().Length > 0)
{
txtDocumentsToModifyPath.Text = browseDocumentsDialog.FileName;
}
```

```
}
```

Double click on btnBrowseDigitalCertificate button and add the following code the the Click delegate handler method:

```
private void btnBrowseDigitalCertificate_Click (object sender, EventArgs e)
{/// Browsing for the Digital Certificate Path
browseDocumentsDialog.FileName = string.Empty;
browseDocumentsDialog.ShowDialog();
if (browseDocumentsDialog.FileName.Trim().Length > 0)
{
txtDigitalCertificatePath.Text = browseDocumentsDialog.FileName;
}
}
```

Add a reference to the **OXPTWorkFlowLibrary** project and to the System.Workflow.Activities, System.Workflow.ComponentModel and System.Workflow.Runtime assemblies and import the corresponding namespaces (named after the assemblies) at the beginning of the form code file. Double click the “Process Documents” button and the following code to the Click event handler method:

```
private void btnProcessDocuments_Click(object sender, EventArgs e)
        {
///Initializing the dictionary to pass the params values to the Workflow
Dictionary<string, object> workflowParameters =
                new Dictionary<string, object>();

//Seting the params values from the corresponding controls at the form
workflowParameters.Add("TemplateDocumentPath", txtTemplateDocumentPath.Text);
workflowParameters.Add("DocumentsToModifyPath",
txtDocumentsToModifyPath.Text);
workflowParameters.Add("WatermarkText", txtWatermarkText.Text);
workflowParameters.Add("DigitalCertificatePath",
txtDigitalCertificatePath.Text);

//Creating a new workflow execution engine (runtime)
WorkflowRuntime wr = new WorkflowRuntime();
//Start the workflow execution engine
wr.StartRuntime();
//Initializing an instance of our process documents workflow
WorkflowInstance processDocumentsWorkflow =
wr.CreateWorkflow(typeof(CmdletWF), workflowParameters);
//Start the process documents workflow
processDocumentsWorkflow.Start();
}
```

We are done. Build the solution. Execute the project and provide the required parameter values. Once you click the “Process Documents” button the Open Xml Documents located at “Document to modify path” will have all the same corporate style (copied from the template document located at “Template document path”) and they will be locked and digital signed depending if you indicated “confidential” as the Watermark Text .

Coming next

So far we have learned how to call local Power Shell scripts but… what about if they are located remotely? Don't worry. We are going to add Windows Communication Foundation to this blog series in order to do that. Stay tune!!!