A. Leonard, *Building Custom Tasks for SQL Server Integration Services*
https://doi.org/10.1007/978-1-4842-6482-9_14

14. Implement New Connection

Andy Leonard[1]
(1) Farmville, VA, USA

In Chapter 13, we implemented `GeneralView` and made a good start of implementing `SettingsView`. This chapter is all about completing the `SettingsView SourceConnection` property implementation, which is the most complex part of building the Execute Catalog Package Task Complex editor (thus far, at least).

Implementing New Connection Functionality

To implement "<New Connection...>" functionality in the `SourceConnection` dropdown, we first need to add an event handler for changes to manage and respond changes in PropertyGrid values.

Begin by adding the code in Listing 14-1 to the `SettingsView` class:

```
private void propertyGridSettings_PropertyValueChanged(object s, System.
```

Listing 14-1 Adding the propertyGridSettings_PropertyValueChanged event handler

Once added, the code appears as shown in Figure 14-1:

```
private void propertyGridSettings_PropertyValueChanged(object s, System.Windows.Forms.PropertyValueChangedEventArgs e) { }
```

Figure 14-1 The propertyGridSettings_PropertyValueChanged event handler method added

After adding the method to handle `propertyGridSettings_PropertyValueChanged` events, the next step is to update the `SettingsView InitializeComponent` method to assign the event handler to respond to the `propertyGridSettings' PropertyValueChanged` event, adding the code in Listing 14-2:

```
this.settingsPropertyGrid.PropertyValueChanged += new System.Windows.For
```

Listing 14-2 Assigning propertyGridSettings_PropertyValueChanged to propertyGrid

Once added, the code appears as shown in Figure 14-2:

```
private void InitializeComponent()
{
    this.settingsPropertyGrid = new System.Windows.Forms.PropertyGrid();
    this.SuspendLayout();
    // settingsPropertyGrid
    this.settingsPropertyGrid.Anchor = ((System.Windows.Forms.AnchorStyles)((((System.Windows.Forms.AnchorStyles.Top
                | System.Windows.Forms.AnchorStyles.Bottom)
                | System.Windows.Forms.AnchorStyles.Left)
                | System.Windows.Forms.AnchorStyles.Right)));
    this.settingsPropertyGrid.Location = new System.Drawing.Point(3, 0);
    this.settingsPropertyGrid.Name = "settingsPropertyGrid";
    this.settingsPropertyGrid.PropertySort = System.Windows.Forms.PropertySort.Categorized;
    this.settingsPropertyGrid.Size = new System.Drawing.Size(387, 400);
    this.settingsPropertyGrid.TabIndex = 0;
    this.settingsPropertyGrid.ToolbarVisible = false;
    // SettingsView
    this.Controls.Add(this.settingsPropertyGrid);
    this.Name = "SettingsView";
    this.Size = new System.Drawing.Size(390, 400);
    this.ResumeLayout(false);
    this.settingsPropertyGrid.PropertyValueChanged += new System.Windows.Forms.PropertyValueChangedEventHandler(this.propertyGridSettings_PropertyValueChanged);
}
```

Figure 14-2 The propertyGridSettings_PropertyValueChanged event handler method assigned to the propertyGridSettings.PropertyValueChanged event

The `propertyGridSettings_PropertyValueChanged` event handler method is now called whenever *any* property value changes in the `propertyGridSettings` PropertyGrid.

The next step is tuning the logic of the `propertyGridSettings_PropertyValueChanged` event handler method so it responds as we desire to the value changing in *each* `propertyGridSettings` property. We begin with detecting – and responding to – changes in the `SourceConnection` property.

SourceConnection Property Value Changes

The first step in detecting and responding to changes in the `SourceConnection` property is detecting changes to the `SourceConnection` property. Add the code in Listing 14-3 to the `propertyGridSettings_PropertyValueChanged` event handler method:

```
if (e.ChangedItem.PropertyDescriptor.Name.CompareTo("SourceConnection")
```

Listing 14-3 Respond to SourceConnection property changes

Once added, the `propertyGridSettings_PropertyValueChanged` event handler method appears as shown in Figure 14-3:

```
private void propertyGridSettings_PropertyValueChanged(object s
                                                       , System.Windows.Forms.PropertyValueChangedEventArgs e)
{
    if (e.ChangedItem.PropertyDescriptor.Name.CompareTo("SourceConnection") == 0) { }
}
```

Figure 14-3 SourceConnection property changes only, please

The next step is to check if "New Connection" was selected. Before proceeding, however, add the code in Listing 14-4 to define the NEW_CONNECTION string constant in `SettingsView`:

```
private const string NEW_CONNECTION = "<New Connection...>";
```

Listing 14-4 Add the NEW_CONNECTION string constant to SettingsView

Once added, the declarations in `SettingsView` appears as shown in Figure 14-4:

```
public partial class SettingsView : System.Windows.Forms.UserControl, IDTSTaskUIView
{
    private SettingsNode settingsNode = null;
    private System.Windows.Forms.PropertyGrid settingsPropertyGrid;
    private ExecuteCatalogPackageTask.ExecuteCatalogPackageTask theTask = null;
    private System.ComponentModel.Container components = null;

    private const string NEW_CONNECTION = "<New Connection...>";
```

Figure 14-4 Declaring NEW_CONNECTION constant in SettingsView

Add the code in Listing 14-5 inside the previous `if` conditional statement in the `propertyGridSettings_PropertyValueChanged` method to determine if "New Connection" was clicked:

```
if (e.ChangedItem.Value.Equals(NEW_CONNECTION)) { }
```

Listing 14-5 Checking for "New Connection" click

Once the check for a "New Connection" click is added, the `propertyGridSettings_PropertyValueChanged` method appears as shown in Figure 14-5:

```
private void propertyGridSettings_PropertyValueChanged(object s
                                                       , System.Windows.Forms.PropertyValueChangedEventArgs e)
{
    if (e.ChangedItem.PropertyDescriptor.Name.CompareTo("SourceConnection") == 0)
    {
        if (e.ChangedItem.Value.Equals(NEW_CONNECTION)) { }
    }
}
```

Figure 14-5 Checking for New Connection clicks

Before we add the code to create a new ADO.Net connection manager, we must first add and initialize (a little later) an `IDtsConnectionService` member named `connectionService` to the `SettingsView` class by adding the code in Listing 14-6 to declare the `connectionService` member:

```
private IDtsConnectionService connectionService;
```

Listing 14-6 Adding connectionService to SettingsView

When the code in Listing 14-6 is added, the `SettingsView` declarations appear as shown in Figure 14-6:

```
public partial class SettingsView : System.Windows.Forms.UserControl, IDTSTaskUIView
{
    private SettingsNode settingsNode = null;
    private System.Windows.Forms.PropertyGrid settingsPropertyGrid;
    private ExecuteCatalogPackageTask.ExecuteCatalogPackageTask theTask = null;
    private System.ComponentModel.Container components = null;

    private const string NEW_CONNECTION = "<New Connection...>";
    private IDtsConnectionService connectionService;
```

Figure 14-6 connectionService added to SettingsView

The next step is to initialize the `connectionService IDtsConnectionService` member by adding the code in Listing 14-7 to the `SettingsView OnInitialize` method:

```
connectionService = (IDtsConnectionService)connections;
```

Listing 14-7 Initializing connectionService in OnInitialize

When the code in Listing 14-7 is added, the `SettingsView OnInitialize` method appears as shown in Figure 14-7:

```
public virtual void OnInitialize(IDTSTaskUIHost treeHost
                        , System.Windows.Forms.TreeNode viewNode
                        , object taskHost
                        , object connections)
{
    if (taskHost == null)
    {
        throw new ArgumentNullException("Attempting to initialize the ExecuteCatalogPackageTask UI with a null TaskHost");
    }

    if (!(((TaskHost)taskHost).InnerObject is ExecuteCatalogPackageTask.ExecuteCatalogPackageTask))
    {
        throw new ArgumentException("Attempting to initialize the ExecuteCatalogPackageTask UI with a task that is not a ExecuteCatalogPackageTask.");
    }

    theTask = ((TaskHost)taskHost).InnerObject as ExecuteCatalogPackageTask.ExecuteCatalogPackageTask;

    this.settingsNode = new SettingsNode(taskHost as TaskHost, connections);

    settingsPropertyGrid.SelectedObject = this.settingsNode;

    connectionService = (IDtsConnectionService)connections;
}
```

Figure 14-7 connectionService initialized in SettingsView OnInitialize

Add the code in Listing 14-8 inside the previous `if` conditional statement – `if (e.ChangedItem.Value.Equals(NEW_CONNECTION))` – in the `propertyGridSettings_PropertyValueChanged` method to respond to "New Connection" clicks:

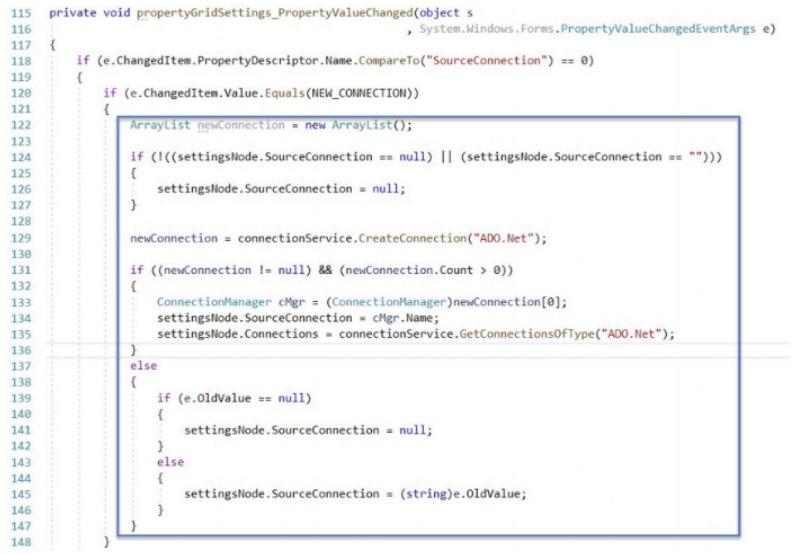
```
ArrayList newConnection = new ArrayList();
if (!((settingsNode.SourceConnection == null) || (settingsNode.SourceCon
  {
```

```
      settingsNode.SourceConnection = null;
    }
  newConnection = connectionService.CreateConnection("ADO.Net");
  if ((newConnection != null) && (newConnection.Count > 0))
  {
    ConnectionManager cMgr = (ConnectionManager)newConnection[0];
    settingsNode.SourceConnection = cMgr.Name;
    settingsNode.Connections = connectionService.GetConnectionsOfType("ADO
  }
  else
  {
    if (e.OldValue == null)
    {
      settingsNode.SourceConnection = null;
    }


    else
    {
      settingsNode.SourceConnection = (string)e.OldValue;
    }
  }
```

Listing 14-8 Respond to “New Connection” clicks

When the code in Listing 14-8 is added, the `propertyGridSettings_PropertyValueChanged` method appears as shown in Figure 14-8:



```
149     }
150 }
```

Figure 14-8 Responding to New Connection clicks

On line 118, the code checks to see if the change that triggered the call to the `propertyGridSettings_PropertyValueChanged` method was a change to the SourceConnection property. Line 120 checks to see if the developer clicked the "<New Connection…>" item in the `SourceConnection` property dropdown. If the developer clicked the "<New Connection…>" item in the `SourceConnection` property dropdown, a new ArrayList variable named `newConnection` is declared on line 122.

On lines 124–127, the `settingsNode SourceConnection` value is set to null if and only if the current `settingsNode SourceConnection` value is *not* null or empty.

On line 129, a new ADO.Net connection manager is created. Creating a new ADO.Net connection manager involves

- Creating a new generic ADO.Net connection manager
- Adding the new generic ADO.Net connection manager to the SSIS package's connection collection
- Opening the editor for the new ADO.Net connection manager so the SSIS developer can configure the connection manager

On lines 131–136, the `settingsNode SourceConnection` value is assigned to the new connection if the `newConnection ArrayList` is not null and contains at least one value. The `settingsView.Connections` property is reloaded to add the new connection manager. If the `newConnection ArrayList` is null or contains no values, the code on lines 139–146 restores the previous value to the `settingsNode SourceConnection` value.

The next step is to test the code.

Let's Test It!

Before testing, let's define a couple use cases:

- Use case 1: As before, we may configure the Execute Catalog Package Task to use an ADO.Net connection manager that was previously configured in the test SSIS package. We may configure the FolderName, ProjectName, and PackageName properties on the Settings page to an SSIS package deployed to the SSIS Catalog.

Assert: Test SSIS package debug execution succeeds and SSIS package deployed to the SSIS Catalog executes.

- Use case 2: We may configure the Execute Catalog Package Task to use a *new* ADO.Net connection manager by clicking "<New Connection…>" in the Settings page SourceConnection property dropdown. The "Configure New ADO.Net Connection" dialog will display and allow the developer to configure a new ADO.Net connection manager. As in use case 1, we configure the FolderName, ProjectName, and PackageName properties on the Settings page to an SSIS package deployed to the SSIS Catalog.

Assert: A new ADO.Net connection manager is added to the Test SSIS package. Test SSIS package debug execution succeeds and SSIS package deployed to the SSIS Catalog executes.

Use Case 1

Build the solution and then open the test SSIS project. To test use case 1, add an Execute Catalog Package Task to the control, open the editor, and then configure the task to execute an SSIS package in the SSIS Catalog, as shown in Figure **14-9**:

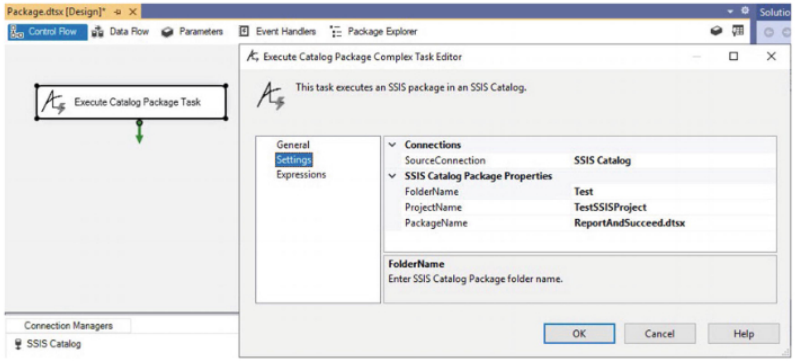
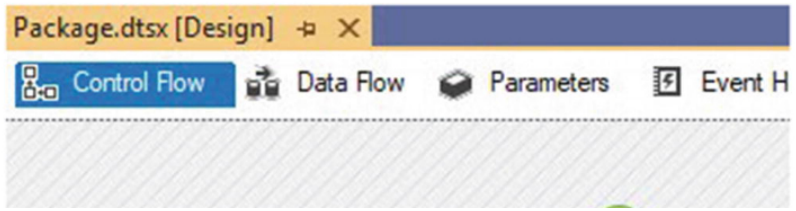


Figure 14-9 The Execute Catalog Package Task configured for use case 1

Execute the test SSIS package in the debugger. If all goes as planned, the test SSIS package should succeed, as shown in Figure **14-10**:



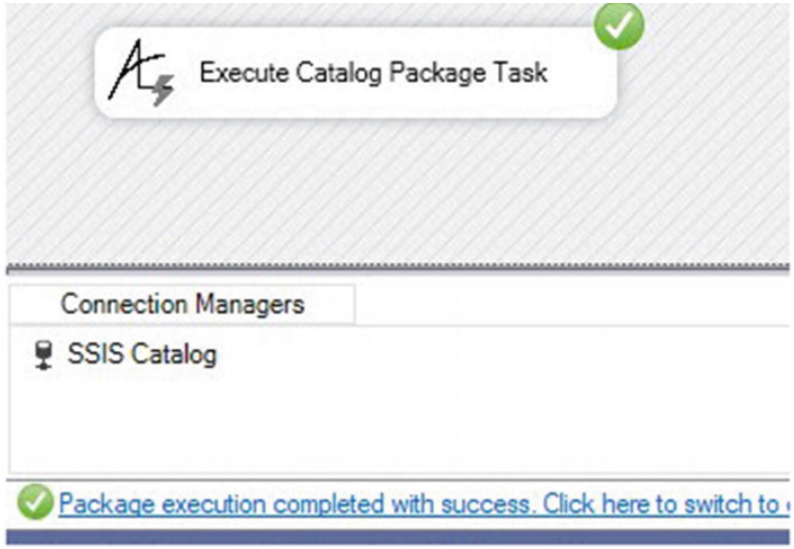


Figure 14-10 Use case 1 success for test SSIS package execution

Check the SSIS Catalog All Execution report to be sure the package executed in the SSIS Catalog, as shown in Figure 14-11:

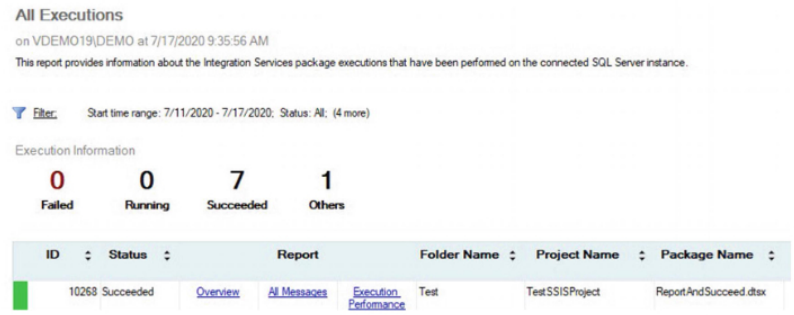
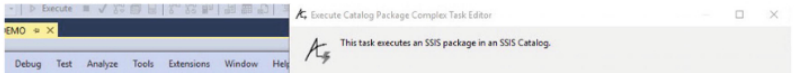


Figure 14-11 Use case 1 success for configured package execution

Success. It's not just for breakfast anymore.

Use Case 2

To test use case 2, add an Execute Catalog Package Task to the control, open the editor, and then select "<new Connection…>" from the SourceConnection property dropdown, as shown in Figure 14-12:



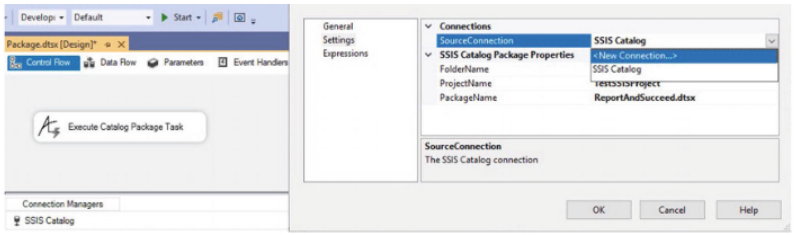


Figure 14-12 The Execute Catalog Package Task configured for use case 2

A new ASO.Net connection manager is created and displays in the Connection Managers pane of the SSIS package, and the "Configure ADO.NET Connection Manager" displays, as shown in Figure 14-13:

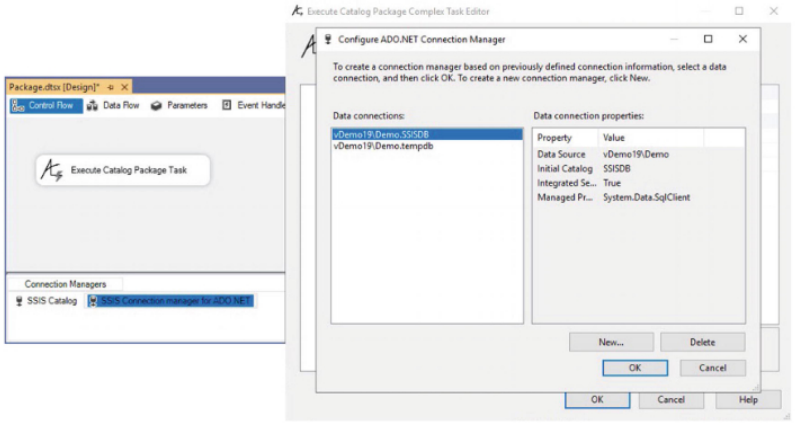
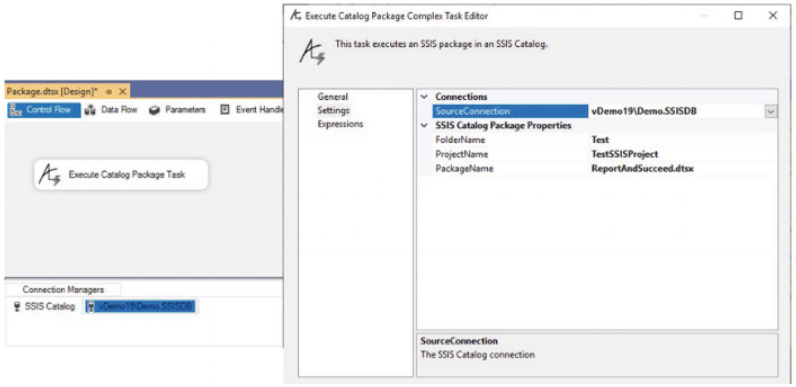


Figure 14-13 Creating a New Connection for use case 2

Configure an ADO.Net connection to the SSISDB database and then click the OK button. When the new connection is selected in the SourceConnection property, the Editor appears as shown in Figure 14-14:



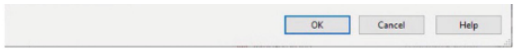


Figure 14-14 New connection configured for use case 2

Click the OK button to close the editor.

Execute the test SSIS package in the debugger. If all goes as planned, the test SSIS package should succeed, as shown in Figure **14-15**:

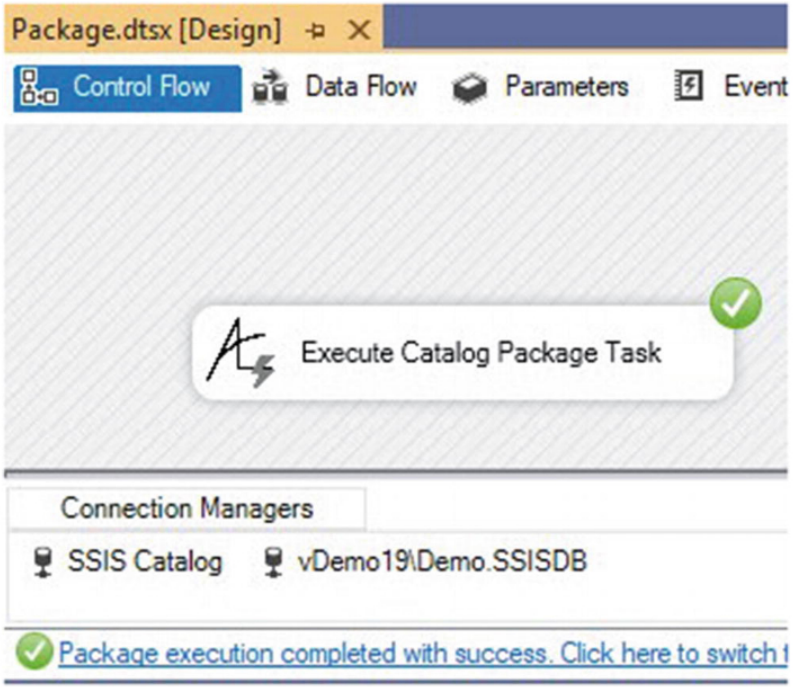


Figure 14-15 Use case 2 success for test SSIS package execution

Check the SSIS Catalog All Execution report to be sure the package executed in the SSIS Catalog, as shown in Figure **14-16**:

Figure 14-16 Use case 2 success for configured package execution

Conclusion

The New Connection option makes building the `SettingsView SourceConnection` property a complex part of building the Execute Catalog Package Task Complex editor, which is why the author isolated this portion of the custom task project in its own chapter.

The next step is to add more properties to the `SettingsView`.

Now would be an excellent time to check in your code.