A. Leonard, *Building Custom Tasks for SQL Server Integration Services*
https://doi.org/10.1007/978-1-4842-6482-9_9

# 9. Signing and Binding

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

We began this book with a rambling introduction that disclosed some of how my brain works. I asked a question: "Do you think it is possible to create a custom SSIS task using Visual Studio Community Edition?"

Next, we configured our development machine and Visual Studio, and then we created a new project. We signed the project so that it would be accepted in the Global Assembly Cache (GAC) and prepared the Visual Studio environment with all the accouterments and references necessary to build a custom SSIS task. We coded the task and its editor. All that brings us here.

In this chapter, we will sign the task editor project and bind the task to the editor. Then we'll code our task's functionality, add an icon, and build and test the task.

## Creating a New Public Key Token Value

The first edition of this book was written in 2017. Consider this chapter a "reboot" of the task editor design. Just to be sure, let's regenerate a key and re-extract the public key.

Before we bind the task editor to the task, let's make a new key.

Open a command prompt as an administrator as shown in Figure 9-1:

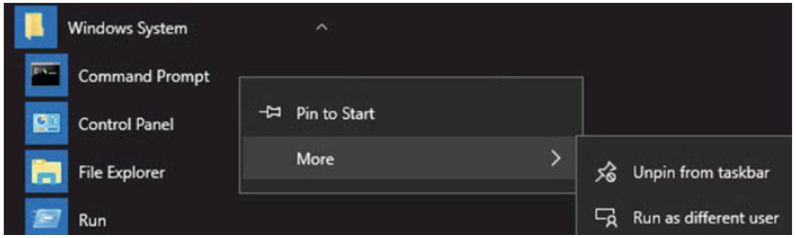



***Figure 9-1*** Opening an Administrator command prompt

In the command prompt window, navigate to the ExecuteCatalogPackageTaskUI folder as shown in Figure 9-2:

```
Administrator: Command Prompt
E:\git\Book Code\ExecuteCatalogPackageTask>cd ExecuteCatalogPackageTaskUI

E:\git\Book Code\ExecuteCatalogPackageTask\ExecuteCatalogPackageTaskUI>dir
 Volume in drive E is vDemo19_E
 Volume Serial Number is 0CEC-CEEC

 Directory of E:\git\Book Code\ExecuteCatalogPackageTask\ExecuteCatalogPackageTaskUI

12/14/2019  12:11 PM    <DIR>          .
12/14/2019  12:11 PM    <DIR>          ..
12/12/2019  05:24 PM    <DIR>          bin
12/14/2019  11:59 AM               956 ExecuteCatalogPackageTaskUI.cs
12/14/2019  12:35 PM             3,916 ExecuteCatalogPackageTaskUI.csproj
12/14/2019  12:02 PM             1,530 ExecuteCatalogPackageTaskUIForm.cs
12/14/2019  10:48 AM             6,388 ExecuteCatalogPackageTaskUIForm.Designer.cs
12/14/2019  10:48 AM             5,817 ExecuteCatalogPackageTaskUIForm.resx
12/14/2019  12:11 PM               596 key.snk
12/12/2019  05:23 PM    <DIR>          obj
12/12/2019  05:23 PM    <DIR>          Properties
               6 File(s)         19,203 bytes
               5 Dir(s)  117,553,102,848 bytes free

E:\git\Book Code\ExecuteCatalogPackageTask\ExecuteCatalogPackageTaskUI>
```

***Figure 9-2*** Navigating to the ExecuteCatalogPackageTaskUI folder

Open the Notes.txt file saved earlier in the ExecuteCatalogPackageTask project, and copy the key retrieval creation and retrieval commands as shown in Listing 9-1 and Figure 9-7:

```
"C:\Program Files (x86)\Microsoft SDKs\Windows\v10.0A\bin\NETFX 4.8 ➥ T
"C:\Program Files (x86)\Microsoft SDKs\Windows\v10.0A\bin\NETFX 4.8 ➥ T
"C:\Program Files (x86)\Microsoft SDKs\Windows\v10.0A\bin\NETFX 4.8 ➥ T
```

***Listing 9-1*** Strong name commands

```
"C:\Program Files (x86)\Microsoft SDKs\Windows\v10.0A\bin\NETFX 4.8 Tools\sn.exe" -k key.snk
"C:\Program Files (x86)\Microsoft SDKs\Windows\v10.0A\bin\NETFX 4.8 Tools\sn.exe" -p key.snk public.out
"C:\Program Files (x86)\Microsoft SDKs\Windows\v10.0A\bin\NETFX 4.8 Tools\sn.exe" -t public.out
```



***Figure 9-3*** Copying the public key creation and retrieval commands

Paste the public key creation and retrieval commands into the Administrator command prompt window as shown in Figure 9-4:

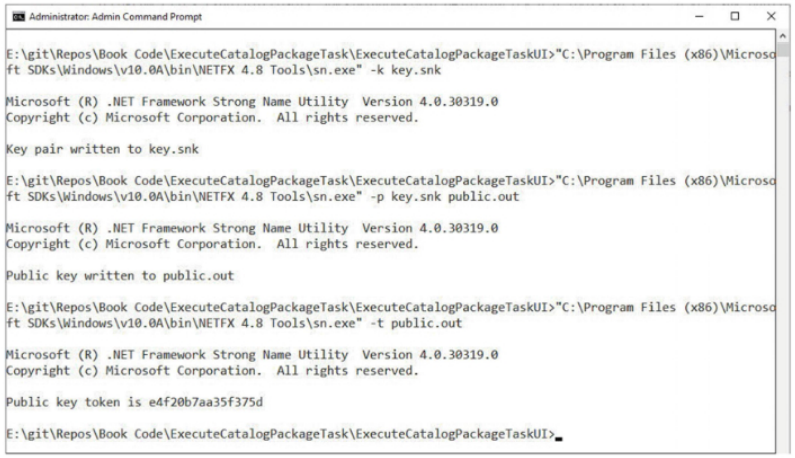



***Figure 9-4*** Creating and retrieving the public key token

Highlight the public key token value as shown in Figure 9-5:

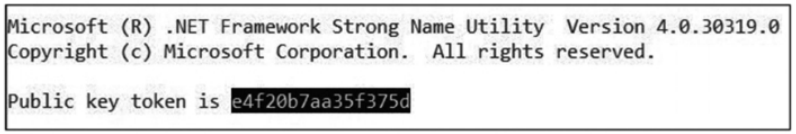



***Figure 9-5*** Highlighting the public key token value

Right-click the selection to copy it to the clipboard. Paste the clipboard contents in the ExecuteCatalogPackageTask.cs file near the original Public key value for comparison as shown in Figure 9-6:

```
public class ExecuteCatalogPackageTask : Microsoft.SqlServer.Dts.Runtime.Task
{
    // Public key: e86e33313a45419e
    //             e4f20b7aa35f375d -- new key for simple UI
```

***Figure 9-6*** Comparing the original and new public key values

| **Note** A new public key is required for signing the UI assembly. |
|---|

## Signing the Task Editor Project

We haven't yet signed the Task Editor project. Let's do that now.

In Solution Explorer, double-click Properties under the ExecuteCatalogPackageTaskUI project to open the project properties.

Click on the Signing page, check the "Sign the assembly" checkbox, click the "Choose a strong key name file" dropdown, click Browse, browse to key.snk in the ExecuteCatalogPackageTaskUI project folder – the key.snk file we just created – and select that file as shown in Figure 9-7:

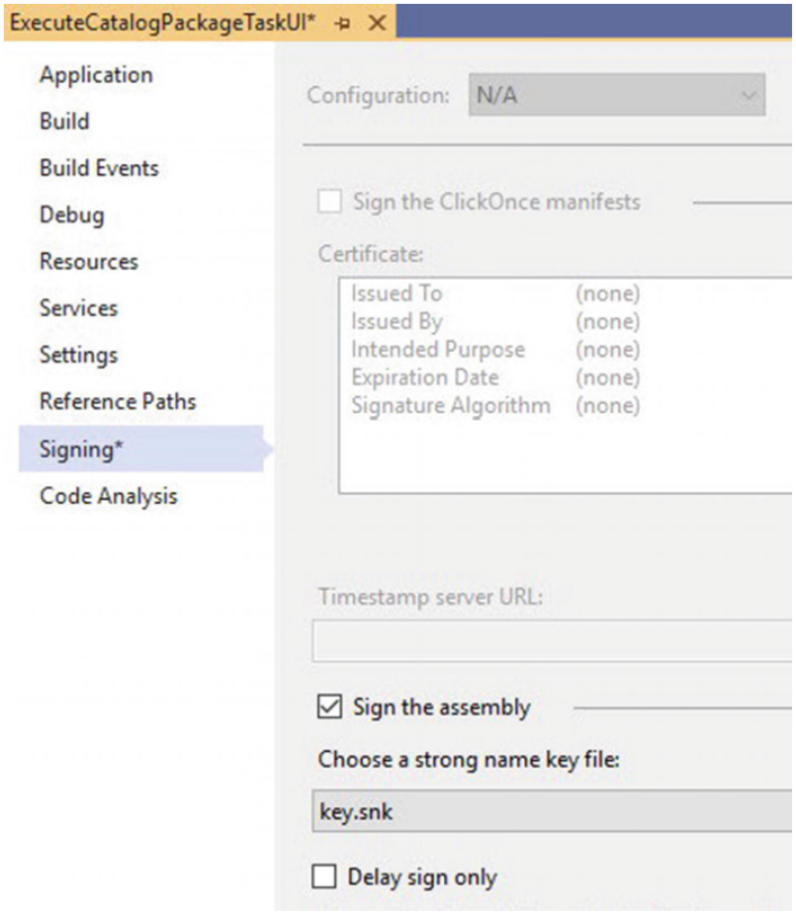
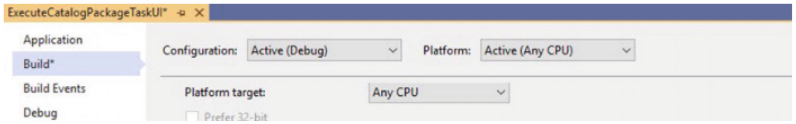



***Figure 9-7*** Signing the UI project

As with the Task project properties, click on the Build page and set the Build output path to *<drive>*:\Program Files (x86)\Microsoft SQL Server*<version>*\DTS\Tasks where *<drive>* represents the installation drive for SQL Server and version represents the numeric version value of the SQL Server for which you are building this task, as seen in Figure 9-8:



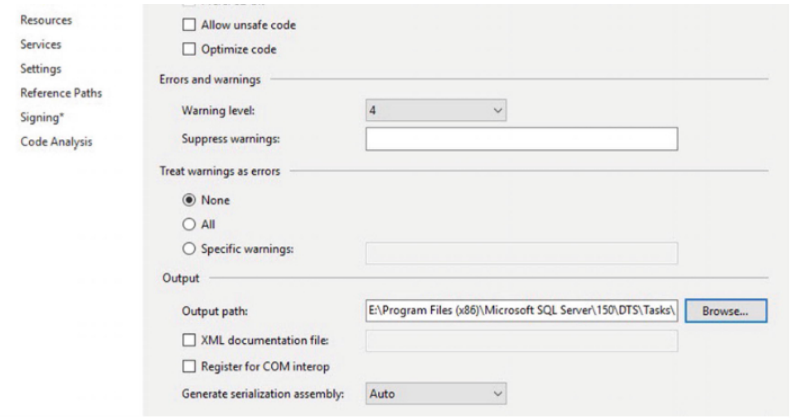



***Figure 9-8*** Setting the Build output path for the ExecuteCatalogPackageTaskUI project

As with the Task project, we may automate the gacutil unregister and register functions. Click the Build Events page, click the Edit Pre-build... button, and then add the code shown in Listing 9-2 and Figure 9-9:

```
"C:\Program Files (x86)\Microsoft SDKs\Windows\v10.0A\bin\NETFX 4.8 ➥ T
```

***Listing 9-2*** Gacutil unregister command

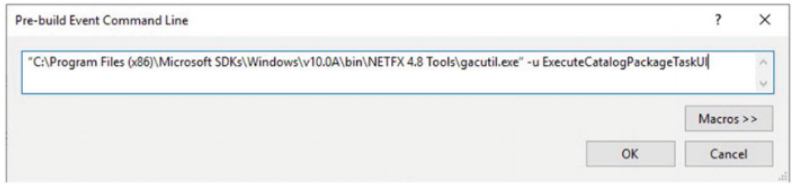



***Figure 9-9*** Adding the gacutil unregister command to the pre-build build event

Click the Edit Post-build... button and add the code shown in Listing 9-3 and Figure 9-10:

```
"C:\Program Files (x86)\Microsoft SDKs\Windows\v10.0A\bin\NETFX 4.8 ➥ T
```

***Listing 9-3*** Gacutil register command

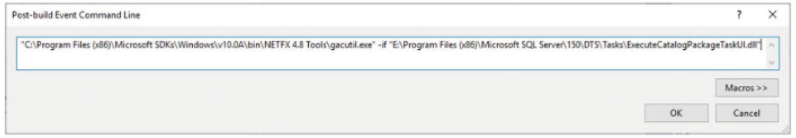



***Figure 9-10*** Adding the gacutil register command to the post-build build event

*Figure 9-10* Adding the gacutil register command to the post-build build event

Once complete, save and close the project properties.

## Binding the Task Editor to the Task

We next need to inform the task that it has an editor. Open the ExecuteCatalogPackageTask solution and the ExecuteCatalogPackageTask project in Visual Studio. Open the ExecuteCatalogPackageTask class as shown in Figure 9-11:

```
namespace ExecuteCatalogPackageTask
{
    [DtsTask(
            TaskType = "DTS150"
          , DisplayName = "ExecuteCatalogPackageTask"
          , Description = "A task to execute packages stored in the SSIS Catalog."
            )]
```

***Figure 9-11*** ExecuteCatalogPackageTask, before changes

The DtsTask attribute (decoration) is shown and documented **here**. In its current state, our task decoration has three values defined: TaskType, DisplayName, and Description. To couple the editor (UI) to the task, we add the multipart attribute UITypeName by adding the code in Listing 9-4 to the existing decoration:

```
, UITypeName= "ExecuteCatalogPackageTaskUI.ExecuteCatalogPackageTaskUI ➥
",ExecuteCatalogPackageTaskUI,Version=1.0.0.0,Culture=Neutral ➥
,PublicKeyToken=<Your public key>"
, TaskContact = "ExecuteCatalogPackageTask; Building Custom Tasks for ➥
 SQL Server Integration Services, 2019 Edition; © 2020 Andy Leonard; ➥
```

***Listing 9-4*** Updating the DtsTask decoration

Once added to the DtsTask attribute decoration, the decoration will appear similar to Figure 9-12:

***Figure 9-12*** Adding the UITypeName and TaskContact attributes to the DtsTask decoration

You can find (some) documentation for the UITypeName attribute **at docs.microsoft.com/en-us/dotnet/api/microsoft.sqlserver.dts.runtime.dtstaskattribute.uitypename?view=sqlserver-2017**. The property/value pairs are

- Type Name: ExecuteCatalogPackageTaskUI.ExecuteCatalogPackageTaskUI
- Assembly Name: ExecuteCatalogPackageTaskUI
- Version: 1.0.0.0

- Culture: Neutral
- Public Key: <*Your public key*>

## Coding the Task Functionality

Our task is *almost* ready to build and test. What's left? SSIS Catalog Package execution functionality. Look in the Execute method shown in Figure 9-13:

```
0 references
public override DTSExecResult Execute(
    Connections connections,
    VariableDispenser variableDispenser,
    IDTSComponentEvents componentEvents,
    IDTSLogging log,
    object transaction)
{
    return DTSExecResult.Success;
}
```

***Figure 9-13*** The Execute method

We are going to add SSIS Catalog Package execution functionality here. There are several good articles available that walk one through executing SSIS Catalog Packages via .Net. A good summary of the process of executing SSIS 2019 package programmatically is found at docs.microsoft.com/en-us/sql/integration-services/run-manage-packages-programmatically/running-and-managing-packages-programmatically?view=sql-server-ver15.

> **Caution** Before we proceed, I want to remind you that we are not building a production-ready Execute Catalog Package Task. We are building a minimum amount of functionality to demonstrate the steps required to code a custom SSIS task. A production-ready custom SSIS task would include more functionality that we will cover here.

We first need more .Net Framework References. Add the following references to the ExecuteCatalogPackageTask project:

- Microsoft.SqlServer.ConnectionInfo
- Microsoft.SqlServer.Management.IntegrationServices
- Microsoft.SqlServer.Management.Sdk.Sfc
- Microsoft.SqlServer.Smo

I found Microsoft.SqlServer.ConnectionInfo in the C:\Windows\Microsoft.NET\assembly\GAC_MSIL\Microsoft.SqlServer.ConnectionInfo\v4.0_15.0.0.0__89845dcd8080cc91\ folder on my development VM, as shown in Figure 9-14:

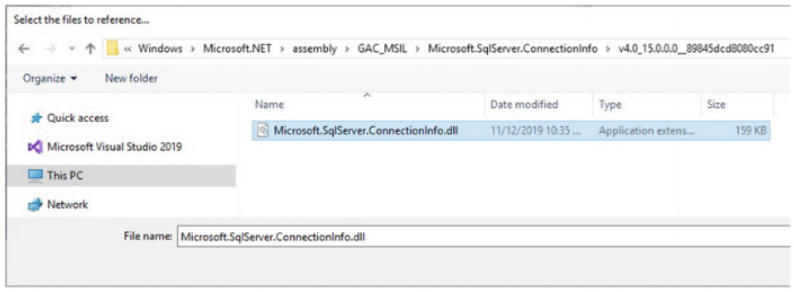



***Figure 9-14*** Adding a reference for Microsoft.SqlServer.ConnectionInfo

I found Microsoft.SqlServer.Management.IntegrationServices, Microsoft.SqlServer.Management.Sdk.Sfc, and Microsoft.SqlServer.Smo in the C:\Windows\Microsoft.NET\assembly\GAC_MSIL\ path, as well, as shown in Figure 9-15:

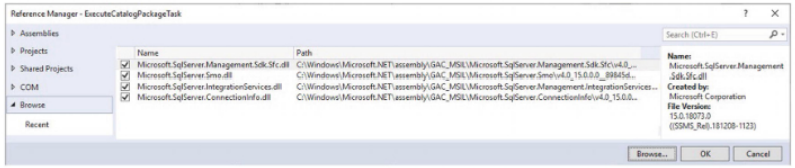
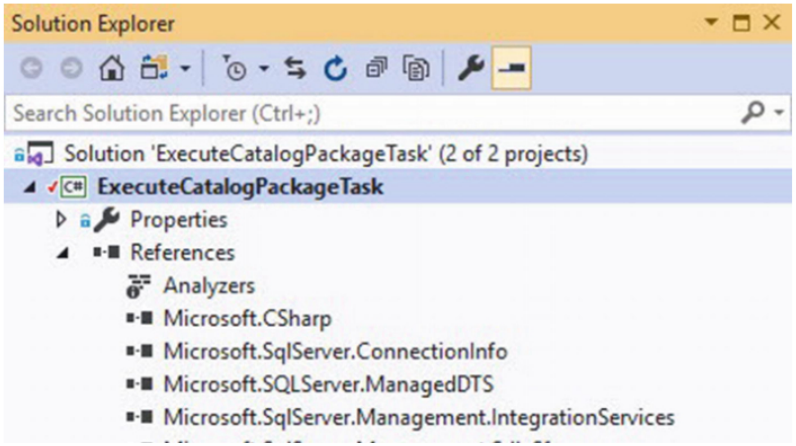



***Figure 9-15*** Browsing to reference additional assemblies

The Solution Explorer References virtual folder for the ExecuteCatalogPackageTask project should now appear as shown in Figure 9-16:



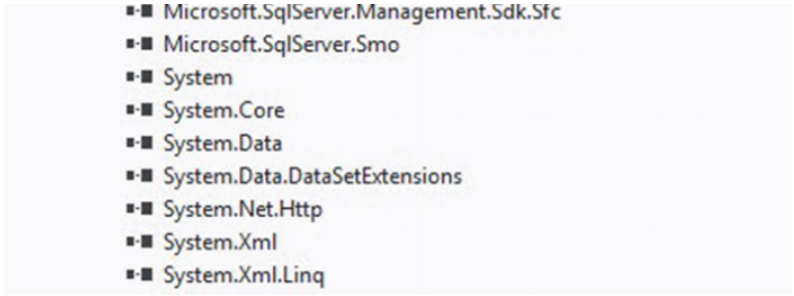



***Figure 9-16*** Viewing the ExecuteCatalogPackageTask project References

We next need to declare reference assemblies in ExecuteCatalogPackageTask.cs for use in our project, as shown in Listing 9-5:

```
using Microsoft.SqlServer.Management.IntegrationServices;
using Microsoft.SqlServer.Management.Smo;
```

***Listing 9-5*** Using referenced assemblies

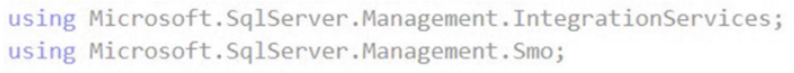



***Figure 9-17*** Importing Referenced Assemblies

We're now ready to add functionality to the `Execute` function. Declare and initialize some variables using the code shown in Listing 9-6:

```
Server catalogServer = new Server(ServerName);
IntegrationServices integrationServices = new ➥ IntegrationServices(cat
Catalog catalog = integrationServices.Catalogs[PackageCatalog];
CatalogFolder catalogFolder = catalog.Folders[PackageFolder];
ProjectInfo catalogProject = catalogFolder.Projects[PackageProject];
Microsoft.SqlServer.Management.IntegrationServices.PackageInfo ➥ catalo
```

***Listing 9-6*** Coding the Execute method, part 1

Your Execute method should appear similar to that shown in Figure 9-18:

```
    return DTSExecResult.Success;
}
```

***Figure 9-18*** Execute SSIS Catalog Package method partially coded

Finally, add the call to execute the SSIS Package object (catalogPackage) as shown in Listing 9-7 and Figure 9-19:

```
catalogPackage.Execute(False, Nothing)
```

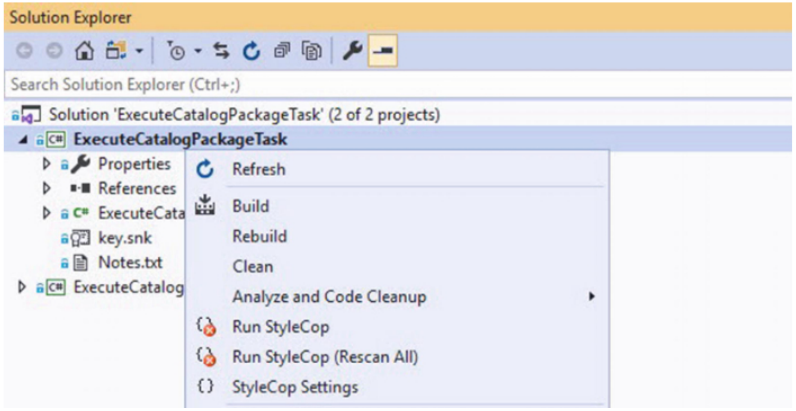
***Listing 9-7*** Adding package execution code


***Figure 9-19*** Calling the CatalogPackage.Execute Method

I don't want to make too big a deal over this, but we did *all this work* to add that one line of code…

## Add an Icon

Before you can use an icon, you must import it into your project. Right-click the ExecuteCatalogPackageTask project in Solution Explorer, hover over Add, and then click Existing Item as shown in Figure 9-20:



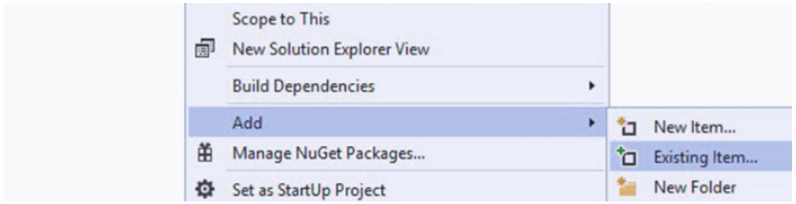



***Figure 9-20*** Adding an Existing Item to the ExecuteCatalogPackageTask Project

Navigate to the icon file you wish to use as shown in Figure 9-21:

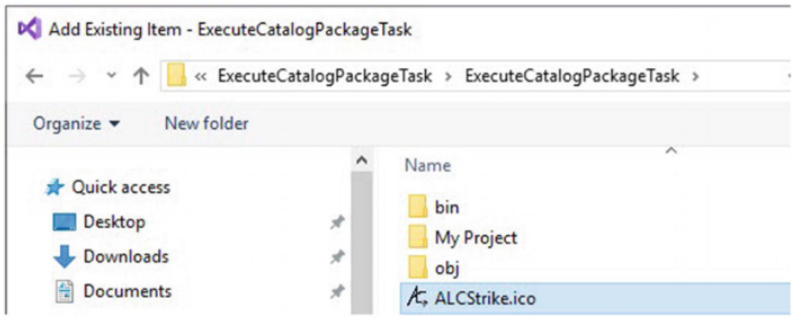



***Figure 9-21*** Selecting the icon

The icon file will appear in Solution Explorer as shown in Figure 9-22:

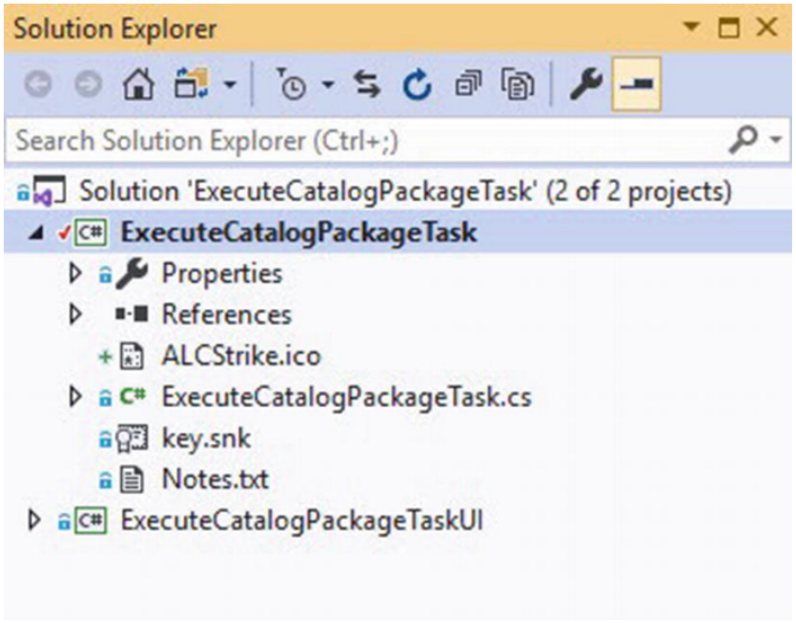


*Figure 9-22* Viewing the icon file

With the icon file selected in Solution Explorer, press the F4 key to display the Properties. Change the Build Action property of the icon file from Content to Embedded Resource as shown in Figure 9-23:

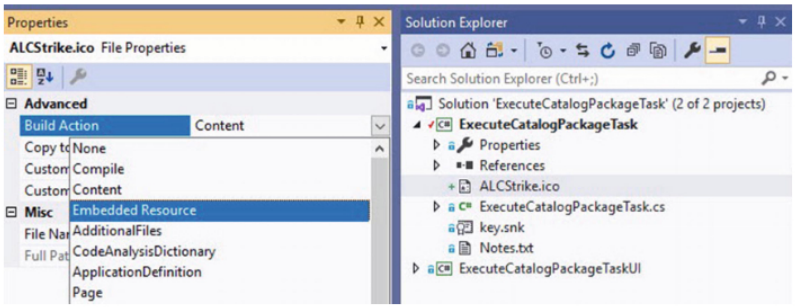



*Figure 9-23* Changing the Build Action property of the icon file

Let's add the icon to the ExecuteCatalogPackageTaskUI form. Open the form and view the Properties. Click the ellipsis beside the Icon property to open the icon selection dialog as shown in Figure 9-24:

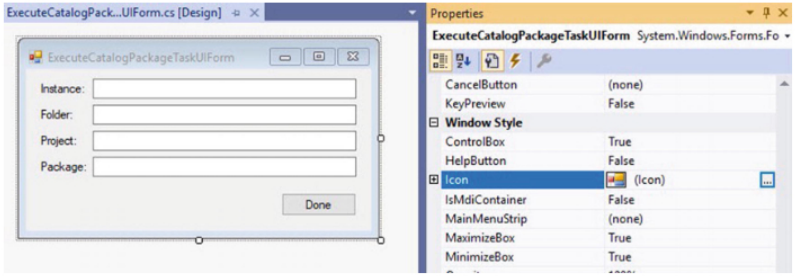



*Figure 9-24* Opening the form icon selection dialog

Select the icon as shown in Figure 9-25:

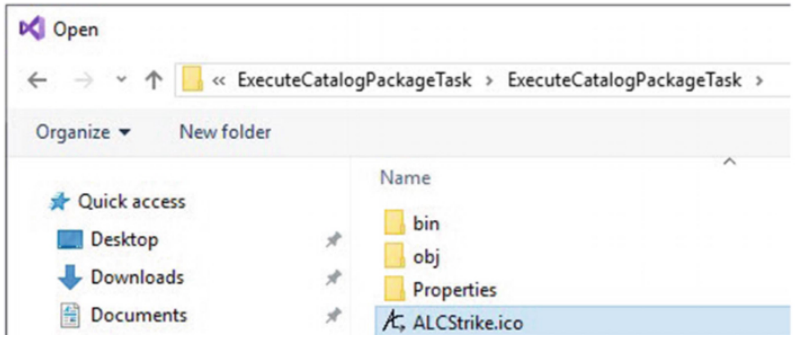



*Figure 9-25* Selecting the form icon

The selected icon now appears as the icon for the ExecuteCatalogPackageTaskForm form as shown in Figure 9-26:

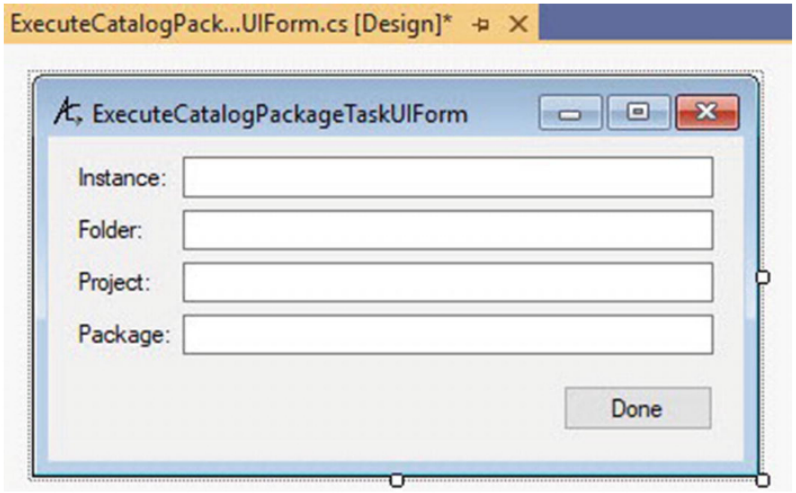



*Figure 9-26* Viewing the form icon

While we're here, let's update the Text property of the form to "Execute Catalog Package Task Editor" as shown in Figure 9-27:

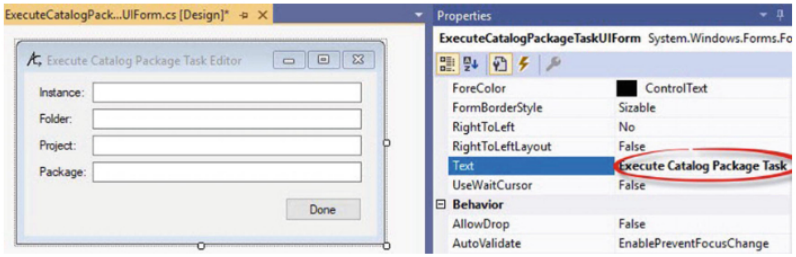



*Figure 9-27* Updating the form's text property

SSIS won't know which icon to display until we update the DtsTask decoration for ExecuteCatalogPackageTask. Open ExecuteCatalogPackageTask.cs and add the line shown in Listing 9-8 to the decoration:

```
, IconResource = "ExecuteCatalogPackageTask.ALCStrike.ico"
```

*Listing 9-8.* Adding the icon to the DtsTask decoration

*Listing 9-8* Adding the icon to the DtsTask decoration

Your decoration should now appear similar to that shown in Figure 9-28:

```
[DtsTask(
        TaskType = "DTS150"
      , DisplayName = "ExecuteCatalogPackageTask"
      , IconResource = "ExecuteCatalogPackageTask.ALCStrike.ico"
      , Description = "A task to execute packages stored in the SSIS Catalog."
      , UITypeName= "ExecuteCatalogPackageTask.ExecuteCatalogPackageTaskUI,ExecuteCatalogP
      , TaskContact = "ExecuteCatalogPackageTask; Building Custom Tasks for SQL Server Int
0 references
public class ExecuteCatalogPackageTask : Microsoft.SqlServer.Dts.Runtime.Task
```

***Figure 9-28*** Viewing updated DtsTask decoration

Now would be an excellent time to check your code into source control.

## Building the Task

We are now code-complete! It is time to build our solution, which will compile the code into an executable. From the Build dropdown menu, click Build Solution as shown in Figure 9-29:

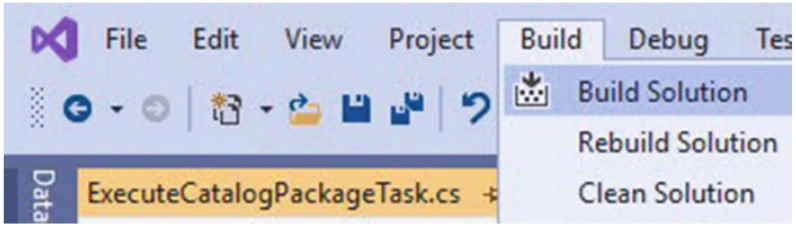



***Figure 9-29*** Building the solution

If all goes well, you should see verbiage in the Output window similar to that shown in Figure 9-30:

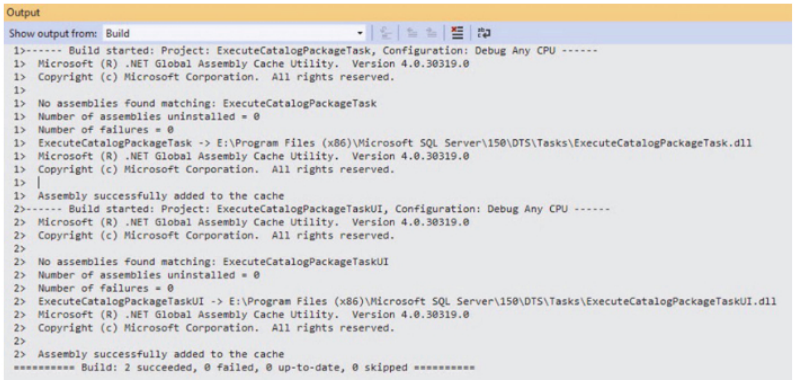


*Figure 9-30* Build output

If all goes as expected, you should have two assemblies successfully built to the DTS\Tasks\ folder as shown in Figure 9-31:

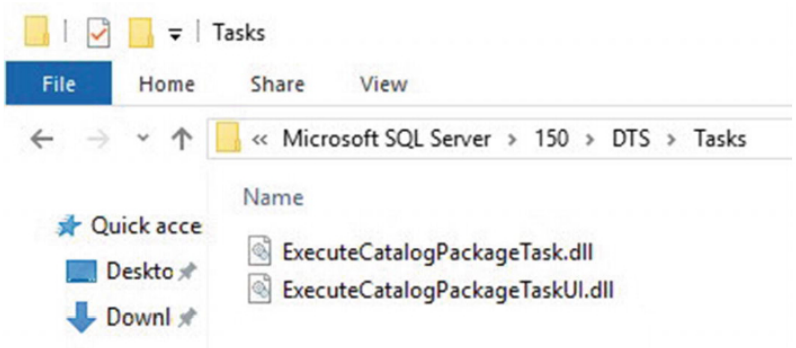



*Figure 9-31* ExecuteCatalogPackageTask and ExecuteCatalogPackageTaskUI assemblies in the DTS\Tasks\ folder

You should also find the assemblies in the GAC as shown in Figure 9-32:

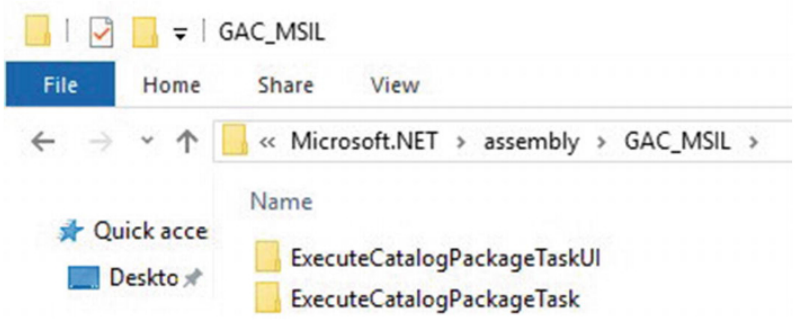



*Figure 9-32* ExecuteCatalogPackageTask and ExecuteCatalogPackageTaskUI assemblies in the GAC

## Testing the Task

The moment of truth has arrived. Will the task work? Will it even show up in the SSIS Toolbox? Let's open SQL Server Data Tools (SSDT) and find out! If all has gone according to plan, you will be able to open a test SSIS project and see the following on the SSIS Toolbox as shown in Figure 9-33:

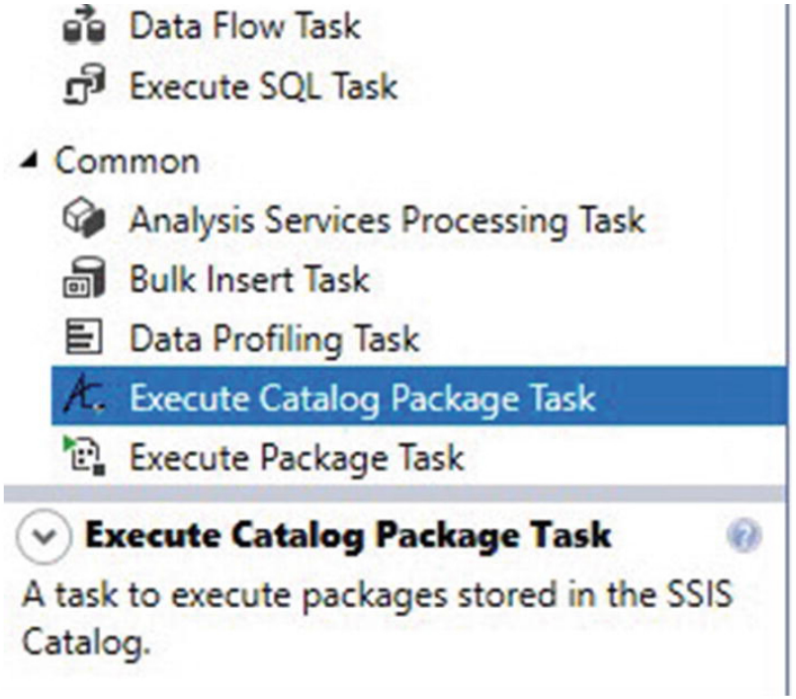
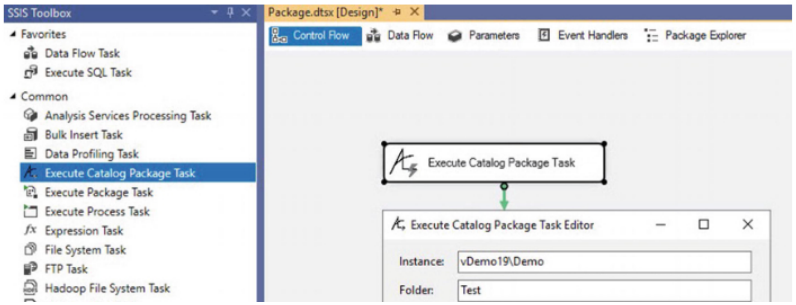



***Figure 9-33*** Execute Catalog Package Task in the SSIS Toolbox

Drag it onto the surface of the Control Flow as shown in Figure 9-34:


***Figure 9-34*** Execute Catalog Package Task on the SSIS control flow

Double-click the task to open the editor and configure the task as shown in Figure 9-35:



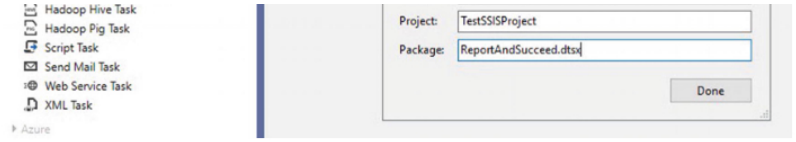



***Figure 9-35*** Editing the task

Click the Done button and execute the package in the debugger. If all goes well, our Execute Catalog Package Task will succeed as shown in Figure 9-36:

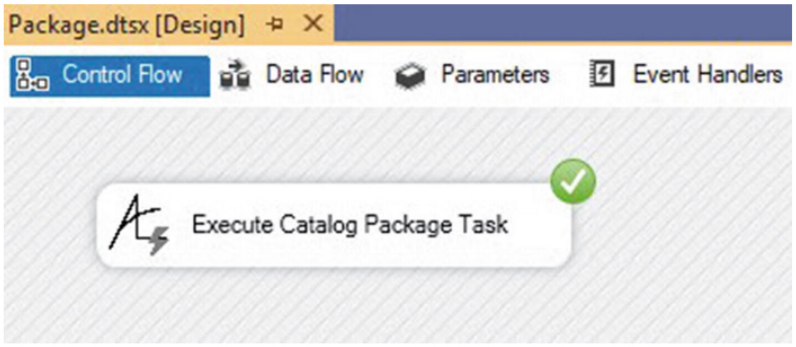



***Figure 9-36*** Task execution

Did the task *really* succeed or did it just report success? We can check by examining the SSIS Catalog Reports which are built into SQL Server Management Studio (SSMS). Open SSMS and connect to the instance that hosts your SSIS Catalog. Expand the Integration Services Catalogs node, right-click on SSISDB, hover over Reports, hover over Standard Reports, and then click All Executions as shown in Figure 9-37:

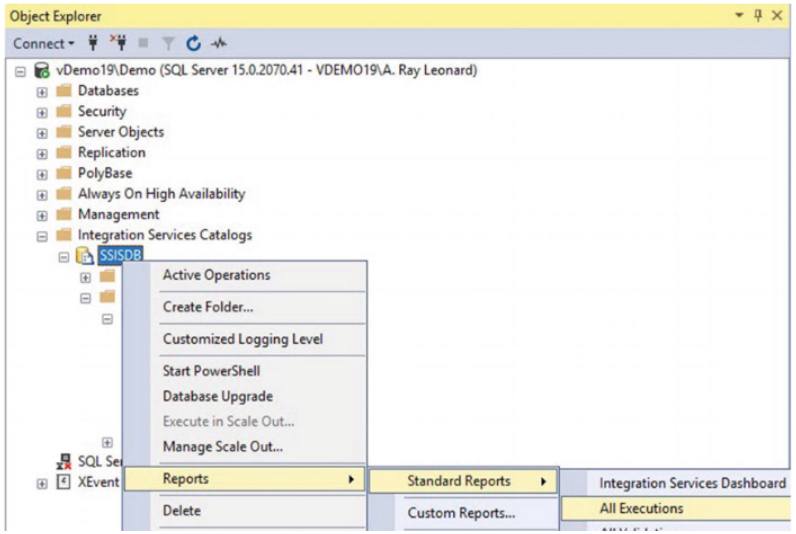



***Figure 9-37*** Opening the SSIS Catalog All Executions Report in SSMS

The All Executions Report displays SSIS Package executions in the past 7 days (by default). I confess. I executed it twice as shown in Figure 9-38:

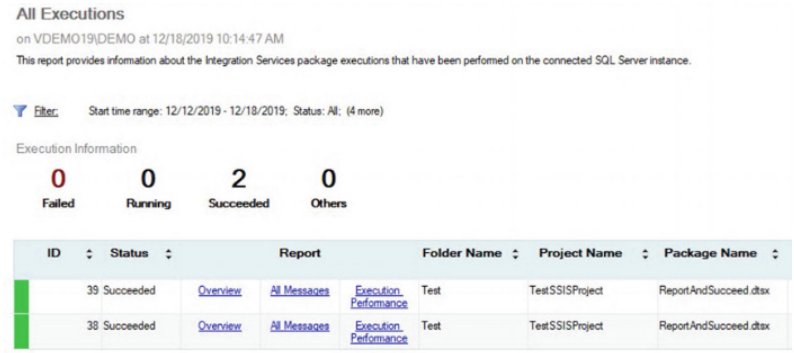



***Figure 9-38*** Viewing SSIS Catalog All Executions Report

Excellent!

## Conclusion

The code and editor now function as a unit.

Now would be a good time to execute a commit and push to Azure DevOps.

At this point in development, we have

- Created and configured an Azure DevOps project
- Connected Visual Studio to the Azure DevOps project
- Cloned the Azure DevOps Git repo locally
- Added a reference to the Visual Studio project
- Performed an initial check-in of the project code
- Signed the assembly
- Checked in an update
- Configured the build output path and build events
- Overridden three methods from the Task base class
- Added and coded most of a project for the Task editor.
- Knit the Task Editor to the Task.

But what if things *don't* happen like this? What if there's an error?

We next extend editor functionality.

Previous chapter
< 8. Coding a Simple Task Editor

Next chapter
10. Expanding Editor Function... >