

Building Custom Tasks for SQL Server Integration Services

Taking a low-cost approach to
applying the power of .NET in
ETL solutions

Andy Leonard

Apress®

Building Custom Tasks for SQL Server Integration Services



Andy Leonard

Apress®

Building Custom Tasks for SQL Server Integration Services

Andy Leonard
Farmville, Virginia, USA

ISBN-13 (pbk): 978-1-4842-2939-2
DOI 10.1007/978-1-4842-2940-8

ISBN-13 (electronic): 978-1-4842-2940-8

Library of Congress Control Number: 2017947306

Copyright © 2017 by Andy Leonard

This work is subject to copyright. All rights are reserved by the Publisher, whether the whole or part of the material is concerned, specifically the rights of translation, reprinting, reuse of illustrations, recitation, broadcasting, reproduction on microfilms or in any other physical way, and transmission or information storage and retrieval, electronic adaptation, computer software, or by similar or dissimilar methodology now known or hereafter developed.

Trademarked names, logos, and images may appear in this book. Rather than use a trademark symbol with every occurrence of a trademarked name, logo, or image we use the names, logos, and images only in an editorial fashion and to the benefit of the trademark owner, with no intention of infringement of the trademark.

The use in this publication of trade names, trademarks, service marks, and similar terms, even if they are not identified as such, is not to be taken as an expression of opinion as to whether or not they are subject to proprietary rights.

While the advice and information in this book are believed to be true and accurate at the date of publication, neither the authors nor the editors nor the publisher can accept any legal responsibility for any errors or omissions that may be made. The publisher makes no warranty, express or implied, with respect to the material contained herein.

Managing Director: Welmoed Spahr
Editorial Director: Todd Green
Acquisitions Editor: Jonathan Gennick
Development Editor: Jonathan Gennick
Technical Reviewer: Luther Atkinson
Coordinating Editor: Jill Balzano
Copy Editor: Lori Jacobs

Distributed to the book trade worldwide by Springer Science+Business Media New York, 233 Spring Street, 6th Floor, New York, NY 10013. Phone 1-800-SPRINGER, fax (201) 348-4505, e-mail orders-ny@springer-sbm.com, or visit www.springeronline.com. Apress Media, LLC is a California LLC and the sole member (owner) is Springer Science + Business Media Finance Inc (SSBM Finance Inc). SSBM Finance Inc is a **Delaware** corporation.

For information on translations, please e-mail rights@apress.com, or visit <http://www.apress.com/rights-permissions>.

Apress titles may be purchased in bulk for academic, corporate, or promotional use. eBook versions and licenses are also available for most titles. For more information, reference our Print and eBook Bulk Sales web page at <http://www.apress.com/bulk-sales>.

Any source code or other supplementary material referenced by the author in this book is available to readers on GitHub via the book's product page, located at www.apress.com/9781484229392. For more detailed information, please visit <http://www.apress.com/source-code>.

Printed on acid-free paper

For Christy

Contents at a Glance

About the Author	xi
About the Technical Reviewer	xiii
Acknowledgments	xv
Introduction	xvii
■ Chapter 1: Story of This Book.....	1
■ Chapter 2: Creating the Assembly	3
■ Chapter 3: Signing the Assembly	11
■ Chapter 4: Preparing the Environment	25
■ Chapter 5: Coding the Task.....	35
■ Chapter 6: Coding the Task Editor	47
■ Chapter 7: Signing and Binding	65
■ Chapter 8: Tips on Troubleshooting	87
■ Chapter 9: Notes from Experience.....	99
■ Chapter 10: Demonstration Code.....	107
Index.....	109

Contents

About the Author	xi
About the Technical Reviewer	xiii
Acknowledgments	xv
Introduction	xvii
■ Chapter 1: Story of This Book.....	1
Tribal Knowledge.....	1
A Starting Point	2
What Problem Are We Trying to Solve?	2
■ Chapter 2: Creating the Assembly	3
Opening Visual Studio IDE	3
Adding a Reference	6
■ Chapter 3: Signing the Assembly	11
Preparing to Add a Key	11
Creating the Key	16
Applying the Key	21
■ Chapter 4: Preparing the Environment	25
Adding the Solution to Source Control	25
Preparing to Build.....	32
Setting the Output Path	33

Chapter 5: Coding the Task.....	35
Using a Reference	35
Decorating the Class	36
Inheriting from Task	36
Adding a Property.....	37
Investigating Task Methods.....	41
Overriding the Validate Method	44
Overriding the Execute Method	45
Chapter 6: Coding the Task Editor	47
Adding a Task Editor.....	47
Adding References	49
Some Implementation	50
Adding a Form.....	55
Coding the Click Event and the Form	59
Chapter 7: Signing and Binding.....	65
Signing the Task Editor Project	65
Reviewing the Public Key Token Value	67
Binding the Task Editor to the Task	70
Coding the Task Functionality	71
Add an Icon	75
Building the Task.....	80
Testing the Task.....	82
Chapter 8: Tips on Troubleshooting	87
Unregistering the Task.....	87
Troubleshoot the Issue	89

■ Chapter 9: Notes from Experience.....	99
Start Visual Studio as an Administrator.....	99
Learn How to Recover	100
Building a Notes File.....	100
Cleaning the Solution.....	101
Change the Icon File's Build Action Property.....	105
■ Chapter 10: Demonstration Code.....	107
Index.....	109

About the Author



Andy Leonard is a Data Philosopher at Enterprise Data & Analytics, an SSIS Trainer, Consultant, developer of the Data Integration Lifecycle Management (DILM) Suite, a Business Intelligence Markup Language (Biml) developer, and BimlHero. He is also a SQL Server database and data warehouse developer, community mentor, engineer, and farmer. Andy is coauthor of *SQL Server Integration Services Design Patterns* and of *Stairway to Integration Services*.

About the Technical Reviewer

An avid dabbler and hobbyist in all things computer and electronics, **Luther Atkinson** has been an applications developer since before the time of MS-DOS and Windows. Introduced to SQL Server version 7 and DTS in the late 1990s, he's been a fan and user of the product ever since and is now employed as an SSIS (SQL Server Integration Services) developer in Virginia. Luther has a master's degree in computer science from the University of Florida, and a master's in decision science from Virginia Commonwealth University's School of Business.

Acknowledgments

This small book would not have been possible without the help of others. Kirk Haselden, author of *Microsoft SQL Server 2005 Integration Services* (Sams, 2006, www.amazon.com/Microsoft-Server-2005-Integration-Services/dp/0672327813), wrote extensively in this book about developing custom tasks for SQL Server Integration Services (SSIS). Matt Masson's blog includes several posts tagged "extensions" which cover the topic of custom SSIS task development.

Thanks to Brian Moran, my cofounding partner at Linchpin People, for his support and suggestions.

Thanks Luther Atkinson for proofreading and providing feedback on the manuscript, and for your encouragement.

I owe my coworkers at Enterprise Data & Analytics (entdna.com) a debt of gratitude for their encouragement and for covering for me when I stayed up too late trying to figure out how to make this revised code work. We have an awesome team, especially my brothers from other mothers—Nick Harris and Kent Bradshaw.

Donald Farmer inspires me every time we interact. As Principal Program Manager at Microsoft, Donald worked extensively with SSIS and helped shape the product. Donald continues to shape software by providing vendors unique strategic guidance at TreeHive Strategy (treehivestrategy.com).

I am certain there are many excellent editors in this business. Jonathan Gennick is the best with whom I have had the privilege to work.

Finally, I thank my family for their understanding. My children Stevie Ray, Emma, and Riley who live at home at the time of this writing, and Manda and Penny who have children of their own. And Christy, to whom this book is dedicated: my wife, my love. Thank you.

Introduction

This small book was originally written as a series of blog posts hosted on the Linchpin People web site in 2012.

I have delivered SQL Server Integration Services (SSIS) solutions for over ten years at the time of this writing. Before SSIS, I worked with Data Transformation Services (DTS) for a couple years. Although SSIS has supported custom tasks and components since its 2005 release, I initially recommended that my consulting clients not develop custom SSIS tasks and components. Why?

The source code needs to be maintained and supported throughout the life cycle of SSIS solutions that use the custom tasks and components.

One needs to invest in Visual Studio Professional (roughly \$1,000) to build the code.

One needs fairly serious .Net developer skills to understand the intricacies of building a Visual Studio toolbox component.

My first concern stands but with a much lower barrier to entry—at least cost-wise.

When Microsoft® released free versions of Visual Studio (Visual Studio Express and, more recently, Visual Studio Community Edition), my second concern lapsed once I learned I could use the free tools to build a custom SSIS task (the topic of this book).

This book addresses my third concern.

Writing the blog series and revisiting the series to produce this book was fun for me. I pray you enjoy reading it as much as I enjoyed writing it.

—Andy
Farmville Virginia
April 2017

CHAPTER 1



Story of This Book

Occasionally my engineering curiosity gets the better of me and I ask myself questions like the following: "Self, do I think such-and-such is possible?" Most of the time these questions lead to some brain exercise and little else of use. Sometimes some good comes from the effort.

One day in late summer 2012, I asked myself, "Do you think it is possible to create a custom SSIS task using Visual Basic Express?" Crazy question, right?

The short answer is: Yes.

Why is this significant? Because the [Visual Studio IDE](#) (Integrated Development Environment) is [free](#). This means that if you have a license for SQL Server, you do not need to spend more money to extend SSIS with custom tasks and components. You may *want* to spend money to obtain some of the tools and features of other versions of Visual Studio, though.

I revised the blog series to produce this book. Experienced developers will notice I used Visual Studio 2015 Community Edition and targeted the SQL Server 2016 Integration Services (SSIS 2016) versions of .Net Framework.

Tribal Knowledge

I ran into some issues along the way. Initially, I didn't know if the issues were related to the free Express Edition. It is difficult to work through issues when you do not know something is possible. It took time and determination, but I worked through enough issues to write a series of blog posts, which became the inspiration for this book.

In 2012 I was admittedly an out-of-practice software developer. Before the .Net era I was a Microsoft Certified Solutions Developer (MCSD). For you young whipper-snappers, I achieved that certification back in the good ol' days when the years began with "1" and we had to carve our own chips out of wood. #GetOffMyLawn ☺

Some things I encountered are dreadfully obvious to experienced software developers. After years of developing software, those experienced developers possess [tribal knowledge](#)—they don't remember all the stuff they know. So when a relative .Net n00b (like me) asks a question, they assume I've done all the things they would have done by the intellectual equivalent of rote muscle memory.

I had not done all those things (did I mention I was a .Net n00b?).

One result was that the things that tripped me up were relatively minor. It was all well and good once I knew the tricks, but the tricks are considered so basic that people rarely share them. It is assumed that if you build assemblies aimed for the Global Assembly Cache (GAC), you must know the essentials.

If you are not a .Net n00b, you probably know these essentials. But that did not apply to me at the outset.

Now, I am no slouch when it comes to coding. But I was also no longer a professional software developer in 2012. Before .Net arrived on the scene, I developed software. I already mentioned the MCSD certification. I learned BASIC in 1975 and began writing Visual Basic (VB) when the second version was released. Many VB programmers never made the leap to object-oriented programming (OOP) first supported by VB 4 (if memory serves), but I did. Modeling objects via classes made sense to me, and it has served me well in the years since. It even led me to the concept of design patterns which have some application in [data integration development with SSIS](#).

I've since made the leap to C#, developing the tools and utilities found in the [DILM Suite](#) using C#.

A Starting Point

My goal in this book is to produce a template that anyone can use as a starting point for creating a custom SSIS task using [Visual Studio IDE](#). In the chapters that follow I will demonstrate one way to achieve this.

For development, I am using a virtual machine running Windows Server 2016. The VM is configured with four central processing units (CPUs) (although you do not need four CPUs) and 8GB RAM (although you can most likely get away with 2GB RAM, I recommend a minimum of 4GB RAM). SQL Server 2016 is installed along with [Visual Studio IDE](#).

What Problem Are We Trying to Solve?

What is the problem we are trying to solve? This is always an important question. We're trying to solve *two* problems at once:

1. Document the steps necessary to build a relatively simple custom SSIS task.
2. Build a useful custom SSIS task.

The SSIS catalog is an awesome piece of software engineering. There are gaps in the functionality provided by the SSIS catalog—especially for enterprises practicing [Data Integration Lifecycle Management](#) (DILM) and/or DevOps.

One of the gaps is a limitation of the SSIS Execute Package Task. Developers cannot select an SSIS package in another project.

The Execute Catalog Package Task that we will build in this exercise will allow us to do just that: start an SSIS package that resides in an SSIS Catalog.

Is this a complete, production-ready implementation of an SSIS catalog package execution functionality? Goodness, no! This is a basic example of a custom SSIS task. But since you are building the functionality you can extend it at will. I *want* you to extend it!

CHAPTER 2



Creating the Assembly

Creating a custom SSIS (SQL Server Integration Services) task begins with creating a .Net assembly. We engage in nonstandard practices because we are creating a control that will eventually be used in Visual Studio projects.

In this chapter we take steps to establish the foundation of our project while considering the following steps. The very first step is how we open Visual Studio IDE (Integrated Development Environment).

Opening Visual Studio IDE

I can hear you thinking, “Hold on a minute, Andy. Do you propose to start by telling us *how to open Visual Studio?*” Yes. Yes, I do. “Why?” I’m glad you asked. This is one the tricky things that real software developers do not remember to tell you. They are not intentionally leaving stuff out to trip you up, nor are they bad people; they simply do not remember making changes to their development environments to support this kind of development.

When you open Visual Studio, right-click Visual Studio in the Windows Start menu, hover over “More,” and then click “Run as administrator” as shown in Figure 2-1.

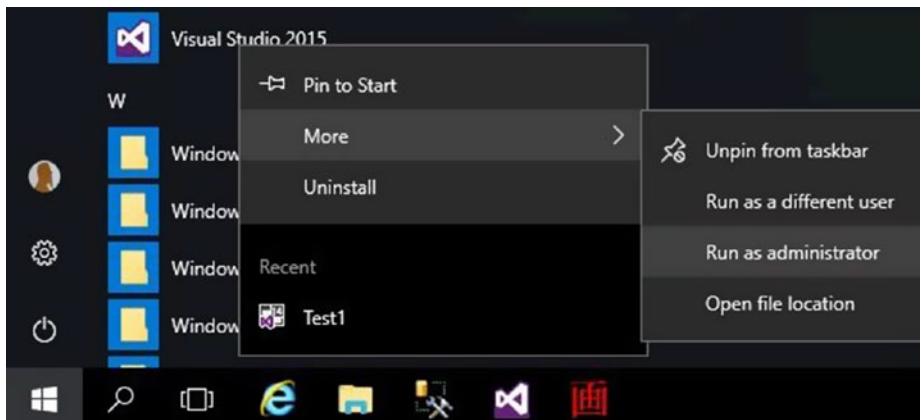


Figure 2-1. Run Visual Studio as administrator

You do not *have* to take this step. But if you do, it will save you time and effort later. Why? You may wish to attach to the devenv.exe process to debug your custom SSIS task.

When you click Run as administrator, you will be prompted to confirm that you wish to run Visual Studio as shown in Figure 2-2.

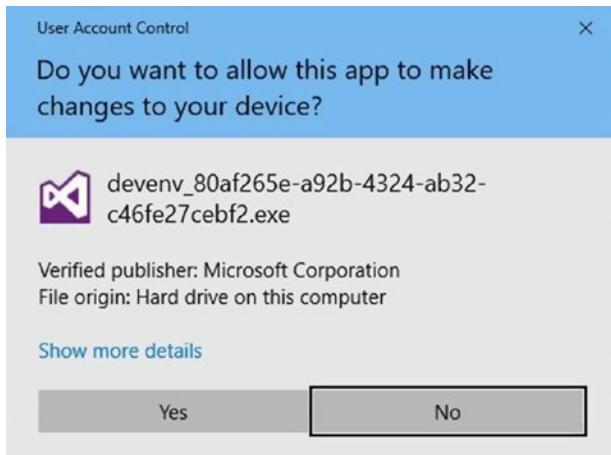


Figure 2-2. Confirm you really want to run Visual Studio as administrator

Once the Visual Studio IDE opens, click the New Project link to begin a new project as shown in Figure 2-3.

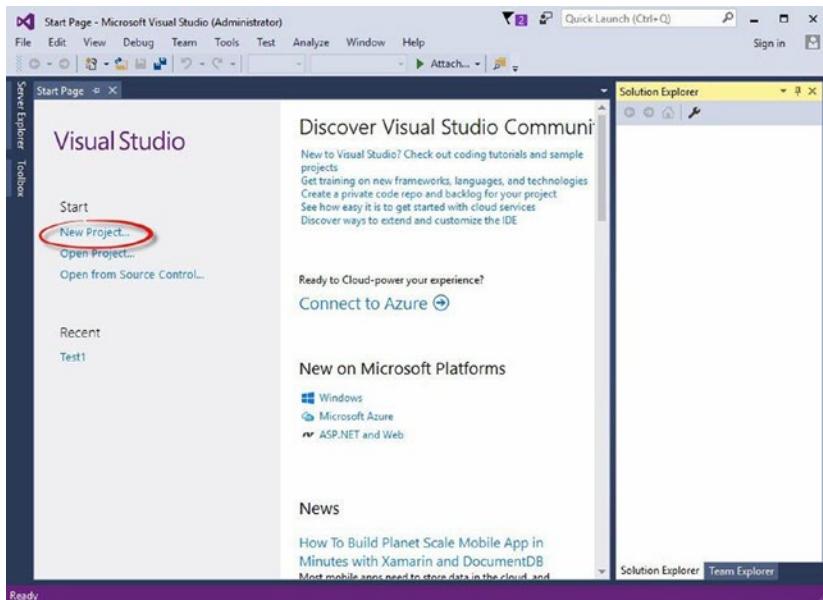


Figure 2-3. Start a new project

When the New Project window displays, select a Class Library project type in the .Net language of your choosing, and give the project a name. I chose Visual Basic for the language and named the project ExecuteCatalogPackageTask as shown in Figure 2-4.

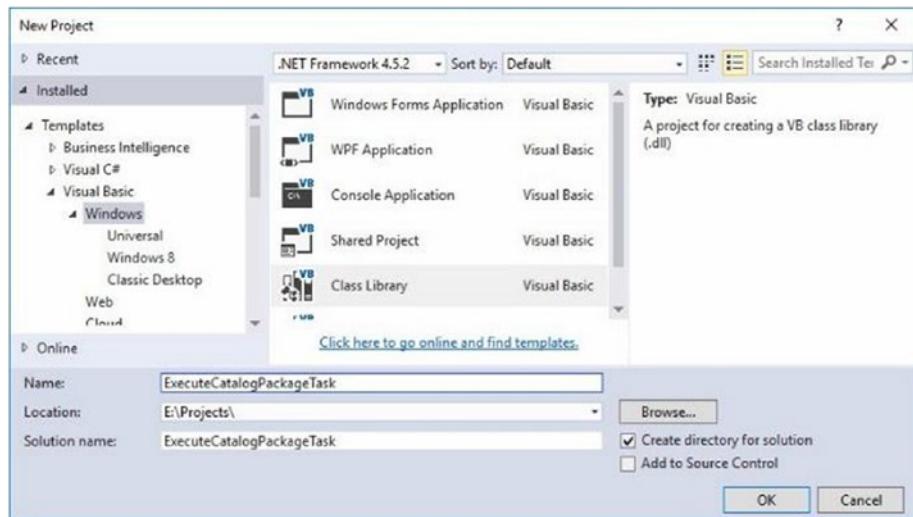


Figure 2-4. Selecting a project type and name

Click the OK button to create the project.

The next step is renaming the default class that was created as Class1 (unless you named the class as shown in Figure 2-4). In Solution Explorer, click the Class1.vb class twice (slowly) and Class1 will enter rename-edit mode. Change the name to ExecuteCatalogPackageTask as shown in Figure 2-5.

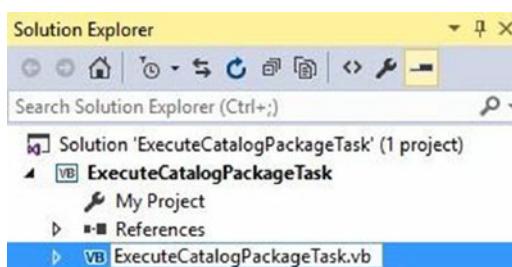


Figure 2-5. Renaming Class1 to ExecuteCatalogPackageTask

When you hit the Enter key you will be prompted to change all references to Class1 to ExecuteCatalogPackageTask. Click the Yes button as shown in Figure 2-6.

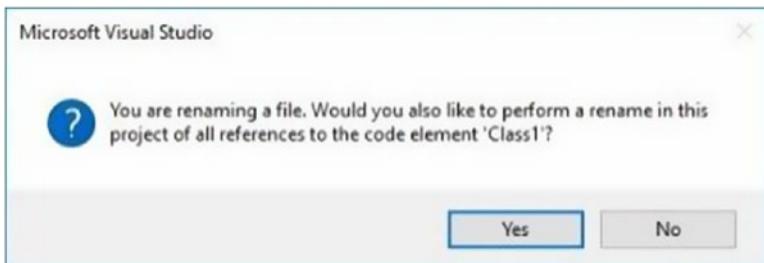


Figure 2-6. Rename all references

These are the first steps to creating a custom SSIS task using [Visual Studio](#). Experienced developers will likely find this information trivial—this book is not written for experienced developers. A key takeaway from this chapter: you must always remember to start [Visual Studio](#) as an administrator.

Adding a Reference

The .Net Framework was invented so developers could get a few hours of sleep. It encapsulates a host of common functions which developers can use in their applications. That's pretty cool, but even cooler is the fact that many platforms and applications expose functionality via .Net assemblies. Coolest of all (for our purposes), SSIS assemblies allow us to create custom tasks!

First, we must *reference* these assemblies. To do that, open the ExecuteCatalogPackageTask Visual Studio solution. Open the ExecuteCatalogPackageTask project in Solution Explorer and then expand the References virtual folder as shown in Figure 2-7.

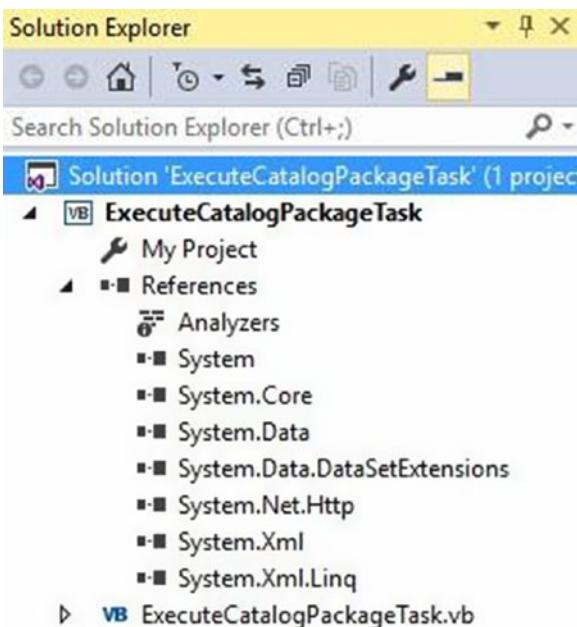


Figure 2-7. Project references in Solution Explorer

Let's add some references to SSIS assemblies. Right-click the References virtual folder, and then click Add Reference... as shown in Figure 2-8.

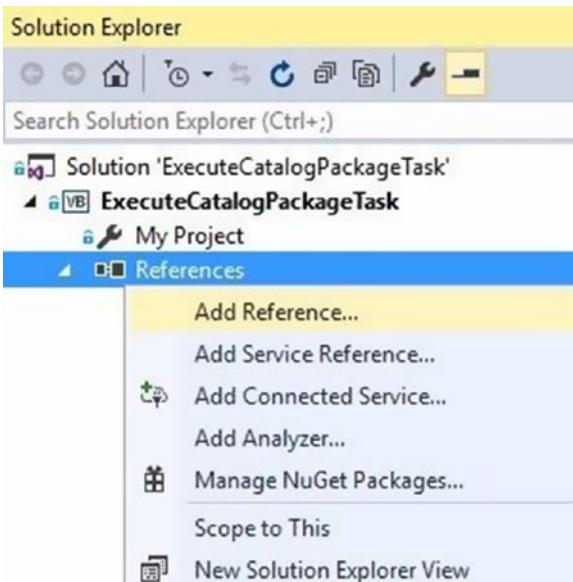


Figure 2-8. Preparing to add a reference

This opens the Reference Manager window. Expand the Assemblies item in the Reference Types list on the left. Click the Extensions item in the Assemblies list. Scroll until you find the Microsoft.SqlServer.ManagedDTS assembly and select its check box as shown in Figure 2-9.

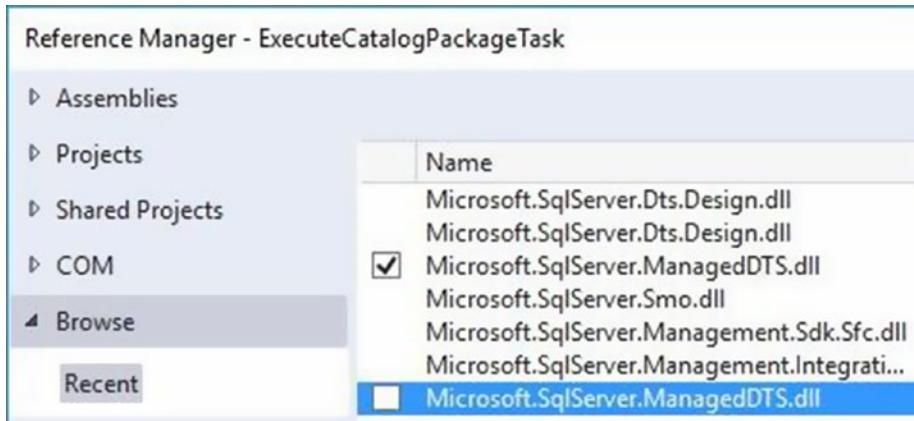


Figure 2-9. Adding the Microsoft.SqlServer.ManagedDTS Assembly reference from Assemblies\Extensions

If the assembly is *not* located in the Assemblies\Extensions list, you will have to browse for the Microsoft.SqlServer.ManagedDTS assembly. On my VM (virtual machine) I located the file in the following folder:

```
C:\Windows\Microsoft.NET\assembly\GAC_MSIL\Microsoft.SqlServer.ManagedDTS\v4.0_13.0.0.0_89845dcd8080cc91\
```

Add a reference to the Microsoft.SqlServer.ManagedDTS assembly as shown in Figure 2-10.

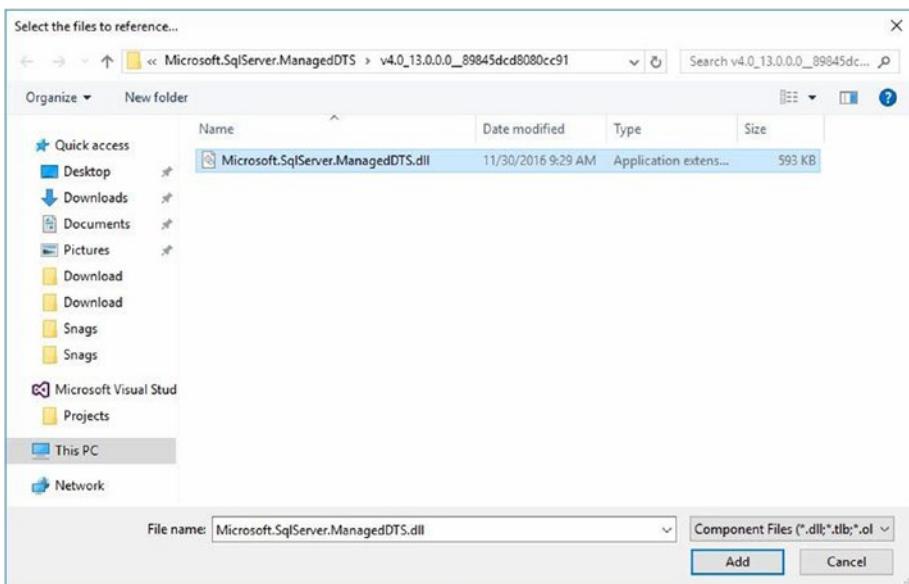


Figure 2-10. Adding the Microsoft.SqlServer.ManagedDTS Assembly reference from the file system

When you click the OK button, the References virtual folder appears as shown in Figure 2-11.

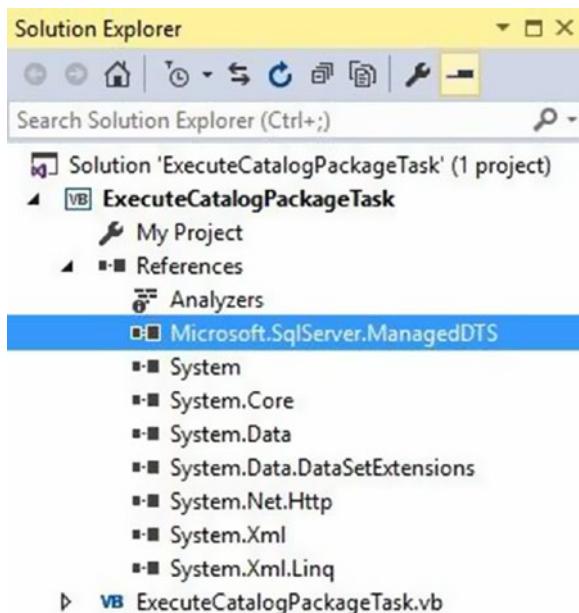


Figure 2-11. Reference successfully added!

CHAPTER 3



Signing the Assembly

In order to be used by Visual Studio, an assembly must be signed. The .Net Framework includes tools for creating and managing “key” files which are used for signing assemblies.

Preparing to Add a Key

I recommend opening a command window as an administrator to begin. As when you opened Visual Studio Express for Windows as an administrator, navigate the Windows Start menu and locate Command Prompt. Right-click Command Prompt, hover over “More,” and then click “Run as administrator” as shown in Figure 3-1.

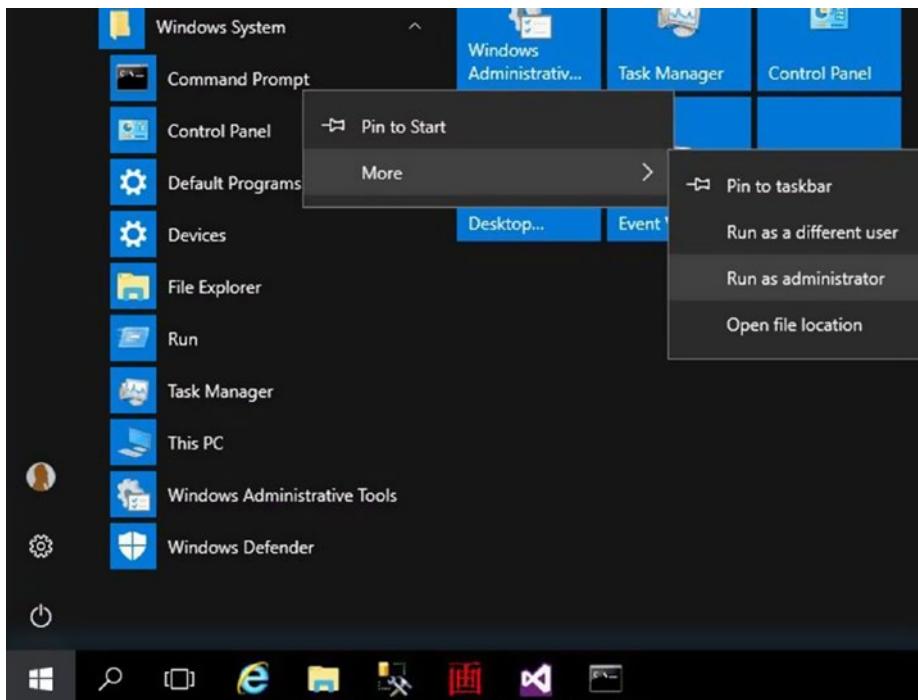


Figure 3-1. Opening an administrator command window

When prompted, click the Yes button as shown in Figure 3-2.



Figure 3-2. Allowing Command Prompt to execute as administrator

When open, the Administrator command window appears as shown in Figure 3-3.

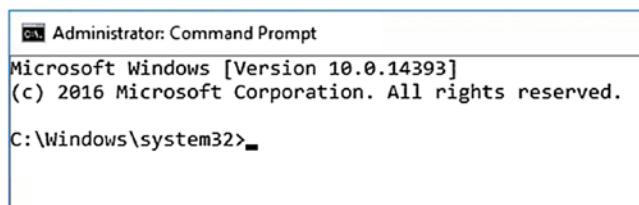


Figure 3-3. Administrator command window

We are creating an assembly (actually, a couple of assemblies) that will reside in the GAC (Global Assembly Cache). In order to live in the GAC, assemblies must be *signed*. We sign assemblies using *strong name key files*. At this point in our project configuration, we need to generate a key suitable to sign our assemblies. To that end, there is good and bad news.

- The good news: Microsoft provides a strong name utility for creating keys.
- The bad news: It moves around with new releases of the .Net Framework.

If you are developing on Windows Server 2016, you can find the strong name utility in the C:\Program Files (x86)\Microsoft SDKs\Windows\v10.0A\bin\NETFX 4.6.1 Tools folder. It is named "sn.exe" so the full path is "C:\Program Files (x86)\Microsoft SDKs\Windows\v10.0A\bin\NETFX 4.6.1 Tools\sn.exe". If you are reading the post in the distant future and snickering about how archaic Windows Server 2016 was, search for "sn.exe" and use the latest version you find (smarty-pants).

Tip You can do this however you like, but I am going to recommend you start a text file to preserve these command lines for future use. I firmly believe you will thank me later.

I open Notepad and paste the following lines:

```
-- key generation
"C:\Program Files (x86)\Microsoft\SDKs\Windows\v10.0A\bin\NETFX 4.6.1
Tools\sn.exe" → -k key.snk
"C:\Program Files (x86)\Microsoft\SDKs\Windows\v10.0A\bin\NETFX 4.6.1
Tools\sn.exe" → -p key.snk public.out
"C:\Program Files (x86)\Microsoft\SDKs\Windows\v10.0A\bin\NETFX 4.6.1
Tools\sn.exe" → -t public.out
```

The first line creates the public/private key pair and puts them in a file named key.snk. The second line extracts the public part of the key pair to a file named public.out. The third line reads the public key from the public.out file.

Note The → symbol means this should all be on one line, but Microsoft Word doesn't create wide enough lines for me to represent the code as such.

Add the Notes File to the Project. If you have followed my advice, you have a Visual Studio solution, a command window, and Notepad open. Save the Notepad file as ExecuteCatalogPackageTaskNotes.txt and remember where you save it. Return to Visual Studio. In Solution Explorer right-click the ExecuteCatalogPackageTask project, hover over Add, then click Existing Item... as shown in Figure 3-4.



Figure 3-4. Adding an existing item to the ExecuteCatalogPackageTask project

Navigate to the location of your `ExecuteCatalogPackageTaskNotes.txt` file, select it, then click the Add button as shown in Figure 3-5.

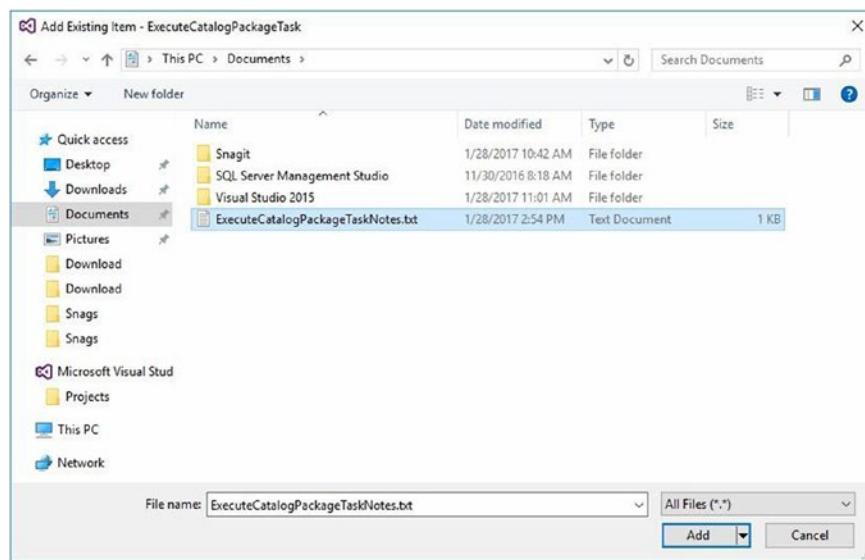


Figure 3-5. Find the `ExecuteCatalogPackageTaskNotes.txt` file

`ExecuteCatalogPackageTaskNotes.txt` now appears in Solution Explorer as shown in Figure 3-6.

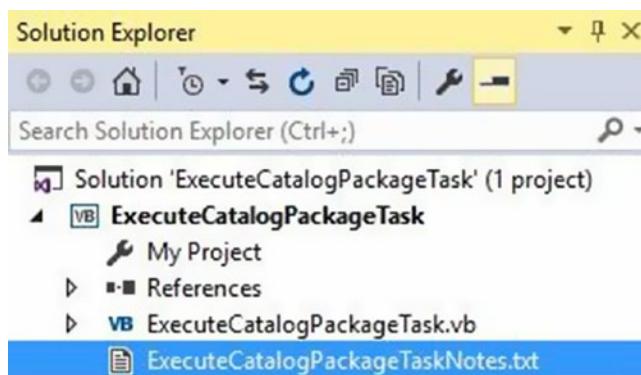


Figure 3-6. `ExecuteCatalogPackageTaskNotes.txt` in the solution

`ExecuteCatalogPackageTaskNotes.txt` has also been added to your project folder as shown in Figure 3-7.

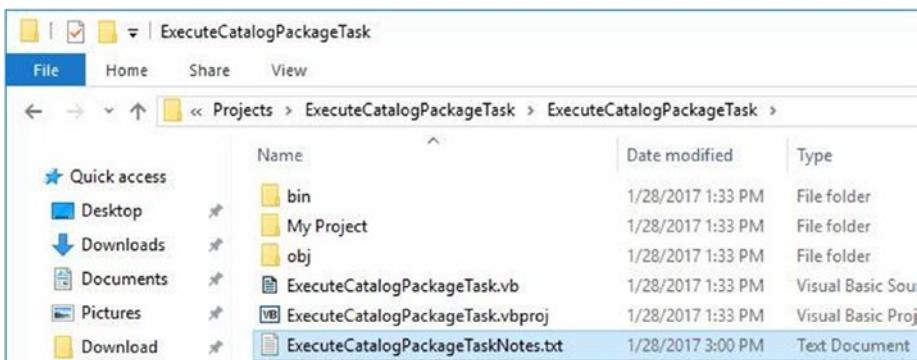


Figure 3-7. *ExecuteCatalogPackageTaskNotes.txt* in the project folder

Now that you have a copy of the file included with your project, delete the original from the location where you originally stored it as shown in Figure 3-8.

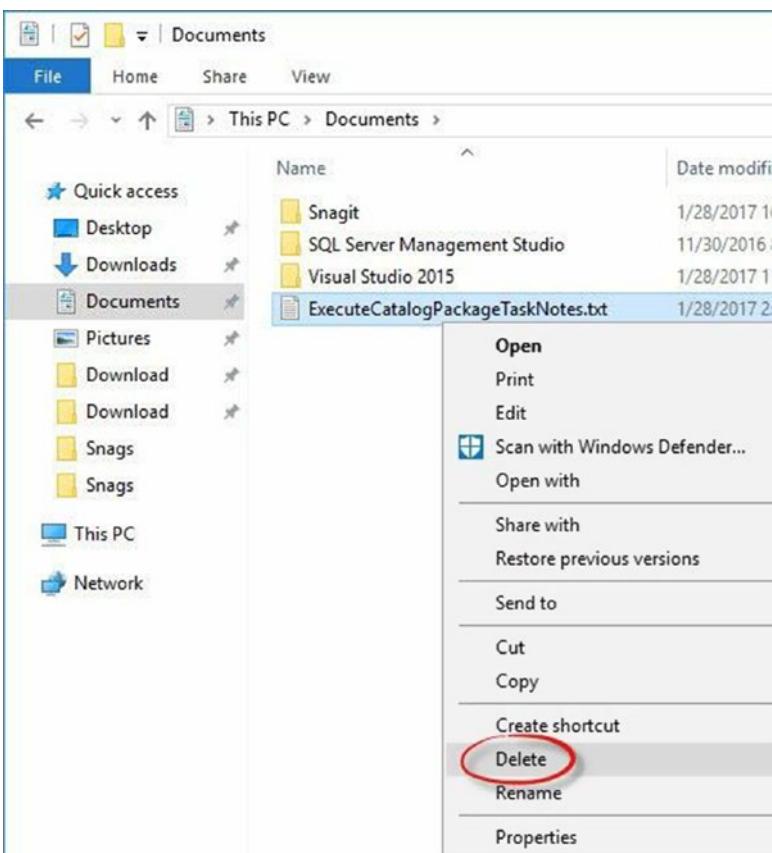


Figure 3-8. Deleting the original *ExecuteCatalogPackageTaskNotes.txt*

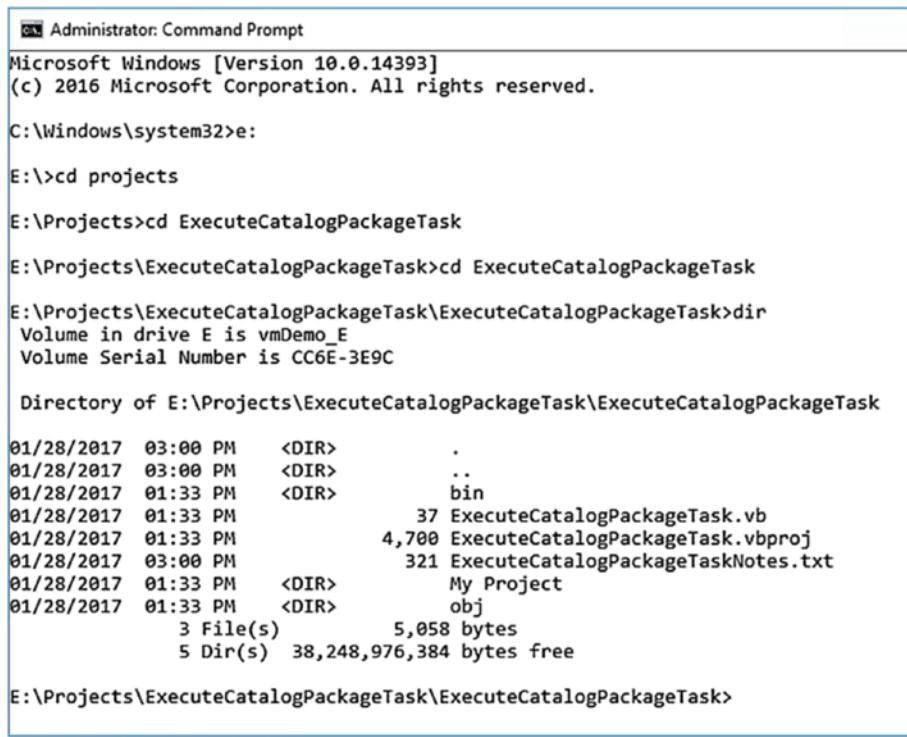
Why delete the original? Visual Studio made a *copy* of your file when it added it to your solution. We want to continue to work with that copy, not the one we made before adding it to the project.

Now would be an excellent time to take advantage of some free source control functionality available at visualstudio.com/team-services.

There are two types of software developers: Those who use source control and those who will. The first time you lose a few hours of awesome-break-through coding, you will. I promise. If you use a source control product to maintain your solution, the file you just added is part of that solution. That's why we added it to the project.

Creating the Key

In the command window, navigate to the ExecuteCatalogPackageTask project directory. The project directory will contain ExecuteCatalogPackageTask.vb, ExecuteCatalogPackageTask.vbproj, and ExecuteCatalogPackageTask Notes.txt as shown in Figure 3-9.



```
Administrator: Command Prompt
Microsoft Windows [Version 10.0.14393]
(c) 2016 Microsoft Corporation. All rights reserved.

C:\Windows\system32>e:

E:\>cd projects

E:\Projects>cd ExecuteCatalogPackageTask

E:\Projects\ExecuteCatalogPackageTask>cd ExecuteCatalogPackageTask

E:\Projects\ExecuteCatalogPackageTask\ExecuteCatalogPackageTask>dir
Volume in drive E is vmDemo_E
Volume Serial Number is CCG6E-3E9C

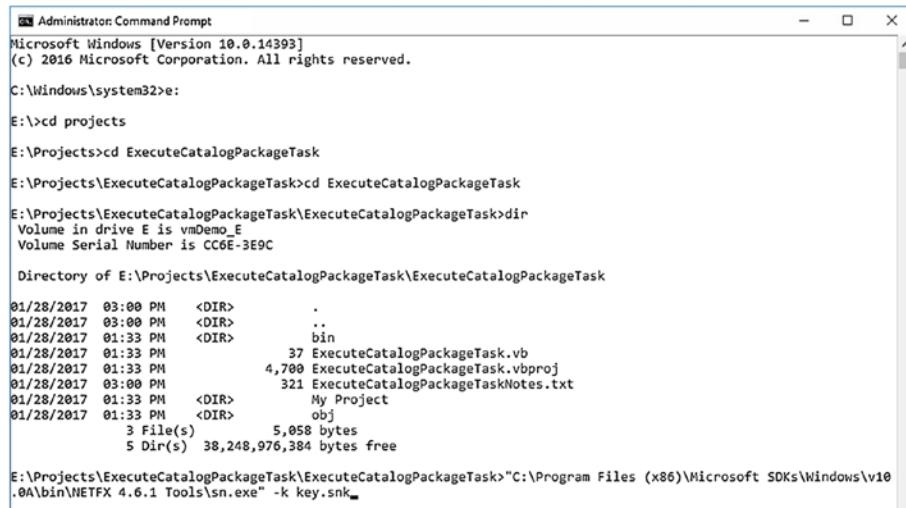
Directory of E:\Projects\ExecuteCatalogPackageTask\ExecuteCatalogPackageTask

01/28/2017  03:00 PM    <DIR>      .
01/28/2017  03:00 PM    <DIR>      ..
01/28/2017  01:33 PM    <DIR>      bin
01/28/2017  01:33 PM                37 ExecuteCatalogPackageTask.vb
01/28/2017  01:33 PM                4,700 ExecuteCatalogPackageTask.vbproj
01/28/2017  03:00 PM                321 ExecuteCatalogPackageTaskNotes.txt
01/28/2017  01:33 PM    <DIR>      My Project
01/28/2017  01:33 PM    <DIR>      obj
                  3 File(s)      5,058 bytes
                  5 Dir(s)  38,248,976,384 bytes free

E:\Projects\ExecuteCatalogPackageTask\ExecuteCatalogPackageTask>
```

Figure 3-9. Navigating to the ExecuteCatalogPackageTask project directory

If it is not already open, open the ExecuteCatalogPackageTaskNotes.txt in Visual Studio from Solution Explorer. Select the first line, copy it, and paste it into the command window as shown in Figure 3-10.



The screenshot shows a Windows Command Prompt window titled "Administrator: Command Prompt". The window displays the following command history and output:

```
Administrator: Command Prompt
Microsoft Windows [Version 10.0.14393]
(c) 2016 Microsoft Corporation. All rights reserved.

C:\Windows\system32>e:
E:>cd projects
E:\Projects>cd ExecuteCatalogPackageTask
E:\Projects\ExecuteCatalogPackageTask>cd ExecuteCatalogPackageTask
E:\Projects\ExecuteCatalogPackageTask\ExecuteCatalogPackageTask>dir
Volume in drive E is vMDemo_E
Volume Serial Number is CC6E-3E9C

Directory of E:\Projects\ExecuteCatalogPackageTask\ExecuteCatalogPackageTask

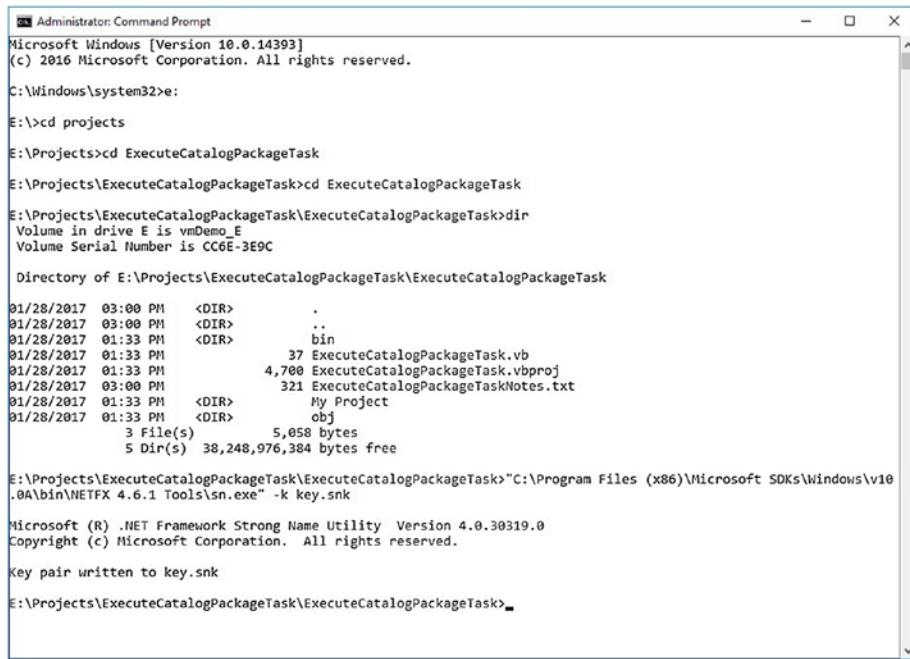
01/28/2017  03:00 PM    <DIR>          .
01/28/2017  03:00 PM    <DIR>          ..
01/28/2017  01:33 PM    <DIR>          bin
01/28/2017  01:33 PM            37 ExecuteCatalogPackageTask.vb
01/28/2017  01:33 PM        4,700 ExecuteCatalogPackageTask.vbproj
01/28/2017  03:00 PM        321 ExecuteCatalogPackageTaskNotes.txt
01/28/2017  01:33 PM    <DIR>          My Project
01/28/2017  01:33 PM    <DIR>          obj
            3 File(s)           5,058 bytes
            5 Dir(s)  38,248,976,384 bytes free

E:\Projects\ExecuteCatalogPackageTask\ExecuteCatalogPackageTask>"C:\Program Files (x86)\Microsoft SDKs\Windows\v10.0A\bin\NETFX 4.6.1 Tools\sn.exe" -k key.snk
```

Figure 3-10. Pasting into the command window

Press the Enter key to execute the sn.exe command, creating the key file in the ExecuteCatalogPackageTask project directory as shown in Figure 3-11.

CHAPTER 3 ■ SIGNING THE ASSEMBLY



The screenshot shows a Windows Command Prompt window titled "Administrator: Command Prompt". The command history and output are as follows:

```
Microsoft Windows [Version 10.0.14393]
(c) 2016 Microsoft Corporation. All rights reserved.

C:\Windows\system32>e:
E:>cd projects
E:\Projects>cd ExecuteCatalogPackageTask
E:\Projects\ExecuteCatalogPackageTask>cd ExecuteCatalogPackageTask
E:\Projects\ExecuteCatalogPackageTask\ExecuteCatalogPackageTask>dir
Volume in drive E is vmdemo_E
Volume Serial Number is CCE6-3E9C

Directory of E:\Projects\ExecuteCatalogPackageTask\ExecuteCatalogPackageTask

01/28/2017  03:00 PM    <DIR>        .
01/28/2017  03:00 PM    <DIR>        ..
01/28/2017  01:33 PM    <DIR>        bin
01/28/2017  01:33 PM           37 ExecuteCatalogPackageTask.vb
01/28/2017  01:33 PM          4,700 ExecuteCatalogPackageTask.vbproj
01/28/2017  03:00 PM          321 ExecuteCatalogPackageTaskNotes.txt
01/28/2017  01:33 PM    <DIR>        My Project
01/28/2017  01:33 PM    <DIR>        obj
               3 File(s)      5,058 bytes
               5 Dir(s)   38,248,976,384 bytes free

E:\Projects\ExecuteCatalogPackageTask\ExecuteCatalogPackageTask>"C:\Program Files (x86)\Microsoft SDKs\Windows\v10.0A\bin\NETFX 4.6.1 Tools\snk.exe" -k key.snk

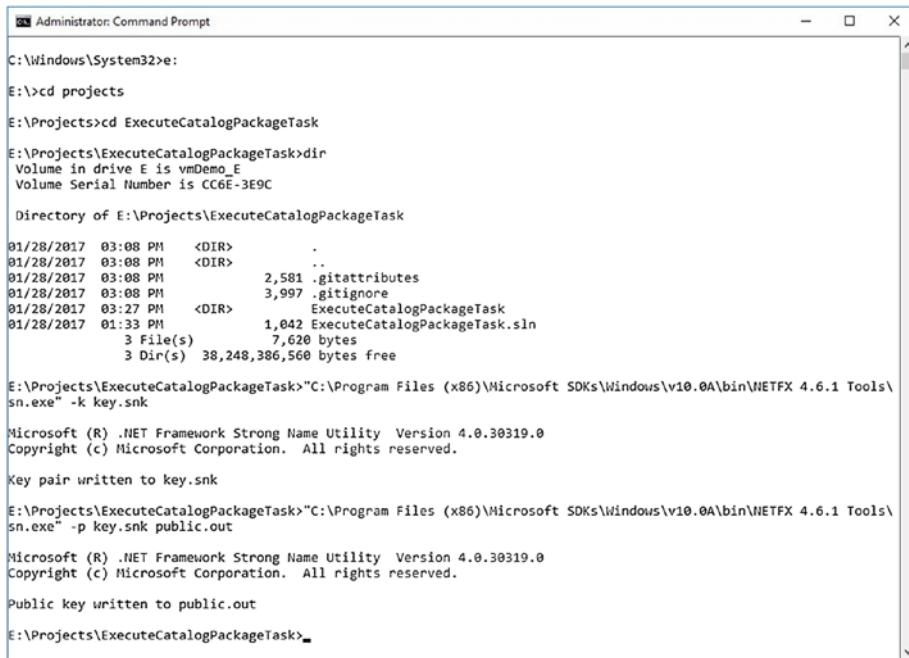
Microsoft (R) .NET Framework Strong Name Utility Version 4.0.30319.0
Copyright (c) Microsoft Corporation. All rights reserved.

Key pair written to key.snk

E:\Projects\ExecuteCatalogPackageTask\ExecuteCatalogPackageTask>
```

Figure 3-11. Key.snk file created!

Next we need to get the public key from key.snk's public/private pair. First, we extract the public key to a file named public.out using the second line stored in ExecuteCatalogPackageTaskNotes.txt as shown in Figure 3-12.



```

Administrator: Command Prompt
C:\Windows\System32>e:
E:>cd projects
E:\Projects>cd ExecuteCatalogPackageTask
E:\Projects\ExecuteCatalogPackageTask>dir
Volume in drive E is vmdemo_E
Volume Serial Number is CC6E-3E9C

Directory of E:\Projects\ExecuteCatalogPackageTask

01/28/2017  03:08 PM    <DIR>          .
01/28/2017  03:08 PM    <DIR>          ..
01/28/2017  03:08 PM           2,581 .gitattributes
01/28/2017  03:08 PM           3,997 .gitignore
01/28/2017  03:27 PM    <DIR>          ExecuteCatalogPackageTask
01/28/2017  01:33 PM           1,042 ExecuteCatalogPackageTask.sln
               3 File(s)        7,620 bytes
               3 Dir(s) 38,248,386,560 bytes free

E:\Projects\ExecuteCatalogPackageTask>"C:\Program Files (x86)\Microsoft SDKs\Windows\v10.0A\bin\NETFX 4.6.1 Tools\sn.exe" -k key.snk

Microsoft (R) .NET Framework Strong Name Utility Version 4.0.30319.0
Copyright (c) Microsoft Corporation. All rights reserved.

Key pair written to key.snk

E:\Projects\ExecuteCatalogPackageTask>"C:\Program Files (x86)\Microsoft SDKs\Windows\v10.0A\bin\NETFX 4.6.1 Tools\sn.exe" -p key.snk public.out

Microsoft (R) .NET Framework Strong Name Utility Version 4.0.30319.0
Copyright (c) Microsoft Corporation. All rights reserved.

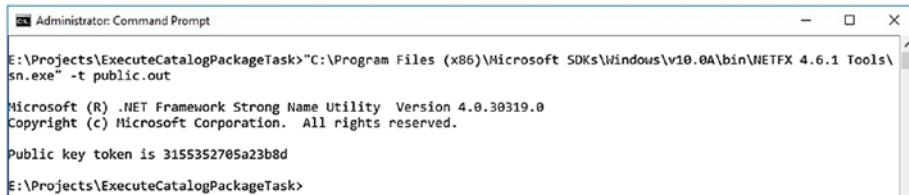
Public key written to public.out

E:\Projects\ExecuteCatalogPackageTask>_

```

Figure 3-12. Public key extraction

Finally, we need to read the public key value out of the file to which we extracted it as shown in Figure 3-13.



```

Administrator: Command Prompt
E:\Projects\ExecuteCatalogPackageTask>"C:\Program Files (x86)\Microsoft SDKs\Windows\v10.0A\bin\NETFX 4.6.1 Tools\sn.exe" -t public.out

Microsoft (R) .NET Framework Strong Name Utility Version 4.0.30319.0
Copyright (c) Microsoft Corporation. All rights reserved.

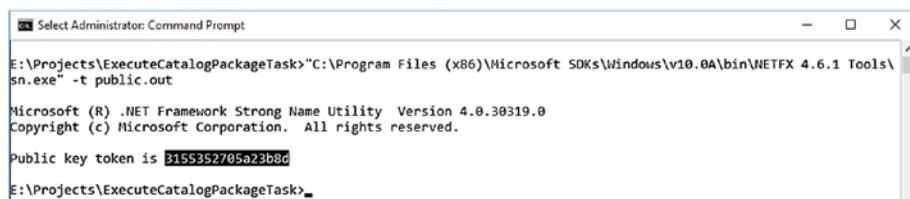
Public key token is 3155352705a23b8d

E:\Projects\ExecuteCatalogPackageTask>_

```

Figure 3-13. Reading the public key

Using the left mouse button, highlight the public key value as shown in Figure 3-14.

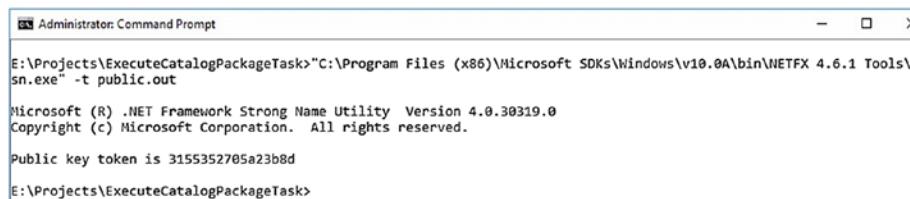


```
Administrator: Command Prompt
E:\Projects\ExecuteCatalogPackageTask>"C:\Program Files (x86)\Microsoft SDKs\Windows\v10.0A\bin\NETFX 4.6.1 Tools\sn.exe" -t public.out
Microsoft (R) .NET Framework Strong Name Utility Version 4.0.30319.0
Copyright (c) Microsoft Corporation. All rights reserved.

Public key token is 3155352705a23b8d
E:\Projects\ExecuteCatalogPackageTask>
```

Figure 3-14. Selecting the public key in the command window

Right-click the highlighted text. Note that no context menu will appear. Your indication that the text has been copied to the clipboard is that the highlighting and “Select” text in the title bar disappear as shown in Figure 3-15.

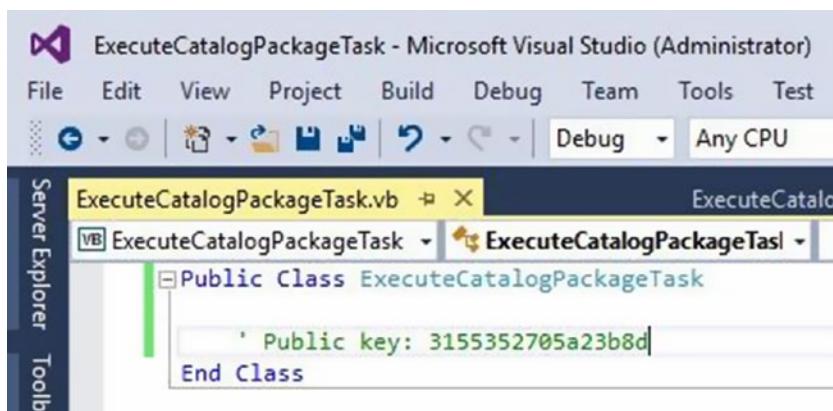


```
Administrator: Command Prompt
E:\Projects\ExecuteCatalogPackageTask>"C:\Program Files (x86)\Microsoft SDKs\Windows\v10.0A\bin\NETFX 4.6.1 Tools\sn.exe" -t public.out
Microsoft (R) .NET Framework Strong Name Utility Version 4.0.30319.0
Copyright (c) Microsoft Corporation. All rights reserved.

Public key token is 3155352705a23b8d
E:\Projects\ExecuteCatalogPackageTask>
```

Figure 3-15. Text selected to the clipboard

Store the public key in a comment inside the class ExecuteCatalogPackageTask.vb in Visual Studio. We will use it here later as shown in Figure 3-16.



```
ExecuteCatalogPackageTask - Microsoft Visual Studio (Administrator)
File Edit View Project Build Debug Team Tools Test
ExecuteCatalogPackageTask.vb X ExecuteCatalog...
VB ExecuteCatalogPackageTask ExecuteCatalogPackageTask
Public Class ExecuteCatalogPackageTask
    ' Public key: 3155352705a23b8d
End Class
```

Figure 3-16. Public key stored

Applying the Key

The key has been created and we have extracted the required metadata. It is now time to apply the key file to the solution. In Solution Explorer, open My Project as shown in Figure 3-17.

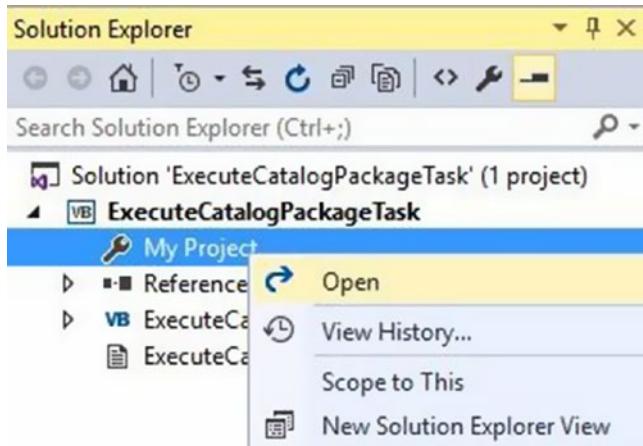


Figure 3-17. Opening My Project

Once My Project opens, navigate to the Signing page. Check the “Sign the assembly” check box. Click the “Choose a strong name key file” drop-down and select “<Browse...>” as shown in Figure 3-18.

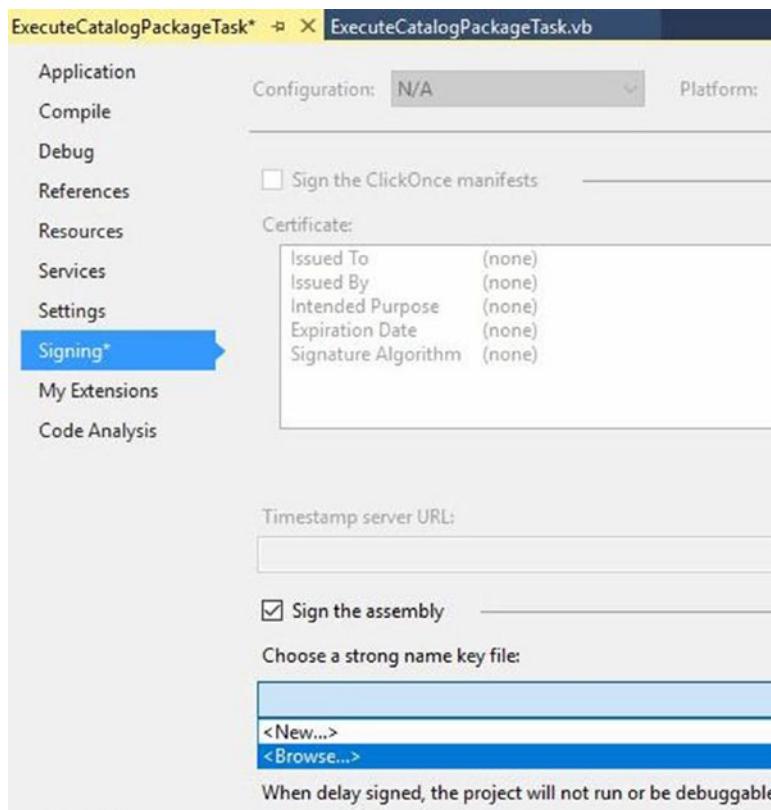


Figure 3-18. Signing the assembly

Browse to the key.snk file in the ExecuteCatalogPackageTask project folder as shown in Figure 3-19.

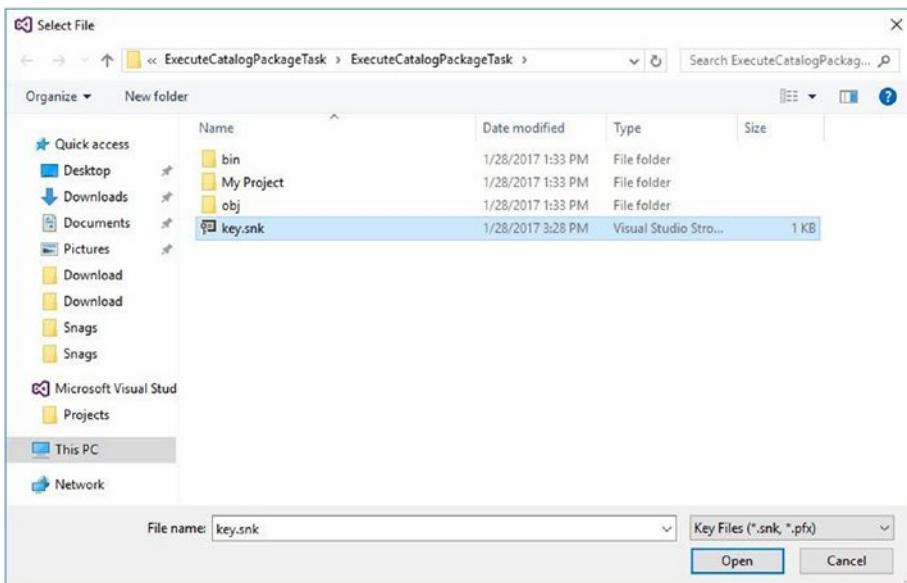


Figure 3-19. Navigating to the key.snk file

Click the Open button to use the key.snk file to sign the assembly as shown in Figure 3-20.

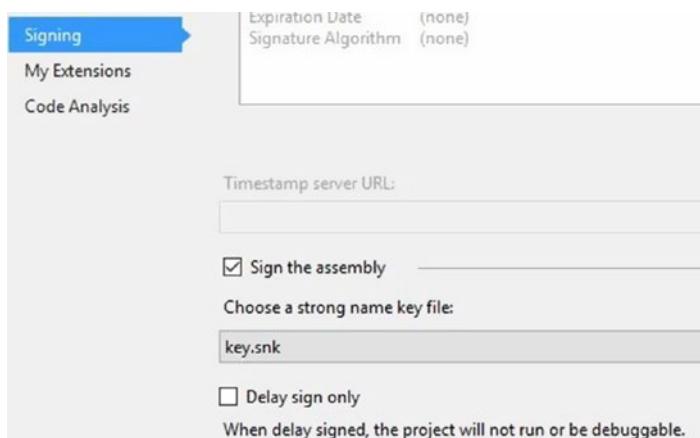


Figure 3-20. Signed assembly!

Click the Save All button (or File ▶ Save All) to save your Visual Studio solution. If you are using source control, this is an excellent time to save your file to Source Control.

We have laid a solid foundation in creating and signing our assembly. Our next move is to protect our work via source control.

CHAPTER 4



Preparing the Environment

One of the first things martial arts students are taught is how to fall. Why? So they don't hurt themselves. As we begin developing this task, you are going to experience failures. Tests will fail, code will not behave as anticipated, and things will go wrong. When they do, I want you to know a couple things:

1. Failure is normal.
2. How to recover.

I cannot overemphasize the need for source control. I use [Microsoft's Visual Studio Team Services](#) (VSTS) but feel free to use any of the excellent source control solutions available. At the time of this writing, VSTS is free for teams of five or fewer. Source control is your number-one method for code recovery. Do it. (You have been warned.)

Adding the Solution to Source Control

In this section, I assume you have established an account with a source control provider. Having done so, you can configure Visual Studio to use that provider.

First, configure Visual Studio Source Control options by clicking Tools ▶ Options as shown in Figure 4-1.

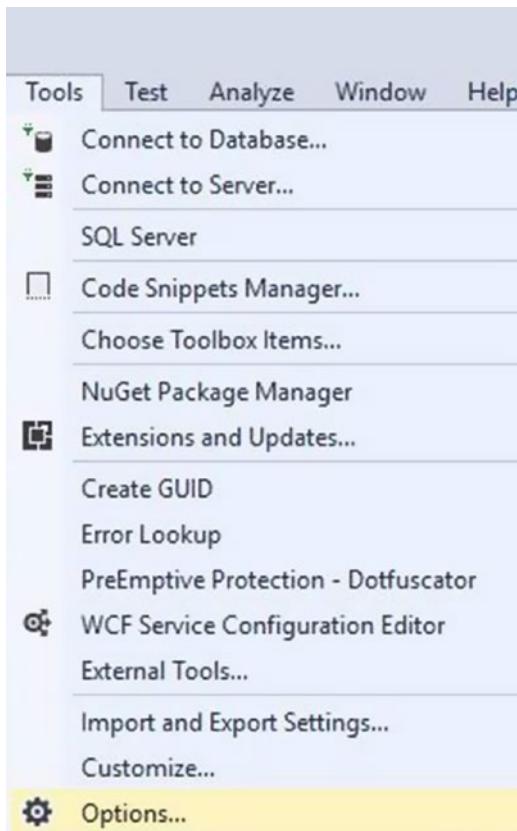


Figure 4-1. Opening Tools ► Options

When the Options window displays, expand the Source Control node and select Plug-in Selection. Select your source control plug-in from the “Current source control plug-in” drop-down as shown in Figure 4-2.

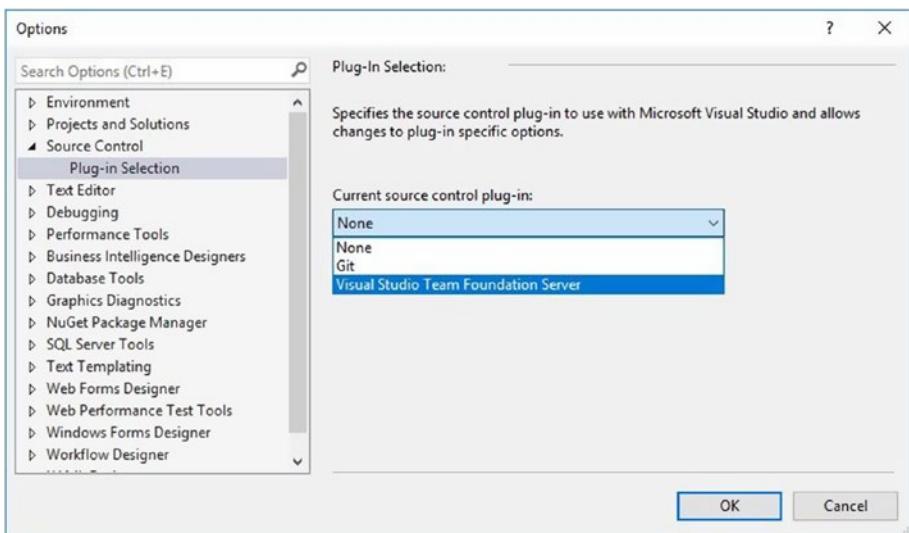


Figure 4-2. Selecting current source control plug-in

If you are using Visual Studio Team Foundation Server like me, click Team ► Manage Connections as shown in Figure 4-3.

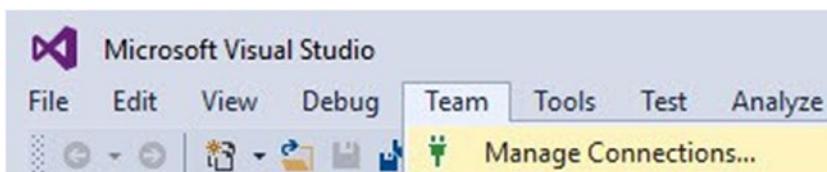


Figure 4-3. Managing connections

You may need to configure a new server and new connection to a team project during this step. Once configured, use Team Explorer to select the team project. My team project is named “Community,” as you can see in Figure 4-4.

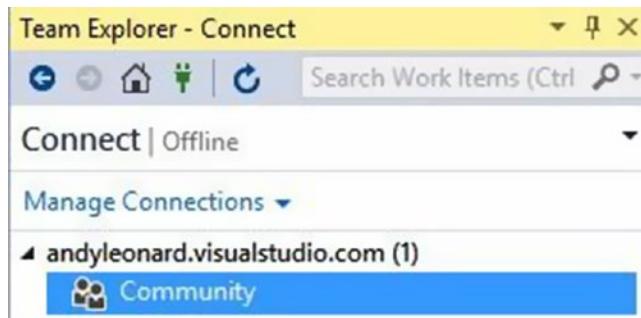


Figure 4-4. Connecting to the Community team project

Return to Solution Explorer. Right-click the ExecuteCatalogPackageTask solution and click “Add Solution to Source Control...,” as shown in Figure 4-5.

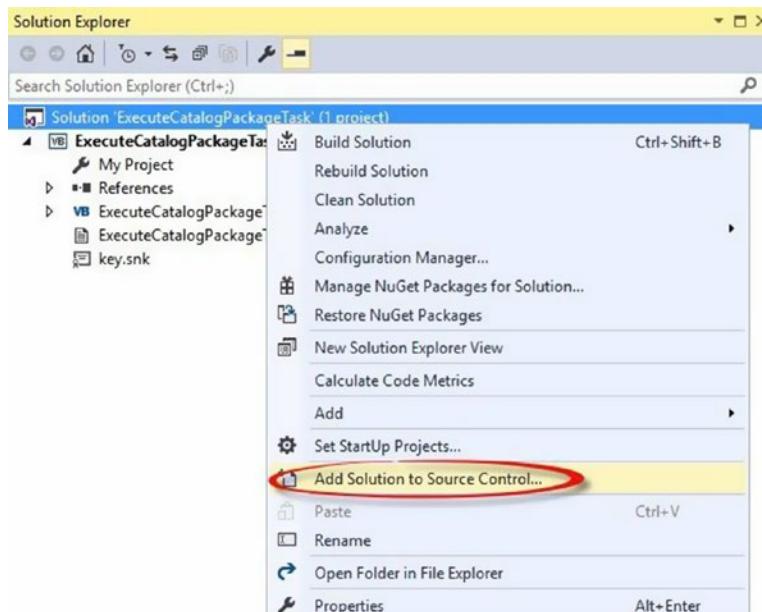


Figure 4-5. Adding solution to Source Control

When the “Add Solution ExecuteCatalogPackageTask to Source Control” window displays, I select the “Community” team project as shown in Figure 4-6.

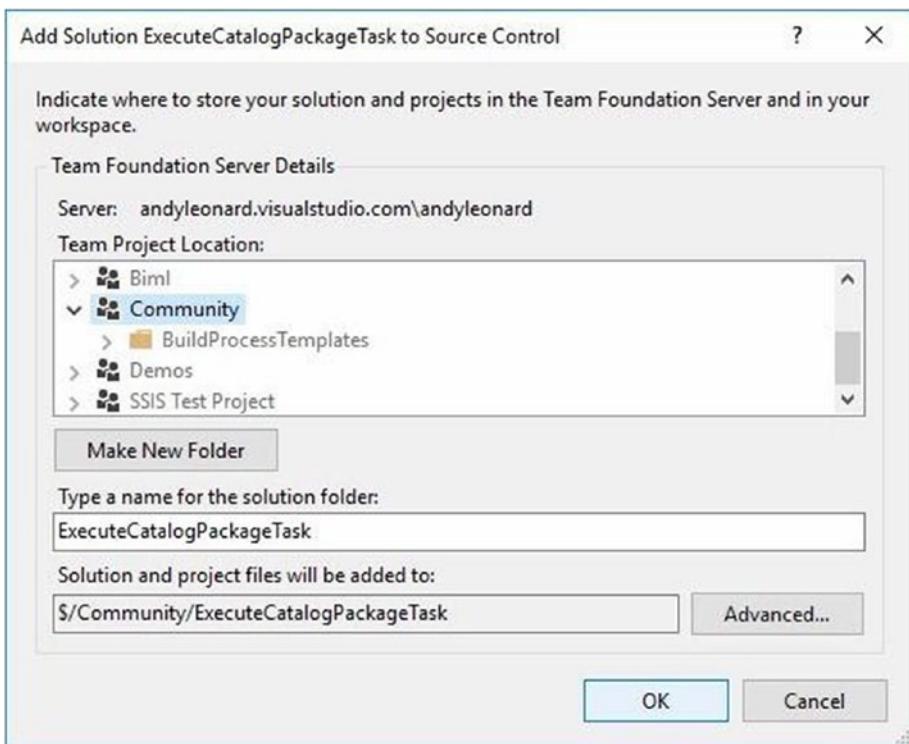


Figure 4-6. Selecting the Community team project

Red check marks beside each artifact indicate that the solution, project, and other project artifacts have been added to source control as seen in Figure 4-7.

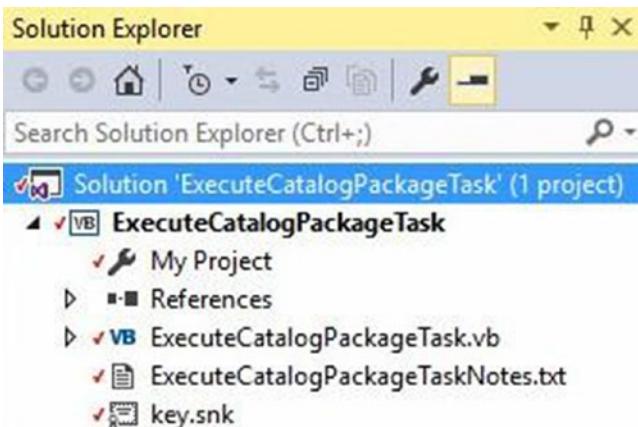


Figure 4-7. Added to source control

To check in the solution, right-click the solution name in Solution Explorer and then click “Check In...” as shown in Figure 4-8.

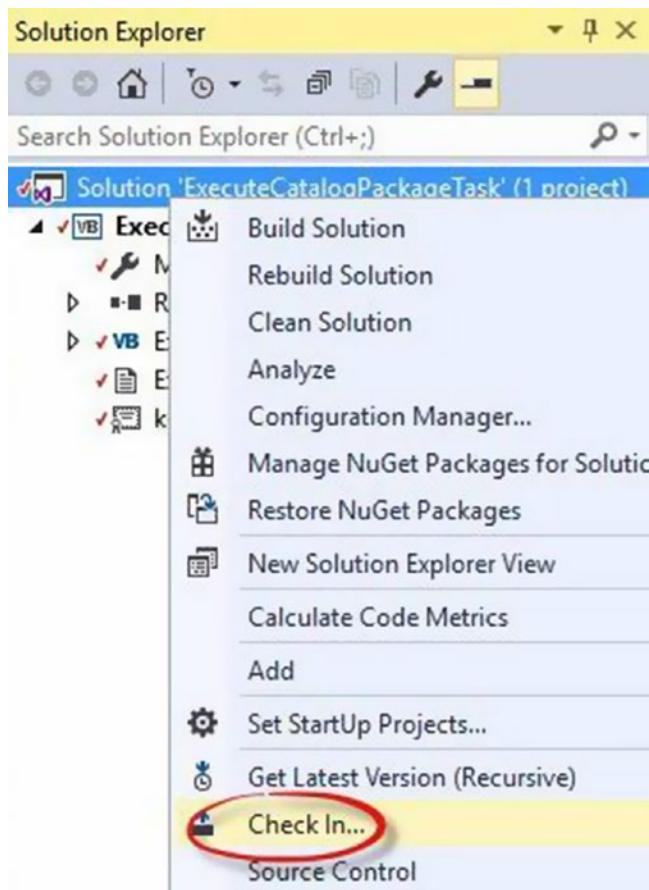


Figure 4-8. Checking in the solution

The Team Explorer – Pending Changes window displays. Enter an optional comment and click the Check In button to check in this version of the code as seen in Figure 4-9.

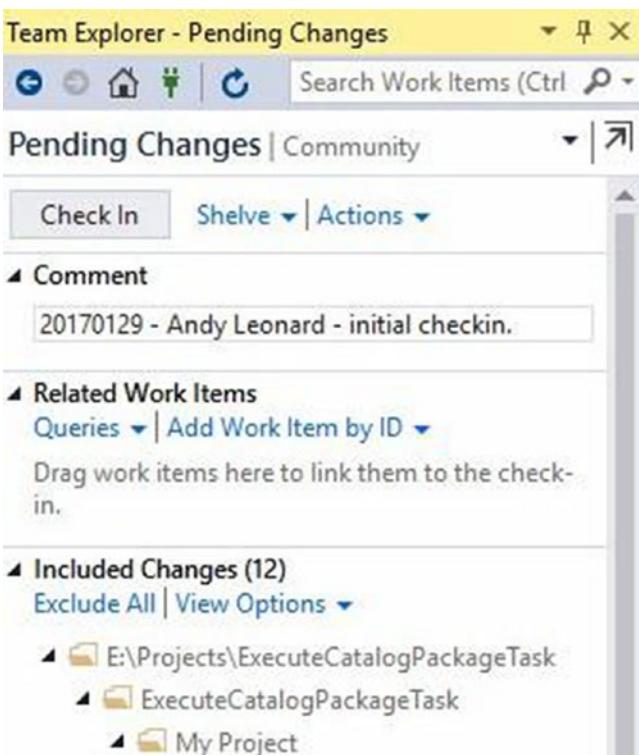


Figure 4-9. Annotating pending changes

Figure 4-10 shows blue lock icons mark items that have been checked into source control.

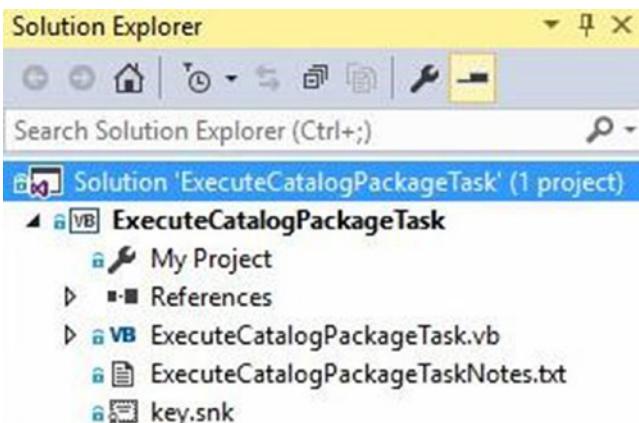


Figure 4-10. Observing checked-in artifacts

Preparing to Build

Building custom SSIS (SQL Server Integration Services) tasks is neither trivial nor easy. You are building components that will be used to construct software. Perhaps you have done this before. Many have not. One key is recovering gracefully when the code does not perform as expected. Here's what I do when that happens.

- Unregister the assembly from the GAC (Global Assembly Cache).
- Clean the Visual Studio solution.
- Make another attempt at coding the desired functionality.
- Build the assembly.
- Register the assembly in the GAC.

I find these steps get me out of trouble faster than I get myself into it, and that is a good thing. How to accomplish it? First, find gacutil.exe. Like sn.exe mentioned in Chapter 3, gacutil moves around with each release of the .Net Framework. At the time of this writing, I find the version I want in the C:\Program Files (x86)\Microsoft SDKs\Windows\v10.0A\bin\NETFX 4.6.1 Tools folder.

Open the ExecuteCatalogPackageTask solution, and then open the ExecuteCatalogPackageTaskNotes.txt file. Assuming you substitute the proper paths, add the following text to the file:

```
-- register
"C:\Program Files (x86)\Microsoft SDKs\Windows\v10.0A\bin\NETFX 4.6.1 Tools\
gacutil.exe" -if "<SSIS Tasks subfolder>ExecuteCatalogPackageTask.dll"
-- unregister
"C:\Program Files (x86)\Microsoft SDKs\Windows\v10.0A\bin\NETFX 4.6.1 Tools\
gacutil.exe" -u ExecuteCatalogPackageTask
```

How does this work? The first line—labeled “register”—uses gacutil to register the ExecuteCatalogPackageTask dynamic-link library (DLL) file with the GAC. The second line unregisters the same assembly—effectively removing it from the GAC.

Note The line breaks in this document do **not** represent CRLF (Carriage Return Line Feed) line breaks. Please do not press the Enter key until the entire command string has been entered. The text in your Command Prompt window may also wrap to a new line. This characters-per-line limitation is common in Command Prompt windows and in books containing software code.

These comprise the last and first steps (respectively) for recovery from failure listed previously. You run these steps in a command window opened with “Run as Administrator” permissions as discussed in Chapter 3.

With other editions of Visual Studio such as Professional and Ultimate, you can add pre- and post-build events to execute these commands automagically. If you choose to develop in Microsoft Visual C# in Visual Studio Community, you can also add pre- and post-build events. Just not in Visual Basic. I am unsure why this functionality was not included with Visual Basic Community Edition.

You clean and build the solution from the Build drop-down menu in Visual Studio as shown in Figure 4-11.

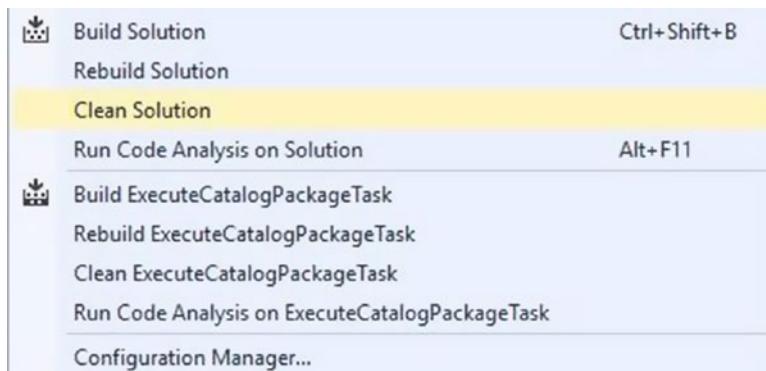


Figure 4-11. The Visual Studio Build Menu

Cleaning is the second step and Building is the fourth step listed earlier for recovering from a failed attempt. I often switch steps two and three, diving into a fix, patching, or adding additional functionality before Cleaning and then Building the solution. For some steps, the order of execution matters—but not for all.

Setting the Output Path

The reason we need to open Visual Studio Community in “Run as administrator” mode is so we can build the output—the DLL, or assembly—to a subfolder in the 32-bit Program Files folder. There are other ways to get the assembly into the Program Files subfolder, but I doubt there are many with fewer steps (without using a non-Community edition of Visual Studio or Visual C#).

Locate the SSIS Tasks subfolder used by SQL Server Data Tools (SSDT) to build SSIS packages. It will be located at `<installation drive>:\Program Files (x86)\Microsoft SQL Server\<version>\DTS\Tasks`. I installed SQL Server 2016 to my E: drive, so my path to the SSIS Tasks folder is: “E:\Program Files (x86)\Microsoft SQL Server\130\DTS\Tasks”. Replace the `<SSIS Tasks subfolder>` placeholder with the path to your SSIS Tasks subfolder in the “register” command listed earlier and in your `ExecuteCatalogPackageTaskNotes.txt` file.

For me, register now appears as follows:

```
-- register
"C:\Program Files (x86)\Microsoft SDKs\Windows\v10.0A\bin\NETFX 4.6.1 Tools\
gacutil.exe" -if "E:\Program Files (x86)\Microsoft SQL Server\130\DTS\Tasks\
ExecuteCatalogPackageTask.dll"
```

In Solution Explorer, open My Project and click the Compile page as shown in Figure 4-12.

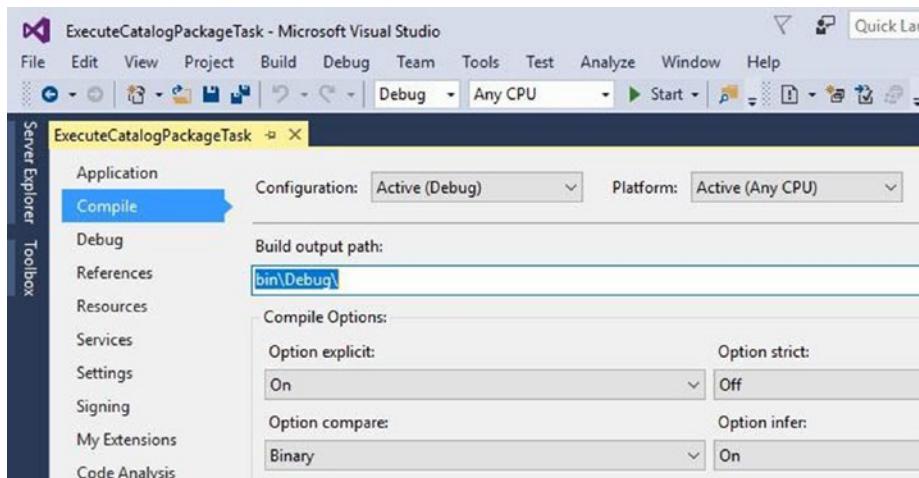


Figure 4-12. Viewing the Compile page

Copy the SSIS Tasks subfolder path into the “Build output path:” text box as shown in Figure 4-13.

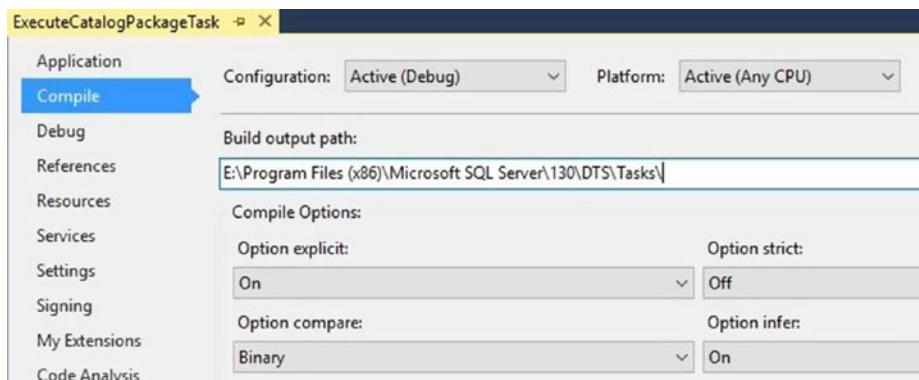


Figure 4-13. Altering the Build output path

With this change, the assembly will be built and delivered to the folder SSDT uses to populate the SSIS Toolbox. Note: this is *not* the location of tasks that are executed at runtime. The runtime executables must be registered in the GAC.

The project now sports a solid foundation and is protected via source control. We now have all the required pieces in place to begin coding our custom task functionality.

CHAPTER 5



Coding the Task

We've worked through several chapters in preparation for this chapter, but we are finally ready to begin coding the custom task in earnest. We start by adding references to the project, decorating the class, inheriting from a base class, and adding a property.

Using a Reference

Open the Visual Studio solution `ExecuteCatalogPackageTask`. In Solution Explorer open the class `ExecuteCatalogPackageTask.vb`. At the very top of the class, add the following line:

```
Imports Microsoft.SqlServer.Dts.Runtime
```

Your class now appears as shown in Figure 5-1.

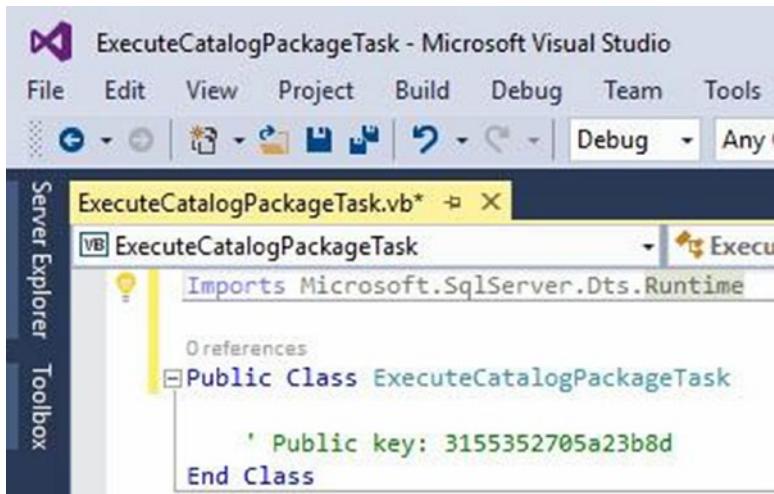


Figure 5-1. Using a reference

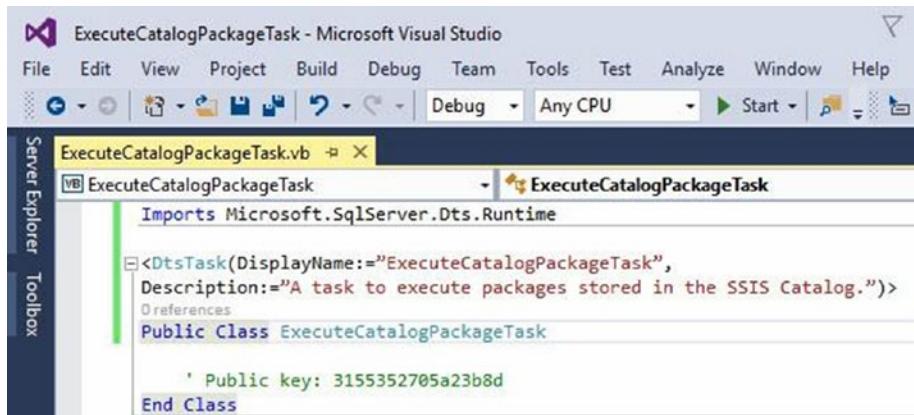
This line of code allows you to use objects, methods, and properties contained in the assembly Microsoft.SqlServer.ManagedDTS—we added a reference to this assembly in Chapter 2.

Decorating the Class

If you are reading this near the holidays, I know what you're thinking: "Andy, the holidays are almost over. Why are we decorating now?" It's not that kind of decorating.

We decorate to inform Visual Studio this class is "different." In this case, the class is different because it is part of an SSIS (SQL Server Integration Services) task. Our decoration code goes just prior to the class definition and is composed of the following lines, as shown in Figure 5-2:

```
<DtsTask(DisplayName:="ExecuteCatalogPackageTask", Description:="A task to execute packages stored in the SSIS Catalog.")> _
```



```
Imports Microsoft.SqlServer.Dts.Runtime

<DtsTask(DisplayName:="ExecuteCatalogPackageTask",
Description:="A task to execute packages stored in the SSIS Catalog.")>
Public Class ExecuteCatalogPackageTask

    ' Public key: 3155352705a23b8d
End Class
```

Figure 5-2. Decorating the class

Visual Studio now knows more about what we're building.

Inheriting from Task

Beneath the class definition place the following line of code:

```
Inherits Task
```

Your class should appear as shown in Figure 5-3.

```

ExecuteCatalogPackageTask - Microsoft Visual Studio
File Edit View Project Build Debug Team Tools Test Analyze Window Help
Debug Any CPU Start
ExecuteCatalogPackageTask.vb* ExecuteCatalogPackageTask
Imports Microsoft.SqlServer.Dts.Runtime

<DtsTask(DisplayName:="ExecuteCatalogPackageTask",
Description:="A task to execute packages stored in the SSIS Catalog.")>
Public Class ExecuteCatalogPackageTask
Inherits Task

' Public key: 3155352705a23b8d
End Class

```

Figure 5-3. Inheriting task

The actions taken so far combine to create a relationship between our code and the *base code* included in the ManagedDTS assembly we referenced and *included (imported)* earlier. Now it's time to build on that relationship by adding the things we want our task to do.

What do we want our task to do? We want to start an SSIS Package that's been deployed to an SSIS Catalog. To accomplish this, we need to provide the name of a SQL Server Instance that hosts an SSIS Catalog, the name of the Catalog Folder that contains the SSIS Project, the name of the SSIS Project that contains the SSIS Package, and the name of the SSIS Package.

Adding a Property

In a class, a property is coded using an internal private variable and publicly accessible property. Inside the class, we will use the private variable. The property provides a mechanism to expose the value of the internal variable to objects outside the class.

Create the internal variable by adding the following lines of code after the Inherits line:

```

Private _serverName As String
Private _packageCatalog As String = "SSISDB"
Private _packageFolder As String
Private _packageProject As String
Private _packageName As String

```

Beginning the name of an internal variable with “_” is a “coding convention,” as is provisioning the variable as private. Your class should now appear as shown in Figure 5-4.

The screenshot shows the Microsoft Visual Studio interface with the title bar "ExecuteCatalogPackageTask - Microsoft Visual Studio". The menu bar includes File, Edit, View, Project, Build, Debug, Team, Tools, Test, Analyze, Window, and Help. The toolbar below the menu bar has icons for back, forward, search, and other common functions. The status bar at the bottom shows "Any CPU" and "Start". The code editor window displays the VB file "ExecuteCatalogPackageTask.vb". The code defines a class "ExecuteCatalogPackageTask" that inherits from "Task". It contains five private string variables: `_serverName`, `_packageCatalog`, `_packageFolder`, `_packageProject`, and `_packageName`. A note in the code indicates a public key: `' Public key: 3155352705a23b8d`.

```
Imports Microsoft.SqlServer.Dts.Runtime

<DtsTask(DisplayName:="ExecuteCatalogPackageTask",
  Description:="A task to execute packages stored in the SSIS Catalog.")>
  References
Public Class ExecuteCatalogPackageTask
  Inherits Task

  Private _serverName As String
  Private _packageCatalog As String = "SSISDB"
  Private _packageFolder As String
  Private _packageProject As String
  Private _packageName As String

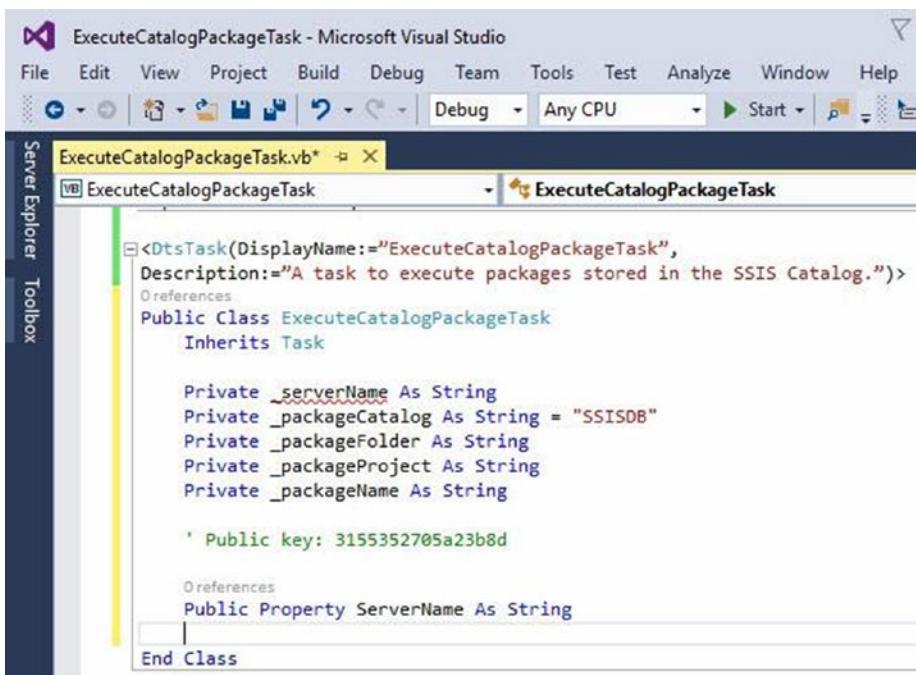
  ' Public key: 3155352705a23b8d
End Class
```

Figure 5-4. Adding private variables

The variables will hold a string values, and we will use the values internally in our task. For example, we will expose the value of `_serverName` externally via a property named `ServerName`. Let's construct that property now by adding the following line of code to the `ExecuteCatalogPackageTask` class:

```
Public Property ServerName As String
```

Your class should now appear as shown in Figure 5-5.



The screenshot shows the Microsoft Visual Studio IDE interface. The title bar reads "ExecuteCatalogPackageTask - Microsoft Visual Studio". The menu bar includes File, Edit, View, Project, Build, Debug, Team, Tools, Test, Analyze, Window, and Help. The toolbar has standard icons for file operations. The status bar shows "Any CPU" and "Start". The left sidebar has "Server Explorer" and "Toolbox" buttons. The main code editor window displays the following VB.NET code:

```
<DtsTask(DisplayName:="ExecuteCatalogPackageTask",  
Description:="A task to execute packages stored in the SSIS Catalog.")>  
Public Class ExecuteCatalogPackageTask  
    Inherits Task  
  
    Private _serverName As String  
    Private _packageCatalog As String = "SSISDB"  
    Private _packageFolder As String  
    Private _packageProject As String  
    Private _packageName As String  
  
    ' Public key: 3155352705a23b8d  
  
    Public Property ServerName As String  
End Class
```

Figure 5-5. Adding a property, part 1

Properties can be read-only, write-only, and subject to many other conditions. Most properties are read-write, like the `ServerName` property we are constructing. To continue coding the property, type “Get” and press Enter on the line beneath the `Property` line. The remaining code for the property is added automatically as shown in Figure 5-6.

```

<DtsTask(DisplayName:="ExecuteCatalogPackageTask",
  Description:="A task to execute packages stored in the SSIS Catalog.")>
  References
  Public Class ExecuteCatalogPackageTask
    Inherits Task

    Private _serverName As String
    Private _packageCatalog As String = "SSISDB"
    Private _packageFolder As String
    Private _packageProject As String
    Private _packageName As String

    ' Public key: 3155352705a23b8d

    References
    Public Property ServerName As String
      Get
        ' End Get
        Set(value As String)
          ' End Set
        End Property
      End Class

```

Figure 5-6. Adding a property, part 2

We couple the value of `_serverName` to the property by returning the value of the variable on read (Get) and writing the value passed to the property on write (Set). The read code looks as follows:

```
Return _serverName
```

The write code looks as follows:

```
_serverName = value
```

When the read and write code is added to the property, it is complete as shown in Figure 5-7.

The screenshot shows the Visual Studio IDE with the code editor open for the file 'ExecuteCatalogPackageTask.vb'. The title bar says 'ExecuteCatalogPackageTask' and the tab bar says 'ExecuteCatalog'. The code is written in Visual Basic and defines a class 'ExecuteCatalogPackageTask' that inherits from 'Task'. It contains private fields for server name, package catalog, package folder, package project, and package name. A public property 'ServerName' is defined with get and set methods. The code editor has syntax highlighting and a vertical scroll bar.

```

Public Class ExecuteCatalogPackageTask
    Inherits Task

    Private _serverName As String
    Private _packageCatalog As String = "SSISDB"
    Private _packageFolder As String
    Private _packageProject As String
    Private _packageName As String

    ' Public key: 3155352705a23b8d

    Public Property ServerName As String
        Get
            Return _serverName
        End Get
        Set(value As String)
            _serverName = value
        End Set
    End Property

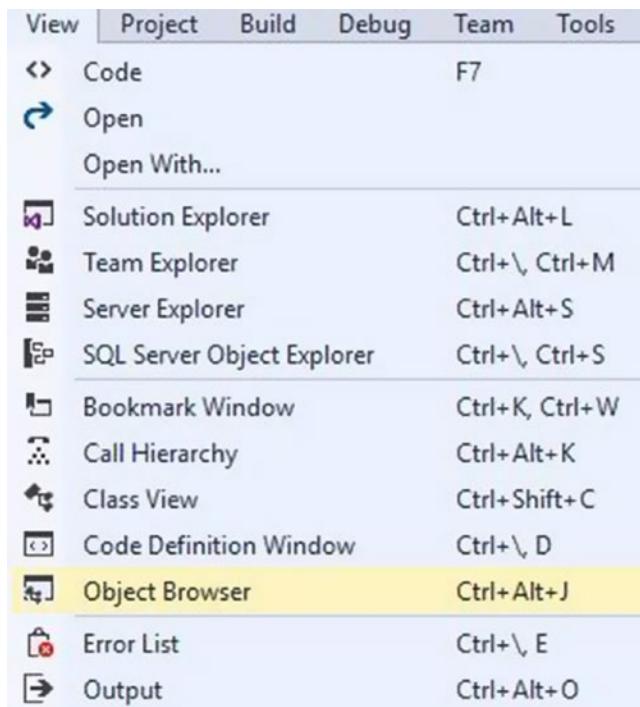
End Class

```

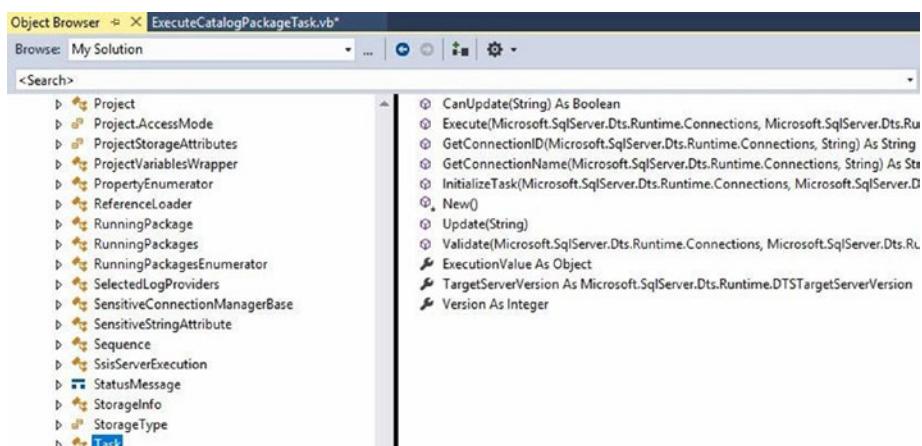
Figure 5-7. Completing the *ServerName* property add similar properties for *PackageCatalog*, *PackageFolder*, *PackageProject*, and *PackageName*

Investigating Task Methods

Earlier we added an `Inherits` statement to the `ExecuteCatalogPackageTask` class—`Inherits Task`, remember? When you inherit one class in another, the inherited class is identified as the Base Class. In Visual Basic, it can be addressed using the keyword `MyBase`. You can observe a bunch of information about a base class by clicking `View > Object Browser` as shown in Figure 5-8.

**Figure 5-8.** Opening Object Browser

Once open, navigate Object Browser to the `Microsoft.SqlServer.ManagedDTS\Microsoft.SqlServer.ManagedDTS .Runtime\Task` base class, as shown in Figure 5-9.

**Figure 5-9.** Viewing the task class in Object Browser

Methods contained in this class are listed on the right. If we select them we get details below as shown in Figure 5-10.

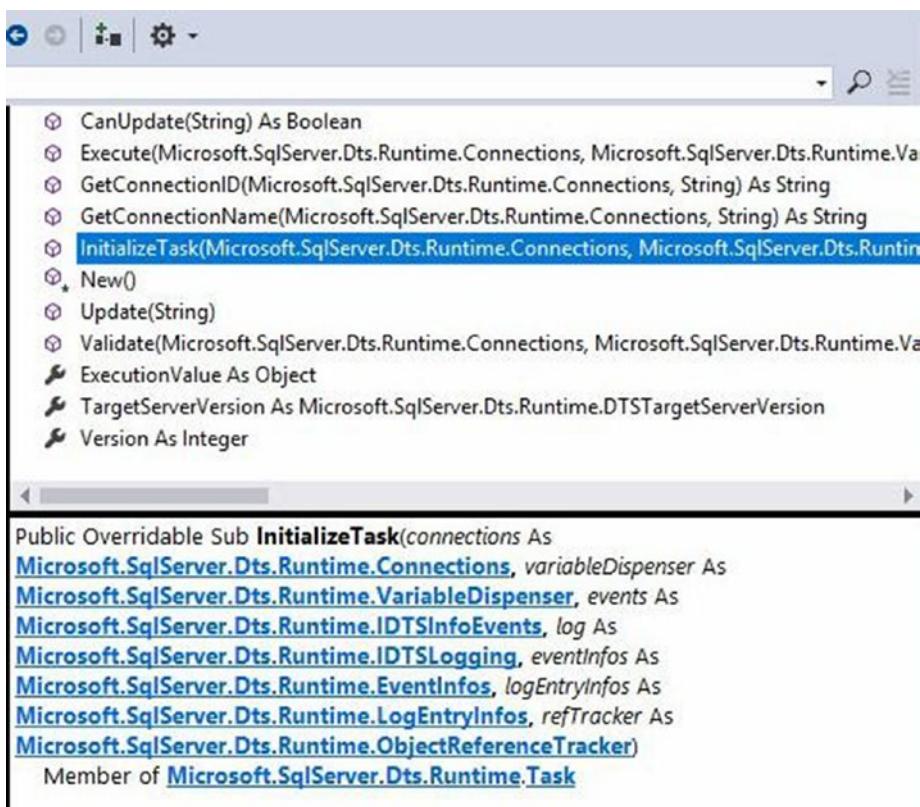


Figure 5-10. Viewing details of the InitializeTask method

Notice the InitializeTask subroutine is marked as “Overridable.” When we inherit the Task class, we can override the InitializeTask method in our inheriting class. We will override this method and two others: Validate and Execute. To override the InitializeTask method add the following code:

```
Public Overrides Sub InitializeTask(ByVal connections As Connections, _
    ByVal variableDispenser As VariableDispenser, _
    ByVal events As IDTSInfoEvents, _
    ByVal log As IDTSLocking, _
    ByVal eventInfos As EventInfos, _
    ByVal logEntryInfos As LogEntryInfos, _
    ByVal refTracker As ObjectReferenceTracker)
    End Sub
```

When you add this code, your code should appear as shown in Figure 5-11.

```

ExecuteCatalogPackageTask.vb*  X
VB ExecuteCatalogPackageTask  ExecuteCatalogPackageTask  InitializeT
    References
    Public Property PackageName As String
        Get
            Return _packageName
        End Get
        Set(value As String)
            _packageName = value
        End Set
    End Property

    References
    Public Overrides Sub InitializeTask(ByVal connections As Connections,
                                         ByVal variableDispenser As VariableDispenser,
                                         ByVal events As IDTSInfoEvents,
                                         ByVal log As IDTSLocking,
                                         ByVal eventInfos As EventInfos,
                                         ByVal logEntryInfos As LogEntryInfos,
                                         ByVal refTracker As ObjectReferenceTracker)

    End Sub

```

Figure 5-11. Overriding InitializeTask

When does the InitializeTask method fire? The InitializeTask method executes when the task is added to an SSIS Control Flow.

Overriding the Validate Method

The Validate method should be inherited and overridden in our class. In this book we will *not* implement validation code in this method. As noted throughout this book, this custom SSIS task is not production-ready code. A production-ready custom SSIS task would definitely implement validation, and that validation code would be implemented here.

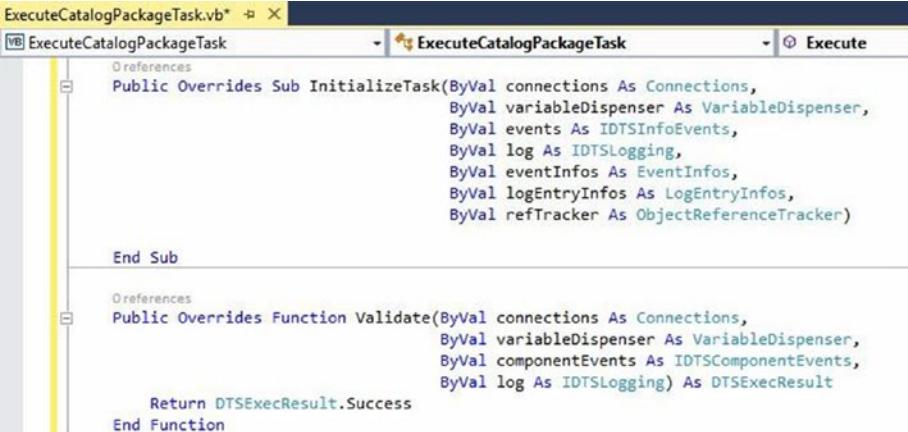
Override the Validate method by adding the following function to our class:

```

Public Overrides Function Validate(ByVal connections As Connections, _
                                  ByVal variableDispenser As VariableDispenser, _
                                  ByVal componentEvents As IDTSCOMPONENTEvents, _
                                  ByVal log As IDTSLocking) As DTSExecResult
    Return DTSExecResult.Success
End Function

```

After adding this code, your class should appear similar to that shown in Figure 5-12.



```

VB ExecuteCatalogPackageTask   ExecuteCatalogPackageTask   Execute
ExecuteCatalogPackageTask
    Public Overrides Sub InitializeTask(ByVal connections As Connections,
                                         ByVal variableDispenser As VariableDispenser,
                                         ByVal events As IDTSInfoEvents,
                                         ByVal log As IDTSLocking,
                                         ByVal eventInfos As EventInfos,
                                         ByVal logEntryInfos As LogEntryInfos,
                                         ByVal refTracker As ObjectReferenceTracker)
    End Sub

    Public Overrides Function Validate(ByVal connections As Connections,
                                      ByVal variableDispenser As VariableDispenser,
                                      ByVal componentEvents As IDTSCOMPONENTEvents,
                                      ByVal log As IDTSLocking) As DTSExecResult
        Return DTSExecResult.Success
    End Function

```

Figure 5-12. Overriding the Validate function

The Validate method is called when the task is added to the SSIS Control Flow, when a property changes, and when a task is executed. The line `Return DTSExecResult.Success` tells the method to return Success—which translates to “I think I am ready to run” in *Task-Speak*—to the Control Flow when the method is invoked.

Overriding the Execute Method

As with the Validate method, we override the inherited Execute method with a bare minimum of functionality. Again, this is not production-ready code.

Add the Execute method override by adding the following code to the `ExecuteCatalogPackageTask` class:

```

Public Overrides Function Execute(ByVal connections As Connections,
                                 ByVal variableDispenser As VariableDispenser,
                                 ByVal componentEvents As IDTSCOMPONENTEvents,
                                 ByVal log As IDTSLocking,
                                 ByVal transaction As Object) As DTSExecResult
    Return DTSExecResult.Success
End Function

```

Your class should resemble that shown in Figure 5-13.

```
ExecuteCatalogPackageTask.vb*  X  ExecuteCatalogPackageTask  Execute
Public Overrides Sub InitializeTask(ByVal connections As Connections,
    ByVal variableDispenser As VariableDispenser,
    ByVal events As IDTSInfoEvents,
    ByVal log As IDTSLocking,
    ByVal eventInfos As EventInfos,
    ByVal logEntryInfos As LogEntryInfos,
    ByVal refTracker As ObjectReferenceTracker)
End Sub

Public Overrides Function Validate(ByVal connections As Connections,
    ByVal variableDispenser As VariableDispenser,
    ByVal componentEvents As IDTSCOMPONENTEvents,
    ByVal log As IDTSLocking) As DTSExecResult
    Return DTSExecResult.Success
End Function

Public Overrides Function Execute(ByVal connections As Connections,
    ByVal variableDispenser As VariableDispenser,
    ByVal componentEvents As IDTSCOMPONENTEvents,
    ByVal log As IDTSLocking,
    ByVal transaction As Object) As DTSExecResult
    Return DTSExecResult.Success
End Function
```

Figure 5-13. Overriding Execute

That is all we are going to code in this task. Trust me, there are a lot of other things you want to consider if you're building a commercially viable product. But this series, as involved as it is, is merely an introduction.

CHAPTER 6



Coding the Task Editor

An SSIS (SQL Server Integration Services) task needs an editor. We begin creating the editor for ExecuteCatalogPackageTask by adding a project to the ExecuteCatalogPackageTask solution.

Adding a Task Editor

Open the Visual Studio solution ExecuteCatalogPackageTask.sln. In Solution Explorer right-click the solution, hover over Add, and click New Project as shown in Figure 6-1.

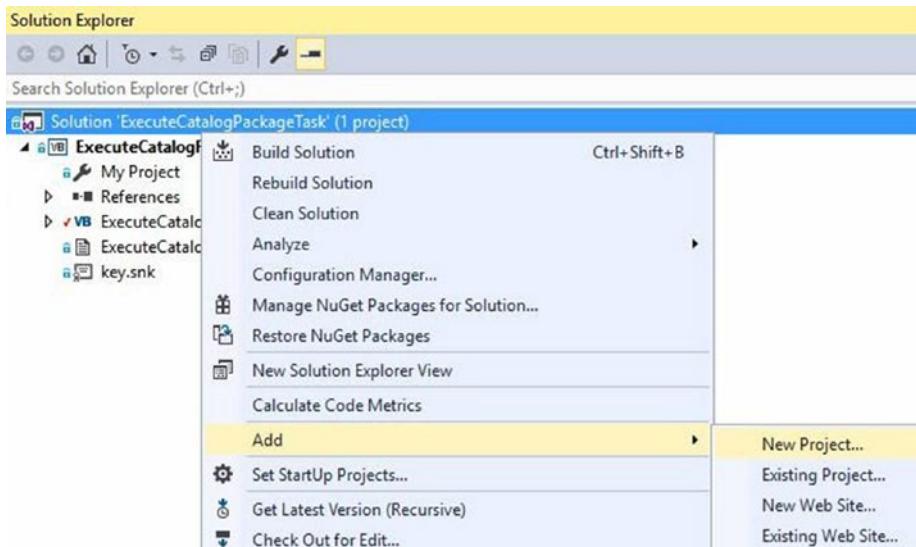


Figure 6-1. Adding a new project

Select a Visual Basic Windows Class Library and name it ExecuteCatalogPackageTaskUI as shown in Figure 6-2.

CHAPTER 6 ■ CODING THE TASK EDITOR

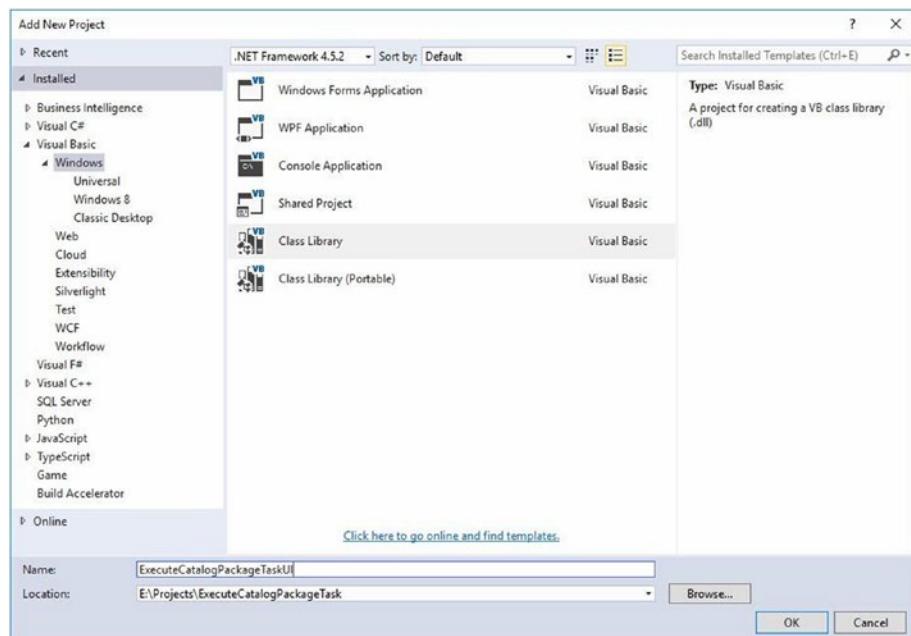


Figure 6-2. Selecting and naming the project template

Rename Class1.vb to ExecuteCatalogPackageTaskUI.vb (shown in Figure 6-3).

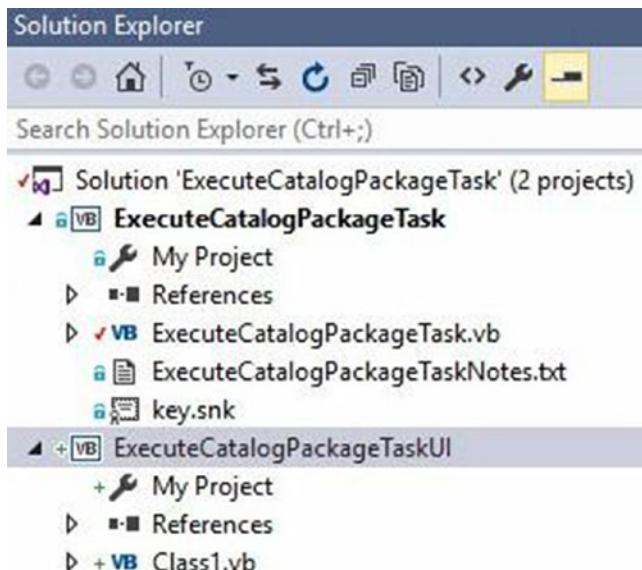


Figure 6-3. Renaming ExecuteCatalogPackageTaskUI

When prompted, click Yes to rename all Class1 references to ExecuteCatalogPackageTaskUI references as shown in Figure 6-4.

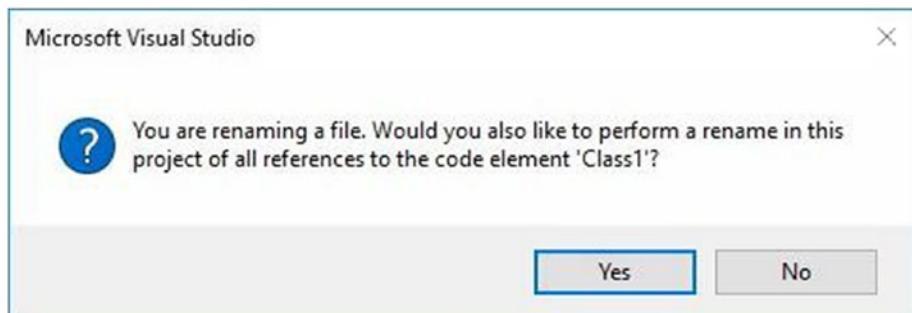


Figure 6-4. Confirming the Class1 rename

Adding References

As with the task project, we need to add references to this project. In Solution Explorer, right-click ExecuteCatalogPackageTaskUI and click Add Reference. When the Reference Manager displays, expand Assemblies and click Extensions. Scroll until you find Microsoft.SqlServer.Dts.Design and Microsoft.SqlServer.ManagedDTS, and select them as shown in Figure 6-5.

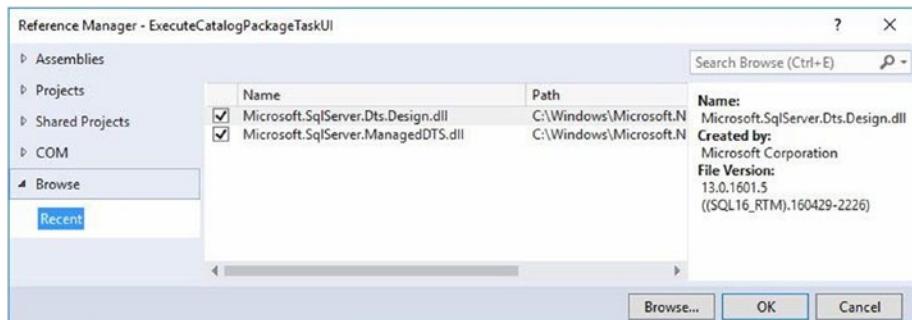


Figure 6-5. Adding references

To use the new references, open the class named ExecuteCatalogPackageTaskUI.vb. At the very top of this class, add the following Imports statements:

```
Imports System
Imports System.Windows.Forms
Imports Microsoft.SqlServer.Dts.Runtime
Imports Microsoft.SqlServer.Dts.Runtime.Design
```

ExecuteCatalogPackageTaskUI.vb should now appear as shown in Figure 6-6.

```

ExecuteCatalogPackageTaskUI.vb*  X
VB ExecuteCatalogPackageTaskUI - ExecuteCatalogPackageTaskUI
Imports System
Imports System.Windows.Forms
Imports Microsoft.SqlServer.Dts.Runtime
Imports Microsoft.SqlServer.Dts.Runtime.Design

Public Class ExecuteCatalogPackageTaskUI

End Class

```

Figure 6-6. Importing .Net assemblies

Some Implementation

Next, we implement the IDtsTaskUI interface with the following statement added just after the ExecuteCatalogPackageTaskUI class declaration:

Implements IDtsTaskUI

Your class will appear as shown in Figure 6-7:

```

ExecuteCatalogPackageTaskUI.vb*  X
VB ExecuteCatalogPackageTaskUI - ExecuteCatalogPackageTaskUI
Imports System
Imports System.Windows.Forms
Imports Microsoft.SqlServer.Dts.Runtime
Imports Microsoft.SqlServer.Dts.Runtime.Design

Public Class ExecuteCatalogPackageTaskUI
    Implements IDtsTaskUI

End Class

```

Figure 6-7. Implementing IDtsTaskUI

The squiggly line beneath the interface name (IDtsTaskUI is a .Net interface) informs us of an issue. Hovering over the squiggly line causes a tooltip to display as shown in Figure 6-8.

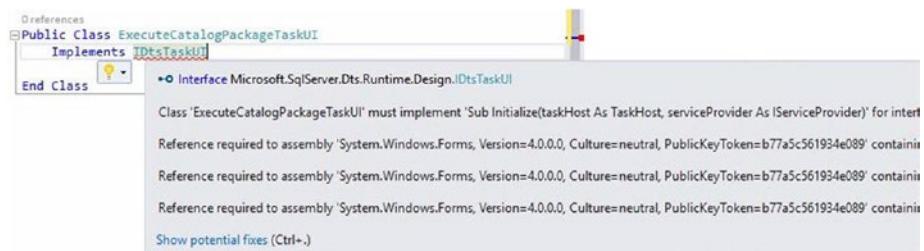


Figure 6-8. Interface implementation requirements

This message informs us we need to implement an Initialize subroutine to make ExecuteCatalogPackageTaskUI compliant to the IDtsTaskUI interface we are implementing. The Error List (View ► Error List) displays additional subroutines and one function required for implementing the IDtsTaskUI interface, shown in Figure 6-9.

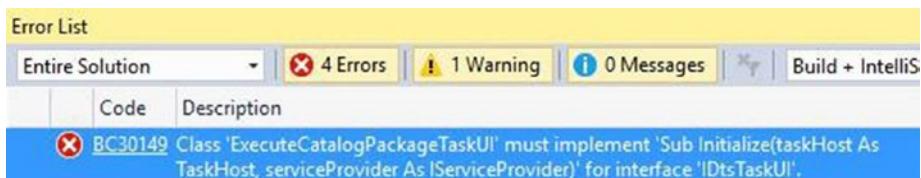
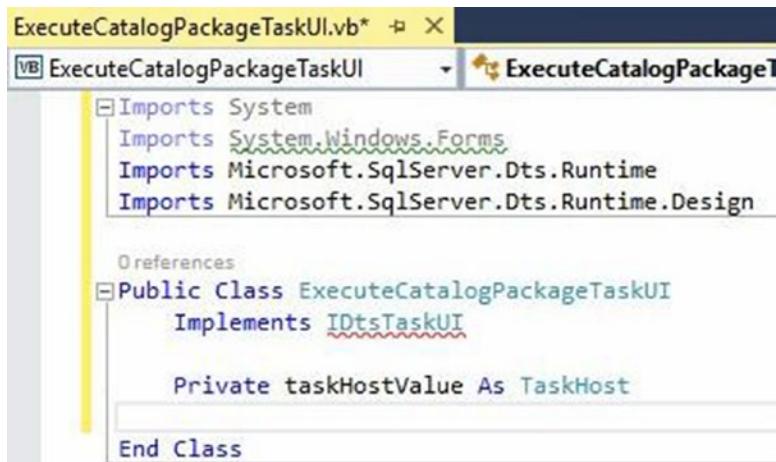


Figure 6-9. Error List showing method required for IDtsTaskUI implementation

We will need a TaskHost object variable to implement some of this functionality, so let's add a new TaskHost variable just after the Implements statement as shown in Figure 6-10.

```
Private taskHostValue As TaskHost
```



```

ExecuteCatalogPackageTaskUI.vb*  X
VB ExecuteCatalogPackageTaskUI  ExecuteCatalogPackageT
Imports System
Imports System.Windows.Forms
Imports Microsoft.SqlServer.Dts.Runtime
Imports Microsoft.SqlServer.Dts.Runtime.Design

0 references
Public Class ExecuteCatalogPackageTaskUI
    Implements IDtsTaskUI

    Private taskHostValue As TaskHost

End Class

```

Figure 6-10. Adding a TaskHost variable

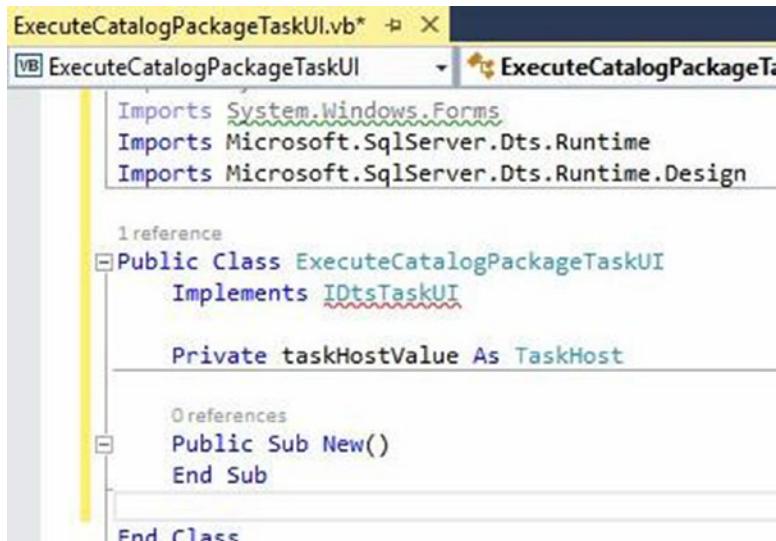
Also, just because it's a good idea to have one, let's add a generic constructor (New method) for our class by adding the following code after the TaskHost variable declaration:

```

Public Sub New()
End Sub

```

Your class should now appear as shown in Figure 6-11.



```

ExecuteCatalogPackageTaskUI.vb*  X
VB ExecuteCatalogPackageTaskUI  ExecuteCatalogPackageT
Imports System.Windows.Forms
Imports Microsoft.SqlServer.Dts.Runtime
Imports Microsoft.SqlServer.Dts.Runtime.Design

1 reference
Public Class ExecuteCatalogPackageTaskUI
    Implements IDtsTaskUI

    Private taskHostValue As TaskHost

    Public Sub New()
    End Sub

End Class

```

Figure 6-11. Adding a generic constructor

Let's implement the `GetView` method of the `IDtsTaskUI` interface by adding the following code:

```
Function GetView() As System.Windows.Forms.ContainerControl Implements
IDtsTaskUI.GetView
Return New ExecuteCatalogPackageTaskForm(Me.taskHostValue)
End Function
```

Your class should now appear as shown in Figure 6-12.

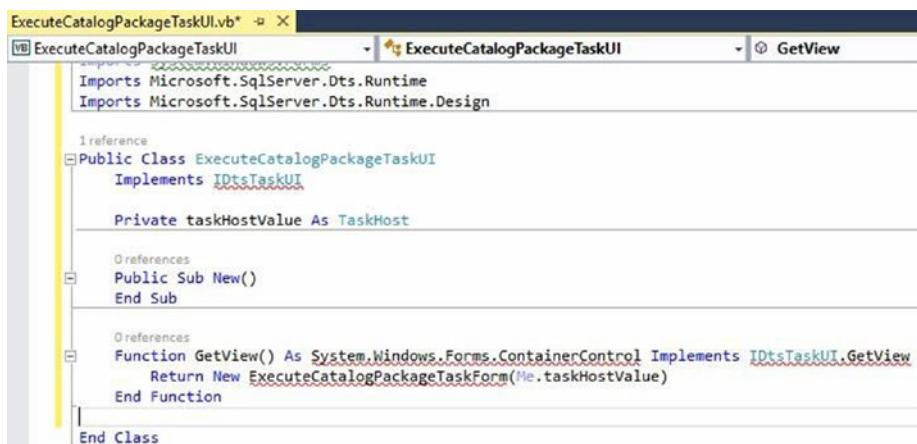


Figure 6-12. Implementing `GetView`

The squiggly lines indicate that we need to add that there are no constructors (`New` methods) for the `ExecuteCatalogPackageTaskForm` that accept arguments. We will address this issue in the near future.

Next, let's implement another `New` method—this one implementing the `New` method required by the `IDtsTaskUI` interface—by adding the following code:

```
Sub [New](ByVal parentWindow As IWin32Window) Implements IDtsTaskUI.New
End Sub
```

Your class should now appear as shown in Figure 6-13.

CHAPTER 6 ■ CODING THE TASK EDITOR

The screenshot shows the Microsoft Visual Studio IDE with the code editor open. The file is named "ExecuteCatalogPackageTaskUI.vb". The code implements the `IDtsTaskUI` interface, specifically the `New` method. The code includes imports for `Microsoft.SqlServer.Dts.Runtime` and `Microsoft.SqlServer.Dts.Runtime.Design`. It defines a class `ExecuteCatalogPackageTaskUI` that implements `IDtsTaskUI`. The `New` method returns a new instance of `ExecuteCatalogPackageTaskForm` with the value of `taskHostValue`.

```
ExecuteCatalogPackageTaskUI.vb*  X  ExecuteCatalogPackageTaskUI  New
ExecuteCatalogPackageTaskUI
Imports Microsoft.SqlServer.Dts.Runtime
Imports Microsoft.SqlServer.Dts.Runtime.Design

1 reference
Public Class ExecuteCatalogPackageTaskUI
    Implements IDtsTaskUI

    Private taskHostValue As TaskHost

    0 references
    Public Sub New()
    End Sub

    0 references
    Function GetView() As System.Windows.Forms.ContainerControl Implements IDtsTaskUI.GetView
        Return New ExecuteCatalogPackageTaskForm(Me.taskHostValue)
    End Function

    0 references
    Sub [New](ByVal parentWindow As IWin32Window) Implements IDtsTaskUI.New
    End Sub

End Class
```

Figure 6-13. Implementing `IDtsTaskUI.New`

Let's next add code to implement the `Initialize` method by adding the following to your class:

```
Sub Initialize(ByVal taskHost As TaskHost, ByVal serviceProvider As
IServiceProvider) Implements IDtsTaskUI.Initialize
Me.taskHostValue = taskHost
End Sub
```

Your class should now appear as shown in Figure 6-14.

The screenshot shows the Microsoft Visual Studio IDE with the code editor open. The file is named "ExecuteCatalogPackageTaskUI.vb". The code implements the `IDtsTaskUI` interface, specifically the `Initialize` method. The `Initialize` method takes `taskHost` and `serviceProvider` parameters and assigns `taskHostValue` to `taskHost`.

```
ExecuteCatalogPackageTaskUI.vb*  X  ExecuteCatalogPackageTaskUI  Initialize
ExecuteCatalogPackageTaskUI
Imports Microsoft.SqlServer.Dts.Runtime.Design

1 reference
Public Class ExecuteCatalogPackageTaskUI
    Implements IDtsTaskUI

    Private taskHostValue As TaskHost

    0 references
    Public Sub New()
    End Sub

    0 references
    Function GetView() As System.Windows.Forms.ContainerControl Implements IDtsTaskUI.GetView
        Return New ExecuteCatalogPackageTaskForm(Me.taskHostValue)
    End Function

    0 references
    Sub [New](ByVal parentWindow As IWin32Window) Implements IDtsTaskUI.New
    End Sub

    1 reference
    Sub Initialize(ByVal taskHost As TaskHost, ByVal serviceProvider As IServiceProvider) Implements IDtsTaskUI.Initialize
        Me.taskHostValue = taskHost
    End Sub

End Class
```

Figure 6-14. Implementing `Initialize`

Finally, let's implement the Delete method by adding the following code:

```
Sub Delete(ByVal parentWindow As IWin32Window) Implements IDtsTaskUI.Delete
End Sub
```

Your class should now appear as shown in Figure 6-15.

```
ExecuteCatalogPackageTaskUI.vb*  X  ExecuteCatalogPackageTaskUI  Delete
Public Class ExecuteCatalogPackageTaskUI
    Implements IDtsTaskUI

    Private taskHostValue As TaskHost

    Public Sub New()
    End Sub

    Function GetView() As System.Windows.Forms.ContainerControl Implements IDtsTaskUI.GetView
        Return New ExecuteCatalogPackageTaskForm(Me.taskHostValue)
    End Function

    Sub [New](ByVal parentWindow As IWin32Window) Implements IDtsTaskUI.New
    End Sub

    Sub Initialize(ByVal taskHost As TaskHost, ByVal serviceProvider AsIServiceProvider) Implements IDtsTaskUI.Initialize
        Me.taskHostValue = taskHost
    End Sub

    Sub Delete(ByVal parentWindow As IWin32Window) Implements IDtsTaskUI.Delete
    End Sub
End Class
```

Figure 6-15. Implementing `IDtsTaskUI.Delete`

Note the squiggly lines beneath `GetView()`. We will resolve this error as we continue coding `ExecuteCatalogPackageTaskUI`. Promise.

We have completed building the `ExecuteCatalogPackageTaskUI` class. Save the class before proceeding and remember to regularly check your code into Source Control.

Adding a Form

To add a form to our User Interface (Task Editor), right-click `ExecuteCatalogPackageTaskUI`, hover over Add, and then click Windows Form (shown in Figure 6-16).

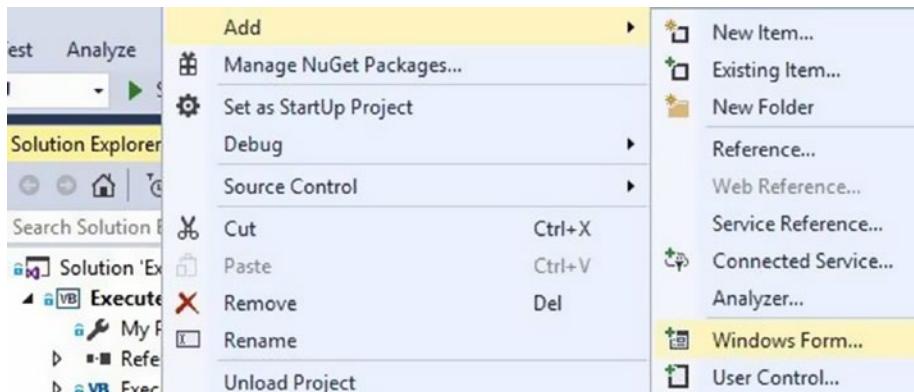


Figure 6-16. Preparing to add a form

When the Add New Item... window displays, name the form ExecuteCatalogPackageTaskForm as shown in Figure 6-17.

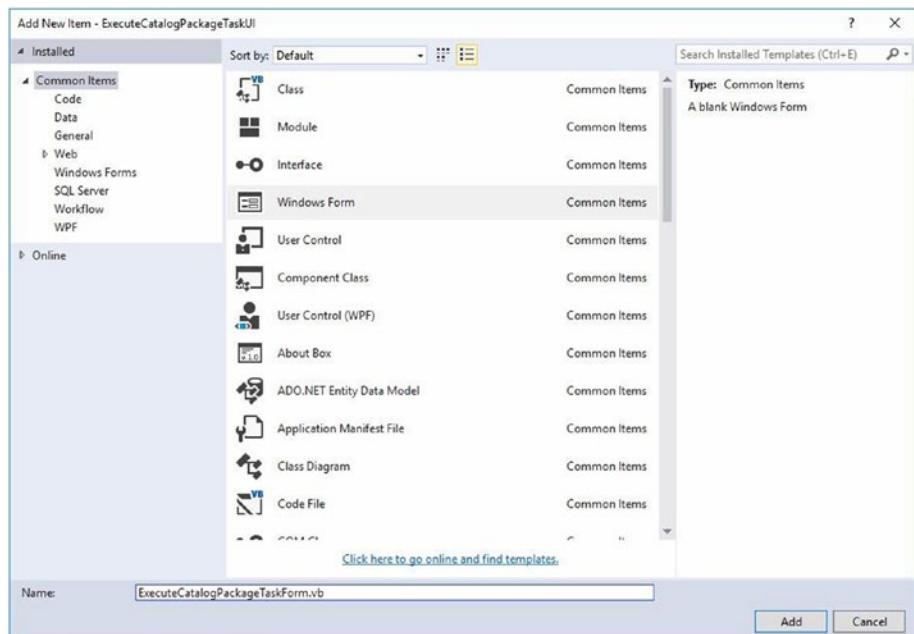


Figure 6-17. Adding a new form

Add a label and a text box to the form. Change the text of the label to “Instance:” and name the text box “txtInstance” as shown in Figure 6-18.

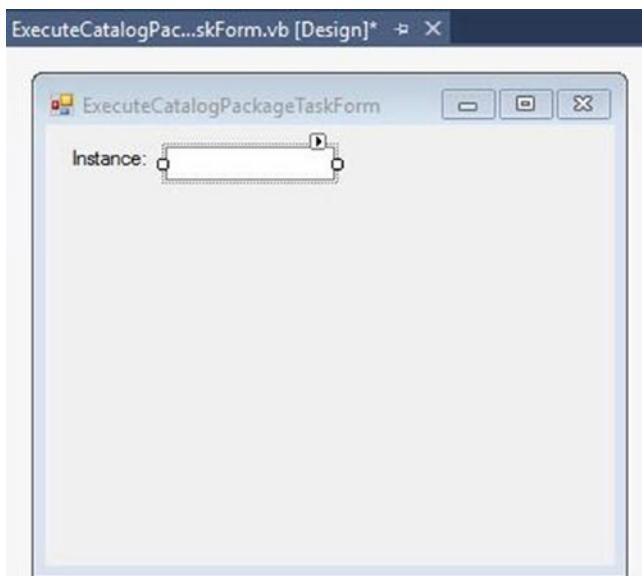


Figure 6-18. Adding the label and text box

Add label and text box controls for the Folder, Project, and Package properties as shown in Figure 6-19.

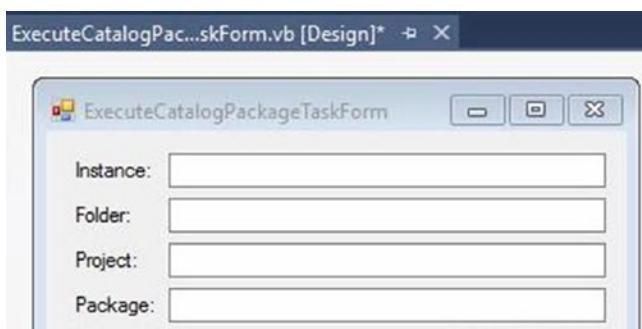


Figure 6-19. Adding controls for Folder, Project, and Package properties

Next, add a button named btnDone and set the Text property to “Done” as shown in Figure 6-20.

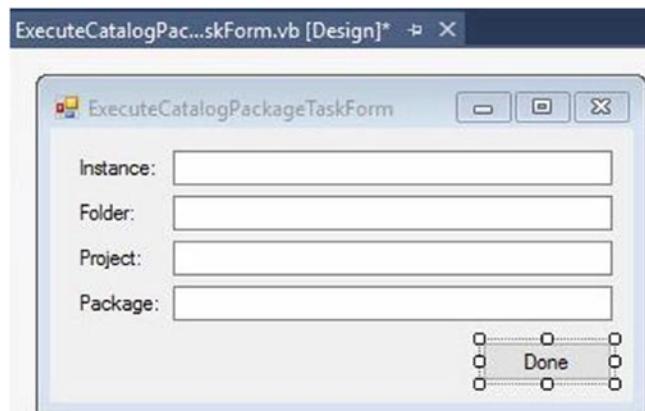


Figure 6-20. Adding the Done button

Finally, change the DialogResult property of btnDone to “OK” as shown in Figure 6-21.

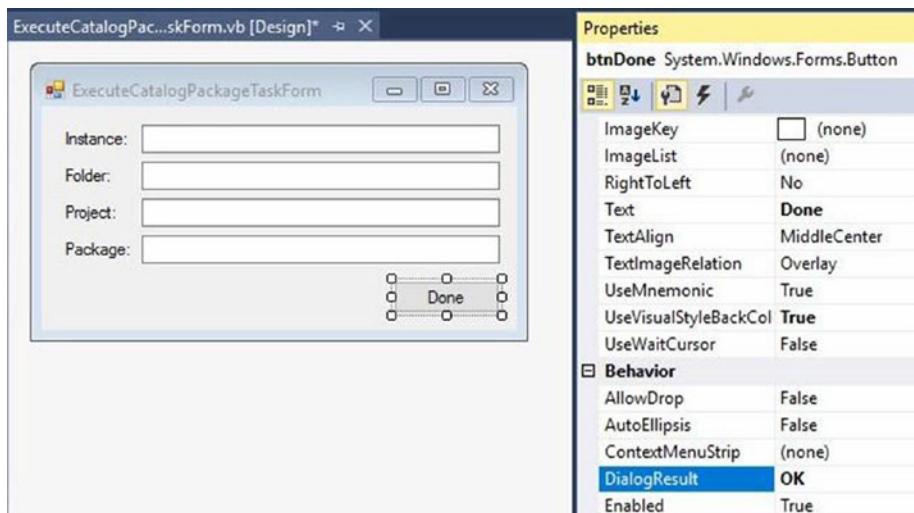


Figure 6-21. Changing the DialogResult property

We will use the button's Click event to set the Catalog Package Path properties (Instance, Folder, Project, and Package) of ExecuteCatalogPackageTask. Setting the DialogResult property on the OK or Done button is important. It accomplishes a couple of automagic functions:

- It closes the editor form.
- Sending OK as the DialogResult triggers the Validate method of the Task.

Coding the Click Event and the Form

I deliberately choose to begin the discussion of coding the form in the Click event of btnDone. I can hear you thinking, “Why, Andy?” I’m glad you asked. The two bullets above represent a “gotcha”—an easy place to fail when coding a custom SSIS task editor. If you build this incorrectly, you will be hard-pressed to find the answer by searching for “Validate method doesn’t fire” for custom SSIS task.

We’ll get to the actual code for the Click event in a bit. First, let’s build out the ExecuteCatalogPackageTaskForm class. Begin by double-clicking btnDone to open the editor at the Click event.

Before we add code directly to the Click event of btnDone, add the following Imports statements to the very top of the Form class:

```
Imports System
Imports System.Globalization
Imports System.Collections
Imports System.ComponentModel
Imports System.Windows.Forms
Imports Microsoft.SqlServer.Dts.Runtime
```

Your form code should now appear as shown in Figure 6-22.

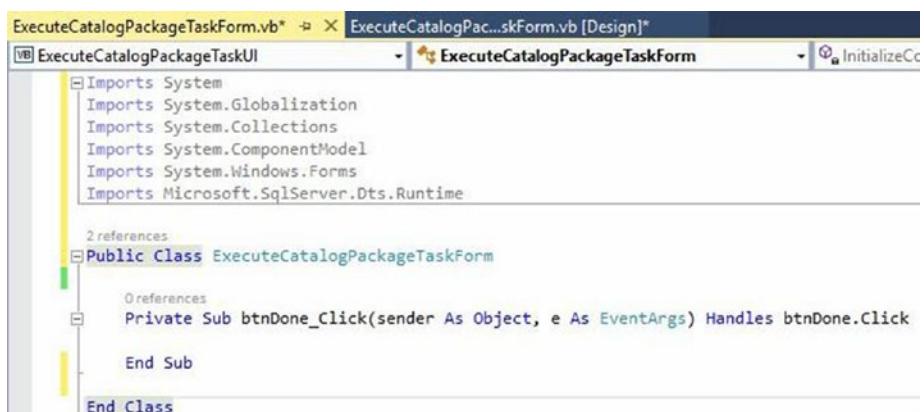
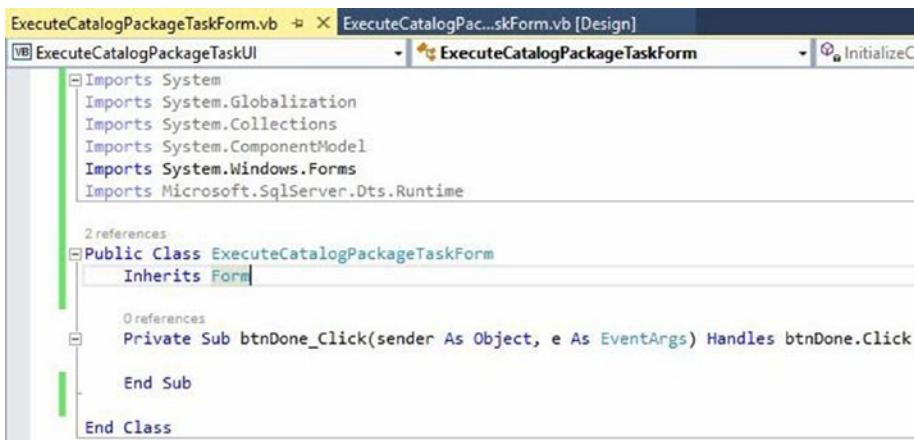


Figure 6-22. Adding the Imports statements to the ExecuteCatalogPackageTaskForm class

Next, add an Inherits statement just after the ExecuteCatalogPackageTaskForm class declaration:

Inherits Form

Your form class should now appear as shown in Figure 6-23.



```
ExecuteCatalogPackageTaskForm.vb  X  ExecuteCatalogPac...skForm.vb [Design]
VB ExecuteCatalogPackageTaskUI  ExecuteCatalogPackageTaskForm  InitializeC
Imports System
Imports System.Globalization
Imports System.Collections
Imports System.ComponentModel
Imports System.Windows.Forms
Imports Microsoft.SqlServer.Dts.Runtime

2 references
Public Class ExecuteCatalogPackageTaskForm
    Inherits Form

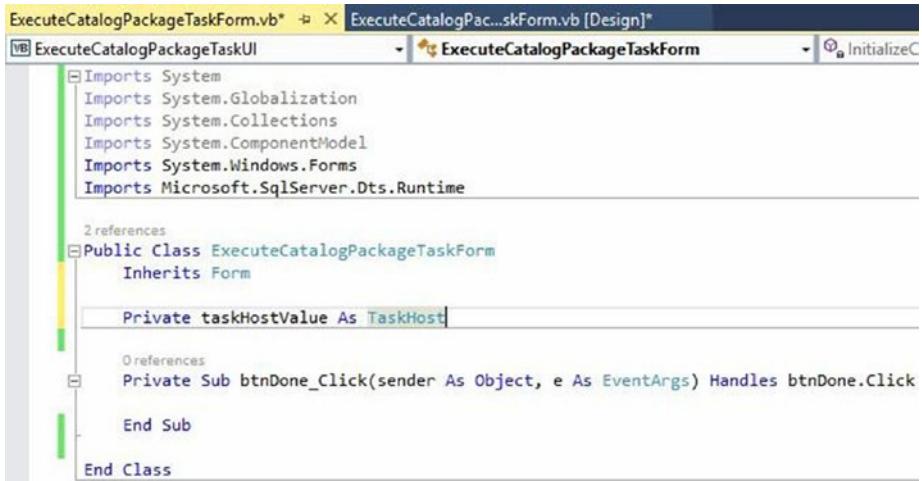
    0 references
    Private Sub btnDone_Click(sender As Object, e As EventArgs) Handles btnDone.Click
        End Sub
    End Class
```

Figure 6-23. Inheriting the form class

Declare a new TaskHost variable named TaskHostValue by adding the following code:

```
Private taskHostValue As TaskHost
```

Your form class should now appear as shown in Figure 6-24.



```
ExecuteCatalogPackageTaskForm.vb*  X ExecuteCatalogPac...skForm.vb [Design]*

VB ExecuteCatalogPackageTaskUI  - ExecuteCatalogPackageTaskForm  InitializeC

Imports System
Imports System.Globalization
Imports System.Collections
Imports System.ComponentModel
Imports System.Windows.Forms
Imports Microsoft.SqlServer.Dts.Runtime

Public Class ExecuteCatalogPackageTaskForm
    Inherits Form

    Private taskHostValue As TaskHost

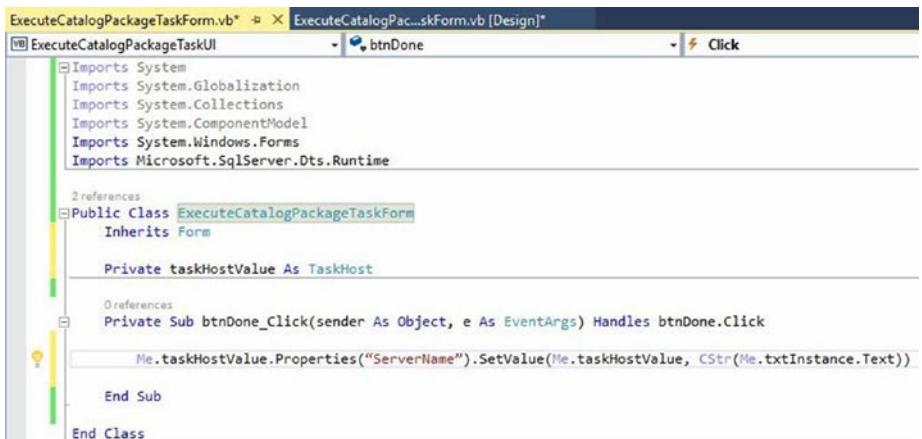
    Private Sub btnDone_Click(sender As Object, e As EventArgs) Handles btnDone.Click
        End Sub
    End Class
```

Figure 6-24. Declaring the TaskHost variable

To begin coding the Click event, we need to address the task class *via* a class hosting the SSIS custom task class:

```
Me.taskHostValue.Properties("ServerName").SetValue(Me.taskHostValue,
CStr(Me.txtInstance.Text))
```

Your form class should now appear as shown in Figure 6-25.



```
ExecuteCatalogPackageTaskForm.vb*  X ExecuteCatalogPac...skForm.vb [Design]*

VB ExecuteCatalogPackageTaskUI  - ExecuteCatalogPackageTaskForm  Click

Imports System
Imports System.Globalization
Imports System.Collections
Imports System.ComponentModel
Imports System.Windows.Forms
Imports Microsoft.SqlServer.Dts.Runtime

Public Class ExecuteCatalogPackageTaskForm
    Inherits Form

    Private taskHostValue As TaskHost

    Private Sub btnDone_Click(sender As Object, e As EventArgs) Handles btnDone.Click
        Me.taskHostValue.Properties("ServerName").SetValue(Me.taskHostValue, CStr(Me.txtInstance.Text))
        End Sub
    End Class
```

Figure 6-25. Building the Click event

In this subroutine—which fires when a data integration developer clicks the Done button on the ExecuteCatalogPackageTask editor—the `ServerName` property of the `TaskHost` that is coupled to this form is set to the text property value of the form text box named `txtInstance`. We built a lot of code to get here, but this is why we did it; so we could edit a property from a task editor.

Before proceeding, add control-to-property mappings for the `PackageFolder`, `PackageProject`, and `PackageName` properties:

```
Me.taskHostValue.Properties("PackageFolder").SetValue(Me.taskHostValue,
CStr(Me.txtFolder.Text))
Me.taskHostValue.Properties("PackageProject").SetValue(Me.taskHostValue,
CStr(Me.txtProject.Text))
Me.taskHostValue.Properties("PackageName").SetValue(Me.taskHostValue,
CStr(Me.txtPackage.Text))
```

Once you've added this code to the Click event subroutine, your form class should appear as shown in Figure 6-26.

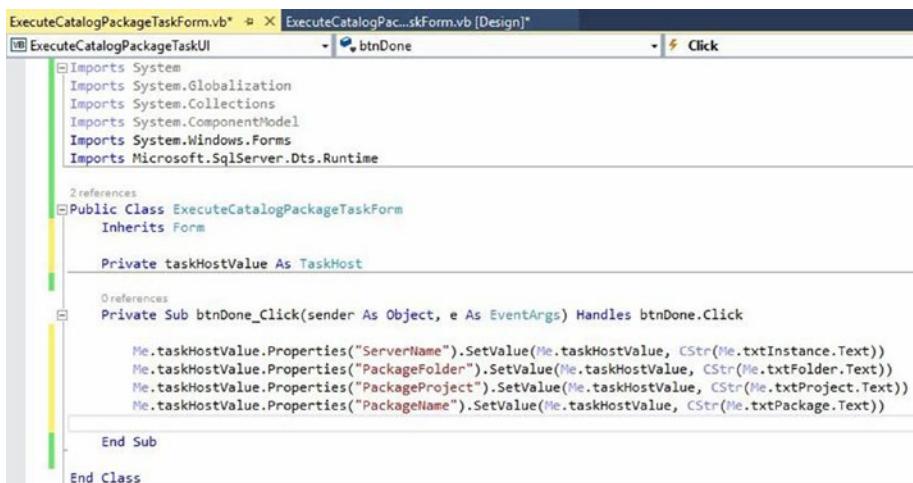


Figure 6-26. Completing the Click event coding

The next step is the complement of the previous coding. When the form displays we want to retrieve the values of the task object's properties (`ServerName`, `PackageFolder`, `PackageProject`, and `PackageName`) and display them in the `txtInstance`, `txtFolder`, `txtProject`, and `txtPackage` text boxes on the editor (form). To accomplish this functionality, add the following constructor to the `ExecuteCatalogPackageTaskForm`:

```

Public Sub New(ByVal taskHost As TaskHost)
InitializeComponent()
Me.taskHostValue = taskHost
Me.txtInstance.Text = CStr(Me.taskHostValue.Properties("ServerName").
GetValue(Me.taskHostValue))
Me.txtFolder.Text = CStr(Me.taskHostValue.Properties("PackageFolder").
GetValue(Me.taskHostValue))
Me.txtProject.Text = CStr(Me.taskHostValue.Properties("PackageProject").
GetValue(Me.taskHostValue))
Me.txtPackage.Text = CStr(Me.taskHostValue.Properties("PackageName").
GetValue(Me.taskHostValue))
End Sub

```

When done, your form class should appear as shown in Figure 6-27.

```

ExecuteCatalogPackageTaskForm.vb  ExecuteCatalogPac...skForm.vb [Design]
ExecuteCatalogPackageTaskUI  ExecuteCatalogPackageTaskForm  New
Public Class ExecuteCatalogPackageTaskForm
    Inherits Form

    Private taskHostValue As TaskHost

    Private Sub btnDone_Click(sender As Object, e As EventArgs) Handles btnDone.Click
        Me.taskHostValue.Properties("ServerName").SetValue(Me.taskHostValue, CStr(Me.txtInstance.Text))
        Me.taskHostValue.Properties("PackageFolder").SetValue(Me.taskHostValue, CStr(Me.txtFolder.Text))
        Me.taskHostValue.Properties("PackageProject").SetValue(Me.taskHostValue, CStr(Me.txtProject.Text))
        Me.taskHostValue.Properties("PackageName").SetValue(Me.taskHostValue, CStr(Me.txtPackage.Text))
    End Sub

    Public Sub New(ByVal taskHost As TaskHost)
        InitializeComponent()
        Me.taskHostValue = taskHost
        Me.txtInstance.Text = CStr(Me.taskHostValue.Properties("ServerName").GetValue(Me.taskHostValue))
        Me.txtFolder.Text = CStr(Me.taskHostValue.Properties("PackageFolder").GetValue(Me.taskHostValue))
        Me.txtProject.Text = CStr(Me.taskHostValue.Properties("PackageProject").GetValue(Me.taskHostValue))
        Me.txtPackage.Text = CStr(Me.taskHostValue.Properties("PackageName").GetValue(Me.taskHostValue))
    End Sub
End Class

```

Figure 6-27. Adding the form constructor

The form constructor takes a TaskHost argument named taskHost, and this clears the error in the code in the ExecuteCatalogPackageTaskUI class's GetView function as shown in Figure 6-28.

```
1 reference
Function GetView() As System.Windows.Forms.ContainerControl Implements IDtsTaskUI.GetView
    Return New ExecuteCatalogPackageTaskForm(Me.taskHostValue)
End Function
```

Figure 6-28. No more squiggly line

Next, we put together the task and editor! In this chapter we developed an editor for the custom SSIS task. We added multiple controls to a form and prepared the form for binding the editor to the custom SSIS task object. Now it's time (in Chapter 7) to build the task and the editor for the task.

CHAPTER 7



Signing and Binding

We began this series 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 Basic 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 (SQL Server Integration Services) task. We coded the task and its editor. And that brings us to 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.

Signing the Task Editor Project

We haven’t yet signed the Task Editor project. Let’s do that now. Double-click My Project under the ExecuteCatalogPackageTaskUI project to open the project properties. Click the Signing page, check the “Sign the assembly” check box, click the “Choose a strong name key file” drop-down, click Browse, browse to key.snk in the ExecuteCatalogPackageTask project folder, and select that file as shown in Figure 7-1.

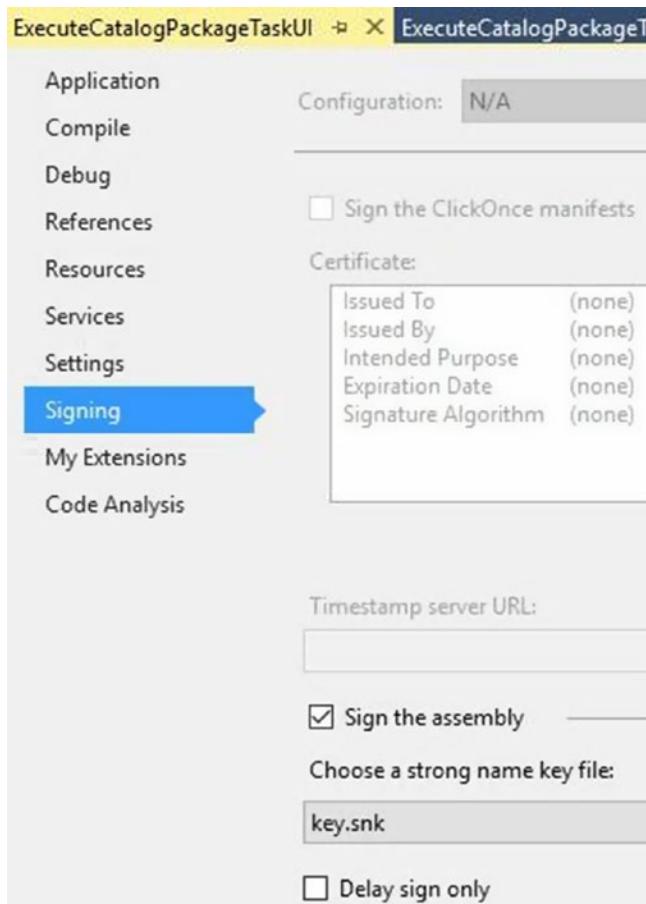


Figure 7-1. Signing the UI project

Click the Compile page and set the Build output path to <drive>:\Program Files (x86)\Microsoft SQL Server\130\DTs\Tasks where <drive> represents the installation drive for SQL Server as seen in Figure 7-2.

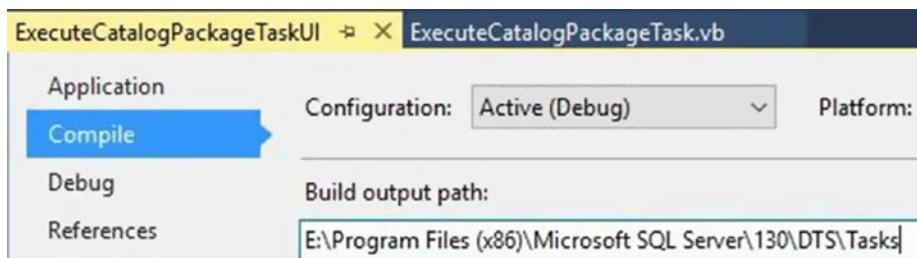


Figure 7-2. Setting the build path for ExecuteCatalogPackageTaskUI

Save the My Project file and close it.

Earlier, we added a text file to ExecuteCatalogPackageTask called ExecuteCatalogPackageTaskNotes.txt. Open that file now and copy and paste the line beneath the “-register” heading. Edit the new line so ExecuteCatalogPackageTask.dll now reads ExecuteCatalogPackageTaskUI.dll as shown in Figure 7-3.

```
3.0A\bin\NETFX 4.6.1 Tools\gacutil.exe" -if "E:\Program Files (x86)\Microsoft SQL Server\130\DTS\Tasks\ExecuteCatalogPackageTask.dll"
3.0A\bin\NETFX 4.6.1 Tools\gacutil.exe" -if "E:\Program Files (x86)\Microsoft SQL Server\130\DTS\Tasks\ExecuteCatalogPackageTaskUI.dll"
```

Figure 7-3. Adding a new registration command

Similarly, copy, paste, and edit the “-unregister” command as shown in Figure 7-4.

```
3.0A\bin\NETFX 4.6.1 Tools\gacutil.exe" -u ExecuteCatalogPackageTask
3.0A\bin\NETFX 4.6.1 Tools\gacutil.exe" -u ExecuteCatalogPackageTaskUI
```

Figure 7-4. Adding a new unregistration command

Reviewing the Public Key Token Value

Before we bind the task editor to the task, let’s take another look at the value of key.snk’s public key.

Why review the key token? Because adding a previously created strong name key file did not change the value of the Public Key in earlier versions of Visual Studio. That behavior has changed.

Open a command prompt as an administrator as shown in Figure 7-5.

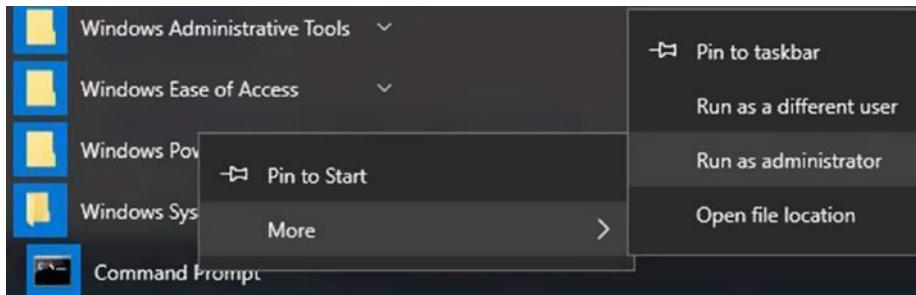


Figure 7-5. Opening an administrator command prompt

During the signing process, we selected the file named key.snk from the ExecuteCatalogPackageTask folder. Visual Studio copied the key.snk file into the ExecuteCatalogPackageTaskUI folder.

Navigate to the ExecuteCatalogPackageTaskUI folder as shown in Figure 7-6.

```
E:\>cd projects
E:\Projects>cd executecatalogpackagetask
E:\Projects\ExecuteCatalogPackageTask>cd executecatalogpackagetaskui
E:\Projects\ExecuteCatalogPackageTask\ExecuteCatalogPackageTaskUI>dir
Volume in drive E is vxDemo_E
Volume Serial Number is CC6E-3E9C
Directory of E:\Projects\ExecuteCatalogPackageTask\ExecuteCatalogPackageTaskUI

02/02/2017 12:30 PM <DIR> .
02/02/2017 12:30 PM <DIR> ..
01/30/2017 12:57 PM <DIR> bin
02/02/2017 12:22 PM 5,421 ExecuteCatalogPackageTaskForm.Designer.vb
02/02/2017 12:18 PM 5,817 ExecuteCatalogPackageTaskForm.resx
02/02/2017 12:18 PM 1,397 ExecuteCatalogPackageTaskForm.vb
02/02/2017 09:39 AM 847 ExecuteCatalogPackageTaskUI.vb
02/02/2017 12:25 PM 6,450 ExecuteCatalogPackageTaskUI.vbproj
01/30/2017 12:52 PM 257 ExecuteCatalogPackageTaskUI.vbproj.vspscc
02/02/2017 12:30 PM 596 key.snk
02/02/2017 11:41 AM <DIR> My Project
01/30/2017 12:52 PM <DIR> obj
02/02/2017 12:30 PM 160 public.out
   8 File(s)      20,945 bytes
   5 Dir(s) 38,088,110,080 bytes free

E:\Projects\ExecuteCatalogPackageTask\ExecuteCatalogPackageTaskUI>
```

Figure 7-6. Navigating to the ExecuteCatalogPackageTaskUI folder

Open the ExecuteCatalogPackageTaskNotes.txt file and copy the public key retrieval commands as shown in Figure 7-7.

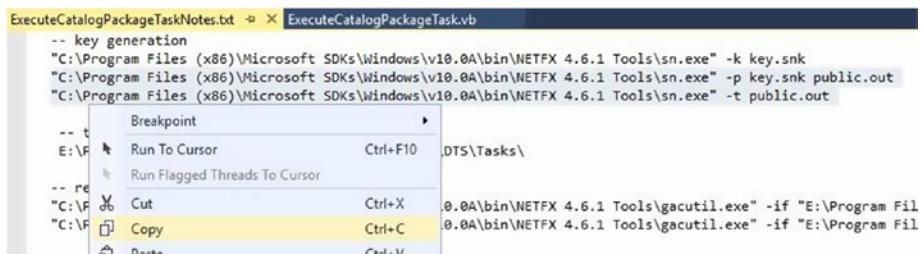
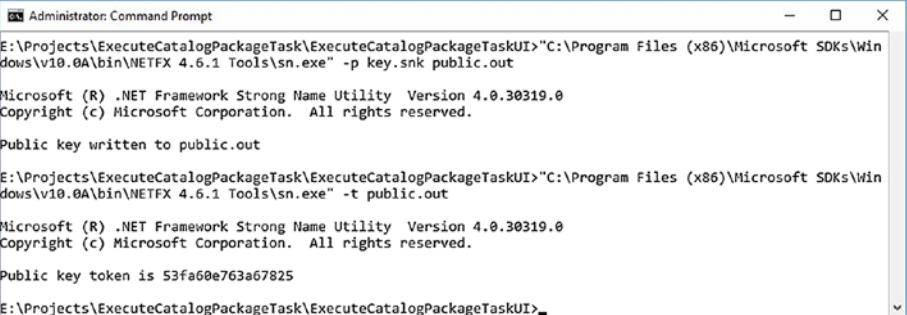


Figure 7-7. Copying the public key retrieval commands

Paste the public key retrieval commands into the Administrator Command Prompt window as shown in Figure 7-8.



```
E:\Projects\ExecuteCatalogPackageTask\ExecuteCatalogPackageTaskUI>"C:\Program Files (x86)\Microsoft SDKs\Windows\v10.0A\bin\NETFX 4.6.1 Tools\sn.exe" -p key.snk public.out
Microsoft (R) .NET Framework Strong Name Utility Version 4.0.30319.0
Copyright (c) Microsoft Corporation. All rights reserved.

Public key written to public.out

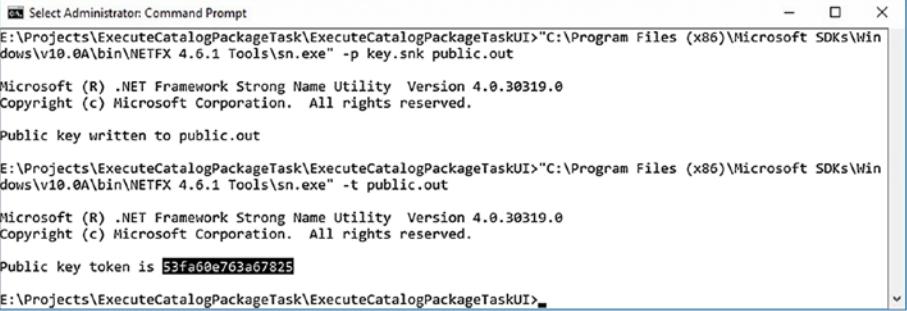
E:\Projects\ExecuteCatalogPackageTask\ExecuteCatalogPackageTaskUI>"C:\Program Files (x86)\Microsoft SDKs\Windows\v10.0A\bin\NETFX 4.6.1 Tools\sn.exe" -t public.out
Microsoft (R) .NET Framework Strong Name Utility Version 4.0.30319.0
Copyright (c) Microsoft Corporation. All rights reserved.

Public key token is 53fa60e763a67825

E:\Projects\ExecuteCatalogPackageTask\ExecuteCatalogPackageTaskUI>
```

Figure 7-8. Retrieving the public key token

Highlight the public key token value as shown in Figure 7-9.



```
E:\Select Administrator: Command Prompt
E:\Projects\ExecuteCatalogPackageTask\ExecuteCatalogPackageTaskUI>"C:\Program Files (x86)\Microsoft SDKs\Windows\v10.0A\bin\NETFX 4.6.1 Tools\sn.exe" -p key.snk public.out
Microsoft (R) .NET Framework Strong Name Utility Version 4.0.30319.0
Copyright (c) Microsoft Corporation. All rights reserved.

Public key written to public.out

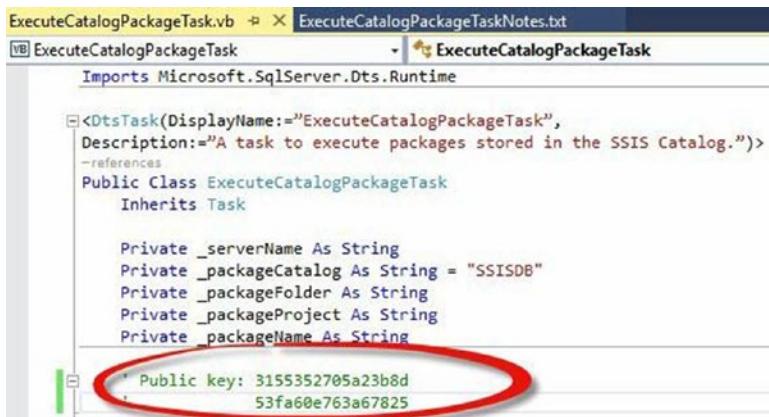
E:\Projects\ExecuteCatalogPackageTask\ExecuteCatalogPackageTaskUI>"C:\Program Files (x86)\Microsoft SDKs\Windows\v10.0A\bin\NETFX 4.6.1 Tools\sn.exe" -t public.out
Microsoft (R) .NET Framework Strong Name Utility Version 4.0.30319.0
Copyright (c) Microsoft Corporation. All rights reserved.

Public key token is 53fa60e763a67825

E:\Projects\ExecuteCatalogPackageTask\ExecuteCatalogPackageTaskUI>
```

Figure 7-9. Highlighting the public key token value

Right-click the selection to copy it to the clipboard. Paste the clipboard contents in the ExecuteCatalogPackageTask.vb file near the original Public key value for comparison as shown in Figure 7-10.



```

ExecuteCatalogPackageTask.vb  X  ExecuteCatalogPackageTaskNotes.txt
VB ExecuteCatalogPackageTask  - ExecuteCatalogPackageTask
Imports Microsoft.SqlServer.Dts.Runtime

<DtsTask(DisplayName:="ExecuteCatalogPackageTask",
Description:="A task to execute packages stored in the SSIS Catalog.")>

Public Class ExecuteCatalogPackageTask
Inherits Task

Private _serverName As String
Private _packageCatalog As String = "SSISDB"
Private _packageFolder As String
Private _packageProject As String
Private _packageName As String

* Public key: 3155352705a23b8d
53fa60e763a67825

```

Figure 7-10. Comparing the original and new public key values

As you can see, the public key token value was updated when the key.snk file was copied into the ExecuteCatalogPackageTaskUI folder (during signing). Aren't you glad we checked? I am!

Binding the Task Editor to the Task

We now 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 7-11.



```

<DtsTask(DisplayName:="ExecuteCatalogPackageTask",
Description:="A task to execute packages stored in the SSIS Catalog.")>

Public Class ExecuteCatalogPackageTask
Inherits Task

```

Figure 7-11. ExecuteCatalogPackageTask, as we begin

The DTSTask attribute (decoration) is shown [here](#) and documented [here](#). In its current state our task decoration has two values defined: DisplayName and Description. To couple the editor we add the multipart attribute UITypeName by adding the following code:

```

, UITypeName:="ExecuteCatalogPackageTaskUI.ExecuteCatalogPackageTaskUI, " & _
"ExecuteCatalogPackageTaskUI,Version=1.0.0.0," & _
"Culture=Neutral," & _
"PublicKeyToken=<Your public key>"
```

Once added to the DTSTask attribute decoration, it will appear similar to that shown in Figure 7-12.

```

ExecuteCatalogPackageTask.vb  X  ExecuteCatalogPackageTaskNotes.txt
VB ExecuteCatalogPackageTask  ExecuteCatalogPackageTask
Imports Microsoft.SqlServer.Dts.Runtime

<DtsTask(DisplayName:="ExecuteCatalogPackageTask",
Description:="A task to execute packages stored in the SSIS Catalog.",
UITypeName:="ExecuteCatalogPackageTaskUI.ExecuteCatalogPackageTaskUI, " &
"ExecuteCatalogPackageTaskUI,Version=1.0.0.0," &
"Culture=Neutral," &
"PublicKeyToken=53fa60e763a67825">
-references
Public Class ExecuteCatalogPackageTask
    Inherits Task

    Private _serverName As String
    Private _packageCatalog As String = "SSISDB"
    Private _packageFolder As String
    Private _packageProject As String
    Private _packageName As String

    ' Public key: 3155352705a23b8d
    '                 53fa60e763a67825

```

Figure 7-12. Adding the UITypeName decoration

You can find (some) documentation for the UITypeName attribute [here](#). The property/value pairs are:

- Type Name: ExecuteCatalogPackageTask.
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? Look in the Execute method shown in Figure 7-13.

```
    References
Public Overrides Function Execute(ByVal connections As Connections,
    ByVal variableDispenser As VariableDispenser,
    ByVal componentEvents As IDTSCOMPONENTEvents,
    ByVal log As IDTSLogging,
    ByVal transaction As Object) As DTSExecResult
    Return DTSExecResult.Success
End Function
```

Figure 7-13. The Execute method

We are going to add SSIS Catalog Package execution functionality here. There are several good articles available that walk you through executing SSIS Catalog Packages via .Net. A good summary of MSDN articles is found [here](#).

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 than we will cover here.

First we 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

The Solution Explorer References virtual folder for the ExecuteCatalogPackageTask project should now appear as shown in Figure 7-14.

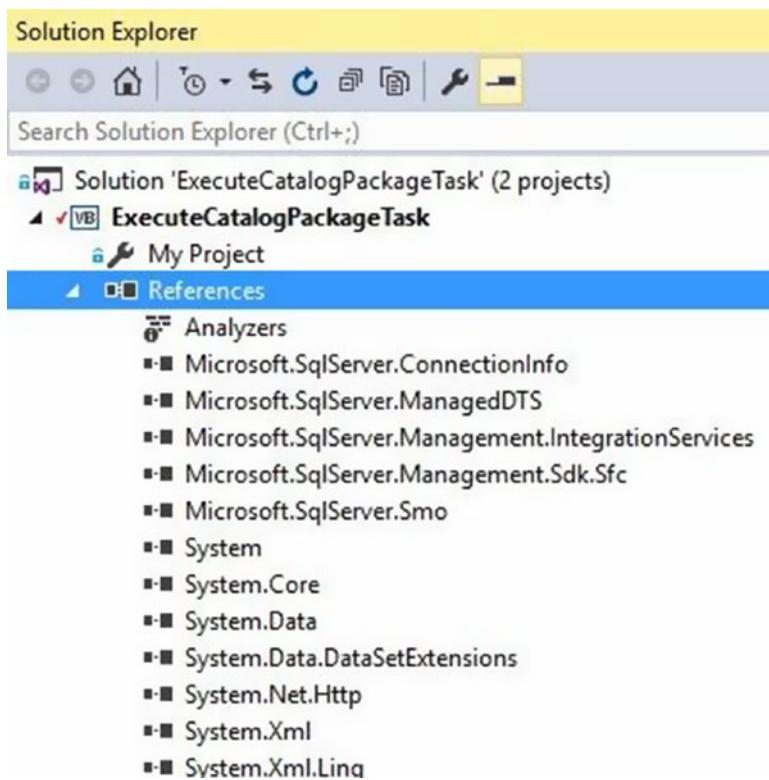


Figure 7-14. Viewing the ExecuteCatalogPackageTask project references

Next, we need to import reference assemblies into `ExecuteCatalogPackageTask.vb` for use in our project as shown in Figure 7-15.

```
Imports Microsoft.SqlServer.Management.IntegrationServices
Imports Microsoft.SqlServer.Management.Smo
```

```
Imports Microsoft.SqlServer.Dts.Runtime
Imports Microsoft.SqlServer.Management.IntegrationServices
Imports Microsoft.SqlServer.Management.Smo
```

Figure 7-15. Importing referenced assemblies

We're now ready to add functionality to the Execute function. Declare and initialize some variables using the following code:

```
Dim catalogServer As New Server(Me.ServerName)
Dim integrationServices As New IntegrationServices(catalogServer)
Dim catalog As Catalog = integrationServices.Catalogs(Me.PackageCatalog)
Dim catalogFolder As CatalogFolder = catalog.Folders(Me.PackageFolder)
Dim catalogProject As ProjectInfo = catalogFolder.Projects(Me.
PackageProject)
Dim catalogPackage As Microsoft.SqlServer.Management.IntegrationServices.
PackageInfo = catalogProject.Packages(Me.PackageName)
```

Your Execute method should appear similar to that shown in Figure 7-16.

```
Public Overrides Function Execute(ByVal connections As Connections,
                                ByVal variableDispenser As VariableDispenser,
                                ByVal componentEvents As IDTSCOMPONENTEvents,
                                ByVal log As IDTSLogging,
                                ByVal transaction As Object) As DTSExecResult

    Dim catalogServer As New Server(Me.ServerName)
    Dim integrationServices As New IntegrationServices(catalogServer)
    Dim catalog As Catalog = integrationServices.Catalogs(Me.PackageCatalog)
    Dim catalogFolder As CatalogFolder = catalog.Folders(Me.PackageFolder)
    Dim catalogProject As ProjectInfo = catalogFolder.Projects(Me.PackageProject)
    Dim catalogPackage As Microsoft.SqlServer.Management.IntegrationServices.PackageInfo

    Return DTSExecResult.Success
End Function
```

Figure 7-16. Execute SSIS Catalog Package method partially coded

Finally, add the call to execute the SSIS Package object (catalogPackage) as shown in Figure 7-17.

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

```
Public Overrides Function Execute(ByVal connections As Connections,
                                ByVal variableDispenser As VariableDispenser,
                                ByVal componentEvents As IDTSCOMPONENTEvents,
                                ByVal log As IDTSLogging,
                                ByVal transaction As Object) As DTSExecResult

    Dim catalogServer As New Server(Me.ServerName)
    Dim integrationServices As New IntegrationServices(catalogServer)
    Dim catalog As Catalog = integrationServices.Catalogs(Me.PackageCatalog)
    Dim catalogFolder As CatalogFolder = catalog.Folders(Me.PackageFolder)
    Dim catalogProject As ProjectInfo = catalogFolder.Projects(Me.PackageProject)
    Dim catalogPackage As Microsoft.SqlServer.Management.IntegrationServices.PackageInfo

    catalogPackage.Execute(False, Nothing)

    Return DTSExecResult.Success
End Function
```

Figure 7-17. 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 7-18.

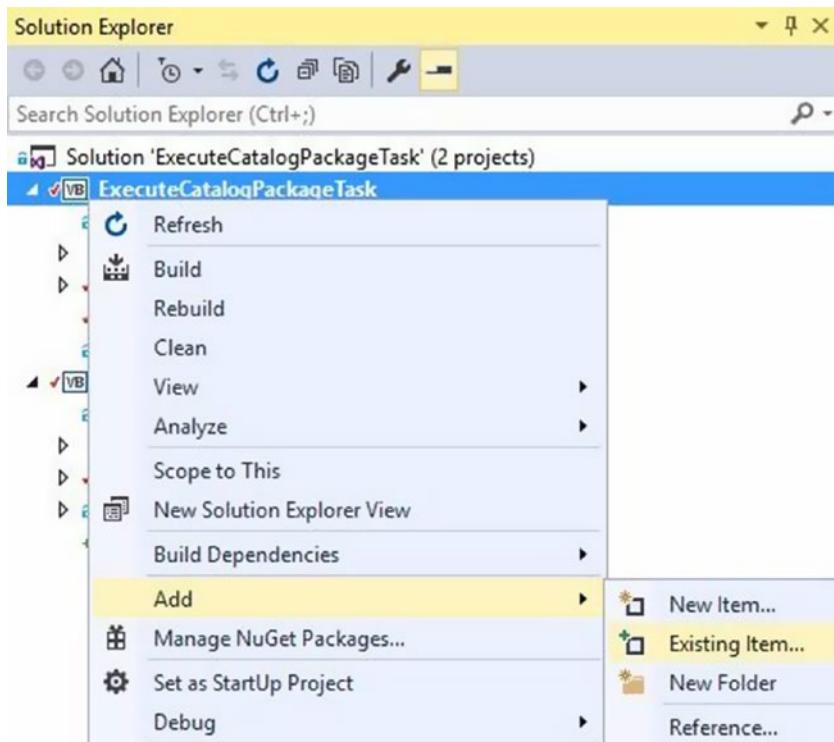


Figure 7-18. Adding an existing item to the ExecuteCatalogPackageTask project

Navigate to the icon file you wish to use as shown in Figure 7-19.

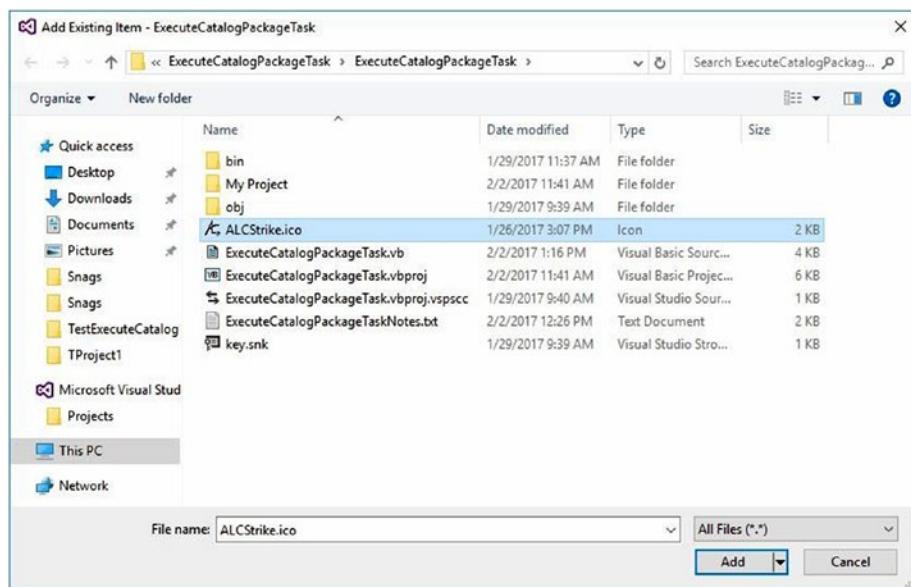


Figure 7-19. Selecting the icon

The icon file will appear in Solution Explorer as shown in Figure 7-20.

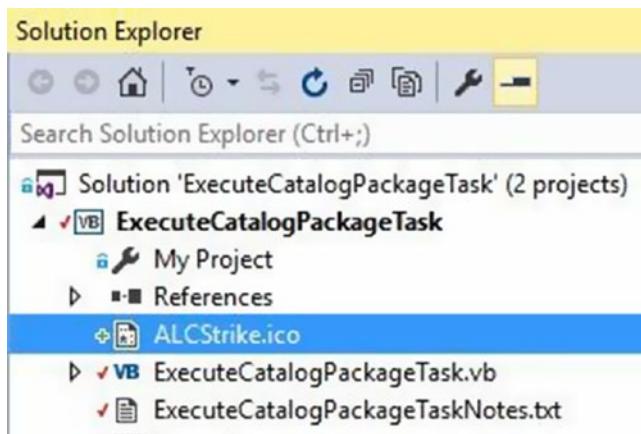


Figure 7-20. Viewing the icon file

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 7-21.

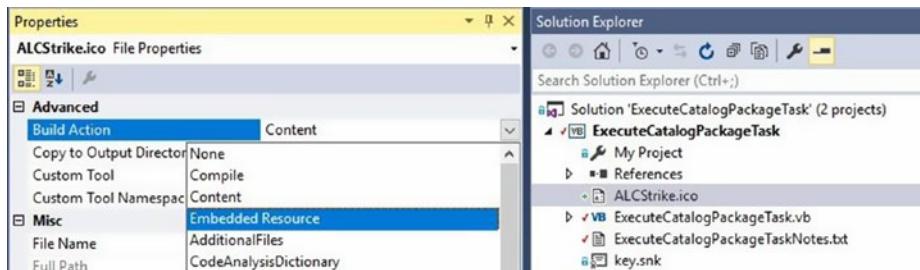


Figure 7-21. Changing the Build Action property of the icon file

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 7-22.

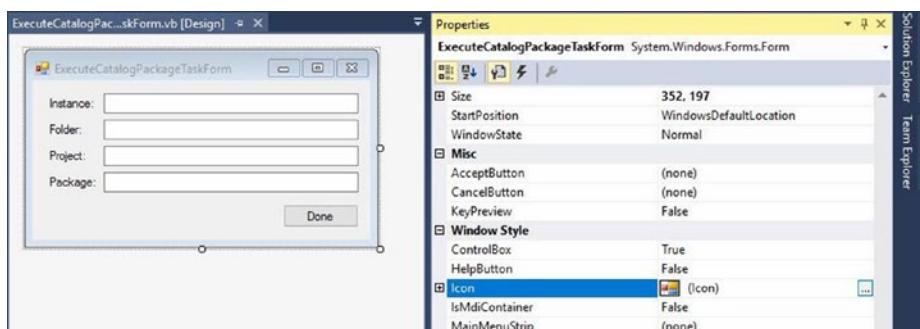


Figure 7-22. Opening the icon selection dialog

Select the icon as shown in Figure 7-23.

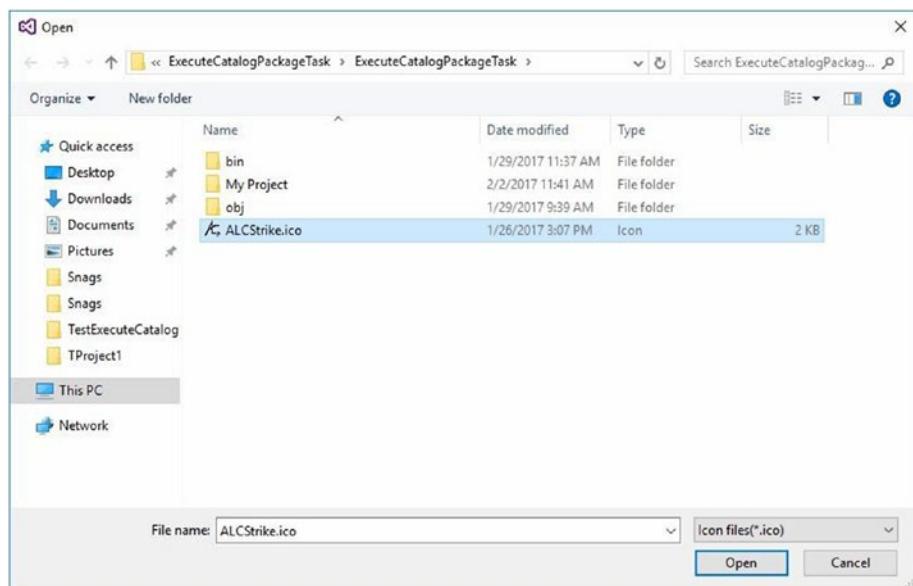


Figure 7-23. Selecting the form icon

The selected icon now appears as the icon for the ExecuteCatalogPackageTaskForm form as shown in Figure 7-24.

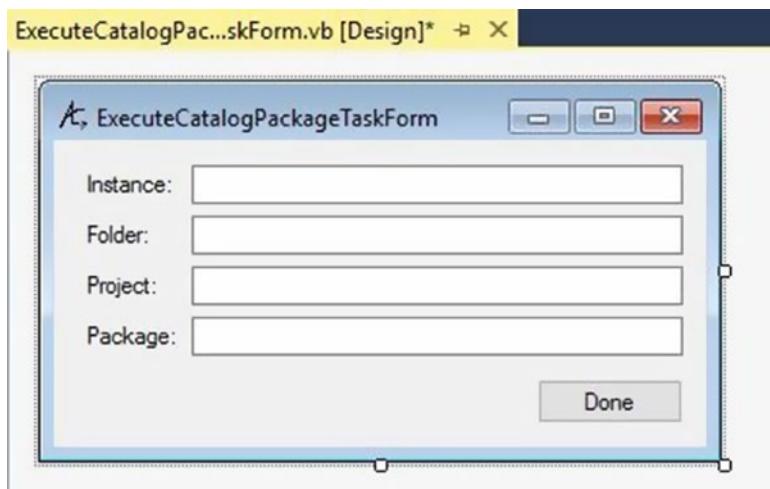


Figure 7-24. Viewing the form icon

While we're here, let's update the Text property of the form to "Execute Catalog Package Task Editor" as shown in Figure 7-25.

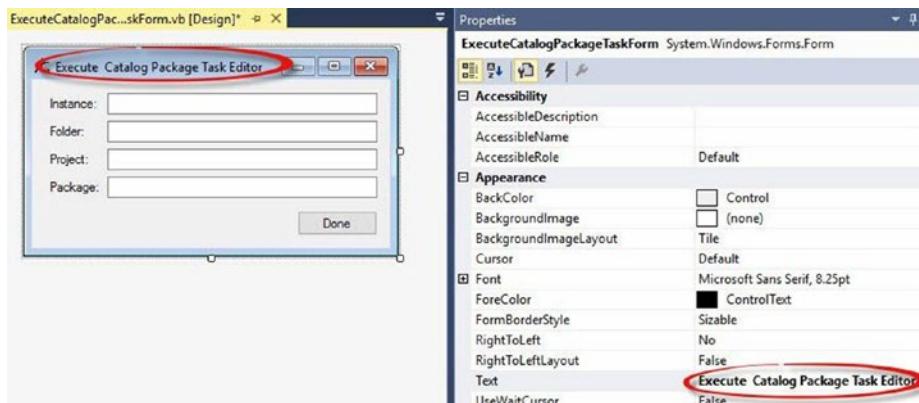


Figure 7-25. Updating the form's text property

SSIS won't know which icon to display until we update the DTSTask decoration for ExecuteCatalogPackageTask. Open ExecuteCatalogPackageTask.vb and add the following line to the decoration:

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

Your decoration should now appear similar to that shown in Figure 7-26.

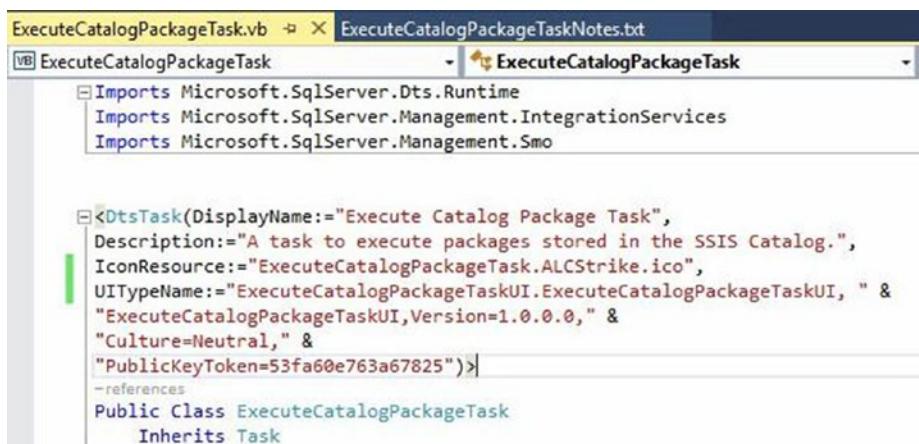


Figure 7-26. Viewing updated DtsTask decoration

Now would be an excellent time to check your code into Source Control.

Building the Task

Coding is done! Now it's time to build our solution, which will compile the code into an executable. From the Build drop-down menu, click Build Solution as shown in Figure 7-27.

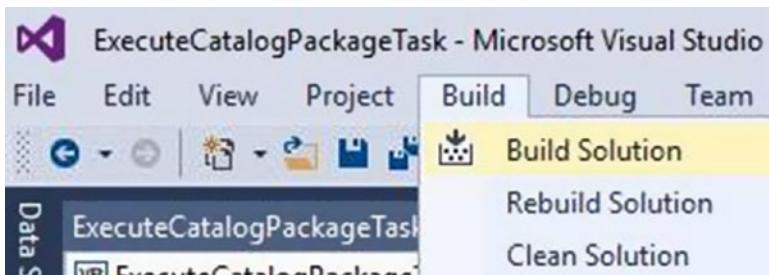


Figure 7-27. Building the solution

If all goes well, you should see verbiage in the Output window similar to that shown in Figure 7-28.

```
Output
Show output from: Build
1>----- Build started: Project: ExecuteCatalogPackageTask, Configuration: Debug Any CPU -----
2>----- Build started: Project: ExecuteCatalogPackageTaskUI, Configuration: Debug Any CPU -----
2> ExecuteCatalogPackageTaskUI -> E:\Program Files (x86)\Microsoft SQL Server\130\DTS\Tasks\ExecuteCatalogPackageTaskUI
1> ExecuteCatalogPackageTask -> E:\Program Files (x86)\Microsoft SQL Server\130\DTS\Tasks\ExecuteCatalogPackageTask.dll
===== Build: 2 succeeded, 0 failed, 0 up-to-date, 0 skipped =====
```

Figure 7-28. Build output

Next, copy the register commands from `ExecuteCatalogPackageTaskNotes.txt` to the clipboard as shown in Figure 7-29.

```
-- register
"C:\Program Files (x86)\Microsoft SDKs\Windows\v10.0A\bin\NETFX 4.6.1 Tools\gacutil.exe" -if "E:\Program Files (x86)\Microsoft SDKs\Windows\v10.0A\bin\NETFX 4.6.1 Tools\gacutil.exe" -if "E:\Program Files (x86)\Microsoft SDKs\Windows\v10.0A\bin\NETFX 4.6.1 Tools\gacutil.exe"
```

Figure 7-29. Copying the GACUtil register command to the clipboard

Open the command prompt as an administrator as shown in Figure 7-30.

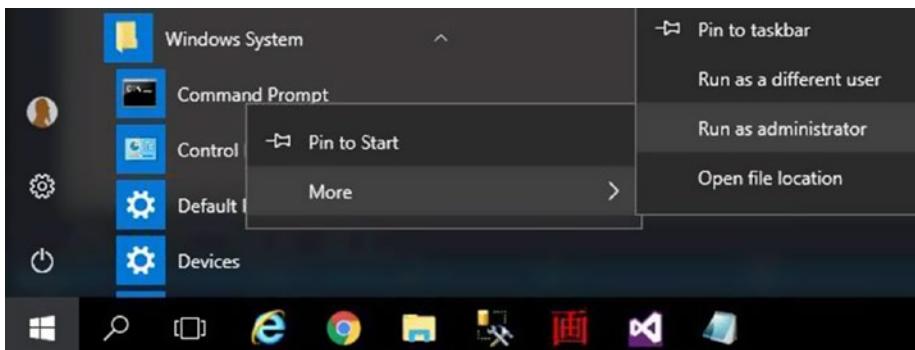


Figure 7-30. Opening the Command Prompt as administrator

Paste the commands into the administrator command prompt as shown in Figure 7-31.

A screenshot of an Administrator Command Prompt window titled "Administrator: Command Prompt". The window shows the following text:

```
Microsoft Windows [Version 10.0.14393]
(c) 2016 Microsoft Corporation. All rights reserved.

C:\Windows\system32>"C:\Program Files (x86)\Microsoft SDKs\Windows\v10.0A\bin\NETFX 4.6.1 Tools\gacutil.exe" -if "E:\Program Files (x86)\Microsoft SQL Server\130\Tasks\ExecuteCatalogPackageTask.dll"
Microsoft (R) .NET Global Assembly Cache Utility. Version 4.0.30319.0
Copyright (c) Microsoft Corporation. All rights reserved.

Assembly successfully added to the cache

C:\Windows\system32>"C:\Program Files (x86)\Microsoft SDKs\Windows\v10.0A\bin\NETFX 4.6.1 Tools\gacutil.exe" -if "E:\Program Files (x86)\Microsoft SQL Server\130\Tasks\ExecuteCatalogPackageTaskUI.dll"
Microsoft (R) .NET Global Assembly Cache Utility. Version 4.0.30319.0
Copyright (c) Microsoft Corporation. All rights reserved.

Assembly successfully added to the cache

C:\Windows\system32>
```

Figure 7-31. Pasting the commands

If all goes as expected, you should see two assemblies successfully registered.

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 7-32.

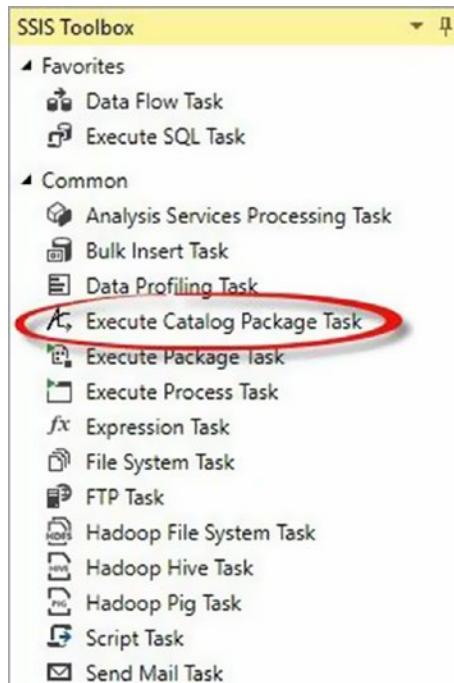


Figure 7-32. In the toolbox!

Drag it onto the surface of the control flow as shown in Figure 7-33.



Figure 7-33. On the control flow!

Double-click the task to open the editor and configure the task as shown in Figure 7-34.

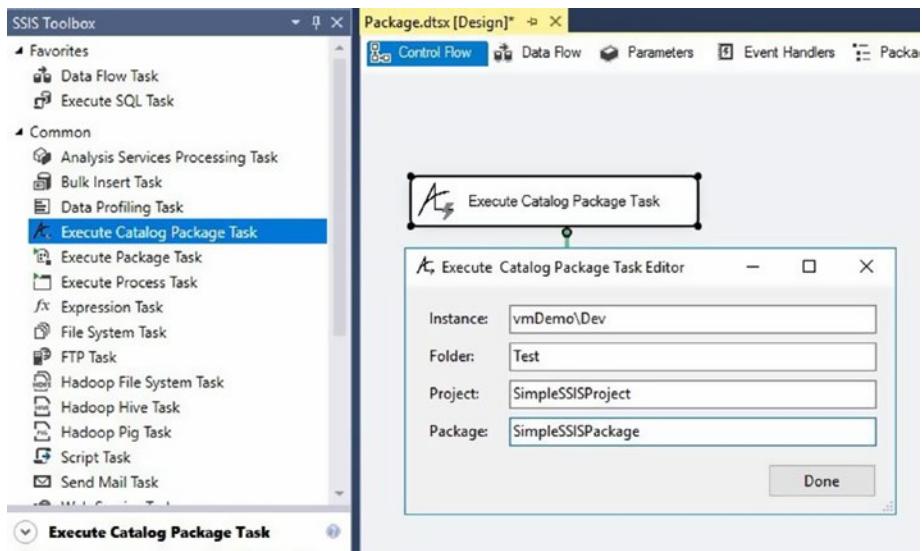


Figure 7-34. Editing the task

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 7-35.

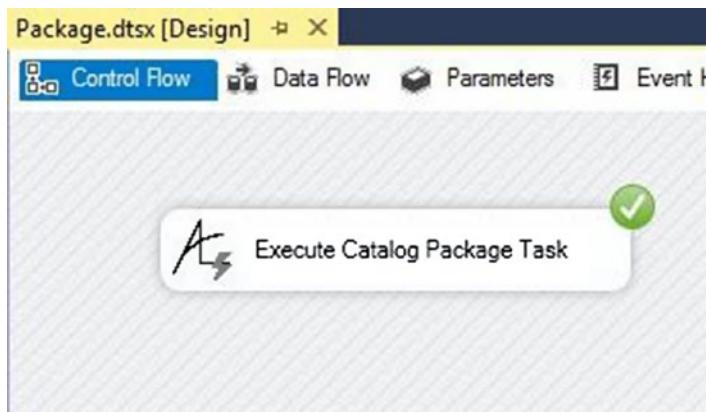


Figure 7-35. Success!

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 SSISDB (SSISDB = SQL Server Integration Services Database), hover over Reports, hover over Standard Reports, and then click All Executions as shown in Figure 7-36.

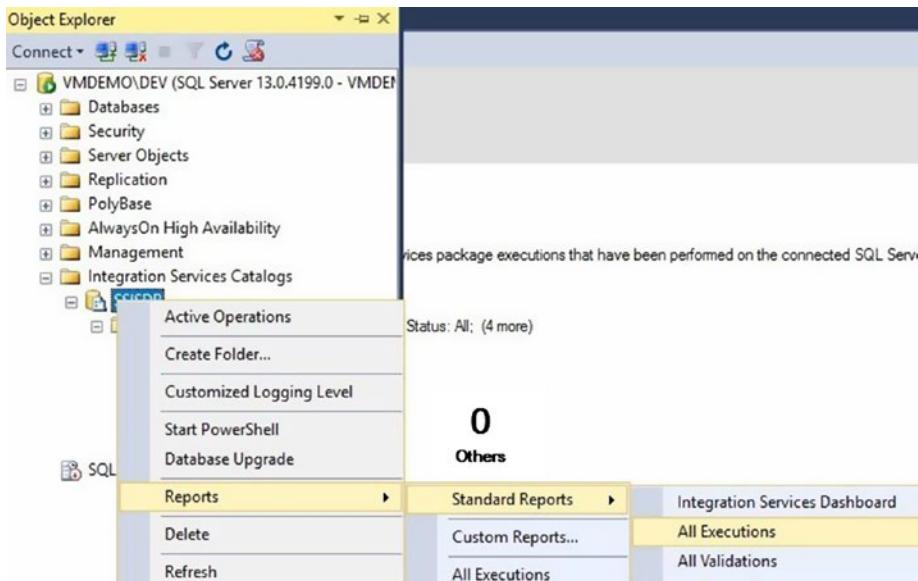


Figure 7-36. Opening the SSIS Catalog All Executions Report in SSMS

The All Executions Report displays SSIS Package executions over the past seven days (by default). I confess. I executed it twice as shown in Figure 7-37.



Figure 7-37. Viewing SSIS Catalog All Executions report

Excellent!

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

CHAPTER 8



Tips on Troubleshooting

In the example to follow in this chapter, I will show you how to set a breakpoint in the `ExecuteCatalogPackageTask`'s `Execute` method, set a breakpoint in the pre-execute event of the Execute Catalog Package Task in a test SSIS (SQL Server Integration Services) project, attach the task's source code to an executing instance of the test SSIS project, and view some metadata about the executing code.

Caution! If this sort of thing—failure—forces your blood pressure to rise, then stop. Stop right now. Don't do any more. And I'm not just talking about this chapter, or even this book; you're going to want to consider some other type of software development at a minimum. Perhaps you want to consider a different career. In software development, failure is a step in the right direction. If you cannot accept this fact on an emotional *and* intellectual level, you are in the wrong field.

Unregistering the Task

First, delete the task from the Control Flow of the test SSIS project, save the test SSIS project, and close SSDT (SQL Server Data Tools). Copy the unregister commands from the `ExecuteCatalogPackageTaskNotes.txt` text file and paste them into the Administrator Command Prompt window as shown in Figure 8-1.

CHAPTER 8 ■ TIPS ON TROUBLESHOOTING



```
C:\>"C:\Program Files (x86)\Microsoft SDKs\Windows\v10.0A\bin\NETFX 4.6.1 Tools\gacutil.exe" -u ExecuteCatalog  
PackageTask  
Microsoft (R) .NET Global Assembly Cache Utility. Version 4.0.30319.0  
Copyright (c) Microsoft Corporation. All rights reserved.  
  
Assembly: ExecuteCatalogPackageTask, Version=1.0.0.0, Culture=neutral, PublicKeyToken=9105ead18a6c17eb, processorArchitecture=MSIL  
Uninstalled: ExecuteCatalogPackageTask, Version=1.0.0.0, Culture=neutral, PublicKeyToken=9105ead18a6c17eb, processorArchitecture=MSIL  
Number of assemblies uninstalled = 1  
Number of failures = 0  
  
C:\>"C:\Program Files (x86)\Microsoft SDKs\Windows\v10.0A\bin\NETFX 4.6.1 Tools\gacutil.exe" -u ExecuteCatalog  
PackageTaskUI  
Microsoft (R) .NET Global Assembly Cache Utility. Version 4.0.30319.0  
Copyright (c) Microsoft Corporation. All rights reserved.  
  
Assembly: ExecuteCatalogPackageTaskUI, Version=1.0.0.0, Culture=neutral, PublicKeyToken=53fa60e763a67825, processorArchitecture=MSIL  
Uninstalled: ExecuteCatalogPackageTaskUI, Version=1.0.0.0, Culture=neutral, PublicKeyToken=53fa60e763a67825, processorArchitecture=MSIL  
Number of assemblies uninstalled = 1  
Number of failures = 0  
  
C:\>_
```

Figure 8-1. Unregistering the assemblies from the GAC

Return to Visual Studio, click the Build drop-down menu, and click Clean Solution as shown in Figure 8-2.

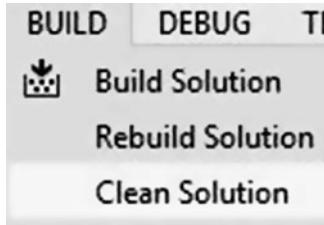


Figure 8-2. Cleaning the solution

If all goes according to plan, you will see the following output as shown in Figure 8-3.

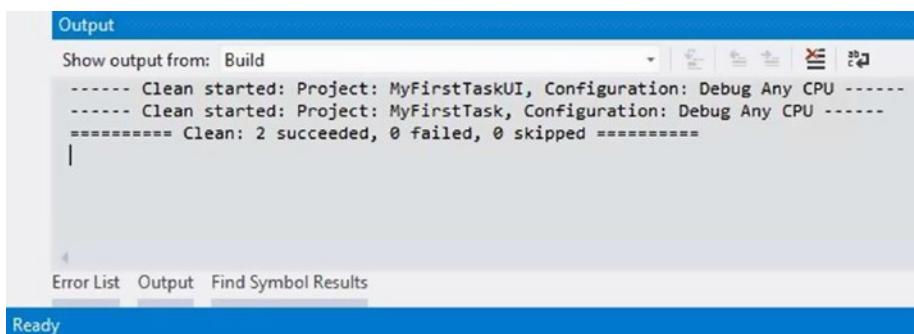


Figure 8-3. Solution cleaned

Troubleshoot the Issue

Let's imagine there's an issue in the Execute method of ExecuteCatalogPackageTask. How could we troubleshoot it? You could open the Execute method, right-click a line of code, and place a breakpoint there as shown in Figure 8-4.

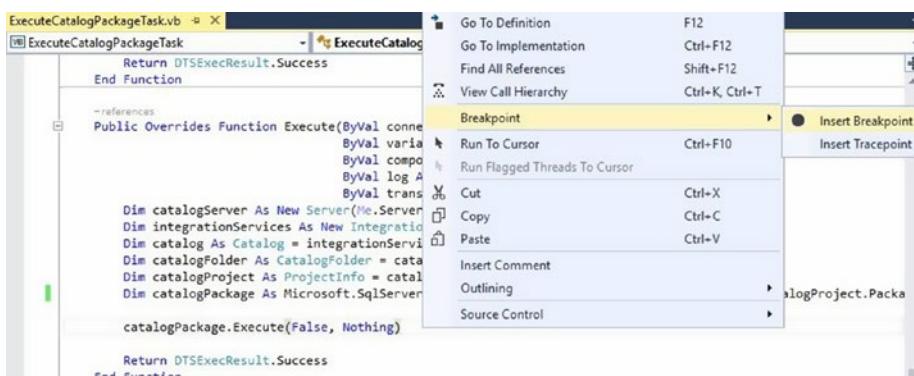


Figure 8-4. Setting a breakpoint

When set, a breakpoint appears as shown in Figure 8-5.



Figure 8-5. Viewing a set breakpoint

Next, let's return to our test SSIS project. Add and configure an Execute Catalog Package Task. Once configured, right-click the task and click Edit Breakpoints... as shown in Figure 8-6.

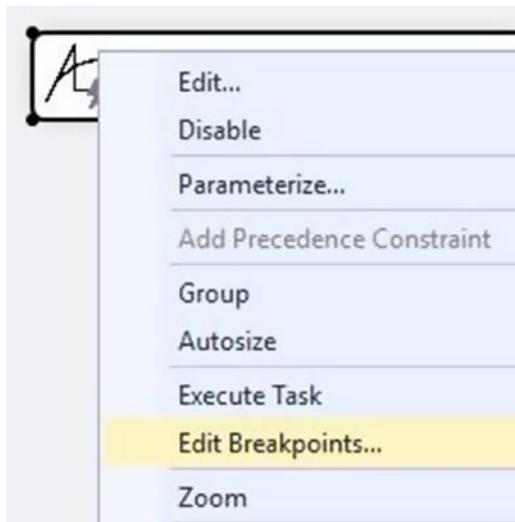


Figure 8-6. Editing the SSIS task breakpoints

When the Set Breakpoints dialog displays, check the check box next to “Break when the container receives the OnPreExecute event” as shown in Figure 8-7.

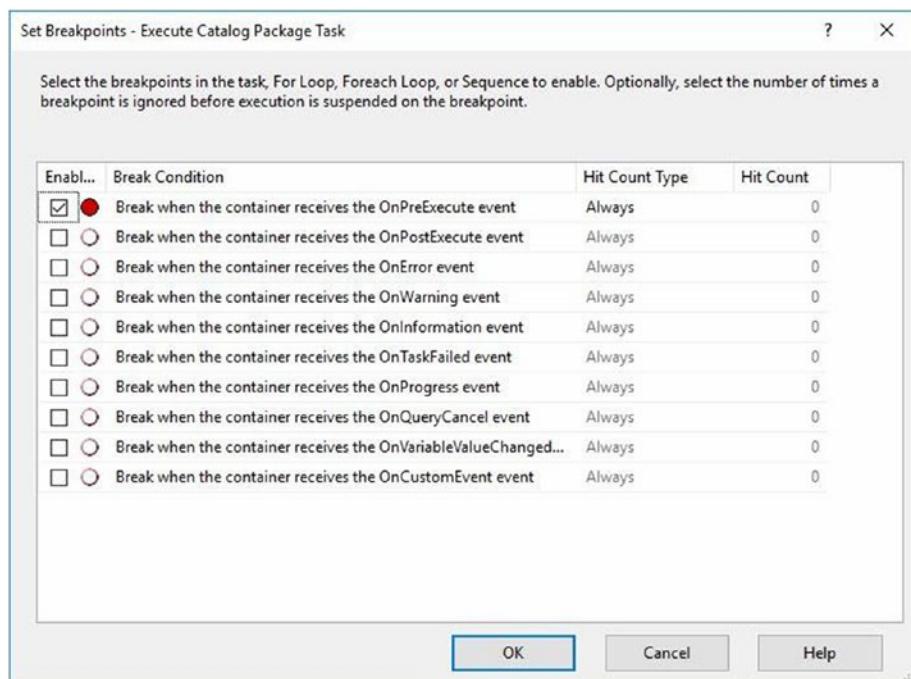


Figure 8-7. Setting the OnPreExecute event breakpoint

Click the OK button and Start the SSIS package in Debug. Package execution will “pause” at the PreExecute of the Execute Catalog Package Task execution as shown in Figure 8-8:



Figure 8-8. Hitting a breakpoint

Return to the ExecuteCatalogPackageTask instance of Visual Studio. Click Debug ► Attach to Process as shown in Figure 8-9.

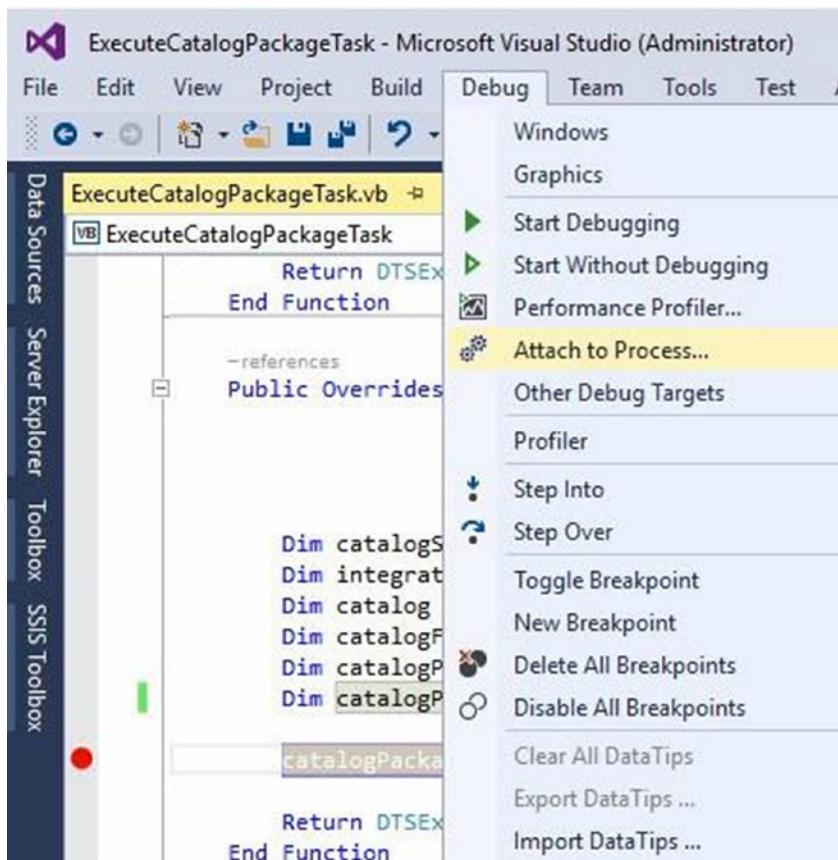


Figure 8-9. Preparing to attach to a process

When the Attach to Process window displays, search the list of executing Windows processes for DtsDebugHost.exe. There will likely be two DtsDebugHost.exe process instances executing for each SSIS process executing in debug mode. I highly recommend you start debugging *only one* SSIS process when attempting to attach to a process.

The Type column of the process to which you want to attach will start with text "Managed" as shown in Figure 8-10.

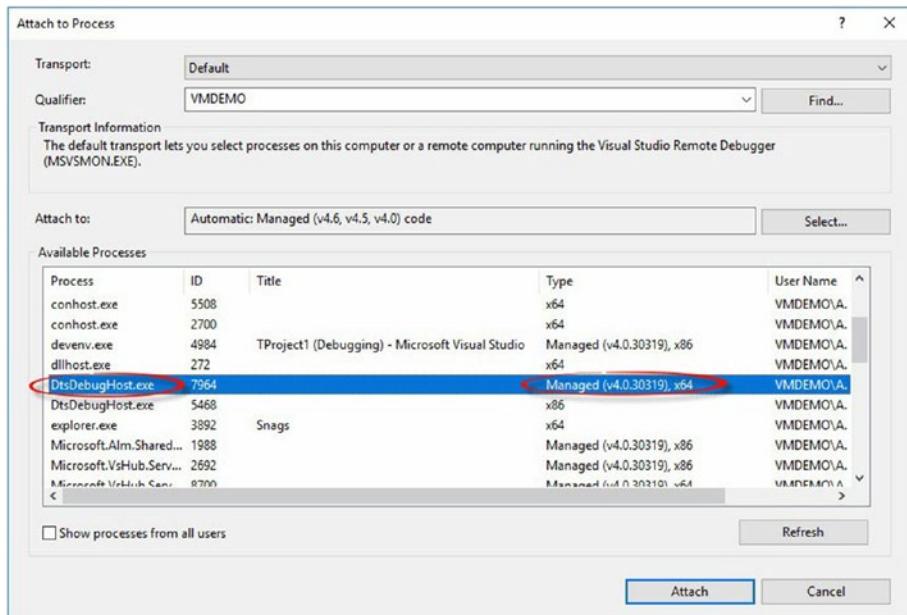


Figure 8-10. Selecting the process

There are two indicators that you have successfully attached to the correct process.

1. The Debug controls for pausing and stopping debugging will be enabled. This indicates you are, in fact, executing in Task source project in Debug mode.
2. The breakpoint will display as a solid circle. Visual Studio will display the breakpoint as an unfilled circle to indicate that the breakpoint may not be hit during execution (see Figure 8-11).



Figure 8-11. A breakpoint that may not be hit

Figure 8-12 shows Visual Studio properly attached to the process with a breakpoint configured—one that *will* be hit.

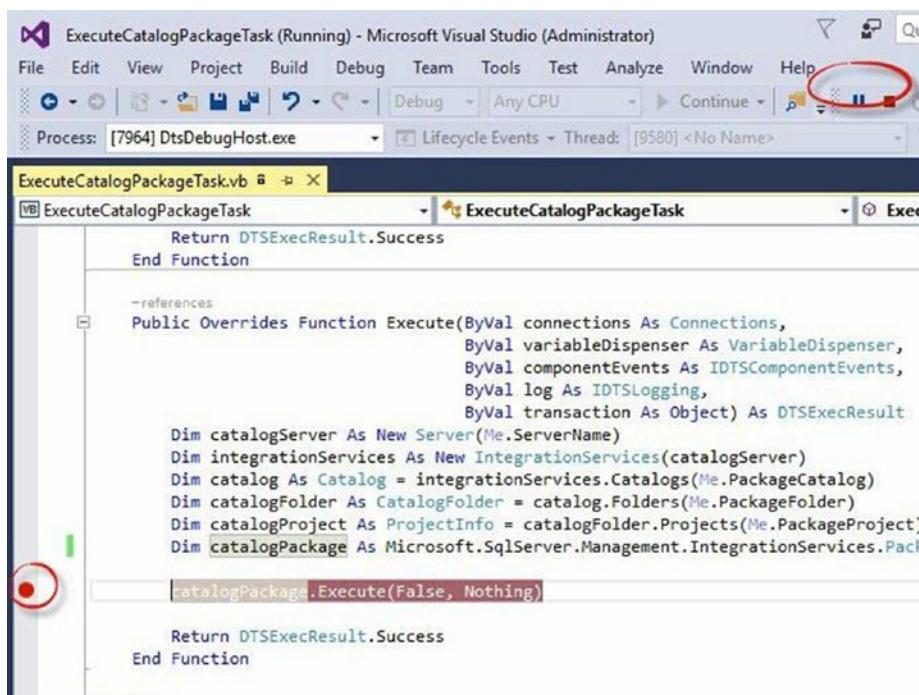


Figure 8-12. Breakpoint properly configured

Return to the test SSIS project, which should have remained “paused” at the Execute Catalog Package Task OnPreExecute breakpoint all this time. Click the Continue button as shown in Figure 8-13.

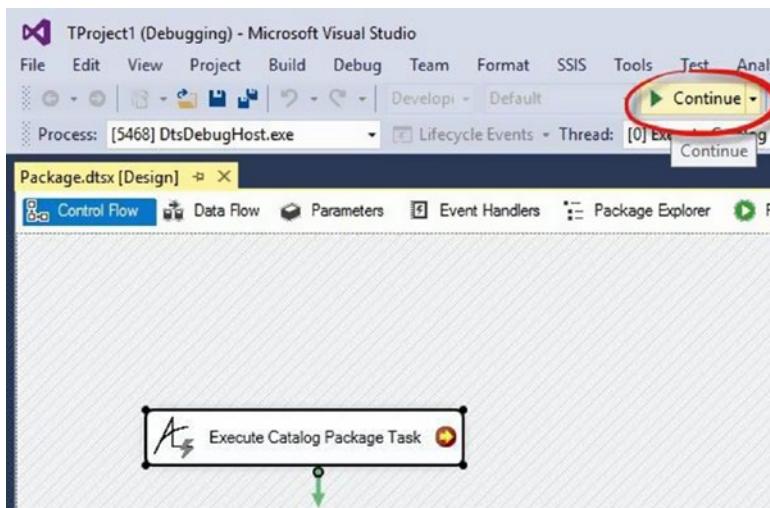
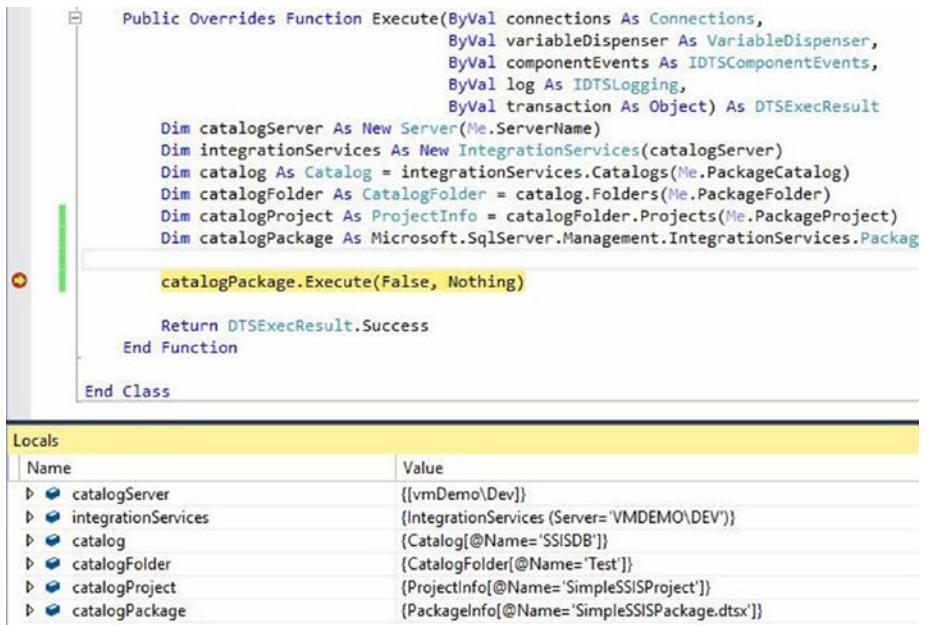


Figure 8-13. Continuing after hitting the breakpoint

The SSIS Package will continue executing and the ExecuteCatalogPackageTask instance of Visual Studio should hit the breakpoint in the Execute method as shown in Figure 8-14.



The screenshot shows a Visual Studio interface. In the top pane, there is C# code for an SSIS package execution. A yellow box highlights the line of code where the breakpoint was hit: `catalogPackage.Execute(False, Nothing)`. Below the code, the 'Locals' window is open, displaying variable names and their current values.

```

Public Overrides Function Execute(ByVal connections As Connections,
                                  ByVal variableDispenser As VariableDispenser,
                                  ByVal componentEvents As IDTSCOMPONENTEvents,
                                  ByVal log As IDTSLogging,
                                  ByVal transaction As Object) As DTSExecResult
    Dim catalogServer As New Server(Me.ServerName)
    Dim integrationServices As New IntegrationServices(catalogServer)
    Dim catalog As Catalog = integrationServices.Catalogs(Me.PackageCatalog)
    Dim catalogFolder As CatalogFolder = catalog.Folders(Me.PackageFolder)
    Dim catalogProject As ProjectInfo = catalogFolder.Projects(Me.PackageProject)
    Dim catalogPackage As Microsoft.SqlServer.Management.IntegrationServices.Package
    catalogPackage.Execute(False, Nothing)

    Return DTSExecResult.Success
End Function

End Class

```

Locals	
Name	Value
catalogServer	{[vmDemo\Dev]}
integrationServices	{IntegrationServices (Server='VMDEMO\DEV')}
catalog	{Catalog[@Name='SSISDB']}
catalogFolder	{CatalogFolder[@Name='Test']}
catalogProject	{ProjectInfo[@Name='SimpleSSISProject']}
catalogPackage	{PackageInfo[@Name='SimpleSSISPackage.dtsx']}

Figure 8-14. Hitting the breakpoint in the attached Execute method

Figure 8-15 shows the Locals window which is a very powerful feature of Visual Studio. You can use Locals to view the current values of variables visible to the current operation's scope. You can open the Locals window while paused at a breakpoint by clicking Debug ▶ Windows ▶ Locals as shown in Figure 8-15.

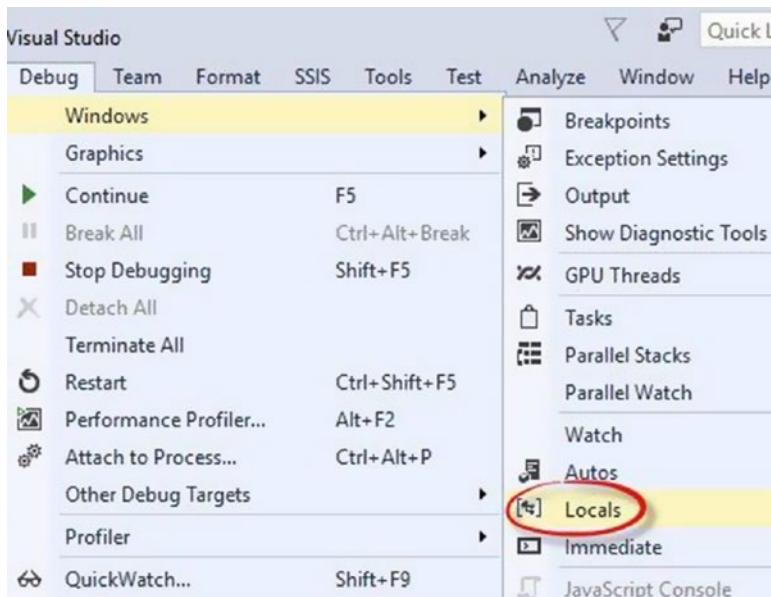


Figure 8-15. Opening the Locals window

Where you go from here depends on the problem you are trying to solve and what you learn along your troubleshooting journey.

CHAPTER 9



Notes from Experience

While learning to build custom SSIS (SQL Server Integration Services) tasks, I encountered elementary errors that I believe experienced .Net developers avoid. When I searched for solutions, I found very little help. The reason? I believe the issues I encountered are not typical for developers building solutions targeted at the Global Assembly Cache (GAC). When one begins developing in the Controls space, one is expected to *just to know* certain things about .Net development.

I did not know some of those things. What problems did I encounter, and what practices did I learn that should be done? The following sections highlight some of my findings.

Start Visual Studio as an Administrator

You'll want to be able to build system folders. To build those, you need to be running Visual Studio as an administrator. When you open Visual Studio, right-click the Visual Studio tile and then click the "Run as administrator" button at the bottom of the screen as shown as Figure 9-1.

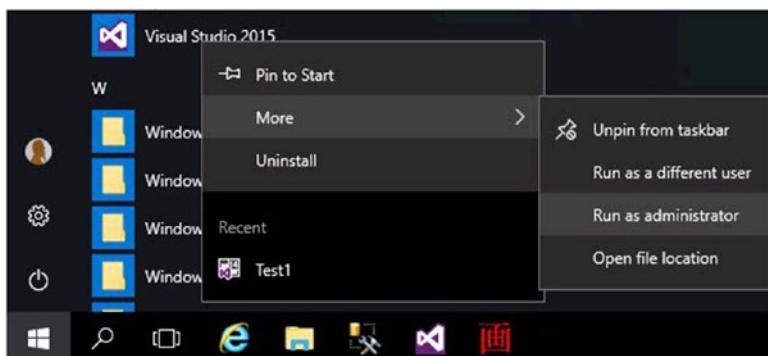


Figure 9-1. Run Visual Studio as administrator

When you click “Run Visual Studio as administrator,” you will be prompted to confirm you wish to run Visual Studio as shown in Figure 9-2.

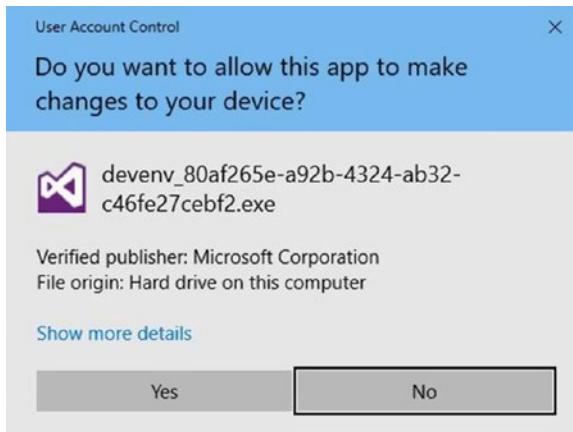


Figure 9-2. Confirm you really want to run Visual Studio as administrator

Why?

This saves the additional step of copying the DLL assemblies from the bin\Debug folder to the ...DTS\Tasks folder. You need the DLLs (dynamic-link libraries) in the Tasks folder so SQL Server Data Tools (SSDT) can locate (and load) them for SSIS development. You need the DLLs in the GAC so SSIS can utilize them at runtime.

Learn How to Recover

Backups are useless. Recoveries are priceless. Start with a recovery strategy. Ask yourself, “Self, if something tragic happens, how will I get back to square one?” Answer that question early. You’ll be glad you did.

Building a Notes File

You can configure Post-Build events in Visual Studio Community Edition when building Visual Basic applications. Building a Notes file helped make us familiar with the Tools, though. I’m glad we took this approach. My Notes file holds commands to unregister and register my DLLs in the GAC, along with key-generation commands for building public/private key pairs using the strong-name utility. As of the end of this project, my Notes file appears as shown in Listing 9-1.

Listing 9-1. Contents of ExecuteCatalogPackageTaskNotes.txt

```
-- key generation  
"C:\Program Files (x86)\Microsoft SDKs\Windows\v10.0A\bin\NETFX 4.6.1 Tools\sn.exe" -k key.snk  
"C:\Program Files (x86)\Microsoft SDKs\Windows\v10.0A\bin\NETFX 4.6.1 Tools\sn.exe" -p key.snk public.out  
"C:\Program Files (x86)\Microsoft SDKs\Windows\v10.0A\bin\NETFX 4.6.1 Tools\sn.exe" -t public.out  
  
-- target folder  
E:\Program Files (x86)\Microsoft SQL Server\130\DTSTasks\  
  
-- register  
"C:\Program Files (x86)\Microsoft SDKs\Windows\v10.0A\bin\NETFX 4.6.1 Tools\gacutil.exe" -if "E:\Program Files (x86)\Microsoft SQL Server\130\DTSTasks\ExecuteCatalogPackageTask.dll"  
"C:\Program Files (x86)\Microsoft SDKs\Windows\v10.0A\bin\NETFX 4.6.1 Tools\gacutil.exe" -if "E:\Program Files (x86)\Microsoft SQL Server\130\DTSTasks\ExecuteCatalogPackageTaskUI.dll"  
  
-- unregister  
"C:\Program Files (x86)\Microsoft SDKs\Windows\v10.0A\bin\NETFX 4.6.1 Tools\gacutil.exe" -u ExecuteCatalogPackageTask  
"C:\Program Files (x86)\Microsoft SDKs\Windows\v10.0A\bin\NETFX 4.6.1 Tools\gacutil.exe" -u ExecuteCatalogPackageTaskUI
```

Cleaning the Solution

Cleaning the solution is the other step in my four-step recovery process outlined in the following synopsis. When something goes wrong,

1. Unregister the assemblies from the GAC.
2. Clean the solution.
3. Correct the issue in the code and Build the solution.
4. Register the assemblies in the GAC.

Open the Command Prompt as an Administrator as shown in Figure 9-3.

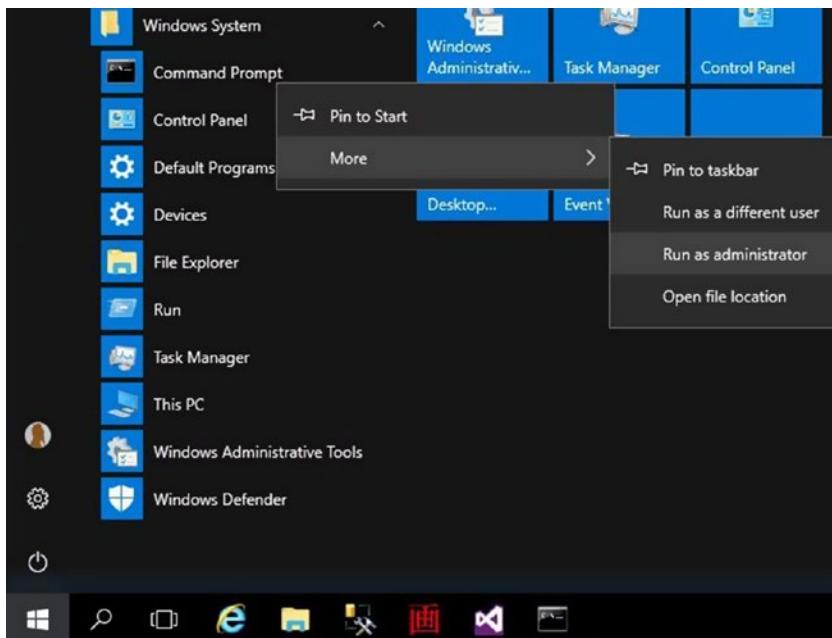


Figure 9-3. Opening the Command Prompt as Administrator

Copy the unregister commands from the ExecuteCatalogPackageTaskNotes.txt text file and paste them into the Administrator Command Prompt window as shown in Figure 9-4.



```
C:\>"C:\Program Files (x86)\Microsoft SDKs\Windows\v10.0A\bin\NETFX 4.6.1 Tools\gacutil.exe" -u ExecuteCatalogPackageTaskUI
Microsoft (R) .NET Global Assembly Cache Utility. Version 4.0.30319.0
Copyright (c) Microsoft Corporation. All rights reserved.

Assembly: ExecuteCatalogPackageTask, Version=1.0.0.0, Culture=neutral, PublicKeyToken=9105ead18a6c17eb, processorArchitecture=MSIL
Uninstalled: ExecuteCatalogPackageTask, Version=1.0.0.0, Culture=neutral, PublicKeyToken=9105ead18a6c17eb, processorArchitecture=MSIL
Number of assemblies uninstalled = 1
Number of failures = 0

C:\>"C:\Program Files (x86)\Microsoft SDKs\Windows\v10.0A\bin\NETFX 4.6.1 Tools\gacutil.exe" -u ExecuteCatalogPackageTaskUI
Microsoft (R) .NET Global Assembly Cache Utility. Version 4.0.30319.0
Copyright (c) Microsoft Corporation. All rights reserved.

Assembly: ExecuteCatalogPackageTaskUI, Version=1.0.0.0, Culture=neutral, PublicKeyToken=53fa60e763a67825, processorArchitecture=MSIL
Uninstalled: ExecuteCatalogPackageTaskUI, Version=1.0.0.0, Culture=neutral, PublicKeyToken=53fa60e763a67825, processorArchitecture=MSIL
Number of assemblies uninstalled = 1
Number of failures = 0

C:\>_
```

Figure 9-4. Unregistering the assemblies from the GAC

Return to Visual Studio, click the Build drop-down menu, and click Clean Solution as shown in Figure 9-5.

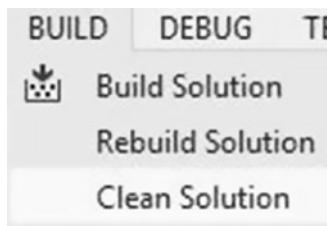
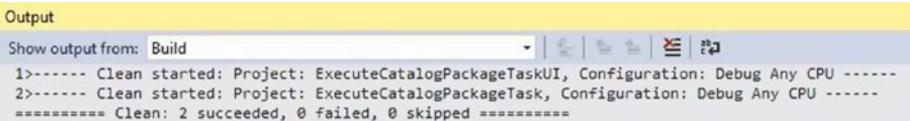


Figure 9-5. Cleaning the solution

If all goes according to plan, you will see the following output as shown in Figure 9-6:



```
Output
Show output from: Build
1>----- Clean started: Project: ExecuteCatalogPackageTaskUI, Configuration: Debug Any CPU -----
2>----- Clean started: Project: ExecuteCatalogPackageTask, Configuration: Debug Any CPU -----
===== Clean: 2 succeeded, 0 failed, 0 skipped =====
```

Figure 9-6. Solution cleaned

Code until you are ready for the next test.

From the Build drop-down menu, click Build Solution as shown in Figure 9-7.

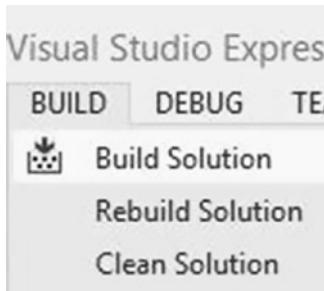


Figure 9-7. Building the solution

If all goes well, you should see verbiage similar to that shown in Figure 9-8.

A screenshot of the Visual Studio Output window titled 'Output'. It shows the following build logs:

```
Output
Show output from: Build
1>----- Build started: Project: ExecuteCatalogPackageTask, Configuration: Debug Any CPU -----
2>----- Build started: Project: ExecuteCatalogPackageTaskUI, Configuration: Debug Any CPU -----
2> ExecuteCatalogPackageTaskUI -> E:\Program Files (x86)\Microsoft SQL Server\130\DTS\Tasks\ExecuteCatalogPackageTaskUI
1> ExecuteCatalogPackageTask -> E:\Program Files (x86)\Microsoft SQL Server\130\DTS\Tasks\ExecuteCatalogPackageTask.dll
===== Build: 2 succeeded, 0 failed, 0 up-to-date, 0 skipped =====
```

Figure 9-8. Build output

Next, as in Figure 9-9, copy the register commands from `ExecuteCatalogPackageTaskNotes.txt` to the clipboard:

```
-- register
"C:\Program Files (x86)\Microsoft SDKs\Windows\v10.0A\bin\NETFX 4.6.1 Tools\gacutil.exe" -if "E:\Program Files (x86)"
"C:\Program Files (x86)\Microsoft SDKs\Windows\v10.0A\bin\NETFX 4.6.1 Tools\gacutil.exe" -if "E:\Program Files (x86)"
```

Figure 9-9. Copy the GACUtil register command

Paste the commands into the Administrator command prompt and if all goes as expected, you should see two assemblies successfully registered as shown in Figure 9-10.

```
C:\>"C:\Program Files (x86)\Microsoft SDKs\Windows\v10.0A\bin\NETFX 4.6.1 Tools\gacutil.exe" -if "E:\Program Files (x86)\Microsoft SQL Server\130\DTSTasks\ExecuteCatalogPackageTask.dll"
Microsoft (R) .NET Global Assembly Cache Utility. Version 4.0.30319.0
Copyright (c) Microsoft Corporation. All rights reserved.

Assembly successfully added to the cache

C:\>"C:\Program Files (x86)\Microsoft SDKs\Windows\v10.0A\bin\NETFX 4.6.1 Tools\gacutil.exe" -if "E:\Program Files (x86)\Microsoft SQL Server\130\DTSTasks\ExecuteCatalogPackageTaskUI.dll"
Microsoft (R) .NET Global Assembly Cache Utility. Version 4.0.30319.0
Copyright (c) Microsoft Corporation. All rights reserved.

Assembly successfully added to the cache

C:\>
```

Figure 9-10. Successfully registered

Change the Icon File's Build Action Property

This particular issue stumped me for ____ (I am embarrassed to say how long, but it was *too* long). You will want to change your icon file's Build Action property to be Embedded Resource. Only then will your task icon display the SSIS Toolbox.

By default, the file's Build Action property will be Content. Change it to Embedded Resource as shown in Figure 9-11.

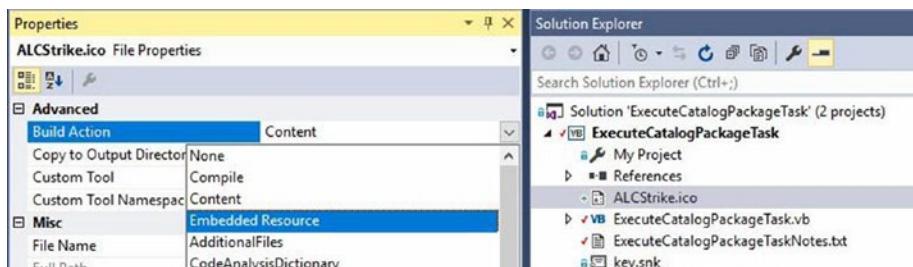


Figure 9-11. Changing the icon file Build Action

This is important. If you do not change the Build Action property of the icon file from Content to Embedded Resource, the icon will *not* show up on your task.

CHAPTER 10



Demonstration Code

The source code for the version of the Execute Catalog Package Task we built is available [here](#) from the [DILM Suite](#) web site, and it's free. You can also access the code from this book's catalog page on the Apress web site (www.apress.com/978-1-4842-2932-2).

Caution The Demo code is *not* production code!

I've placed several this-is-not-production-code warnings throughout the book because, well, *this is not production code*. What would I do to make my example code into production code? That's a great question, and following is my take on answering it:

1. Error handling: Try-catch is completely missing from the sample code. Also missing is validation logic.
2. Features: Executing an SSIS package in the catalog usually involves more than just starting the package. Package execution can be customized and parameterized in a number of ways, including running in 32-bit, executing synchronously, using different logging levels, and using references—just to name a few.
3. Installer: In my opinion, Production-grade code should include an installation process that wraps the code in either a setup.exe or *.msi file. I usually add the installer project to the code project(s) so that the installer is source-controlled with the code project(s).

Kudos to Microsoft for providing developers with the ability to build custom SSIS (SQL Server Integration Services) tasks in Visual Studio Community Edition! I hope you have enjoyed reading this material as much as I enjoyed creating it.

As always, I welcome feedback, suggestions, and comments.

Index

■ A, B

- Base class, Object Browser
 - open, 42
 - task class, 42
- Build Action property, 77, 105

■ C

- Class
 - base, 41
 - decorating, 36
 - inheriting task, 37
 - private variable, 37–38
 - property, 37–41
- Click event
 - add form constructor, 63
 - of btnDone, 59
 - building, 61
 - code, 62
 - form class, 59–61

■ D

- Data Integration Lifecycle Management (DILM), 2
- Delete method, 55
- Demo code, 107
- DialogResult property, 58, 59
- DTSTask attribute, 70–71

■ E

- Execute method, 45–46
 - task functionality, 72
 - troubleshoot, 89
 - attach to process, 92
 - breakpoint properly configured, 94

- continuing after hit
 - breakpoint, 95
- correct process, 93
- edit SSIS breakpoint, 90
- hit breakpoint, 91
- Locals window, 97
- OnPostExecute event
 - breakpoint, 91
- set breakpoint, 89, 90

■ F

- Form
 - add, 56
 - add done button, 58
 - add label and text box, 57
 - coding click event and, 59–64
 - constructor, 63
 - controls, 57
 - DialogResult property, 58–59

■ G, H

- GetView method, 53
- Global Assembly Cache (GAC), 12, 32, 65, 88, 99

■ I, J

- Icon file
 - add existing item, 75
 - Build Action property, 77, 105
 - DtsTask decoration, 79
 - form, 78
 - open selection dialog, 77
 - select, 76
 - update form text property, 79
 - view, 76

■ INDEX

Inheriting task, 37
Initialize method, 54
InitializeTask method, 43–44
Internal variable, 37

■ K, L

Key files, managing
administrator command window
command prompt
to execute, 12
open, 11
apply, 21–23
ExecuteCatalogPackageTaskNotes.txt
file
add, 14
delete, 15–16
find, 14
project, 15
solution, 14
ExecuteCatalogPackageTask project
directory
Key.snk file, 18
navigate, 16
paste, 17
key.snk, 13
public key
extraction, 19
read, 19
select in command
window, 20
store, 20
strong name, 12
Windows Server 2016, 12

■ M

Microsoft’s Visual Studio Team Services
(VSTS), 25

■ N

New method, 53–54
Notes file, 100–101

■ O

Object Browser, 42

■ P, Q

Private variable, 37
Production code
error handling, 107
features, 107
installer, 107
Public key token value
administrator command prompt, 67
comparing, 70
copying retrieval commands, 68
ExecuteCatalogPackageTaskUI
folder, 68
highlighting, 69
retrieving, 69

■ R

Recovery
building notes file, 100–101
build solution, 104
clean solution, 101
build output, 104
click, 103
open as administrator, 102
successfully registered, 105
unregister commands, 102–103

Reference, 35

■ S

ServerName property, 39, 41, 62
SQL Server Data Tools (SSDT), 33–34, 82,
87, 100
SQL Server Management Studio (SSMS), 84

■ T

Task
build, 80
building solution, 80
commands, 81
copy register commands, 80
open as administrator, 81
output, 80
testing
Catalog All Executions report, 85
control flow, 82

- edit, 83
 - SSMS, 84
 - success, 83
 - toolbox, 82
 - Task editor
 - binding
 - DTSTask attribute, 70–71
 - UITypeName attribute, 71
 - coding
 - add, 47, 49
 - add form, 55–59
 - add references, 49
 - implement IDtsTaskUI
 - add generic constructor, 52
 - add TaskHost variable, 52
 - Delete method, 55
 - Error List, 51
 - GetView method, 53
 - Initialize method, 54
 - interface, 51
 - New method, 53–54
 - signing
 - registration command, 67
 - setting build path, 66
 - UI project, 66
 - unregistration command, 67
 - Task functionality
 - ExecuteCatalogPackageTask
 - project, 72–73
 - Execute method, 72
 - import referenced assemblies, 73
 - initialize variables, 74
 - SSIS package, 74
 - TaskHost variable, 52
 - Troubleshoot
 - Execute method, 89
 - attach to process, 92
 - breakpoint properly configured, 94
 - continuing after hit breakpoint, 95
 - correct process, 93
 - edit SSIS breakpoint, 90
 - hit breakpoint, 91
 - Locals window, 97
 - OnPreExecute event breakpoint, 91
 - set breakpoint, 89, 90
- U**
- UITypeName attribute, 71
 - Unregister commands
 - clean solution, 88, 89
 - from GAC, 88
- V**
- Validate method, 44–45
 - Variables
 - definition, 38
 - internal, 37
 - private, 37–38
 - TaskHost, 52
 - Visual Studio
 - build menu, 33
 - keys, 16–17, 20
 - run as administrator, 99–100
 - Visual Studio Community, 33
 - Visual Studio IDE
 - administrator, 3
 - ExecuteCatalogPackageTask, 5
 - new project, 4
 - reference manager, 6–7, 9
 - Visual Studio Source Control
 - add solution, 28
 - annotating pending changes, 31
 - check in, 30
 - community team project, 29
 - connect to community, 28
 - options, 25–26
 - source control plug-in, 27
 - Visual Studio Team Foundation Server, 27
 - VSTS. *See* Microsoft's Visual Studio Team Services (VSTS)
- W, X, Y, Z**
- Windows Server 2016, 12