Integration Services: Extending Packages with Scripting

SQL Server 2012 Books Online

Reference

Microsoft

Integration Services: Extending Packages with Scripting

SQL Server 2012 Books Online

Summary: You can extend the power of Integration Services (SSIS) by adding code within the wrappers provided by the Script task and the Script component. This section of the Developer Reference provides instructions and examples for extending the control flow and data flow of an SSIS package using the Script task and the Script component.

Category: Reference
Applies to: SQL Server 2012
Source: SQL Server 2012 Books Online (link to source content)
E-book publication date: January 2013

Microsoft

Contents

Extending Packages with Scripting

If you find that the built-in components Integration Services do not meet your requirements, you can extend the power of Integration Services by coding your own extensions. You have two discrete options for extending your packages: you can write code within the powerful wrappers provided by the Script task and the Script component, or you can create custom Integration Services extensions from scratch by deriving from the base classes provided by the Integration Services object model.

This section explores the simpler of the two options — .

The Script task and the Script component let you extend both the control flow and the data flow of an Integration Services package with very little coding. Both objects use the Microsoft Visual Studio Tools for Applications (VSTA) development environment and the Microsoft Visual Basic or Microsoft Visual C# programming languages, and benefit from all the functionality offered by the Microsoft .NET Framework class library, as well as custom assemblies. The Script task and the Script component let the developer create custom functionality without having to write all the infrastructure code that is typically required when developing a custom task or custom data flow component.

In This Section

Comparing the Script Task and the Script Component

Discusses the similarities and differences between the Script task and the Script component.

Comparing Scripting Solutions and Custom Objects

Discusses the criteria to use in choosing between a scripting solution and the development of a custom object.

Using Other Assemblies in Scripting Solutions

Discusses the steps required to reference and use external assemblies and namespaces in a scripting project.

Extending the Package with the Script Task

Discusses how to create custom tasks by using the Script task. A task is typically called one time per package execution, or one time for each data source opened by a package.

Extending the Data Flow with the Script Component

Discusses how to create custom data flow sources, transformations, and destinations by using the Script component. A data flow component is typically called one time for each row of data that is processed.

Reference

Integration Services Error Reference

Lists the predefined Integration Services error codes with their symbolic names and descriptions.

Related Sections

Extending Packages with Custom Objects

Discusses how to create program custom tasks, data flow components, and other package objects for use in multiple packages.

Working with Packages Programmatically

Describes how to create, configure, run, load, save, and manage Integration Services packages programmatically.

Stay Up to Date with Integration Services

For the latest downloads, articles, samples, and videos from Microsoft, as well as selected solutions from the community, visit the Integration Services page on MSDN:

Visit the Integration Services page on MSDN

For automatic notification of these updates, subscribe to the RSS feeds available on the page.

See Also

SQL Server Integration Services (SSIS)

Comparing the Script Task and the Script Component

The Script task, available in the Control Flow window of the Integration Services designer, and the Script component, available in the Data Flow window, have very different purposes in an Integration Services package. The task is a general-purpose control flow tool, whereas the component serves as a source, transformation, or destination in the data flow. Despite their different purposes, however, the Script task and the Script component have some similarities in the coding tools that they use and the objects in the package that they make available to the developer. Understanding their similarities and differences may help you to use both the task and the component more effectively.

Similarities between the Script Task and the Script Component

The Script task and the Script component share the following common features.

Feature	Description
Two design-time modes	In both the task and the component, you begin by specifying properties in the editor, and then switch to the development environment to write code.
Microsoft Visual Studio Tools for Applications (VSTA)	Both the task and the component use the same VSTA IDE, and support code written in either Microsoft Visual Basic or Microsoft Visual C#.
Precompiled scripts	Beginning in SQL Server 2008 Integration Services (SSIS), all scripts are precompiled. In earlier versions, you could specify whether scripts were precompiled. The script is precompiled into binary code, permitting faster execution, but at the cost of increased package size.
Debugging	Both the task and the component support breakpoints and stepping through code while debugging in the design environment. For more information, see Coding and Debugging the Script Task and Coding and Debugging the Script Component.

Differences between the Script Task and the Script Component

The Script task and the Script component have the following noteworthy differences.

Feature	Script Task	Script Component
Control flow / Data flow	The Script task is configured on the Control Flow tab of the designer and runs outside the data flow of the package.	The Script component is configured on the Data Flow page of the designer and represents a source, transformation, or destination in the Data Flow task.
Purpose	A Script task can accomplish almost any general-purpose task.	You must specify whether you want to create a source, transformation, or destination with the Script component.
Execution	A Script task runs custom code at some point in the package workflow. Unless you put it in a loop container or an event handler, it only runs once.	A Script component also runs once, but typically it runs its main processing routine once for each row of data in the data flow.
Editor	The **Script Task Editor** has three pages: **General**, **Script**, and **Expressions**. Only the **ReadOnlyVariables** and **ReadWriteVariables**, and **ScriptLanguage** properties directly affect the code that you can write.	The **Script Transformation Editor** has up to four pages: **Input Columns**, **Inputs and Outputs**, **Script**, and **Connection Managers**. The metadata and properties that you configure on each of these pages determines the members of the base classes that are autogenerated for your use in coding.
Interaction with the package	In the code written for a Script task, you use the **Dts** property to access other features of the package. The **Dts** property is a member of the **ScriptMain** class.	In Script component code, you use typed accessor properties to access certain package features such as variables and connection managers. The **PreExecute** method can access only read-only variables. The **PostExecute** method can access both read-only and read/write variables. For more information about these methods, see Coding and Debugging the Script Component.
Using variables	The Script task uses the **P:Microsoft.SqlServer.Dts.Tasks.ScriptT**	The Script component uses typed accessor properties of the autogenerated

Feature	Script Task	Script Component

	ask.ScriptObjectModel.Variables property of the **Dts** object to access variables that are available through the task's **P:Microsoft.SqlServer.Dts.Tasks.ScriptTask.ScriptTask.ReadOnlyVariables** and **P:Microsoft.SqlServer.Dts.Tasks.ScriptTask.ScriptTask.ReadWriteVariables** properties. For example:
	`Dim myVar as String`
	`myVar = Dts.Variables("MyStringVariable").Value.ToString`
	`string myVar;`
	`myVar = Dts.Variables["MyStringVariable"].Value.ToString();`
		based class, created from the component's **P:Microsoft.SqlServer.Dts.Pipeline.ScriptComponent.ReadOnlyVariables** and **P:Microsoft.SqlServer.Dts.Pipeline.ScriptComponent.ReadWriteVariables** properties. For example:
		`Dim myVar as String`
		`myVar = Me.Variables.MyStringVariable`
		`string myVar;`
		`myVar = this.Variables.MyStringVariable;`

Using connections	The Script task uses the **P:Microsoft.SqlServer.Dts.Tasks.ScriptTask.ScriptObjectModel.Connections** property of the **Dts** object to access connection managers defined in the package. For example:
	`Dim myFlatFileConnection As String`
	`myFlatFileConnection = _`
	`DirectCast(Dts.Connections("Test Flat File Connection").AcquireConnection(Dts.Transaction), _`
	`String)`
	`string myFlatFileConnection;`
	`myFlatFileConnection = (Dts.Connections["Test Flat File`
		The Script component uses typed accessor properties of the autogenerated base class, created from the list of connection managers entered by the user on the Connection Managers page of the editor. For example:
		`Dim connMgr As IDTSConnectionManager100`
		`connMgr = Me.Connections.MyADONETConnection`
		`IDTSConnectionManager100 connMgr;`
		`connMgr = this.Connections.MyADONETConnection;`

Feature	Script Task	Script Component
	`Connection"].AcquireConnection( Dts.Transaction) as String);`	

Raising events	The Script task uses the **P:Microsoft.SqlServer.Dts.Tasks.ScriptTask.ScriptObjectModel.Events** property of the **Dts** object to raise events. For example:
	`Dts.Events.FireError(0, "Event Snippet", _`
	`  ex.Message & ControlChars.CrLf & ex.StackTrace, _`
	`  "", 0)`
	
	`Dts.Events.FireError(0, "Event Snippet", ex.Message + "\r" + ex.StackTrace, "", 0);`
		The Script component raises errors, warnings, and informational messages by using the methods of the **T:Microsoft.SqlServer.Dts.Pipeline.Wrapper.IDTSComponentMetaData100** interface returned by the **P:Microsoft.SqlServer.Dts.Pipeline.ScriptComponent.ComponentMetaData** property. For example:
		`Dim myMetadata as IDTSComponentMetaData100`
		`myMetaData = Me.ComponentMetaData`
		`myMetaData.FireError(...)`

Logging	The Script task uses the **M:Microsoft.SqlServer.Dts.Tasks.ScriptTask.ScriptObjectModel.Log(System.String,System.Int32,System.Byte[])** method of the **Dts** object to log information to enabled log providers. For example:
	`Dim bt(0) As Byte`
	`Dts.Log("Test Log Event", _`
	`  0, _`
	`  bt)`
	`byte[] bt = new byte[0];`
	`Dts.Log("Test Log Event", 0, bt);`
		The Script component uses the **M:Microsoft.SqlServer.Dts.Pipeline.ScriptComponent.Log(System.String,System.Int32,System.Byte[])** method of the autogenerated base class to log information to enabled log providers. For example:
		[Visual Basic]
		`Dim bt(0) As Byte`
		`Me.Log("Test Log Event", _`
		`  0, _`
		`  bt)`
		`byte[] bt = new byte[0];`
		`this.Log("Test Log Event", 0, bt);`

Returning results	The Script task uses both the **P:Microsoft.SqlServer.Dts.Tasks.ScriptTask.ScriptObjectModel.TaskResult** property and the optional	The Script component runs as a part of the Data Flow task and does not report results using either of these properties.

Feature	Script Task	Script Component
	P:Microsoft.SqlServer.Dts.Tasks.ScriptTask.ScriptObjectModel.ExecutionValue property of the **Dts** object to notify the runtime of its results.	

Stay Up to Date with Integration Services

For the latest downloads, articles, samples, and videos from Microsoft, as well as selected solutions from the community, visit the Integration Services page on MSDN:

Visit the Integration Services page on MSDN

For automatic notification of these updates, subscribe to the RSS feeds available on the page.

See Also

Extending the Package with the Script Task

Extending the Data Flow with the Script Component

Comparing Scripting Solutions and Custom Objects

An Integration Services Script task or Script component can implement much of the same functionality that is possible in a custom managed task or data flow component. Here are some considerations to help you choose the appropriate type of task for your needs:

- If the configuration or functionality is specific to an individual package, you should use the Script task or the Script component instead of a developing a custom object.

- If the functionality is generic, and might be used in the future for other packages and by other developers, you should create a custom object instead of using a scripting solution. You can use a custom object in any package, whereas a script can be used only in the package for which it was created.
- If the code will be reused within the same package, you should consider creating a custom object. Copying code from one Script task or component to another leads to reduced maintainability by making it more difficult to maintain and update multiple copies of the code.
- If the implementation will change over time, consider using a custom object. Custom objects can be developed and deployed separately from the parent package, whereas an update made to a scripting solution requires the redeployment of the whole package.

Stay Up to Date with Integration Services

For the latest downloads, articles, samples, and videos from Microsoft, as well as selected solutions from the community, visit the Integration Services page on MSDN:

Visit the Integration Services page on MSDN

For automatic notification of these updates, subscribe to the RSS feeds available on the page.

See Also

Extending Packages with Custom Objects

Referencing Other Assemblies in Scripting Solutions

The Microsoft .NET Framework class library provides the script developer with a powerful set of tools for implementing custom functionality in Integration Services packages. The Script task and the Script component can also use custom managed assemblies.

Note
To enable your packages to use the objects and methods from a Web service, use the **Add Web Reference** command available in Microsoft Visual Studio Tools for Applications (VSTA). In earlier versions of Integration Services, you had to generate a proxy class to use a Web service.

Using a Managed Assembly

For Integration Services to find a managed assembly at design time, you must do the following steps:

1. Store the managed assembly in any folder on your computer.

   Note
   In earlier versions of Integration Services, you could only add a reference to a managed assembly that was stored in the %windir%\Microsoft.NET\Framework\vx.x.xxxxx folder or the %ProgramFiles%\Microsoft SQL Server\100\SDK\Assemblies folder.

2. Add a reference to the managed assembly.

   To add the reference, in VSTA, in the **Add Reference** dialog box, on the **Browse** tab, locate and add the managed assembly.

For Integration Services to find the managed assembly at run time, you must do the following steps:

1. Sign the managed assembly with a strong name.
2. Install the assembly in the global assembly cache on the computer on which the package is run.

   For more information, see Building, Deploying, and Debugging Custom Objects.

Using the Microsoft .NET Framework Class Library

The Script task and the Script component can take advantage of all the other objects and functionality exposed by the .NET Framework class library. For example, by using the .NET Framework, you can retrieve information about your environment and interact with the computer that is running the package.

This list describes several of the more frequently used .NET Framework classes:

- **System.Data** Contains the ADO.NET architecture.
- **System.IO** Provides an interface to the file system and streams.
- **System.Windows.Forms** Provides form creation.
- **System.Text.RegularExpressions** Provides classes for working with regular expressions.
- **System.Environment** Returns information about the local computer, the current user, and computer and user settings.
- **System.Net** Provides network communications.

- **System.DirectoryServices** Exposes Active Directory.
- **System.Drawing** Provides extensive image manipulation libraries.
- **System.Threading** Enables multithreaded programming.

For more information about the .NET Framework, see the MSDN Library.

Stay Up to Date with Integration Services

For the latest downloads, articles, samples, and videos from Microsoft, as well as selected solutions from the community, visit the Integration Services page on MSDN:

Visit the Integration Services page on MSDN

For automatic notification of these updates, subscribe to the RSS feeds available on the page.

See Also

Debug a Script by Setting Breakpoints in a Script Task and Script Component

This procedure describes how to set breakpoints in the scripts that are used in the Script task and Script component.

After you set breakpoints in the script, the **Set Breakpoints - <object name>** dialog box lists the breakpoints, along with the built-in breakpoints.

Important
Under certain circumstances, breakpoints in the Script task and Script component are ignored. For more information, see the **Debugging the Script Task** section in Coding and Debugging the Script Task and the **Debugging the Script Component** section in Coding and Debugging the Script Component.

Procedures

To set a breakpoint in script

1. In SQL Server Data Tools (SSDT), open the Integration Services project that contains the package you want.
2. Double-click the package that contains the script in which you want to set breakpoints.
3. To open the Script task, click the **Control Flow** tab, and then double-click the Script task.
4. To open the Script component, click the **Data Flow** tab, and then double-click the Script component.
5. Click **Script** and then click **Edit Script**.
6. In Microsoft Visual Studio Tools for Applications (VSTA), locate the line of script on which you want to set a breakpoint, right-click that line, point to **Breakpoint**, and then click **Insert Breakpoint**.

   The breakpoint icon appears on the line of code.
7. On the **File** menu, click **Exit**.
8. Click **OK**.
9. To save the package, click **Save Selected Items** on the **File** menu.

Extending the Package with the Script Task

The Script task extends the run-time capabilities of Microsoft Integration Services packages with custom code written in Microsoft Visual Basic or Microsoft Visual C# that is compiled and executed at package run time. The Script task simplifies the development of a custom run-time task when the tasks included with Integration Services do not fully satisfy your requirements. The Script task writes all the required infrastructure code for you, letting you focus exclusively on the code that is required for your custom processing.

A Script task interacts with the containing package through the global **Dts** object, an instance of the **T:Microsoft.SqlServer.Dts.Tasks.ScriptTask.ScriptObjectModel** class that is exposed in the scripting environment. You can write code in a Script task that modifies the values stored in Integration Services variables; later, the package can use those updated values to determine the path of its workflow. The Script task can also use the Visual Basic namespace and the .NET Framework class library, as well as custom assemblies, to implement custom functionality.

The Script task and the infrastructure code that it generates for you simplify significantly the process of developing a custom task. However, to understand how the Script task works, you may find it useful to read the section Extending the Control Flow with Custom Tasks to understand the steps that are involved in developing a custom task.

If you are creating a task that you plan to reuse in multiple packages, you should consider developing a custom task instead of using the Script task. For more information, see Comparing Scripting Solutions and Custom Objects.

In This Section

The following topics provide more information about the Script task.

Configuring the Script Task

Explains how the properties that you configure in the **Script Task Editor** affect the capabilities and the performance of the code in the Script task.

Coding the Script Task

Explains how to use Microsoft Visual Studio Tools for Applications (VSTA) to develop the scripts that are contained in the Script task.

Using Variables in the Script Task

Explains how to use variables through the **P:Microsoft.SqlServer.Dts.Tasks.ScriptTask.ScriptObjectModel.Variables** property.

Using Connections in the Script Task

Explains how to use connections through the **P:Microsoft.SqlServer.Dts.Tasks.ScriptTask.ScriptObjectModel.Connections** property.

Raising Events in the Script Task

Explains how to raise events through the **P:Microsoft.SqlServer.Dts.Tasks.ScriptTask.ScriptObjectModel.Events** property.

Logging in the Script Task

Explains how to log information through the **M:Microsoft.SqlServer.Dts.Tasks.ScriptTask.ScriptObjectModel.Log(System.String, System.Int32,System.Byte[])** method.

Returning Results from the Script Task

Explains how to return results through the property **P:Microsoft.SqlServer.Dts.Tasks.ScriptTask.ScriptObjectModel.TaskResult** and the **P:Microsoft.SqlServer.Dts.Tasks.ScriptTask.ScriptObjectModel.ExecutionValue** property.

Script Task Examples

Provides simple examples that demonstrate several possible uses for a Script task.

Stay Up to Date with Integration Services

For the latest downloads, articles, samples, and videos from Microsoft, as well as selected solutions from the community, visit the Integration Services page on MSDN:

Visit the Integration Services page on MSDN

For automatic notification of these updates, subscribe to the RSS feeds available on the page.

See Also

Script Task

Comparing the Script Task and the Script Component

Configuring the Script Task in the Script Task Editor

Before you write custom code in the Script task, you configure its principal properties in the three pages of the **Script Task Editor**. You can configure additional task properties that are not unique to the Script task by using the Properties window.

Note
Unlike earlier versions where you could indicate whether scripts were precompiled, all scripts are precompiled beginning in SQL Server 2008 Integration Services (SSIS).

General Page of the Script Task Editor

On the **General** page of the **Script Task Editor**, you assign a unique name and a description for the Script task.

Script Page of the Script Task Editor

The **Script** page of the **Script Task Editor** displays the custom properties of the Script task.

ScriptLanguage Property

Microsoft Visual Studio Tools for Applications (VSTA) supports the Microsoft Visual Basic or Microsoft Visual C# programming languages. After you create a script in the Script task, you cannot change value of the **ScriptLanguage** property.

To set the default script language for Script tasks and Script components, use the **ScriptLanguage** property on the **General** page of the **Options** dialog box. For more information, see Select Variables Page (VSTA).

EntryPoint Property

The **EntryPoint** property specifies the method on the **ScriptMain** class in the VSTA project that the Integration Services runtime calls as the entry point into the Script task code. The **ScriptMain** class is the default class generated by the script templates.

If you change the name of the method in the VSTA project, you must change the value of the **EntryPoint** property.

ReadOnlyVariables and ReadWriteVariables Properties

You can enter comma-delimited lists of existing variables as the values of these properties to make the variables available for read-only or read/write access within the Script task code. Variables of both types are accessed in code through the **P:Microsoft.SqlServer.Dts.Tasks.ScriptTask.ScriptObjectModel.Variables** property of the **Dts** object. For more information, see Using Variables in the Script Task.

Note
Variable names are case-sensitive.

To select the variables, click the ellipsis (**...**) button next to the property field. For more information, see Select Variables Page.

Edit Script Button

The **Edit Script** button launches the VSTA development environment in which you write your custom script. For more information, see Coding the Script Task.

Expressions Page of the Script Task Editor

On the **Expressions** page of the **Script Task Editor**, you can use expressions to provide values for the properties of the Script task listed above and for many other task properties. For more information, see Using Expressions in Packages.

Stay Up to Date with Integration Services

For the latest downloads, articles, samples, and videos from Microsoft, as well as selected solutions from the community, visit the Integration Services page on MSDN:

Visit the Integration Services page on MSDN

For automatic notification of these updates, subscribe to the RSS feeds available on the page.

See Also

Coding the Script Task

Coding and Debugging the Script Task

After configuring the Script task in the **Script Task Editor**, you write your custom code in the Script task development environment.

Script Task Development Environment

The Script task uses Microsoft Visual Studio Tools for Applications (VSTA) as the development environment for the script itself.

Script code is written in Microsoft Visual Basic or Microsoft Visual C#. You specify the script language by setting the **ScriptLanguage** property in the **Script Task Editor**. If you prefer to use another programming language, you can develop a custom assembly in your language of choice and call its functionality from the code in the Script task.

The script that you create in the Script task is stored in the package definition. There is no separate script file. Therefore, the use of the Script task does not affect package deployment.

Note

When you design the package and debug the script, the script code is temporarily written to a project file. Because storing sensitive information in a file is a potential security risk, we recommend that you do not include sensitive information such as passwords in the script code.

By default, **Option Strict** is disabled in the IDE.

Script Task Project Structure

When you create or modify the script that is contained in a Script task, VSTA opens an empty new project or reopens the existing project. The creation of this VSTA project does not affect the deployment of the package, because the project is saved inside the package file; the Script task does not create additional files.

Project Items and Classes in the Script Task Project

By default, the Script task project displayed in the VSTA Project Explorer window contains a single item, **ScriptMain**. The **ScriptMain** item, in turn, contains a single class, also named **ScriptMain**. The code elements in the class vary depending on the programming language that you selected for the Script task:

- When the Script task is configured for the Visual Basic 2010 programming language, the **ScriptMain** class has a public subroutine, **Main**. The **ScriptMain.Main** subroutine is the method that the runtime calls when you run your Script task.

  By default, the only code in the **Main** subroutine of a new script is the line `Dts.TaskResult = ScriptResults.Success`. This line informs the runtime that the task was successful in its operation. The **Dts.TaskResult** property is discussed in Referencing Other Assemblies in Scripting Solutions (VSTA)/Execution Value.
- When the Script task is configured for the Visual C# programming language, the **ScriptMain** class has a public method, **Main**. The method is called when the Script task runs.

  By default, the **Main** method includes the line `Dts.TaskResult = (int)ScriptResults.Success`. This line informs the runtime that the task was successful in its operation.

The **ScriptMain** item can contain classes other than the **ScriptMain** class. Classes are available only to the Script task in which they reside.

By default, the **ScriptMain** project item contains the following autogenerated code. The code template also provides an overview of the Script task, and additional information on how to retrieve and manipulate SSIS objects, such as variables, events, and connections.

```
' Microsoft SQL Server Integration Services Script Task
' Write scripts using Microsoft Visual Basic 2008.
' The ScriptMain is the entry point class of the script.

Imports System
Imports System.Data
Imports System.Math
Imports Microsoft.SqlServer.Dts.Runtime.VSTAProxy
```

```
<System.AddIn.AddIn("ScriptMain", Version:="1.0", Publisher:="",
Description:="")> _
Partial Class ScriptMain

Private Sub ScriptMain_Startup(ByVal sender As Object, ByVal e As
System.EventArgs) Handles Me.Startup

End Sub

Private Sub ScriptMain_Shutdown(ByVal sender As Object, ByVal e As
System.EventArgs) Handles Me.Shutdown
Try
' Unlock variables from the read-only and read-write variable collection
properties
If (Dts.Variables.Count <> 0) Then
Dts.Variables.Unlock()
End If
Catch ex As Exception
        End Try
End Sub

Enum ScriptResults
Success = DTSExecResult.Success
Failure = DTSExecResult.Failure
End Enum



' The execution engine calls this method when the task executes.
' To access the object model, use the Dts property. Connections, variables,
events,
' and logging features are available as members of the Dts property as shown
in the following examples.
'
' To reference a variable, call
Dts.Variables("MyCaseSensitiveVariableName").Value
```

```
' To post a log entry, call Dts.Log("This is my log text", 999, Nothing)
' To fire an event, call Dts.Events.FireInformation(99, "test", "hit the help
message", "", 0, True)
'
' To use the connections collection use something like the following:
' ConnectionManager cm = Dts.Connections.Add("OLEDB")
' cm.ConnectionString = "Data Source=localhost;Initial
Catalog=AdventureWorks;Provider=SQLNCLI10;Integrated Security=SSPI;Auto
Translate=False;"
'
' Before returning from this method, set the value of Dts.TaskResult to
indicate success or failure.
'
' To open Help, press F1.

Public Sub Main()
'
' Add your code here
'
Dts.TaskResult = ScriptResults.Success
End Sub

End Class
/*
   Microsoft SQL Server Integration Services Script Task
   Write scripts using Microsoft Visual C# 2008.
   The ScriptMain is the entry point class of the script.
*/

using System;
using System.Data;
using Microsoft.SqlServer.Dts.Runtime.VSTAProxy;
using System.Windows.Forms;
```

```
namespace ST_1bcfdbad36d94f8ba9f23a10375abe53.csproj
{
    [System.AddIn.AddIn("ScriptMain", Version = "1.0", Publisher = "",
Description = "")]
    public partial class ScriptMain
    {
        private void ScriptMain_Startup(object sender, EventArgs e)
        {

        }

        private void ScriptMain_Shutdown(object sender, EventArgs e)
        {
            try
            {
                // Unlock variables from the read-only and read-write
variable collection properties
                if (Dts.Variables.Count != 0)
                {
                    Dts.Variables.Unlock();
                }
            }
            catch
            {
            }
        }

        #region VSTA generated code
        private void InternalStartup()
        {
            this.Startup += new System.EventHandler(ScriptMain_Startup);
            this.Shutdown += new System.EventHandler(ScriptMain_Shutdown);
        }
        enum ScriptResults
```

```
        {
            Success = DTSExecResult.Success,
            Failure = DTSExecResult.Failure
        };

        #endregion

        /*
The execution engine calls this method when the task executes.
To access the object model, use the Dts property. Connections, variables,
events,
and logging features are available as members of the Dts property as shown in
the following examples.

To reference a variable, call
Dts.Variables["MyCaseSensitiveVariableName"].Value;
To post a log entry, call Dts.Log("This is my log text", 999, null);
To fire an event, call Dts.Events.FireInformation(99, "test", "hit the help
message", "", 0, true);

To use the connections collection use something like the following:
ConnectionManager cm = Dts.Connections.Add("OLEDB");
cm.ConnectionString = "Data Source=localhost;Initial
Catalog=AdventureWorks;Provider=SQLNCLI10;Integrated Security=SSPI;Auto
Translate=False;";

Before returning from this method, set the value of Dts.TaskResult to
indicate success or failure.

To open Help, press F1.
*/

        public void Main()
        {
            // TODO: Add your code here
```

```
            Dts.TaskResult = (int)ScriptResults.Success;
        }
    }
```

Additional Project Items in the Script Task Project

The Script task project can include items other than the default **ScriptMain** item. You can add classes, modules, and code files to the project. You can also use folders to organize groups of items. All the items that you add are persisted inside the package.

References in the Script Task Project

You can add references to managed assemblies by right-clicking the Script task project in **Project Explorer**, and then clicking **Add Reference**. For more information, see Referencing Other Assemblies in Scripting Solutions.

Note
You can view project references in the VSTA IDE in **Class View** or in **Project Explorer**. You open either of these windows from the **View** menu. You can add a new reference from the **Project** menu, from **Project Explorer**, or from **Class View**.

Interacting with the Package in the Script Task

The Script task uses the global **Dts** object, which is an instance of the **T:Microsoft.SqlServer.Dts.Tasks.ScriptTask.ScriptObjectModel** class, and its members to interact with the containing package and with the Integration Services runtime.

The following table lists the principal public members of the **T:Microsoft.SqlServer.Dts.Tasks.ScriptTask.ScriptObjectModel** class, which is exposed to Script task code through the global **Dts** object. The topics in this section discuss the use of these members in more detail.

Member	Purpose
P:Microsoft.SqlServer.Dts.Tasks.ScriptTask.ScriptObjectModel.Connections	Provides access to connection managers defined in the package.
P:Microsoft.SqlServer.Dts.Tasks.ScriptTask.ScriptObjectModel.Events	Provides an events interface to let the Script task raise errors, warnings, and informational messages.
P:Microsoft.SqlServer.Dts.Tasks.ScriptTask.ScriptObjectModel.ExecutionValue	Provides a simple way to return a single object to the runtime (in addition to the **TaskResult**) that can also be used for workflow branching.
M:Microsoft.SqlServer.Dts.Tasks.ScriptTask.ScriptObjectModel.Log(System.String, System.Int32,System.Byte[])	Logs information such as task progress and results to enabled log providers.

Member	Purpose
P:Microsoft.SqlServer.Dts.Tasks.ScriptTask.ScriptObjectModel.TaskResult	Reports the success or failure of the task.
P:Microsoft.SqlServer.Dts.Tasks.ScriptTask.ScriptObjectModel.Transaction	Provides the transaction, if any, within which the task's container is running.
P:Microsoft.SqlServer.Dts.Tasks.ScriptTask.ScriptObjectModel.Variables	Provides access to the variables listed in the **ReadOnlyVariables** and **ReadWriteVariables** task properties for use within the script.

The **T:Microsoft.SqlServer.Dts.Tasks.ScriptTask.ScriptObjectModel** class also contains some public members that you will probably not use.

Member	Description
P:Microsoft.SqlServer.Dts.Tasks.ScriptTask.ScriptObjectModel.VariableDispenser	The **P:Microsoft.SqlServer.Dts.Tasks.ScriptTask.ScriptObjectModel.Variables** property provides more convenient access to variables. Although you can use the **P:Microsoft.SqlServer.Dts.Tasks.ScriptTask.ScriptObjectModel.VariableDispenser**, you must explicitly call methods to lock and unlock variables for reading and writing. The Script task handles locking semantics for you when you use the **P:Microsoft.SqlServer.Dts.Tasks.ScriptTask.ScriptObjectModel.Variables** property.

Debugging the Script Task

To debug the code in your Script task, set at least one breakpoint in the code, and then close the VSTA IDE to run the package in SQL Server Data Tools (SSDT). When package execution enters the Script task, the VSTA IDE reopens and displays your code in read-only mode. After execution reaches your breakpoint, you can examine variable values and step through the remaining code.

Warning

You can debug the Script task when you run the package in 64-bit mode.

Note

You must execute the package to debug into your Script task. If you execute only the individual task, breakpoints in the Script task code are ignored.

Note
You cannot debug a Script task when you run the Script task as part of a child package that is run from an Execute Package task. Breakpoints that you set in the Script task in the child package are disregarded in these circumstances. You can debug the child package normally by running it separately.

Note
When you debug a package that contains multiple Script tasks, the debugger debugs one Script task. The system can debug another Script task if the debugger completes, as in the case of a Foreach Loop or For Loop container.

External Resources

- Blog entry, VSTA setup and configuration troubles for SSIS 2008 and R2 installations, on blogs.msdn.com.

Stay Up to Date with Integration Services

For the latest downloads, articles, samples, and videos from Microsoft, as well as selected solutions from the community, visit the Integration Services page on MSDN:

- Visit the Integration Services page on MSDN

For automatic notification of these updates, subscribe to the RSS feeds available on the page.

See Also

Referencing Other Assemblies in Scripting Solutions
Configuring the Script Task in the Script Task Editor

Using Variables in the Script Task

Variables make it possible for the Script task to exchange data with other objects in the package. For more information, see Integration Services Variables.

The Script task uses the **P:Microsoft.SqlServer.Dts.Tasks.ScriptTask.ScriptObjectModel.Variables** property of the **Dts** object to read from and write to **T:Microsoft.SqlServer.Dts.Runtime.Variable** objects in the package.

Note
The **P:Microsoft.SqlServer.Dts.Runtime.Variable.Value** property of the **T:Microsoft.SqlServer.Dts.Runtime.Variable** class is of type **Object**. Because the Script task has **Option Strict** enabled, you must cast the **P:Microsoft.SqlServer.Dts.Runtime.Variable.Value** property to the appropriate type before you can use it.

You add existing variables to the **P:Microsoft.SqlServer.Dts.Tasks.ScriptTask.ScriptTask.ReadOnlyVariables** and **P:Microsoft.SqlServer.Dts.Tasks.ScriptTask.ScriptTask.ReadWriteVariables** lists in the **Script Task Editor** to make them available to the custom script. Keep in mind that variable names are case-sensitive. Within the script, you access variables of both types through the **P:Microsoft.SqlServer.Dts.Tasks.ScriptTask.ScriptObjectModel.Variables** property of the **Dts** object. Use the **Value** property to read from and write to individual variables. The Script task transparently manages locking as the script reads and modifies the values of variables.

You can use the **M:Microsoft.SqlServer.Dts.Runtime.Variables.Contains(System.Object)** method of the **T:Microsoft.SqlServer.Dts.Runtime.Variables** collection returned by the **P:Microsoft.SqlServer.Dts.Tasks.ScriptTask.ScriptObjectModel.Variables** property to check for the existence of a variable before using it in your code.

You can also use the **P:Microsoft.SqlServer.Dts.Tasks.ScriptTask.ScriptObjectModel.VariableDispenser** property (**Dts.VariableDispenser**) to work with variables in the Script task. When using the **P:Microsoft.SqlServer.Dts.Tasks.ScriptTask.ScriptObjectModel.VariableDispenser**, you must handle both the locking semantics and the casting of data types for variable values in your own code. You may need to use the **P:Microsoft.SqlServer.Dts.Tasks.ScriptTask.ScriptObjectModel.VariableDispenser** property instead of the **P:Microsoft.SqlServer.Dts.Tasks.ScriptTask.ScriptObjectModel.Variables** property if you want to work with a variable that is not available at design time but is created programmatically at run time.

Using the Script Task within a Foreach Loop Container

When a Script task runs repeatedly within a Foreach Loop container, the script usually needs to work with the contents of the current item in the enumerator. For example, when using a Foreach File enumerator, the script needs to know the current file name; when using a Foreach ADO enumerator, the script needs to know the contents of the columns in the current row of data.

Variables make this communication between the Foreach Loop container and the Script task possible. On the **Variable Mappings** page of the **Foreach Loop Editor**, assign variables to each item of data that is returned by a single enumerated item. For example, a Foreach File

enumerator returns only a file name at Index 0 and therefore requires only one variable mapping, whereas an enumerator that returns several columns of data in each row requires you to map a different variable to each column that you want to use in the Script task.

After you have mapped enumerated items to variables, then you must add the mapped variables to the **ReadOnlyVariables** property on the **Script** page of the **Script Task Editor** to make them available to your script. For an example of a Script task within a Foreach Loop container that processes the image files in a folder, see Script Task Example: Working with Images.

Variables Example

The following example demonstrates how to access and use variables in a Script task to determine the path of package workflow. The sample assumes that you have created integer variables named `CustomerCount` and `MaxRecordCount` and added them to the **ReadOnlyVariables** collection in the **Script Task Editor**. The `CustomerCount` variable contains the number of customer records to be imported. If its value is greater than the value of `MaxRecordCount`, the Script task reports failure. When a failure occurs because the `MaxRecordCount` threshold has been exceeded, the error path of the workflow can implement any required clean-up.

To successfully compile the sample, you need to add a reference to the Microsoft.SqlServer.ScriptTask assembly.

```
Public Sub Main()

    Dim customerCount As Integer
    Dim maxRecordCount As Integer

    If Dts.Variables.Contains("CustomerCount") = True AndAlso _
        Dts.Variables.Contains("MaxRecordCount") = True Then

        customerCount = _
            CType(Dts.Variables("CustomerCount").Value, Integer)
        maxRecordCount = _
            CType(Dts.Variables("MaxRecordCount").Value, Integer)

    End If

    If customerCount > maxRecordCount Then
            Dts.TaskResult = ScriptResults.Failure
```

```
    Else
            Dts.TaskResult = ScriptResults.Success
    End If

End Sub
using System;
using System.Data;
using Microsoft.SqlServer.Dts.Runtime;

public class ScriptMain
{



    public void Main()
    {
        int customerCount;
        int maxRecordCount;

        if
(Dts.Variables.Contains("CustomerCount")==true&&Dts.Variables.Contains("MaxRe
cordCount")==true)

        {
            customerCount = (int) Dts.Variables["CustomerCount"].Value;
            maxRecordCount = (int) Dts.Variables["MaxRecordCount"].Value;

        }

        if (customerCount>maxRecordCount)
        {
            Dts.TaskResult = (int)ScriptResults.Failure;
        }
        else
```

```
        {
            Dts.TaskResult = (int)ScriptResults.Success;
        }

    }

}
```

Stay Up to Date with Integration Services

For the latest downloads, articles, samples, and videos from Microsoft, as well as selected solutions from the community, visit the Integration Services page on MSDN:

Visit the Integration Services page on MSDN

For automatic notification of these updates, subscribe to the RSS feeds available on the page.

See Also

Integration Services Variables

Using Variables in Packages

Connecting to Data Sources in the Script Task

Connection managers provide access to data sources that have been configured in the package. For more information, see Integration Services Connections.

The Script task can access these connection managers through the **P:Microsoft.SqlServer.Dts.Tasks.ScriptTask.ScriptObjectModel.Connections** property of the **Dts** object. Each connection manager in the **T:Microsoft.SqlServer.Dts.Runtime.Connections** collection stores information about how to connect to the underlying data source.

When you call the **M:Microsoft.SqlServer.Dts.Runtime.ConnectionManager.AcquireConnection(System.Object)** method of a connection manager, the connection manager connects to the data source, if it is not already connected, and returns the appropriate connection or connection information for you to use in your Script task code.

Note
You must know the type of connection returned by the connection manager before calling **AcquireConnection**. Because the Script task has **Option Strict** enabled, you must cast the connection, which is returned as type **Object**, to the appropriate connection type before you can use it.

You can use the **M:Microsoft.SqlServer.Dts.Runtime.Connections.Contains(System.Object)** method of the **T:Microsoft.SqlServer.Dts.Runtime.Connections** collection returned by the **P:Microsoft.SqlServer.Dts.Tasks.ScriptTask.ScriptObjectModel.Connections** property to look for an existing connection before using the connection in your code.

Important

- You cannot call the **AcquireConnection** method of connection managers that return unmanaged objects, such as the OLE DB connection manager and the Excel connection manager, in the managed code of a Script task. However, you can read the **ConnectionString** property of these connection managers, and connect to the data source directly in your code by using the connection string with an **OledbConnection** from the **System.Data.OleDb** namespace.
- If you must call the **AcquireConnection** method of a connection manager that returns an unmanaged object, use an ADO.NET connection manager. When you configure the ADO.NET connection manager to use an OLE DB provider, it connects by using the .NET Framework Data Provider for OLE DB. In this case, the **AcquireConnection** method returns a **System.Data.OleDb.OleDbConnection** instead of an unmanaged object. To configure an ADO.NET connection manager for use with an Excel data source, select the Microsoft OLE DB Provider for Jet, specify an Excel file, and enter `Excel 8.0` (for Excel 97 and later) as the value of **Extended Properties** on the **All** page of the **Connection Manager** dialog box.

Connections Example

The following example demonstrates how to access connection managers from within the Script task. The sample assumes that you have created and configured an ADO.NET connection manager named **Test ADO.NET Connection** and a Flat File connection manager named **Test Flat File Connection**. Note that the ADO.NET connection manager returns a **SqlConnection** object that you can use immediately to connect to the data source. The Flat File connection manager, on the other hand, returns only a string that contains the path and file name. You must use methods from the **System.IO** namespace to open and work with the flat file.

```
Public Sub Main()
```

```
    Dim myADONETConnection As SqlClient.SqlConnection
    myADONETConnection = _
        DirectCast(Dts.Connections("Test ADO.NET
Connection").AcquireConnection(Dts.Transaction), _
        SqlClient.SqlConnection)
    MsgBox(myADONETConnection.ConnectionString, _
        MsgBoxStyle.Information, "ADO.NET Connection")

    Dim myFlatFileConnection As String
    myFlatFileConnection = _
        DirectCast(Dts.Connections("Test Flat File
Connection").AcquireConnection(Dts.Transaction), _
        String)
    MsgBox(myFlatFileConnection, MsgBoxStyle.Information, "Flat File
Connection")

    Dts.TaskResult = ScriptResults.Success

End Sub
using System;
using System.Data.SqlClient;
using Microsoft.SqlServer.Dts.Runtime;
using System.Windows.Forms;

public class ScriptMain
{



        public void Main()
        {
            SqlConnection myADONETConnection = new SqlConnection();
            myADONETConnection = (SqlConnection)(Dts.Connections["Test
ADO.NET Connection"].AcquireConnection(Dts.Transaction)as SqlConnection);
```

```
            MessageBox.Show(myADONETConnection.ConnectionString, "ADO.NET
Connection");

            string myFlatFileConnection;
            myFlatFileConnection = (string)(Dts.Connections["Test Flat File
Connection"].AcquireConnection(Dts.Transaction) as String);
            MessageBox.Show(myFlatFileConnection, "Flat File Connection");

            Dts.TaskResult = (int)ScriptResults.Success;

        }

}
```

Stay Up to Date with Integration Services

For the latest downloads, articles, samples, and videos from Microsoft, as well as selected solutions from the community, visit the Integration Services page on MSDN:

Visit the Integration Services page on MSDN

For automatic notification of these updates, subscribe to the RSS feeds available on the page.

See Also

Integration Services Connections

Creating Connection Managers

Raising Events in the Script Task

Events provide a way to report errors, warnings, and other information, such as task progress or status, to the containing package. The package provides event handlers for managing event notifications. The Script task can raise events by calling methods on the **P:Microsoft.SqlServer.Dts.Tasks.ScriptTask.ScriptObjectModel.Events** property of the **Dts** object. For more information about how Integration Services packages handle events, see DTS Event Handlers.

Events can be logged to any log provider that is enabled in the package. Log providers store information about events in a data store. The Script task can also use the **M:Microsoft.SqlServer.Dts.Tasks.ScriptTask.ScriptObjectModel.Log(System.String,System.Int32,System.Byte[])** method to log information to a log provider without raising an event. For more information about how to use the **M:Microsoft.SqlServer.Dts.Tasks.ScriptTask.ScriptObjectModel.Log(System.String,System.Int32,System.Byte[])** method, see Logging.

To raise an event, the Script task calls one of the methods exposed by the **P:Microsoft.SqlServer.Dts.Tasks.ScriptTask.ScriptObjectModel.Events** property. The following table lists the methods exposed by the **P:Microsoft.SqlServer.Dts.Tasks.ScriptTask.ScriptObjectModel.Events** property.

Event	Description
M:Microsoft.SqlServer.Dts.Runtime.IDTSComponentEvents.FireCustomEvent(System.String,System.String,System.Object[]@,System.String,System.Boolean@)	Raises a user-defined custom event in the package.
M:Microsoft.SqlServer.Dts.Runtime.IDTSComponentEvents.FireError(System.Int32,System.String,System.String,System.String,System.Int32)	Informs the package of an error condition.
M:Microsoft.SqlServer.Dts.Runtime.IDTSComponentEvents.FireInformation(System.Int32,System.String,System.String,System.String,System.Int32,System.Boolean@)	Provides information to the user.
M:Microsoft.SqlServer.Dts.Runtime.IDTSComponentEvents.FireProgress(System.String,System.Int32,System.Int32,System.Int32,System.String,System.Boolean@)	Informs the package of the progress of the task.
M:Microsoft.SqlServer.Dts.Runtime.IDTSComponentEvents.FireQueryCancel	Returns a value that indicates whether the package needs the task to shut down prematurely.

Event	Description
M:Microsoft.SqlServer.Dts.Runtime.IDTSComponentEvents.FireWarning(System.Int32,System.String,System.String,System.String,System.Int32)	Informs the package that the task is in a state that warrants user notification, but is not an error condition.

Events Example

The following example demonstrates how to raise events from within the Script task. The example uses a native Windows API function to determine whether an Internet connection is available. If no connection is available, it raises an error. If a potentially volatile modem connection is in use, the example raises a warning. Otherwise, it returns an informational message that an Internet connection has been detected.

```
Private Declare Function InternetGetConnectedState Lib "wininet" _
    (ByRef dwFlags As Long, ByVal dwReserved As Long) As Long


Private Enum ConnectedStates
    LAN = &H2
    Modem = &H1
    Proxy = &H4
    Offline = &H20
    Configured = &H40
    RasInstalled = &H10
End Enum


Public Sub Main()


    Dim dwFlags As Long
    Dim connectedState As Long
    Dim fireAgain as Boolean


    connectedState = InternetGetConnectedState(dwFlags, 0)


    If connectedState <> 0 Then
        If (dwFlags And ConnectedStates.Modem) = ConnectedStates.Modem Then
```

```
            Dts.Events.FireWarning(0, "Script Task Example", _
                "Volatile Internet connection detected.", String.Empty, 0)
        Else
            Dts.Events.FireInformation(0, "Script Task Example", _
                "Internet connection detected.", String.Empty, 0, fireAgain)
        End If
    Else
        ' If not connected to the Internet, raise an error.
        Dts.Events.FireError(0, "Script Task Example", _
            "Internet connection not available.", String.Empty, 0)
    End If

    Dts.TaskResult = ScriptResults.Success

End Sub
using System;
using System.Data;
using Microsoft.SqlServer.Dts.Runtime;
using System.Windows.Forms;
using System.Runtime.InteropServices;

public class ScriptMain
{



[DllImport("wininet")]
        private extern static long InternetGetConnectedState(ref long
dwFlags, long dwReserved);

        private enum ConnectedStates
        {
            LAN = 0x2,
            Modem = 0x1,
            Proxy = 0x4,
```

```
            Offline = 0x20,
            Configured = 0x40,
            RasInstalled = 0x10
        };

        public void Main()
        {
            //
            long dwFlags = 0;
            long connectedState;
            bool fireAgain = true;
            int state;

            connectedState = InternetGetConnectedState(ref dwFlags, 0);
            state = (int)ConnectedStates.Modem;
            if (connectedState != 0)
            {
                if ((dwFlags & state) == state)
                {
                    Dts.Events.FireWarning(0, "Script Task Example",
"Volatile Internet connection detected.", String.Empty, 0);
                }
                else
                {
                    Dts.Events.FireInformation(0, "Script Task Example",
"Internet connection detected.", String.Empty, 0, ref fireAgain);
                }
            }
            else
            {
                // If not connected to the Internet, raise an error.
                Dts.Events.FireError(0, "Script Task Example", "Internet
connection not available.", String.Empty, 0);
            }
```

```
            Dts.TaskResult = (int)ScriptResults.Success;

        }
```

Stay Up to Date with Integration Services

For the latest downloads, articles, samples, and videos from Microsoft, as well as selected solutions from the community, visit the Integration Services page on MSDN:

Visit the Integration Services page on MSDN

For automatic notification of these updates, subscribe to the RSS feeds available on the page.

See Also

Integration Services Event Handlers

Add an Event Handler to a Package

Logging in the Script Task

The use of logging in Integration Services packages lets you record detailed information about execution progress, results, and problems by recording predefined events or user-defined messages for later analysis. The Script task can use the **M:Microsoft.SqlServer.Dts.Tasks.ScriptTask.ScriptObjectModel.Log(System.String,System.Int32,System.Byte[])** method of the **Dts** object to log user-defined data. If logging is enabled, and the **ScriptTaskLogEntry** event is selected for logging on the **Details** tab of the **Configure SSIS Logs** dialog box, a single call to the **M:Microsoft.SqlServer.Dts.Tasks.ScriptTask.ScriptObjectModel.Log(System.String,System.Int32,System.Byte[])** method stores the event information in all the log providers configured for the task.

Note

Although you can perform logging directly from your Script task, you may want to consider implementing events rather than logging. When using events, not only can you enable the logging of event messages, but you can also respond to the event with default or user-defined event handlers.

For more information about logging, see Integration Services Logging.

Logging Example

The following example demonstrates logging from the Script task by logging a value that represents the number of rows processed.

```
Public Sub Main()

    Dim rowsProcessed As Integer = 100
    Dim emptyBytes(0) As Byte

    Try
        Dts.Log("Rows processed: " & rowsProcessed.ToString, _
            0, _
            emptyBytes)
        Dts.TaskResult = ScriptResults.Success
    Catch ex As Exception
        'An error occurred.
        Dts.Events.FireError(0, "Script Task Example", _
            ex.Message & ControlChars.CrLf & ex.StackTrace, _
            String.Empty, 0)
        Dts.TaskResult = ScriptResults.Failure
    End Try

End Sub
using System;
using System.Data;
using Microsoft.SqlServer.Dts.Runtime;

public class ScriptMain
{
```

```
    public void Main()
        {
            //
            int rowsProcessed = 100;
            byte[] emptyBytes = new byte[0];

            try
            {
                Dts.Log("Rows processed: " + rowsProcessed.ToString(), 0,
emptyBytes);
                Dts.TaskResult = (int)ScriptResults.Success;
            }
            catch (Exception ex)
            {
                //An error occurred.
                Dts.Events.FireError(0, "Script Task Example", ex.Message +
"\r" + ex.StackTrace, String.Empty, 0);
                Dts.TaskResult = (int)ScriptResults.Failure;
            }

        }
}
```

External Resources

- Blog entry, Logging custom events for Integration Services tasks, on dougbert.com

Stay Up to Date with Integration Services

For the latest downloads, articles, samples, and videos from Microsoft, as well as selected solutions from the community, visit the Integration Services page on MSDN:

- Visit the Integration Services page on MSDN

For automatic notification of these updates, subscribe to the RSS feeds available on the page.

See Also

Integration Services Logging

Returning Results from the Script Task

The Script task uses the **P:Microsoft.SqlServer.Dts.Tasks.ScriptTask.ScriptObjectModel.TaskResult** and the optional **P:Microsoft.SqlServer.Dts.Tasks.ScriptTask.ScriptObjectModel.ExecutionValue** properties to return status information to the Integration Services runtime that can be used to determine the path of the workflow after the Script task has finished.

TaskResult

The **P:Microsoft.SqlServer.Dts.Tasks.ScriptTask.ScriptObjectModel.TaskResult** property reports whether the task succeeded or failed. For example:

```
Dts.TaskResult = ScriptResults.Success
```

ExecutionValue

The **P:Microsoft.SqlServer.Dts.Tasks.ScriptTask.ScriptObjectModel.ExecutionValue** property optionally returns a user-defined object that quantifies or provides more information about the success or failure of the Script task. For example, the FTP task uses the **P:Microsoft.SqlServer.Dts.Tasks.ScriptTask.ScriptObjectModel.ExecutionValue** property to return the number of files transferred. The Execute SQL task returns the number of rows affected by the task. The **P:Microsoft.SqlServer.Dts.Tasks.ScriptTask.ScriptObjectModel.ExecutionValue** can also be used to determine the path of the workflow. For example:

```
Dim rowsAffected as Integer
```

```
...
rowsAffected = 1000
Dts.ExecutionValue = rowsAffected
```

Stay Up to Date with Integration Services

For the latest downloads, articles, samples, and videos from Microsoft, as well as selected solutions from the community, visit the Integration Services page on MSDN:

Visit the Integration Services page on MSDN

For automatic notification of these updates, subscribe to the RSS feeds available on the page.

Script Task Examples

The Script task is a multi-purpose tool that you can use in a package to fill almost any requirement that is not met by the tasks included with Integration Services. This topic lists Script task code samples that demonstrate some of the available functionality.

Note

If you want to create tasks that you can more easily reuse across multiple packages, consider using the code in these Script task samples as the starting point for custom tasks. For more information, see Extending the Package with Custom Tasks.

In This Section

Example Topics

This section contains code examples that demonstrate various uses of the .NET Framework classes that you can incorporate into an Integration Services Script task:

Script Task Example: Detecting an Empty Flat File

Checks a flat file to determine whether it contains rows of data, and saves the result to a variable for use in control flow branching.

Script Task Example: Gathering a List for the ForEach Enumerator

Gathers a list of files that meet user-specified criteria, and populates a variable for later use by the Foreach from Variable Enumerator.

Script Task Example: Querying the Active Directory

Retrieves user information from Active Directory based on the value of an Integration Services variable, by using classes in the **System.DirectoryServices** namespace.

Script Task Example: Monitoring Performance Counters

Creates a custom performance counter that can be used to track the execution progress of an Integration Services package, by using classes in the **System.Diagnostics** namespace.

Script Task Example: Working with Images

Compresses images into the JPEG format and creates thumbnail images from them, by using classes in the **System.Drawing** namespace.

Script Task Example: Finding Installed Printers

Locates installed printers that support a specific paper size, by using classes in the **System.Drawing.Printing** namespace.

Sending an HTML Mail Message with the Script Task

Sends a mail message in HTML format instead of plain text format.

Listing Excel Worksheets with the Script Task

Lists the worksheets in an Excel file and checks for the existence of a specific worksheet.

Sending to a Remote Private Message Queue with the Script Task

Sends a message to a remote private message queue.

Other Examples

The following topics also contain code examples for use with the Script task:

Variables in the Script Task

Asks the user for confirmation of whether the package should continue to run, based on the value of a package variable that may exceed the limit specified in another variable.

Connections in the Script Task

Retrieves a connection or connection information from connection managers defined in the package.

Events in the Script Task

Raises an error, a warning, or an informational message based on the status of the Internet connection on the server.

Logging in the Script Task

Logs the number of items processed by the task to enabled log providers.

Stay Up to Date with Integration Services

For the latest downloads, articles, samples, and videos from Microsoft, as well as selected solutions from the community, visit the Integration Services page on MSDN:

Visit the Integration Services page on MSDN

For automatic notification of these updates, subscribe to the RSS feeds available on the page.

Detecting an Empty Flat File with the Script Task

The Flat File source does not determine whether a flat file contains rows of data before attempting to process it. You may want to improve the efficiency of a package, especially of a package that iterates over numerous flat files, by skipping files that do not contain any rows of data. The Script task can look for an empty flat file before the package begins to process the data flow.

Note

If you want to create a task that you can more easily reuse across multiple packages, consider using the code in this Script task sample as the starting point for a custom task. For more information, see Extending the Package with Custom Tasks.

Description

The following example uses methods from the **System.IO** namespace to test the flat file specified in a Flat File connection manager to determine whether the file is empty, or whether it contains only expected non-data rows such as column headers or an empty line. The script checks the size of the file first; if the size is zero bytes, the file is empty. If the file size is greater than zero, the script reads lines from the file until there are no more lines, or until the number of lines exceeds the expected number of non-data rows. If the number of lines in the file is less than or equal to the expected number of non-data rows, then the file is considered empty. The result is returned as a Boolean value in a user variable, the value of which can be used for branching in the package's control flow. The **FireInformation** method also displays the result in the **Output** window of the Microsoft Visual Studio Tools for Applications (VSTA).

To configure this Script Task example

1. Create and configure a flat file connection manager named **EmptyFlatFileTest**.
2. Create an integer variable named `FFNonDataRows` and set its value to the number of non-data rows expected in the flat file.
3. Create a Boolean variable named `FFIsEmpty`.
4. Add the `FFNonDataRows` variable to the Script task's **ReadOnlyVariables** property.
5. Add the `FFIsEmpty` variable to the Script task's **ReadWriteVariables** property.
6. In your code, import the **System.IO** namespace.

If you are iterating over files with a Foreach File enumerator, instead of using a single Flat File connection manager, you will need to modify the sample code below to obtain the file name and path from the variable in which the enumerated value is stored instead of from the connection manager.

Code

```
  Public Sub Main()

    Dim nonDataRows As Integer = _
        DirectCast(Dts.Variables("FFNonDataRows").Value, Integer)
    Dim ffConnection As String = _

DirectCast(Dts.Connections("EmptyFlatFileTest").AcquireConnection(Nothing), _
        String)
    Dim flatFileInfo As New FileInfo(ffConnection)
    ' If file size is 0 bytes, flat file does not contain data.
    Dim fileSize As Long = flatFileInfo.Length
    If fileSize > 0 Then
      Dim lineCount As Integer = 0
      Dim line As String
      Dim fsFlatFile As New StreamReader(ffConnection)
      Do Until fsFlatFile.EndOfStream
        line = fsFlatFile.ReadLine
        lineCount += 1
        ' If line count > expected number of non-data rows,
        '  flat file contains data (default value).
        If lineCount > nonDataRows Then
```

```
          Exit Do
        End If
        ' If line count <= expected number of non-data rows,
        '  flat file does not contain data.
        If lineCount <= nonDataRows Then
          Dts.Variables("FFIsEmpty").Value = True
        End If
      Loop
    Else
      Dts.Variables("FFIsEmpty").Value = True
    End If

    Dim fireAgain As Boolean = False
    Dts.Events.FireInformation(0, "Script Task", _
        String.Format("{0}: {1}", ffConnection, _
        Dts.Variables("FFIsEmpty").Value.ToString), _
        String.Empty, 0, fireAgain)

    Dts.TaskResult = ScriptResults.Success

  End Sub
public void Main()
        {

            int nonDataRows = (int)(Dts.Variables["FFNonDataRows"].Value);
            string ffConnection =
(string)(Dts.Connections["EmptyFlatFileTest"].AcquireConnection(null) as
String);
            FileInfo flatFileInfo = new FileInfo(ffConnection);
            // If file size is 0 bytes, flat file does not contain data.
            long fileSize = flatFileInfo.Length;
            if (fileSize > 0)
            {
```

```
            int lineCount = 0;
            string line;
            StreamReader fsFlatFile = new StreamReader(ffConnection);
            while (!(fsFlatFile.EndOfStream))
            {
                Console.WriteLine (fsFlatFile.ReadLine());
                lineCount += 1;
                // If line count > expected number of non-data rows,
                //  flat file contains data (default value).
                if (lineCount > nonDataRows)
                {
                    break;
                }
                // If line count <= expected number of non-data rows,
                //  flat file does not contain data.
                if (lineCount <= nonDataRows)
                {
                    Dts.Variables["FFIsEmpty"].Value = true;
                }
            }
        }
        else
        {
            Dts.Variables["FFIsEmpty"].Value = true;
        }

        bool fireAgain = false;
        Dts.Events.FireInformation(0, "Script Task", String.Format("{0}:
{1}", ffConnection, Dts.Variables["FFIsEmpty"].Value), String.Empty, 0, ref
fireAgain);

        Dts.TaskResult = (int)ScriptResults.Success;
```

```
}
```

Stay Up to Date with Integration Services

For the latest downloads, articles, samples, and videos from Microsoft, as well as selected solutions from the community, visit the Integration Services page on MSDN:

Visit the Integration Services page on MSDN

For automatic notification of these updates, subscribe to the RSS feeds available on the page.

See Also

Script Task Examples

Gathering a List for the ForEach Loop with the Script Task

The Foreach from Variable Enumerator enumerates over the items in a list that is passed to it in a variable and performs the same tasks on each item. You can use custom code in a Script task to populate a list for this purpose. For more information about the enumerator, see Foreach Loop Container.

Note

If you want to create a task that you can more easily reuse across multiple packages, consider using the code in this Script task sample as the starting point for a custom task. For more information, see Extending the Package with Custom Tasks.

Description

The following example uses methods from the **System.IO** namespace to gather a list of Excel workbooks on the computer that are either newer or older than a number of days specified by the user in a variable. It searches directories on Drive C recursively for files that have the .xls extension and examines the date on which each file was last modified to determine whether the file belongs in the list. It adds qualifying files to an **ArrayList** and saves the **ArrayList** to a

variable for later use in a Foreach Loop container. The Foreach Loop container is configured to use the Foreach from Variable enumerator.

Note
The variable that you use with the Foreach from Variable Enumerator must be of type **Object**. The object that you place in the variable must implement one of the following interfaces: **System.Collections.IEnumerable**, **System.Runtime.InteropServices.ComTypes.IEnumVARIANT**, **System.ComponentModel IListSource**, or **Microsoft.SqlServer.Dts.Runtime.Wrapper.ForEachEnumeratorHost**. An **Array** or **ArrayList** is commonly used. The **ArrayList** requires a reference and an **Imports** statement for the **System.Collections** namespace.

You can experiment with this task by using different positive and negative values for the `FileAge` package variable. For example, you can enter 5 to search for files created in the last five days, or enter -3 to search for files that were created more than three days ago. This task may take a minute or two on a drive with many folders to search.

To configure this Script Task example

1. Create a package variable named `FileAge` of type integer and enter a positive or negative integer value. When the value is positive, the code searches for files newer than the specified number of days; when negative, for files older than the specified number of days.
2. Create a package variable named `FileList` of type **Object** to receive the list of files gathered by the Script task for later use by the Foreach from Variable Enumerator.
3. Add the `FileAge` variable to the Script task's **ReadOnlyVariables** property, and add the `FileList` variable to the **ReadWriteVariables** property.
4. In your code, import the **System.Collections** and the **System.IO** namespaces.

Code

```
Imports System

Imports System.Data

Imports System.Math

Imports Microsoft.SqlServer.Dts.Runtime

Imports System.Collections

Imports System.IO


Public Class ScriptMain


  Private Const FILE_AGE As Integer = -50
```

```
  Private Const FILE_ROOT As String = "C:\"
  Private Const FILE_FILTER As String = "*.xls"

  Private isCheckForNewer As Boolean = True
  Dim fileAgeLimit As Integer
  Private listForEnumerator As ArrayList

  Public Sub Main()

    fileAgeLimit = DirectCast(Dts.Variables("FileAge").Value, Integer)

    ' If value provided is positive, we want files NEWER THAN n days.
    '  If negative, we want files OLDER THAN n days.
    If fileAgeLimit < 0 Then
      isCheckForNewer = False
    End If
    ' Extract number of days as positive integer.
    fileAgeLimit = Math.Abs(fileAgeLimit)

    listForEnumerator = New ArrayList

    GetFilesInFolder(FILE_ROOT)

    ' Return the list of files to the variable
    '  for later use by the Foreach from Variable enumerator.
    System.Windows.Forms.MessageBox.Show("Matching files: " &
listForEnumerator.Count.ToString, "Results",
Windows.Forms.MessageBoxButtons.OK, Windows.Forms.MessageBoxIcon.Information)
    Dts.Variables("FileList").Value = listForEnumerator

    Dts.TaskResult = ScriptResults.Success

  End Sub
```

```
  Private Sub GetFilesInFolder(ByVal folderPath As String)

    Dim localFiles() As String
    Dim localFile As String
    Dim fileChangeDate As Date
    Dim fileAge As TimeSpan
    Dim fileAgeInDays As Integer
    Dim childFolder As String

    Try
      localFiles = Directory.GetFiles(folderPath, FILE_FILTER)
      For Each localFile In localFiles
        fileChangeDate = File.GetLastWriteTime(localFile)
        fileAge = DateTime.Now.Subtract(fileChangeDate)
        fileAgeInDays = fileAge.Days
        CheckAgeOfFile(localFile, fileAgeInDays)
      Next

      If Directory.GetDirectories(folderPath).Length > 0 Then
        For Each childFolder In Directory.GetDirectories(folderPath)
          GetFilesInFolder(childFolder)
        Next
      End If

    Catch
      ' Ignore exceptions on special folders such as System Volume
Information.
    End Try

  End Sub

  Private Sub CheckAgeOfFile(ByVal localFile As String, ByVal fileAgeInDays
As Integer)
```

```
    If isCheckForNewer Then
      If fileAgeInDays <= fileAgeLimit Then
        listForEnumerator.Add(localFile)
      End If
    Else
      If fileAgeInDays > fileAgeLimit Then
        listForEnumerator.Add(localFile)
      End If
    End If

  End Sub

End Class
using System;
using System.Data;
using System.Math;
using Microsoft.SqlServer.Dts.Runtime;
using System.Collections;
using System.IO;

public partial class ScriptMain :
Microsoft.SqlServer.Dts.Tasks.ScriptTask.VSTARTScriptObjectModelBase
    {


        private const int FILE_AGE = -50;

        private const string FILE_ROOT = "C:\\";
        private const string FILE_FILTER = "*.xls";

        private bool isCheckForNewer = true;
        int fileAgeLimit;
        private ArrayList listForEnumerator;
```

```
        public void Main()
  {

    fileAgeLimit = (int)(Dts.Variables["FileAge"].Value);

    // If value provided is positive, we want files NEWER THAN n days.
    // If negative, we want files OLDER THAN n days.
    if (fileAgeLimit<0)
    {
      isCheckForNewer = false;
    }
    // Extract number of days as positive integer.
    fileAgeLimit = Math.Abs(fileAgeLimit);

    ArrayList listForEnumerator = new ArrayList();

    GetFilesInFolder(FILE_ROOT);

    // Return the list of files to the variable
    // for later use by the Foreach from Variable enumerator.
    System.Windows.Forms.MessageBox.Show("Matching files: "+
listForEnumerator.Count, "Results",
MessageBoxButtons.OK, MessageBoxIcon.Information);
    Dts.Variables["FileList"].Value = listForEnumerator;

    Dts.TaskResult = (int)ScriptResults.Success;

  }

        private void GetFilesInFolder(string folderPath)
        {

            string[] localFiles;
```

```
            DateTime fileChangeDate;
            TimeSpan fileAge;
            int fileAgeInDays;

            try
            {
                localFiles = Directory.GetFiles(folderPath, FILE_FILTER);
                foreach (string localFile in localFiles)
                {
                    fileChangeDate = File.GetLastWriteTime(localFile);
                    fileAge = DateTime.Now.Subtract(fileChangeDate);
                    fileAgeInDays = fileAge.Days;
                    CheckAgeOfFile(localFile, fileAgeInDays);
                }

                if (Directory.GetDirectories(folderPath).Length > 0)
                {
                    foreach (string childFolder in
Directory.GetDirectories(folderPath))
                    {
                        GetFilesInFolder(childFolder);
                    }
                }

            }
            catch
            {
                // Ignore exceptions on special folders, such as System
Volume Information.
            }

        }

        private void CheckAgeOfFile(string localFile, int fileAgeInDays)
```

```
    {

        if (isCheckForNewer)
        {
            if (fileAgeInDays <= fileAgeLimit)
            {
                listForEnumerator.Add(localFile);
            }
        }
        else
        {
            if (fileAgeInDays > fileAgeLimit)
            {
                listForEnumerator.Add(localFile);
            }
        }

    }

}
```

Stay Up to Date with Integration Services

For the latest downloads, articles, samples, and videos from Microsoft, as well as selected solutions from the community, visit the Integration Services page on MSDN:

Visit the Integration Services page on MSDN

For automatic notification of these updates, subscribe to the RSS feeds available on the page.

See Also

Foreach Loop Container

How to: Configure a Foreach Loop Container

Querying the Active Directory with the Script Task

Enterprise data processing applications, such as Integration Services packages, often need to process data differently based on the rank, job title, or other characteristics of employees stored in Active Directory. Active Directory is a Microsoft Windows directory service that provides a centralized store of metadata, not only about users, but also about other organizational assets such as computers and printers. The **System.DirectoryServices** namespace in the Microsoft .NET Framework provides classes for working with Active Directory, to help you direct data processing workflow based on the information that it stores.

Note

If you want to create a task that you can more easily reuse across multiple packages, consider using the code in this Script task sample as the starting point for a custom task. For more information, see Extending the Package with Custom Tasks.

Description

The following example retrieves an employee's name, title, and phone number from Active Directory based on the value of the `email` variable, which contains the e-mail address of the employee. Precedence constraints in the package can use the retrieved information to determine, for example, whether to send a low-priority e-mail message or a high-priority page, based on the job title of the employee.

To configure this Script Task example

1. Create the three string variables `email`, `name`, and `title`. Enter a valid corporate email address as the value of the `email` variable.
2. On the **Script** page of the **Script Task Editor**, add the `email` variable to the **ReadOnlyVariables** property.
3. Add the `name` and `title` variables to the **ReadWriteVariables** property.
4. In the script project, add a reference to the **System.DirectoryServices** namespace.
5. . In your code, use an **Imports** statement to import the **DirectoryServices** namespace.

Note

To run this script successfully, your company must be using Active Directory on its network and storing the employee information that this example uses.

Code

```
Public Sub Main()

    Dim directory As DirectoryServices.DirectorySearcher
    Dim result As DirectoryServices.SearchResult
    Dim email As String

    email = Dts.Variables("email").Value.ToString

    Try
        directory = New _
            DirectoryServices.DirectorySearcher("(mail=" & email & ")")
        result = directory.FindOne
        Dts.Variables("name").Value = _
            result.Properties("displayname").ToString
        Dts.Variables("title").Value = _
            result.Properties("title").ToString
        Dts.TaskResult = ScriptResults.Success
    Catch ex As Exception
        Dts.Events.FireError(0, _
            "Script Task Example", _
            ex.Message & ControlChars.CrLf & ex.StackTrace, _
```

```
            String.Empty, 0)
        Dts.TaskResult = ScriptResults.Failure
    End Try

End Sub
```

```
        public void Main()
        {
            //
            DirectorySearcher directory;
            SearchResult result;
            string email;

            email = (string)Dts.Variables["email"].Value;

            try
            {
                directory = new DirectorySearcher("(mail=" + email + ")");
                result = directory.FindOne();
                Dts.Variables["name"].Value =
result.Properties["displayname"].ToString();
                Dts.Variables["title"].Value =
result.Properties["title"].ToString();
                Dts.TaskResult = (int)ScriptResults.Success;
            }
            catch (Exception ex)
            {
                Dts.Events.FireError(0, "Script Task Example", ex.Message +
"\n" + ex.StackTrace, String.Empty, 0);
                Dts.TaskResult = (int)ScriptResults.Failure;
            }

        }
```

External Resources

- Technical article, Processing Active Directory Information in SSIS, on social.technet.microsoft.com

Stay Up to Date with Integration Services

For the latest downloads, articles, samples, and videos from Microsoft, as well as selected solutions from the community, visit the Integration Services page on MSDN:

- Visit the Integration Services page on MSDN

For automatic notification of these updates, subscribe to the RSS feeds available on the page.

Monitoring Performance Counters with the Script Task

Administrators may need to monitor the performance of Integration Services packages that perform complex transformations on large amounts of data. The **System.Diagnostics** namespace of the Microsoft .NET Framework provides classes for using existing performance counters and for creating your own performance counters.

Performance counters store application performance information that you can use to analyze the performance of software over time. Performance counters can be monitored locally or remotely by using the **Performance Monitor** tool. You can store the values of performance counters in variables for later control flow branching in the package.

As an alternative to using performance counters, you can raise the **M:Microsoft.SqlServer.Dts.Runtime.IDTSComponentEvents.FireProgress(System.String,System.Int32,System.Int32,System.Int32,System.String,System.Boolean@)** event through the **P:Microsoft.SqlServer.Dts.Tasks.ScriptTask.ScriptObjectModel.Events** property of the **Dts** object. The **M:Microsoft.SqlServer.Dts.Runtime.IDTSComponentEvents.FireProgress(System.String,System.Int32,System.Int32,System.Int32,System.String,System.Boolean@)** event returns both incremental progress and percentage complete information to the Integration Services runtime.

Note

If you want to create a task that you can more easily reuse across multiple packages, consider using the code in this Script task sample as the starting point for a custom task. For more information, see Extending the Package with Custom Tasks.

Description

The following example creates a custom performance counter and increments the counter. First, the example determines whether the performance counter already exists. If the performance counter has not been created, the script calls the **Create** method of the **PerformanceCounterCategory** object to create it. After the performance counter has been created, the script increments the counter. Finally, the example follows the best practice of calling the **Close** method on the performance counter when it is no longer needed.

Note

Creating a new performance counter category and performance counter requires administrative rights. Also, the new category and counter persist on the computer after creation.

To configure this Script Task example

- Use an **Imports** statement in your code to import the **System.Diagnostics** namespace.

Example Code

```
Public Sub Main()

    Dim myCounter As PerformanceCounter

    Try
        'Create the performance counter if it does not already exist.
        If Not _
        PerformanceCounterCategory.Exists("TaskExample") Then
            PerformanceCounterCategory.Create("TaskExample", _
                "Task Performance Counter Example", "Iterations", _
                "Number of times this task has been called.")
        End If

        'Initialize the performance counter.
        myCounter = New PerformanceCounter("TaskExample", _
            "Iterations", String.Empty, False)

        'Increment the performance counter.
```

```
        myCounter.Increment()

         myCounter.Close()
        Dts.TaskResult = ScriptResults.Success
    Catch ex As Exception
        Dts.Events.FireError(0, _
            "Task Performance Counter Example", _
            ex.Message & ControlChars.CrLf & ex.StackTrace, _
            String.Empty, 0)
        Dts.TaskResult = ScriptResults.Failure
    End Try

End Sub

public class ScriptMain
{



public void Main()
        {

            PerformanceCounter myCounter;

            try
            {
                //Create the performance counter if it does not already
exist.
                if (!PerformanceCounterCategory.Exists("TaskExample"))
                {
                    PerformanceCounterCategory.Create("TaskExample", "Task
Performance Counter Example", "Iterations", "Number of times this task has
been called.");
                }
```

```
                //Initialize the performance counter.
                myCounter = new PerformanceCounter("TaskExample",
"Iterations", String.Empty, false);

                //Increment the performance counter.
                myCounter.Increment();

                myCounter.Close();
                Dts.TaskResult = (int)ScriptResults.Success;
            }
            catch (Exception ex)
            {
                Dts.Events.FireError(0, "Task Performance Counter Example",
ex.Message + "\r" + ex.StackTrace, String.Empty, 0);
                Dts.TaskResult = (int)ScriptResults.Failure;
            }

            Dts.TaskResult = (int)ScriptResults.Success;
        }
```

Stay Up to Date with Integration Services

For the latest downloads, articles, samples, and videos from Microsoft, as well as selected solutions from the community, visit the Integration Services page on MSDN:

Visit the Integration Services page on MSDN

For automatic notification of these updates, subscribe to the RSS feeds available on the page.

Working with Images with the Script Task

A database of products or users frequently includes images in addition to text and numeric data. The **System.Drawing** namespace in the Microsoft .NET Framework provides classes for manipulating images.

Example 1: Convert Images to JPEG Format

Example 2: Create and Save Thumbnail Images

Note

If you want to create a task that you can more easily reuse across multiple packages, consider using the code in this Script task sample as the starting point for a custom task. For more information, see Extending the Package with Custom Tasks.

Example 1 Description: Convert Images to JPEG Format

The following example opens an image file specified by a variable and saves it as a compressed JPEG file by using an encoder. The code to retrieve encoder information is encapsulated in a private function.

To configure this Script Task example for use with a single image file

1. Create a string variable named `CurrentImageFile` and set its value to the path and file name of an existing image file.
2. On the **Script** page of the **Script Task Editor**, add the `CurrentImageFile` variable to the **ReadOnlyVariables** property.
3. In the script project, set a reference to the **System.Drawing** namespace.

4. In your code, use **Imports** statements to import the **System.Drawing** and **System.IO** namespaces.

To configure this Script Task example for use with multiple image files

1. Place the Script task within a Foreach Loop container.
2. On the **Collection** page of the **Foreach Loop Editor**, select the **Foreach File Enumerator** as the enumerator, and specify the path and file mask of the source files, such as "*.bmp."
3. On the **Variable Mappings** page, map the `CurrentImageFile` variable to Index 0. This variable passes the current file name to the Script task on each iteration of the enumerator.

> **Note**
> These steps are in addition to the steps listed in the procedure for use with a single image file.

Example 1 Code

```
Public Sub Main()

    'Create and initialize variables.
    Dim currentFile As String
    Dim newFile As String
    Dim bmp As Bitmap
    Dim eps As New Imaging.EncoderParameters(1)
    Dim ici As Imaging.ImageCodecInfo
    Dim supportedExtensions() As String = _
        {".BMP", ".GIF", ".JPG", ".JPEG", ".EXIF", ".PNG", _
        ".TIFF", ".TIF", ".ICO", ".ICON"}

    Try
        'Store the variable in a string for local manipulation.
        currentFile = Dts.Variables("CurrentImageFile").Value.ToString
        'Check the extension of the file against a list of
        'files that the Bitmap class supports.
        If Array.IndexOf(supportedExtensions, _
            Path.GetExtension(currentFile).ToUpper) > -1 Then
```

```
            'Load the file.
            bmp = New Bitmap(currentFile)

            'Calculate the new name for the compressed image.
            'Note: This will overwrite existing JPEGs.
            newFile = Path.Combine( _
                Path.GetDirectoryName(currentFile), _
                String.Concat(Path.GetFileNameWithoutExtension(currentFile),
_
                ".jpg"))

            'Specify the compression ratio (0=worst quality, 100=best
quality).
            eps.Param(0) = New Imaging.EncoderParameter( _
                Imaging.Encoder.Quality, 75)

            'Retrieve the ImageCodecInfo associated with the jpeg format.
            ici = GetEncoderInfo("image/jpeg")

            'Save the file, compressing it into the jpeg encoding.
            bmp.Save(newFile, ici, eps)
        Else
            'The file is not supported by the Bitmap class.
            Dts.Events.FireWarning(0, "Image Resampling Sample", _
                "File " & currentFile & " is not a supported format.", _
                "", 0)
         End If
        Dts.TaskResult = ScriptResults.Success
    Catch ex As Exception
        'An error occurred.
        Dts.Events.FireError(0, "Image Resampling Sample", _
            ex.Message & ControlChars.CrLf & ex.StackTrace, _
            String.Empty, 0)
        Dts.TaskResult = ScriptResults.Failure
```

```
    End Try

End Sub

Private Function GetEncoderInfo(ByVal mimeType As String) As
Imaging.ImageCodecInfo

    'The available image codecs are returned as an array,
    'which requires code to iterate until the specified codec is found.

    Dim count As Integer
    Dim encoders() As Imaging.ImageCodecInfo

    encoders = Imaging.ImageCodecInfo.GetImageEncoders()

    For count = 0 To encoders.Length
        If encoders(count).MimeType = mimeType Then
            Return encoders(count)
        End If
    Next

    'This point is only reached if a codec is not found.
    Err.Raise(513, "Image Resampling Sample", String.Format( _
        "The {0} codec is not available. Unable to compress file.", _
            mimeType))
    Return Nothing

End Function
```

Example 2 Description: Create and Save Thumbnail Images

The following example opens an image file specified by a variable, creates a thumbnail of the image while maintaining a constant aspect ratio, and saves the thumbnail with a modified file name. The code that calculates the height and width of the thumbnail while maintaining a constant aspect ratio is encapsulated in a private subroutine.

To configure this Script Task example for use with a single image file

1. Create a string variable named `CurrentImageFile` and set its value to the path and file name of an existing image file.
2. Also create the `MaxThumbSize` integer variable and assign a value in pixels, such as 100.
3. On the **Script** page of the **Script Task Editor**, add both variables to the **ReadOnlyVariables** property.
4. In the script project, set a reference to the **System.Drawing** namespace.
5. In your code, use **Imports** statements to import the **System.Drawing** and **System.IO** namespaces.

To configure this Script Task example for use with multiple image files

1. Place the Script task within a Foreach Loop container.
2. On the **Collection** page of the **Foreach Loop Editor**, select the **Foreach File Enumerator** as the **Enumerator**, and specify the path and file mask of the source files, such as "*.jpg."
3. On the **Variable Mappings** page, map the `CurrentImageFile` variable to Index 0. This variable passes the current file name to the Script task on each iteration of the enumerator.

> **Note**
> These steps are in addition to the steps listed in the procedure for use with a single image file.

Example 2 Code

```
Public Sub Main()

    Dim currentImageFile As String
    Dim currentImage As Image
    Dim maxThumbSize As Integer
    Dim thumbnailImage As Image
    Dim thumbnailFile As String
    Dim thumbnailHeight As Integer
    Dim thumbnailWidth As Integer

    currentImageFile = Dts.Variables("CurrentImageFile").Value.ToString
    thumbnailFile = Path.Combine( _
        Path.GetDirectoryName(currentImageFile), _
```

```
        String.Concat(Path.GetFileNameWithoutExtension(currentImageFile), _
        "_thumbnail.jpg"))

    Try
        currentImage = Image.FromFile(currentImageFile)

        maxThumbSize = CType(Dts.Variables("MaxThumbSize").Value, Integer)
        CalculateThumbnailSize( _
            maxThumbSize, currentImage, thumbnailWidth, thumbnailHeight)

        thumbnailImage = currentImage.GetThumbnailImage( _
           thumbnailWidth, thumbnailHeight, Nothing, Nothing)
        thumbnailImage.Save(thumbnailFile)
        Dts.TaskResult = ScriptResults.Success
    Catch ex As Exception
        Dts.Events.FireError(0, "Script Task Example", _
        ex.Message & ControlChars.CrLf & ex.StackTrace, _
        String.Empty, 0)
        Dts.TaskResult = ScriptResults.Failure
    End Try

End Sub

Private Sub CalculateThumbnailSize( _
    ByVal maxSize As Integer, ByVal sourceImage As Image, _
    ByRef thumbWidth As Integer, ByRef thumbHeight As Integer)

    If sourceImage.Width > sourceImage.Height Then
        thumbWidth = maxSize
        thumbHeight = CInt((maxSize / sourceImage.Width) *
sourceImage.Height)
    Else
        thumbHeight = maxSize
        thumbWidth = CInt((maxSize / sourceImage.Height) * sourceImage.Width)
```

```
    End If

End Sub
bool ThumbnailCallback()
        {
            return false;
        }
        public void Main()
        {

            string currentImageFile;
            Image currentImage;
            int maxThumbSize;
            Image thumbnailImage;
            string thumbnailFile;
            int thumbnailHeight = 0;
            int thumbnailWidth = 0;



            currentImageFile =
Dts.Variables["CurrentImageFile"].Value.ToString();
            thumbnailFile =
Path.Combine(Path.GetDirectoryName(currentImageFile),
String.Concat(Path.GetFileNameWithoutExtension(currentImageFile),
"_thumbnail.jpg"));

            try
            {

                currentImage = Image.FromFile(currentImageFile);

                maxThumbSize = (int)Dts.Variables["MaxThumbSize"].Value;
                CalculateThumbnailSize(maxThumbSize, currentImage, ref
thumbnailWidth, ref thumbnailHeight);
```

```
                Image.GetThumbnailImageAbort myCallback = new
Image.GetThumbnailImageAbort(ThumbnailCallback);

                thumbnailImage =
currentImage.GetThumbnailImage(thumbnailWidth, thumbnailHeight,
ThumbnailCallback, IntPtr.Zero);
                thumbnailImage.Save(thumbnailFile);
                Dts.TaskResult = (int)ScriptResults.Success;
            }
            catch (Exception ex)
            {
                Dts.Events.FireError(0, "Script Task Example", ex.Message +
"\r" + ex.StackTrace, String.Empty, 0);
                Dts.TaskResult = (int)ScriptResults.Failure;
            }

        }

        private void CalculateThumbnailSize(int maxSize, Image sourceImage,
ref int thumbWidth, ref int thumbHeight)
        {

            if (sourceImage.Width > sourceImage.Height)
            {
                thumbWidth = maxSize;
                thumbHeight = (int)(sourceImage.Height * maxSize /
sourceImage.Width);
            }
            else
            {
                thumbHeight = maxSize;
                thumbWidth = (int)(sourceImage.Width * maxSize /
sourceImage.Height);
```

```
        }

    }
```

Stay Up to Date with Integration Services

For the latest downloads, articles, samples, and videos from Microsoft, as well as selected solutions from the community, visit the Integration Services page on MSDN:

Visit the Integration Services page on MSDN

For automatic notification of these updates, subscribe to the RSS feeds available on the page.

Finding Installed Printers with the Script Task

The data that is transformed by Integration Services packages often has a printed report as its final destination. The **System.Drawing.Printing** namespace in the Microsoft .NET Framework provides classes for working with printers.

Note

If you want to create a task that you can more easily reuse across multiple packages, consider using the code in this Script task sample as the starting point for a custom task. For more information, see Extending the Package with Custom Tasks.

Description

The following example locates printers installed on the server that support legal size paper (as used in the United States). The code to check supported paper sizes is encapsulated in a private function. To enable you to track the progress of the script as it checks the settings for each printer, the script uses the **M:Microsoft.SqlServer.Dts.Tasks.ScriptTask.ScriptObjectModel.Log(System.String,System.Int32,System.Byte[])** method to raise an informational message for printers with legal size paper, and to raise a warning for printers without legal size paper. These messages appear in the **Output**

window of the Microsoft Visual Studio Tools for Applications (VSTA) IDE when you run the package in the designer.

To configure this Script Task example

1. Create the variable named `PrinterList` with type **Object**.
2. On the **Script** page of the **Script Task Editor**, add this variable to the **ReadWriteVariables** property.
3. In the script project, add a reference to the **System.Drawing** namespace.
4. In your code, use **Imports** statements to import the **System.Collections** and the **System.Drawing.Printing** namespaces.

Code

```
Public Sub Main()

    Dim printerName As String
    Dim currentPrinter As New PrinterSettings
    Dim size As PaperSize

    Dim printerList As New ArrayList
    For Each printerName In PrinterSettings.InstalledPrinters
        currentPrinter.PrinterName = printerName
        If PrinterHasLegalPaper(currentPrinter) Then
            printerList.Add(printerName)
            Dts.Events.FireInformation(0, "Example", _
                "Printer " & printerName & " has legal paper.", _
                String.Empty, 0, False)
        Else
            Dts.Events.FireWarning(0, "Example", _
                "Printer " & printerName & " DOES NOT have legal paper.", _
                String.Empty, 0)
        End If
    Next

    Dts.Variables("PrinterList").Value = printerList
```

```
    Dts.TaskResult = ScriptResults.Success

End Sub

Private Function PrinterHasLegalPaper( _
    ByVal thisPrinter As PrinterSettings) As Boolean

    Dim size As PaperSize
    Dim hasLegal As Boolean = False

    For Each size In thisPrinter.PaperSizes
        If size.Kind = PaperKind.Legal Then
            hasLegal = True
        End If
    Next

    Return hasLegal

End Function
public void Main()
        {

            PrinterSettings currentPrinter = new PrinterSettings();
            PaperSize size;
            Boolean Flag = false;

            ArrayList printerList = new ArrayList();
            foreach (string printerName in PrinterSettings.InstalledPrinters)
            {
                currentPrinter.PrinterName = printerName;
                if (PrinterHasLegalPaper(currentPrinter))
                {
                    printerList.Add(printerName);
```

```
                    Dts.Events.FireInformation(0, "Example", "Printer " +
printerName + " has legal paper.", String.Empty, 0, ref Flag);
                }
                else
                {
                    Dts.Events.FireWarning(0, "Example", "Printer " +
printerName + " DOES NOT have legal paper.", String.Empty, 0);
                }
            }

            Dts.Variables["PrinterList"].Value = printerList;

            Dts.TaskResult = (int)ScriptResults.Success;

        }

        private bool PrinterHasLegalPaper(PrinterSettings thisPrinter)
        {

            bool hasLegal = false;

            foreach (PaperSize size in thisPrinter.PaperSizes)
            {
                if (size.Kind == PaperKind.Legal)
                {
                    hasLegal = true;
                }
            }

            return hasLegal;

        }
```

Stay Up to Date with Integration Services

For the latest downloads, articles, samples, and videos from Microsoft, as well as selected solutions from the community, visit the Integration Services page on MSDN:

Visit the Integration Services page on MSDN

For automatic notification of these updates, subscribe to the RSS feeds available on the page.

See Also

Script Task Examples

Sending an HTML Mail Message with the Script Task

The Integration Services SendMail task only supports mail messages in plain text format. However you can easily send HTML mail messages by using the Script task and the mail capabilities of the .NET Framework.

Note

If you want to create a task that you can more easily reuse across multiple packages, consider using the code in this Script task sample as the starting point for a custom task. For more information, see Extending the Package with Custom Tasks.

Description

The following example uses the **System.Net.Mail** namespace to configure and send an HTML mail message. The script obtains the To, From, Subject, and body of the e-mail from package variables, uses them to create a new **MailMessage**, and sets its **IsBodyHtml** property to **True**. Then it obtains the SMTP server name from another package variable, initializes an instance of **System.Net.Mail.SmtpClient**, and calls its **Send** method to send the HTML message. The sample encapsulates the message sending functionality in a subroutine that could be reused in other scripts.

To configure this Script Task example without an SMTP Connection Manager

1. Create string variables named `HtmlEmailTo`, `HtmlEmailFrom`, and `HtmlEmailSubject`

and assign appropriate values to them for a valid test message.

2. Create a string variable named `HtmlEmailBody` and assign a string of HTML markup to it. For example:

```
<html><body><h1>Testing</h1><p>This is a <b>test</b>
message.</p></body></html>
```

3. Create a string variable named `HtmlEmailServer` and assign the name of an available SMTP server that accepts anonymous outgoing messages.
4. Assign all five of these variables to the **ReadOnlyVariables** property of a new Script task.
5. Import the **System.Net** and **System.Net.Mail** namespaces into your code.

The sample code in this topic obtains the SMTP server name from a package variable. However, you could also take advantage of an SMTP connection manager to encapsulate the connection information, and extract the server name from the connection manager in your code. The **M:Microsoft.SqlServer.Dts.ManagedConnections.SMTPConn.AcquireConnection(System.Object)** method of the SMTP connection manager returns a string in the following format:

```
SmtpServer=smtphost;UseWindowsAuthentication=False;EnableSsl=False;
```

You can use the **String.Split** method to separate this argument list into an array of individual strings at each semicolon (;) or equal sign (=), and then extract the second argument (subscript 1) from the array as the server name.

To configure this Script Task example with an SMTP Connection Manager

1. Modify the Script task configured earlier by removing the `HtmlEmailServer` variable from the list of **ReadOnlyVariables**.
2. Replace the line of code that obtains the server name:

```
Dim smtpServer As String = _
  Dts.Variables("HtmlEmailServer").Value.ToString
```

   with the following lines:

```
Dim smtpConnectionString As String = _
  DirectCast(Dts.Connections("SMTP Connection
Manager").AcquireConnection(Dts.Transaction), String)
Dim smtpServer As String = _
  smtpConnectionString.Split(New Char() {"="c, ";"c})(1)
```

Code

```
Public Sub Main()

  Dim htmlMessageTo As String = _
```

```
    Dts.Variables("HtmlEmailTo").Value.ToString
  Dim htmlMessageFrom As String = _
    Dts.Variables("HtmlEmailFrom").Value.ToString
  Dim htmlMessageSubject As String = _
    Dts.Variables("HtmlEmailSubject").Value.ToString
  Dim htmlMessageBody As String = _
    Dts.Variables("HtmlEmailBody").Value.ToString
  Dim smtpServer As String = _
    Dts.Variables("HtmlEmailServer").Value.ToString

  SendMailMessage( _
      htmlMessageTo, htmlMessageFrom, _
      htmlMessageSubject, htmlMessageBody, _
      True, smtpServer)

  Dts.TaskResult = ScriptResults.Success

End Sub

Private Sub SendMailMessage( _
    ByVal SendTo As String, ByVal From As String, _
    ByVal Subject As String, ByVal Body As String, _
    ByVal IsBodyHtml As Boolean, ByVal Server As String)

  Dim htmlMessage As MailMessage
  Dim mySmtpClient As SmtpClient

  htmlMessage = New MailMessage( _
    SendTo, From, Subject, Body)
  htmlMessage.IsBodyHtml = IsBodyHtml

  mySmtpClient = New SmtpClient(Server)
  mySmtpClient.Credentials = CredentialCache.DefaultNetworkCredentials
```

```
    mySmtpClient.Send(htmlMessage)

  End Sub
public void Main()
        {

            string htmlMessageTo =
Dts.Variables["HtmlEmailTo"].Value.ToString();
            string htmlMessageFrom =
Dts.Variables["HtmlEmailFrom"].Value.ToString();
            string htmlMessageSubject =
Dts.Variables["HtmlEmailSubject"].Value.ToString();
            string htmlMessageBody =
Dts.Variables["HtmlEmailBody"].Value.ToString();
            string smtpServer =
Dts.Variables["HtmlEmailServer"].Value.ToString();

            SendMailMessage(htmlMessageTo, htmlMessageFrom,
htmlMessageSubject, htmlMessageBody, true, smtpServer);

            Dts.TaskResult = (int)ScriptResults.Success;

        }

        private void SendMailMessage(string SendTo, string From, string
Subject, string Body, bool IsBodyHtml, string Server)
        {

            MailMessage htmlMessage;
            SmtpClient mySmtpClient;

            htmlMessage = new MailMessage(SendTo, From, Subject, Body);
            htmlMessage.IsBodyHtml = IsBodyHtml;

            mySmtpClient = new SmtpClient(Server);
```

```
            mySmtpClient.Credentials =
CredentialCache.DefaultNetworkCredentials;
            mySmtpClient.Send(htmlMessage);

        }
```

Stay Up to Date with Integration Services

For the latest downloads, articles, samples, and videos from Microsoft, as well as selected solutions from the community, visit the Integration Services page on MSDN:

Visit the Integration Services page on MSDN

For automatic notification of these updates, subscribe to the RSS feeds available on the page.

See Also

Send Mail Task

Working with Excel Files with the Script Task

Integration Services provides the Excel connection manager, Excel source, and Excel destination for working with data stored in spreadsheets in the Microsoft Excel file format. The techniques described in this topic use the Script task to obtain information about available Excel databases (workbook files) and tables (worksheets and named ranges). These samples can easily be modified to work with any of the other file-based data sources supported by the Microsoft Jet OLE DB Provider.

Configuring a Package to Test the Samples

Example1: Check Whether an Excel File Exists

Example 2: Check Whether an Excel Table Exists

Example 3: Get a List of Excel Files in a Folder

Example 4: Get a List of Tables in an Excel File

Displaying the Results of the Samples

Note

If you want to create a task that you can more easily reuse across multiple packages, consider using the code in this Script task sample as the starting point for a custom task. For more information, see Extending the Package with Custom Tasks.

Configuring a Package to Test the Samples

You can configure a single package to test all the samples in this topic. The samples use many of the same package variables and the same .NET Framework classes.

To configure a package for use with the examples in this topic

1. Create a new Integration Services project in SQL Server Data Tools (SSDT) and open the default package for editing.
2. **Variables**. Open the **Variables** window and define the following variables:
   - `ExcelFile`, of type **String**. Enter the complete path and filename to an existing Excel workbook.
   - `ExcelTable`, of type **String**. Enter the name of an existing worksheet or named range in the workbook named in the value of the `ExcelFile` variable. This value is case-sensitive.
   - `ExcelFileExists`, of type **Boolean**.
   - `ExcelTableExists`, of type **Boolean**.
   - `ExcelFolder`, of type **String**. Enter the complete path of a folder that contains at least one Excel workbook.
   - `ExcelFiles`, of type **Object**.
   - `ExcelTables`, of type **Object**.
3. **Imports statements**. Most of the code samples require you to import one or both of the following .NET Framework namespaces at the top of your script file:
   - **System.IO**, for file system operations.
   - **System.Data.OleDb**, to open Excel files as data sources.
4. **References**. The code samples that read schema information from Excel files require an additional reference in the script project to the **System.Xml** namespace.
5. Set the default scripting language for the Script component by using the **Scripting language** option on the **General** page of the **Options** dialog box. For more information, see General Page.

Example 1 Description: Check Whether an Excel File Exists

This example determines whether the Excel workbook file specified in the `ExcelFile` variable exists, and then sets the Boolean value of the `ExcelFileExists` variable to the result. You can use this Boolean value for branching in the workflow of the package.

To configure this Script Task example

1. Add a new Script task to the package and change its name to **ExcelFileExists**.
2. In the **Script Task Editor**, on the **Script** tab, click **ReadOnlyVariables** and enter the property value using one of the following methods:
   - Type **ExcelFile**.

     -or-
   - Click the ellipsis (**…**) button next to the property field, and in the **Select variables** dialog box, select the **ExcelFile** variable.
3. Click **ReadWriteVariables** and enter the property value using one of the following methods:
   - Type **ExcelFileExists**.

     -or-
   - Click the ellipsis (**…**) button next to the property field, and in the **Select variables** dialog box, select the **ExcelFileExists** variable.
4. Click **Edit Script** to open the script editor.
5. Add an **Imports** statement for the **System.IO** namespace at the top of the script file.
6. Add the following code.

Example 1 Code

```
Public Class ScriptMain
  Public Sub Main()
    Dim fileToTest As String

    fileToTest = Dts.Variables("ExcelFile").Value.ToString
    If File.Exists(fileToTest) Then
      Dts.Variables("ExcelFileExists").Value = True
    Else
      Dts.Variables("ExcelFileExists").Value = False
    End If

    Dts.TaskResult = ScriptResults.Success
  End Sub
End Class
public class ScriptMain
{
  public void Main()
  {
```

```
    string fileToTest;

    fileToTest = Dts.Variables["ExcelFile"].Value.ToString();
    if (File.Exists(fileToTest))
    {
      Dts.Variables["ExcelFileExists"].Value = true;
    }
    else
    {
      Dts.Variables["ExcelFileExists"].Value = false;
    }

    Dts.TaskResult = (int)ScriptResults.Success;
  }
}
```

Example 2 Description: Check Whether an Excel Table Exists

This example determines whether the Excel worksheet or named range specified in the `ExcelTable` variable exists in the Excel workbook file specified in the `ExcelFile` variable, and then sets the Boolean value of the `ExcelTableExists` variable to the result. You can use this Boolean value for branching in the workflow of the package.

To configure this Script Task example

1. Add a new Script task to the package and change its name to **ExcelTableExists**.
2. In the **Script Task Editor**, on the **Script** tab, click **ReadOnlyVariables** and enter the property value using one of the following methods:
   - Type **ExcelTable** and **ExcelFile** separated by commas**.**

     -or-
   - Click the ellipsis (**...**) button next to the property field, and in the **Select variables** dialog box, select the **ExcelTable** and **ExcelFile** variables.
3. Click **ReadWriteVariables** and enter the property value using one of the following methods:
   - Type **ExcelTableExists**.

     -or-
   - Click the ellipsis (**...**) button next to the property field, and in the **Select variables** dialog box, select the **ExcelTableExists** variable.
4. Click **Edit Script** to open the script editor.

5. Add a reference to the **System.Xml** assembly in the script project.
6. Add **Imports** statements for the **System.IO** and **System.Data.OleDb** namespaces at the top of the script file.
7. Add the following code.

Example 2 Code

```
Public Class ScriptMain
  Public Sub Main()
    Dim fileToTest As String
    Dim tableToTest As String
    Dim connectionString As String
    Dim excelConnection As OleDbConnection
    Dim excelTables As DataTable
    Dim excelTable As DataRow
    Dim currentTable As String

    fileToTest = Dts.Variables("ExcelFile").Value.ToString
    tableToTest = Dts.Variables("ExcelTable").Value.ToString

    Dts.Variables("ExcelTableExists").Value = False
    If File.Exists(fileToTest) Then
      connectionString = "Provider=Microsoft.Jet.OLEDB.4.0;" & _
        "Data Source=" & fileToTest & _
        ";Extended Properties=Excel 8.0"
      excelConnection = New OleDbConnection(connectionString)
      excelConnection.Open()
      excelTables = excelConnection.GetSchema("Tables")
      For Each excelTable In excelTables.Rows
        currentTable = excelTable.Item("TABLE_NAME").ToString
        If currentTable = tableToTest Then
          Dts.Variables("ExcelTableExists").Value = True
        End If
      Next
    End If
```

```
    Dts.TaskResult = ScriptResults.Success
  End Sub
End Class
public class ScriptMain
{
    public void Main()
        {
            string fileToTest;
            string tableToTest;
            string connectionString;
            OleDbConnection excelConnection;
            DataTable excelTables;
            string currentTable;

            fileToTest = Dts.Variables["ExcelFile"].Value.ToString();
            tableToTest = Dts.Variables["ExcelTable"].Value.ToString();

            Dts.Variables["ExcelTableExists"].Value = false;
            if (File.Exists(fileToTest))
            {
                connectionString = "Provider=Microsoft.Jet.OLEDB.4.0;" +
                "Data Source=" + fileToTest + ";Extended Properties=Excel
8.0";
                excelConnection = new OleDbConnection(connectionString);
                excelConnection.Open();
                excelTables = excelConnection.GetSchema("Tables");
                foreach (DataRow excelTable in excelTables.Rows)
                {
                    currentTable = excelTable["TABLE_NAME"].ToString();
                    if (currentTable == tableToTest)
                    {
                        Dts.Variables["ExcelTableExists"].Value = true;
                    }
                }
```

```
            }

            Dts.TaskResult = (int)ScriptResults.Success;

        }
}
```

Example 3 Description: Get a List of Excel Files in a Folder

This example fills an array with the list of Excel files found in the folder specified in the value of the `ExcelFolder` variable, and then copies the array into the `ExcelFiles` variable. You can use the Foreach from Variable enumerator to iterate over the files in the array.

To configure this Script Task example

1. Add a new Script task to the package and change its name to **GetExcelFiles**.
2. Open the **Script Task Editor**, on the **Script** tab, click **ReadOnlyVariables** and enter the property value using one of the following methods:
   - Type **ExcelFolder**

     -or-
   - Click the ellipsis (**...**) button next to the property field, and in the **Select variables** dialog box, select the ExcelFolder variable.
3. Click **ReadWriteVariables** and enter the property value using one of the following methods:
   - Type **ExcelFiles**.

     -or-
   - Click the ellipsis (**...**) button next to the property field, and in the **Select variables** dialog box, select the ExcelFiles variable.
4. Click **Edit Script** to open the script editor.
5. Add an **Imports** statement for the **System.IO** namespace at the top of the script file.
6. Add the following code.

Example 3 Code

```
Public Class ScriptMain
  Public Sub Main()
    Const FILE_PATTERN As String = "*.xls"

    Dim excelFolder As String
    Dim excelFiles As String()
```

```
    excelFolder = Dts.Variables("ExcelFolder").Value.ToString
    excelFiles = Directory.GetFiles(excelFolder, FILE_PATTERN)

    Dts.Variables("ExcelFiles").Value = excelFiles

    Dts.TaskResult = ScriptResults.Success
  End Sub
End Class
public class ScriptMain
{
  public void Main()
  {
    const string FILE_PATTERN = "*.xls";

    string excelFolder;
    string[] excelFiles;

    excelFolder = Dts.Variables["ExcelFolder"].Value.ToString();
    excelFiles = Directory.GetFiles(excelFolder, FILE_PATTERN);

    Dts.Variables["ExcelFiles"].Value = excelFiles;

    Dts.TaskResult = (int)ScriptResults.Success;
  }
}
```

Alternate Solution

Instead of using a Script task to gather a list of Excel files into an array, you can also use the ForEach File enumerator to iterate over all the Excel files in a folder. For more information, see How to: Loop through Excel Files and Tables.

Example 4 Description: Get a List of Tables in an Excel File

This example fills an array with the list of worksheets and named ranges found in the Excel workbook file specified by the value of the `ExcelFile` variable, and then copies the array into the `ExcelTables` variable. You can use the Foreach from Variable Enumerator to iterate over the tables in the array.

Note
The list of tables in an Excel workbook includes both worksheets (which have the $ suffix) and named ranges. If you have to filter the list for only worksheets or only named ranges, you may have to add additional code for this purpose.

To configure this Script Task example

1. Add a new Script task to the package and change its name to **GetExcelTables**.
2. Open the **Script Task Editor**, on the **Script** tab, click **ReadOnlyVariables** and enter the property value using one of the following methods:
   - Type **ExcelFile**.

     -or-
   - Click the ellipsis (**...**) button next to the property field, and in the **Select variables** dialog box, select the ExcelFile variable.
3. Click **ReadWriteVariables** and enter the property value using one of the following methods:
   - Type **ExcelTables**.

     -or-
   - Click the ellipsis (**...**) button next to the property field, and in the **Select variables** dialog box, select the ExcelTables variable.
4. Click **Edit Script** to open the script editor.
5. Add a reference to the **System.Xml** namespace in the script project.
6. Add an **Imports** statement for the **System.Data.OleDb** namespace at the top of the script file.
7. Add the following code.

Example 4 Code

```
Public Class ScriptMain
  Public Sub Main()
    Dim excelFile As String
    Dim connectionString As String
    Dim excelConnection As OleDbConnection
    Dim tablesInFile As DataTable
    Dim tableCount As Integer = 0
    Dim tableInFile As DataRow
    Dim currentTable As String
    Dim tableIndex As Integer = 0
```

```
    Dim excelTables As String()

    excelFile = Dts.Variables("ExcelFile").Value.ToString
    connectionString = "Provider=Microsoft.Jet.OLEDB.4.0;" & _
        "Data Source=" & excelFile & _
        ";Extended Properties=Excel 8.0"
    excelConnection = New OleDbConnection(connectionString)
    excelConnection.Open()
    tablesInFile = excelConnection.GetSchema("Tables")
    tableCount = tablesInFile.Rows.Count
    ReDim excelTables(tableCount - 1)
    For Each tableInFile In tablesInFile.Rows
      currentTable = tableInFile.Item("TABLE_NAME").ToString
      excelTables(tableIndex) = currentTable
      tableIndex += 1
    Next

    Dts.Variables("ExcelTables").Value = excelTables

    Dts.TaskResult = ScriptResults.Success
  End Sub
End Class
public class ScriptMain
{
  public void Main()
        {
            string excelFile;
            string connectionString;
            OleDbConnection excelConnection;
            DataTable tablesInFile;
            int tableCount = 0;
            string currentTable;
            int tableIndex = 0;
```

```
            string[] excelTables = new string[5];

            excelFile = Dts.Variables["ExcelFile"].Value.ToString();
            connectionString = "Provider=Microsoft.Jet.OLEDB.4.0;" +
                "Data Source=" + excelFile + ";Extended Properties=Excel
8.0";
            excelConnection = new OleDbConnection(connectionString);
            excelConnection.Open();
            tablesInFile = excelConnection.GetSchema("Tables");
            tableCount = tablesInFile.Rows.Count;

            foreach (DataRow tableInFile in tablesInFile.Rows)
            {
                currentTable = tableInFile["TABLE_NAME"].ToString();
                excelTables[tableIndex] = currentTable;
                tableIndex += 1;
            }

            Dts.Variables["ExcelTables"].Value = excelTables;

            Dts.TaskResult = (int)ScriptResults.Success;
        }
}
```

Alternate Solution

Instead of using a Script task to gather a list of Excel tables into an array, you can also use the ForEach ADO.NET Schema Rowset Enumerator to iterate over all the tables (that is, worksheets and named ranges) in an Excel workbook file. For more information, see How to: Loop through Excel Files and Tables.

Displaying the Results of the Samples

If you have configured each of the examples in this topic in the same package, you can connect all the Script tasks to an additional Script task that displays the output of all the examples.

To configure a Script task to display the output of the examples in this topic

1. Add a new Script task to the package and change its name to **DisplayResults**.

2. Connect each of the four example Script tasks to one another, so that each task runs after the preceding task completes successfully, and connect the fourth example task to the **DisplayResults** task.
3. Open the **DisplayResults** task in the **Script Task Editor**.
4. On the **Script** tab, click **ReadOnlyVariables** and use one of the following methods to add all seven variables listed in Configuring a Package to Test the Samples:
   - Type the name of each variable separated by commas.

     -or-
   - Click the ellipsis (**...**) button next to the property field, and in the **Select variables** dialog box, selecting the variables.
5. Click **Edit Script** to open the script editor.
6. Add **Imports** statements for the **Microsoft.VisualBasic** and **System.Windows.Forms** namespaces at the top of the script file.
7. Add the following code.
8. Run the package and examine the results displayed in a message box.

Code to Display the Results

```
Public Class ScriptMain

  Public Sub Main()

    Const EOL As String = ControlChars.CrLf


    Dim results As String

    Dim filesInFolder As String()

    Dim fileInFolder As String

    Dim tablesInFile As String()

    Dim tableInFile As String


    results = _

      "Final values of variables:" & EOL & _

      "ExcelFile: " & Dts.Variables("ExcelFile").Value.ToString & EOL & _

      "ExcelFileExists: " & Dts.Variables("ExcelFileExists").Value.ToString &
EOL & _

      "ExcelTable: " & Dts.Variables("ExcelTable").Value.ToString & EOL & _

      "ExcelTableExists: " & Dts.Variables("ExcelTableExists").Value.ToString
& EOL & _

      "ExcelFolder: " & Dts.Variables("ExcelFolder").Value.ToString & EOL & _
```

```
      EOL

    results &= "Excel files in folder: " & EOL
    filesInFolder = DirectCast(Dts.Variables("ExcelFiles").Value, String())
    For Each fileInFolder In filesInFolder
      results &= " " & fileInFolder & EOL
    Next
    results &= EOL

    results &= "Excel tables in file: " & EOL
    tablesInFile = DirectCast(Dts.Variables("ExcelTables").Value, String())
    For Each tableInFile In tablesInFile
      results &= " " & tableInFile & EOL
    Next

    MessageBox.Show(results, "Results", MessageBoxButtons.OK,
MessageBoxIcon.Information)

    Dts.TaskResult = ScriptResults.Success
  End Sub
End Class
public class ScriptMain
{
  public void Main()
        {
            const string EOL = "\r";

            string results;
            string[] filesInFolder;
            //string fileInFolder;
            string[] tablesInFile;
            //string tableInFile;
```

```
            results = "Final values of variables:" + EOL + "ExcelFile: " +
Dts.Variables["ExcelFile"].Value.ToString() + EOL + "ExcelFileExists: " +
Dts.Variables["ExcelFileExists"].Value.ToString() + EOL + "ExcelTable: " +
Dts.Variables["ExcelTable"].Value.ToString() + EOL + "ExcelTableExists: " +
Dts.Variables["ExcelTableExists"].Value.ToString() + EOL + "ExcelFolder: " +
Dts.Variables["ExcelFolder"].Value.ToString() + EOL + EOL;

            results += "Excel files in folder: " + EOL;
            filesInFolder = (string[])(Dts.Variables["ExcelFiles"].Value);
            foreach (string fileInFolder in filesInFolder)
            {
                results += " " + fileInFolder + EOL;
            }
            results += EOL;

            results += "Excel tables in file: " + EOL;
            tablesInFile = (string[])(Dts.Variables["ExcelTables"].Value);
            foreach (string tableInFile in tablesInFile)
            {
                results += " " + tableInFile + EOL;
            }

            MessageBox.Show(results, "Results", MessageBoxButtons.OK,
MessageBoxIcon.Information);

            Dts.TaskResult = (int)ScriptResults.Success;
        }
}
```

Stay Up to Date with Integration Services

For the latest downloads, articles, samples, and videos from Microsoft, as well as selected solutions from the community, visit the Integration Services page on MSDN:

Visit the Integration Services page on MSDN

For automatic notification of these updates, subscribe to the RSS feeds available on the page.

See Also

Excel Connection Manager

How to: Loop through Excel Files and Tables

Sending to a Remote Private Message Queue with the Script Task

Message Queuing (also known as MSMQ) makes it easy for developers to communicate with application programs quickly and reliably by sending and receiving messages. A message queue may be located on the local computer or a remote computer, and may be public or private. In Integration Services, the MSMQ connection manager and Message Queue task do not support sending to a private queue on a remote computer. However, by using the Script task, it is easy to send a message to a remote private queue.

Note

If you want to create a task that you can more easily reuse across multiple packages, consider using the code in this Script task sample as the starting point for a custom task. For more information, see Extending the Package with Custom Tasks.

Description

The following example uses an existing MSMQ connection manager, together with objects and methods from the **System.Messaging** namespace, to send the text contained in a package variable to a remote private message queue. The call to the **M:Microsoft.SqlServer.Dts.ManagedConnections.MSMQConn.AcquireConnection(System.Object)** method of the MSMQ connection manager returns a **MessageQueue** object whose **Send** method accomplishes this task.

To configure this Script Task example

1. Create an MSMQ connection manager with the default name. Set the path of a valid remote private queue, in the following format:

   ```
   FORMATNAME:DIRECT=OS:<computername>\private$\<queuename>
   ```

2. Create an Integration Services variable named **MessageText** of type **String** to pass the message text into the script. Enter a default message as the value of the variable.
3. Add a Script Task to the design surface and edit it. On the **Script** tab of the **Script Task Editor**, add the `MessageText` variable to the **ReadOnlyVariables** property to make the variable available inside the script.
4. Click **Edit Script** to open the Microsoft Visual Studio Tools for Applications (VSTA) script editor.
5. Add a reference in the script project to the **System.Messaging** namespace.
6. Replace the contents of the script window with the code in the following section.

Code

```
Imports System
Imports Microsoft.SqlServer.Dts.Runtime
Imports System.Messaging

Public Class ScriptMain

    Public Sub Main()

        Dim remotePrivateQueue As MessageQueue
        Dim messageText As String

        remotePrivateQueue = _
            DirectCast(Dts.Connections("Message Queue Connection
Manager").AcquireConnection(Dts.Transaction), _
            MessageQueue)
        messageText = DirectCast(Dts.Variables("MessageText").Value, String)
        remotePrivateQueue.Send(messageText)

        Dts.TaskResult = ScriptResults.Success
```

```
    End Sub

End Class
using System;
using Microsoft.SqlServer.Dts.Runtime;
using System.Messaging;

public class ScriptMain
{

    public void Main()
        {

            MessageQueue remotePrivateQueue = new MessageQueue();
            string messageText;



            remotePrivateQueue = (MessageQueue)(Dts.Connections["Message
Queue Connection Manager"].AcquireConnection(Dts.Transaction) as
MessageQueue);
            messageText = (string)(Dts.Variables["MessageText"].Value);
            remotePrivateQueue.Send(messageText);

            Dts.TaskResult = (int)ScriptResults.Success;

        }

}
```

Stay Up to Date with Integration Services

For the latest downloads, articles, samples, and videos from Microsoft, as well as selected solutions from the community, visit the Integration Services page on MSDN:

Visit the Integration Services page on MSDN

For automatic notification of these updates, subscribe to the RSS feeds available on the page.

See Also

Message Queue Task

Extending the Data Flow with the Script Component

The Script component extends the data flow capabilities of Microsoft Integration Services packages with custom code written in Microsoft Visual Basic or Microsoft Visual C# that is compiled and executed at package run time. The Script component simplifies the development of a custom data flow source, transformation, or destination when the sources, transformations, and destinations included with Integration Services do not fully satisfy your requirements. After you configure the component with the expected inputs and outputs, it writes all the required infrastructure code for you, letting you focus exclusively on the code that is required for your custom processing.

A Script component interacts with the containing package and with the data flow through the autogenerated classes in the **ComponentWrapper** and **BufferWrapper** project items, which are instances of the **T:Microsoft.SqlServer.Dts.Pipeline.ScriptComponent** and the **T:Microsoft.SqlServer.Dts.Pipeline.ScriptBuffer** classes respectively. These classes make connections, variables, and other package items available as typed objects, and manage inputs and outputs. The Script component can also use the Visual Basic namespace and the .NET Framework class library, as well as custom assemblies, to implement custom functionality.

The Script component and the infrastructure code that it generates for you simplify significantly the process of developing a custom data flow component. However, to understand how the Script component works, you may find it useful to read the section Extending the Data Flow with Custom Components to understand the steps that are involved in developing a custom data flow component.

If you are creating a source, transformation, or destination that you plan to reuse in multiple packages, you should consider developing a custom component instead of using the Script component. For more information, see Extending the Data Flow Task with Custom Components.

In This Section

The following topics provide more information about the Script component.

Configuring the Script Component

Properties that you configure in the **Script Transformation Editor** affect the capabilities and the performance of Script component code.

Coding the Script Component

You use the Microsoft Visual Studio Tools for Applications (VSTA) development environment to develop the scripts contained in the Script component.

Understanding the Script Component Object Model

A new Script component project contains three project items with several classes and autogenerated properties and methods.

Using Variables in the Script Component

The **ComponentWrapper** project item contains strongly-typed accessor properties for package variables.

Using Connections in the Script Component

The **ComponentWrapper** project item also contains strongly-typed accessor properties for connections defined in the package.

Raising Events in the Script Component

You can raise events to provide notification of problems and errors.

Logging in the Script Component

You can log information to log providers enabled on the package.

Developing Specific Types of Script Components

These simple examples explain and demonstrate how to use the Script component to develop data flow sources, transformations, and destinations.

Script Component Examples

These simple examples explain and demonstrate a few possible uses for the Script component.

Stay Up to Date with Integration Services

For the latest downloads, articles, samples, and videos from Microsoft, as well as selected solutions from the community, visit the Integration Services page on MSDN:

Visit the Integration Services page on MSDN

For automatic notification of these updates, subscribe to the RSS feeds available on the page.

See Also

Script Component

Comparing the Script Task and the Script Component

Configuring the Script Component in the Script Component Editor

Before you write custom code in the Script component, you must select the type of data flow component that you want to create—source, transformation, or destination—and then configure the component's metadata and properties in the **Script Transformation Editor**.

Selecting the Type of Component to Create

When you add a Script component to the Data Flow pane of SSIS Designer, the **Select Script Component Type** dialog box appears. You preconfigure the component as a source, transformation, or destination. After you make this initial selection, you can continue to configure the component in the **Script Transformation Editor**.

To set the default script language for the Script component, use the **Scripting language** option on the **General** page of the **Options** dialog box. For more information, see General Page.

Understanding the Two Design-Time Modes

In SSIS Designer, the Script component has two modes: metadata design mode and code design mode.

When you open the **Script Transformation Editor**, the component enters metadata design mode. In this mode, you can select input columns, and add or configure outputs and output columns, but you cannot write code. After you have configured the component's metadata, you can switch to code design mode to write the script.

When you switch to code design mode by clicking **Edit Script**, the Script component locks metadata to prevent additional changes, and then automatically generates base code from the metadata of the inputs and outputs. After the autogenerated code is complete, you will be able to enter your custom code. Your code uses the auto-generated base classes to process input rows, to access buffers and columns in the buffers, and to retrieve connection managers and variables from the package, all as strongly-typed objects.

After entering your custom code in code design mode, you can switch back to metadata design mode. This does not delete any code that you have entered; however, subsequent changes to the metadata cause the base class to be regenerated. Afterward, your component may fail validation because objects referenced by your custom code may no longer exist or may have been modified. In this case, you must fix your code manually so that it can be compiled successfully against the regenerated base class.

Configuring the Component in Metadata Design Mode

In metadata design mode, you can select input columns, and add and configure outputs and output columns, but you cannot write code. After you have configured the component's metadata, switch to code design mode to write the script.

The properties that you must configure in the custom editor depend on the usage of the Script component. The Script component can be configured as a source, a transformation, or a destination. Depending on how the component is used, it supports either an input or outputs or both. The custom code that you will write processes the input and output rows and columns.

Inputs Columns Page of the Script Transformation Editor

The **Input Columns** page of the **Script Transformation Editor** is displayed for transformations and destinations, but not for sources. On this page, you select the available input columns that you want to make available to your custom script, and specify read-only or read/write access to them.

In the code project that will be generated based on this metadata, the BufferWrapper project item contains a class for each input, and this class contains typed accessor properties for each input column selected. For example, if you select an integer **CustomerID** column and a string **CustomerName** column from an input named **CustomerInput**, the BufferWrapper project item will contain a **CustomerInput** class that derives from **T:Microsoft.SqlServer.Dts.Pipeline.ScriptBuffer**, and the **CustomerInput** class will expose an integer property named **CustomerID** and a string property named **CustomerName**. This convention makes it possible to write code with type-checking like the following:

```
Dim currentCustomerID as Integer = CustomerInput.CustomerID

Dim currentCustomerName as String = CustomerInput.CustomerName
```

For more information about how to configure input columns for a specific type of data flow component, see the appropriate example under Developing Specific Types of Script Components.

Inputs and Outputs Page of the Script Transformation Editor

The **Input and Outputs** page of the **Script Transformation Editor** is displayed for sources, transformations, and destinations. On this page, you add, remove, and configure inputs, outputs, and output columns that you want to use in your custom script, within the following limitations:

- When used as a source, the Script component has no input and supports multiple outputs.
- When used as a transformation, the Script component supports one input and multiple outputs.
- When used as a destination, the Script component supports one input and has no outputs.

In the code project that will be generated based on this metadata, the BufferWrapper project item contains a class for each input and output. For example, if you create an output named **CustomerOutput**, the BufferWrapper project item will contain a **CustomerOutput** class that derives from **T:Microsoft.SqlServer.Dts.Pipeline.ScriptBuffer**, and the **CustomerOutput** class will contain typed accessor properties for each output column created.

You can configure output columns only on the **Input and Outputs** page. You can select input columns for transformations and destinations on the **Input Columns** page. The typed accessor properties created for you in the BufferWrapper project item will be write-only for output columns. The accessor properties for input columns will be read-only or read/write depending on the usage type that you have selected for each column on the **Input Columns** page.

For more information about configuring inputs and outputs for a specific type of data flow component see the appropriate example under Developing Specific Types of Script Components.

Note

Although you cannot directly configure an output as an error output in the Script component for automatic handling of error rows, you can reproduce the functionality of an error output by creating an additional output and using script to direct rows to this output when appropriate. For more information, see Simulating an Error Output in the Script Component.

ExclusionGroup and SynchronousInputID Properties of Outputs

The **ExclusionGroup** property has a non-zero value only in transformations with synchronous outputs, where your code performs filtering or branching and directs each row to one of the outputs that share the same non-zero **ExclusionGroup** value. For example, the transformation can direct rows either to the default output or to an error output. When you create additional outputs for this scenario, make sure to set the value of the **SynchronousInputID** property to the integer that matches the **ID** of the component's input.

The **SynchronousInputID** property has a non-zero value only in transformations with synchronous outputs. If the value of this property is zero, it means that the output is

asynchronous. For a synchronous output, where rows are passed through to the selected output or outputs without adding any new rows, this property should contain the **ID** of the component's input.

Note

- When the **Script Transformation Editor** creates the first output, the editor sets the **SynchronousInputID** property of the output to the **ID** of the component's input. However, when the editor creates subsequent outputs, the editor sets the **SynchronousInputID** properties of those outputs to zero.
- If you are creating a component with synchronous outputs, each output must have its **SynchronousInputID** property set to the **ID** of the component's input. Therefore, each output that the editor creates after the first output must have its **SynchronousInputID** value changed from zero to the **ID** of the component's input.
- If you are creating a component with asynchronous outputs, each output must have its **SynchronousInputID** property set to zero. Therefore, the first output must have its **SynchronousInputID** value changed from the **ID** of the component's input to zero.

For an example of directing rows to one of two synchronous outputs in the Script component, see Creating a Synchronous Transformation with the Script Component.

Object Names in Generated Script

The Script component parses the names of inputs and outputs, and parse the names of columns in the inputs and outputs, and based on these names generates classes and properties in the BufferWrapper project item. If the found names include characters that do not belong to the Unicode categories **UppercaseLetter**, **LowercaseLetter**, **TitlecaseLetter**, **ModifierLetter**, **OtherLetter**, or **DecimalDigitLetter**, the invalid characters are dropped in the generated names. For example, spaces are dropped, therefore two input columns that have the names **FirstName** and [**First Name**] are both interpreted as having the column name **FirstName**, with unpredictable results. To avoid this situation, the names of inputs and outputs and of input and output columns used by the Script component should contain only characters in the Unicode categories listed in this section.

Script Page of the Script Transformation Editor

On the **Script** page of the **Script Task Editor**, you assign a unique name and a description for the Script task. You can also assign values for the following properties.

Note

In SQL Server 2008 Integration Services (SSIS) and later versions, all scripts are precompiled. In previous versions, you specified whether scripts were precompiled by setting a **Precompile** property for the task.

ValidateExternalMetadata Property

The Boolean value of the **ValidateExternalMetadata** property specifies whether the component should perform validation against external data sources at design time, or whether it should

postpone validation until run time. By default, the value of this property is **True**; that is, the external metadata is validated both at design time and at run time. You may want to set the value of this property to **False** when an external data source is not available at design time: for example, when the package downloads the source or creates the destination only at run time.

ReadOnlyVariables and ReadWriteVariables Properties

You can enter comma-delimited lists of existing variables as the values of these properties to make the variables available for read-only or read/write access within the Script component code. Variables are accessed in code through the **P:Microsoft.SqlServer.Dts.Pipeline.ScriptComponent.ReadOnlyVariables** and **P:Microsoft.SqlServer.Dts.Pipeline.ScriptComponent.ReadWriteVariables** properties of the autogenerated base class. For more information, see Using Variables in the Script Component.

Note
Variable names are case-sensitive.

ScriptLanguage

You can select either Microsoft Visual Basic or Microsoft Visual C# as the programming language for the Script component.

Edit Script Button

The **Edit Script** button opens the Microsoft Visual Studio Tools for Applications (VSTA) IDE in which you write your custom script. For more information, see Coding the Script Component.

Connection Managers Page of the Script Transformation Editor

On the **Connection Managers** page of the **Script Transformation Editor**, you add and remove connection managers that you want to use in your custom script. Normally you need to reference connection managers when you create a source or destination component.

In the code project that will be generated based on this metadata, the **ComponentWrapper** project item contains a **Connections** collection class that has a typed accessor property for each selected connection manager. Each typed accessor property has the same name as the connection manager itself and returns a reference to the connection manager as an instance of **T:Microsoft.SqlServer.Dts.Runtime.Wrapper.IDTSConnectionManager100**. For example, if you have added a connection manager named `MyADONETConnection` on the **Connection Managers** page of the editor, you can obtain a reference to the connection manager in your script by using the following code:

```
Dim myADONETConnectionManager As IDTSConnectionManager100 = _
    Me.Connections.MyADONETConnection
```

For more information, see Connecting to Data Sources in the Script Component.

Stay Up to Date with Integration Services

For the latest downloads, articles, samples, and videos from Microsoft, as well as selected solutions from the community, visit the Integration Services page on MSDN:

Visit the Integration Services page on MSDN

For automatic notification of these updates, subscribe to the RSS feeds available on the page.

See Also

Coding and Debugging the Script Component

Coding and Debugging the Script Component

In SSIS Designer, the Script component has two modes: metadata design mode and code design mode. When you open the **Script Transformation Editor**, the component enters metadata design mode, in which you configure metadata and set component properties. After you have set the properties of the Script component and configured the input and outputs in metadata design mode, you can switch to code design mode to write your custom script. For more information about metadata design mode and code design mode, see Configuring the Script Component.

Writing the Script in Code Design Mode

Script Component Development Environment

To write your script, click **Edit Script** on the **Script** page of the **Script Transformation Editor** to open the Microsoft Visual Studio Tools for Applications (VSTA) IDE. The VSTA IDE includes all the standard features of the Visual Studio .NET environment, such as the color-coded Visual Studio editor, IntelliSense, and Object Browser.

Script code is written in Microsoft Visual Basic or Microsoft Visual C#. You specify the script language by setting the **ScriptLanguage** property in the **Script Transformation Editor**. If you prefer to use another programming language, you can develop a custom assembly in your language of choice and call its functionality from the code in the Script component.

The script that you create in the Script component is stored in the package definition. There is no separate script file. Therefore, the use of the Script component does not affect package deployment.

Note
While you design the package, the script code is temporarily written to a project file. Because storing sensitive information in a file is a potential security risk, we recommended that you do not include sensitive information such as passwords in the script code.

By default, **Option Strict** is disabled in the IDE.

Script Component Project Structure

The power of the Script component is that it can generate infrastructure code that reduces the amount of code that you must write. This feature relies on the fact that inputs and outputs and their columns and properties are fixed and known in advance. Therefore, any subsequent changes that you make to the component's metadata may invalidate the code that you have written. This causes compilation errors during execution of the package.

Project Items and Classes in the Script Component Project

When you switch to code design mode, the VSTA IDE opens and displays the **ScriptMain** project item. The **ScriptMain** project item contains the editable **ScriptMain** class, which serves as the entry point for the script and which is where you write your code. The code elements in the class vary depending on the programming language that you selected for the Script task.

The script project contains two additional auto-generated read-only project items:

- The **ComponentWrapper** project item contains three classes:
  - The **UserComponent** class, which inherits from **T:Microsoft.SqlServer.Dts.Pipeline.ScriptComponent** and contains the methods and properties that you will use to process data and to interact with the package. The **ScriptMain** class inherits from the **UserComponent** class.
  - A **Connections** collection class that contains references to the connections selected on the Connection Manager page of the Script Transformation Editor.
  - A **Variables** collection class that contains references to the variables entered in the **ReadOnlyVariable** and **ReadWriteVariables** properties on the **Script** page of the **Script Transformation Editor**.
- The **BufferWrapper** project item contains a class that inherits from **T:Microsoft.SqlServer.Dts.Pipeline.ScriptBuffer** for each input and output configured on the **Inputs and Outputs** page of the **Script Transformation Editor**. Each of these classes contains typed accessor properties that correspond to the configured input and output columns, and the data flow buffers that contain the columns.

For information about how to use these objects, methods, and properties, see Using the Script Component Object Model. For information about how to use the methods and properties of

these classes in a particular type of Script component, see the section Examples of Specific Types of Script Components. The example topics also contain complete code samples.

When you configure the Script component as a transformation, the **ScriptMain** project item contains the following autogenerated code. The code template also provides an overview of the Script component, and additional information on how to retrieve and manipulate SSIS objects, such as variables, events, and connections.

```
' Microsoft SQL Server Integration Services Script Component
' Write scripts using Microsoft Visual Basic 2008.
' ScriptMain is the entry point class of the script.

Imports System
Imports System.Data
Imports System.Math
Imports Microsoft.SqlServer.Dts.Pipeline.Wrapper
Imports Microsoft.SqlServer.Dts.Runtime.Wrapper

<Microsoft.SqlServer.Dts.Pipeline.SSISScriptComponentEntryPointAttribute> _
<CLSCompliant(False)> _
Public Class ScriptMain
    Inherits UserComponent

    Public Overrides Sub PreExecute()
        MyBase.PreExecute()
        '
        ' Add your code here for preprocessing or remove if not needed
        '
    End Sub

    Public Overrides Sub PostExecute()
        MyBase.PostExecute()
        '
        ' Add your code here for postprocessing or remove if not needed
        ' You can set read/write variables here, for example:
        ' Me.Variables.MyIntVar = 100
```

```
        '
    End Sub

    Public Overrides Sub Input0_ProcessInputRow(ByVal Row As Input0Buffer)
        '
        ' Add your code here
        '
    End Sub

End Class
/* Microsoft SQL Server Integration Services user script component
*  Write scripts using Microsoft Visual C# 2008.
*  ScriptMain is the entry point class of the script.*/

using System;
using System.Data;
using Microsoft.SqlServer.Dts.Pipeline.Wrapper;
using Microsoft.SqlServer.Dts.Runtime.Wrapper;

[Microsoft.SqlServer.Dts.Pipeline.SSISScriptComponentEntryPointAttribute]
public class ScriptMain : UserComponent
{

    public override void PreExecute()
    {
        base.PreExecute();
        /*
          Add your code here for preprocessing or remove if not needed
        */
    }

    public override void PostExecute()
    {
```

```
        base.PostExecute();
        /*
          Add your code here for postprocessing or remove if not needed
          You can set read/write variables here, for example:
          Variables.MyIntVar = 100
        */
    }

    public override void Input0_ProcessInputRow(Input0Buffer Row)
    {
        /*
          Add your code here
        */
    }

}
```

Additional Project Items in the Script Component Project

The Script component project can include items other than the default **ScriptMain** item. You can add classes, modules, code files, and folders to the project, and you can use folders to organize groups of items.

All the items that you add are persisted inside the package.

References in the Script Component Project

You can add references to managed assemblies by right-clicking the Script task project in **Project Explorer**, and then clicking **Add Reference**. For more information, see Referencing Other Assemblies in Scripting Solutions.

Note

You can view project references in the VSTA IDE in **Class View** or in **Project Explorer**. You open either of these windows from the **View** menu. You can add a new reference from the **Project** menu, from **Project Explorer**, or from **Class View**.

Interacting with the Package in the Script Component

The custom script that you write in the Script component can access and use variables and connection managers from the containing package through strongly-typed accessors in the auto-generated base classes. However, you must configure both variables and connection managers before entering code-design mode if you want to make them available to your script. You can also raise events and perform logging from your Script component code.

The autogenerated project items in the Script component project provide the following objects, methods, and properties for interacting with the package.

Package Feature	Access Method
Variables	Use the named and typed accessor properties in the **Variables** collection class in the **ComponentWrapper** project item, exposed through the **Variables** property of the **ScriptMain** class. The **PreExecute** method can access only read-only variables. The **PostExecute** method can access both read-only and read/write variables.
Connections	Use the named and typed accessor properties in the **Connections** collection class in the **ComponentWrapper** project item, exposed through the **Connections** property of the **ScriptMain** class.
Events	Raise events by using the **P:Microsoft.SqlServer.Dts.Pipeline.ScriptComponent.ComponentMetaData** property of the **ScriptMain** class and the **Fire<X>** methods of the **T:Microsoft.SqlServer.Dts.Pipeline.Wrapper.IDTSComponentMetaData100** interface.
Logging	Perform logging by using the **M:Microsoft.SqlServer.Dts.Pipeline.ScriptComponent.Log(System.String,System.Int32, System.Byte[])** method of the **ScriptMain** class.

Debugging the Script Component

To debug the code in your Script component, set at least one breakpoint in the code, and then close the VSTA IDE to run the package in SQL Server Data Tools (SSDT). When package execution enters the Script component, the VSTA IDE reopens and displays your code in read-only mode. After execution reaches your breakpoint, you can examine variable values and step through the remaining code.

Note

You cannot debug a Script component when you run the Script component as part of a child package that is run from an Execute Package task. Breakpoints that you set in the Script component in the child package are disregarded in these circumstances. You can debug the child package normally by running it separately.

Note

When you debug a package that contains multiple Script components, the debugger debugs one Script component. The system can debug another Script component if the debugger completes, as in the case of a Foreach Loop or For Loop container.

You can also monitor the execution of the Script component by using the following methods:

- Interrupt execution and display a modal message by using the **MessageBox.Show** method in the **System.Windows.Forms** namespace. (Remove this code after you complete the debugging process.)
- Raise events for informational messages, warnings, and errors. The **FireInformation**, **FireWarning**, and **FireError** methods display the event description in the Visual Studio **Output** window. However, the **FireProgress** method, the **Console.Write** method, and **Console.WriteLine** method do not display any information in the **Output** window. Messages from the **FireProgress** event appear on the **Progress** tab of SSIS Designer. For more information, see Raising Events in the Script Component.
- Log events or user-defined messages to enabled logging providers. For more information, see Logging in the Script Component.

If you just want to examine the output of a Script component configured as a source or as a transformation, without saving the data to a destination, you can stop the data flow with a Row Count Transformation and attach a data viewer to the output of the Script component. For information about data viewers, see Debugging Data Flow.

In This Section

For more information about coding the Script component, see the following topics in this section.

Using the Script Component Object Model

Explains how to use the objects, methods, and properties available in the Script component.

Referencing Other Assemblies in Scripting Solutions

Explains how to reference objects from the .NET Framework class library in the Script component.

Simulating an Error Output for the Script Component

Explains how to simulate an error output for rows that raise errors during processing by the Script component.

External Resources

- Blog entry, VSTA setup and configuration troubles for SSIS 2008 and R2 installations, on blogs.msdn.com.

Stay Up to Date with Integration Services

For the latest downloads, articles, samples, and videos from Microsoft, as well as selected solutions from the community, visit the Integration Services page on MSDN:

- Visit the Integration Services page on MSDN

For automatic notification of these updates, subscribe to the RSS feeds available on the page.

See Also

Configuring the Script Component in the Script Component Editor

Understanding the Script Component Object Model

As discussed in Coding the Script Component, the Script component project contains three project items:

1. The **ScriptMain** item, which contains the **ScriptMain** class in which you write your code. The **ScriptMain** class inherits from the **UserComponent** class.
2. The **ComponentWrapper** item, which contains the **UserComponent** class, an instance of **T:Microsoft.SqlServer.Dts.Pipeline.ScriptComponent** that contains the methods and properties that you will use to process data and to interact with the package. The **ComponentWrapper** item also contains **Connections** and **Variables** collection classes.
3. The **BufferWrapper** item, which contains classes that inherits from **T:Microsoft.SqlServer.Dts.Pipeline.ScriptBuffer** for each input and output, and typed properties for each column.

As you write your code in the **ScriptMain** item, you will use the objects, methods, and properties discussed in this topic. Each component will not use all the methods listed here; however, when used, they are used in the sequence shown.

The **T:Microsoft.SqlServer.Dts.Pipeline.ScriptComponent** base class does not contain any implementation code for the methods discussed in this topic. Therefore it is unnecessary, but harmless, to add a call to the base class implementation to your own implementation of the method.

For information about how to use the methods and properties of these classes in a particular type of Script component, see the section Examples of Specific Types of Script Components. The example topics also contain complete code samples.

AcquireConnections Method

Sources and destinations generally must connect to an external data source. Override the **M:Microsoft.SqlServer.Dts.Pipeline.ScriptComponent.AcquireConnections(System.Object)** method of the **T:Microsoft.SqlServer.Dts.Pipeline.ScriptComponent** base class to retrieve the connection or the connection information from the appropriate connection manager.

The following example returns a **System.Data.SqlClient.SqlConnection** from an ADO.NET connection manager.

```
Dim connMgr As IDTSConnectionManager100
Dim sqlConn As SqlConnection

Public Overrides Sub AcquireConnections(ByVal Transaction As Object)

    connMgr = Me.Connections.MyADONETConnection
    sqlConn = CType(connMgr.AcquireConnection(Nothing), SqlConnection)

End Sub
```

The following example returns a complete path and file name from a Flat File Connection Manager, and then opens the file by using a **System.IO.StreamReader**.

```
Private textReader As StreamReader
Public Overrides Sub AcquireConnections(ByVal Transaction As Object)

    Dim connMgr As IDTSConnectionManager100 = _
        Me.Connections.MyFlatFileSrcConnectionManager
    Dim exportedAddressFile As String = _
        CType(connMgr.AcquireConnection(Nothing), String)
    textReader = New StreamReader(exportedAddressFile)

End Sub
```

PreExecute Method

Override the **M:Microsoft.SqlServer.Dts.Pipeline.ScriptComponent.PreExecute** method of the **T:Microsoft.SqlServer.Dts.Pipeline.ScriptComponent** base class whenever you have processing that you must perform one time only before you start processing rows of data. For example, in a

destination, you may want to configure the parameterized command that the destination will use to insert each row of data into the data source.

```
    Dim sqlConn As SqlConnection
    Dim sqlCmd As SqlCommand
    Dim sqlParam As SqlParameter
...
    Public Overrides Sub PreExecute()

        sqlCmd = New SqlCommand("INSERT INTO Person.Address2(AddressID, City)
" & _
            "VALUES(@addressid, @city)", sqlConn)
        sqlParam = New SqlParameter("@addressid", SqlDbType.Int)
        sqlCmd.Parameters.Add(sqlParam)
        sqlParam = New SqlParameter("@city", SqlDbType.NVarChar, 30)
        sqlCmd.Parameters.Add(sqlParam)

    End Sub
    SqlConnection sqlConn;
    SqlCommand sqlCmd;
    SqlParameter sqlParam;

    public override void PreExecute()
    {

        sqlCmd = new SqlCommand("INSERT INTO Person.Address2(AddressID, City)
" + "VALUES(@addressid, @city)", sqlConn);
        sqlParam = new SqlParameter("@addressid", SqlDbType.Int);
        sqlCmd.Parameters.Add(sqlParam);
        sqlParam = new SqlParameter("@city", SqlDbType.NVarChar, 30);
        sqlCmd.Parameters.Add(sqlParam);

    }
```

Processing Inputs and Outputs

Processing Inputs

Script components that are configured as transformations or destinations have one input.

What the BufferWrapper Project Item Provides

For each input that you have configured, the **BufferWrapper** project item contains a class that derives from **T:Microsoft.SqlServer.Dts.Pipeline.ScriptBuffer** and has the same name as the input. Each input buffer class contains the following properties, functions, and methods:

- Named, typed accessor properties for each selected input column. These properties are read-only or read/write depending on the **Usage Type** specified for the column on the **Input Columns** page of the **Script Transformation Editor**.
- A **<column>_IsNull** property for each selected input column. This property is also read-only or read/write depending on the **Usage Type** specified for the column.
- A **DirectRowTo<outputbuffer>** method for each configured output. You will use these methods when filtering rows to one of several outputs in the same **ExclusionGroup**.
- A **NextRow** function to get the next input row, and an **EndOfRowset** function to determine whether the last buffer of data has been processed. You typically do not need these functions when you use the input processing methods implemented in the **UserComponent** base class. The next section provides more information about the **UserComponent** base class.

What the ComponentWrapper Project Item Provides

The ComponentWrapper project item contains a class named **UserComponent** that derives from **T:Microsoft.SqlServer.Dts.Pipeline.ScriptComponent**. The **ScriptMain** class in which you write your custom code derives in turn from **UserComponent**. The **UserComponent** class contains the following methods:

- An overridden implementation of the **ProcessInput** method. This is the method that the data flow engine calls next at run time after the **PreExecute** method, and it may be called multiple times. **ProcessInput** hands off processing to the **<inputbuffer>_ProcessInput** method. Then the **ProcessInput** method checks for the end of the input buffer and, if the end of the buffer has been reached, calls the overridable **FinishOutputs** method and the private **MarkOutputsAsFinished** method. The **MarkOutputsAsFinished** method then calls **SetEndOfRowset** on the last output buffer.
- An overridable implementation of the **<inputbuffer>_ProcessInput** method. This default implementation simply loops through each input row and calls **<inputbuffer>_ProcessInputRow**.
- An overridable implementation of the **<inputbuffer>_ProcessInputRow** method. The default implementation is empty. This is the method that you will normally override to write your custom data processing code.

What Your Custom Code Should Do

You can use the following methods to process input in the **ScriptMain** class:

- Override **<inputbuffer>_ProcessInputRow** to process the data in each input row as it passes through.

- Override **<inputbuffer>_ProcessInput** only if you have to do something additional while looping through input rows. (For example, you have to test for **EndOfRowset** to take some other action after all rows have been processed.) Call **<inputbuffer>_ProcessInputRow** to perform the row processing.
- Override **FinishOutputs** if you have to do something to the outputs before they are closed.

The **ProcessInput** method ensures that these methods are called at the appropriate times.

Processing Outputs

Script components configured as sources or transformations have one or more outputs.

What the BufferWrapper Project Item Provides

For each output that you have configured, the BufferWrapper project item contains a class that derives from **T:Microsoft.SqlServer.Dts.Pipeline.ScriptBuffer** and has the same name as the output. Each input buffer class contains the following properties and methods:

- Named, typed, write-only accessor properties for each output column.
- A write-only **<column>_IsNull** property for each selected output column that you can use to set the column value to **null**.
- An **AddRow** method to add an empty new row to the output buffer.
- A **SetEndOfRowset** method to let the data flow engine know that no more buffers of data are expected. There is also an **EndOfRowset** function to determine whether the current buffer is the last buffer of data. You generally do not need these functions when you use the input processing methods implemented in the **UserComponent** base class.

What the ComponentWrapper Project Item Provides

The ComponentWrapper project item contains a class named **UserComponent** that derives from **T:Microsoft.SqlServer.Dts.Pipeline.ScriptComponent**. The **ScriptMain** class in which you write your custom code derives in turn from **UserComponent**. The **UserComponent** class contains the following methods:

- An overridden implementation of the **PrimeOutput** method. The data flow engine calls this method before **ProcessInput** at run time, and it is only called one time. **PrimeOutput** hands off processing to the **CreateNewOutputRows** method. Then, if the component is a source (that is, the component has no inputs), **PrimeOutput** calls the overridable **FinishOutputs** method and the private **MarkOutputsAsFinished** method. The **MarkOutputsAsFinished** method calls **SetEndOfRowset** on the last output buffer.
- An overridable implementation of the **CreateNewOutputRows** method. The default implementation is empty. This is the method that you will normally override to write your custom data processing code.

What Your Custom Code Should Do

You can use the following methods to process outputs in the **ScriptMain** class:

- Override **CreateNewOutputRows** only when you can add and populate output rows before processing input rows. For example, you can use **CreateNewOutputRows** in a source, but in

a transformation with asynchronous outputs, you should call **AddRow** during or after the processing of input data.

- Override **FinishOutputs** if you have to do something to the outputs before they are closed.

The **PrimeOutput** method ensures that these methods are called at the appropriate times.

PostExecute Method

Override the **M:Microsoft.SqlServer.Dts.Pipeline.ScriptComponent.PostExecute** method of the **T:Microsoft.SqlServer.Dts.Pipeline.ScriptComponent** base class whenever you have processing that you must perform one time only after you have processed the rows of data. For example, in a source, you may want to close the **System.Data.SqlClient.SqlDataReader** that you have used to load data into the data flow.

Important

The collection of **ReadWriteVariables** is available only in the **PostExecute** method. Therefore you cannot directly increment the value of a package variable as you process each row of data. Instead, increment the value of a local variable, and set the value of the package variable to the value of the local variable in the **PostExecute** method after all data has been processed.

ReleaseConnections Method

Sources and destinations typically must connect to an external data source. Override the **M:Microsoft.SqlServer.Dts.Pipeline.ScriptComponent.ReleaseConnections** method of the **T:Microsoft.SqlServer.Dts.Pipeline.ScriptComponent** base class to close and release the connection that you have opened previously in the **M:Microsoft.SqlServer.Dts.Pipeline.ScriptComponent.AcquireConnections(System.Object)** method.

```
    Dim connMgr As IDTSConnectionManager100
...
    Public Overrides Sub ReleaseConnections()

        connMgr.ReleaseConnection(sqlConn)

    End Sub
```

```
    IDTSConnectionManager100 connMgr;

    public override void ReleaseConnections()
    {

        connMgr.ReleaseConnection(sqlConn);
```

```
}
```

Stay Up to Date with Integration Services

For the latest downloads, articles, samples, and videos from Microsoft, as well as selected solutions from the community, visit the Integration Services page on MSDN:

Visit the Integration Services page on MSDN

For automatic notification of these updates, subscribe to the RSS feeds available on the page.

See Also

Configuring the Script Component in the Script Component Editor

Coding the Script Component

Using Variables in the Script Component

Variables store values that a package and its containers, tasks, and event handlers can use at run time. For more information, see Integration Services Variables.

You can make existing variables available for read-only or read/write access by your custom script by entering comma-delimited lists of variables in the **ReadOnlyVariables** and **ReadWriteVariables** fields on the **Script** page of the **Script Transformation Editor**. Keep in mind that variable names are case-sensitive. Use the **Value** property to read from and write to individual variables. The Script component handles any required locking behind the scenes as your script manipulates the variables at run time.

Important

The collection of **ReadWriteVariables** is only available in the **PostExecute** method to maximize performance and minimize the risk of locking conflicts. Therefore you cannot directly increment the value of a package variable as you process each row of data. Increment the value of a local variable instead, and set the value of the package variable to the value of the local variable in the **PostExecute** method after all data has been processed. You can also use the

P:Microsoft.SqlServer.Dts.Pipeline.ScriptComponent.VariableDispenser property to work around this limitation, as described later in this topic. However, writing directly to a package variable as each row is processed will negatively impact performance and increase the risk of locking conflicts.

For more information about the **Script** page of the **Script Transformation Editor**, see Configuring the Script Component and Script Transformation Editor (Script Page).

The Script component creates a **Variables** collection class in the **ComponentWrapper** project item with a strongly-typed accessor property for the value of each preconfigured variable where the property has the same name as the variable itself. This collection is exposed through the **Variables** property of the **ScriptMain** class. The accessor property provides read-only or read/write permission to the value of the variable as appropriate. For example, if you have added an integer variable named `MyIntegerVariable` to the **ReadOnlyVariables** list, you can retrieve its value in your script by using the following code:

```
Dim myIntegerVariableValue As Integer = Me.Variables.MyIntegerVariable
```

You can also use the **P:Microsoft.SqlServer.Dts.Pipeline.ScriptComponent.VariableDispenser** property, accessed by calling `Me.VariableDispenser`, to work with variables in the Script component. In this case you are not using the typed and named accessor properties for variables, but accessing the variables directly. When using the **P:Microsoft.SqlServer.Dts.Pipeline.ScriptComponent.VariableDispenser**, you must handle both the locking semantics and the casting of data types for variable values in your own code. You have to use the **P:Microsoft.SqlServer.Dts.Pipeline.ScriptComponent.VariableDispenser** property instead of the named and typed accessor properties if you want to work with a variable that is not available at design time but is created programmatically at run time.

See Also

Integration Services Variables

Using Variables in Packages

Connecting to Data Sources in the Script Component

A connection manager is a convenient unit that encapsulates and stores the information that is required to connect to a data source of a particular type. For more information, see Integration Services Connections.

You can make existing connection managers available for access by the custom script in the source or destination component by clicking the **Add** and **Remove** buttons on the **Connection Managers** page of the **Script Transformation Editor**. However, you must write your own custom code to load or save your data, and possibly to open and close the connection to the data source. For more information about the **Connection Managers** page of the **Script Transformation Editor**, see Configuring the Script Component and Script Transformation Editor (Connection Managers Page).

The Script component creates a **Connections** collection class in the **ComponentWrapper** project item that contains a strongly-typed accessor for each connection manager that has the same name as the connection manager itself. This collection is exposed through the **Connections** property of the **ScriptMain** class. The accessor property returns a reference to the connection manager as an instance of **T:Microsoft.SqlServer.Dts.Runtime.Wrapper.IDTSConnectionManager100**. For example, if you have added a connection manager named `MyADONETConnection` on the Connection Managers page of the dialog box, you can obtain a reference to it in your script by adding the following code:

```
Dim myADONETConnectionManager As IDTSConnectionManager100 = _
    Me.Connections.MyADONETConnection
```

Note

You must know the type of connection that is returned by the connection manager before you call **AcquireConnection**. Because the Script task has **Option Strict** enabled, you must cast the connection, which is returned as type **Object**, to the appropriate connection type before you can use it.

Next, you call the **AcquireConnection** method of the specific connection manager to obtain either the underlying connection or the information that is required to connect to the data source. For example, you obtain a reference to the **System.Data.SqlConnection** wrapped by an ADO.NET connection manager by using the following code:

```
Dim myADOConnection As SqlConnection = _
    CType(MyADONETConnectionManager.AcquireConnection(Nothing),
SqlConnection)
```

In contrast, the same call to a flat file connection manager returns only the path and file name of the file data source.

```
Dim myFlatFile As String = _
    CType(MyFlatFileConnectionManager.AcquireConnection(Nothing), String)
```

You then must provide this path and file name to a **System.IO.StreamReader** or **Streamwriter** to read or write the data in the flat file.

Important

- When you write managed code in a Script component, you cannot call the **AcquireConnection** method of connection managers that return unmanaged objects, such as the OLE DB connection manager and the Excel connection manager. However, you can read the **ConnectionString** property of these connection managers, and connect to the data source directly in your code by using the connection string of an OLEDB **connection** from the **System.Data.OleDb** namespace.
- If you need to call the **AcquireConnection** method of a connection manager that returns an unmanaged object, use an ADO.NET connection manager. When you configure the ADO.NET connection manager to use an OLE DB provider, it connects by using the .NET Framework Data Provider for OLE DB. In this case, the **AcquireConnection** method returns a **System.Data.OleDb.OleDbConnection** instead of an unmanaged object. To configure an ADO.NET connection manager for use with an Excel data source, select the Microsoft OLE DB Provider for Jet, specify an Excel workbook, and then enter `Excel 8.0` (for Excel 97 and later) as the value of **Extended Properties** on the **All** page of the **Connection Manager** dialog box.

For more information about how to use connection managers with the script component, see Creating a Source with the Script Component and Creating a Destination with the Script Component.

See Also

Integration Services Connections

Creating Connection Managers

Raising Events in the Script Component

Events provide a way to report errors, warnings, and other information, such as task progress or status, to the containing package. The package provides event handlers for managing event notifications. The Script component can raise events by calling methods on the **P:Microsoft.SqlServer.Dts.Pipeline.ScriptComponent.ComponentMetaData** property of the **ScriptMain** class. For more information about how Integration Services packages handle events, see DTS Event Handlers.

Events can be logged to any log provider that is enabled in the package. Log providers store information about events in a data store. The Script component can also use the **M:Microsoft.SqlServer.Dts.Pipeline.ScriptComponent.Log(System.String,System.Int32,System.Byte[])** method to log information to a log provider without raising an event. For more information about how to use the **M:Microsoft.SqlServer.Dts.Pipeline.ScriptComponent.Log(System.String,System.Int32,System.Byte[])** method, see the following section.

To raise an event, the Script task calls one of the following methods of the **T:Microsoft.SqlServer.Dts.Pipeline.Wrapper.IDTSComponentMetaData100** interface exposed by the **P:Microsoft.SqlServer.Dts.Pipeline.ScriptComponent.ComponentMetaData** property:

Event	Description
M:Microsoft.SqlServer.Dts.Pipeline.Wrapper.IDTSComponentMetaData100.FireCustomEvent(System.String,System.String,System.Object[]@,System.String,System.Boolean@)	Raises a user-defined custom event in the package.
M:Microsoft.SqlServer.Dts.Pipeline.Wrapper.IDTSComponentMetaData100.FireError(System.Int32,System.String,System.String,System.String,System.Int32,System.Boolean@)	Informs the packa

Event	Description
	ge of an error condition.
M:Microsoft.SqlServer.Dts.Pipeline.Wrapper.IDTSComponentMetaData100.FireInformation(System.Int32,System.String,System.String,System.String,System.Int32,System.Boolean@)	Provides information to the user.
M:Microsoft.SqlServer.Dts.Pipeline.Wrapper.IDTSComponentMetaData100.FireProgress(System.String,System.Int32,System.Int32,System.Int32,System.String,System.Boolean@)	Informs the package of the progress of the component.
M:Microsoft.SqlServer.Dts.Pipeline.Wrapper.IDTSComponentMetaData100.FireWarning(System.Int32,System.String,System.String,System.String,System.Int32)	Informs the package that the component is in a state that warrants user

Event	Description
	notification, but is not an error condition.

Here is a simple example of raising an Error event:

```
Dim myMetadata as IDTSComponentMetaData100
myMetaData = Me.ComponentMetaData
myMetaData.FireError(...)
```

Stay Up to Date with Integration Services

For the latest downloads, articles, samples, and videos from Microsoft, as well as selected solutions from the community, visit the Integration Services page on MSDN:

Visit the Integration Services page on MSDN

For automatic notification of these updates, subscribe to the RSS feeds available on the page.

See Also

Integration Services Event Handlers

Add an Event Handler to a Package

Logging in the Script Component

Logging in Integration Services packages lets you save detailed information about execution progress, results, and problems by recording predefined events or user-defined messages for later analysis. The Script component can use the **M:Microsoft.SqlServer.Dts.Pipeline.ScriptComponent.Log(System.String,System.Int32,System.Byte[])** method of the **ScriptMain** class to log user-defined data. If logging is enabled, and the **ScriptComponentLogEntry** event is selected for logging on the **Details** tab of the **Configure SSIS Logs** dialog box, a single call to the **M:Microsoft.SqlServer.Dts.Pipeline.ScriptComponent.Log(System.String,System.Int32,System.Byte[])** method stores the event information in all the log providers that have been configured for the data flow task.

Here is a simple example of logging:

```
Dim bt(0) As Byte
Me.Log("Test Log Event", _
    0, _
    bt)
```

Note

Although you can perform logging directly from your Script component, you may want to consider implementing events rather than logging. When using events, not only can you enable the logging of event messages, but you can respond to the event with default or user-defined event handlers.

For more information about logging, see Integration Services Logging.

Stay Up to Date with Integration Services

For the latest downloads, articles, samples, and videos from Microsoft, as well as selected solutions from the community, visit the Integration Services page on MSDN:

Visit the Integration Services page on MSDN

For automatic notification of these updates, subscribe to the RSS feeds available on the page.

See Also

Integration Services Logging

Developing Specific Types of Script Components

The Script component is a configurable tool that you can use in the data flow of a package to fill almost any requirement that is not met by the sources, transformations, and destinations that are included with Integration Services. This section contains Script component code samples that demonstrate the four options for configuring the Script component:

- As a source.
- As a transformation with synchronous outputs.
- As a transformation with asynchronous outputs.
- As a destination.

For additional examples of the Script component, see Script Component Examples.

In This Section

Creating a Source with the Script Component

Explains and demonstrates how to create a data flow source by using the Script component.

Creating a Synchronous Transformation with the Script Component

Explains and demonstrates how to create a data flow transformation with synchronous outputs by using the Script component. This kind of transformation modifies rows of data in place as they pass through the component.

Creating an Asynchronous Transformation with the Script Component

Explains and demonstrates how to create a data flow transformation with asynchronous outputs by using the Script component. This kind of transformation has to read all rows of data before it can add more information, such as calculated aggregates, to the data that passes through the component.

Creating a Destination with the Script Component

Explains and demonstrates how to create a data flow destination by using the Script component.

Stay Up to Date with Integration Services

For the latest downloads, articles, samples, and videos from Microsoft, as well as selected solutions from the community, visit the Integration Services page on MSDN:

Visit the Integration Services page on MSDN

For automatic notification of these updates, subscribe to the RSS feeds available on the page.

See Also

Comparing Scripting Solutions and Custom Objects

Developing Specific Types of Data Flow Components

Creating a Source with the Script Component

You use a source component in the data flow of an Integration Services package to load data from a data source to pass on to downstream transformations and destinations. Ordinarily you connect to the data source through an existing connection manager.

For an overview of the Script component, see Programming the Script Component.

The Script component and the infrastructure code that it generates for you simplify significantly the process of developing a custom data flow component. However, to understand how the Script component works, you may find it useful to read through the steps that are involved in

developing a custom data flow component. See the section Extending the Data Flow with Custom Components, especially the topic Creating a Source Component.

Getting Started with a Source Component

When you add a Script component to the Data Flow pane of SSIS Designer, the **Select Script Component Type** dialog box opens and prompts you to select a Source, Destination, or Transformation script. In this dialog box, select **Source**.

Configuring a Source Component in Metadata-Design Mode

After selecting to create a source component, you configure the component by using the **Script Transformation Editor**. For more information, see Configuring the Script Component.

A data flow source component has no inputs and supports one or more outputs. Configuring the outputs for the component is one of the steps that you must complete in metadata design mode, by using the **Script Transformation Editor**, before you write your custom script.

You can also specify the script language by setting the **ScriptLanguage** property on the **Script** page of the **Script Transformation Editor**.

Note
To set the default script language for Script components and Script Tasks, use the **Scripting language** option on the **General** page of the **Options** dialog box. For more information, see General Page.

Adding Connection Managers

Ordinarily a source component uses an existing connection manager to connect to the data source from which it loads data into the data flow. On the **Connection Managers** page of the **Script Transformation Editor**, click **Add** to add the appropriate connection manager.

However, a connection manager is only a convenient unit that encapsulates and stores the information that it must have to connect to a data source of a particular type. You must write your own custom code to load or save your data, and possibly to open and close the connection to the data source also.

For general information about how to use connection managers with the Script component, see Connecting to Data Sources in the Script Component.

For more information about the **Connection Managers** page of the **Script Transformation Editor**, see Script Transformation Editor (Connection Managers Page).

Configuring Outputs and Output Columns

A source component has no inputs and supports one or more outputs. On the **Inputs and Outputs** page of the **Script Transformation Editor**, a single output has been created by default, but no output columns have been created. On this page of the editor, you may need or want to configure the following items.

- You must add and configure output columns manually for each output. Select the Output Columns folder for each output, and then use the **Add Column** and **Remove Column** buttons to manage the output columns for each output of the source component. Later, you

will refer to the output columns in your script by the names that you assign here, by using the typed accessor properties created for you in the auto-generated code.

- You may want to create one or more additional outputs, such as a simulated error output for rows that contain unexpected values. Use the **Add Output** and **Remove Output** buttons to manage the outputs of the source component. All input rows are directed to all available outputs unless you also specify an identical non-zero value for the **ExclusionGroup** property of those outputs where you intend to direct each row to only one of the outputs that share the same **ExclusionGroup** value. The particular integer value selected to identify the **ExclusionGroup** is not significant.

  Note
  You can also use a non-zero **ExclusionGroup** property value with a single output when you do not want to output all rows. In this case, however, you must explicitly call the **DirectRowTo<outputbuffer>** method for each row that you want to send to the output.

- You may want to assign a friendly name to the outputs. Later, you will refer to the outputs by their names in your script, by using the typed accessor properties created for you in the auto-generated code.
- Ordinarily multiple outputs in the same **ExclusionGroup** have the same output columns. However, if you are creating a simulated error output, you may want to add more columns to store error information. For information about how the data flow engine processes error rows, see Creating and Using Error Outputs. In the Script component, however, you must write your own code to fill the additional columns with appropriate error information. For more information, see Simulating an Error Output for the Script Component.

For more information about the **Inputs and Outputs** page of the **Script Transformation Editor**, see Script Transformation Editor (Inputs and Outputs Page).

Adding Variables

If there are any existing variables whose values you want to use in your script, you can add them in the **ReadOnlyVariables** and **ReadWriteVariables** property fields on the **Script** page of the **Script Transformation Editor**.

When you enter multiple variables in the property fields, separate the variable names by commas. You can also enter multiple variables by clicking the ellipsis (**...**) button next to the **ReadOnlyVariables** and **ReadWriteVariables** property fields and selecting variables in the **Select variables** dialog box.

For general information about how to use variables with the Script component, see Using Variables in the Script Component.

For more information about the **Script** page of the **Script Transformation Editor**, see Script Transformation Editor (Script Page).

Scripting a Source Component in Code-Design Mode

After you have configured the metadata for your component, open the Microsoft Visual Studio Tools for Applications (VSTA) IDE to code your custom script. To open VSTA, click **Edit Script** on

the **Script** page of the **Script Transformation Editor**. You can write your script by using either Microsoft Visual Basic or Microsoft Visual C#, depending on the script language selected for the **ScriptLanguage** property.

For important information that applies to all kinds of components created by using the Script component, see Coding the Script Component.

Understanding the Auto-generated Code

When you open the VSTA IDE after creating and configuring a source component, the editable **ScriptMain** class appears in the code editor. You write your custom code in the **ScriptMain** class.

The **ScriptMain** class includes a stub for the **CreateNewOutputRows** method. The **CreateNewOutputRows** is the most important method in a source component.

If you open the **Project Explorer** window in VSTA, you can see that the Script component has also generated read-only **BufferWrapper** and **ComponentWrapper** project items. The **ScriptMain** class inherits from **UserComponent** class in the **ComponentWrapper** project item.

At run time, the data flow engine invokes the **PrimeOutput** method in the **UserComponent** class, which overrides the **M:Microsoft.SqlServer.Dts.Pipeline.ScriptComponent.PrimeOutput(System.Int32,System.Int32[],Microsoft.SqlServer.Dts.Pipeline.PipelineBuffer[])** method of the **T:Microsoft.SqlServer.Dts.Pipeline.ScriptComponent** parent class. The **PrimeOutput** method in turn calls the following methods:

1. The **CreateNewOutputRows** method, which you override in **ScriptMain** to add rows from the data source to the output buffers, which are empty at first.
2. The **FinishOutputs** method, which is empty by default. Override this method in **ScriptMain** to perform any processing that is required to complete the output.
3. The private **MarkOutputsAsFinished** method, which calls the **M:Microsoft.SqlServer.Dts.Pipeline.ScriptBuffer.SetEndOfRowset** method of the **T:Microsoft.SqlServer.Dts.Pipeline.ScriptBuffer** parent class to indicate to the data flow engine that the output is finished. You do not have to call **SetEndOfRowset** explicitly in your own code.

Writing Your Custom Code

To finish creating a custom source component, you may want to write script in the following methods available in the **ScriptMain** class.

1. Override the **AcquireConnections** method to connect to the external data source. Extract the connection object, or the required connection information, from the connection manager.
2. Override the **PreExecute** method to load data, if you can load all the source data at the same time. For example, you can execute a **SqlCommand** against an ADO.NET connection to a SQL Server database and load all the source data at the same time into a **SqlDataReader**. If you must load the source data one row at a time (for example, when

reading a text file), you can load the data as you loop through rows in **CreateNewOutputRows**.

3. Use the overridden **CreateNewOutputRows** method to add new rows to the empty output buffers and to fill in the values of each column in the new output rows. Use the **AddRow** method of each output buffer to add an empty new row, and then set the values of each column. Typically you copy values from the columns loaded from the external source.
4. Override the **PostExecute** method to finish processing the data. For example, you can close the **SqlDataReader** that you used to load data.
5. Override the **ReleaseConnections** method to disconnect from the external data source, if required.

Examples

The following examples demonstrate the custom code that is required in the **ScriptMain** class to create a source component.

Note

These examples use the **Person.Address** table in the **AdventureWorks** sample database and pass its first and fourth columns, the **int AddressID** and **nvarchar(30) City** columns, through the data flow. The same data is used in the source, transformation, and destination samples in this section. Additional prerequisites and assumptions are documented for each example.

ADO.NET Source Example

This example demonstrates a source component that uses an existing ADO.NET connection manager to load data from a SQL Server table into the data flow.

If you want to run this sample code, you must configure the package and the component as follows:

1. Create an ADO.NET connection manager that uses the **SqlClient** provider to connect to the **AdventureWorks** database.
2. Add a new Script component to the Data Flow designer surface and configure it as a source.
3. Open the **Script Transformation Editor**. On the **Inputs and Outputs** page, rename the default output with a more descriptive name such as **MyAddressOutput**, and add and configure the two output columns, **AddressID** and **City**.

   

   Note

   Be sure to change the data type of the **City** output column to DT_WSTR.
4. On the **Connection Managers** page, add or create the ADO.NET connection manager and give it a name such as **MyADONETConnection**.
5. On the **Script** page, click **Edit Script** and enter the script that follows. Then close the script development environment and the **Script Transformation Editor**.
6. Create and configure a destination component, such as a SQL Server destination, or the sample destination component demonstrated in Creating a Destination with the Script

Component, that expects the **AddressID** and **City** columns. Then connect the source component to the destination. (You can connect a source directly to a destination without any transformations.) You can create a destination table by running the following Transact-SQL command in the **AdventureWorks** database:

```
CREATE TABLE [Person].[Address2](
    [AddressID] [int] NOT NULL,
    [City] [nvarchar](30) NOT NULL
)
```

7. Run the sample.

```
Imports System.Data.SqlClient
...
Public Class ScriptMain
    Inherits UserComponent

    Dim connMgr As IDTSConnectionManager100
    Dim sqlConn As SqlConnection
    Dim sqlReader As SqlDataReader

    Public Overrides Sub AcquireConnections(ByVal Transaction As
Object)

        connMgr = Me.Connections.MyADONETConnection
        sqlConn = CType(connMgr.AcquireConnection(Nothing),
SqlConnection)

    End Sub

    Public Overrides Sub PreExecute()

        Dim cmd As New SqlCommand("SELECT AddressID, City,
StateProvinceID FROM Person.Address", sqlConn)
        sqlReader = cmd.ExecuteReader

    End Sub
```

```
    Public Overrides Sub CreateNewOutputRows()

        Do While sqlReader.Read
            With MyAddressOutputBuffer
                .AddRow()
                .AddressID = sqlReader.GetInt32(0)
                .City = sqlReader.GetString(1)
            End With
        Loop

    End Sub

    Public Overrides Sub PostExecute()

        sqlReader.Close()

    End Sub

    Public Overrides Sub ReleaseConnections()

        connMgr.ReleaseConnection(sqlConn)

    End Sub

End Class

using System.Data.SqlClient;
public class ScriptMain:
    UserComponent

{
    IDTSConnectionManager100 connMgr;
    SqlConnection sqlConn;
```

```
    SqlDataReader sqlReader;

    public override void AcquireConnections(object Transaction)
    {
        connMgr = this.Connections.MyADONETConnection;
        sqlConn = (SqlConnection)connMgr.AcquireConnection(null);

    }

    public override void PreExecute()
    {

        SqlCommand cmd = new SqlCommand("SELECT AddressID, City,
StateProvinceID FROM Person.Address", sqlConn);
        sqlReader = cmd.ExecuteReader();

    }

    public override void CreateNewOutputRows()
    {

        while (sqlReader.Read())
        {
            {
                MyAddressOutputBuffer.AddRow();
                MyAddressOutputBuffer.AddressID =
sqlReader.GetInt32(0);
                MyAddressOutputBuffer.City = sqlReader.GetString(1);
            }
        }

    }

    public override void PostExecute()
```

```
        {
            sqlReader.Close();
        }
        public override void ReleaseConnections()
        {
            connMgr.ReleaseConnection(sqlConn);
        }
    }
```

Flat File Source Example

This example demonstrates a source component that uses an existing Flat File connection manager to load data from a flat file into the data flow. The flat file source data is created by exporting it from SQL Server.

If you want to run this sample code, you must configure the package and the component as follows:

1. Use the SQL Server Import and Export Wizard to export the **Person.Address** table from the **AdventureWorks** sample database to a comma-delimited flat file. This sample uses the file name ExportedAddresses.txt.
2. Create a Flat File connection manager that connects to the exported data file.
3. Add a new Script component to the Data Flow designer surface and configure it as a source.
4. Open the **Script Transformation Editor**. On the **Inputs and Outputs** page, rename the default output with a more descriptive name such as **MyAddressOutput**. Add and configure the two output columns, **AddressID** and **City**.
5. On the **Connection Managers** page, add or create the Flat File connection manager, using a descriptive name such as **MyFlatFileSrcConnectionManager**.
6. On the **Script** page, click **Edit Script** and enter the script that follows. Then close the script development environment and the **Script Transformation Editor**.
7. Create and configure a destination component, such as a SQL Server destination, or the sample destination component demonstrated in Creating a Destination with the Script Component. Then connect the source component to the destination. (You can connect a source directly to a destination without any transformations.) You can create a destination table by running the following Transact-SQL command in the **AdventureWorks** database:

```
CREATE TABLE [Person].[Address2](
    [AddressID] [int] NOT NULL,
    [City] [nvarchar](30) NOT NULL
)
```

8. Run the sample.

```
Imports System.IO
...
Public Class ScriptMain
    Inherits UserComponent

    Private textReader As StreamReader
    Private exportedAddressFile As String

    Public Overrides Sub AcquireConnections(ByVal Transaction As
Object)

        Dim connMgr As IDTSConnectionManager100 = _
            Me.Connections.MyFlatFileSrcConnectionManager
        exportedAddressFile = _
            CType(connMgr.AcquireConnection(Nothing), String)

    End Sub

    Public Overrides Sub PreExecute()
        MyBase.PreExecute()
        textReader = New StreamReader(exportedAddressFile)
    End Sub

    Public Overrides Sub CreateNewOutputRows()

        Dim nextLine As String
        Dim columns As String()

        Dim delimiters As Char()
```

```
        delimiters = ",".ToCharArray

        nextLine = textReader.ReadLine
        Do While nextLine IsNot Nothing
            columns = nextLine.Split(delimiters)
            With MyAddressOutputBuffer
                .AddRow()
                .AddressID = columns(0)
                .City = columns(3)
            End With
            nextLine = textReader.ReadLine
        Loop

    End Sub

    Public Overrides Sub PostExecute()
        MyBase.PostExecute()
        textReader.Close()

    End Sub

End Class

using System.IO;
public class ScriptMain:
    UserComponent

{
    private StreamReader textReader;
    private string exportedAddressFile;

    public override void AcquireConnections(object Transaction)
    {
```

```
        IDTSConnectionManager100 connMgr =
this.Connections.MyFlatFileSrcConnectionManager;
        exportedAddressFile = (string)connMgr.AcquireConnection(null);

    }

    public override void PreExecute()
    {
        base.PreExecute();
        textReader = new StreamReader(exportedAddressFile);
    }

    public override void CreateNewOutputRows()
    {

        string nextLine;
        string[] columns;

        char[] delimiters;
        delimiters = ",".ToCharArray();

        nextLine = textReader.ReadLine();
        while (nextLine != null)
        {
            columns = nextLine.Split(delimiters);
            {
                MyAddressOutputBuffer.AddRow();
                MyAddressOutputBuffer.AddressID = columns[0];
                MyAddressOutputBuffer.City = columns[3];
            }
            nextLine = textReader.ReadLine();
        }
```

```
        }

        public override void PostExecute()
        {

            base.PostExecute();
            textReader.Close();

        }

    }
```

Stay Up to Date with Integration Services

For the latest downloads, articles, samples, and videos from Microsoft, as well as selected solutions from the community, visit the Integration Services page on MSDN:

Visit the Integration Services page on MSDN

For automatic notification of these updates, subscribe to the RSS feeds available on the page.

See Also

Creating a Destination with the Script Component

Developing a Custom Source Component

Creating a Synchronous Transformation with the Script Component

You use a transformation component in the data flow of an Integration Services package to modify and analyze data as it passes from source to destination. A transformation with synchronous outputs processes each input row as it passes through the component. A transformation with asynchronous outputs waits until it has received all input rows to complete

its processing. This topic discusses a synchronous transformation. For information about asynchronous transformations, see Creating an Asynchronous Transformation with the Script Component. For more information about the difference between synchronous and asynchronous components, see Understanding Synchronous and Asynchronous Transformations.

For an overview of the Script component, see Programming the Script Component.

The Script component and the infrastructure code that it generates for you simplify significantly the process of developing a custom data flow component. However, to understand how the Script component works, you may find it useful to read the steps that you must follow in developing a custom data flow component in the section on Extending the Data Flow with Custom Components, and especially Creating a Transformation Component with Synchronous Outputs.

Getting Started with a Synchronous Transformation Component

When you add a Script component to the Data Flow pane of SSIS Designer, the **Select Script Component Type** dialog box opens and prompts you to select a Source, Destination, or Transformation component type. In this dialog box, select **Transformation**.

Configuring a Synchronous Transformation Component in Metadata-Design Mode

After you select the option to create a transformation component, you configure the component by using the **Script Transformation Editor**. For more information, see Configuring the Script Component.

To set the script language for the Script component, you set the **ScriptLanguage** property on the **Script** page of the **Script Transformation Editor**.

Note
To set the default scripting language for the Script component, use the **Scripting language** option on the **General** page of the **Options** dialog box. For more information, see General Page.

A data flow transformation component has one input, and supports one or more outputs. Configuring the input and outputs for the component is one of the steps that you must complete in metadata design mode, by using the **Script Transformation Editor**, before you write your custom script.

Configuring Input Columns

A transformation component has one input.

On the **Input Columns** page of the **Script Transformation Editor,** the column list shows the available columns from the output of the upstream component in the data flow. Select the columns that you want to transform or pass through. Mark any columns that you want to transform in place as Read/Write.

For more information about the **Input Columns** page of the **Script Transformation Editor**, see Script Transformation Editor (Input Columns Page).

Configuring Inputs, Outputs, and Output Columns

A transformation component supports one or more outputs.

On the **Inputs and Outputs** page of the **Script Transformation Editor**, you can see that a single output has been created, but the output has no columns. On this page of the editor, you may need or want to configure the following items.

- Create one or more additional outputs, such as a simulated error output for rows that contain unexpected values. Use the **Add Output** and **Remove Output** buttons to manage the outputs of your synchronous transformation component. All input rows are directed to all available outputs unless you indicate that you intend to redirect each row to one output or the other. You indicate that you intend to redirect rows by specifying a non-zero integer value for the **ExclusionGroup** property on the outputs. The specific integer value entered in **ExclusionGroup** to identify the outputs is not significant, but you must use the same integer consistently for the specified group of outputs.

  Note
  You can also use a non-zero **ExclusionGroup** property value with a single output when you do not want to output all rows. However, in this case, you must explicitly call the **DirectRowTo<outputbuffer>** method for each row that you want to send to the output.
- Assign a more descriptive name to the input and outputs. The Script component uses these names to generate the typed accessor properties that you will use to refer to the input and outputs in your script.
- Leave columns as is for synchronous transformations. Typically a synchronous transformation does not add columns to the data flow. Data is modified in place in the buffer, and the buffer is passed on to the next component in the data flow. If this is the case, you do not have to add and configure output columns explicitly on the transformation's outputs. The outputs appear in the editor without any explicitly defined columns.
- Add new columns to simulated error outputs for row-level errors. Ordinarily multiple outputs in the same **ExclusionGroup** have the same set of output columns. However, if you are creating a simulated error output, you may want to add more columns to contain error information. For information about how the data flow engine processes error rows, see Creating and Using Error Outputs. Note that in the Script component you must write your own code to fill the additional columns with appropriate error information. For more information, see Simulating an Error Output for the Script Component.

For more information about the **Inputs and Outputs** page of the **Script Transformation Editor**, see Script Transformation Editor (Inputs and Outputs Page).

Adding Variables

If you want to use existing variables in your script, you can add them in the **ReadOnlyVariables** and **ReadWriteVariables** property fields on the **Script** page of the **Script Transformation Editor**.

When you add multiple variables in the property fields, separate the variable names by commas. You can also select multiple variables by clicking the ellipsis (**...**) button next to the

ReadOnlyVariables and **ReadWriteVariables** property fields, and then selecting the variables in the **Select variables** dialog box.

For general information about how to use variables with the Script component, see Using Variables in the Script Component.

For more information about the **Script** page of the **Script Transformation Editor**, see Script Transformation Editor (Script Page).

Scripting a Synchronous Transformation Component in Code-Design Mode

After you have configured the metadata for your component, you can write your custom script. In the **Script Transformation Editor**, on the **Script** page, click **Edit Script** to open the Microsoft Visual Studio Tools for Applications (VSTA) IDE where you can add your custom script. The scripting language that you use depends on whether you selected Microsoft Visual Basic or Microsoft Visual C# as the script language for the **ScriptLanguage** property on the **Script** page.

For important information that applies to all kinds of components created by using the Script component, see Coding the Script Component.

Understanding the Auto-generated Code

When you open the VSTA IDE after you create and configuring a transformation component, the editable **ScriptMain** class appears in the code editor with a stub for the **ProcessInputRow** method. The **ScriptMain** class is where you will write your custom code, and **ProcessInputRow** is the most important method in a transformation component.

If you open the **Project Explorer** window in VSTA, you can see that the Script component has also generated read-only **BufferWrapper** and **ComponentWrapper** project items. The **ScriptMain** class inherits from the **UserComponent** class in the **ComponentWrapper** project item.

At run time, the data flow engine invokes the **ProcessInput** method in the **UserComponent** class, which overrides the **M:Microsoft.SqlServer.Dts.Pipeline.ScriptComponent.ProcessInput(System.Int32,Microsoft.SqlServer.Dts.Pipeline.PipelineBuffer)** method of the **T:Microsoft.SqlServer.Dts.Pipeline.ScriptComponent** parent class. The **ProcessInput** method in turn loops through the rows in the input buffer and calls the **ProcessInputRow** method one time for each row.

Writing Your Custom Code

A transformation component with synchronous outputs is the simplest of all data flow components to write. For example, the single-output example shown later in this topic consists of the following custom code:

```
Row.City = UCase(Row.City)
```

```
Row.City = (Row.City).ToUpper();
```

To finish creating a custom synchronous transformation component, you use the overridden **ProcessInputRow** method to transform the data in each row of the input buffer. The data flow engine passes this buffer, when full, to the next component in the data flow.

Depending on your requirements, you may also want to write script in the **PreExecute** and **PostExecute** methods, available in the **ScriptMain** class, to perform preliminary or final processing.

Working with Multiple Outputs

Directing input rows to one of two or more possible outputs does not require much more custom code than the single-output scenario discussed earlier. For example, the two-output example shown later in this topic consists of the following custom code:

```
Row.City = UCase(Row.City)
If Row.City = "REDMOND" Then
    Row.DirectRowToMyRedmondAddresses()
Else
    Row.DirectRowToMyOtherAddresses()
End If
```

```
    Row.City = (Row.City).ToUpper();

    if (Row.City=="REDMOND")
    {
        Row.DirectRowToMyRedmondAddresses();
    }
    else
    {
        Row.DirectRowToMyOtherAddresses();
    }
```

In this example, the Script component generates the **DirectRowTo<OutputBufferX>** methods for you, based on the names of the outputs that you configured. You can use similar code to direct error rows to a simulated error output.

Examples

The examples here demonstrate the custom code that is required in the **ScriptMain** class to create a synchronous transformation component.

Note

These examples use the **Person.Address** table in the **AdventureWorks** sample database and pass its first and fourth columns, the **int AddressID** and **nvarchar(30) City** columns, through the data flow. The same data is used in the source, transformation, and destination samples in this section. Additional prerequisites and assumptions are documented for each example.

Single Output Synchronous Transformation Example

This example demonstrates a synchronous transformation component with a single output. This transformation passes through the **AddressID** column and converts the **City** column to uppercase.

If you want to run this sample code, you must configure the package and the component as follows:

1. Add a new Script component to the Data Flow designer surface and configure it as a transformation.
2. Connect the output of a source or of another transformation to the new transformation component in SSIS Designer. This output should provide data from the **Person.Address** table of the **AdventureWorks** sample database that contains the **AddressID** and **City** columns.
3. Open the **Script Transformation Editor**. On the **Input Columns** page, select the **AddressID** and **City** columns. Mark the **City** column as Read/Write.
4. On the **Inputs and Outputs** page, rename the input and output with more descriptive names, such as **MyAddressInput** and **MyAddressOutput**. Notice that the **SynchronousInputID** of the output corresponds to the **ID** of the input. Therefore you do not have to add and configure output columns.
5. On the **Script** page, click **Edit Script** and enter the script that follows. Then close the script development environment and the **Script Transformation Editor**.
6. Create and configure a destination component that expects the **AddressID** and **City** columns, such as a SQL Server destination, or the sample destination component demonstrated in Creating a Destination with the Script Component. Then connect the output of the transformation to the destination component. You can create a destination table by running the following Transact-SQL command in the **AdventureWorks** database:

```
CREATE TABLE [Person].[Address2](
    [AddressID] [int] NOT NULL,
    [City] [nvarchar](30) NOT NULL
)
```

7. Run the sample.

```
Public Class ScriptMain
    Inherits UserComponent

    Public Overrides Sub MyAddressInput_ProcessInputRow(ByVal Row As
MyAddressInputBuffer)

        Row.City = UCase(Row.City)

    End Sub
```

```
End Class
public class ScriptMain:
    UserComponent


{
    public override void MyAddressInput_ProcessInputRow(MyAddressInputBuffer
Row)
    {

        Row.City = (Row.City).ToUpper();


    }


}
```

Two-Output Synchronous Transformation Example

This example demonstrates a synchronous transformation component with two outputs. This transformation passes through the **AddressID** column and converts the **City** column to uppercase. If the city name is Redmond, it directs the row to one output; it directs all other rows to another output.

If you want to run this sample code, you must configure the package and the component as follows:

1. Add a new Script component to the Data Flow designer surface and configure it as a transformation.
2. Connect the output of a source or of another transformation to the new transformation component in SSIS Designer. This output should provide data from the **Person.Address** table of the **AdventureWorks** sample database that contains at least the **AddressID** and **City** columns.
3. Open the **Script Transformation Editor**. On the **Input Columns** page, select the **AddressID** and **City** columns. Mark the **City** column as Read/Write.
4. On the **Inputs and Outputs** page, create a second output. After you add the new output, make sure that you set its **SynchronousInputID** to the **ID** of the input. This property is already set on the first output, which is created by default. For each output, set the **ExclusionGroup** property to the same non-zero value to indicate that you will split the input rows between two mutually exclusive outputs. You do not have to add any output columns to the outputs.

5. Rename the input and outputs with more descriptive names, such as **MyAddressInput**, **MyRedmondAddresses**, and **MyOtherAddresses**.
6. On the **Script** page, click **Edit Script** and enter the script that follows. Then close the script development environment and the **Script Transformation Editor**.
7. Create and configure two destination components that expect the **AddressID** and **City** columns, such as a SQL Server destination, a Flat File destination, or the sample destination component demonstrated in Creating a Destination with the Script Component. Then connect each of the outputs of the transformation to one of the destination components. You can create destination tables by running a Transact-SQL command similar to the following (with unique table names) in the **AdventureWorks** database:

```
CREATE TABLE [Person].[Address2](
    [AddressID] [int] NOT NULL,
    [City] [nvarchar](30) NOT NULL
```

8. Run the sample.

```
Public Class ScriptMain
    Inherits UserComponent

    Public Overrides Sub MyAddressInput_ProcessInputRow(ByVal Row As
MyAddressInputBuffer)

        Row.City = UCase(Row.City)

        If Row.City = "REDMOND" Then
            Row.DirectRowToMyRedmondAddresses()
        Else
            Row.DirectRowToMyOtherAddresses()
        End If

    End Sub

End Class
public class ScriptMain:
    UserComponent

public override void MyAddressInput_ProcessInputRow(MyAddressInputBuffer Row)
```

```
    {

        Row.City = (Row.City).ToUpper();

        if (Row.City == "REDMOND")
        {
            Row.DirectRowToMyRedmondAddresses();
        }
        else
        {
            Row.DirectRowToMyOtherAddresses();
        }

    }
}
```

Stay Up to Date with Integration Services

For the latest downloads, articles, samples, and videos from Microsoft, as well as selected solutions from the community, visit the Integration Services page on MSDN:

Visit the Integration Services page on MSDN

For automatic notification of these updates, subscribe to the RSS feeds available on the page.

See Also

Understanding Synchronous and Asynchronous Outputs

Creating an Asynchronous Transformation with the Script Component

Developing a Custom Transformation Component with Synchronous Outputs

Creating an Asynchronous Transformation with the Script Component

You use a transformation component in the data flow of an Integration Services package to modify and analyze data as it passes from source to destination. A transformation with synchronous outputs processes each input row as it passes through the component. A transformation with asynchronous outputs may wait to complete its processing until the transformation has received all input rows, or the transformation may output certain rows before it has received all input rows. This topic discusses an asynchronous transformation. If your processing requires a synchronous transformation, see Developing a Custom Transformation Component with Asynchronous Outputs. For more information about the differences between synchronous and asynchronous components, see Understanding Synchronous and Asynchronous Transformations.

For an overview of the Script component, see Programming the Script Component.

The Script component and the infrastructure code that it generates for you simplify the process of developing a custom data flow component. However, to understand how the Script component works, you may find it useful to read through the steps that you must follow in developing a custom data flow component in the Extending the Data Flow with Custom Components section, and especially Creating a Transformation Component with Synchronous Outputs.

Getting Started with an Asynchronous Transformation Component

When you add a Script component to the Data Flow tab of SSIS Designer, the **Select Script Component Type** dialog box appears, prompting you to preconfigure the component as a source, transformation, or destination. In this dialog box, select **Transformation**.

Configuring an Asynchronous Transformation Component in Metadata-Design Mode

After you select the option to create a transformation component, you configure the component by using the **Script Transformation Editor**. For more information, see Configuring the Script Component.

To select the script language that the Script component will use, you set the **ScriptLanguage** property on the **Script** page of the **Script Transformation Editor** dialog box.

Note
To set the default scripting language for the Script component, use the **Scripting language** option on the **General** page of the **Options** dialog box. For more information, see General Page.

A data flow transformation component has one input and supports one or more outputs. Configuring the input and outputs of your component is one of the steps that you must complete in metadata design mode, by using the **Script Transformation Editor**, before you write your custom script.

Configuring Input Columns

A transformation component created by using the Script component has a single input.

On the **Input Columns** page of the **Script Transformation Editor**, the columns list shows the available columns from the output of the upstream component in the data flow. Select the columns that you want to transform or pass through. Mark any columns that you want to transform in place as Read/Write.

For more information about the **Input Columns** page of the **Script Transformation Editor**, see Script Transformation Editor (Input Columns Page).

Configuring Inputs, Outputs, and Output Columns

A transformation component supports one or more outputs.

Frequently a transformation with asynchronous outputs has two outputs. For example, when you count the number of addresses located in a specific city, you may want to pass the address data through to one output, while sending the result of the aggregation to another output. The aggregation output also requires a new output column.

On the **Inputs and Outputs** page of the **Script Transformation Editor**, you see that a single output has been created by default, but no output columns have been created. On this page of the editor, you can configure the following items:

- You may want to create one or more additional outputs, such as an output for the result of an aggregation. Use the **Add Output** and **Remove Output** buttons to manage the outputs of your asynchronous transformation component. Set the **SynchronousInputID** property of each output to zero to indicate that the output does not simply pass through data from an upstream component or transform it in place in the existing rows and columns. It is this setting that makes the outputs asynchronous to the input.
- You may want to assign a friendly name to the input and outputs. The Script component uses these names to generate the typed accessor properties that you will use to refer to the input and outputs in your script.
- Frequently an asynchronous transformation adds columns to the data flow. When the **SynchronousInputID** property of an output is zero, indicating that the output does not simply pass through data from an upstream component or transform it in place in the existing rows and columns, you must add and configure output columns explicitly on the output. Output columns do not have to have the same names as the input columns to which they are mapped.
- You may want to add more columns to contain additional information. You must write your own code to fill the additional columns with data. For information about reproducing the behavior of a standard error output, see Simulating an Error Output for the Script Component.

For more information about the **Inputs and Outputs** page of the **Script Transformation Editor**, see Script Transformation Editor (Inputs and Outputs Page).

Adding Variables

If there are any existing variables whose values you want to use in your script, you can add them in the **ReadOnlyVariables** and **ReadWriteVariables** property fields on the **Script** page of the **Script Transformation Editor**.

When you add multiple variables in the property fields, separate the variable names by commas. You can also select multiple variables by clicking the ellipsis (**...**) button next to the **ReadOnlyVariables** and **ReadWriteVariables** property fields, and then selecting the variables in the **Select variables** dialog box.

For general information about how to use variables with the Script component, see Using Variables in the Script Component.

For more information about the **Script** page of the **Script Transformation Editor**, see Script Transformation Editor (Script Page).

Scripting an Asynchronous Transformation Component in Code-Design Mode

After you have configured all the metadata for your component, you can write your custom script. In the **Script Transformation Editor**, on the **Script** page, click **Edit Script** to open the Microsoft Visual Studio Tools for Applications (VSTA) IDE where you can add your custom script. The scripting language that you use depends on whether you selected Microsoft Visual Basic or Microsoft Visual C# as the script language for the **ScriptLanguage** property on the **Script** page.

For important information that applies to all kinds of components created by using the Script component, see Coding the Script Component.

Understanding the Auto-generated Code

When you open the VSTA IDE after creating and configuring a transformation component, the editable **ScriptMain** class appears in the code editor with stubs for the **ProcessInputRow** and the **CreateNewOutputRows** methods. The **ScriptMain** class is where you will write your custom code, and **ProcessInputRow** is the most important method in a transformation component. The **CreateNewOutputRows** method is more typically used in a source component, which is like an asynchronous transformation in that both components must create their own output rows.

If you open the VSTA **Project Explorer** window, you can see that the Script component has also generated read-only **BufferWrapper** and **ComponentWrapper** project items. The **ScriptMain** class inherits from the **UserComponent** class in the **ComponentWrapper** project item.

At run time, the data flow engine calls the **PrimeOutput** method in the **UserComponent** class, which overrides the **M:Microsoft.SqlServer.Dts.Pipeline.ScriptComponent.PrimeOutput(System.Int32,System.Int32[],Microsoft.SqlServer.Dts.Pipeline.PipelineBuffer[])** method of the **T:Microsoft.SqlServer.Dts.Pipeline.ScriptComponent** parent class. The **PrimeOutput** method in turn calls the **CreateNewOutputRows** method.

Next, the data flow engine invokes the **ProcessInput** method in the **UserComponent** class, which overrides the **M:Microsoft.SqlServer.Dts.Pipeline.ScriptComponent.ProcessInput(System.Int32,Microsoft.SqlServer.Dts.Pipeline.PipelineBuffer)** method of the **T:Microsoft.SqlServer.Dts.Pipeline.ScriptComponent** parent class. The **ProcessInput** method in turn loops through the rows in the input buffer and calls the **ProcessInputRow** method one time for each row.

Writing Your Custom Code

To finish creating a custom asynchronous transformation component, you must use the overridden **ProcessInputRow** method to process the data in each row of the input buffer. Because the outputs are not synchronous to the input, you must explicitly write rows of data to the outputs.

In an asynchronous transformation, you can use the **AddRow** method to add rows to the output as appropriate from within the **ProcessInputRow** or **ProcessInput** methods. You do not have to use the **CreateNewOutputRows** method. If you are writing a single row of results, such as aggregation results, to a particular output, you can create the output row beforehand by using the **CreateNewOutputRows** method, and fill in its values later after processing all input rows. However it is not useful to create multiple rows in the **CreateNewOutputRows** method, because the Script component only lets you use the current row in an input or output. The **CreateNewOutputRows** method is more important in a source component where there are no input rows to process.

You may also want to override the **ProcessInput** method itself, so that you can do additional preliminary or final processing before or after you loop through the input buffer and call **ProcessInputRow** for each row. For example, one of the code examples in this topic overrides **ProcessInput** to count the number of addresses in a specific city as **ProcessInputRow** loops through rows**.** The example writes the summary value to the second output after all rows have been processed. The example completes the output in **ProcessInput** because the output buffers are no longer available when **PostExecute** is called.

Depending on your requirements, you may also want to write script in the **PreExecute** and **PostExecute** methods available in the **ScriptMain** class to perform any preliminary or final processing.

Note

If you were developing a custom data flow component from scratch, it would be important to override the **PrimeOutput** method to cache references to the output buffers so that you could add rows of data to the buffers later. In the Script component, this is not necessary because you have an automatically generated class representing each output buffer in the **BufferWrapper** project item.

Example

This example demonstrates the custom code that is required in the **ScriptMain** class to create an asynchronous transformation component.

Note

These examples use the **Person.Address** table in the **AdventureWorks** sample database and pass its first and fourth columns, the **int AddressID** and **nvarchar(30) City** columns, through the data flow. The same data is used in the source, transformation, and destination samples in this section. Additional prerequisites and assumptions are documented for each example.

This example demonstrates an asynchronous transformation component with two outputs. This transformation passes through the **AddressID** and **City** columns to one output, while it counts

the number of addresses located in a specific city (Redmond, Washington, U.S.A.), and then outputs the resulting value to a second output.

If you want to run this sample code, you must configure the package and the component as follows:

1. Add a new Script component to the Data Flow designer surface and configure it as a transformation.
2. Connect the output of a source or of another transformation to the new transformation component in the designer. This output should provide data from the **Person.Address** table of the **AdventureWorks** sample database that contains at least the **AddressID** and **City** columns.
3. Open the **Script Transformation Editor**. On the **Input Columns** page, select the **AddressID** and **City** columns.
4. On the **Inputs and Outputs** page, add and configure the **AddressID** and **City** output columns on the first output. Add a second output, and add an output column for the summary value on the second output. Set the SynchronousInputID property of the first output to 0, because this example copies each input row explicitly to the first output. The SynchronousInputID property of the newly-created output is already set to 0.
5. Rename the input, the outputs, and the new output column to give them more descriptive names. The example uses **MyAddressInput** as the name of the input, **MyAddressOutput** and **MySummaryOutput** for the outputs, and **MyRedmondCount** for the output column on the second output.
6. On the **Script** page, click **Edit Script** and enter the script that follows. Then close the script development environment and the **Script Transformation Editor**.
7. Create and configure a destination component for the first output that expects the **AddressID** and **City** columns, such as a SQL Server destination, or the sample destination component demonstrated in Creating a Destination with the Script Component, . Then connect the first output of the transformation, **MyAddressOutput**, to the destination component. You can create a destination table by running the following Transact-SQL command in the **AdventureWorks** database:

```
CREATE TABLE [Person].[Address2](
    [AddressID] [int] NOT NULL,
    [City] [nvarchar](30) NOT NULL
)
```

8. Create and configure another destination component for the second output. Then connect the second output of the transformation, **MySummaryOutput**, to the destination component. Because the second output writes a single row with a single value, you can easily configure a destination with a Flat File connection manager that connects to a new file that has a single column. In the example, this destination column is named **MyRedmondCount**.
9. Run the sample.

```
Public Class ScriptMain
    Inherits UserComponent

    Private myRedmondAddressCount As Integer

    Public Overrides Sub CreateNewOutputRows()

        MySummaryOutputBuffer.AddRow()

    End Sub

    Public Overrides Sub MyAddressInput_ProcessInput(ByVal Buffer As
MyAddressInputBuffer)

        While Buffer.NextRow()
            MyAddressInput_ProcessInputRow(Buffer)
        End While

        If Buffer.EndOfRowset Then
            MyAddressOutputBuffer.SetEndOfRowset()
            MySummaryOutputBuffer.MyRedmondCount = myRedmondAddressCount
            MySummaryOutputBuffer.SetEndOfRowset()
        End If

    End Sub

    Public Overrides Sub MyAddressInput_ProcessInputRow(ByVal Row As
MyAddressInputBuffer)

        With MyAddressOutputBuffer
            .AddRow()
            .AddressID = Row.AddressID
            .City = Row.City
        End With
```

```
        If Row.City.ToUpper = "REDMOND" Then
            myRedmondAddressCount += 1
        End If

    End Sub

End Class
public class ScriptMain:
    UserComponent

{
    private int myRedmondAddressCount;

    public override void CreateNewOutputRows()
    {

        MySummaryOutputBuffer.AddRow();

    }

    public override void MyAddressInput_ProcessInput(MyAddressInputBuffer
Buffer)
    {

        while (Buffer.NextRow())
        {
            MyAddressInput_ProcessInputRow(Buffer);
        }

        if (Buffer.EndOfRowset())
        {
            MyAddressOutputBuffer.SetEndOfRowset();
            MySummaryOutputBuffer.MyRedmondCount = myRedmondAddressCount;
```

```
            MySummaryOutputBuffer.SetEndOfRowset();
        }

    }

    public override void MyAddressInput_ProcessInputRow(MyAddressInputBuffer
Row)
    {

        {
            MyAddressOutputBuffer.AddRow();
            MyAddressOutputBuffer.AddressID = Row.AddressID;
            MyAddressOutputBuffer.City = Row.City;
        }

        if (Row.City.ToUpper() == "REDMOND")
        {
            myRedmondAddressCount += 1;
        }

    }

}
```

Stay Up to Date with Integration Services

For the latest downloads, articles, samples, and videos from Microsoft, as well as selected solutions from the community, visit the Integration Services page on MSDN:

Visit the Integration Services page on MSDN

For automatic notification of these updates, subscribe to the RSS feeds available on the page.

See Also

Understanding Synchronous and Asynchronous Outputs

Creating a Synchronous Transformation with the Script Component

Developing a Custom Transformation Component with Asynchronous Outputs

Creating a Destination with the Script Component

You use a destination component in the data flow of an Integration Services package to save data received from upstream sources and transformations to a data source. Ordinarily the destination component connects to the data source through an existing connection manager.

For an overview of the Script component, see Developing a Custom Destination Component.

The Script component and the infrastructure code that it generates for you simplify significantly the process of developing a custom data flow component. However, to understand how the Script component works, you may find it useful to read through the steps for developing a custom data flow components in the Extending the Data Flow with Custom Components section, and especially Creating a Destination Component.

Getting Started with a Destination Component

When you add a Script component to the Data Flow tab of SSIS Designer, the **Select Script Component Type** dialog box opens and prompts you to select a **Source**, **Destination**, or **Transformation** script. In this dialog box, select **Destination**.

Next, connect the output of a transformation to the destination component in SSIS Designer. For testing, you can connect a source directly to a destination without any transformations.

Configuring a Destination Component in Metadata-Design Mode

After you select the option to create a destination component, you configure the component by using the **Script Transformation Editor**. For more information, see Configuring the Script Component.

To select the script language that the Script destination will use, you set the **ScriptLanguage** property on the **Script** page of the **Script Transformation Editor** dialog box.

Note
To set the default scripting language for the Script component, use the **Scripting language** option on the **General** page of the **Options** dialog box. For more information, see General Page.

A data flow destination component has one input and no outputs. Configuring the input for the component is one of the steps that you must complete in metadata design mode, by using the **Script Transformation Editor**, before you write your custom script.

Adding Connection Managers

Ordinarily a destination component uses an existing connection manager to connect to the data source to which it saves data from the data flow. On the **Connection Managers** page of the **Script Transformation Editor**, click **Add** to add the appropriate connection manager.

However, a connection manager is just a convenient unit that encapsulates and stores the information that is required to connect to a data source of a particular type. You must write your own custom code to load or save your data, and possibly to open and close the connection to the data source.

For general information about how to use connection managers with the Script component, see Connecting to Data Sources in the Script Component.

For more information about the **Connection Managers** page of the **Script Transformation Editor**, see Script Transformation Editor (Connection Managers Page).

Configuring Inputs and Input Columns

A destination component has one input and no outputs.

On the **Input Columns** page of the **Script Transformation Editor**, the column list shows the available columns from the output of the upstream component in the data flow. Select the columns that you want to save.

For more information about the **Input Columns** page of the **Script Transformation Editor**, see Script Transformation Editor (Input Columns Page).

The **Inputs and Outputs** page of the **Script Transformation Editor** shows a single input, which you can rename. You will refer to the input by its name in your script by using the accessor property created in the auto-generated code.

For more information about the **Inputs and Outputs** page of the **Script Transformation Editor**, see Script Transformation Editor (Inputs and Outputs Page).

Adding Variables

If you want to use existing variables in your script, you can add them in the **ReadOnlyVariables** and **ReadWriteVariables** property fields on the **Script** page of the **Script Transformation Editor**.

When you add multiple variables in the property fields, separate the variable names by commas. You can also select multiple variables by clicking the ellipsis (**...**) button next to the **ReadOnlyVariables** and **ReadWriteVariables** property fields, and then selecting the variables in the **Select variables** dialog box.

For general information about how to use variables with the Script component, see Using Variables in the Script Component.

For more information about the **Script** page of the **Script Transformation Editor**, see Script Transformation Editor (Script Page).

Scripting a Destination Component in Code-Design Mode

After you have configured the metadata for your component, you can write your custom script. In the **Script Transformation Editor**, on the **Script** page, click **Edit Script** to open the Microsoft Visual Studio Tools for Applications (VSTA) IDE where you can add your custom script. The scripting language that you use depends on whether you selected Microsoft Visual Basic or Microsoft Visual C# as the script language for the **ScriptLanguage** property on the **Script** page.

For important information that applies to all kinds of components created by using the Script component, see Coding the Script Component.

Understanding the Auto-generated Code

When you open the VSTA IDE after you create and configuring a destination component, the editable **ScriptMain** class appears in the code editor with a stub for the **ProcessInputRow** method. The **ScriptMain** class is where you will write your custom code, and **ProcessInputRow** is the most important method in a destination component.

If you open the **Project Explorer** window in VSTA, you can see that the Script component has also generated read-only **BufferWrapper** and **ComponentWrapper** project items. The **ScriptMain** class inherits from **UserComponent** class in the **ComponentWrapper** project item.

At run time, the data flow engine invokes the **ProcessInput** method in the **UserComponent** class, which overrides the **M:Microsoft.SqlServer.Dts.Pipeline.ScriptComponent.ProcessInput(System.Int32,Microsoft.SqlServer.Dts.Pipeline.PipelineBuffer)** method of the **T:Microsoft.SqlServer.Dts.Pipeline.ScriptComponent** parent class. The **ProcessInput** method in turn loops through the rows in the input buffer and calls the **ProcessInputRow** method one time for each row.

Writing Your Custom Code

To finish creating a custom destination component, you may want to write script in the following methods available in the **ScriptMain** class.

1. Override the **AcquireConnections** method to connect to the external data source. Extract the connection object, or the required connection information, from the connection manager.

2. Override the **PreExecute** method to prepare to save the data. For example, you may want to create and configure a **SqlCommand** and its parameters in this method.
3. Use the overridden **ProcessInputRow** method to copy each input row to the external data source. For example, for a SQL Server destination, you can copy the column values into the parameters of a **SqlCommand** and execute the command one time for each row. For a flat file destination, you can write the values for each column to a **StreamWriter**, separating the values by the column delimiter.
4. Override the **PostExecute** method to disconnect from the external data source, if required, and to perform any other required cleanup.

Examples

The examples that follow demonstrate code that is required in the **ScriptMain** class to create a destination component.

Note

These examples use the **Person.Address** table in the **AdventureWorks** sample database and pass its first and fourth columns, the **int *AddressID*** and **nvarchar(30) City** columns, through the data flow. The same data is used in the source, transformation, and destination samples in this section. Additional prerequisites and assumptions are documented for each example.

ADO.NET Destination Example

This example demonstrates a destination component that uses an existing ADO.NET connection manager to save data from the data flow into a SQL Server table.

If you want to run this sample code, you must configure the package and the component as follows:

1. Create an ADO.NET connection manager that uses the **SqlClient** provider to connect to the **AdventureWorks** database.
2. Create a destination table by running the following Transact-SQL command in the **AdventureWorks** database:

```
CREATE TABLE [Person].[Address2](
    [AddressID] [int] NOT NULL,
    [City] [nvarchar](30) NOT NULL
)
```

3. Add a new Script component to the Data Flow designer surface and configure it as a destination.
4. Connect the output of an upstream source or transformation to the destination component in SSIS Designer. (You can connect a source directly to a destination without any transformations.) This output should provide data from the **Person.Address** table of the **AdventureWorks** sample database that contains at least the **AddressID** and **City** columns.

5. Open the **Script Transformation Editor**. On the **Input Columns** page, select the **AddressID** and **City** input columns.
6. On the **Inputs and Outputs** page, rename the input with a more descriptive name such as **MyAddressInput**.
7. On the **Connection Managers** page, add or create the ADO.NET connection manager with a name such as **MyADONETConnectionManager**.
8. On the **Script** page, click **Edit Script** and enter the script that follows. Then close the script development environment.
9. Close the **Script Transformation Editor** and run the sample.

```
Imports System.Data.SqlClient
...
Public Class ScriptMain
    Inherits UserComponent

    Dim connMgr As IDTSConnectionManager100
    Dim sqlConn As SqlConnection
    Dim sqlCmd As SqlCommand
    Dim sqlParam As SqlParameter

    Public Overrides Sub AcquireConnections(ByVal Transaction As Object)

        connMgr = Me.Connections.MyADONETConnectionManager
        sqlConn = CType(connMgr.AcquireConnection(Nothing), SqlConnection)

    End Sub

    Public Overrides Sub PreExecute()

        sqlCmd = New SqlCommand("INSERT INTO Person.Address2(AddressID, City)
" & _
            "VALUES(@addressid, @city)", sqlConn)
        sqlParam = New SqlParameter("@addressid", SqlDbType.Int)
        sqlCmd.Parameters.Add(sqlParam)
        sqlParam = New SqlParameter("@city", SqlDbType.NVarChar, 30)
        sqlCmd.Parameters.Add(sqlParam)
```

```
    End Sub

    Public Overrides Sub MyAddressInput_ProcessInputRow(ByVal Row As
MyAddressInputBuffer)
        With sqlCmd
            .Parameters("@addressid").Value = Row.AddressID
            .Parameters("@city").Value = Row.City
            .ExecuteNonQuery()
        End With
    End Sub

    Public Overrides Sub ReleaseConnections()

        connMgr.ReleaseConnection(sqlConn)

    End Sub

End Class
using System.Data.SqlClient;
public class ScriptMain:
    UserComponent

{
    IDTSConnectionManager100 connMgr;
    SqlConnection sqlConn;
    SqlCommand sqlCmd;
    SqlParameter sqlParam;

    public override void AcquireConnections(object Transaction)
    {

        connMgr = this.Connections.MyADONETConnectionManager;
        sqlConn = (SqlConnection)connMgr.AcquireConnection(null);
```

```
    }

    public override void PreExecute()
    {

        sqlCmd = new SqlCommand("INSERT INTO Person.Address2(AddressID, City)
" +
            "VALUES(@addressid, @city)", sqlConn);
        sqlParam = new SqlParameter("@addressid", SqlDbType.Int);
        sqlCmd.Parameters.Add(sqlParam);
        sqlParam = new SqlParameter("@city", SqlDbType.NVarChar, 30);
        sqlCmd.Parameters.Add(sqlParam);

    }

    public override void MyAddressInput_ProcessInputRow(MyAddressInputBuffer
Row)
    {
        {
            sqlCmd.Parameters["@addressid"].Value = Row.AddressID;
            sqlCmd.Parameters["@city"].Value = Row.City;
            sqlCmd.ExecuteNonQuery();
        }
    }

    public override void ReleaseConnections()
    {

        connMgr.ReleaseConnection(sqlConn);

    }

}
```

Flat File Destination Example

This example demonstrates a destination component that uses an existing Flat File connection manager to save data from the data flow to a flat file.

If you want to run this sample code, you must configure the package and the component as follows:

1. Create a Flat File connection manager that connects to a destination file. The file does not have to exist; the destination component will create it. Configure the destination file as a comma-delimited file that contains the **AddressID** and **City** columns.
2. Add a new Script component to the Data Flow designer surface and configure it as a destination.
3. Connect the output of an upstream source or transformation to the destination component in SSIS Designer. (You can connect a source directly to a destination without any transformations.) This output should provide data from the **Person.Address** table of the **AdventureWorks** sample database, and should contain at least the **AddressID** and **City** columns.
4. Open the **Script Transformation Editor**. On the **Input Columns** page, select the **AddressID** and **City** columns.
5. On the **Inputs and Outputs** page, rename the input with a more descriptive name, such as **MyAddressInput**.
6. On the **Connection Managers** page, add or create the Flat File connection manager with a descriptive name such as **MyFlatFileDestConnectionManager**.
7. On the **Script** page, click **Edit Script** and enter the script that follows. Then close the script development environment.
8. Close the **Script Transformation Editor** and run the sample.

```
Imports System.IO

...

Public Class ScriptMain

    Inherits UserComponent


    Dim copiedAddressFile As String

    Private textWriter As StreamWriter

    Private columnDelimiter As String = ","


    Public Overrides Sub AcquireConnections(ByVal Transaction As Object)


        Dim connMgr As IDTSConnectionManager100 = _

            Me.Connections.MyFlatFileDestConnectionManager
```

```
        copiedAddressFile = CType(connMgr.AcquireConnection(Nothing), String)

    End Sub

    Public Overrides Sub PreExecute()

        textWriter = New StreamWriter(copiedAddressFile, False)

    End Sub

    Public Overrides Sub MyAddressInput_ProcessInputRow(ByVal Row As
MyAddressInputBuffer)

        With textWriter
            If Not Row.AddressID_IsNull Then
                .Write(Row.AddressID)
            End If
            .Write(columnDelimiter)
            If Not Row.City_IsNull Then
                .Write(Row.City)
            End If
            .WriteLine()
        End With

    End Sub

    Public Overrides Sub PostExecute()

        textWriter.Close()

    End Sub

End Class
using System.IO;
```

```
public class ScriptMain:
    UserComponent

{
    string copiedAddressFile;
    private StreamWriter textWriter;
    private string columnDelimiter = ",";

    public override void AcquireConnections(object Transaction)
    {

        IDTSConnectionManager100 connMgr =
this.Connections.MyFlatFileDestConnectionManager;
        copiedAddressFile = (string) connMgr.AcquireConnection(null);

    }

    public override void PreExecute()
    {

        textWriter = new StreamWriter(copiedAddressFile, false);

    }

    public override void MyAddressInput_ProcessInputRow(MyAddressInputBuffer
Row)
    {

        {
            if (!Row.AddressID_IsNull)
            {
                textWriter.Write(Row.AddressID);
            }
            textWriter.Write(columnDelimiter);
```

```
            if (!Row.City_IsNull)
            {
                textWriter.Write(Row.City);
            }
            textWriter.WriteLine();
        }

    }

    public override void PostExecute()
    {

        textWriter.Close();

    }

}
```

Stay Up to Date with Integration Services

For the latest downloads, articles, samples, and videos from Microsoft, as well as selected solutions from the community, visit the Integration Services page on MSDN:

Visit the Integration Services page on MSDN

For automatic notification of these updates, subscribe to the RSS feeds available on the page.

See Also

Creating a Source with the Script Component

Additional Script Component Examples

The Script component is a configurable tool that you can use in the data flow of a package to fill almost any requirement that is not met by the sources, transformations, and destinations that are included with Integration Services. This section contains Script component code samples that demonstrate the various types of functionality that are available.

For samples that demonstrate how to configure the Script component as a basic source, transformation, or destination, see Developing Specific Types of Script Components.

Note

If you want to create components that you can more easily reuse across multiple Data Flow tasks and multiple packages, consider using the code in these Script component samples as the starting point for custom data flow components. For more information, see Extending the Data Flow with Custom Components.

In This Section

Simulating an Error Output for the Script Component

The Script component does not support a standard error output, but you can simulate a standard error output with very little additional configuration and coding.

Enhancing an Error Output by Using the Script Component

Explains and demonstrates how to add additional information to a standard error output by using the Script component.

Creating an ODBC Destination with the Script Component

Explains and demonstrates how to create an ODBC data flow destination by using the Script component.

Parsing Non-Standard Text File Formats with the Script Component

Explains and demonstrates how to parse two different non-standard text file formats into destination tables.

Stay Up to Date with Integration Services

For the latest downloads, articles, samples, and videos from Microsoft, as well as selected solutions from the community, visit the Integration Services page on MSDN:

Visit the Integration Services page on MSDN

For automatic notification of these updates, subscribe to the RSS feeds available on the page.

Simulating an Error Output for the Script Component

Although you cannot directly configure an output as an error output in the Script component for automatic handling of error rows, you can reproduce the functionality of a built-in error output by creating an additional output and using conditional logic in your script to direct rows to this output when appropriate. You may want to imitate the behavior of a built-in error output by adding two additional output columns to receive the error number and the ID of the column in which an error occurred.

If you want to add the error description that corresponds to a specific predefined Integration Services error code, you can use the **M:Microsoft.SqlServer.Dts.Pipeline.Wrapper.IDTSComponentMetaData100.GetErrorDescription(System.Int32)** method of the **T:Microsoft.SqlServer.Dts.Pipeline.Wrapper.IDTSComponentMetaData100** interface, available through the Script component's **P:Microsoft.SqlServer.Dts.Pipeline.ScriptComponent.ComponentMetaData** property.

Example

The example shown here uses a Script component configured as a transformation that has two synchronous outputs. The purpose of the Script component is to filter error rows from address data in the AdventureWorks sample database. This fictitious example assumes that we are preparing a promotion for North American customers and need to filter out addresses that are not located in North America.

To configure the example

1. Before creating the new Script component, create a connection manager and configure

a data flow source that selects address data from the AdventureWorks sample database. For this example, which only looks at the CountryRegionName column, you can simply use the Person.vStateCountryProvinceRegion view, or you can select data by joining the Person.Address, Person.StateProvince, and Person.CountryRegion tables.

2. Add a new Script component to the Data Flow designer surface and configure it as a transformation. Open the **Script Transformation Editor**.
3. On the **Script** page, set the **ScriptLanguage** property to the script language that you want to use to code the script.
4. Click **Edit Script** to open Microsoft Visual Studio Tools for Applications (VSTA).
5. In the **Input0_ProcessInputRow** method, type or paste the sample code shown below.
6. Close VSTA.
7. On the **Input Columns** page, select the columns that you want to process in the Script transformation. This example uses only the CountryRegionName column. Available input columns that you leave unselected will simply be passed through unchanged in the data flow.
8. On the **Inputs and Outputs** page, add a new, second output, and set its **SynchronousInputID** value to the ID of the input, which is also the value of the **SynchronousInputID** property of the default output. Set the **ExclusionGroup** property of both outputs to the same non-zero value (for example, 1) to indicate that each row will be directed to only one of the two outputs. Give the new error output a distinctive name, such as "MyErrorOutput."
9. Add additional output columns to the new error output to capture the desired error information, which may include the error code, the ID of the column in which the error occurred, and possibly the error description. This example creates the new columns, ErrorColumn and ErrorMessage. If you are catching predefined Integration Services errors in your own implementation, make sure to add an ErrorCode column for the error number.
10. Note the ID value of the input column or columns that the Script component will check for error conditions. This example uses this column identifier to populate the ErrorColumn value.
11. Close the **Script Transformation Editor.**
12. Attach the outputs of the Script component to suitable destinations. Flat file destinations are the easiest to configure for ad hoc testing.
13. Run the package.

```
Public Overrides Sub Input0_ProcessInputRow(ByVal Row As Input0Buffer)


  If Row.CountryRegionName <> "Canada" _
      And Row.CountryRegionName <> "United States" Then
```

```
    Row.ErrorColumn = 68 ' ID of CountryRegionName column
    Row.ErrorMessage = "Address is not in North America."
    Row.DirectRowToMyErrorOutput()

  Else

    Row.DirectRowToOutput0()

  End If

End Sub
public override void Input0_ProcessInputRow(Input0Buffer Row)
{

  if (Row.CountryRegionName!="Canada"&&Row.CountryRegionName!="United
States")

  {
    Row.ErrorColumn = 68; // ID of CountryRegionName column
    Row.ErrorMessage = "Address is not in North America.";
    Row.DirectRowToMyErrorOutput();

  }
  else
  {

    Row.DirectRowToOutput0();

  }

}
```

Stay Up to Date with Integration Services

For the latest downloads, articles, samples, and videos from Microsoft, as well as selected solutions from the community, visit the Integration Services page on MSDN:

Visit the Integration Services page on MSDN

For automatic notification of these updates, subscribe to the RSS feeds available on the page.

See Also

Handling Errors in Data

Using Error Outputs

Creating a Synchronous Transformation with the Script Component

Enhancing an Error Output with the Script Component

By default, the two extra columns in an Integration Services error output, ErrorCode and ErrorColumn, contain only numeric codes that represent an error number, and the ID of the column in which the error occurred. These numeric values may be of limited use without the corresponding error description.

This topic describes how to add an error description column to existing error output data in the data flow by using the Script component. The example adds the error description that corresponds to a specific predefined Integration Services error code by using the **M:Microsoft.SqlServer.Dts.Pipeline.Wrapper.IDTSComponentMetaData100.GetErrorDescription(System.Int32)** method of the **T:Microsoft.SqlServer.Dts.Pipeline.Wrapper.IDTSComponentMetaData100** interface, available through the **P:Microsoft.SqlServer.Dts.Pipeline.ScriptComponent.ComponentMetaData** property of the Script component.

Note

If you want to create a component that you can more easily reuse across multiple Data Flow tasks and multiple packages, consider using the code in this Script component sample as the starting point for a custom data flow component. For more information, see Extending the Data Flow with Custom Components.

Example

The example shown here uses a Script component configured as a transformation to add an error description column to existing error output data in the data flow.

For more information about how to configure the Script component for use as a transformation in the data flow, see Creating a Synchronous Transformation with the Script Component and Creating an Asynchronous Transformation with the Script Component.

To configure this Script Component example

1. Before creating the new Script component, configure an upstream component in the data flow to redirect rows to its error output when an error or truncation occurs. For testing purposes, you may want to configure a component in a manner that ensures that errors will occur—for example, by configuring a Lookup transformation between two tables where the lookup will fail.
2. Add a new Script component to the Data Flow designer surface and configure it as a transformation.
3. Connect the error output from the upstream component to the new Script component.
4. Open the **Script Transformation Editor**, and on the **Script** page, for the **ScriptLanguage** property, select the script language.
5. Click **Edit Script** to open the Microsoft Visual Studio Tools for Applications (VSTA) IDE and add the sample code shown below.
6. Close VSTA.
7. In the Script Transformation Editor, on the **Input Columns** page, select the ErrorCode column.
8. On the **Inputs and Outputs** page, add a new output column of type **String** named **ErrorDescription**. Increase the default length of the new column to 255 to support long messages.
9. Close the **Script Transformation Editor.**
10. Attach the output of the Script component to a suitable destination. A Flat File destination is the easiest to configure for ad hoc testing.
11. Run the package.

```
Public Class ScriptMain

    Inherits UserComponent

    Public Overrides Sub Input0_ProcessInputRow(ByVal Row As Input0Buffer)


  Row.ErrorDescription = _

    Me.ComponentMetaData.GetErrorDescription(Row.ErrorCode)
```

```
    End Sub
End Class
public class ScriptMain:
    UserComponent
{
    public override void Input0_ProcessInputRow(Input0Buffer Row)
    {

  Row.ErrorDescription =
this.ComponentMetaData.GetErrorDescription(Row.ErrorCode);


    }
}
```

Stay Up to Date with Integration Services

For the latest downloads, articles, samples, and videos from Microsoft, as well as selected solutions from the community, visit the Integration Services page on MSDN:

Visit the Integration Services page on MSDN

For automatic notification of these updates, subscribe to the RSS feeds available on the page.

See Also

Handling Errors in Data

Using Error Outputs

Creating a Synchronous Transformation with the Script Component

Creating an ODBC Destination with the Script Component

In SQL Server Integration Services, you typically save data to an ODBC destination by using an ADO.NET destination and the .NET Framework Data Provider for ODBC. However, you can also create an ad hoc ODBC destination for use in a single package. To create this ad hoc ODBC destination, you use the Script component as shown in the following example.

Note

If you want to create a component that you can more easily reuse across multiple Data Flow tasks and multiple packages, consider using the code in this Script component sample as the starting point for a custom data flow component. For more information, see Extending the Data Flow with Custom Components.

Example

The following example demonstrates how to create a destination component that uses an existing ODBC connection manager to save data from the data flow into a Microsoft SQL Server table.

This example is a modified version of the custom ADO.NET destination that was demonstrated in the topic, Creating a Destination with the Script Component. However, in this example, the custom ADO.NET destination has been modified to work with an ODBC connection manager and save data to an ODBC destination. These modifications also include the following changes:

- You cannot call the **AcquireConnection** method of the ODBC connection manager from managed code, because it returns a native object. Therefore, this sample uses the connection string of the connection manager to connect to the data source directly by using the managed ODBC .NET Framework Data Provider.
- The **OdbcCommand** expects positional parameters. The positions of the parameters are indicated by the question marks (?) in the text of the command. (In contrast, a **SqlCommand** expects named parameters.)

This example uses the **Person.Address** table in the **AdventureWorks** sample database. The example passes the first and fourth columns, the **int *AddressID*** and **nvarchar(30) City** columns, of this table through the data flow. This same data is used in the source, transformation, and destination samples in the topic, Developing Specific Types of Script Components.

To configure this Script Component example

1. Create an ODBC connection manager that connects to the **AdventureWorks** database.
2. Create a destination table by running the following Transact-SQL command in the **AdventureWorks** database:

```
CREATE TABLE [Person].[Address2](
    [AddressID] [int] NOT NULL,
    [City] [nvarchar](30) NOT NULL
)
```

3. Add a new Script component to the Data Flow designer surface and configure it as a destination.
4. Connect the output of an upstream source or transformation to the destination component in SSIS Designer. (You can connect a source directly to a destination without any transformations.) To ensure that this sample works, the output of the upstream component must include at least the **AddressID** and **City** columns from the **Person.Address** table of the **AdventureWorks** sample database.
5. Open the **Script Transformation Editor**. On the **Input Columns** page, select the **AddressID** and **City** columns.
6. On the **Inputs and Outputs** page, rename the input with a more descriptive name such as **MyAddressInput**.
7. On the **Connection Managers** page, add or create the ODBC connection manager with a descriptive name such as **MyODBCConnectionManager**.
8. On the **Script** page, click **Edit Script**, and then enter the script shown below in the **ScriptMain** class.
9. Close the script development environment, close the **Script Transformation Editor**, and then run the sample.

```
Imports System.Data.Odbc
...
Public Class ScriptMain
    Inherits UserComponent

    Dim odbcConn As OdbcConnection
    Dim odbcCmd As OdbcCommand
    Dim odbcParam As OdbcParameter

    Public Overrides Sub AcquireConnections(ByVal Transaction As
Object)

        Dim connectionString As String
        connectionString =
Me.Connections.MyODBCConnectionManager.ConnectionString
        odbcConn = New OdbcConnection(connectionString)
        odbcConn.Open()

    End Sub
```

```
    Public Overrides Sub PreExecute()

        odbcCmd = New OdbcCommand("INSERT INTO
Person.Address2(AddressID, City) " & _
            "VALUES(?, ?)", odbcConn)
        odbcParam = New OdbcParameter("@addressid", OdbcType.Int)
        odbcCmd.Parameters.Add(odbcParam)
        odbcParam = New OdbcParameter("@city", OdbcType.NVarChar,
30)
        odbcCmd.Parameters.Add(odbcParam)

    End Sub

    Public Overrides Sub MyAddressInput_ProcessInputRow(ByVal Row
As MyAddressInputBuffer)

        With odbcCmd
            .Parameters("@addressid").Value = Row.AddressID
            .Parameters("@city").Value = Row.City
            .ExecuteNonQuery()
        End With

    End Sub

    Public Overrides Sub ReleaseConnections()

        odbcConn.Close()

    End Sub

End Class

using System.Data.Odbc;
```

```
...
public class ScriptMain :
    UserComponent
{
    OdbcConnection odbcConn;
    OdbcCommand odbcCmd;
    OdbcParameter odbcParam;

    public override void AcquireConnections(object Transaction)
    {

        string connectionString;
        connectionString =
this.Connections.MyODBCConnectionManager.ConnectionString;
        odbcConn = new OdbcConnection(connectionString);
        odbcConn.Open();

    }

    public override void PreExecute()
    {

        odbcCmd = new OdbcCommand("INSERT INTO
Person.Address2(AddressID, City) " +
            "VALUES(?, ?)", odbcConn);
        odbcParam = new OdbcParameter("@addressid",
OdbcType.Int);
        odbcCmd.Parameters.Add(odbcParam);
        odbcParam = new OdbcParameter("@city", OdbcType.NVarChar,
30);
        odbcCmd.Parameters.Add(odbcParam);

    }
```

```
    public override void
MyAddressInput_ProcessInputRow(MyAddressInputBuffer Row)
    {

        {
            odbcCmd.Parameters["@addressid"].Value =
Row.AddressID;
            odbcCmd.Parameters["@city"].Value = Row.City;
            odbcCmd.ExecuteNonQuery();
        }

    }

    public override void ReleaseConnections()
    {

        odbcConn.Close();

    }
}
```

Stay Up to Date with Integration Services

For the latest downloads, articles, samples, and videos from Microsoft, as well as selected solutions from the community, visit the Integration Services page on MSDN:

Visit the Integration Services page on MSDN

For automatic notification of these updates, subscribe to the RSS feeds available on the page.

See Also

Creating a Destination with the Script Component

Parsing Non-Standard Text File Formats with the Script Component

When your source data is arranged in a non-standard format, you may find it more convenient to consolidate all your parsing logic in a single script than to chain together multiple Integration Services transformations to achieve the same result.

Example 1: Parsing Row-Delimited Records

Example 2: Splitting Parent and Child Records

Note

If you want to create a component that you can more easily reuse across multiple Data Flow tasks and multiple packages, consider using the code in this Script component sample as the starting point for a custom data flow component. For more information, see Extending the Data Flow with Custom Components.

Example 1: Parsing Row-Delimited Records

This example shows how to take a text file in which each column of data appears on a separate line and parse it into a destination table by using the Script component.

For more information about how to configure the Script component for use as a transformation in the data flow, see Creating a Synchronous Transformation with the Script Component and Creating an Asynchronous Transformation with the Script Component.

To configure this Script Component example

1. Create and save a text file named **rowdelimiteddata.txt** that contains the following source data:

```
FirstName: Nancy
LastName: Davolio
Title: Sales Representative
City: Seattle
StateProvince: WA

FirstName: Andrew
LastName: Fuller
Title: Vice President, Sales
City: Tacoma
StateProvince: WA

FirstName: Steven
LastName: Buchanan
Title: Sales Manager
City: London
StateProvince:
```

2. Open Management Studio and connect to an instance of SQL Server.
3. Select a destination database, and open a new query window. In the query window, execute the following script to create the destination table:

```
create table RowDelimitedData
(
FirstName varchar(32),
LastName varchar(32),
Title varchar(32),
City varchar(32),
StateProvince varchar(32)
)
```

4. Open SQL Server Data Tools and create a new Integration Services package named ParseRowDelim.dtsx.

5. Add a Flat File connection manager to the package, name it RowDelimitedData, and configure it to connect to the rowdelimiteddata.txt file that you created in a previous step.
6. Add an OLE DB connection manager to the package and configure it to connect to the instance of SQL Server and the database in which you created the destination table.
7. Add a Data Flow task to the package and click the **Data Flow** tab of SSIS Designer.
8. Add a Flat File Source to the data flow and configure it to use the RowDelimitedData connection manager. On the **Columns** page of the **Flat File Source Editor**, select the single available external column.
9. Add a Script Component to the data flow and configure it as a transformation. Connect the output of the Flat File Source to the Script Component.
10. Double-click the Script component to display the **Script Transformation Editor**.
11. On the **Input Columns** page of the **Script Transformation Editor**, select the single available input column.
12. On the **Inputs and Outputs** page of the **Script Transformation Editor**, select Output 0 and set its **SynchronousInputID** to None. Create 5 output columns, all of type string [DT_STR] with a length of 32:
    - FirstName
    - LastName
    - Title
    - City
    - StateProvince
13. On the **Script** page of the **Script Transformation Editor**, click **Edit Script** and enter the code shown in the **ScriptMain** class of the example. Close the script development environment and the **Script Transformation Editor**.
14. Add a SQL Server Destination to the data flow. Configure it to use the OLE DB connection manager and the RowDelimitedData table. Connect the output of the Script Component to this destination.
15. Run the package. After the package has finished, examine the records in the SQL Server destination table.

```
Public Overrides Sub Input0_ProcessInputRow(ByVal Row As Input0Buffer)

    Dim columnName As String
    Dim columnValue As String

    ' Check for an empty row.
    If Row.Column0.Trim.Length > 0 Then
```

```
            columnName = Row.Column0.Substring(0, Row.Column0.IndexOf(":"))
            ' Check for an empty value after the colon.
            If Row.Column0.Substring(Row.Column0.IndexOf(":")).TrimEnd.Length
> 1 Then
                ' Extract the column value from after the colon and space.
                columnValue = Row.Column0.Substring(Row.Column0.IndexOf(":")
+ 2)
                Select Case columnName
                    Case "FirstName"
                        ' The FirstName value indicates a new record.
                        Me.Output0Buffer.AddRow()
                        Me.Output0Buffer.FirstName = columnValue
                    Case "LastName"
                        Me.Output0Buffer.LastName = columnValue
                    Case "Title"
                        Me.Output0Buffer.Title = columnValue
                    Case "City"
                        Me.Output0Buffer.City = columnValue
                    Case "StateProvince"
                        Me.Output0Buffer.StateProvince = columnValue
                End Select
            End If
        End If

    End Sub
public override void Input0_ProcessInputRow(Input0Buffer Row)
    {

        string columnName;
        string columnValue;

        // Check for an empty row.
        if (Row.Column0.Trim().Length > 0)
        {
```

```
            columnName = Row.Column0.Substring(0, Row.Column0.IndexOf(":"));
            // Check for an empty value after the colon.
            if
(Row.Column0.Substring(Row.Column0.IndexOf(":")).TrimEnd().Length > 1)
            // Extract the column value from after the colon and space.
            {
                columnValue = Row.Column0.Substring(Row.Column0.IndexOf(":")
+ 2);
                switch (columnName)
                {
                    case "FirstName":
                        // The FirstName value indicates a new record.
                        this.Output0Buffer.AddRow();
                        this.Output0Buffer.FirstName = columnValue;
                        break;
                    case "LastName":
                        this.Output0Buffer.LastName = columnValue;
                        break;
                    case "Title":
                        this.Output0Buffer.Title = columnValue;
                        break;
                    case "City":
                        this.Output0Buffer.City = columnValue;
                        break;
                    case "StateProvince":
                        this.Output0Buffer.StateProvince = columnValue;
                        break;
                }
            }
        }

    }
```

Example 2: Splitting Parent and Child Records

This example shows how to take a text file, in which a separator row precedes a parent record row that is followed by an indefinite number of child record rows, and parse it into properly normalized parent and child destination tables by using the Script component. This simple example could easily be adapted for source files that use more than one row or column for each parent and child record, as long as there is some way to identify the beginning and end of each record.

Caution
This sample is intended for demonstration purposes only. If you run the sample more than once, it inserts duplicate key values into the destination table.

For more information about how to configure the Script component for use as a transformation in the data flow, see Creating a Synchronous Transformation with the Script Component and Creating an Asynchronous Transformation with the Script Component.

To configure this Script Component example

1. Create and save a text file named **parentchilddata.txt** that contains the following source data:

```
**********
PARENT 1 DATA
child 1 data
child 2 data
child 3 data
child 4 data
**********
PARENT 2 DATA
child 5 data
child 6 data
child 7 data
child 8 data
**********
```

2. Open SQL Server Management Studio and connect to an instance of SQL Server.
3. Select a destination database, and open a new query window. In the query window, execute the following script to create the destination tables:

```
CREATE TABLE [dbo].[Parents](
[ParentID] [int] NOT NULL,
[ParentRecord] [varchar](32) NOT NULL,
```

```
 CONSTRAINT [PK_Parents] PRIMARY KEY CLUSTERED
([ParentID] ASC)
)
GO
CREATE TABLE [dbo].[Children](
[ChildID] [int] NOT NULL,
[ParentID] [int] NOT NULL,
[ChildRecord] [varchar](32) NOT NULL,
 CONSTRAINT [PK_Children] PRIMARY KEY CLUSTERED
([ChildID] ASC)
)
GO
ALTER TABLE [dbo].[Children] ADD CONSTRAINT [FK_Children_Parents]
FOREIGN KEY([ParentID])
REFERENCES [dbo].[Parents] ([ParentID])
```

4. Open SQL Server Data Tools (SSDT) and create a new Integration Services package named SplitParentChild.dtsx.
5. Add a Flat File connection manager to the package, name it ParentChildData, and configure it to connect to the parentchilddata.txt file that you created in a previous step.
6. Add an OLE DB connection manager to the package and configure it to connect to the instance of SQL Server and the database in which you created the destination tables.
7. Add a Data Flow task to the package and click the **Data Flow** tab of SSIS Designer.
8. Add a Flat File Source to the data flow and configure it to use the ParentChildData connection manager. On the **Columns** page of the **Flat File Source Editor**, select the single available external column.
9. Add a Script Component to the data flow and configure it as a transformation. Connect the output of the Flat File Source to the Script Component.
10. Double-click the Script component to display the **Script Transformation Editor**.
11. On the **Input Columns** page of the **Script Transformation Editor**, select the single available input column.
12. On the **Inputs and Outputs** page of the **Script Transformation Editor**, select Output 0, rename it to ParentRecords, and set its **SynchronousInputID** to None. Create 2 output columns:
    - ParentID (the primary key), of type four-byte signed integer [DT_I4]

- ParentRecord, of type string [DT_STR] with a length of 32.

13. Create a second output and name it ChildRecords. The **SynchronousInputID** of the new output is already set to None. Create 3 output columns:
    - ChildID (the primary key), of type four-byte signed integer [DT_I4]
    - ParentID (the foreign key), also of type four-byte signed integer [DT_I4]
    - ChildRecord, of type string [DT_STR] with a length of 50
14. On the **Script** page of the **Script Transformation Editor**, click **Edit Script**. In the **ScriptMain** class, enter the code shown in the example. Close the script development environment and the **Script Transformation Editor**.
15. Add a SQL Server Destination to the data flow. Connect the ParentRecords output of the Script Component to this destination.Configure it to use the OLE DB connection manager and the Parents table.
16. Add another SQL Server Destination to the data flow. Connect the ChildRecords output of the Script Component to this destination. Configure it to use the OLE DB connection manager and the Children table.
17. Run the package. After the package has finished, examine the parent and child records in the two SQL Server destination tables.

```
Public Overrides Sub Input0_ProcessInputRow(ByVal Row As Input0Buffer)

    Static nextRowIsParent As Boolean = False
    Static parentCounter As Integer = 0
    Static childCounter As Integer = 0

    ' If current row starts with separator characters,
    '  then following row contains new parent record.
    If Row.Column0.StartsWith("***") Then
        nextRowIsParent = True
    Else
        If nextRowIsParent Then
            ' Current row contains parent record.
            parentCounter += 1
            Me.ParentRecordsBuffer.AddRow()
            Me.ParentRecordsBuffer.ParentID = parentCounter
            Me.ParentRecordsBuffer.ParentRecord = Row.Column0
            nextRowIsParent = False
        Else
```

```
                ' Current row contains child record.
                childCounter += 1
                Me.ChildRecordsBuffer.AddRow()
                Me.ChildRecordsBuffer.ChildID = childCounter
                Me.ChildRecordsBuffer.ParentID = parentCounter
                Me.ChildRecordsBuffer.ChildRecord = Row.Column0
            End If
        End If

    End Sub
public override void Input0_ProcessInputRow(Input0Buffer Row)
    {

    int static_Input0_ProcessInputRow_childCounter = 0;
    int static_Input0_ProcessInputRow_parentCounter = 0;
    bool static_Input0_ProcessInputRow_nextRowIsParent = false;

        // If current row starts with separator characters,
        // then following row contains new parent record.
        if (Row.Column0.StartsWith("***"))
        {
            static_Input0_ProcessInputRow_nextRowIsParent = true;
        }
        else
        {
            if (static_Input0_ProcessInputRow_nextRowIsParent)
            {
                // Current row contains parent record.
                static_Input0_ProcessInputRow_parentCounter += 1;
                this.ParentRecordsBuffer.AddRow();
                this.ParentRecordsBuffer.ParentID =
static_Input0_ProcessInputRow_parentCounter;
                this.ParentRecordsBuffer.ParentRecord = Row.Column0;
                static_Input0_ProcessInputRow_nextRowIsParent = false;
```

```
            }
            else
            {
                // Current row contains child record.
                static_Input0_ProcessInputRow_childCounter += 1;
                this.ChildRecordsBuffer.AddRow();
                this.ChildRecordsBuffer.ChildID =
static_Input0_ProcessInputRow_childCounter;
                this.ChildRecordsBuffer.ParentID =
static_Input0_ProcessInputRow_parentCounter;
                this.ChildRecordsBuffer.ChildRecord = Row.Column0;
            }
        }

    }
```

Stay Up to Date with Integration Services

For the latest downloads, articles, samples, and videos from Microsoft, as well as selected solutions from the community, visit the Integration Services page on MSDN:

Visit the Integration Services page on MSDN

For automatic notification of these updates, subscribe to the RSS feeds available on the page.

See Also

Creating a Synchronous Transformation with the Script Component

Creating an Asynchronous Transformation with the Script Component