

THE EXPERT'S VOICE® IN SQL SERVER

SECOND EDITION

SQL Server Integration Services Design Patterns

*TOWARD FASTER AND MORE ROBUST
DATA INTEGRATION WITH SQL SERVER
2012 AND 2014*

Andy Leonard, Tim Mitchell, Matt Masson,
Jessica Moss, and Michelle Ufford

Apress®

www.it-ebooks.info

For your convenience Apress has placed some of the front matter material after the index. Please use the Bookmarks and Contents at a Glance links to access them.



Contents at a Glance

First-Edition Foreword.....	xv
About the Authors.....	xvii
About the Technical Reviewer	xix
■ Chapter 1: Metadata Collection	1
■ Chapter 2: Execution Patterns	27
■ Chapter 3: Scripting Patterns	71
■ Chapter 4: SQL Server Source Patterns	87
■ Chapter 5: Data Correction with Data Quality Services	101
■ Chapter 6: DB2 Source Patterns	125
■ Chapter 7: Flat File Source Patterns	135
■ Chapter 8: Loading a PDW Region in APS.....	171
■ Chapter 9: XML Patterns.....	193
■ Chapter 10: Expression Language Patterns.....	213
■ Chapter 11: Data Warehouse Patterns.....	227
■ Chapter 12: OData Source	251
■ Chapter 13: Slowly Changing Dimensions.....	261
■ Chapter 14: Loading the Cloud	275
■ Chapter 15: Logging and Reporting Patterns	281
■ Chapter 16: Parent-Child Patterns.....	293

■ Chapter 17: Configuration	305
■ Chapter 18: Deployment	331
■ Chapter 19: Business Intelligence Markup Language	343
■ Chapter 20: Biml and SSIS Frameworks.....	369
■ Appendix A: Evolution of an SSIS Framework	377
Index.....	435



Metadata Collection

The first Integration Services design pattern we will cover is metadata collection. What do we mean by *metadata collection*? Good question. This chapter could also be called “Using SSIS to Save Time and Become an Awesome DBA.” Many DBAs spend a large portion of time monitoring activities such as verifying backups, alerting on scheduled job failures, creating schema snapshots (“just in case”), examining space utilization, and logging database growth over time, to name just a very few. Most Relational Database Management Systems (RDBMS’s) provide metadata to help DBAs monitor their systems. If you’ve been a DBA for a few years, you may even have a “tool bag” of scripts that you use to interrogate metadata. Running these scripts manually is easy when you have just one or two servers; however, this can quickly become unwieldy and consume a large portion of your time as your enterprise grows and as the number of database servers increases.

This chapter examines how to use Integration Services and the metadata that exists within SQL Server to automate some of these routine tasks.

About SQL Server Data Tools

SQL Server Data Tools - Business Intelligence (SSDT-BI) is Microsoft’s IDE for developing Integration Services packages. It leverages the maturity and familiarity of Visual Studio to provide a unified development platform for SQL Server Business Intelligence projects, including Integration Services, Analysis Services, and Reporting Services projects. This book is written using SSDT-BI for Visual Studio 2013 and SSIS 2014.

Tip Don’t have SSDT-BI installed yet? SSDT-BI is available from Microsoft’s Download Center. Please note that SSDT-BI is not backward compatible, so make sure you verify that the version you download is appropriate for your environment.

A Peek at the Final Product

Let’s discuss the Integration Services package you will be creating in this chapter.

In SQL Server, you will do the following:

1. Create a database to act as your central repository for database monitoring.
2. Create a table to store a list of SQL Server instances that you wish to monitor.
3. Create a table for each of the data elements you wish to monitor (unused indexes and database growth).

In Integration Services, you will do the following:

1. Create a new Integration Services package.
2. Retrieve a list of SQL Server instances and store the list in a variable.
3. Create an OLE DB connection with a dynamically populated server name.
4. Iterate through each database and
 - a. Retrieve current database and log file sizes for historical monitoring.
 - b. Retrieve a list of index candidates for potential redesign or dropping.
 - c. Update the Last Monitored value for each SQL Server instance.

This is a very flexible model that you can easily expand to include many more monitoring tasks. A screenshot of the completed package is displayed in Figure 1-1.

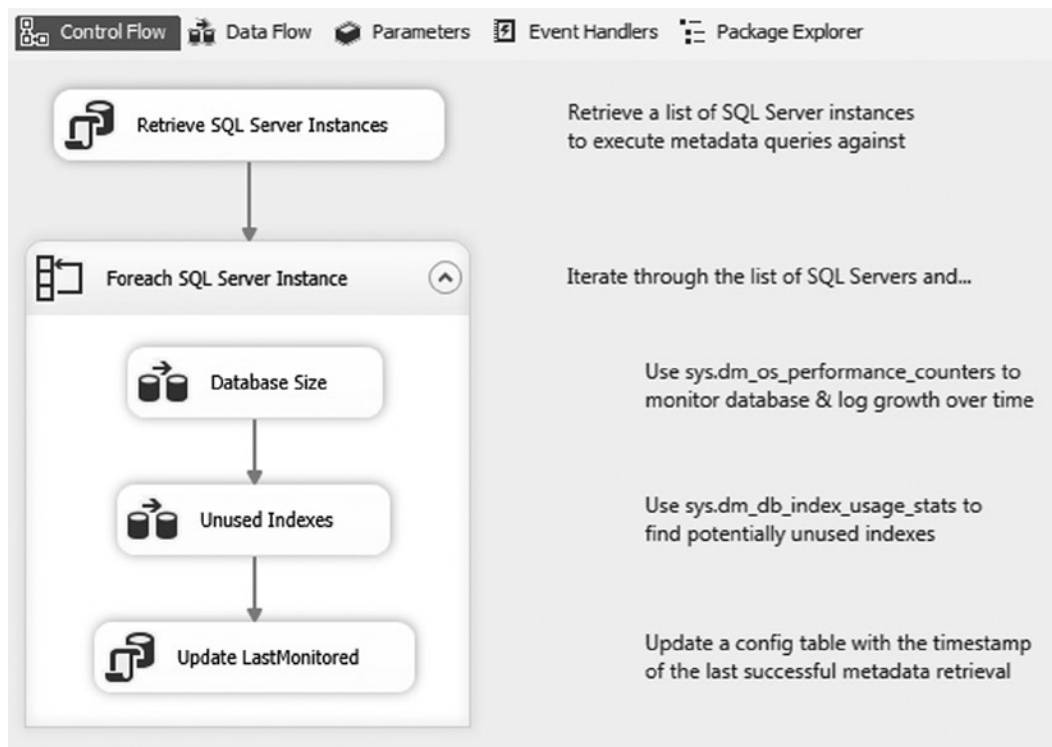


Figure 1-1. The *MetadataCollection* package

If this is not your first Integration Services package, maybe you've noticed that this package is missing a few best practices, such as error handling. In the interest of clarity, the package you create in this chapter will focus only on core design patterns; however, we will call out best practices when applicable.

Also, please note that the T-SQL examples will only work with SQL Server 2005 or later.

SQL Server Metadatalog

Although metadata can be collected from any RDBMS that provides an interface for accessing it, this chapter uses SQL Server as its metadata source. The focus of this chapter is not on the actual metadata, but rather the pattern of metadata collection. Still, it is useful for you to have a basic understanding of the type of metadata that is available.

SQL Server exposes a wealth of information through catalog views, system functions, dynamic management views (DMVs), and dynamic management functions (DMFs). Let's briefly examine some of the metadata you will be using in this chapter.

■ **Tip** SQL Server Books Online is a great resource for learning more about the types of metadata available in SQL Server. Try searching for “metadata functions,” “catalog views,” and “DMVs” for more information.

sys.dm_os_performance_counters

The `sys.dm_os_performance_counters` DMV returns server performance counters on areas including memory, wait stats, and transactions. This DMV is useful for reporting file sizes, page life expectancy, page reads and writes per second, and transactions per second, to name but a few.

sys.dm_db_index_usage_stats

The `sys.dm_db_index_usage_stats` DMV contains information on index utilization. Specifically, a counter is incremented every time a seek, scan, lookup, or update is performed on an index. These counters are reinitialized whenever the SQL Server service is started. If you do not see a row in this DMV for a particular index, it means that a seek, scan, lookup, or update has not yet been performed on that index since the last server reboot.

sys.dm_os_sys_info

The `sys.dm_os_sys_info` DMV contains information about server resources. Perhaps the most frequently used piece of information in this DMV is the `sqlserver_start_time` column, which tells you the last time the SQL Server service was started.

sys.tables

The `sys.tables` catalog view contains information about every table that exists within the database.

sys.indexes

The `sys.indexes` catalog view contains information about every index in the database. This includes information such as whether an index is clustered or nonclustered and whether the index is unique or nonunique.

sys.partitions

The `sys.partitions` catalog view gives visibility into the partitioning structure of an index. When an index has more than one partition, the data in the index is split into multiple physical structures that can be accessed using the single logical name. This technique is especially useful for dealing with large tables, such as a transaction history table. If a table is not partitioned, the table will still have a single row in `sys.partitions`.

sys.allocation_units

The `sys.allocation_units` catalog view contains information about the number of pages and rows that exist for an object. This information can be joined to the `sys.partitions` catalog view by joining the `container_id` to the `partition_id`.

Setting Up the Central Repository

Before you can begin development on your Integration Services package, you need to set up some prerequisites in SQL Server. First and foremost, you need to create a database that will act as your central data repository. This is where your list of SQL Server instances will reside and where you will store the metadata you retrieve for each SQL Server instance. Many enterprises also find it convenient to store all error and package logging to this same central database. This is especially beneficial in environments where there are numerous DBAs, developers, and servers, because it makes it easy for everyone to know where to look for information. The T-SQL code in Listing 1-1 creates the database you will use throughout the rest of this chapter.

Listing 1-1. Example of T-SQL Code to Create a SQL Server Database

```
USE [master];
GO

CREATE DATABASE [dbaCentralLogging]
ON PRIMARY
(
    NAME = N'dbaCentralLogging'
    , FILENAME = N'C:\Program Files\Microsoft SQL Server\MSSQL12.MSSQLSERVER\MSSQL\DATA\
      dbaCentralLogging.mdf'
    , SIZE = 1024MB
    , MAXSIZE = UNLIMITED
    , FILEGROWTH = 1024MB
)
LOG ON
(
    NAME = N'dbaCentralLogging_log'
    , FILENAME = N'C:\Program Files\Microsoft SQL Server\MSSQL12.MSSQLSERVER\MSSQL\DATA\
      dbaCentralLogging_log.ldf'
    , SIZE = 256MB
    , MAXSIZE = UNLIMITED
    , FILEGROWTH = 256MB
);
GO
```

Please note that your file directory may differ from the one in the preceding example.

This code can be executed from SQL Server Management Studio (SSMS), as demonstrated in Figure 1-2, or from your favorite query tool.

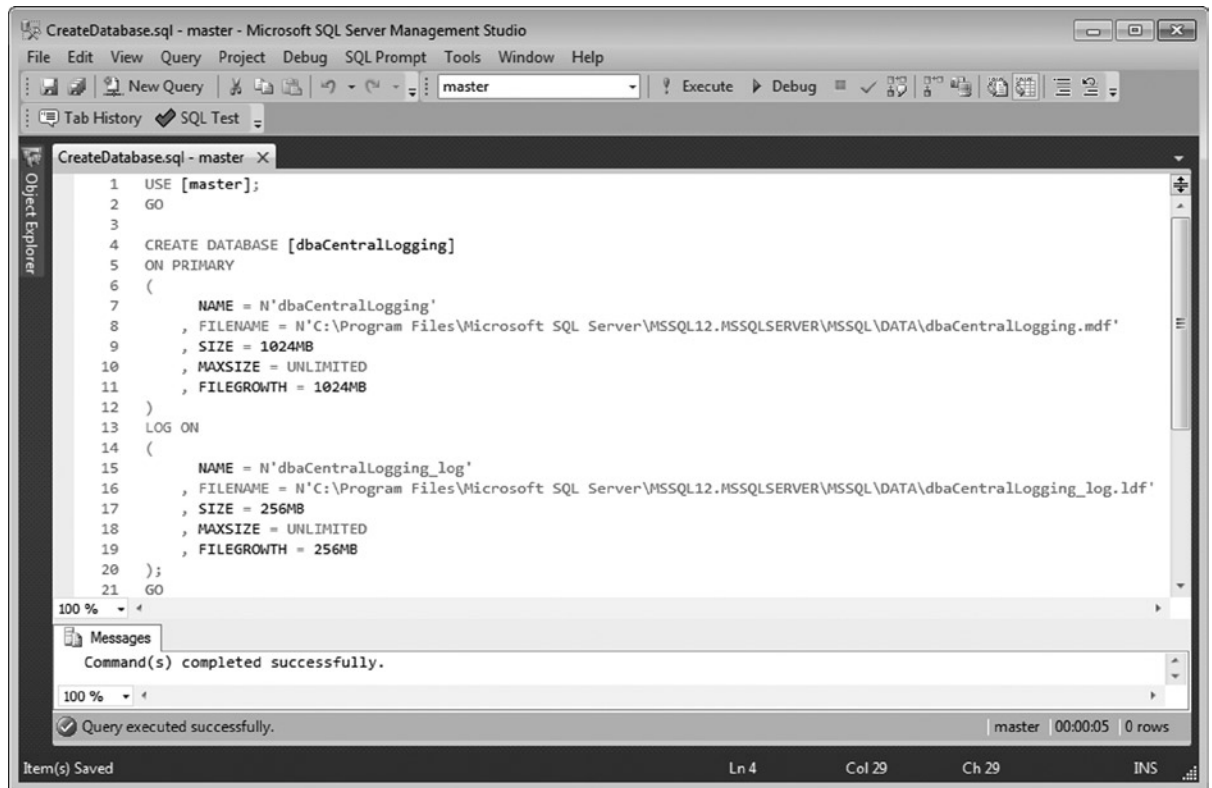


Figure 1-2. SQL Server Management Studio 2012

Next, you need a list of SQL Server instances to monitor. The easiest way to accomplish this is to store a list of database instance names in a file or table. You will use the latter method. Using the code in Listing 1-2, create that table now inside your newly created database.

Listing 1-2. Example of T-SQL Code to Create a Table for Monitoring SQL Server Instances

```

USE dbaCentralLogging;
GO

CREATE TABLE dbo.dba_monitor_SQLServerInstances
(
    SQLServerInstance NVARCHAR(128)
    , LastMonitored    SMALLDATETIME    NULL

    CONSTRAINT PK_dba_monitor_SQLServerInstances
    PRIMARY KEY CLUSTERED( SQLServerInstance )
);

```

Now you're ready to populate the table with a list of SQL Server instances to monitor. The code in Listing 1-3 will walk you through how to do this, although you will need to update the placeholders with SQL Server instances that exist in your environment.

Listing 1-3. Example of T-SQL Code to Insert Data into the `dba_monitor_SQLServerInstances` Table

```
INSERT INTO dbo.dba_monitor_SQLServerInstances
(
    SQLServerInstance
)
SELECT @@SERVERNAME-- The name of the server that hosts the central repository
UNION ALL
SELECT 'YourSQLServerInstanceHere'-- Example of a SQL Server instance
UNION ALL
SELECT 'YourSQLServerInstance\Here';-- Example of a server with multiple instances
```

You still need to create two tables to store the metadata you collect, but you will create these as you get to the relevant section in this chapter. Next, you will create your Integration Services package.

The Iterative Framework

In this section, you lay the foundation for your iterative framework. Specifically, you will demonstrate a repeatable pattern for populating a variable with a list of SQL Server instances and then iterating through the list and performing an action on each server.

First, open Visual Studio. Create a new project by navigating to **File ► New ► Project**. Expand the **Business Intelligence** section (under **Installed ► Templates**), and then click **Integration Services Project**. Name the project **MetadataCollection**, as illustrated in Figure 1-3.

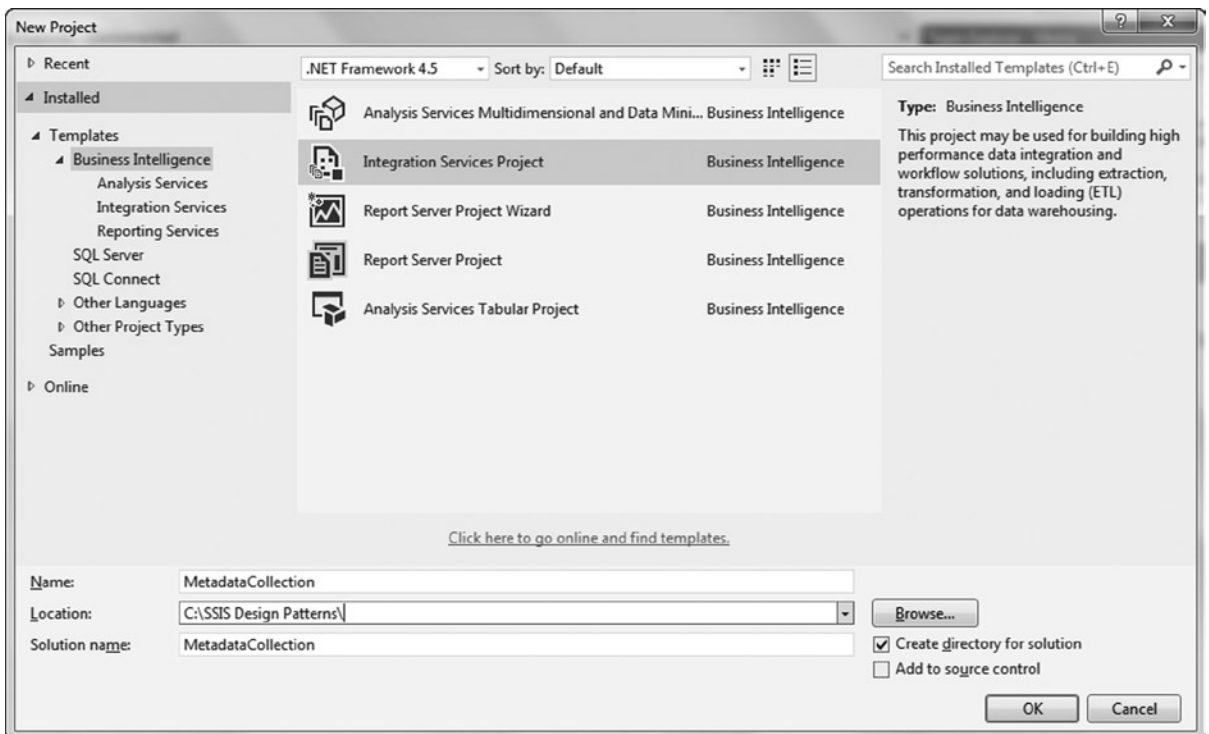


Figure 1-3. New integration services project

Please note that your default Location will be different from the directory pictured in Figure 1-3.

You now need to create two variables. The first variable will be used to store the list of SQL Server instances you retrieve. The second variable will store the value of a single instance as you iterate through your list.

To access the variable menu, select Variables under the SSIS menu (Figure 1-4); you can also access the Variables menu by right-clicking the designer surface.

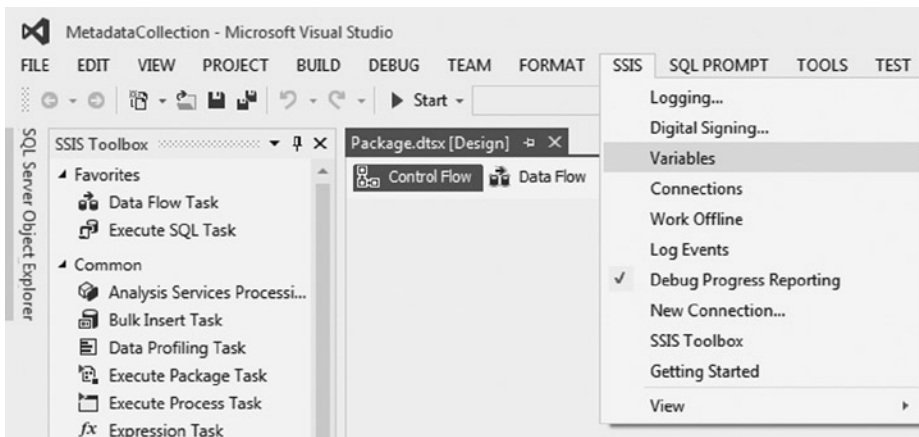


Figure 1-4. Opening the Variables menu

Add the following variables by clicking the Add Variable icon on the far left of the Variables menu, as illustrated in Figure 1-5:

- SQLServerInstance—String data type
- SQLServerList—Object data type

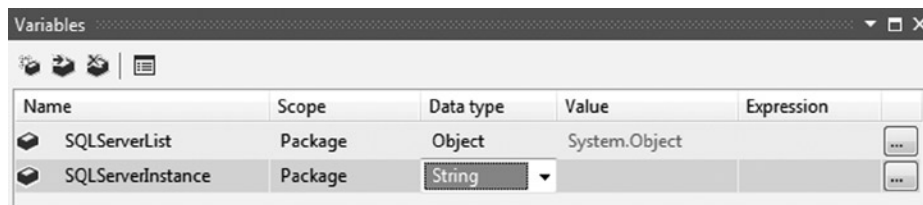


Figure 1-5. Package-scoped variables

Initialize the SQLServerInstance variable with a server that you have permissions to connect to. For simplicity, we suggest using the server where the dbaCentralLogging database was created. This value will be overwritten at runtime.

Now that you have a place to store your list of instances, you're ready to populate the variable. Drag a new Execute SQL task from the SSIS Toolbox onto the designer surface. Rename the task **Retrieve SQL Server Instances** and double-click it to open the Execute SQL Task Editor. Click the drop-down under Connection, and then select <New connection...>, as seen in Figure 1-6.

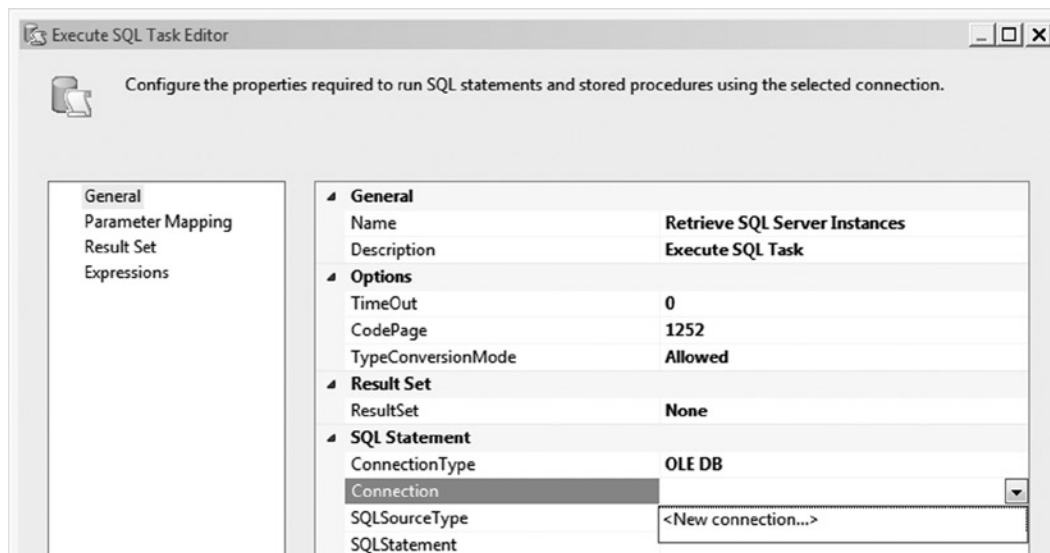


Figure 1-6. The Execute SQL Task Editor

In the Configure OLE DB Connection Manager menu, click New. In the Server Name field, enter the database server where you created the database in Listing 1-1. Regardless of whether you are using Windows or SQL Server authentication, make sure that the account has sufficient permissions to each of the instances in your dba_monitor_SQLServerInstances table. Under Select or Enter a Database Name, select dbaCentralLogging from the drop-down menu, as illustrated in Figure 1-7. Click OK to return to the Execute SQL Task Editor.

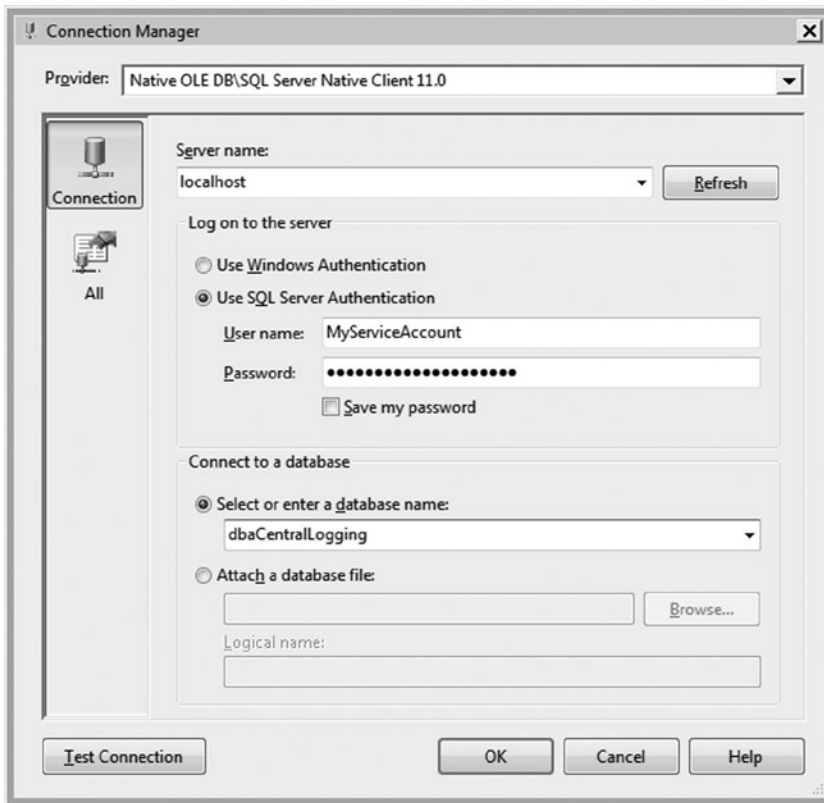


Figure 1-7. *The Connection Manager*

Note Permissions requirements vary depending on the type of metadata you wish to retrieve. For more information on the permissions necessary to access a specific object, please refer to the object type's page within SQL Server Books Online.

You now need to write the SQL statement that will retrieve the list of SQL Server instances. Click the [...] icon to the right of the SQLStatement field, and then enter the T-SQL code from Listing 1-4.

Listing 1-4. T-SQL Statement to Retrieve SQL Server Instances

```
SELECT SQLServerInstance FROM dbo.dba_monitor_SQLServerInstances;
```

Because you are retrieving an array of values, select Full Result Set from the ResultSet drop-down. Your Execute SQL Task Editor should now resemble Figure 1-8; however, your Connection values will likely be different.

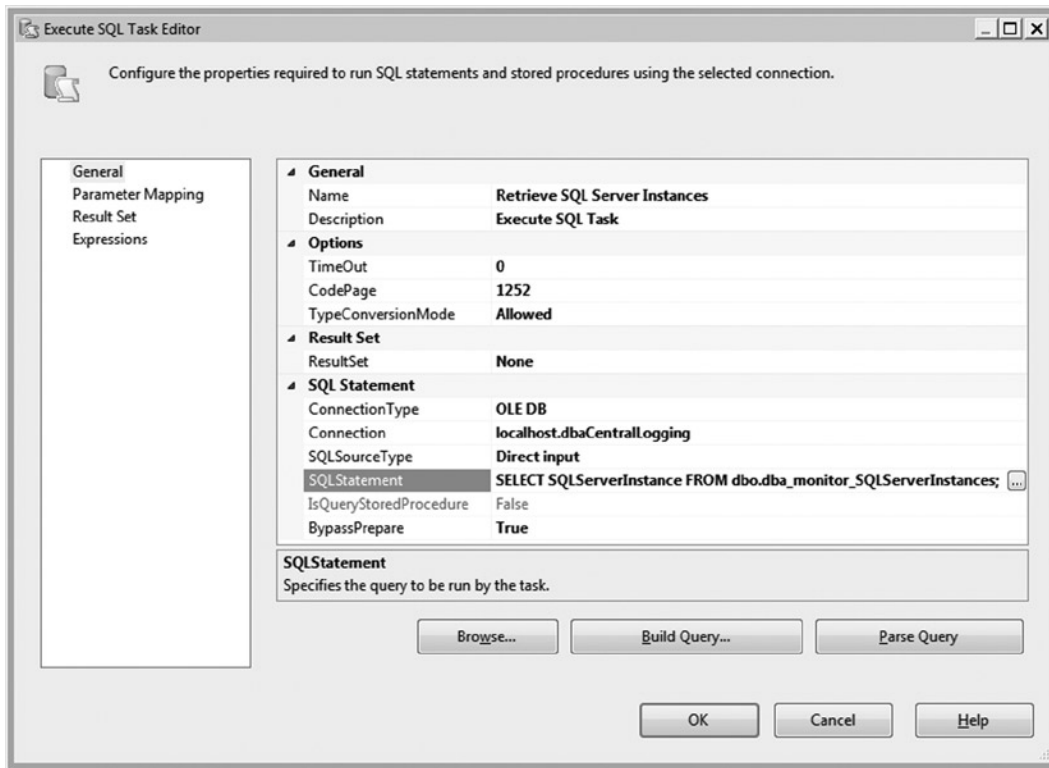


Figure 1-8. The SQL Task Editor

You're almost done configuring the Connection Manager. All you have left is to map your result set to your variable. Select Result Set on the left side of the Execute SQL Task Editor, and then click Add. Because you are using a full result set, you must replace the Result Name with 0. You now need to tell Integration Services which variable to use. Select User::SQLServerList from the drop-down under Variable Name, as illustrated in Figure 1-9. Click OK.

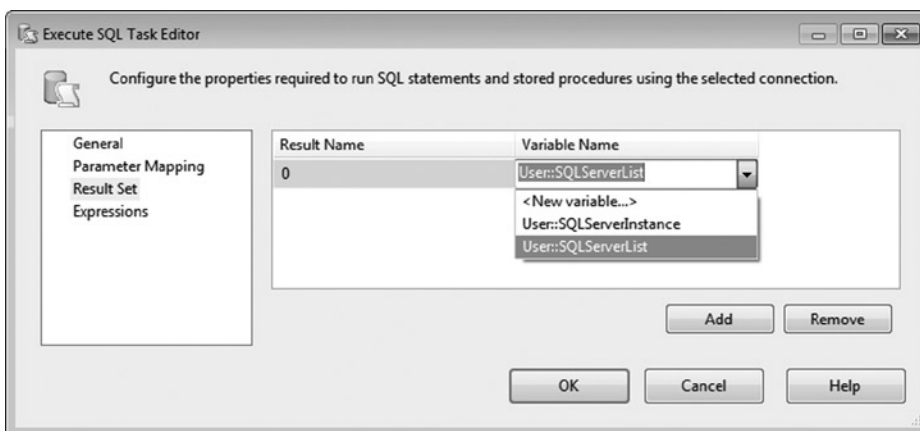


Figure 1-9. Editing the result set

The Execute SQL task is now complete. Next, you need to iterate through each server to retrieve the metadata you plan to monitor. This process will be encapsulated within a Foreach Loop container, which will shred the list of SQL Server instances stored in the `SQLServerList` variable.

Add a Foreach Loop container to the Control Flow and rename it **Foreach SQL Server Instance**. Connect it to the Execute SQL task with a Success Precedence constraint—in other words, drag the green arrow from the Execute SQL task to the Foreach Loop container, as seen in Figure 1-10.

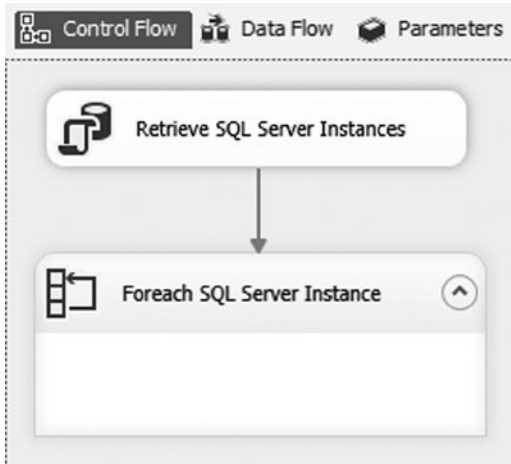


Figure 1-10. Connecting the Execute SQL task to the Foreach Loop container

Double-click the Foreach Loop container to edit its properties. Click the Collection page, and then select Foreach ADO Enumerator in the Enumerator field. Under ADO Object Source Variable, select `User::SQLServerList`; leave Enumeration Mode set to Rows in the First Table. Your Collection properties should match those in Figure 1-11.

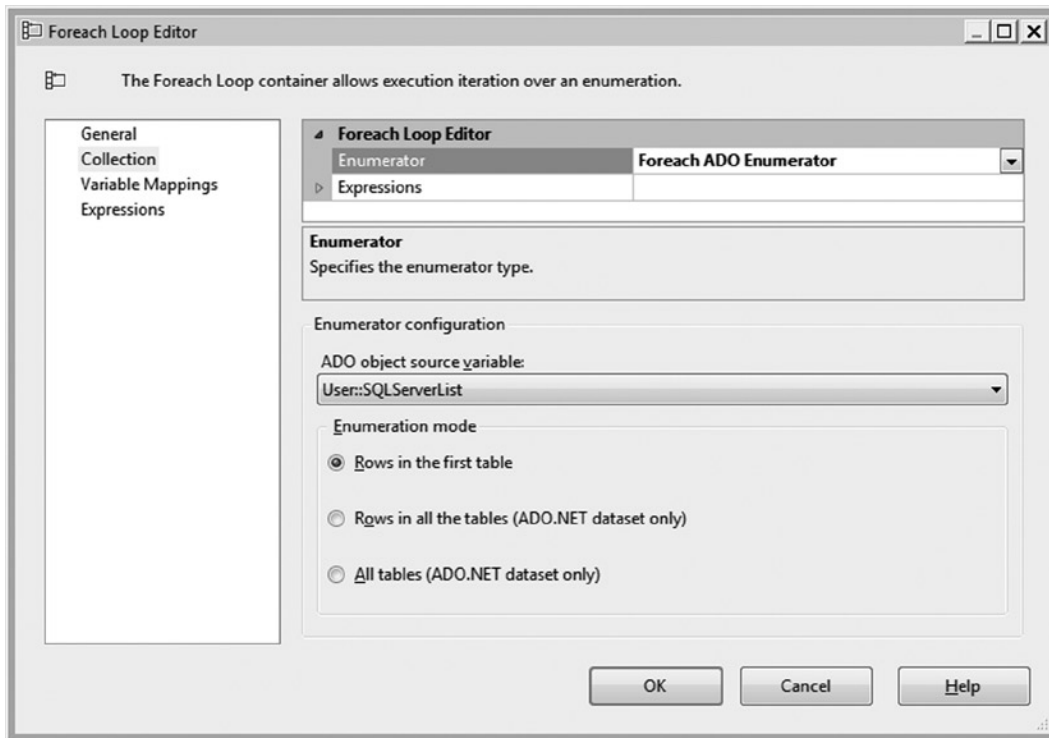


Figure 1-11. The Foreach Loop Editor

On the Variable Mappings page, map the SQLServerInstance variable to Index 0, as demonstrated in Figure 1-12.

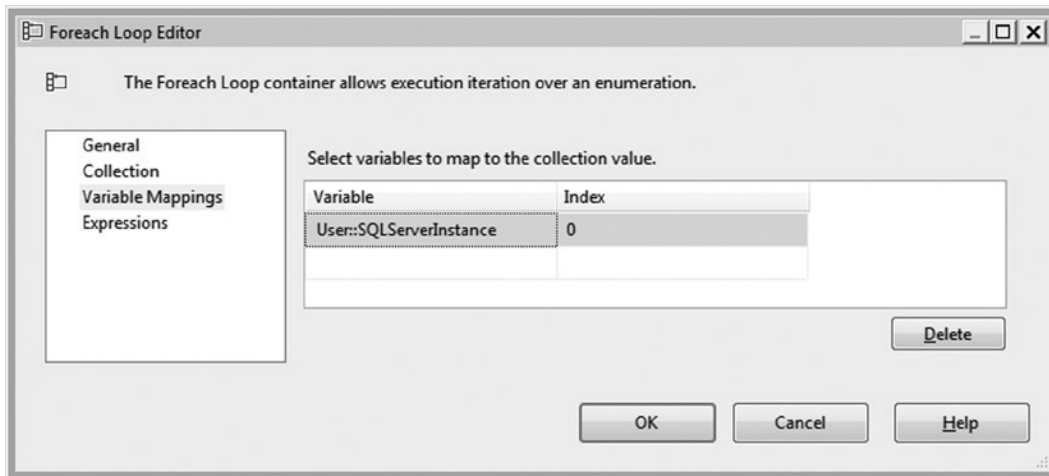


Figure 1-12. Variable mappings

Click the OK button to close the Foreach Loop Editor.

Let's review what you've done so far. You now have a variable, `SQLServerList`, which contains a list of all the SQL Server instances you inserted into the `dba_monitor_SQLServerInstances` table. The Foreach Loop container then shreds this variable, walking through each value—each SQL Server instance, in this case—one at a time. At each pass, it pushes the value of one of those SQL Server instances into another variable, `SQLServerInstance`.

Now it's time to set up the dynamic connection that will be used to connect to each of the SQL Server instances you're monitoring. To do this, you need to create a dummy connection and configure it to use the server name stored in `SQLServerInstance`.

Right-click in the Connection Managers window and select **New OLE DB Connection**. Create a new connection using the same server and security credentials you used previously (Figure 1-7), but select **Master** as the database this time. To be clear, you're using the same server purely for convenience. In reality, the server specified in the dummy connection is irrelevant as long as you have sufficient permissions to log onto it, because whatever value you enter will be overwritten by the `SQLServerInstance` variable at runtime. The database value does matter, however, because the database you select *must* exist on every server. Since **Master** is a system database, it is a natural choice.

Click OK to close the Connection Manager Properties window. But you're not done with this connection just yet. Right-click the newly created connection and select **Properties**. Change the Name property to **DynamicSQLServerInstance**, then click the [...] icon in the Expressions field. This will bring up the Property Expressions Editor. Select the Property value you wish to dynamically populate—**ServerName**, in this case—and enter **@[User::SQLServerInstance]** in the Expression field, as demonstrated in Figure 1-13. Optionally, you can also click the [...] icon in the Expression field to open the Expression Builder, which is helpful if you are not very familiar with Expression syntax.

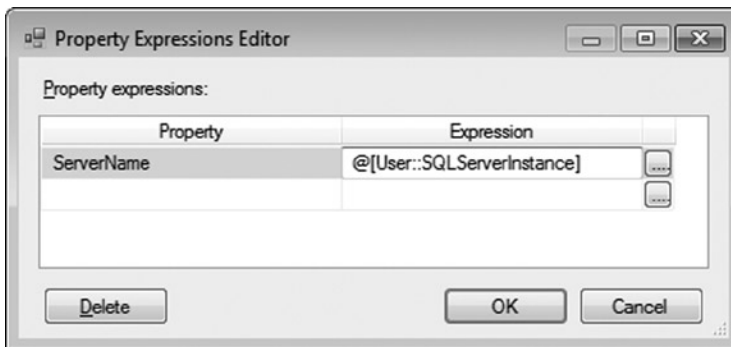


Figure 1-13. *Property Expressions Editor*

The properties of your connection should now resemble those shown in Figure 1-14.

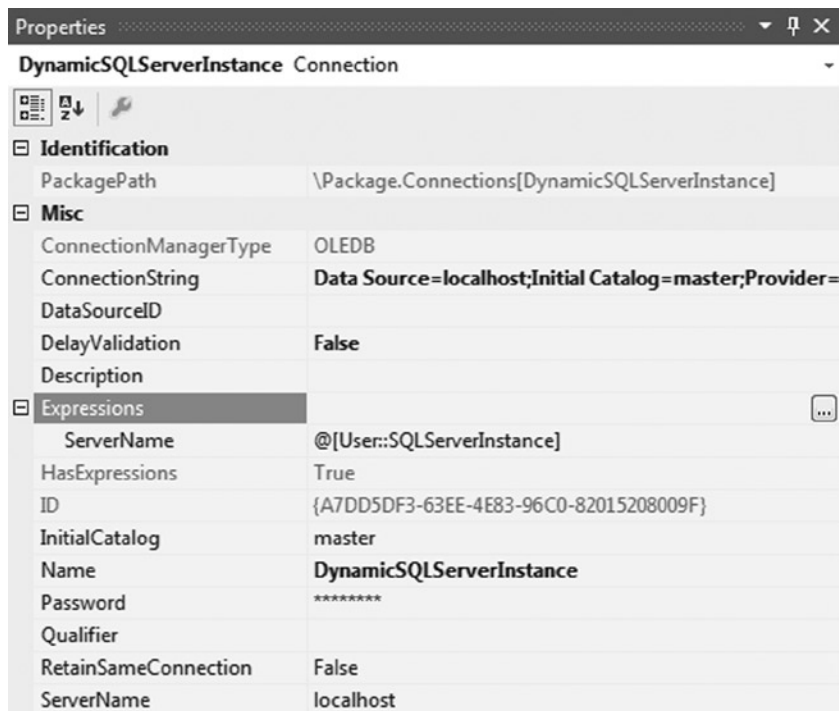


Figure 1-14. Dynamic connection properties

At this point, you now have a reusable framework for iterating through a list of SQL Server instances and doing *something* on each server. This in and of itself is a very valuable design pattern. However, because this is a chapter on metadata collection, we would be remiss if we did not actually demonstrate collecting and storing metadata. The next section will walk you through setting up two useful metadata extracts.

Metadata Collection

You're now ready to retrieve metadata from your list of servers. But what should you collect? An incredible wealth of information is available for retrieval, including security information, usage data, table schema snapshots, failed job details, fragmentation levels, and performance counters, to name just a few. For this first example, let's keep it simple and retrieve current database and log file size. This information is useful for historical database growth and capacity planning.

To accomplish this, you will create data flows within your Foreach Loop container to retrieve the metadata from each server and store it in the dbaCentralLogging database. The Data Flow task is arguably the most frequently used task in Integration Services. It allows you to easily move data between servers and, if necessary, perform data conversions or cleansing.

Drag a Data Flow task from the SSIS Toolbox into the Foreach Loop container and rename it **Database Size**. Double-clicking the Data Flow task will open the Data Flow Designer tab. Notice that the objects available within the toolbox change once you are inside the Data Flow designer. Drag the OLE DB Source icon onto the designer surface and rename it **Dynamic SQL Source**. Double-click the icon to edit its properties.

Select DynamicSQLServerInstance in the OLE DB Connection Manager drop-down. Change the Data Access Mode to SQL Command, and then copy the code from Listing 1-5 into the SQL Command text box.

Listing 1-5. Example of T-SQL to Retrieve Current Data and Log File Sizes for All Databases on the Server

```

SELECT GETDATE() AS [captureDate]
      , @@SERVERNAME AS [serverName]
      , instance_name AS [databaseName]
      , SUM(
          CASE
            WHEN counter_name = 'Data File(s) Size (KB)'
            THEN cntr_value
          END
        ) AS 'dataSizeInKB'
      , SUM(
          CASE
            WHEN counter_name = 'Log File(s) Size (KB)'
            THEN cntr_value
          END
        ) AS 'logSizeInKB'
FROM sys.dm_os_performance_counters
WHERE counter_name IN ('Data File(s) Size (KB)'
      , 'Log File(s) Size (KB)')
/* optional: remove _Total to avoid accidentally
double-counting in queries */
      AND instance_name <> '_Total'
GROUP BY instance_name;

```

This query will produce results similar to the following.

captureDate	serverName	databaseName	dataSizeInKB	logSizeInKB
2014-06-29 19:52:21.543	LOCALHOST	_Total	1320896	274288
2014-06-29 19:52:21.543	LOCALHOST	AdventureWorks2012	193536	496
2014-06-29 19:52:21.543	LOCALHOST	dbaCentralLogging	1048576	262136
2014-06-29 19:52:21.543	LOCALHOST	master	4096	760
2014-06-29 19:52:21.543	LOCALHOST	model	2112	760
2014-06-29 19:52:21.543	LOCALHOST	msdb	14080	760
2014-06-29 19:52:21.543	LOCALHOST	mssqlsystemresource	40960	504
2014-06-29 19:52:21.543	LOCALHOST	ReportServer\$SQL2012	5184	7032
2014-06-29 19:52:21.543	LOCALHOST	ReportServer\$SQL2012TempDB	4160	1080
2014-06-29 19:52:21.543	LOCALHOST	tempdb	8192	760

(10 row(s) affected)

Your OLE DB Source Editor should now resemble the editor in Figure 1-15. Click Parse Query to ensure the SQL syntax is correct, and then click Preview at the bottom of the editor to see a sample of the results. Click OK to exit the OLE DB Source Editor.

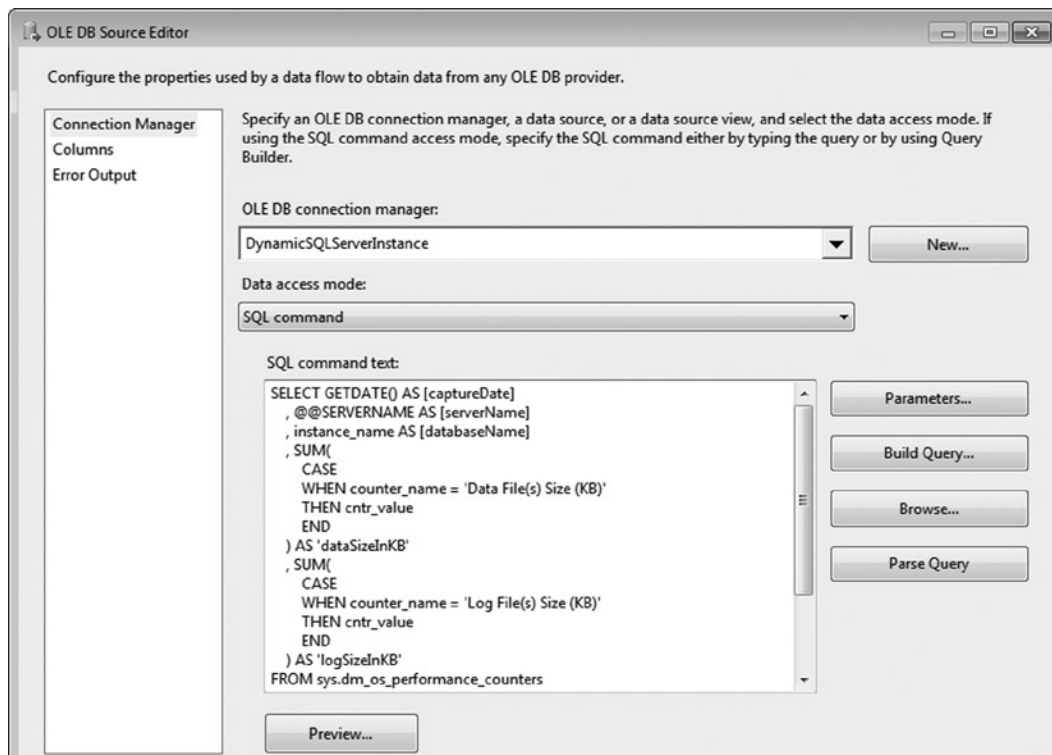


Figure 1-15. OLE DB Source Editor

Let's take a moment to discuss this code. You are using the `sys.dm_os_performance_counters` DMV to retrieve data file and log file sizes. This DMV stores data and log sizes in a separate row for each database, so you are pivoting the data to return one row for each database, with the file size and log size in separate columns. As a reminder, DMVs were introduced in SQL Server 2005, so this example will only work in SQL Server 2005 and newer editions.

It is generally a best practice to create stored procedures for these types of administrative queries and to deploy them to each server, typically into a database like `dbaToolBox`. This introduces some maintenance overhead, but benefits of stored procedures—such as improved security and visibility into dependencies, usage, performance tuning, and troubleshooting—typically outweigh the overhead. Also, it allows a DBA or developer to manually execute these same queries on each server without having to search for the code within an Integration Services package. However, in the interests of simplicity, you will just input the code directly into your Data Flow task.

■ **Tip** The `sys.dm_os_performance_counters` DMV is very useful for database monitoring and contains much more information than just data and log file sizes. You can easily modify the preceding code to include additional performance counters. However, you should be aware that there are three types of `cntr_type` values (value/base, per second, and point-in-time), and the preceding code only works for the point-in-time counter type (`cntr_type = 65792`). Refer to SQL Server Books Online for more information on the types of information available in this DMV and how to work with each counter type.

Now that you understand the expected output of the query, you need a table to store the results. From within SSMS, execute the T-SQL statement in Listing 1-6 within the dbaCentralLogging database.

Listing 1-6. Example of T-SQL Code to Create a Table to Store Data and Log File Size Information

```
USE dbaCentralLogging;
GO

CREATE TABLE dbo.dba_monitor_databaseGrowth
(
    log_id          INT IDENTITY(1,1)
    , captureDate   DATETIME
    , serverName     NVARCHAR(128)
    , databaseName  SYSNAME
    , dataSizeInKB  BIGINT
    , logSizeInKB   BIGINT

    CONSTRAINT PK_dba_monitor_databaseGrowth
        PRIMARY KEY NONCLUSTERED(log_id)
);

CREATE CLUSTERED INDEX CIX_dba_monitor_databaseGrowth
    ON dbo.dba_monitor_databaseGrowth(captureDate,serverName,databaseName);
```

You can now return to your Integration Services package. You do not need to perform any data cleansing or data transformations in this Data Flow task, so you'll proceed straight to storing your results. Select the OLE DB Destination item from the toolbox, drag it onto the designer surface, and rename it **Central Logging Destination**. Connect it to the OLE DB Source by dragging the blue data flow arrow from the source to the destination. Double-clicking the OLE DB destination brings up another editor. This time, select your dbaCentralLogging connection from the OLE DB Connection Manager drop-down. Leave Table or View - Fast Load selected in the Data Access Mode drop-down. In the Name of the Table or the View drop-down, select [dbo].[dba_monitor_databaseGrowth], as seen in Figure 1-16.

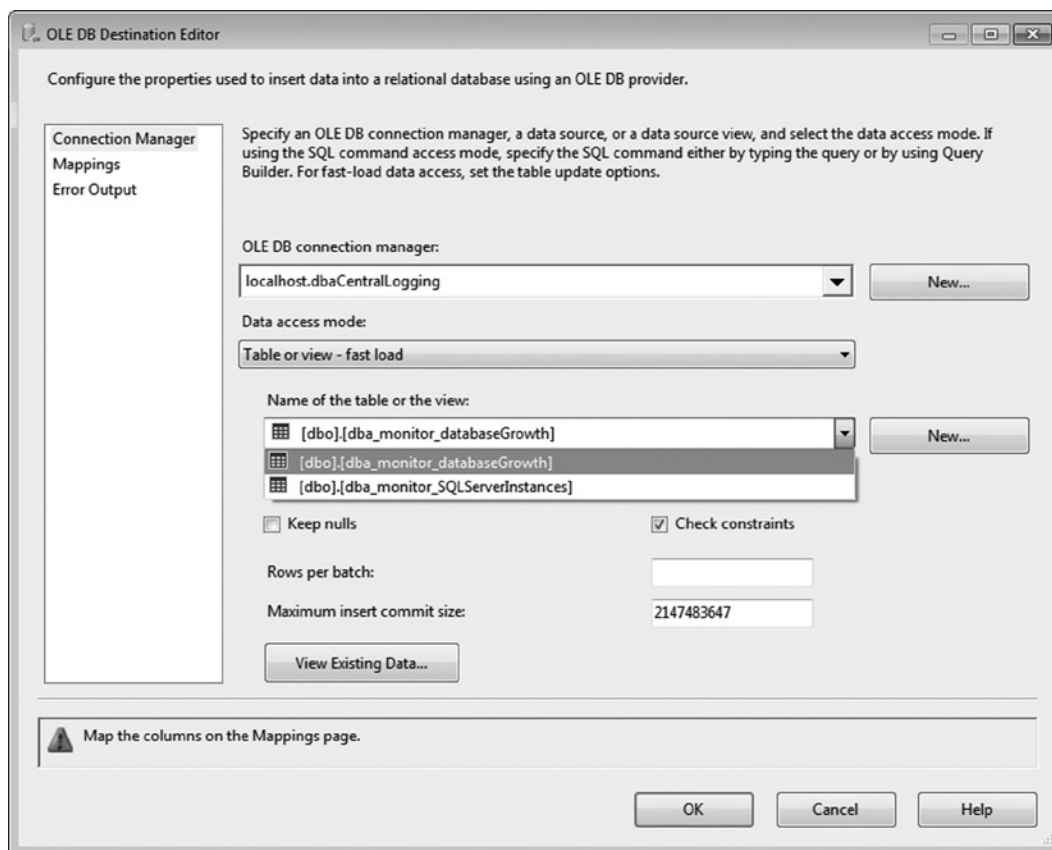


Figure 1-16. Editing the OLE DB Destination Editor's connection manager

When you're done with the Connection Manager page, click the Mappings menu. You'll notice that Integration Services has taken the liberty of performing an initial mapping based on column names. Although this is a nice time-saving feature, be wary in environments where the same column name is used for multiple data elements. Because the `log_id` column is an identity value that is populated during data insertion, you will ignore it in your mappings. Confirm that your mappings resemble those shown in Figure 1-17, and then click OK to return to the Data Flow designer.

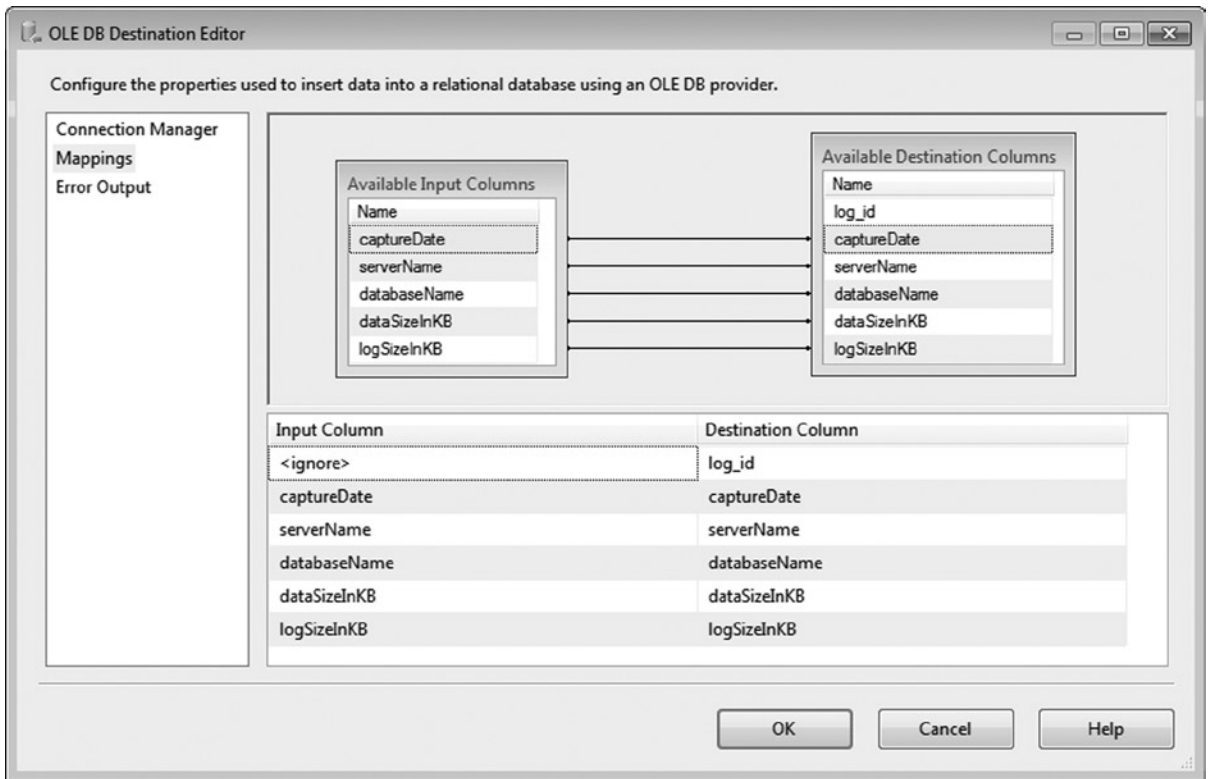


Figure 1-17. Editing the OLE DB destination mappings

Your first data flow is complete, as seen in Figure 1-18.

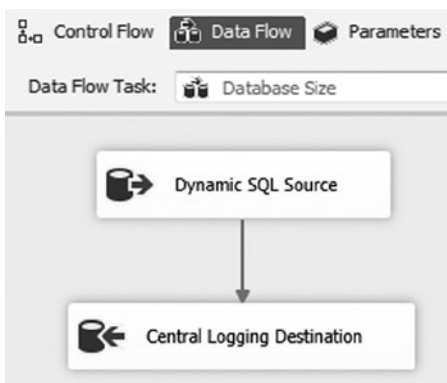


Figure 1-18. The completed Data Flow task

You are now ready to create your second data flow. From the Control Flow tab, copy and paste the existing data flow into the Foreach Loop container. Drag the green arrow—the Success Precedence constraint—from the Database Size data flow to your new data flow. Rename the new data flow **Unused Indexes**, and then double-click it to return to the Data Flow designer.

Double-click the Dynamic SQL Source icon to edit its properties. You need to change the SQL command to use the code in Listing 1-7.

Listing 1-7. Example of T-SQL Query to Retrieve Unused Indexes

```
/* Create a variable to hold a list of indexes */
DECLARE @Indexes TABLE
(
    serverName      NVARCHAR(128)
  ,schemaName      SYSNAME
  ,schemaID        INT
  ,databaseName    SYSNAME
  ,databaseID      INT
  ,tableName       SYSNAME
  ,objectID        INT
  ,indexName       SYSNAME
  ,indexID         INT
  ,indexType       NVARCHAR(60)
  ,isPrimaryKey    BIT
  ,isUnique        BIT
  ,isFiltered      BIT
  ,isPartitioned   BIT
  ,numberOfRows    BIGINT
  ,totalPages      BIGINT
);

/* Iterate through all databases */
INSERT INTO @Indexes (serverName,schemaName,schemaID,databaseName,databaseID,tableName,objectID,
indexName,indexID,indexType,isUnique,isPrimaryKey,isFiltered,isPartitioned,numberOfRows,totalPages)
EXECUTE sys.sp_MSforeachdb
' USE ?;
SELECT @@SERVERNAME
  , SCHEMA_NAME(t.schema_id)
  , t.schema_id
  , DB_NAME()
  , DB_ID()
  , t.name
  , t.object_id
  , i.name
  , i.index_id
  , i.type_desc
  , i.is_primary_key
  , i.is_unique
  , i.has_filter
  , CASE WHEN COUNT(p.partition_id) > 1 THEN 1 ELSE 0 END
  , SUM(p.rows)
  , SUM(au.total_pages)
```

```

FROM sys.tables                AS t WITH (NOLOCK)
JOIN sys.indexes              AS i WITH (NOLOCK)
  ON i.object_id = t.object_id
JOIN sys.partitions          AS p WITH (NOLOCK)
  ON p.object_id = i.object_id
  AND p.index_id = i.index_id
JOIN sys.allocation_units    AS au WITH (NOLOCK)
  ON au.container_id = p.partition_id
WHERE i.index_id <> 0 /* exclude heaps */
GROUP BY SCHEMA_NAME(t.schema_id)
  , t.schema_id
  , t.name
  , t.object_id
  , i.name
  , i.index_id
  , i.type_desc
  , i.has_filter
  , i.is_unique
  , i.is_primary_key;

/* Retrieve index stats for return to our central repository */
SELECT GETDATE() AS [captureDate]
  , i.serverName
  , i.schemaName
  , i.databaseName
  , i.tableName
  , i.indexName
  , i.indexType
  , i.isFiltered
  , i.isPartitioned
  , i.numberOfRows
  , ddius.user_seeks AS [userSeeksSinceReboot]
  , ddius.user_scans AS [userScansSinceReboot]
  , ddius.user_lookups AS [userLookupsSinceReboot]
  , ddius.user_updates AS [userUpdatesSinceReboot]
  , (i.totalPages * 8) / 1024 AS [indexSizeInMB] /* pages are 8KB */
  , dosi.sqlserver_start_time AS [lastReboot]
FROM @Indexes AS i
JOIN sys.dm_db_index_usage_stats AS ddius
  ON i.databaseID = ddius.database_id
  AND i.objectID = ddius.object_id
  AND i.indexID = ddius.index_id
CROSS APPLY sys.dm_os_sys_info AS dosi
WHERE /* exclude system databases */
  i.databaseName NOT IN('master','msdb','tempdb','model')
/* exclude unique indexes; assume they are serving a business function */
  AND i.isUnique = 0
/* exclude primary keys; assume they are serving a business function */
  AND i.isPrimaryKey = 0
/* no seeks have been performed since the last server reboot */
  AND ddius.user_seeks = 0;

```

■ **Tip** The T-SQL in Listing 1-7 is just a starting point. You can easily modify this query to return information such as which clustered indexes may warrant redesign, which tables have the most updates, and which tables are the most frequently queried.

An example of the output follows.

```
captureDate      serverName schemaName databaseName      tableName
-----
2014-06-29 19:37:36.927 LOCALHOST Production AdventureWorks2012 TransactionHistory
2014-06-29 19:37:36.927 LOCALHOST Production AdventureWorks2012 TransactionHistory
2014-06-29 19:37:36.927 LOCALHOST Sales AdventureWorks2012 SalesOrderDetail

indexName                      indexType  isFiltered isPartitioned numberOfRows
-----
IX_TransactionHistory_ProductID NONCLUSTERED 0          0          1134431
IX_TransactionHistory_ReferenceOrderID NONCLUSTERED 0          0          1134431
IX_SalesOrderDetail_ProductID NONCLUSTERED 0          1          1213178

userSeeksSinceReboot userScansSinceReboot userLookupsSinceReboot userUpdatesSinceReboot
-----
0                    0                    0                    98
0                    8                    0                    98
0                    2                    0                    124

indexSizeInMB lastReboot
-----
9            2014-06-28 19:15:28.837
21           2014-06-28 19:15:28.837
28           2014-06-28 19:15:28.837
```

As you can see, this query is a bit more complex than the last one. Let's discuss what you're doing. Developers are usually very good at identifying performance issues. Why? When a query is slow, someone is usually complaining about it! It's not uncommon for the fix to involve creating an index, which can reduce IO and improve query duration. Over time, however, the query may change—resulting in the optimizer using different indexing—or perhaps the query is no longer needed. Unlike the user-impacting issues that garner so much attention, these types of changes tend to creep up silently over time. Eventually that same index that was so beneficial when it was first being used starts consuming unnecessary resources—specifically, it slows down insert speed, consumes precious disk space, and inflates backup sizes.

One way to stay on top of unused indexes is to search the `sys.dm_db_index_usage_stats` DMV. This DMV keeps track of index utilization statistics, including how many times an index has been sought or scanned and how many updates have been performed. This information is refreshed after every reboot, so please note that a server that has been restarted recently may show an inaccurately high number of “unused” indexes. Also, this information is merely a starting point from which you can conduct further research into whether an index should be dropped or redesigned; many organizations may have indexes that are not called frequently but are necessary for important monthly or annual reports.

One other important thing to note is that this script makes use of the undocumented `sp_MSforeachdb` stored procedure. This stored procedure performs a very useful task: it iterates through every database, executing whatever command is passed to it. For numerous reasons—not the least of which is the fact that it is an undocumented, and therefore unsupported, stored procedure that may occasionally skip databases—we recommend using Aaron Bertrand's `sp_foreachdb` stored procedure instead for production workloads. However, to keep things simple, you will use the `sp_MSforeachdb` procedure in your example.

■ **Tip** Aaron Bertrand's `sp_foreachdb` stored procedure can be found at www.mssqltips.com/sqlservertip/2201/making-a-more-reliable-and-flexible-spmsforeachdb.

Now that you better understand the query and the expected output, let's return to your package. Click Parse Query to ensure you do not have any errors in your syntax, and then click Preview to see a sample of the results. Click the Columns page to ensure that the column list has been successfully updated; then click OK to return to the Data Flow designer.

You should now see an error in your data flow, as illustrated in Figure 1-19. This is expected because you've changed the columns that your data source is providing, but your destination still expects the old column list.

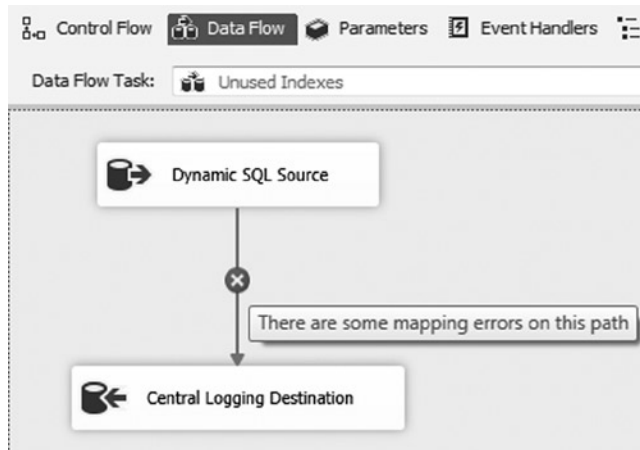


Figure 1-19. The completed data flow task

Before you can fix this error, you need to return to SSMS to create a table to store this data. Create this table now using the code in Listing 1-8.

Listing 1-8. T-SQL Code to Create the `dba_monitor_unusedIndexes` Table

```
USE dbaCentralLogging;
GO

CREATE TABLE dbo.dba_monitor_unusedIndexes
(
    log_id                INT IDENTITY(1,1)
    ,captureDate          DATETIME
    ,serverName           NVARCHAR(128)
    ,schemaName           SYSNAME
    ,databaseName         SYSNAME
    ,tableName            SYSNAME
    ,indexName            SYSNAME
    ,indexType            NVARCHAR(60)
    ,isFiltered           BIT
    ,isPartitioned        BIT
)
```

```

,numberOfRows          BIGINT
,userSeeksSinceReboot   BIGINT
,userScansSinceReboot   BIGINT
,userLookupsSinceReboot BIGINT
,userUpdatesSinceReboot BIGINT
,indexSizeInMB          BIGINT
,lastReboot             DATETIME

CONSTRAINT PK_dba_monitor_unusedIndexes
PRIMARY KEY NONCLUSTERED(log_id)
);

CREATE CLUSTERED INDEX CIX_dba_monitor_unusedIndexes
ON dbo.dba_monitor_unusedIndexes(captureDate);

```

Return to Visual Studio and double-click the Central Logging Database icon to edit its properties. Change the Name of the Table or the View value to [dbo].[dba_monitor_unusedIndexes], and then click the Mappings page. Because your source and destination are using the same column names, you can easily update the mappings by right-clicking in the space between Available Input Columns and Available Destination Columns and selecting Map Items by Matching Names. Figure 1-20 illustrates this option.

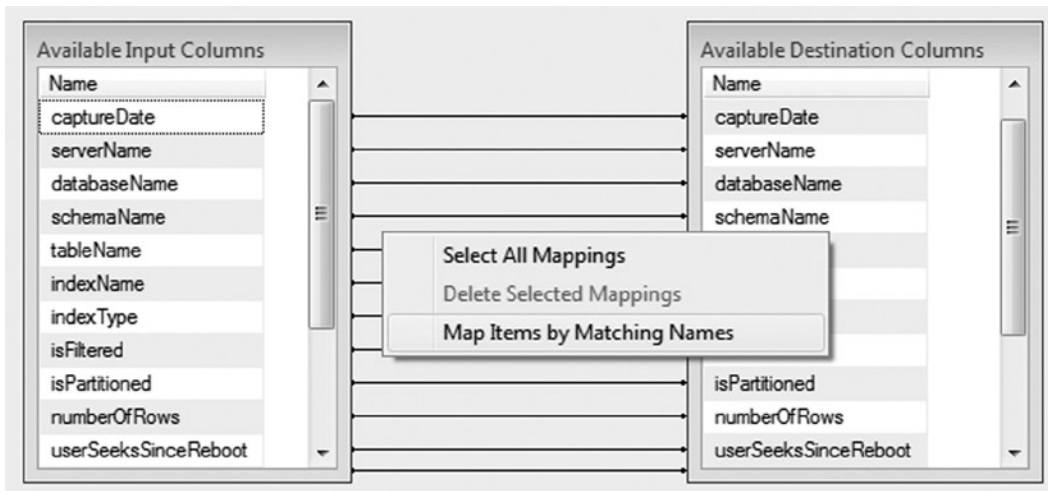


Figure 1-20. The Map Items by Matching Names option in the Mappings page

Once more, the log_id column will not map to anything because it is an identity column. Click OK to return to the Data Flow designer, and then click the Control Flow tab.

See how quickly that second data flow went? You can continue to easily add more metadata collection tasks using this method. All that you have left to do is to update your LastMonitored column in the dba_monitor_SQLServerInstances table.

Tip It may be tempting to create a one-size-fits-all package. However, it is generally a better idea to separate metadata collections into separate packages organized by frequency requirements. For example, the metadata you have collected in this chapter only requires periodic samples, such as daily or weekly collection. Metadata that requires more frequent collection, such as an hourly check for failed SQL Server Agent jobs, should be stored in a separate package.

Add an Execute SQL task to your Foreach Loop container and rename it **Update LastMonitored**. Connect the Unused Indexes data flow to the Update LastMonitored Execute SQL task. Double-click the Execute SQL task to edit its properties. Select the dbaCentralLogging connection in the Connection drop-down, and then enter the code from Listing 1-9 in the SQLStatement field.

Listing 1-9. T-SQL Code to Update the LastMonitored Value in dba_monitor_SQLServerInstances

```
UPDATE dbo.dba_monitor_SQLServerInstances
SET LastMonitored = GETDATE()
WHERE SQLServerInstance= ?;
```

The question mark (?) informs the Execute SQL task to use a parameter to complete the SQL statement. Now you just need to map a variable to the parameter. To do this, click the Parameter Mapping page and click Add. Edit the properties as follows:

- Variable Name = **User::SQLServerInstance**
- Direction = **Input**
- Data Type = **NVARCHAR**
- Parameter Name = **0**
- Parameter Size = **128**

Confirm that your mappings match those shown in Figure 1-21, and then click OK.

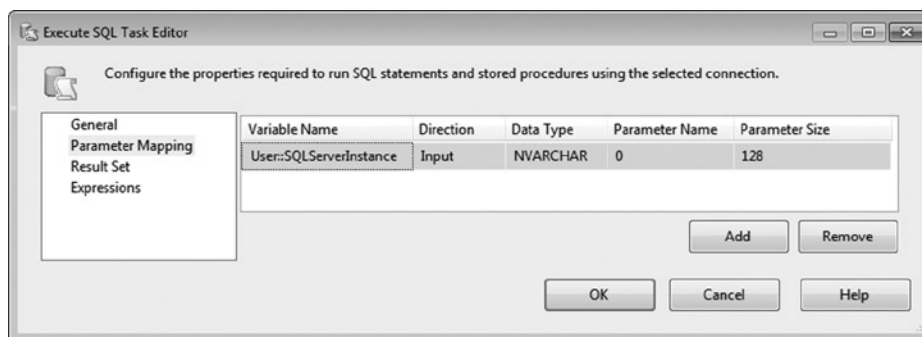


Figure 1-21. Parameter mapping in the Execute SQL Task Editor

You are now ready to execute your package! To do this, select Debug ► Start Debugging from the menu, click the green Run icon in the toolbar, or press F5. Your package should resemble Figure 1-22 when successfully executed.

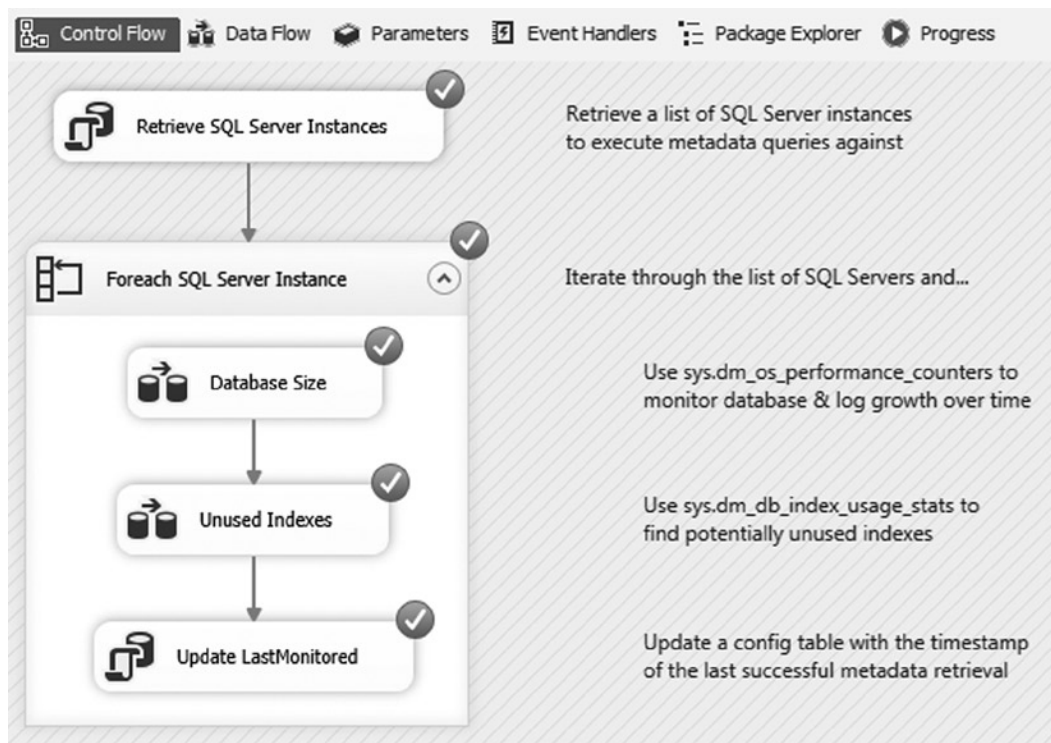


Figure 1-22. Successful execution of the *MetadataCollection* package

Congratulations! You have now collected metadata from a list of servers and stored the results in a single, centralized database.

This concludes our walk-through on collecting metadata via SSIS. However, as a diligent developer or DBA, you may want to consider many more tasks. First, as we discussed early on in this chapter, this package does not contain any exception handling or logging, which is outside the scope of this chapter. However, a best practice is to include some sort of exception handling and logging on every package. Second, we have only tipped the proverbial iceberg when it comes to collecting metadata. There is much more information to consider, such as security audits, error logs, SQL Server Agent job statuses, and much, much more. If you're not sure where to start, consider ranking metadata tasks by criticality and adding incremental monitoring in descending order of importance. As a last piece of homework, you may want to consider setting up monitors to alert you when unfavorable conditions are met (for example, a SQL Server Agent job has failed or available space is getting low).

Summary

In this chapter, we discussed the importance of metadata. We explored some of the metadata that exists within SQL Server and provided two examples of useful T-SQL metadata queries. We identified a very flexible and reusable pattern for collecting database metadata in an enterprise environment. Lastly, we created an Integration Services package that retrieves a list of SQL Server instances to monitor and then logs the results to a centralized repository.



Execution Patterns

To fully understand SQL Server 2014 Integration Services execution, you must first understand the different deployment models. There are two: the package deployment model and the project deployment model. Each exposes and supports a different functionality. The package deployment model primarily supports legacy functionality. It is the model used in SSIS 2005 through SSIS 2008 R2. The new way of doing things involves the project deployment model. Certain execution methods, but not all, are available to both deployment models.

You can build awesome SQL Server Integration Services (SSIS) packages, but they do you no good until you execute them! SSIS provides several methods for package execution. In this chapter, we will examine the following:

- Debug execution
- Command-line execution
- The Execute Package utility
- SQL Server 2014 Integration Services
- Integration Services catalogs
- Integration Services catalog stored procedures
- Scheduling SSIS package execution
- The Execute Package task
- Metadata-driven execution
- Execution from managed code

We'll begin by creating a simple SSIS package to use for demonstration purposes.

Building the Demonstration SSIS Package

Create a new SSIS solution named **Chapter2**. Rename the SSIS package, changing the name from `Package.dtsx` to **Chapter2.dtsx**.

■ **Tip** For more information on creating SSIS solutions and packages, see *Professional SQL Server 2014 Integration Services* by Michael Coles and Francis Rodrigues (Apress, 2012).

Drag a script component onto the control flow canvas and open the editor. Choose your language of choice in the ScriptLanguage property on the Script page. Select the System::PackageName variable in ReadOnlyVariables, and then click the Edit Script button.

If you selected Microsoft Visual Basic 2010 as the ScriptLanguage property setting for the Script task, replace the code in Public Sub Main() with the following:

```
Public Sub Main()

    Dim sPackageName As String = Dts.Variables("PackageName").Value.ToString
    Dim sMsg As String = "Package Name: " & sPackageName

    MsgBox(sMsg, , sPackageName)

    Dts.TaskResult = ScriptResults.Success
End Sub
```

If you selected Microsoft Visual C# 2010 as the ScriptLanguage property setting for the Script task, replace the code in public void Main() with the following:

```
public void Main()
{
    string sPackageName = Dts.Variables["PackageName"].Value.ToString();
    string sMsg = "Package Name: " + sPackageName;

    MessageBox.Show(sMsg, sPackageName);

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

Save the package, project, and solution. You're ready to run!

Debug Execution

Executing the package from within SQL Server Business Intelligence Development Studio (BIDS) is straightforward. It works the same regardless of the deployment model selected. However, as with everything in the Visual Studio Integrated Development Environment (VS IDE), you have several ways to accomplish this.

When you execute an SSIS package inside BIDS, you are invoking the SSIS debugger. The SSIS debugger file is named DtsDebugHost.exe and it's stored in the <drive>:\Program Files\Microsoft SQL Server\120\DTS\Binn folder. It's important to realize you're executing the SSIS package inside a debug host process. Why? There is overhead associated with debugging—those boxes don't change color for free!

To execute the Chapter2.dtsx package in BIDS, press the F5 key. The debug host loads, then it loads the package and executes it. You should see a message box proclaiming the package name. When you click the OK button on the message box, the Script task in the Chapter2 package control flow turns from yellow to green. A link appears beneath the Connections Managers tab to indicate package execution has completed. However, the DtsDebugHost.exe process is still executing. It continues executing until the BIDS debugger is stopped.

Here are some ways to start the BIDS debugger:

- Press the F5 key.
- Click the VCR Play button (the green arrow pointing right) on the toolbar.
- Click the Debug drop-down menu and select Start Debugging.

■ **Note** Actually, selecting Step Into or Step Over from the Debug drop-down menu also starts the BIDS debugger.

- In Solution Explorer, right-click the package and select Execute Package from the menu.
- When the package has completed execution in Debug mode, restart the package in one of these ways:
 - By holding Ctrl+Shift and pressing the F5 key
 - Using the VCR Restart button on the toolbar
 - Clicking the Debug drop-down menu and clicking Restart

Here are some ways to stop the debugger once the package execution completes (or whenever a Debug mode Stop is desired):

- Hold Shift and press the F5 key.
- Click the VCR Stop button (the square) on the toolbar.
- Click the Debug drop-down menu and select Stop Debugging.
- Click the Debug drop-down menu and select Terminate All.
- Click the Package Execution Completed link beneath the Connection Managers tab.

Command-Line Execution

Command-line SSIS package execution uses the DTEXEC utility (DtExec.exe). DTEXEC supports project and package deployment models. You can manually invoke DTEXEC from inside BIDS by clicking the Debug drop-down menu and selecting Start Without Debugging (or by holding the Ctrl key and pressing F5). You can also manually start DTEXEC from a command prompt.

DTEXEC isn't often invoked manually. Instead it's common to see DTEXEC command lines used with scheduling software to execute SSIS packages in production environments. For example, when you schedule an SSIS package using SQL Server Agent (covered later in this chapter), DTEXEC is instantiated.

To execute the Chapter2.dtsx SSIS package using DTEXEC, open a command prompt and enter the following command:

```
dtexec /FILE "G:\Projects\SSIS Design Patterns\SSIS Design Patterns\Chapter2.dtsx"
```

This command executes the Chapter2.dtsx SSIS package located in the G:\Projects\SSIS Design Patterns\SSIS Design Patterns folder. Edit the command line to reflect the location of your SSIS package if you're playing along at home.

When you execute the package from the command line, the message box displays the package name—as it does when the package is executed from inside the BIDS debugger.

If the SSIS package is deployed to the new SSIS catalog, you can still execute it from the command line using a command similar to this one:

```
dtexec.exe /ISSERVER "\"SSISDB\Chapter2\Chapter2\Chapter2.dtsx\"" /SERVER "\"SSISMVP-RCO\"" /Par "\"$ServerOption::SYNCHRONIZED(Boolean)\"";True /REPORTING E /CALLERINFO Andy
```

Execute Package Utility

The Execute Package Utility (DtExecUI) runs in its own process and executes SSIS packages. I like using the Execute Package Utility to build DTEXEC command lines, but it only supports the package deployment model. You can invoke the Execute Package Utility in at least three ways:

- Click Start ► All Programs ► Microsoft SQL Server ► Integration Services ► Execute Package Utility.
- Click Start ► Run and type **dtexecui** in the Open text box.
- Double-click on a dtsx file (if you haven't remapped the default application settings for dtsx files).

My favorite option is double-clicking the dtsx file. This not only opens the Execute Package Utility, but it sets the General page settings to indicate that the package source is the file system and configures the package path text box with the full path of the dtsx file I double-clicked. Neat.

If I execute Package2.dtsx using the Execute Package Utility, the Package Execution Progress form displays, informing me of the package's execution progress (how appropriate) and the message box appears as it did when I executed using the BIDS debugger and the command line.

■ **Note** See *Professional SQL Server 11 Integration Services by Michael Coles and Francis Rodrigues (Apress, 2012)* for more information about the Execute Package Utility.

The SQL Server 2014 Integration Services Service

The SQL Server Integration Services 11.0 service installs with SQL Server 2014. To connect, open SQL Server Management Studio (SSMS). If prompted to connect with the Connect To Server window at SSMS startup, make sure Server Type is set to Integration Services. Enter the name of the server in the Server Name drop-down. Please note there aren't named instances of SSIS: there's one per server (for now, anyway). You can also enter **localhost** to connect to the local server's default instance of SSIS.

Once the connection is configured, click the Connect button. Navigate to the package you desire to execute. SSIS packages stored in the file system or the MSDB database can be executed from the SSIS 2014 service.

SQL Server 2014 provides a new way to manage and execute Integration Services packages: Integration Services catalogs. We explore this method next.

Integration Services Catalogs

You can only manage SSIS projects that use the project deployment model in Integration Services catalogs. To execute a package in the catalog, use SSMS to connect to the instance of SQL Server hosting the SSISDB database. Expand the Integration Services Catalogs node and then expand the SSISDB node. Drill into the folder containing the SSIS project and package(s). Right-click the package you wish to execute and click Execute, as shown in Figure 2-1.

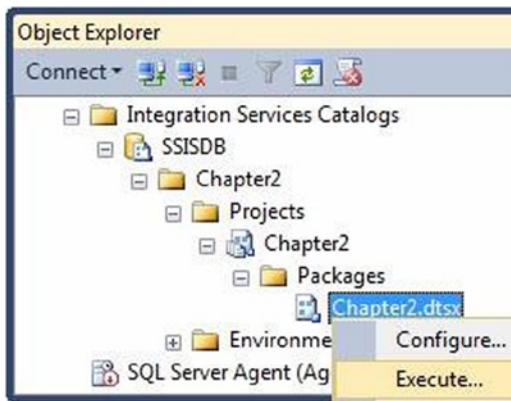


Figure 2-1. Executing an SSIS package deployed to the SSIS Catalog

The Execute Package window displays, as shown in Figure 2-2. It allows you to override parameter values, ConnectionString properties of connection managers built at design-time, or any other externalize-able property accessible from a package path (via the Advanced tab) for this execution instance of the SSIS package stored in the SSIS Catalog.

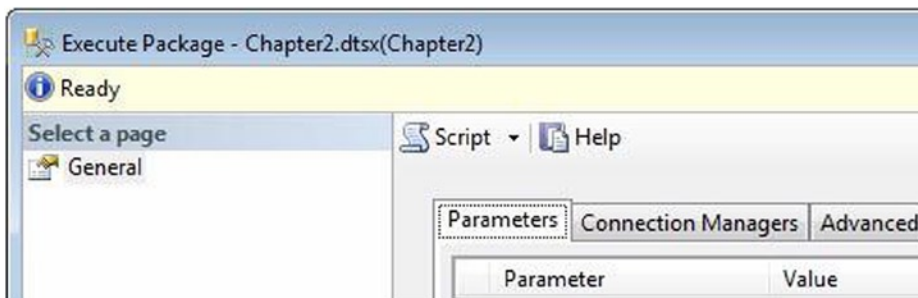


Figure 2-2. Execute Package window

Integration Server Catalog Stored Procedures

Please note the Script button above the Parameters tab in Figure 2-2. This button allows you to generate Transact-SQL (T-SQL) statements that will execute the SSIS package. For the Chapter2.dtsx package stored in the SSIS Catalog, the scripts will appear similar to those in Listing 2-1.

Listing 2-1. T-SQL Script Generated from the Execute Package Window

```
Declare @execution_id bigint
EXEC [SSISDB].[catalog].[create_execution]
    @package_name=N'Chapter2.dtsx'
    ,@execution_id=@execution_id OUTPUT
    ,@folder_name=N'Chapter2'
    ,@project_name=N'Chapter2'
    ,@use32bitruntime=False
    ,@reference_id=NULL
```

```

Select @execution_id
DECLARE @var0 smallint = 1
EXEC [SSISDB].[catalog].[set_execution_parameter_value]
    @execution_id
    ,@object_type=50
    ,@parameter_name=N'LOGGING_LEVEL'
    ,@parameter_value=@var0
EXEC [SSISDB].[catalog].[start_execution] @execution_id
GO

```

You can use these same stored procedures to execute SSIS packages in the SSIS Catalog! In fact, I designed a script to create a wrapper-stored procedure that will call the T-SQL statements executed when an SSIS package is executed in the SSIS Catalog. You can see that script in Listing 2-2.

Listing 2-2. Script to Build a Wrapper-Stored Procedure for Executing SSIS Packages in the SSIS Catalog

```

/* Select the SSISDB database */
Use SSISDB
Go

/* Create a parameter (variable) named @Sql */
Declare @Sql varchar(2000)

/* Create the Custom schema if it does not already exist */
print 'Custom Schema'
If Not Exists(Select name
              From sys.schemas
              Where name = 'custom')
begin
    /* Create Schema statements must occur first in a batch */
    print ' - Creating custom schema'
    Set @Sql = 'Create Schema custom'
    Exec(@Sql)
    print ' - Custom schema created'
end
Else
    print ' - Custom Schema already exists.'
print ''

/* Drop the Custom.execute_catalog_package Stored Procedure if it already exists */
print 'Custom.execute_catalog_package Stored Procedure'
If Exists(Select s.name + '.' + p.name
          From sys.procedures p
          Join sys.schemas s
            On s.schema_id = p.schema_id
          Where s.name = 'custom'
          And p.name = 'execute_catalog_package')
begin
    print ' - Dropping custom.execute_catalog_package'
    Drop Procedure custom.execute_catalog_package
    print ' - Custom.execute_catalog_package dropped'
end

```



```

/* Create the Custom.execute_catalog_package Stored Procedure */
print ' - Creating custom.execute_catalog_package'
go

/*

Stored Procedure: custom.execute_catalog_package
Author: Andy Leonard
Date: 4 Mar 2012
Description: Creates a wrapper around the SSISDB Catalog procedures
             used to start executing an SSIS Package. Packages in the
             SSIS Catalog are referenced by a multi-part identifier
             - or path - that consists of the following hierarchy:
Catalog Name: Implied by the database name in Integration Server 2014
|-Folder Name: A folder created before or at Deployment to contain the SSIS project
|-Project Name: The name of the SSIS Project deployed
|-Package Name: The name(s) of the SSIS Package(s) deployed

Parameters:
@FolderName [nvarchar(128)] {No default} -
    contains the name of the Folder that holds the SSIS Project
@ProjectName [nvarchar(128)] {No default} -
    contains the name of the SSIS Project that holds the SSIS Package
@PackageName [nvarchar(260)] {No default} -
    contains the name of the SSIS Package to be executed
@ExecutionID [bigint] {Output} -
    Output parameter (variable) passed back to the caller
@LoggingLevel [varchar(16)] {Default} -
    contains the (case-insensitive) name of the logging level
    to apply to this execution instance
@Use32BitRunTime [bit] {Default} -
    1 == Use 64-bit run-time
                                0 == Use 32-bit run-time
@ReferenceID [bigint] {Default} -reference to Execution Environment
@ObjectType [smallint] -identifier related to PackageType property
Guessing: @ObjectType == PackageType.ordinal (1-based-array) * 10
    Must be 20, 30, or 50 for catalog.set_execution_parameter_value
    stored procedure

Test:
1. Create and deploy an SSIS Package to the SSIS Catalog.
2. Exec custom.execute_catalog_package and pass it the
   following parameters: @FolderName, @ProjectName, @PackageName, @ExecutionID Output
@LoggingLevel, @Use32BitRunTime, @ReferenceID, and @ObjectType are optional and
defaulted parameters.

```

Example:

```

    Declare @ExecId bigint
    Exec custom.execute_catalog_package
    'Chapter2'
    , 'Chapter2'
    , 'Chapter2.dtsx'
    , @ExecId Output
3. When execution completes, an Execution_Id value should be returned.
View the SSIS Catalog Reports to determine the status of the execution
instance and the test.

```

*/

Create Procedure custom.execute_catalog_package

```

    @FolderName nvarchar(128)
    , @ProjectName nvarchar(128)
    , @PackageName nvarchar(260)
    , @ExecutionID bigint Output
    , @LoggingLevel varchar(16) = 'Basic'
    , @Use32BitRunTime bit = 0
    , @ReferenceID bigint = NULL
    , @ObjectType smallint = 50

```

As

begin

Set NoCount ON

```

    /* Call the catalog.create_execution stored procedure
    to initialize execution location and parameters */

```

```

Exec catalog.create_execution
    @package_name = @PackageName
    , @execution_id = @ExecutionID Output
    , @folder_name = @FolderName
    , @project_name = @ProjectName
    , @use32bitruntime = @Use32BitRunTime
    , @reference_id = @ReferenceID

```

```

    /* Populate the @ExecutionID parameter for OUTPUT */

```

```

Select @ExecutionID As Execution_Id

```

```

    /* Create a parameter (variable) named @Sql */

```

```

Declare @logging_level smallint

```

```

    /* Decode the Logging Level */

```

```

Select @logging_level = Case
    When Upper(@LoggingLevel) = 'BASIC'
    Then 1
    When Upper(@LoggingLevel) = 'PERFORMANCE'
    Then 2
    When Upper(@LoggingLevel) = 'VERBOSE'
    Then 3
    Else 0 /* 'None' */
End

```

```

/* Call the catalog.set_execution_parameter_value stored
   procedure to update the LOGGING_LEVEL parameter */
Exec catalog.set_execution_parameter_value
    @ExecutionID
    ,@object_type = @ObjectType
    ,@parameter_name = N'LOGGING_LEVEL'
    ,@parameter_value = @logging_level

/* Call the catalog.start_execution (self-explanatory) */
Exec catalog.start_execution
    @ExecutionID

end

GO

```

If you execute this script to create the custom schema and stored procedure in your instance of the SSISDB database, you can test it using the statement in Listing 2-3.

Listing 2-3. Testing the SSISDB.custom.execute_catalog_package Stored Procedure

```

Declare @ExecId bigint
Exec SSISDB.custom.execute_catalog_package 'Chapter2', 'Chapter2', 'Chapter2.dtsx',
@ExecId Output

```

Adding a Data Tap

The SSISDB.custom.execute_catalog_package stored procedure can be modified slightly to create a *data tap*—a new feature for packages executed from the SSISDB Catalog in SSIS 2014. Adding a few parameters and some T-SQL to the stored procedure allows it to execute an SSIS package and export a comma-separated values (CSV) file filled with some or all of the rows that flowed through a point in a Data Flow task. Data taps provide a much-needed window on the state of data as they move through an SSIS data flow, facilitating root-cause analysis and troubleshooting in production environments without altering the package code. Data taps are one of the most important enhancements to Integration Services 2014. Listing 2-4 contains the script to build SSISDB.custom.execute_catalog_package_with_data_tap:

Listing 2-4. Script to Build a Wrapper-Stored Procedure for Executing SSIS Packages in the SSIS Catalog

```

/* Select the SSISDB database */
Use SSISDB
Go

/* Create a parameter (variable) named @Sql */
Declare @Sql varchar(2000)

/* Create the Custom schema if it does not already exist */
print 'Custom Schema'
If Not Exists(Select name
              From sys.schemas
              Where name = 'custom')

```

```

begin
    /* Create Schema statements must occur first in a batch */
    print ' - Creating custom schema'
    Set @Sql = 'Create Schema custom'
    Exec(@Sql)
    print ' - Custom schema created'
end
Else
    print ' - Custom Schema already exists.'
print ''

/* Drop the Custom.execute_catalog_package_with_data_tap
Stored Procedure if it already exists */
print 'Custom.execute_catalog_package_with_data_tap Stored Procedure'
If Exists(Select s.name + '.' + p.name
          From sys.procedures p
          Join sys.schemas s
              On s.schema_id = p.schema_id
          Where s.name = 'custom'
          And p.name = 'execute_catalog_package_with_data_tap'))
begin
    print ' - Dropping custom.execute_catalog_package_with_data_tap'
    Drop Procedure custom.execute_catalog_package_with_data_tap
    print ' - Custom.execute_catalog_package_with_data_tap dropped'
end

/* Create the Custom.execute_catalog_package_with_data_tap Stored Procedure */
print ' - Creating custom.execute_catalog_package_with_data_tap'
go

/*

Stored Procedure: custom.execute_catalog_package_with_data_tap
Author: Andy Leonard
Date: 4 Apr 2012
Description: Creates a wrapper around the SSISDB Catalog procedures
            used to start executing an SSIS Package and create a
            data tap. Packages in the
            SSIS Catalog are referenced by a multi-part identifier
            - or path - that consists of the following hierarchy:
Catalog Name: Implied by the database name in Integration Server 2014
|-Folder Name: A folder created before or at Deployment to contain the SSIS project
|-Project Name: The name of the SSIS Project deployed
|-Package Name: The name(s) of the SSIS Package(s) deployed
Parameters:
    @FolderName [nvarchar(128)] {No default} - contains the name of the
        Folder that holds the SSIS Project
    @ProjectName [nvarchar(128)] {No default} - contains the name of the
        SSIS Project that holds the SSIS Package
    @PackageName [nvarchar(260)] {No default} - contains the name of the
        SSIS Package to be executed

```

```
@ExecutionID [bigint] {Output} - Output parameter (variable) passed back
    to the caller
@LogLevel [varchar(16)] {Default} - contains the (case-insensitive)
    name of the logging level to apply to this execution instance
@Use32BitRunTime [bit] {Default} - 1 == Use 64-bit run-time
                                     0 == Use 32-bit run-time
@ReferenceID [bigint] {Default} - contains a reference to an Execution Environment
@ObjectType [smallint] - contains an identifier that appears to be related
    to the SSIS PackageType property

Guessing: @ObjectType == PackageType.ordinal (1-based-array) * 10
    Must be 20, 30, or 50 for catalog.set_execution_parameter_value
    stored procedure
@DataFlowTaskName [nvarchar(255)] - contains the name of the Data Flow Task in which to
    to apply the data tap.
@IdentificationString [nvarchar(255)] - contains the Data Flow Path Identification string
    in which to apply the data tap.
@DataTapFileName [nvarchar(4000)] - contains the name of the file to create to contain
    the rows captured from the data tap.
    Saved in the <drive>:\Program Files\Microsoft SQL Server\120\DTs\DataDumps folder.
@DataTapMaxRows [int] - contains the maximum number of rows to send to the data tap file.
```

Test:

1. Create and deploy an SSIS Package to the SSIS Catalog.
 2. Exec custom.execute_catalog_package_with_data_tap and pass it the following parameters: @FolderName, @ProjectName, @PackageName, @DataFlowTaskName, @IdentificationString, @DataTapFileName, @ExecutionID Output
- @LoggingLevel, @Use32BitRunTime, @ReferenceID, @ObjectType, and @DataTapMaxRows are optional and defaulted parameters.

Example:

```

Declare @ExecId bigint
Exec custom.execute_catalog_package_with_data_tap
'SSISConfig2014','SSISConfig2014','Child1.dtsx',
'Data Flow Task','OLESRC Temperature.OLE DB Source Output',
'Child1 DataFlowTask OLESRCTemperature OLEDBSourceOutput.csv',@ExecId Output

```

3. When execution completes, an `Execution_Id` value should be returned. View the SSIS Catalog Reports to determine the status of the execution instance and the test.

*/

```
Create Procedure [custom].[execute_catalog_package_with_data_tap]
    @FolderName nvarchar(128)
    ,@ProjectName nvarchar(128)
    ,@PackageName nvarchar(260)
    ,@DataFlowTaskName nvarchar(255)
    ,@IdentificationString nvarchar(255)
    ,@DataTapFileName nvarchar(4000)
    ,@ExecutionID bigint Output
```

```

,@LoggingLevel varchar(16) = 'Basic'
,@Use32BitRunTime bit = 0
,@ReferenceID bigint = NULL
,@ObjectType smallint = 50
,@DataTapMaxRows int = NULL
As

begin

Set NoCount ON

/* Call the catalog.create_execution stored procedure
to initialize execution location and parameters */
Exec catalog.create_execution
    @package_name = @PackageName
,@execution_id = @ExecutionID Output
,@folder_name = @FolderName
,@project_name = @ProjectName
,@use32bitruntime = @Use32BitRunTime
,@reference_id = @ReferenceID

/* Populate the @ExecutionID parameter for OUTPUT */
Select @ExecutionID As Execution_Id

/* Configure Data Tap parameters */
If (Left(@DataFlowTaskName, 9) <> '\Package\')
    Set @DataFlowTaskName = '\Package\' + @DataFlowTaskName

If Left(@IdentificationString,6) <> 'Paths['
    Set @IdentificationString = 'Paths[' + @IdentificationString + ']'

/* Create the Data Tap */
EXEC [SSISDB].[catalog].add_data_tap @ExecutionID, @DataFlowTaskName,
    @IdentificationString, @DataTapFileName, @DataTapMaxRows

/* Create a parameter (variable) named @Sql */
Declare @logging_level smallint
/* Decode the Logging Level */
Select @logging_level = Case
    When Upper(@LoggingLevel) = 'BASIC'
        Then 1
    When Upper(@LoggingLevel) = 'PERFORMANCE'
        Then 2
    When Upper(@LoggingLevel) = 'VERBOSE'
        Then 3
    Else 0 /* 'None' */
End

```

```

/* Call the catalog.set_execution_parameter_value stored
   procedure to update the LOGGING_LEVEL parameter */
Exec catalog.set_execution_parameter_value
    @ExecutionID
    ,@object_type = @ObjectType
    ,@parameter_name = N'LOGGING_LEVEL'
    ,@parameter_value = @logging_level

/* Call the catalog.start_execution (self-explanatory) */
Exec catalog.start_execution
    @ExecutionID

end

```

Testing the Data Tap Procedure

Before we begin this exercise, please visit http://andyweather.com/data/WeatherData_Dec08.zip to obtain some real-world temperature and humidity weather data collected at my weather station in Farmville, Virginia. The compressed file (WeatherData_Dec08.zip) contains a single CSV file named sensor1-all.csv. The file is two folders down, located at Dec08\TH\sensor1-all.csv. Unzip the compressed file and store it in your file system. I prefer to store data related to test projects inside a folder named Data in the SSIS solution directory. It doesn't matter where you store the file as long as you remember where you put it.

Add a new SSIS package to the Chapter2 project and rename it **DataTapTest.dtsx**. Drag a Data Flow task onto the control flow and rename it **DFT Load Temperature Data**. Open the Data Flow Editor (tab) and drag a Flat File source adapter onto the design surface. Rename the Flat File source adapter **FFSrc Temperature**. Open the Flat File Source Adapter Editor and click the New button to the right of the Flat File Connection Manager drop-down. Clicking the New button does a couple things for you:

1. It creates a new Flat File connection manager.
2. It opens the new Flat File Connection Manager Editor.

Set the name of the Flat File connection manager to **FFCM Temperature**. Click the Browse button and navigate to the folder containing the sensor1-all.csv file (remember to change the extension filter in the Open dialog from *.txt to *.csv). Select the sensor1-all.csv file and then click the OK button on the Open dialog since we are accepting the Flat File connection manager defaults for the purposes of this demo. Click the OK button to close the Flat File Source Adapter Editor.

Before we proceed, open SQL Server Management Studio (SSMS), connect to an instance of SQL Server 2014, and create a database named **TestDB**.

Return to SQL Server Data Tools - Business Intelligence (SSDT-BI) and drag an OLE DB destination adapter onto the Data Flow task surface. Rename the OLE DB destination adapter **OLEDBDest TemperatureStage** and connect a data flow path (blue arrow) between the FFSrc Temperature Flat File source adapter and the OLEDBDest TemperatureStage OLE DB destination adapter. Open the OLEDBDest TemperatureStage OLE DB Destination Adapter Editor and click the New button to the right of the OLE DB Connection Manager drop-down to create a new OLE DB connection manager and open its editor. When the Configure OLE DB Connection Manager window displays, click the New button to open the Connection Manager Editor window. Enter the server name and user login credentials, then enter or select the TestDB database in the Select or Enter a Database Name drop-down. Close the editor and the Configure OLE DB Connection Manager window to return to the OLEDBDest TemperatureStage OLE DB destination adapter.

Click the New button to the right of the Name of the Table or the View drop-down in the OLEDBDest TemperatureStage OLE DB destination adapter. Modify the contents of the Create Table window to match the data definition language (DDL) statement in Listing 2-5.

Listing 2-5. Data Definition Language (DDL) Create Table Statement for Destination

```
CREATE TABLE [TemperatureStage] (
    [Date] varchar(50),
    [Time] varchar(50),
    [MinT] varchar(50),
    [MaxT] varchar(50),
    [AverageT] varchar(50),
    [MinH] varchar(50),
    [MaxH] varchar(50),
    [AverageH] varchar(50),
    [ComfortZone] varchar(50),
    [MinDP] varchar(50),
    [MaxDP] varchar(50),
    [AverageDP] varchar(50),
    [MinHI] varchar(50),
    [MaxHI] varchar(50),
    [AverageHI] varchar(50)
)
```

This statement modifies the supplied statement by removing the OLEDBDest prefix from the table name and by removing the spaces from the column names. Click the OK button to create the table and close the Create Table window. Click the Mappings page and map the available input columns to their matching available destination columns. Close the OLEDBDest TemperatureStage OLE DB Destination Adapter Editor.

Double-click the data flow path to open its editor. From the General page, copy the IdentificationString property value. The IdentificationString property should be Paths[FFSrc Temperature.Flat File Source Output]. We will need this value and the value of the PackagePath property of the Data Flow task (\Package\DFT Load Temperature Data) to execute this package with a data tap.

First, save the SSIS package and deploy it to the catalog. Next, execute the statement in Listing 2-6 to execute the SSIS package with a data tap.

Listing 2-6. Executing an SSIS Package with a Data Tap

```
Declare @ExecId bigint
Exec SSISDB.custom.execute_catalog_package_with_Data_tap
    @FolderName = 'Chapter2'
    ,@ProjectName = 'Chapter2'
    ,@PackageName = 'DataTapTest.dtsx'
    ,@DataFlowTaskName = '\Package\DFT Load Temperature Data'
    ,@IdentificationString = 'Paths[FFSrc Temperature.Flat File Source Output]'
    ,@DataTapFileName = 'TemperatureRows.csv'
    ,@ExecutionID = @ExecId Output
    ,@DataTapMaxRows = 25
```

Once the package executes (and you can test for rows in the TestDB.dbo.TemperatureStage table), you should find a CSV file named TemperatureRows.csv in the <drive>:\Program Files\Microsoft SQL Server\120\DTS\DataDumps directory, and that file should contain the first 25 rows of data that flowed through the "FFSrc Temperature.Flat File Source Output" data flow path in the "DFT Load Temperature Data" in the DataTapTest.dtsx SSIS package.

Creating a Custom Execution Framework

SSIS execution frameworks support repeatable and reliable SSIS package execution. The `SSISDB.custom.execute_catalog_package` stored procedure can be used as the centerpiece for an SSIS execution framework. To create the tables to support this framework, execute the statements in Listing 2-7.

Listing 2-7. Tables to Support a Custom SSIS Execution Framework

```

/* Switch to SSISDB database */
Use SSISDB
Go

/* Build custom Schema */
print 'Custom Schema'
/* Check for existence of custom Schema */
If Not Exists(Select name
              From sys.schemas
              Where name = 'custom')
begin
    /* Build and execute custom Schema SQL
       if it does not exist */
    print ' - Creating custom schema'
    declare @CustomSchemaSql varchar(32) = 'Create Schema custom'
    exec(@CustomSchemaSql)
    print ' - Custom schema created'
end
Else
    /* If the custom schema exists, tell us */
    print ' - Custom schema already exists.'
    print ''
Go

/* Build custom.Application table */
print 'Custom.Application Table'
/* Check for existence of custom.Application table */
If Not Exists(Select s.name + '.' + t.name
              From sys.tables t
              Join sys.schemas s
                On s.schema_id = t.schema_id
              Where s.name = 'custom'
              And t.name = 'Application')
begin
    /* Create custom.Application table
       if it does not exist */
    print ' - Creating custom.Application Table'
    Create Table custom.Application
    (
        ApplicationID int identity(1,1)
        Constraint PK_custom_Application Primary Key Clustered
        ,ApplicationName nvarchar(256) Not Null
        Constraint U_custom_ApplicationName Unique
        ,ApplicationDescription nvarchar(512) Null
    )

```

```

    print ' - Custom.Application Table created'
end
Else
    /* If the custom.Application table exists, tell us */
    print ' - Custom.Application Table already exists.'
    print ''

/* Build custom.Package table */
print 'Custom.Package Table'
/* Check for existence of custom.Package table */
If Not Exists(Select s.name + '.' + t.name
              From sys.tables t
                  Join sys.schemas s
                  On s.schema_id = t.schema_id
                  Where s.name = 'custom'
                  And t.name = 'Package')
begin
    /* Create custom.Package table
       if it does not exist */
    print ' - Creating custom.Package Table'
    Create Table custom.Package
    (
        PackageID int identity(1,1)
        Constraint PK_custom_Package Primary Key Clustered
        ,FolderName nvarchar(128) Not Null
        ,ProjectName nvarchar(128) Not Null
        ,PackageName nvarchar(256) Not Null
        ,PackageDescription nvarchar(512) Null
    )
    print ' - Custom.Package Table created'
end
Else
    /* If the custom.Package table exists, tell us */
    print ' - Custom.Package Table already exists.'
    print ''

/* Build custom.ApplicationPackage table */
print 'Custom.ApplicationPackage Table'
/* Check for existence of custom.ApplicationPackage table */
If Not Exists(Select s.name + '.' + t.name
              From sys.tables t
                  Join sys.schemas s
                  On s.schema_id = t.schema_id
                  Where s.name = 'custom'
                  And t.name = 'ApplicationPackage')
begin
    /* Create custom.ApplicationPackage table
       if it does not exist */
    print ' - Creating custom.ApplicationPackage Table'

```

```

Create Table custom.ApplicationPackage
(
    ApplicationPackageID int identity(1,1)
        Constraint PK_custom_ApplicationPackage Primary Key Clustered
    ,ApplicationID int Not Null
        Constraint FK_custom_ApplicationPackage_Application
            Foreign Key References custom.Application(ApplicationID)
    ,PackageID int Not Null
        Constraint FK_custom_ApplicationPackage_Package
            Foreign Key References custom.Package(PackageID)
    ,ExecutionOrder int Not Null
        Constraint DF_custom_ApplicationPackage_ExecutionOrder
            Default(10)
    ,ApplicationPackageEnabled bit Not Null
        Constraint DF_custom_ApplicationPackage_ApplicationPackageEnabled
            Default(1)
)
print ' - Custom.ApplicationPackage Table created'
end
Else
    /* If the custom.ApplicationPackage table exists, tell us */
    print ' - Custom.ApplicationPackage Table already exists.'
    print ''

    /* Build custom.GetApplicationPackages stored procedure */
    print 'Custom.GetApplicationPackages'
    /* Check for existence of custom.GetApplicationPackages stored procedure */
    If Exists(Select s.name + '.' + p.name
        From sys.procedures p
            Join sys.schemas s
                On s.schema_id = p.schema_id
        Where s.name = 'custom'
            And p.name = 'GetApplicationPackages')
    begin
        /* If custom.GetApplicationPackages stored procedure
            exists, drop it */
        print ' - Dropping custom.GetApplicationPackages Stored Procedure'
        Drop Procedure custom.GetApplicationPackages
        print ' - custom.GetApplicationPackages Stored Procedure dropped'
    end
    print ' - Creating custom.GetApplicationPackages Stored Procedure'
    go

    /*

        Procedure: custom.GetApplicationPackages
        Author: Andy Leonard
        Parameter(s): ApplicationName [nvarchar(256)]
            - contains the name of the SSIS Application
              for which to retrieve SSIS Packages.
    */

```

Description: Executes against the custom.ApplicationPackages table joined to the custom.Application and custom.Packages tables. Returns a list of enabled Packages related to the Application ordered by ExecutionOrder.

Example: exec custom.GetApplicationPackages 'TestSSISApp'

```

*/
Create Procedure custom.GetApplicationPackages
    @ApplicationName nvarchar(256)
As
begin

    Set NoCount On

    Select p.FolderName, p.ProjectName, p.PackageName, ap.ExecutionOrder
    From custom.ApplicationPackage ap
    Join custom.Package p
        On p.PackageID = ap.PackageID
    Join custom.Application a
        On a.ApplicationID = ap.ApplicationID
    Where a.ApplicationName = @ApplicationName
        And ap.ApplicationPackageEnabled = 1
    Order By ap.ExecutionOrder

end
go
print ' - Custom.GetApplicationPackages Stored Procedure created.'
print ''

/* Build custom.AddApplication stored procedure */
print 'Custom.AddApplication'
If Exists(Select s.name + '.' + p.name
    From sys.procedures p
    Join sys.schemas s
        On s.schema_id = p.schema_id
    Where s.name = 'custom'
        And p.name = 'AddApplication')
begin
    /* If custom.AddApplication stored procedure
    exists, drop it */
    print ' - Dropping custom.AddApplication Stored Procedure'
    Drop Procedure custom.AddApplication
    print ' - custom.AddApplication Stored Procedure dropped'
end
print ' - Creating custom.AddApplication Stored Procedure'
go

/*

```

```

Procedure: custom.AddApplication
Author: Andy Leonard
Parameter(s): ApplicationName [nvarchar(256)]
    - contains the name of the SSIS Application
      to add to the Framework database.
    ApplicationDescription [nvarchar(512)]
    - contains a description of the SSIS Application.
Description: Stores an SSIS Application.
Example: exec custom.AddApplication 'TestSSISApp', 'A test SSIS Application.'

*/
Create Procedure custom.AddApplication
    @ApplicationName nvarchar(256)
    ,@ApplicationDescription nvarchar(512) = NULL
As
begin

    Set NoCount On

    Insert Into custom.Application
    (ApplicationName
    ,ApplicationDescription)
    Output inserted.ApplicationID
    Values
    (@ApplicationName
    ,@ApplicationDescription)

end
go
print ' - Custom.AddApplication Stored Procedure created.'
print ''

/* Build custom.AddPackage stored procedure */
print 'Custom.AddPackage'
If Exists(Select s.name + '.' + p.name
    From sys.procedures p
    Join sys.schemas s
    On s.schema_id = p.schema_id
    Where s.name = 'custom'
    And p.name = 'AddPackage')
begin
    /* If custom.AddPackage stored procedure
    exists, drop it */
    print ' - Dropping custom.AddPackage Stored Procedure'
    Drop Procedure custom.AddPackage
    print ' - custom.AddPackage Stored Procedure dropped'
end
print ' - Creating custom.AddPackage Stored Procedure'
go

```

```

/*
    Procedure: custom.AddPackage
    Author: Andy Leonard
    Parameter(s): FolderName [nvarchar(128)]
        - contains the name of the SSISDB Catalog
          folder containing the SSIS Package.
          ProjectName [nvarchar(128)]
        - contains the name of the SSISDB Catalog
          project containing the SSIS Package.
          PackageName [nvarchar(128)]
        - contains the name of the SSISDB Catalog
          SSIS Package.
          PackageDescription [nvarchar(512)]
        - contains a description of the SSIS Package.
    Description: Stores an SSIS Package.
    Example: exec custom.AddPackage 'Chapter2', 'Chapter2'
            , 'Chapter2.dtsx', 'A test SSIS Package.'

*/
Create Procedure custom.AddPackage
    @FolderName nvarchar(128)
    ,@ProjectName nvarchar(128)
    ,@PackageName nvarchar(256)
    ,@PackageDescription nvarchar(512) = NULL
As
begin

    Set NoCount On

    Insert Into custom.Package
    (FolderName
    ,ProjectName
    ,PackageName
    ,PackageDescription)
    Output inserted.PackageID
    Values
    (@FolderName
    ,@ProjectName
    ,@PackageName
    ,@PackageDescription)

end
go
print ' - Custom.AddPackage Stored Procedure created.'
print ''

```

```

/* Build custom.AddApplicationPackage stored procedure */
print 'Custom.AddApplicationPackage'
If Exists(Select s.name + '.' + p.name
          From sys.procedures p
          Join sys.schemas s
            On s.schema_id = p.schema_id
          Where s.name = 'custom'
          And p.name = 'AddApplicationPackage')
begin
  /* If custom.AddApplicationPackage stored procedure
  exists, drop it */
  print ' - Dropping custom.AddApplicationPackage Stored Procedure'
  Drop Procedure custom.AddApplicationPackage
  print ' - custom.AddApplicationPackage Stored Procedure dropped'
end
print ' - Creating custom.AddApplicationPackage Stored Procedure'
go
/*

  Procedure: custom.AddApplicationPackage
  Author: Andy Leonard
  Parameter(s): ApplicationID [int]
    - contains the ID returned from the execution
      of custom.AddApplication.
    PackageID [int]
    - contains the ID returned from the execution
      of custom.AddPackage.
    ExecutionOrder [int]
    - contains the order the package will execute
      within the SSIS Application.
    ApplicationPackageEnabled [bit]
    - 1 == Enabled and will run as part of the SSIS Application.
    0 == Disabled and will not run as part of the SSIS Application.
  Description: Links an SSIS Package to an SSIS Application
  Example: exec custom.AddApplicationPackage 1, 1, 10, 1

*/
Create Procedure custom.AddApplicationPackage
  @ApplicationID int
  ,@PackageID int
  ,@ExecutionOrder int = 10
  ,@ApplicationPackageEnabled bit = 1
As
begin

  Set NoCount On

  Insert Into custom.ApplicationPackage
  (ApplicationID
  ,PackageID
  ,ExecutionOrder
  ,ApplicationPackageEnabled)

```

```

    Values
    (@ApplicationID
    ,@PackageID
    ,@ExecutionOrder
    ,@ApplicationPackageEnabled)

end
go
print ' - Custom.AddApplicationPackage Stored Procedure created.'
print ''

```

Create a new SSIS Package in the Chapter2 project and rename it **Parent.dtsx**. Click the Parameters tab on the package—it's the third tab from the left (Control Flow, Data Flow, Parameters). Click the Add Parameter button and create a parameter named *ApplicationName*, *String data type*, with a default value of testSSISApp. Set the Required property to True.

Add an Execute SQL task to the control flow and rename it **Get Packages**. Open the editor and set the ConnectionType property to ADO.NET. In the Connection property drop-down, select (or create a connection to) the SSISDB database. In the SQLStatement property, enter **custom.GetApplicationPackages**. Set the IsQueryStoredProcedure property to True. Change the ResultSet property to **Full result set**.

Navigate to the Parameter Mapping page and click the Add button. Click the Variable Name drop-down and select \$Package::ApplicationName at the very top of the list. Change the Data Type to String and the Parameter Name to **ApplicationName**. This maps the value in the parent package parameters into the ApplicationName parameter sent to the custom.GetApplicationPackages stored procedure when it is called by the Execute SQL task.

Navigate to the Result Set page and click the Add button. If the Add button is disabled, you did not change the ResultSet property on the General page from the default setting (None). If ResultSet is set to any other setting, the Add button is enabled. Enter **0** for the Result Name. In the Variable Name drop-down, create a variable named **Packages**. For this variable, set the Value Type property to Object.

■ **Note** Object is an interesting data type. Akin to a variant, Object can contain a scalar like a date or integer. It can also hold a collection or string array. In this example, Object will contain an ADO.Net Dataset value. If we had set the ConnectionType property to OLEDB (the default), this result set variable would be populated with an ADO Recordset. Yes, that is a COM object—in 2014. COM (and COBOL) will never die....

Let's review. First, the task will use an ADO.NET connection to the SSISDB database to execute the custom.GetApplicationPackages stored procedure we created earlier. Because we set the IsQueryStoredProcedure to True, we do not need to add placeholders for parameters or the exec command. Since we used ADO.NET, we can address parameters by name instead of ordinal (*ApplicationName* instead of 0) on the Parameter Mapping page. Finally, we configured the Execute SQL task to push the results of the stored procedure execution into an object variable named Packages.

Click the OK button to close the Execute SQL Task Editor. Drag a Foreach Loop container onto the control flow surface and open its editor. On the General page, change the Name property to **Foreach Package in Packages**. On the Collection page, select the Foreach ADO Enumerator. In the ADO Object Source Variable drop-down, select the Packages variable. Leave the Enumeration Mode default option Rows in the First Table selected.

I can hear you thinking, "So what would I need to do if I had an ADO Recordset in the Packages object variable?" That is an excellent question. The answer is, "Nothing different." Even though object variables can hold ADO Recordsets and ADO.NET datasets (and other collections and scalars), the Foreach ADO Enumerator is smart enough to *detect* the type of object inside the SSIS object variable— and then read it. Isn't that cool? I thought so too.

Navigate to the Variable Mappings page. Create four variables at package scope. These variables match the fields returned from the custom.GetApplicationPackages stored procedure; and subsequently loaded into the first table in an ADO.NET dataset now housed inside the Packages SSIS variable. If you didn't get that sentence, reread it (I'll wait). That's a lot to take in, but it is vital to understanding what we're doing here. Got it? Good.

I will walk you through creating the first variable listed as follows using the method I prefer for variable creation. Click the Variable drop-down and select <New variable...> at the very top of the list. When the Add Variable window displays, make sure the Container property is set to Parent (the name of the package). This ensures the variable has package scope. Enter **FolderName** in the Name text box. Click the OK button and change the Index property to **0**. Leave the Value Type property set to String. I almost always create SSIS variables in this fashion. I have more control over scope, and I am creating and configuring the variable where it will be used. This functionality saves time and simply rocks.

Create the variables in the following order:

```
Container: Parent
Name: FolderName
Namespace: User
Value Type: String
Value:
```

```
Container: Parent
Name: ProjectName
Namespace: User
Value Type: String
Value:
```

```
Container: Parent
Name: ChildPackageName
Namespace: User
Value Type: String
Value:
```

```
Container: Parent
Name: ExecutionOrder
Namespace: User
Value Type: Int32
Value: 0
```

Make sure the index values align as shown here:

```
FolderName: 0
ProjectName: 1
ChildPackageName: 2
ExecutionOrder: 3
```

The fields do not *have* to be listed in this order, but the index values have to align with the (zero-based) ordinal value of the fields returned by the custom.GetApplicationPackages.

Click the OK button to close the Foreach Loop Container Editor. Drag an Execute SQL task into the Foreach Loop container and rename it **Execute Package**. Set the ConnectionType to ADO.NET and select the SSISDB connection you created earlier. Set the IsQueryStoredProcedure property to True and the SQL Statement property to custom.execute_catalog_package. On the Parameter Mapping page, add and create a new variable named ExecutionID, Int32 data type, package scope, default value: 0. Change the Direction of the parameter to Output

and couple the SSIS variable you just created to the ExecutionID parameter by supplying the Parameter Name: ExecutionID. Add three more parameter mappings—one each for FolderName, ProjectName, and ChildPackageName. Map them to the stored procedure parameters FolderName, ProjectName, and PackageName; respectively. The custom.execute_catalog_package stored procedure accepts other parameters: LoggingLevel, Use32BitRunTime, ReferenceID, and ObjectType; but these parameters all contain default values that will serve our purposes. Click the OK button to close the Execute SQL Task Editor.

Your Parent.dtsx SSIS package should appear as shown in Figure 2-3.

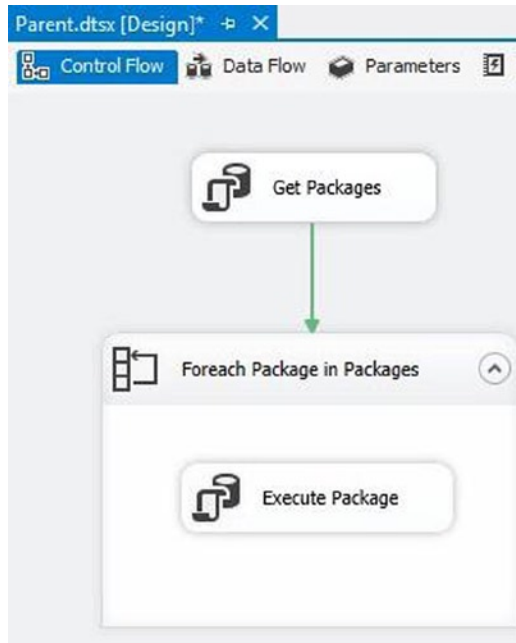


Figure 2-3. Parent package control flow

Return to SSMS. Let's provide our simple execution framework with metadata to execute. Execute the T-SQL statements in Listing 2-8.

Listing 2-8. Building the Metadata for an SSIS Application

Use SSISDB

Go

Set NoCount On

```

Declare @ApplicationName nvarchar(256)
Declare @ApplicationDescription nvarchar(512)
Declare @ApplicationID int
Declare @FolderName nvarchar(256)
Declare @ProjectName nvarchar(256)
Declare @PackageName nvarchar(256)
Declare @PackageDescription nvarchar(512)
Declare @PackageID int
Declare @ExecutionOrder int
  
```

```

Declare @ApplicationPackageEnabled bit
Declare @ApplicationTbl table(ApplicationID int)
Declare @PackageTbl table(PackageID int)

begin tran

    -- Build Application --
    Select @ApplicationName = 'TestSSISApp'
           ,@ApplicationDescription = 'A test SSIS application'

    Insert Into @ApplicationTbl
    Exec custom.AddApplication
           @ApplicationName
           ,@ApplicationDescription

    Select @ApplicationID = ApplicationID
    From @ApplicationTbl

    -- Build Package --
    Select @FolderName = 'Chapter2'
           ,@ProjectName = 'Chapter2'
           ,@PackageName = 'Chapter2.dtsx'
           ,@PackageDescription = 'A test SSIS package'

    Insert Into @PackageTbl
    Exec custom.AddPackage
           @FolderName
           ,@ProjectName
           ,@PackageName
           ,@PackageDescription

    Select @PackageID = PackageID
    From @PackageTbl

    -- Build ApplicationPackage --
    Select @ExecutionOrder = 10
           ,@ApplicationPackageEnabled = 1

    Exec custom.AddApplicationPackage
           @ApplicationID
           ,@PackageID
           ,@ExecutionOrder
           ,@ApplicationPackageEnabled

    Delete @PackageTbl

    -- Build Package --
    Select @FolderName = 'Chapter2'
           ,@ProjectName = 'Chapter2'
           ,@PackageName = 'Chapter2.dtsx'
           ,@PackageDescription = 'Another test SSIS package'

```

```

Insert Into @PackageTbl
Exec custom.AddPackage
    @FolderName
    ,@ProjectName
    ,@PackageName
    ,@PackageDescription

Select @PackageID = PackageID
From @PackageTbl

-- Build ApplicationPackage --
Select @ExecutionOrder = 20
    ,@ApplicationPackageEnabled = 1

Exec custom.AddApplicationPackage
    @ApplicationID
    ,@PackageID
    ,@ExecutionOrder
    ,@ApplicationPackageEnabled

Delete @PackageTbl

Commit

```

The T-SQL in Listing 2-8 builds a simple SSIS application in the execution framework. It calls our `Chapter2.dtsx` SSIS package twice. If you return to SSDT-BI and execute the parent package, you will note the `Chapter2.dtsx` SSIS package executes twice in quick succession. You can see that execution in Figure 2-4.

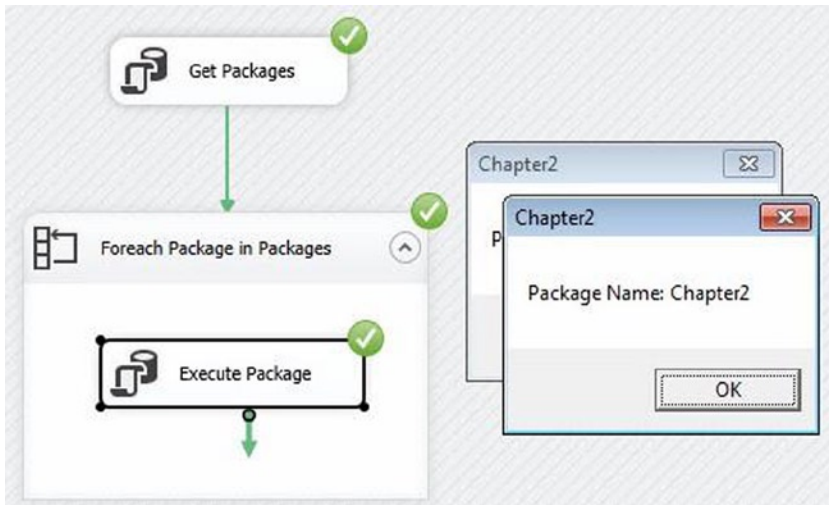


Figure 2-4. *Chapter2.dtsx* executing twice without waiting

It is important to understand that the framework is a “fire and forget” design. The screenshot in Figure 2-4 shows both instances of `Chapter2.dtsx` showing their respective message boxes, yet the tasks in the background have completed. This approach works well if your SSIS packages can be executed in parallel. But what if there are dependencies between packages? This framework does not facilitate dependent package execution, but I will show you a way to couple the framework with the SQL Server Agent Job scheduler in the next section. Coupling will allow you to execute the parent package for each “step” of a process, calling an SSIS application each step, and in turn, calling one or more SSIS packages in parallel.

■ **Note** Appendix A contains information on building a serial SSIS framework that was originally built for SSIS 2005. It works in SSIS 2014 if you use the package deployment model.

Scheduling SSIS Package Execution

There are many commercially available software execution schedulers on the market. They range from relatively simple to highly complex, allowing time- or event-based execution. Many include metadata collection capabilities that track metrics such as execution time. SQL Server Agent is a fairly robust job scheduling application included with SQL Server. We will use SQL Server Agent to schedule the execution of our demo package.

■ **Caution** Before proceeding, deploy the `Chapter2` project to the SSIS Catalog. Doing so will deploy `Chapter2.dtsx` and `Parent.dtsx`.

Scheduling an SSIS Package

Open SSMS and connect to an instance of SQL Server 2014. Open Object Explorer and expand the SQL Server Agent node, if possible. Why might expanding the SQL Server Agent node not be possible? By default, SQL Server Agent is installed as a manual startup service.

Right-click the Jobs virtual folder and click **New ► Job**. When the New Job window displays, name the job **Ch2**. Click on the Steps page and click the **New** button. Name the new step **Execute Chapter 2 Package** and select SQL Server Integration Services Package from the Type drop-down.

Select a Package Source from the drop-down. These are the options:

- SQL Server
- File system
- SSIS Package Store
- SSIS Catalog

Let's schedule an SSIS package from the catalog to start. Type **localhost**, or the name of the SSIS server that contains the SSIS Catalog, into the Server drop-down. Click the ellipsis beside the Package text box and navigate to the demonstration package, `Chapter2.dtsx`, as shown in Figure 2-5.

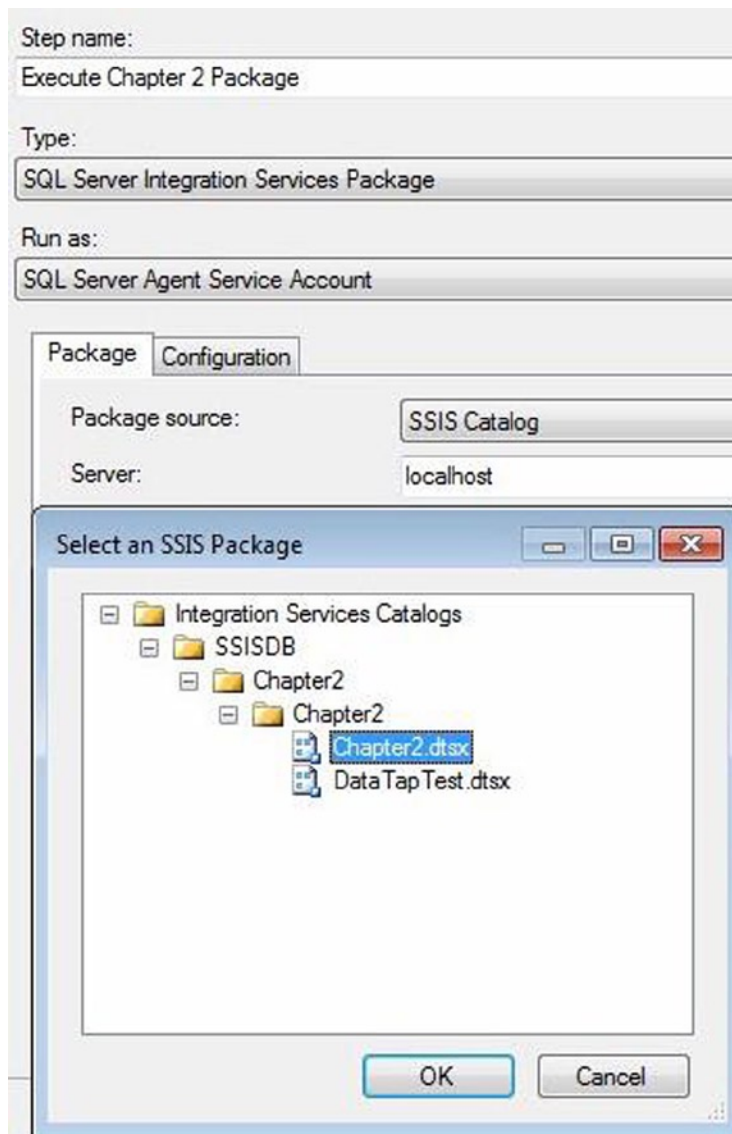


Figure 2-5. Configuring a SQL Server Agent Job to execute an SSIS package in the SSIS Catalog

Clicking the OK button will complete the package selection procedure.

Scheduling a File System Package

To schedule a package stored in the file system, select File System in the Package Source drop-down. Click the ellipsis beside the Package text box and navigate to the desired SSIS package file. Once configured, the step will appear as shown in Figure 2-6.

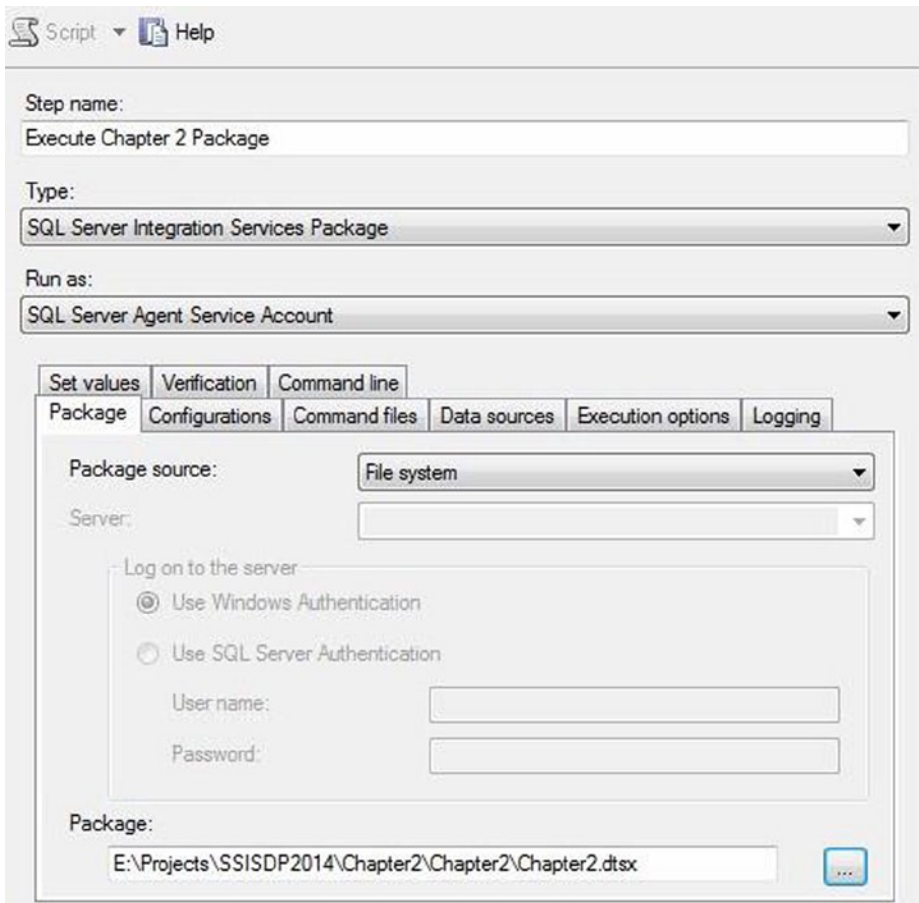


Figure 2-6. SQL Server Agent job step configured to execute an SSIS package from the file system

You can test the job by right-clicking the job name in SSMS and then clicking Start Job at Step. But the job execution may fail. Why? Because the `Chapter2.dtsx` package displays a message box, SQL Server Agent jobs are usually started by a service account, and service accounts are not allowed to display message boxes. If your SQL Server Agent service is started by a User account (or any account acting with the `InteractWithDesktop` role). This is discussed more in the next section.

Running SQL Server Agent Jobs with the Custom Execution Framework

We can run a SQL Server Agent job with our custom execution framework. To demonstrate, create a new SQL Server Agent job named **Framework Execution**. On the Steps page, add a new step named **TestSSISApp Framework Execution**. Select the SSIS Package Step Type and accept the default Package Source property of SSIS Catalog. Enter or select the name of your server in the Server drop-down and click the ellipsis beside the Package text box to open the Select an SSIS Package window. Navigate to the `Parent.dtsx` SSIS package, and then click the OK button.

Click the Configuration tab on the New Job Step window. The package parameter `ApplicationName` should appear in the Parameters list. To enter a value for this parameter, click the ellipsis beside the Value text box, and then enter **TestSSISApp**. Click the OK button to close and save the New Job Step window, and then click the OK button again to close and save the New Job window.

To test, right-click the framework execution SQL Server Agent job and click Start Job at Step. The SQL Server Agent job will execute and succeed, but I have bad news: the package executions will fail. I can hear you thinking, “Wait, what?” I kid you not. Remember about this being a “fire and forget” execution framework? That fact haunts us here—and elsewhere in SSIS execution. It’s better for you to become aware of this now—trust me on this. The other way to gain this knowledge involves arguing with your boss (or worse, your client) that “the job succeeded!” and being wrong.

How do you know the package execution failed? Let’s go look. Expand the Integration Services Catalog’s virtual folder in SSMS Object Explorer. Right-click SSISDB and hover over Reports, then Standard Reports, and click Integration Services Dashboard. If you have followed my instructions, you will see a large, reddish-colored “2” above the word Failed. If you click the “2,” the reports will take you to a page containing a list of failed executions. If you then click the All Messages link, you will see an error message informing you that the Script task experienced an error (Exception has been thrown by the target of an invocation). That message means (among other things) that you used a message box in a Script task. No, I am not making this up.

■ **Note** “Are message boxes bad?” Absolutely not! In fact, they’re the only way to troubleshoot a certain class of errors in SSIS. I use them all the time, but I qualify the message box calls in an If/Then statement. If you don’t do this, the message box calls will execute and cause SQL Server Agent jobs to either fail or lie to you about execution success.

All is not lost. The problem here is that a service account is providing the security context for the execution. The account used to start the SQL Server Agent service is the account used to execute the packages from SQL Server Agent jobs. That account typically does not have the `InteractWithDesktop` role assigned, and you have to admit—a desktop is handy for displaying message boxes. The caveat is this: you cannot include unqualified calls to message box displays in SSIS packages. Use a parameter or variable (I use one called `Debug`) and make sure its value is external to the package so you can turn it on and off when you *want* to display message boxes.

You can also execute the `Parent.dtsx` package from the SSIS Catalog. In SSMS Object Explorer, continue drilling into the `Chapter2` folder. Open Projects, then `Chapter2`, then Packages, and right-click the `Parent.dtsx` package. Click Execute and supply **TestSSISApp** for the `ApplicationName` parameter. When you click the OK button, the package executes and the two message boxes appear. Why? Because you are no longer running the security context of the service account that starts the SQL Server Agent service; you are running in the security context with which you connected to SSMS Object Explorer. This is most likely a domain or machine account that uses Windows Authentication and *your* personal credentials. If you’ve been watching a desktop all this time, you (and all the other users in your domain or machine) have the `InteractWithDesktop` role assigned. But almost all service accounts do not participate in the `InteractWithDesktop` role.

Running the Custom Execution Framework with SQL Server Agent

You can run SQL Server Agent jobs with the custom execution framework. You just cannot pop up message boxes. For example, you can create an SSIS application for each “step” in your process. The SSIS application can contain SSIS packages that can execute in parallel. You then build a SQL Server Agent job with several job steps—one for each SSIS application. A SQL Server Agent job executes its steps serially, waiting for one to succeed (by default) before starting the next.

Most data warehouses require an extraction step that stages all data—from dimension and fact sources—to a staging database. Next, dimension data are loaded into the data warehouse. Finally, fact data are loaded from the staging database into the data warehouse.

The precedence of operations is as follows: extract fact and extract dimensions can run concurrently (in parallel). You can design one package for each dimension and fact source table extract operation, add them to the extract SSIS application, and execute that SSIS application as Step 1 of your DW ETL job. Once that completes, Step 2 can load the dimension data from the stage database to the data warehouse. Once that completes, Step 3 can load the fact data from the staging database to the data warehouse. So while relatively simple and somewhat limited, our custom execution framework can facilitate configurable parallel and serial ETL operations.

Execute Package Task

The Execute Package task is best understood in action. To demonstrate, create a new SSIS package and rename it **Parent2.dtsx**. Add an Execute Package task to the control flow. Open the editor and observe the selections for the ReferenceType property as shown in Figure 2-7.

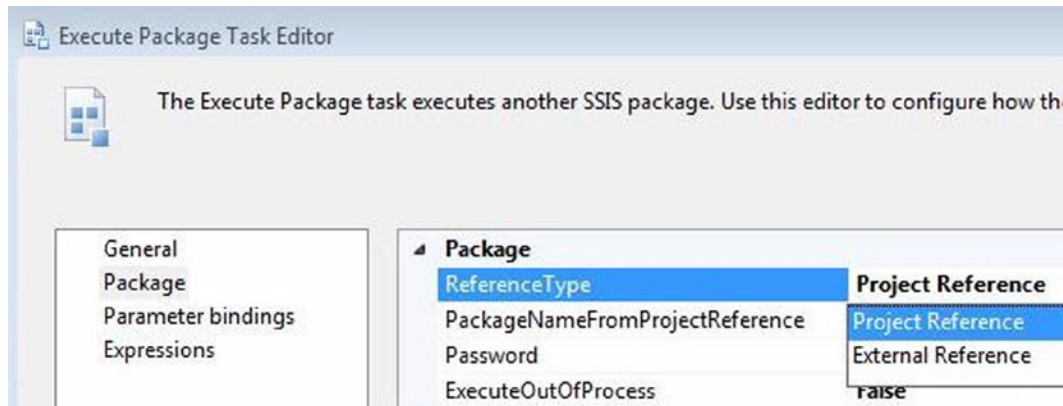


Figure 2-7. The Execute Package Task Reference Property

If the ReferenceType package is set to Project Reference, the Execute Package task can be used to start packages in the SSIS project, supporting the project deployment model. Setting this property to External Reference allows executing SSIS packages to be stored in either the MSDB database or file system, supporting the package deployment model. Figure 2-8 shows the Execute Package task configured to execute the Chapter2.dtsx package.

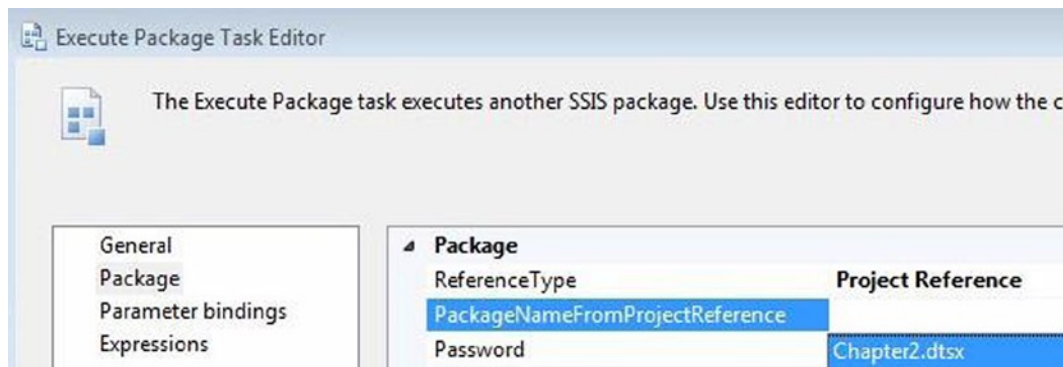


Figure 2-8. Selecting the Project Reference Package

You can close the editor after selecting the package. Test it by executing Parent2.dtsx in the SSIS debugger.

METADATA-DRIVEN EXECUTION

I have used all of the methods for SSIS package execution listed in this chapter. They each have advantages and disadvantages. How then, do you select which method to use? That is an excellent question and I'm glad you asked! I consider the following:

- **Troubleshooting:** At some time in the future, someone will have to figure out why a package failed to execute. Facilitating troubleshooting is not something to be tacked onto the end of a data integration development project; you need to consider it up front. It is as important as security. I select an SSIS package execution method that supports troubleshooting in each enterprise.
- **Code maintenance:** The SSIS project will possibly be modified in the future. This means the packages, projects, and execution methodology need to be documented. It also means I need to consider the skills and comfort levels of the individuals or team maintaining this code. If the individuals or team are skilled .NET developers, I lean toward using the Script task and component for complex operations. I also attempt to develop in the .NET language of their choice, if this is the case. If they found their way to SSIS via a role as a database administrator, I use more T-SQL in Execute SQL tasks and OLE DB Source adapters when developing the solution. If they have DTS, SSIS, or other ETL development platform experience, I develop packages slightly differently to match their comfort zones. Again, this is different for different enterprises.
- **Enterprise requirements:** I often encounter “best practices” at enterprises. I enclose the terms in quotations because, well, some of them aren't actually best. They exist because something bad happened and someone reacted. Sometimes the reactions make sense from an SSIS point of view, sometimes they are security matters that vex the SSIS developers, and sometimes they just do not make good sense for anyone.
- **Complexity:** I do not like complex solutions. I tolerate them if they are the only way to accomplish what needs to be done, but I strive to keep solutions as simple as possible. Fewer moving parts means there is less to break, less to troubleshoot, and less to maintain. That said, flexibility and complexity are often proportional. That means highly flexible solutions are likely to be complex solutions.

I write this here, especially the bullet about complexity, to introduce execution from managed code. Complexity is the only disadvantage of executing SSIS from .NET. Executing SSIS from managed code offers maximum flexibility: If you can think it, you can find a way to build it in .NET. In my opinion, knowing a .NET language is no longer optional for the data integration developer in the Microsoft space.

Execution from Managed Code

There is a ton (or tonne, if you prefer) of benefit from executing SSIS packages from .NET managed code. There are various limitations to executing SSIS in other ways. Without exception, they can all be overcome by controlling execution from .NET. In this section, we will demonstrate the basics of using VB.NET to execute SSIS packages.

The Demo Application

For this demonstration, I used Visual Basic 2013 and the .NET Framework 4.5.1. I downloaded a copy from www.microsoft.com/en-us/download/details.aspx?id=40787. Unless otherwise specified, I accepted the default settings for VB applications in the Visual Studio 2013 Integrated Development Environment (IDE).

To begin, create a new VB Windows Forms project in Visual Studio 2013. Add references for the following assemblies:

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

You will have to search for `Microsoft.SqlServer.Management.IntegrationServices` and `Microsoft.SqlServer.SMO` in the global assembly cache (GAC). The GAC is located in the `Windows\Assembly` folder, and these libraries are in the MSIL folder. Click Browse when adding references to the Visual Studio project.

The frmMain Form

Rename `Form1` to **frmMain**. Add two `GroupBox` controls to the form, arranged with one over the other as shown in Figure 2-9. Change the `Text` property of the top group box to `SSIS Package in the File System` and the `Text` property of the lower group box to `SSIS Package in the Catalog`. In the upper group box, add a label, text box, and two buttons. Change the `Text` property of the label to `Package Path`. Name one of the buttons **btnOpenSSISpkg** and change its `Text` property to `“...”`. Name the other button **btnStartFile** and set its `Text` property to **Start**.

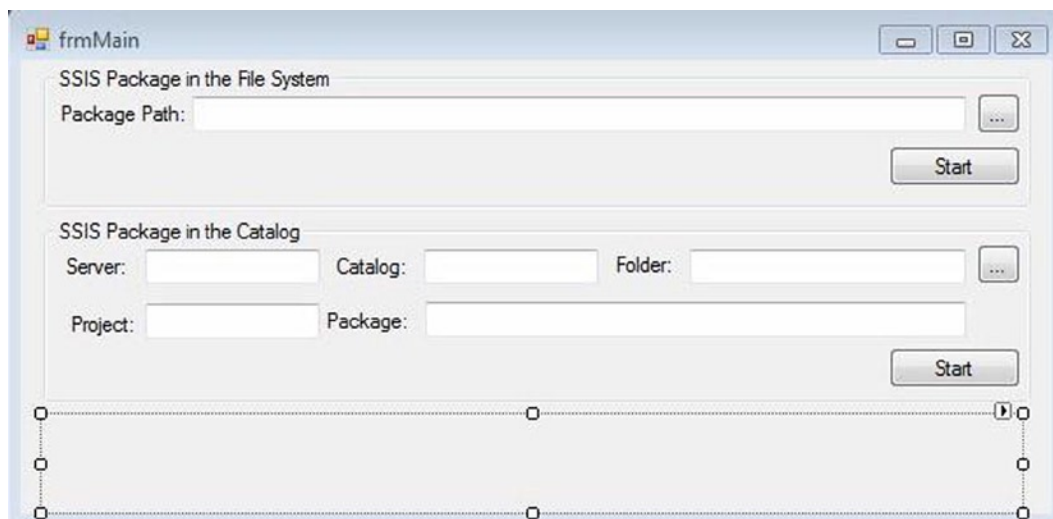


Figure 2-9. The `frmMain` controls layout

In the `SSIS Package in the Catalog` group box, add five labels, five textboxes, and two buttons. Change the `Text` properties of the labels to **Server:**, **Catalog:**, **Folder:**, **Project:**, and **Package:**. Position each text box to the right of each labels and name them **txtSSISCatalogServer**, **txtCatalog**, **txtFolder**, **txtCatalogProject**, and **txtCatalogPackage**, respectively. Name one of the buttons **btnOpenSSISPkgInCatalog** and set its `Text` property to `“...”`. Name the other button **btnStartCatalog** and set its `Text` property to **Start**.

Add a text box beneath the SSIS Package in the Catalog group box. Name it **txtStatus**, set the **MultiLine** property to **True**, **BackColor** to **ButtonFace**, and the **BorderStyle** to **None**. Position the controls similar to the way shown in Figure 2-9. Finally, add a **FileOpenDialog** control to the form, leaving it configured to defaults.

It will likely surprise no one to learn that I was first exposed to design patterns while a software developer. The pattern I use in this application puts a minimum amount of code behind the form. The code that is in the form calls code in a form-specific module. You can view the code for **frmMain** by right-clicking **frmMain** in Solution Explorer and selecting **View Code**. Replace the code displayed with the following:

```
'
' frmMain code
'
' I use a Helper Pattern when developing interfaces.
' Each form is named frm_____ and there is a corresponding module named frm_____Helper.vb.
' Each event method calls a subroutine in the Helper module.
'
Public Class frmMain

    Private Sub frmMain_Load(ByVal sender As System.Object,
                             ByVal e As System.EventArgs) _ Handles MyBase.Load
        InitFrmMain()
    End Sub

    Private Sub btnStartFile_Click(ByVal sender As System.Object, _
                                   ByVal e As System.EventArgs) Handles btnStartFile.Click
        btnStartFileClick()
    End Sub

    Private Sub btnOpenSSISpkg_Click(ByVal sender As System.Object, _
                                      ByVal e As System.EventArgs) Handles _ btnOpenSSISpkg.Click
        btnOpenSSISpkgClick()
    End Sub

    Private Sub btnStartCatalog_Click(ByVal sender As System.Object, _
                                       ByVal e As System.EventArgs) Handles _ btnStartCatalog.Click
        btnStartCatalogClick()
    End Sub

    Private Sub btnOpenSSISpkgInCatalog_Click(ByVal sender As System.Object, _
                                                ByVal e As System.EventArgs)
        Handles _ btnOpenSSISpkgInCatalog.Click
        btnOpenSSISpkgInCatalogClick()
    End Sub
End Class
```

Again, the code behind the form is sparse. Most of the real work is done elsewhere. Let's build that part now.

Add a module to the solution and name it **frmMainHelper**. Add the following code to the new module:

```
'
' frmMainHelper module
'
' I use a Helper Pattern when developing interfaces.
' Each form is named frm_____ and there is a corresponding module named frm_____Helper.vb.
' Each event method calls a subroutine in the Helper module.
'
' This module supports frmMain.
'
Imports System
Imports System.Windows
Imports System.Windows.Forms
Imports Microsoft.SqlServer.Dts.Runtime Wrapper
Imports Microsoft.SqlServer.Management.IntegrationServices
Imports Microsoft.SqlServer.Management.Smo

Module frmMainHelper

    Public Sub InitFrmMain()

        ' initialize and load frmISTree

        ' define the version
        Dim sVer As String = System.Windows.Forms.Application.ProductName & " v" & _
            System.Windows.Forms.Application.ProductVersion

        ' display the version and startup status
        With frmMain
            .Text = sVer
            .txtStatus.Text = sVer & ControlChars.CrLf & "Ready"
        End With

    End Sub

    Public Sub btnStartFileClick()

        ' configure an SSIS application and execute the selected SSIS package file

        With frmMain
            .Cursor = Cursors.WaitCursor
            .txtStatus.Text = "Executing " & .txtSSISPath.Text
            .Refresh()
            Dim ssisApp As New Microsoft.SqlServer.Dts.Runtime Wrapper.Application
            Dim ssisPkg As Package = ssisApp.LoadPackage(.txtSSISPath.Text, _
                AcceptRejectRule.None, Nothing)
            ssisPkg.Execute()
            .Cursor = Cursors.Default
            .txtStatus.Text = .txtSSISPath.Text & " executed."
        End With

    End Sub

End Sub
```

```

Public Sub btnOpenSSISpkgClick()

    ' allow the user to navigate to an SSIS package file

    With frmMain
        .OpenFileDialog1.DefaultExt = "dtsx"
        .OpenFileDialog1.ShowDialog()
        .txtSSISpkgPath.Text = .OpenFileDialog1.FileName
        .txtStatus.Text = .txtSSISpkgPath.Text & " package path loaded."
    End With

End Sub

Sub btnOpenSSISpkgInCatalogClick()

    ' allow the user to navigate to an SSIS package stored in a catalog

    frmISTreeInit()

    Dim sTmp As String = sFullSSISpkgPath
    Dim sServerName As String = Strings.Left(sTmp, Strings.InStr(sTmp, ".") - 1)
    Dim iStart As Integer = Strings.InStr(sTmp, ".") + 1
    Dim iEnd As Integer = Strings.InStr(sTmp, "\")
    Dim iLen As Integer
    Dim sCatalogName As String
    Dim sFolderName As String
    Dim sProjectName As String
    Dim sPackageName As String

    If iEnd > iStart Then
        iLen = iEnd - iStart
        sCatalogName = Strings.Mid(sTmp, iStart, iLen)
        sTmp = Strings.Right(sTmp, Strings.Len(sTmp) - iEnd)

        iStart = 1
        iEnd = Strings.InStr(sTmp, "\")
        If iEnd > iStart Then
            iLen = iEnd - iStart
            sFolderName = Strings.Mid(sTmp, iStart, iLen)
            sTmp = Strings.Right(sTmp, Strings.Len(sTmp) - iEnd)

            iStart = 1
            iEnd = Strings.InStr(sTmp, "\")
            If iEnd > iStart Then
                iLen = iEnd - iStart
                sProjectName = Strings.Mid(sTmp, iStart, iLen)
                sTmp = Strings.Right(sTmp, Strings.Len(sTmp) - iEnd)
                sPackageName = sTmp
            End If
        End If
    End If
End Sub

```

```

With frmMain
    .txtSSISCatalogServer.Text = sServerName
    .txtCatalog.Text = sCatalogName
    .txtFolder.Text = sFolderName
    .txtCatalogProject.Text = sProjectName
    .txtCatalogPackage.Text = sPackageName
    .txtStatus.Text = sFullSSISPkgPath & " metadata loaded and parsed."
End With

End Sub

Sub btnStartCatalogClick()

    ' configure an SSIS application and execute the selected SSIS package from the
    ' catalog

    With frmMain
        .Cursor = Cursors.WaitCursor
        .txtStatus.Text = "Loading " & sFullSSISPkgPath
        .Refresh()
        Dim oServer As New Server(.txtSSISCatalogServer.Text)
        Dim oIS As New IntegrationServices(oServer)
        Dim cat As Catalog = oIS.Catalogs(.txtCatalog.Text)
        Dim fldr As CatalogFolder = cat.Folders(.txtFolder.Text)
        Dim prj As ProjectInfo = fldr.Projects(.txtCatalogProject.Text)
        Dim pkg As PackageInfo = prj.Packages(.txtCatalogPackage.Text)
        .txtStatus.Text = sFullSSISPkgPath & " loaded. Starting validation..."
        .Refresh()
        pkg.Validate(False, PackageInfo.ReferenceUsage.UseAllReferences, Nothing)
        .txtStatus.Text = sFullSSISPkgPath & " validated. Starting execution..."
        .Refresh()
        pkg.Execute(False, Nothing)
        .txtStatus.Text = sFullSSISPkgPath & " execution started."
        .Cursor = Cursors.Default
    End With

End Sub

End Module

```

Let's walk through the portion of this code that executes an SSIS package in the file system. In Figure 2-9, we are looking at the functionality represented in the upper group box.

The application works when the user either enters the full path to an SSIS package in the file system or clicks the ellipsis to browse to a SSIS package (dtsx) file. After selecting a file, the full path will display in the Package Path text box. To execute the package, click the Start button in the SSIS Package in the File System group box. When the Start button is clicked, the form code in the `btnStartFile_Click` subroutine is executed, and it executes a single line of code that calls the `btnStartFileClick` subroutine in the `frmMainHelper` module.

The `btnStartFileClick` subroutine first changes the form cursor to a `WaitCursor`. Next it updates the `txtStatus` text box to display the text `Executing` followed by the full path of the SSIS package in the Package Path text box. The `Refresh` statement causes the form to update, displaying the `WaitCursor` and the message in `txtStatus`. The code then creates an instance of an SSIS application (`Microsoft.SqlServer.Dts.Runtime Wrapper.Application`) in the form

of the `ssisApp` variable. `ssisPkg` is an instance of an SSIS Package object. It is created by calling the `LoadPackage` method of the SSIS Application object (`ssisApp`). We use the Package object's `Execute` method to start the SSIS package. The remainder of the subroutine resets the form cursor and updates the `txtStatus` message to indicate the package executed.

Were I to harden this code for production, I would wrap much of the code in this subroutine in a large Try-Catch block. In the Catch section, I would reset the cursor and update `txtStatus` with the error message. I like logging—a lot. In a Production-hardened version, I would log my intention to execute the package and include the full path displayed in the Package Path text box. I would also log the result of the attempted execution, whether it succeeded or failed.

The code that executes an SSIS package stored in the SSIS Catalog is found in the `frmMainHelper` module's `btnStartCatalogClick` subroutine. The code that manages the cursor and messaging to the `txtStatus` text box is comparable to that found in the `btnStartFileClick` subroutine.

There are a few more moving parts to an SSIS package stored in the SSIS Catalog, shown in Figure 2-10.

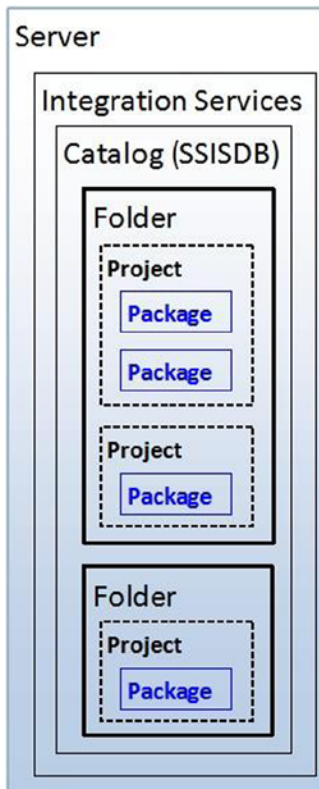


Figure 2-10. A representation of the SSIS Catalog

Integration Services is contained by a server and, in turn, it contains a catalog. In SQL Server 2014, Integration Services contains a single catalog named SSISDB. SSISDB is also the name of the database used to manage SSIS metadata in the catalog. A catalog contains one or more folders. Folders contain one or more projects, which contain one or more packages.

In the `btnStartCatalogClick` subroutine, the code declares variables for the objects in this hierarchy (using `Dim` statements) and sets their value based on the names supplied in the five text boxes: `txtSSISCatalogServer`, `txtCatalog`, `txtFolder`, `txtCatalogProject`, and `txtCatalogPackage`. As you can see by comparing the names of the text boxes to Figure 2-10, an SSIS package stored in the catalog can be uniquely identified using this hierarchy. How are the text

boxes populated? The user can enter the information manually if desired. But the application contains a second form, launched by the ellipsis in the SSIS Package in the Catalog group box, to facilitate SSIS catalog navigation.

To build it, add a second form to the application and name it **frmISTree**. Add a GroupBox control to the form and position it near the top. Change the Text property of the group box to **Connection**. Add a label, text box, and button to the group box. Change the Text property of the label to **Server:**. Name the text box **txtServer**. Name the button **btnConnect** and change its Text property to **Connect**. Add a TreeView control to the lower portion of the form and name it **tvCatalog**. Add a button just below the treeview, name it **btnSelect**, and change its Text property to **Select**. Add an ImageList control and name it **ilSSDB**. You will either have to rustle up your own images or download the demo project containing the four images I used for treeview node levels. Set the treeview's ImageList property to **ilSSDB**. The form should appear as shown in Figure 2-11.

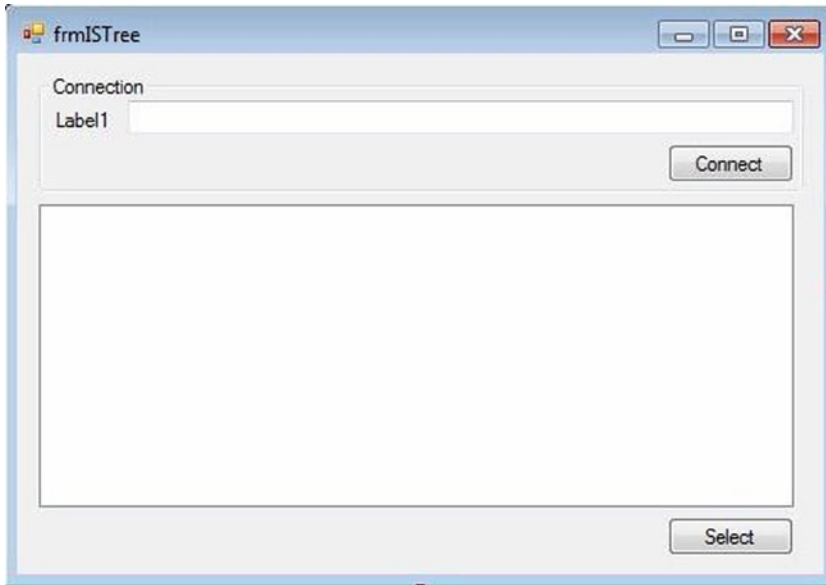


Figure 2-11. *ISTree Form*

Replace the code behind the form with the following code:

```
'
' frmISTree code
'
' I use a Helper Pattern when developing interfaces.
' Each form is named frm_____ and there is a corresponding module named frm_____Helper.vb.
' Each event method calls a subroutine in the Helper module.
'
Public Class frmISTree

    Private Sub btnConnect_Click(ByVal sender As System.Object, _
                                ByVal e As System.EventArgs) Handles btnConnect.Click
        btnConnectClick()
    End Sub
```

```

Private Sub btnSelect_Click(ByVal sender As System.Object, _
                             ByVal e As System.EventArgs) Handles btnSelect.Click
    btnSelectClick()
End Sub

Private Sub tvCatalog_AfterSelect(ByVal sender As System.Object, _
                                   ByVal e As System.Windows.Forms.TreeViewEventArgs) _
Handles tvCatalog.AfterSelect

    End Sub

Private Sub tvCatalog_DoubleClick(ByVal sender As Object, _
                                   ByVal e As System.EventArgs) _
Handles tvCatalog.DoubleClick
    tvCatalogDoubleClick()
End Sub
End Class

```

Again, this code merely points to the Helper module, in this case `frmISTreeHelper`. Add a new module, so named, and replace it with the following code:

```

'
' frmISTreeHelper module
'
' I use a Helper Pattern when developing interfaces.
' Each form is named frm_____ and there is a corresponding module named frm_____Helper.vb.
' Each event method calls a subroutine in the Helper module.
'
' This module supports frmISTree.
'

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

Module frmISTreeHelper

    ' variables
    Public sFullSSISPkgPath As String

    Sub frmISTreeInit()

        ' initialize and load frmISTree

        With frmISTree
            .Text = "Integration Services"
            .txtServer.Text = "localhost"
            .ShowDialog()
        End With

    End Sub

```

```

Sub btnConnectClick()

    ' connect to the server indicated in the txtServer textbox
    ' hook into the SSISDB catalog
    ' build out the SSISDB node by iterating the objects stored therein
    ' load the node and display it

    With frmISTree
        Dim oServer As New Server(.txtServer.Text)
        Dim oIS As New IntegrationServices(oServer)
        Dim cat As Catalog = oIS.Catalogs("SSISDB")
        Dim L1Node As New TreeNode("SSISDB")
        L1Node.ImageIndex = 0
        Dim L2Node As TreeNode
        Dim L3Node As TreeNode
        Dim L4Node As TreeNode

        For Each f As CatalogFolder In cat.Folders
            L2Node = New TreeNode(f.Name)
            L2Node.ImageIndex = 1
            L1Node.Nodes.Add(L2Node)
            '.tvCatalog.Nodes.Add(L2Node)
            For Each pr As ProjectInfo In f.Projects
                L3Node = New TreeNode(pr.Name)
                L3Node.ImageIndex = 2
                L2Node.Nodes.Add(L3Node)
                '.tvCatalog.Nodes.Add(L3Node)
                For Each pkg As PackageInfo In pr.Packages
                    L4Node = New TreeNode(pkg.Name)
                    L4Node.ImageIndex = 3
                    L3Node.Nodes.Add(L4Node)
                    '.tvCatalog.Nodes.Add(L4Node)
                Next
            Next
        Next

        .tvCatalog.Nodes.Add(L1Node)
    End With

End Sub

Sub btnSelectClick()

    ' if the image index level indicates a package,
    ' select this node, populate the sFullSSISPath variable,
    ' and close the form

```

```

        With frmISTree
            If Not .tvCatalog.SelectedNode Is Nothing Then
                If .tvCatalog.SelectedNode.ImageIndex = 3 Then
                    sFullSSISPath = .txtServer.Text & "." & _
                        .tvCatalog.SelectedNode.FullPath
                        .Close()
                End If
            End If
        End With

End Sub

Sub tvCatalogDoubleClick()

    ' run the Select Click logic

    With frmISTree
        btnSelectClick()
    End With

End Sub

End Module

```

All the action in this module happens in the subroutines that populate the TreeView control (`btnConnectClick`) and select the node (`btnSelectClick`). The code defaults the name of the server to “localhost.” The user can change it before clicking the Connect button. Once the button is clicked, the code calls `btnConnectClick`.

The `btnConnectClick` subroutine creates objects for the Server, Integration Services, and Catalog objects in the model. Next, it builds a hierarchy of four levels of nodes starting with the catalog. The variables— `L1Node`, `L2Node`, `L3Node`, and `L4Node`— represent the Catalog, Folder, Project, and Package levels of the hierarchy. The code uses a series of nested For Each loops to iterate the SSIS Catalog and populate the subnodes under `L1Node` (Catalog), and then the `L1Node` is added to the `tvCatalog` treeview.

The `btnSelectClick` subroutine builds a string that represents the unique path to the SSIS package in the catalog hierarchy. The code checks to see if a node is selected and then checks to see if the selected node is at the package level. If all is right with the treeview, the variable `sFullSSISPath` is populated with the path to the SSIS package in the catalog. Immediately thereafter, the `frmISTree` form closes. Users can also double-click on a package node in the treeview and invoke the `btnSelectClick` subroutine.

Execute the application to test it! You should see results as in Figure 2-12.

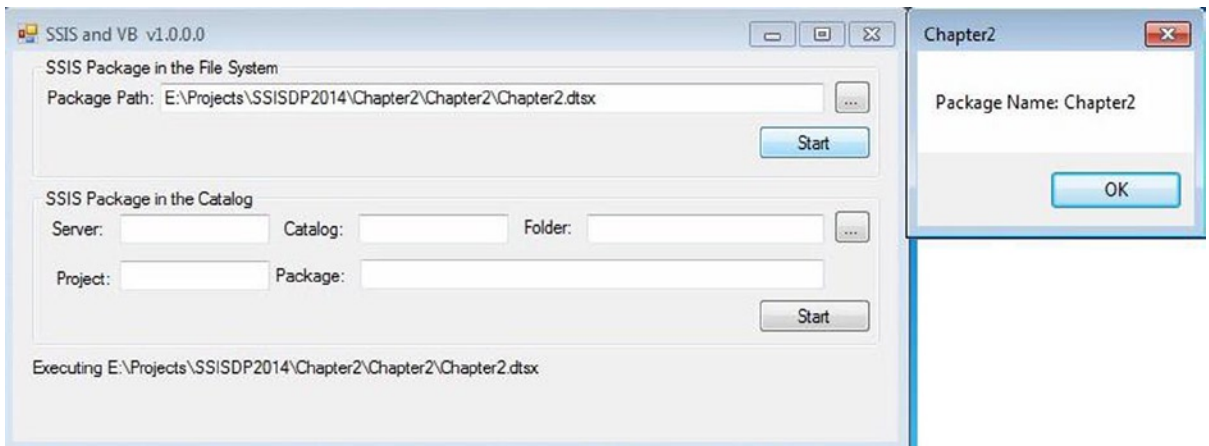


Figure 2-12. Executing a package from the file system

Selection of a package in the SSIS Catalog appears as shown in Figure 2-13.

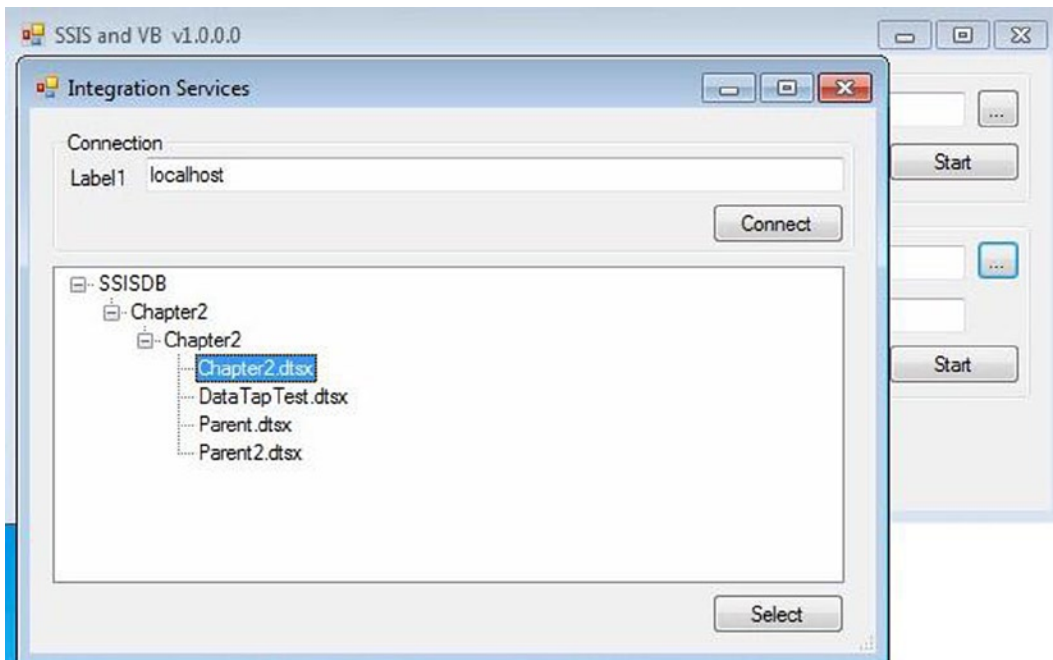


Figure 2-13. Selecting a package in the SSIS Catalog

Execute a package selected from the SSIS Catalog as illustrated in Figure 2-14.

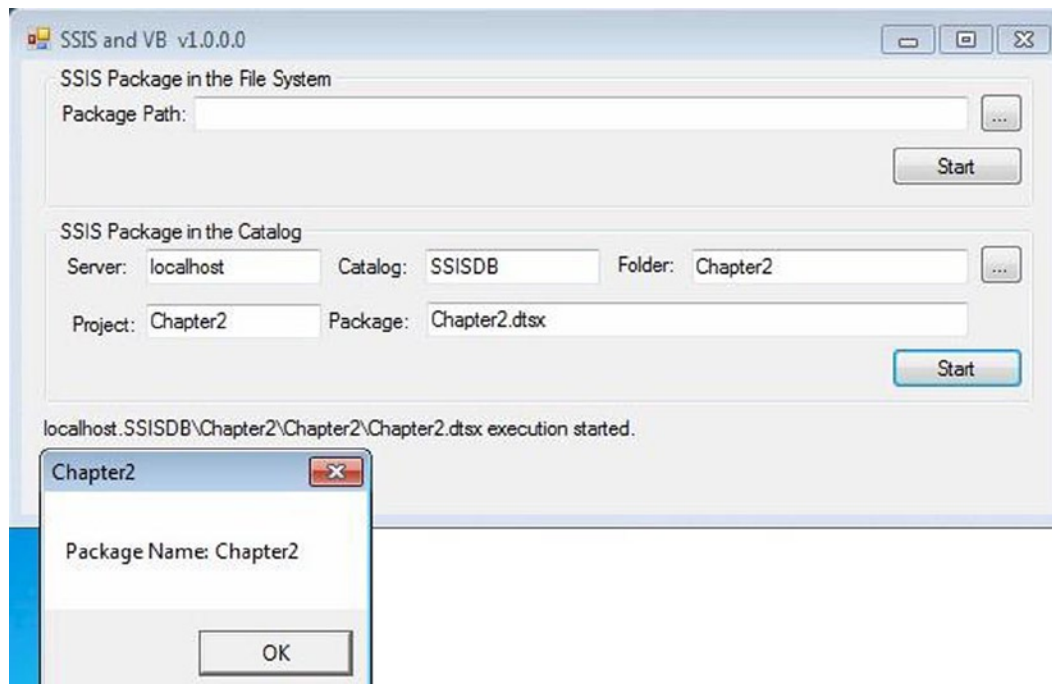


Figure 2-14. Executing an SSIS Package stored in the SSIS Catalog

Conclusion

In this chapter, we surveyed many ways to execute an SSIS package. We examined the many built-in ways for convenient execution of SSIS packages. Then we kicked things up a notch by extending the SSISDB functionality. In the end, we produced a simple, yet functional, custom execution framework and demonstrated how to couple it to the scheduling capabilities of SQL Server Agent jobs to produce a custom parallel-/serial-capable execution engine. We built a .NET application to demonstrate the flexibility (and complexity) of executing SSIS packages from managed code.



Scripting Patterns

As shown throughout this book, SQL Server Integration Services (SSIS) is a multifaceted product with many native capabilities that can handle even the most difficult of data challenges. With highly flexible transformations, such as the Lookup, Conditional Split, Derived Column, and Merge Join, the data flow is well-equipped to perform a limitless number of transformations to in-flight data. On the control flow side, tools including the File System task, the For Each Loop (and its cousin, the For Loop), the File System task, and the Data Profiling task provide critical services to support fundamental ETL operations. Right out of the box, you get a toolbox that's full of flexible and powerful objects.

However, even the most casual ETL developer will eventually encounter scenarios that require more flexibility than what is afforded in the native components. Dealing with data movement and transformation is often ugly and unpredictable, and requires a level of extensibility that would be difficult to build into general-purpose tools. Fortunately, there is a solution for these uncommon ETL needs: custom .NET scripting.

The Toolset

SQL Server Integration Services has the capability to build very powerful ETL logic directly into your SSIS packages. Through Visual Studio and its various niceties (familiar development environment, IntelliSense, and project-based development, among many others), the burden of embedding custom logic in your ETL processes is made significantly easier.

Unlike its predecessor Data Transformation Services (DTS), SQL Server Integration Services exposes the entire .NET runtime within its scripting tools. Gone is the requirement to use only ActiveX scripts within ETL packages (although this capability does still exist in SSIS, for those loyal to VBScript). With the introduction of the rich scripting environments in SSIS, you now have the ability to access the same framework features used in “real” software development. True object-oriented development, events, proper error handling, and other capabilities are now fully accessible within custom scripts in SSIS.

SQL Server Integration Services includes two different vehicles for leveraging .NET code into your packages, each designed to allow different types of custom behaviors. The Script task, which resides in the control flow toolbox, is a broad, general-purpose tool intended to perform support and administrative tasks. Within the data flow toolbox, you'll find the Script component, a versatile yet precise data movement and manipulation tool.

If you're new to scripting in SSIS, you might wonder why there are two different script tools in SSIS. Beyond that, the critical design pattern will include a decision on which of these tools to use in a given scenario. The short answer? It depends. As mentioned, the Script task is typically the better choice for operational behavior and is most commonly used for operations affecting overall package flow (as opposed to data movement). On the other hand, if your ETL needs require generating, consuming, or manipulating rows of data, then the Script component is normally the better tool.

Although they have nearly identical interfaces, the Script task and Script component differ greatly in their default design. As you explore each of these tools, you'll find that there is a significant amount of code automatically added to the script project when you introduce it to your work surface. As a tool designed for direct data interaction, the

Script component will include preconfigured code defining inputs and/or outputs to allow data to flow through the component. In contrast, the behaviors built into the Script task have no facilities for either inputs or puts, further illustrating that this tool was built for purposes other than direct data manipulation.

The Script task and Script component share many similarities. Both tools feature a script designer that resembles the familiar Visual Studio development environment used for mainstream software development. In both tools, you'll find the workspace organized into a virtual solution (displayed very similarly to the solution container found in Visual Studio development) that may include multiple files and folders. Also common to both script tools is the ability to include external code or services within the virtual solution. This capability allows you to leverage code that has already been written and compiled elsewhere, whether it's a DLL file you include with the project or some external resource such as a web service. The language behaviors in both tools will be identical; functionality such as error handling, compiler directives, and core framework functionality is common to both the Script task and the Script component.

Should I Use Script?

Before we get started exploring the design patterns for using scripting in SSIS, a fundamental question should be answered: "Should I be using the scripting tools at all?"

As much as I believe in using the script tools in SSIS to solve difficult problems, there's one piece of advice I almost always dish out when talking or writing about this topic: only use the Script task or Script component in situations where existing tools can't easily address the problem you're trying to solve. Although you gain a great deal in terms of flexibility, you also lose a few things—design-time validation, easy reusability, and a GUI settings editor, among others—when you deploy an instance of script into your packages.

Now don't take any of this to mean that scripting is bad, or that it is reflective of poor package design (after all, this is a chapter describing recommended practices for using script!). Quite the opposite, in fact—I've found many situations where the only tool that could satisfy the ETL requirement was a well-designed script instance. The point is that the Script task and Script component are complex tools intended for use in atypical situations. Native components are much simpler to use and maintain. In situations where one or more native elements of SSIS can easily dispatch any ETL issues, don't complicate the issue by reinventing the wheel with a script.

The Script Editor

Though their purposes differ greatly, you can expect a similar experience in the script editor for both the Script task and the Script component tools. Features of both tools include

- The ubiquitous code window
- Project Explorer
- Full .NET runtime
- Compiler

I'll cover the semantics of writing code in each of these tools later in the chapter. Now, I'll explore some of the other features shared by the Script task and the Script component.

Project Explorer

Software development elements in Visual Studio are stored in logical groups called projects. The Visual Studio environment in SQL Server Data Tools (SSDT) will behave in much the same way; as shown in Figure 3-1, the file(s) for a given script is represented in the Project Explorer window. In the same figure, you can see that the project has a rather lengthy and arbitrary name. This is a system-generated identifier used to identify the project and the namespace in which it resides, but it can be changed if you prefer to have a more standardized naming convention in your script.

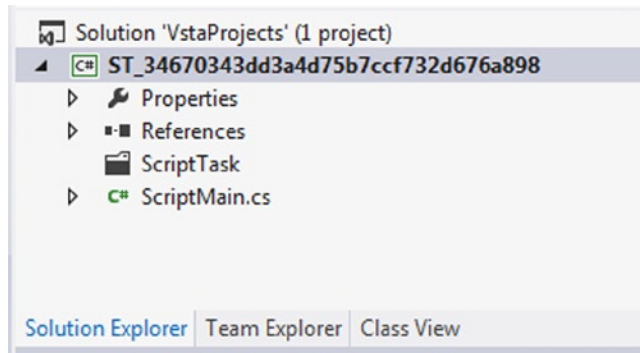


Figure 3-1. Script task Project Explorer

It's worth pointing out that the C# and VB.NET code files that you create in Project Explorer are not physically materialized in your project. Rather, the filenames exist only to logically separate the code files within the project—the code itself is embedded inline within the XML of the package (the dtsx file).

Also included in the Project Explorer is a virtual folder named References. In this folder, you will find the assembly references for core functionality required by the script. In addition, you can add your own references to this list to further extend the capability of the Script task or Script component.

Because each instance of the Script task or Script component in your package is surfaced as a Visual Studio project, you have a significant amount of control over the properties of that project. Much like a full-featured software development project, an instance of the Script task or Script component allows the ETL developer the ability to control various parameters of behavior, including the version of the .NET Framework, the compilation warning level, and the assembly name. The project properties window is shown in Figure 3-2, and it can be accessed by right-clicking the project name in Project Explorer and choosing Properties. Do keep in mind, however, that the scope of any changes you make is limited to the single instance of the Script task or Script component you're currently working with.

In practice, it's quite rare to have to modify the project-level properties of instances of the Script task or Script component. In the majority of cases, the default project settings will suffice.

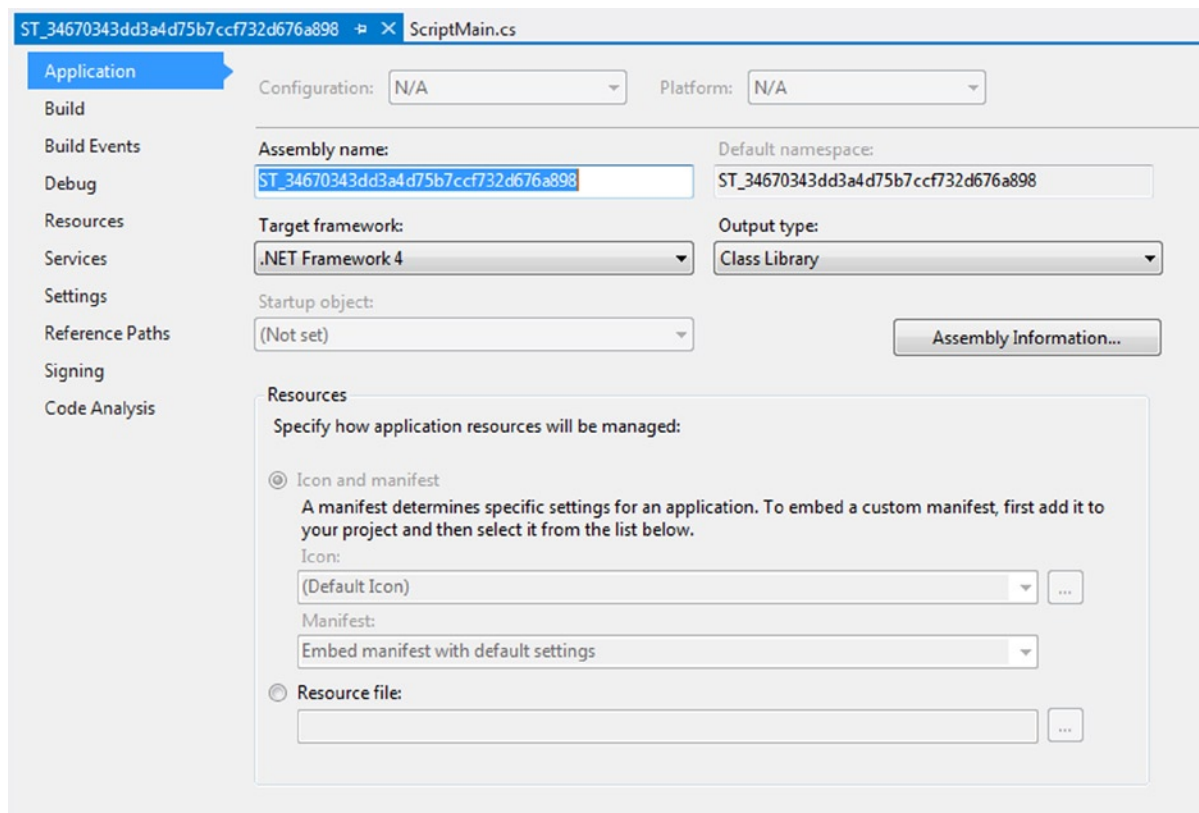


Figure 3-2. Script task project properties

Full .NET Runtime

Both scripting tools in SSIS have full access to the objects, methods, and events within the .NET Framework runtime. This level of accessibility allows the ETL developer to leverage existing assemblies—including network libraries, file system tools, advanced mathematical operations, and structured error handling, among many others—as part of their scripts. Need to create a complex multidimensional array? Looking for an easier way to perform tricks with string manipulation? All this and more is easily accessible, just below the covers of the .NET runtime environment.

Compiler

Finding errors in code is critical and is generally easier to do early in the development process. To understand how this works in SSIS scripting, it's useful to understand the life cycle of script development.

When SSIS first surfaced with SQL Server 2005, the ETL developer could choose to store the code text in a Script task or Script component without actually building (compiling) the script project. By opting not to precompile the script, a trivial amount of processing resources would be saved (or more specifically, delayed) by forcing the SSIS design environment to accept the code as written. Two problems arose from this behavior: first, there was a performance penalty, however slight, caused by compiling the script on the fly during the execution of the package; second, the risk of runtime errors increased, due to the minimized up-front validation in the designer.

Starting with SQL Server 2008, script precompilation is required. Now when you write a script, it is compiled in the background and the resulting binary output is serialized and stored inline within the package XML. As soon as you modify your script and close the editor, the .NET compiler is invoked and creates the serialized binary data. If you've made an error in your code that prevents compilation, you'll be presented with a validation error on the Script task or Script component indicating that the binary code cannot be found (see Figure 3-3).

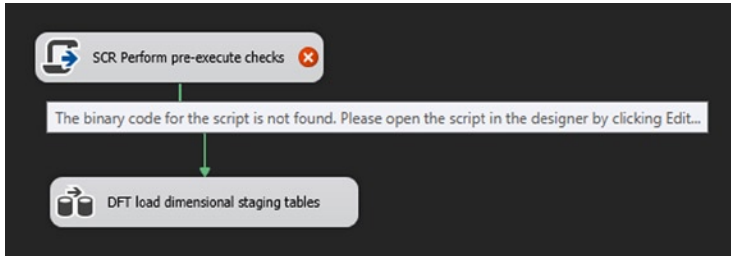


Figure 3-3. *Compilation error with a Script task*

This validation step is a safety mechanism to make sure your packages don't make it to deployment with uncompileable code. However, it's also possible to build the script project from within the script editor, so you can periodically check your code for compilation errors before closing the editing window. In the menu bar of the script editor, you'll find the option to Build ► Build [*script project name*] to compile all of the code in your script project. When you build the project in this manner, any errors found during compilation will be reported in the Error List window (Figure 3-4).

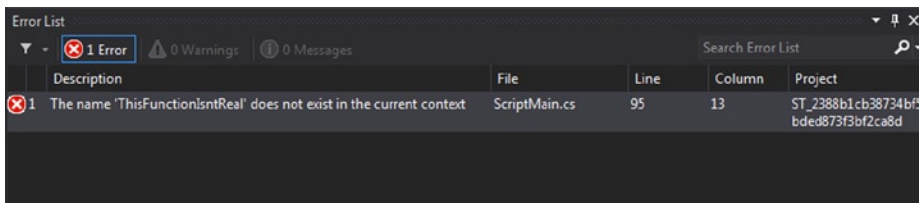


Figure 3-4. *Error List*

Also present in the Error List windows are any warnings generated by the compiler; although they don't prevent the code from being compiled, it's always a wise idea to perform a thorough review of any warnings that appear in this list before sending the code any further down the development process.

The Script Task

Within the control flow pane, the workspace for writing custom code is the Script task. Although it technically can be used for direct data manipulation, this tool is best suited for supporting operations that can't easily be accomplished using native control flow tasks. Think of the Script task as a helper object to be used to tie together other data-centric operations within the package.

The following are a few requirements frequently served by the Script task:

- Checking for the existence and accessibility of source or destination files
- Using a file archive utility API to zip or unzip files
- Generating custom event messages (such as HTML-formatted e-mail messages)
- Setting variable values
- Performing cleanup operations upon package completion or if an error occurs
- Inspecting environmental data, such as available disk space, the status of Windows services, and so on

Because it's not designed for manipulating data, there is no need to define input or output metadata, which makes the script task relatively easy to configure. As shown in Figure 3-5, there are just a few required configuration elements for this task: specify the programming language you want to use, choose the variables you want to make visible within the script, and you're ready to write code!

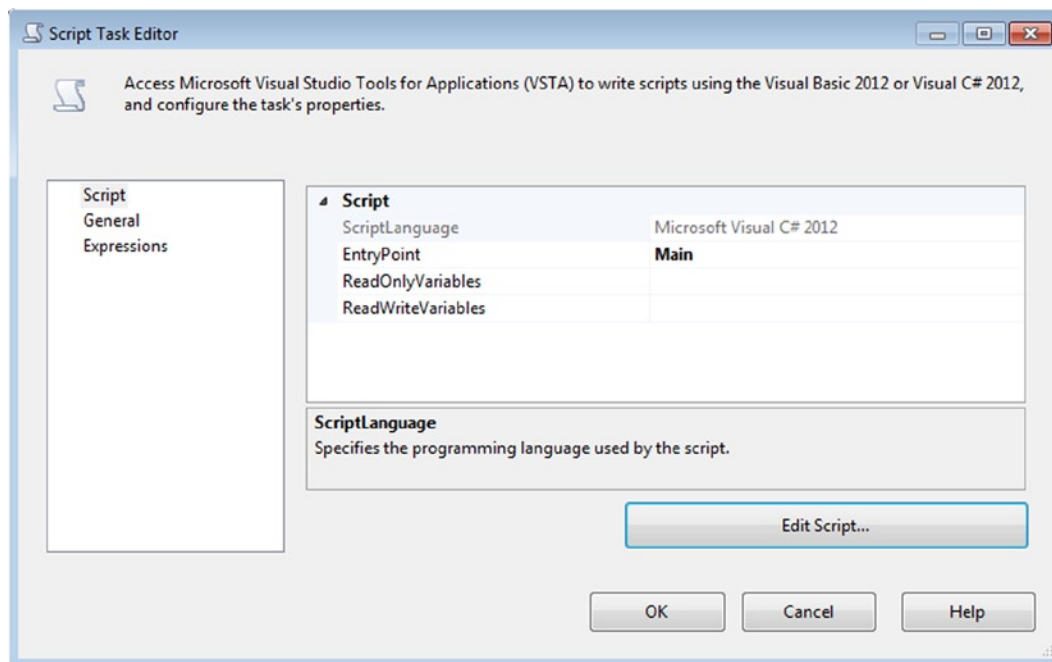


Figure 3-5. Script Task Editor

■ Note For both the Script task and the Script component, the decision to use either C# or VB.NET should be considered permanent. Once you've selected a language and opened the script editor, the language selector is disabled. This behavior is by design; although both languages compile to the same intermediate language (MSIL) code, it would be difficult for any tool to convert source code from one language to another. The default language for both tools is C#, so those who prefer VB.NET will have to change the ScriptLanguage property.

Once the language and variables are set, you're ready to open up the code editor. When you click the Edit Script button, you'll be presented with the familiar Visual Studio development environment. If you have written code using this IDE in the past (to develop Windows applications, web apps, etc.), you'll recognize many entities—the Solution Explorer, the code window, the properties window, and many other common components. Although the behaviors will be a bit different in this minimized version of Visual Studio, the concepts remain the same.

The Script Component

Although both SSIS scripting tools have similar properties, they serve very different roles. Unlike the Script task, which is intended mostly for administrative and operational programmability, the Script component is designed for the more traditional moving parts of ETL: retrieving data from a source, performing some manner of transformation or validation on said data, and loading data to a destination.

Common uses of the script component include:

- Connecting to sources of data for which there is no native source component
- Sending data to destinations that do not offer a native destination or are structured differently than the typical tabular layout
- Performing advanced data manipulation that requires functionality not offered with the built-in SSIS transformations
- Complex splitting, filtering, or aggregating of the in-pipeline data

The Script component is built with the versatility to behave in one of three modes: transformation, source, or destination. When you introduce an instance of the Script component into your data flow workspace, you'll be prompted to select a configuration, as shown in Figure 3-6. This is an important choice, since the selection you make will determine which code template is used to initially configure the component. I'll dig more into each of these roles momentarily.

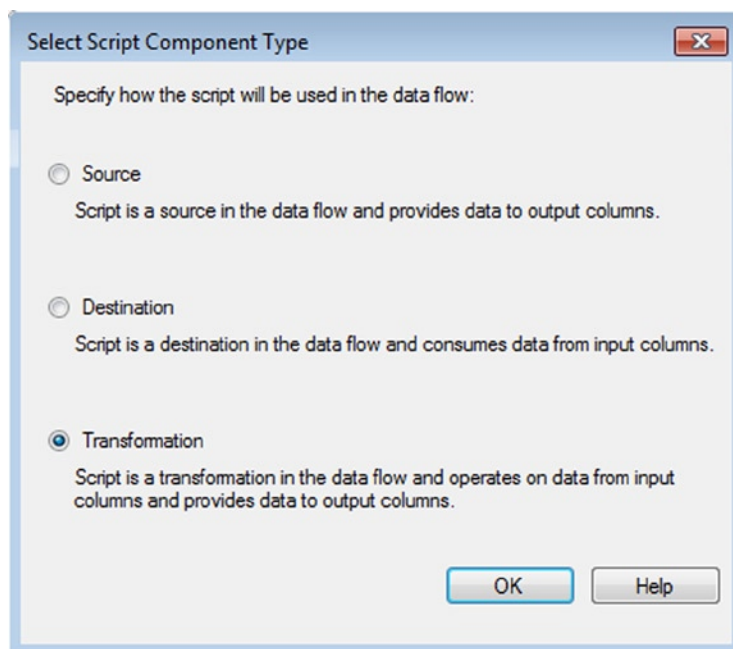


Figure 3-6. Script component configuration

■ **Note** You should consider the selection of the Script component role (transformation, source, or destination) to be permanent. If you mistakenly select the wrong configuration role, it's usually easier to delete and re-create the script instance rather than trying to reconfigure it.

Chances are good that you'll use the Script component in each of these roles (source, transformation, and destination) from time to time. However, my experience has been that transformation is the most frequently used role for the Script component.

Script Maintenance Patterns

Designing custom logic in your ETL processes is a hard, but rewarding, task. However, the job is not done once the code is promoted from development to testing or even to production. The effective ETL architect maintains a long-term vision when designing script solutions, because the packages often survive well beyond the tenures of those who create those solutions.

To that end, an integral part of the solution design process should be evaluating the flexibility, maintainability, and reusability of the project as a whole, making specific allowances for the project-within-a-project instances of the script.

Code Reuse

Laziness is a good thing. (Pause for effect.) To clarify: all great technologists find ways to avoid solving the same problems repeatedly. Let's say you've spent some time working with script tasks and script components, and you come up with a whiz-bang idea for The Next Big ETL Thing. The Thing is so narrowly focused that it adds behavior not present in native SSIS tools, but it's also versatile enough to be used in multiple packages across several domains. Since you worked so hard on it once, you'll want to avoid reinventing The Thing again. The solution: find a way to make The Thing reusable.

To that end, there are several ways to reuse code within SSIS, from the old-fashioned copy/paste to fancy modularization.

Copy/Paste

No further definition is needed here: code reuse via copy/paste is exactly as it sounds. Although copying and pasting code as a reuse mechanism is a bit crude and unstructured, it's also the simplest and most commonly used means to do so within SSIS. The upside is a quick and easy deployment with few administrative limitations. However, this simplicity comes at a cost. Deployed in this manner, each copy of the code exists in its own silo and must be independently and manually maintained.

External Assemblies

As I mentioned earlier in the chapter, both the Script task and the Script component allow you to reference external assemblies (compiled code generated from a separate project) to import supplemental behavior into the instance of the Script task/component. The details of creating an external assembly are beyond the scope of this chapter, but in a nutshell, you would use a separate development environment to code and compile the classes, methods, and events to be used in your ETL processes. The resulting binary file, known as an *assembly*, would be deployed to the

development machine and any server machine(s) that would execute the package. The assembly would then be added to the list of script references, and the result would be that the behaviors defined in the assembly would be accessible from within the instance of the Script task or Script component.

There are several upsides to this approach. First of all, it's a more modular way of handling code reuse within your package. Rather than relying on rudimentary copy/paste operations, this method permits a single point of administration and development for the shared custom functions of your ETL processes. Since all references to the custom behavior would point to a single assembly on each development machine or server, any updates to the code would be addressed at the machine level rather than having to touch every script in every package. In addition, the behaviors built into the external assemblies could be used by other processes or applications; because these standalone assemblies are built using the Common Language Runtime (CLR), their use could be leveraged beyond the borders of SSIS.

There are a few limitations to this approach. First, you cannot use SSDT to create custom assemblies. Although both tools use the Visual Studio shell, they are only installed with the templates to create business intelligence projects and do not natively support other project types. To create an assembly containing custom code, you'd need to use a version of Visual Studio that was configured to generate class library projects (the Standard and Professional versions, or even the language-specific free Express versions)—or, for highly experienced developers, plain old Notepad and the .NET compiler. Another limitation is that any assemblies referenced in your script must be deployed to and registered in the Global Assembly Cache, or GAC, on the machine(s) that will execute the package. This deployment and registration process is not complex, but it does add to the total of moving parts in your ETL infrastructure.

Custom Tasks/Components

At the top of the SSIS reusability food chain you will find the custom task and custom component. As with a custom assembly, the ability to add your own tasks and components to SSIS allows you create highly customized behaviors within SSIS. In addition, custom tasks and components enable you to create a more customized user interface for these behaviors, allowing for relatively simple drag-and-drop use in your SSIS packages. In the interest of brevity, we won't detail the use of custom tasks or custom components in this chapter, but it is worth mentioning that if there is script behavior that is often repeated in many of your packages, it's worth considering converting the Script task or Script component into a custom tool that can easily integrate into SSDT.

Source Control

Ask any good developer for their short list of required project elements, and source control will almost always be near the top of the list. Because any SSIS project really is software development—albeit in a mostly graphical environment—the same consideration must be made by ETL developers as well. Being that the storage for an SSIS package is simply an XML file, it's not difficult to add SSIS packages to most any existing source control system.

To some, the Script task and Script component have the appearance of residing outside the SSIS package—after all, both of these are managed through what appears to be a separate Visual Studio project. This thinking sometimes brings up the question of how to integrate SSIS script logic into source control. The easy answer is that there is no requirement above and beyond source controlling the package itself. Because all of the code is stored inline within the package XML, there is no need to separately track in source control the code within instances of the Script task or Script component.

Scripting Design Patterns

As a born-and-raised Southerner, I was always taught that there's more than one way to skin a cat. Although I've never attempted this particular exercise, I can confirm that for any given problem (in life, as well as in SSIS) there may be dozens or perhaps even hundreds of correct solutions.

Because of the highly flexible nature of scripting solutions, it's not possible to predict every possible permutation of logic that could find its way into SSIS code. However, as Integration Services has evolved, some commonly used design patterns have emerged. In this section, I'll look at some of these patterns.

Connection Managers and Scripting

Connection managers are built into SQL Server Integration Services as a modular way to reuse connections to databases, data files, and other sources of information. Most everyone with even a little experience using SSIS is aware of connection managers and how they relate to conventional components such as OLE DB Source/Destination and Flat File Source/Destination, as well as tasks such as the FTP task and the Execute SQL task. You can instantiate a connection object once as a package-level entity (as shown in Figure 3-7) and use it throughout the remainder of the package.

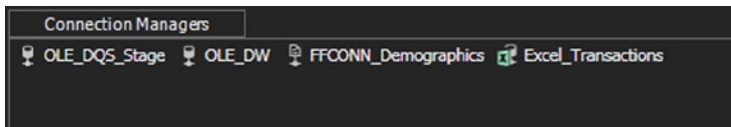


Figure 3-7. Connection manager objects

Not as widely known is the fact that you can access most connection manager objects within instances of the Script task and the Script component as well. Connecting to a data source is a very common task in SSIS scripting, particularly within instances of the Script component used as a source or destination. However, in real-world usage, I find that a lot of ETL developers go through the exercise of programmatically creating a new connection within the script, even if the package already has a connection manager.

If a connection manager object already exists in your SSIS package for a particular connection, it's preferable to use that existing connection manager object when connecting to a data store from within a script. If the connection manager does not yet exist, consider creating one: there are numerous benefits to abstracting the connection from the script, including ease of change and the ability for the connection to engage in transactions within SSIS.

Note For an in-depth analysis on why to use connection manager objects as opposed to connections created entirely in script, you can review an excellent blog post on this topic by Todd McDermid here: <http://toddmcdermid.blogspot.com/2011/05/use-connections-properly-in-ssis-script.html>. In this post, the author specifically discusses the use of connection managers in the Script task, but most of the same principles would apply to the use of connection managers within the Script component as well.

Although it's possible to reuse connection managers for most any connection type within a script, to keep things simple, I'll limit my discussion to SQL Server database connections. With some modification, many of these principles would apply to other connection types as well.

Using Connection Managers in the Script Task

Although not entirely intuitive, the coding syntax to reference an existing connection manager in the Script task is relatively easy to understand. I'll look at examples for the two most common ways to connect to SQL Server—through the OLE DB connection and the ADO.NET connection.

Connecting to an ADO.NET connection manager through the Script task is a two-step process, as shown in Listing 3-1. First, create a reference to your existing ConnectionManager object (using the name you gave it in the SSIS package), and acquire a connection from that object in code.

Listing 3-1. Use an Existing ADO.NET Connection in the Script Task

```
// Create the ADO.NET database connection
ConnectionManager connMgr = Dts.Connections["ADONET_PROD"];
System.Data.SqlClient.SqlConnection theConnection
    = (System.Data.SqlClient.SqlConnection)connMgr.AcquireConnection(Dts.Transaction);
```

Using an OLE DB connection manager in the Script task requires a little more code but is a similar exercise to its ADO.NET counterpart. As shown in Listing 3-2, you have to add an intermediate object to make the appropriate data type cast when using the OLE DB provider:

Listing 3-2. Using an Existing OLE DB Connection in the Script Task

```
// Create the OLEDB database connection
// Note that we'll need to add references to Microsoft.CSharp and Microsoft.SqlServer.DTSRuntimeWrap
ConnectionManager cm = Dts.Connections["AdventureWorks2012"];
Microsoft.SqlServer.Dts.Runtime Wrapper.IDTSConnectionManagerDatabaseParameters100 cmParams
    = cm.InnerObject as Microsoft.SqlServer.Dts.Runtime Wrapper.
    IDTSConnectionManagerDatabaseParameters100;
System.Data.OleDb.OleDbConnection conn = (System.Data.OleDb.OleDbConnection)cmParams.
    GetConnectionForSchema();
```

■ **Note** It's worth mentioning that it is technically possible to reference a connection by number rather than by name (e.g., using `Dts.Connections[3]` rather than `Dts.Connections["Conn_Name"]`). I strongly recommend against this practice! It makes for rather ambiguous connection references, and since the order of connection managers cannot be guaranteed, you might end up with a runtime error at best—and at worst, wildly unexpected behavior in your package.

Using Connection Managers in the Script Component

As I mentioned in the previous section, many of the same concepts apply to the reuse of connection managers, whether you're working in the Script task or the Script component. For the same reasons, it's almost always a best practice to reuse an existing connection manager object (or create a new one if necessary) rather than to build a connection object from scratch within code.

Logistically, connection managers are a little easier to use in the Script component. Because the purpose of this tool is to move data (which is not always the case with the Script task), some additional functionality comes baked in to make the job of reusing connection managers less cumbersome. In the Script component, you can declare the use of one or more connections within the script's graphical editor (much like you declare the use of read-only or read-write variables, to be discussed shortly). As shown in Figure 3-8, this interface allows you to easily reference an existing connection.

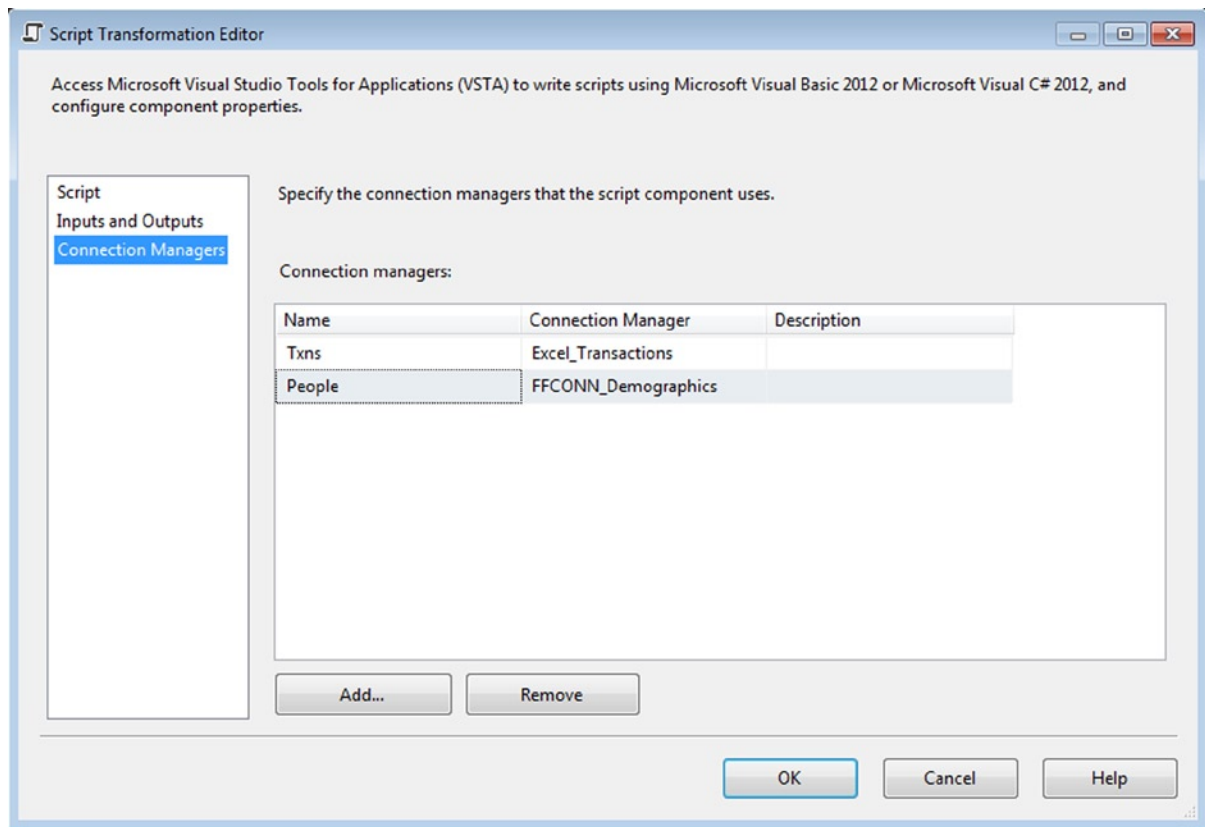


Figure 3-8. Declaring connection managers in the Script component

Once they are referenced here, the syntax to use the connection within your code is much simpler as well. Rather than using the connection name as an indexer, you can access any of these connections through the `UserComponent.Connections` collection. Here's an example:

Listing 3.3. Using a Previously Declared Connection Manager in Script Component

```
// Connect to the ADO database connection
System.Data.SqlClient.SqlConnection conn = (System.Data.SqlClient.SqlConnection)Connections.
OLEDBPROD.AcquireConnection(null);
```

Variables

In many—if not most—instances of the Script task and Script component, you'll need to inspect or manipulate values stored in SSIS variables. Because they are so prevalent in these implementations, it's important to understand how best to address SSIS variables within the scripting tools.

Note It is important to draw a distinction between variables in SSIS and variables declared within the Script task and Script component. Although there's some commonality in their usage, they are separate and distinct entities with very different properties. SSIS variables are defined as part of the SSIS package and may be used across many different tasks and components. Script variables, on the other hand, are declared within individual instances of the Script task or Script component and are only valid within the instance in which they are defined.

Variable Visibility

In both the Script task and Script component, you can explicitly expose one or more variables using the GUI editor for each. In Figure 3-9, you can see that we have the option of including SSIS variables in the script and can specify whether those variables will be surfaced as read-only or read-write.

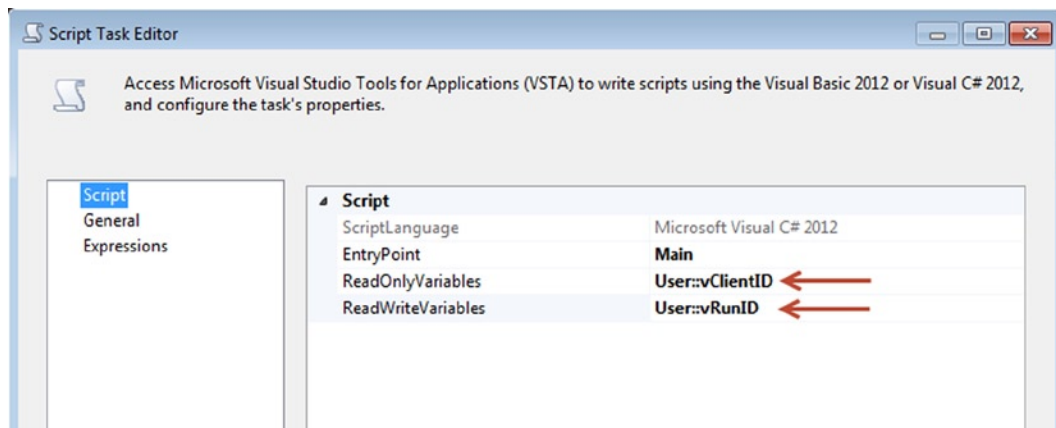


Figure 3-9. Including variables in script

It is possible to read or modify SSIS variables with a script even if you don't explicitly include them. However, it's usually preferable to declare any required variables as shown, since the syntax within the script is much simpler when references to the SSIS variables are declared ahead of time in this manner.

Variable Syntax in Code

Declaring read-only or read-write variables is a similar experience whether you're using the Script task or the Script component. However, the syntax to address these variables in code is different depending on the tool you're using. As shown in Listing 3-4, SSIS variables within a Script task instance are addressed by using a string indexer to specify the name of the variable.

Listing 3-4. Script Task Variable Syntax

```
public void main()
{
    // Get the current RunID
    int runID = int.Parse(Dts.Variables["RunID"].Value.ToString());

    // Set the ClientID
    Dts.Variables["ClientID"].Value = ETL.GetClientID(runID);

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

When using an instance of the Script component, the syntax is noticeably different. Rather than using an indexer to read from or write to the referenced SSIS variables, you can use the `Variables.<Variable Name>` syntax as shown in Listing 3-5:

Listing 3-5. Script Component Variable Syntax

```
public override void Input0_ProcessInputRow(Input0Buffer Row)
{
    // Push the SSIS variable ClientName to the value in the current row
    Row.ClientName = Variables.ClientName;
}
```

It is possible to access variables within instances of the Script task or the Script component even if you do not explicitly declare their use in the graphical settings editor. Within the SSIS scripting toolset, you will find a couple of functions that will allow you to programmatically lock and unlock variables for either read-only or read-write access. As shown in Listing 3-6, you can use the `VariableDispenser.LockOneForRead()` function to capture the value of a variable that was not previously declared.

Listing 3-6. Manually Lock a Variable

```
// Programmatically lock the variable we need
Variable vars = null;
Dts.VariableDispenser.LockOneForRead("RunID", ref vars);

// Assign to script variable
runID = int.Parse(vars["RunID"].Value.ToString());

// Unlock the variable object
vars.Unlock();
```

Using a method similar to the one shown in Listing 3-6, you can manipulate variable values by using the function `VariableDispenser.LockOneForWrite()`, which would allow you to write to as well as read from the variable value.

Variable Data Types

As you may have derived from Listing 3-4 and Listing 3-5, the interpreted data type for variable values will differ between the Script task and the Script component. With the latter, any variable that you declare in the graphical settings editor will surface as the .NET data type equivalent of the SSIS variable type, and there is no need to perform a type cast. When using the Script task (and the Script component, if you opt to use either the `LockOneForRead()` or

LockOneForWrite() method), variables are presented as the generic Object type, and most of the time you'll have to cast to an appropriate type any variable used in script code. As shown in Figure 3-10, you'll get a compiler error if you forget to cast a variable value to the appropriate type.

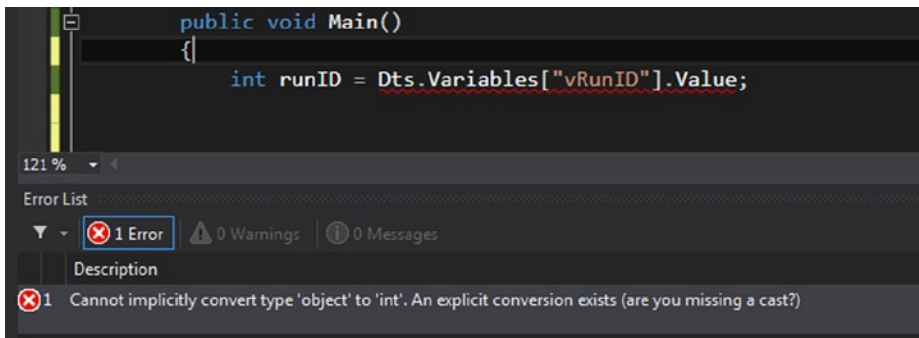


Figure 3-10. Script task variables must be cast

Naming Patterns

If you have worked as a software developer in the past, the following section will be nothing more than a review. If you haven't, I'll share an important tidbit of information: naming conventions are important.

I've known many a developer in my time, and I've never found one who wasn't loyal to some type of naming convention. Why? It's familiar. It's predictable. When patterns emerge in the way you develop software, it becomes easier to maintain—by others as well as yourself. Technically, there's no difference between code written in camel case, Hungarian notation, mnemonic notation, or Pascal style. This is purely a matter of clarity, readability, and maintainability. By finding and sticking with a style (even if it's a hybrid of other styles), you'll have more navigable code and will likely find that your fellow developers appreciate the approach.

Here are a few suggestions regarding naming conventions:

Be consistent: This is the number-one rule and should be followed above all others.

Whatever style you develop, stick with it. You can change or modify your naming convention style later, but at least be consistent within each project.

Be clear: I can't tell you how many times I've had to debug code (and yes, sometimes it was my own) littered with single-character object names, ambiguous function names, and other pull-your-hair-out kinds of practices. Now, don't go overboard here. Most object names don't need to read like `database_write_failed_and_could_not_display_interactive_error`, but there's probably some happy medium between that and variable names such as `f`.

Be a follower: If you don't have your own style, do as those around you do. Some organizations, especially larger ones, may dictate the naming convention style you'll use.

Conclusion

The scripting tools in SQL Server Integration Services are both capable and complex. When scripting for ETL first surfaced with DTS many years ago, it was a quick-and-dirty way to solve some data movement and manipulation problems. With SSIS, the latest generation of scripting tools is robust and enterprise-ready. With a few recommended practices, scripting can be a great addition to any ETL developer's toolkit.



SQL Server Source Patterns

In the first section of this book, we looked at patterns focused on the control flow area of SQL Server Integration Services, including metadata, workflow execution, and scripting. The second section focuses on the data flow area of SQL Server Integration Services. This and the following chapters will discuss source, transformation, and logging patterns in the pipeline area of an Integration Services package.

Integration Services supports a wide variety of sources, including SQL Server, Oracle, and SAP. In addition, developers and third-party vendors have the ability to create custom sources for providers not included out-of-the-box. This technology-agnostic approach creates a very flexible system for loading all sorts of data. Even with all of the potential sources, loading data into or out of a SQL Server database is a very common occurrence, since a company that owns Integration Services typically uses all Microsoft products.

This chapter discusses different patterns associated with using SQL Server as a source. Due to the common occurrence of SQL Server databases in shops using Integration Services, we have a defined set of patterns for extracting data from SQL Server. Specifically, we will look at the best way to connect to a SQL Server database, how to choose the data you will use, and how to more easily use the rest of the data flow's objects. Finally, we will look at a new component in SQL Server Denali that helps jump-start your development when connecting to any source.

Setting Up a Source

When connecting to external data, Integration Services uses a few objects to help make the connection, retrieve the correct data, and start any necessary data manipulations. Every time an Integration Services developer creates a package, the developer will need to select the correct objects and ensure they have all been created. The objects that will need to be set up are as follows:

Connection Manager: The object that tells Integration Services where to get data.
Can be used in the control flow, data flow, and event handlers.

Provider: The object that the connection manager uses to talk to the data source.

Source Component: The object that sets the properties to tell Integration Services what data to get. The matching Connection Manager object is set in this object.

Source Component Query: The information the external data source needs to give Integration Services data. The query is stored in the Source Component object.

Let's take a look at the important factors associated with each of these items. We'll begin in the next section by looking at connection managers.

Selecting a SQL Server Connection Manager and Provider

Between ADO.NET, ODBC, and OLE DB, there are enough connection managers to make you want to pull your hair out! All of these connection managers will connect to SQL Server, so how do you know when you should use which one? To answer that question, let's talk about what the connection manager is actually doing and then look at each connection manager type that you can use to connect to SQL Server.

A *connection manager* is the object that holds connection information for an external source, akin to an application data source or a Reporting Services shared data source. The connection manager provides an abstraction layer between Integration Services and the rest of the components so that information about the external source can be modified in one place to affect all tasks and components. To see all of the connection manager types available, take a look at Figure 4-1.

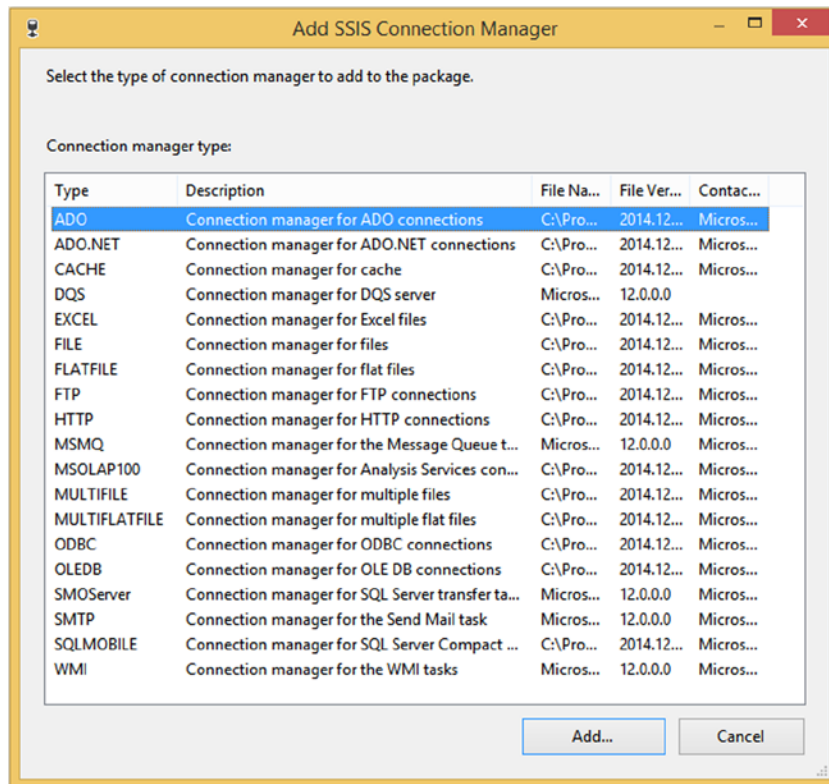


Figure 4-1. Connection manager types

These are the three connection manager types that you can use to connect to a SQL Server database:

- ADO.NET
- ODBC
- OLE DB

Let's take a look at each connection manager type individually.

ADO.NET

The ADO.NET connection manager type is used to make a connection through the .NET Framework. Not only can this type be used for SQL Server, but it also provides access to other applications and other databases. The ADO.NET layer quickly retrieves data from the source using a `DataReader` object in the .NET Framework.

The ADO.NET connection manager for SQL Server is best used as a source when you are using it elsewhere in the package. For example, the Lookup component uses an ADO.NET connection manager to connect, so then you should use it as a source. On the other hand, if a component uses another connection manager type, stick with that connection manager type. Consistency is really the key here. For a sample connection manager property window set to connect to the `AdventureWorks2012` database on the same server, see Figure 4-2.

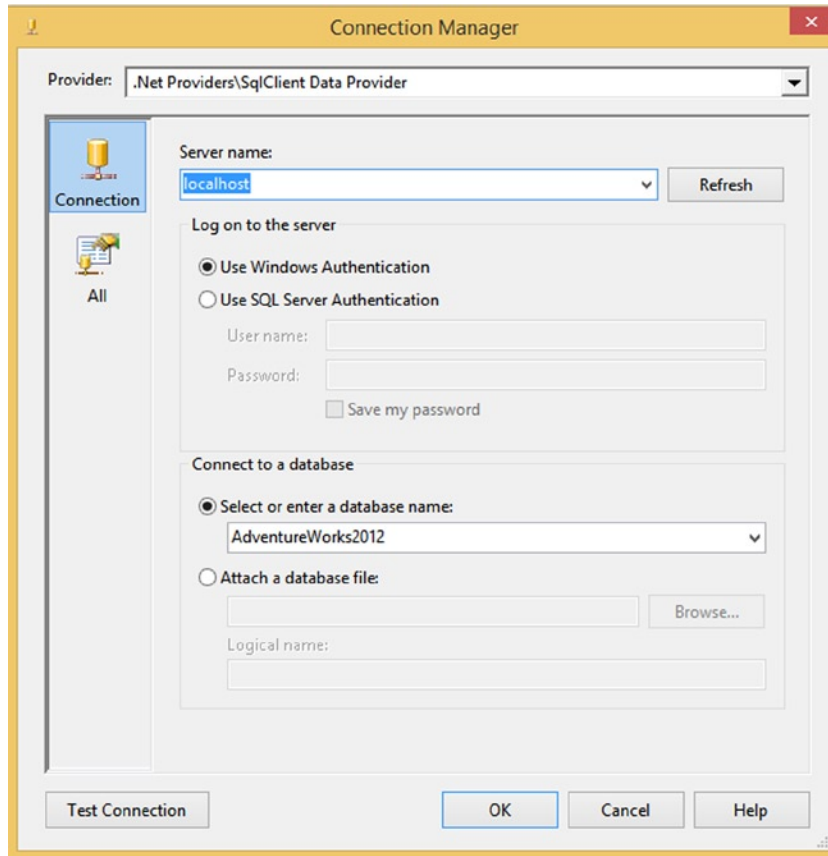


Figure 4-2. ADO.NET connection manager property screen

ODBC

ODBC is the open database connectivity standard. Its purpose is to allow connections from any application to any database, regardless of the vendor. Often, an organization will use DSNs (data source names) to create an abstraction layer between the application and the connection string the ODBC provider uses. If you have an organization that really wants to use DSNs with SQL Server, ODBC is the option for you. Otherwise, stick with an ADO.NET or OLE DB connection manager.

SSIS did not have an ODBC Source component prior to SQL Server 2012. Instead, you could and still can use the ADO.NET connection manager with a few tweaks. After you create the ADO.NET connection manager, change the provider at the top of the window to Odbc Data Provider, as shown in Figure 4-3.

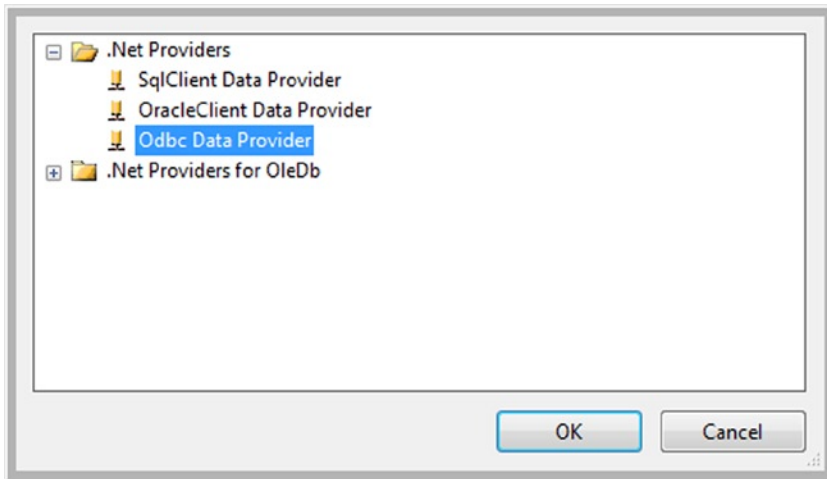


Figure 4-3. ADO.NET providers

Then add the DSN name or connection string. For our local AdventureWorks2012 database, the connection string will look like Listing 4-1.

Listing 4-1. ODBC Connection String

```
Driver={SQL Server Native Client 11.0};  
Server=localhost;  
Database=AdventureWorks2012;  
Trusted_Connection=yes
```

Our completed Connection Manager screen for an ADO.NET connection with an ODBC provider looks like Figure 4-4.

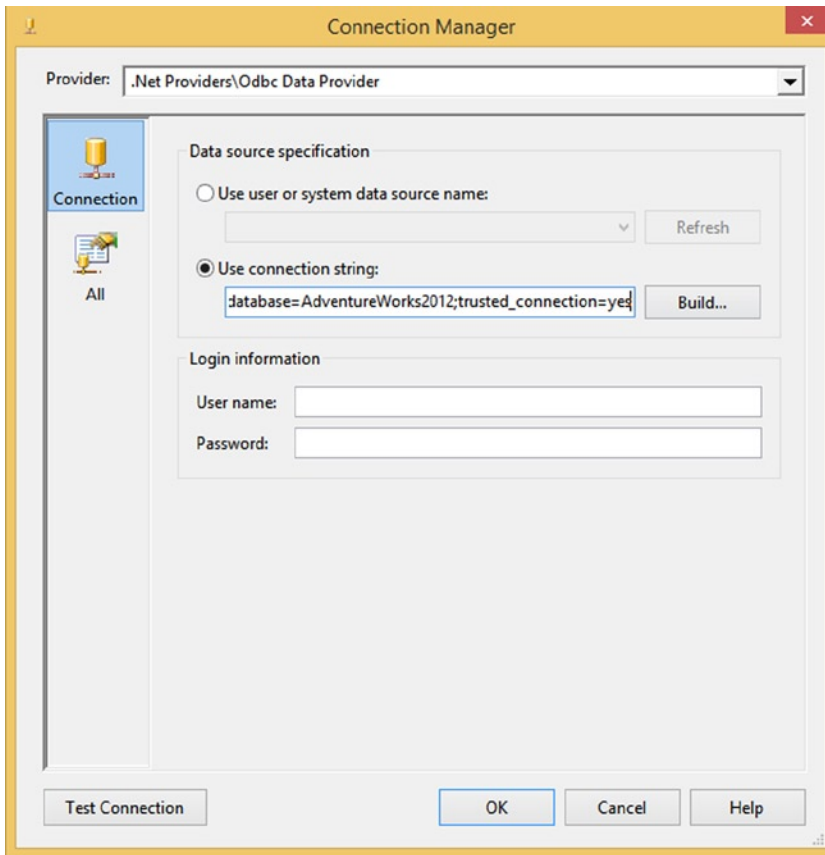


Figure 4-4. ODBC ADO.NET connection manager property screen

If you are using SQL Server 2012 or later, you can use the ODBC Source component. Configuring the ODBC Source is very similar to the ADO.NET Source, except that you use an ODBC connection manager. The ODBC Source provides additional performance benefits over ADO.NET; however, it requires SQL Server Enterprise Edition to run outside of the SQL Server Data Tools environment.

OLE DB

Finally, we move on to what is arguably the most common connection manager used to connect to SQL Server: OLE DB. The OLE DB protocol was written by Microsoft as the next version of the ODBC provider. In addition to SQL Server databases, you can use the OLE DB connection manager to connect to file-based databases or Excel spreadsheets.

OLE DB tends to be my default when I'm connecting to a SQL Server database. If you do not fall into the category of using mostly components that use an ADO.NET connection manager, and you do not fall into the category of having an organization that wants to use a DSN, you will want to use an OLE DB connection manager.

You can fill out the property screen of the OLE DB connection manager as shown in Figure 4-5.

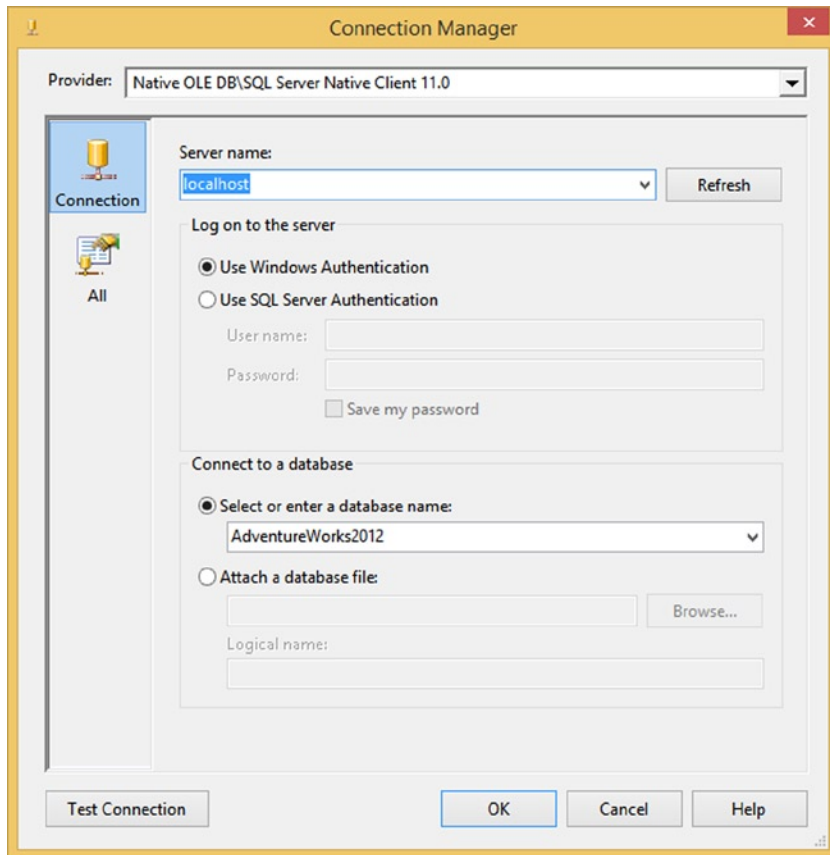


Figure 4-5. OLE DB connection manager property screen

Creating a SQL Server Source Component

Once you've picked the correct connection manager and provider, you need to use them in a Source Component for our data pull. Begin by looking at the SSIS Toolbox when you are on the Data Flow tab. If you have not rearranged the SSIS Toolbox, you will see all possible sources under the Other Sources grouping, as seen in Figure 4-6.

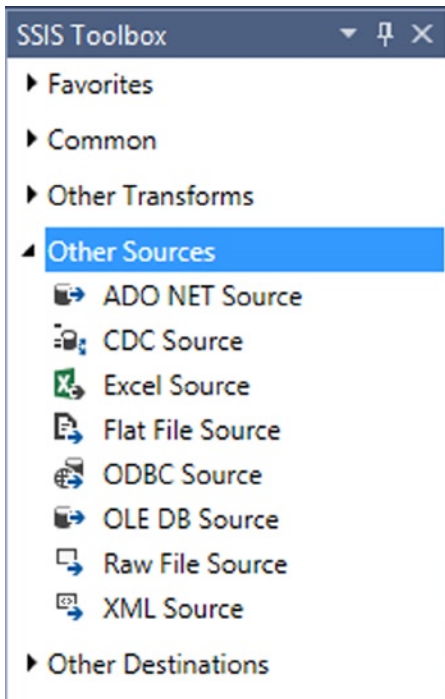


Figure 4-6. Other Sources grouping in the SSIS Toolbox

Now it's time to choose one.

Most of the hard decision-making was already completed when you set up the connection manager. If you use an OLE DB connection manager, then you must use the OLE DB source. If you decided to use an ADO.NET connection manager with either an ADO.NET or ODBC provider, you must use the ADO NET Source.

Once you drag the desired source onto the Data Flow design window, the data flow contains one component, as shown in Figure 4-7. Integration Services lets the developer know that there is an issue with the source by displaying the red circle with the white X in it. In this case, the issue is that you have not yet set any of the source's properties, starting with the destination table, as shown in the tooltip.

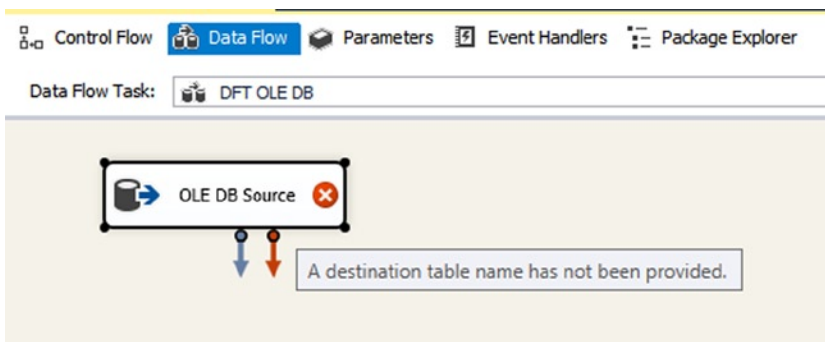


Figure 4-7. Data Flow task with new source

To open the source component, either double-click the component or right-click and select Edit. Inside the source component, you can fill in the Connection Manager property. The first connection manager of the appropriate type will automatically be populated, but selecting the drop-down list arrow will let you select any of the other connection managers that match. You can see the OLE DB Source Editor screen up to this point in Figure 4-8.

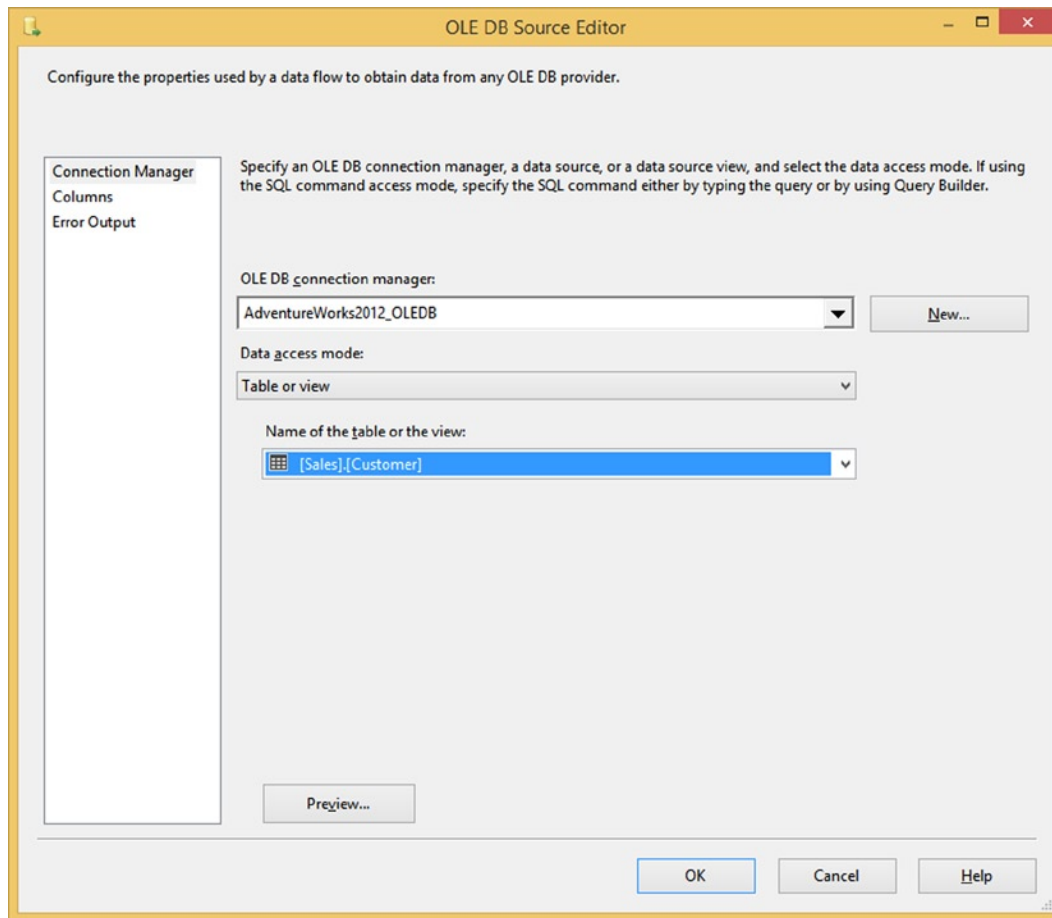


Figure 4-8. Initial OLE DB Source Editor screen

Although most of the decision-making work was already done when you created the connection manager, it is important to understand the part the source plays in the Integration Services package. The source is the glue that holds all of the other pieces together and ensures that you have one place to go to for future maintenance issues or changes. Setting up the SQL Server source was an easy step to get you warmed up before you move on to create and optimize the query that the source uses.

Writing a SQL Server Source Component Query

Now that you have walked through the creation of the connection manager and provider and decided which source component to use, you need to set up the metadata for pulling the data. You do this by selecting what type of access you want to make and then adding the query information to the source component. In addition, you will want to review a few patterns when you're setting up the query and column metadata. Let's get started.

ADO.NET Data Access

If you decided to use the ADO.NET source component, either with the ADO.NET or ODBC provider, you have two options to select what data you want to see:

Table or View: Select which table or view from which you want to receive data. The list of tables and views should be prepopulated and listed based on your access. We do not recommend this option because it includes unnecessary columns, even if you restrict the column list in the component.

SQL Command: Enter text that will be executed on the SQL Server database.

Because the Table or View option is not our recommended option, let's dig into the SQL Command option a little deeper. You can enter either a direct SQL query that returns a dataset or a stored procedure using the EXEC statement.

Whether you are using a SQL query or executing a stored procedure, you will need to be aware that the ADO.NET source does not allow you to use parameters in your query. If you need to modify the query that gets used, you will need to use an expression. Expressions are only set at the control flow level, so you will need to take a look there to set up your expression. Follow these steps to set a new SQL command at design time:

1. When in the Data Flow task, click the background to ensure no components are selected and look in the Properties menu for the Expressions property.
2. Once the Expression Property window is open, select the [ADO NET Source]. [SqlCommand] option in the Property field and click the ellipses button next to the Expression field.
3. In the Expression Editor, create your command using variables. For example, if you were to run a stored procedure where you wanted to pass in an end date, you could use the following expression: `"EXEC GetCustomerData '" + (DT_STR, 29, 1252) @[System::ContainerStartTime] + '""`
4. Once the expression has been validated, click the OK button. The final Property Expressions Editor screen should look like Figure 4-9.

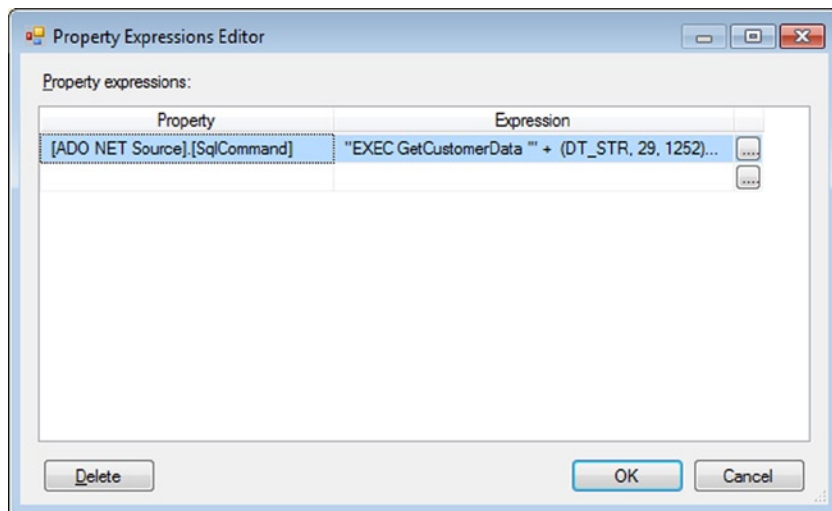


Figure 4-9. *Property Expressions Editor with SqlCommand property*

When the package runs, it will now use the expression you just created.

OLE DB Data Access

If you've selected the OLE DB source component, you have four options for how to retrieve data:

Table or View: Similar to access in the ADO.NET source, this option allows you to select a table or view from which to pull all columns of data into the package. This option is not recommended for the same reason explained in the ADO.NET source.

Table Name or View Name Variable: Instead of hardcoding the name of the table or view, you can instead point to a user-created variable that contains that information. This option is not recommended.

SQL Command: Similar to access in the ADO.NET source, you can enter the SQL query or execution of a stored procedure once you've selected this option.

SQL Command from Variable: If you want to create a query to change at runtime or to pass a variable to a stored procedure, this is the option you will want to use. Instead of creating an expression to modify the SQL Command, as you did with ADO.NET, you will create a variable that creates your expression. You can then select the variable you created after you select this option.

By picking one of these options, you will determine how the data is returned from SQL Server. After you select the type of data retrieval, you'll want to add the appropriate properties. For example, if you select the Table or View option, you'll need to select the object that contains the data. If you select the SQL Command option, you'll need to enter the SQL query or stored procedure execution that returns the data. Once that is set, you can move on to designing the rest of your data flow.

Waste Not, Want Not

As data professionals, we often think that the more data we can get, the better. This isn't always the best scenario when you are dealing with sources. When you are talking about the amount of data to pull, you will want to follow a different pattern.

No matter which query option you selected, it is important to only ask for the columns that you need in your data load process. Requesting all columns is similar to running a `select * from table` query against a database. Not only are you asking the database and network to do more work, but you are also asking Integration Services to do more work. All of that unnecessary data will get stored in memory, or even cause paging if there isn't enough memory, using up space that could be used to grab more data for the important columns and slowing down the overall package execution.

All source components give you the option to pick a subset of columns on the Columns menu. Be sure to make the column reduction in the query itself rather than in the Columns menu to reap the full benefit of a faster package.

Data Translations

Another Integration Services source trap that is easy to fall into is to perform the majority of the data transformations in the source query itself. Because Integration Services developers often have a SQL background, we tend to want to use a familiar tool to accomplish our tasks.

The types of data transformations that can be undertaken in either the source query or the rest of the data flow include merging of datasets, case statements, string concatenation, and more. Remember that SQL Server is very good at set-based actions, whereas Integration Services is very good at computationally expensive tasks that use a lot of memory. Although you should test your individual situation, these are good rules of thumb to follow.

Follow the pattern listed in Table 4-1 when you are deciding where to put your data translation logic.

Table 4-1. *Data Translation Locations*

Data Translation	Concern	Location
Merge datasets	Set-based	Source component
Case statements	Memory intensive	Data flow
String concatenation	Procedural	Data flow
Sorting data	Set-based	Source component

Source Assistant

Now that you have retrieved data from SQL Server the hard way, you're going to learn the easy way to do the same thing. The Source Assistant is a new wizard introduced in SQL Server 2012 that takes you through the steps of setting up your Source objects without having to make many of the same decisions that you just had to go through. This is a great way for people who are just getting started with Integration Services to get up and running quickly.

To begin, create a new Data Flow task. As seen in Figure 4-10, the Source Assistant appears in the SSIS Toolbox. Initially it will be in the Favorites grouping, unless you have moved the items around.

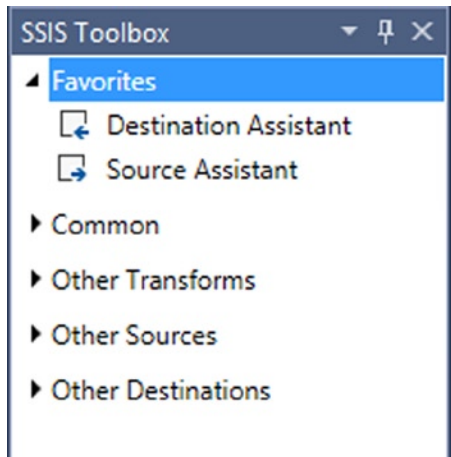


Figure 4-10. Source Assistant in Favorites grouping in SSIS Toolbox

Drag the Source Assistant component onto the Data Flow design area to start the wizard. The first screen can be seen in Figure 4-11.

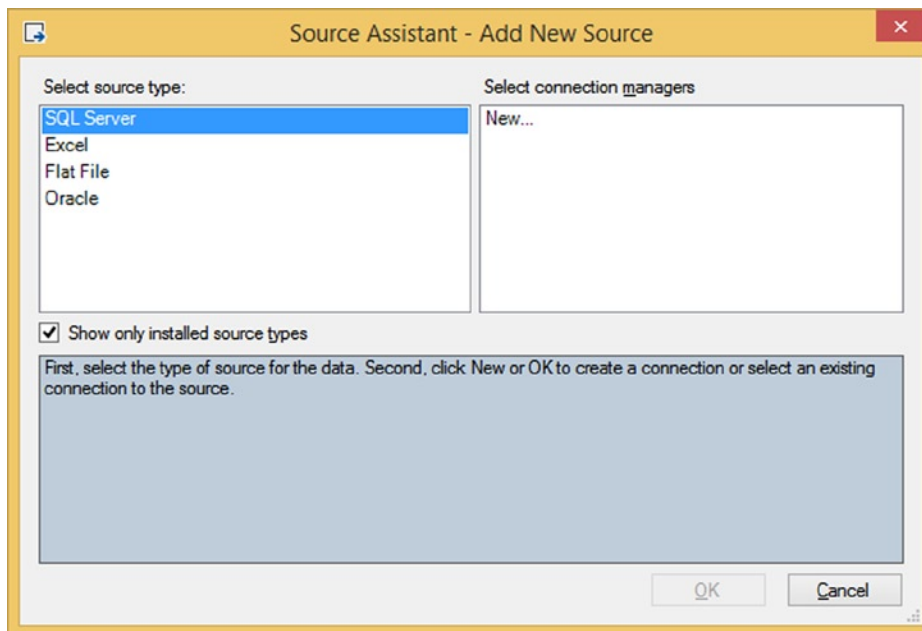


Figure 4-11. Add New Source screen in Source Assistant

To begin, you can see that there are only a few types listed for you to use: SQL Server, Excel, Flat File, and Oracle. If you want to see a list of components available if you install the providers, you can uncheck the Show Only Installed Source Types option. By only offering the one SQL Server choice, Integration Services is making your life easier by directing you to the correct provider immediately. When you select the Source Assistant's SQL Server type, it will use the OLE DB connection manager, which is also your go-to connection manager!

Once you select the SQL Server type, you have the option of selecting an existing connection manager from the pane on the right or creating a new one. Creating a new connection manager will take you through the exact steps you looked at previously for setting up an OLE DB connection manager.

Finally, pick your new or existing connection manager and click the OK button. This will create the connection manager and add the SQL Server source to the Data Flow task. You can then immediately pick up your development by creating and optimizing the SQL query. The Source Assistant is a great way to get started with developing your Integration Services package, especially if you are new to Integration Services. If you know that you want to use one of the other types of connections, you can just create the connection manager and source directly, without having to use the Source Assistant. Either way, you have a few ways to start your development as quickly as possible.

Summary

At this point, we described why you would want to use certain SQL Server sources over others, how to set up the source, and how to clean up the source query to get the best performance out of your package. We also covered sources in general to set up the rest of the source chapters.

Although all of the principles described in this chapter are patterns for SQL Server, you can apply many of them to other source types as well. Be sure to review the rest of the source chapters for patterns that you can use for SQL Server in addition to what we have already discussed.



Data Correction with Data Quality Services

Data Quality Services (DQS) was first introduced in SQL Server 2012. It provides data correction and data deduplication functionality—key components for most *Extract, Transform, and Load* (ETL) processes. This chapter describes how DQS integrates with SSIS and provides patterns that enable you to achieve reliable, low-effort data cleansing within your ETL packages.

■ **Note** The Data Quality Services product requires some manual steps post-installation to create the DQS databases and set default permissions. See the “Install Data Quality Services” page in Books Online for more information: <http://msdn.microsoft.com/en-us/library/gg492277.aspx>.

Overview of Data Quality Services

The data cleansing and matching operations you perform with DQS revolve around the use of a knowledge base. A knowledge base (or KB) is made up of one or more domains. An example domain for doing address cleansing would be City, State, or Country. Each of these fields would be a separate domain. Two or more related domains can be grouped together to form a composite domain (or CD). Composite domains allow you to validate multiple fields as a single unit. For example, a Company composite domain could be made up of Name, Address, City, State, and Country domains. Using a composite domain would allow you to validate that Microsoft Corporation (Name) exists at One Redmond Way (Address), Redmond (City), WA (State), USA (Country). If the DQS KB has all of the relevant knowledge, it would be able to flag the entry as incorrect if you had Las Vegas as the City—even though Las Vegas is a valid city name, the knowledge base has defined that the Microsoft office is located in Redmond.

Data Quality Services has three main components: the client utility (shown in Figure 5-1), which allows you to build and manage your knowledge bases; an SSIS data flow transform for bulk data cleansing; and a server component, where the actual cleansing and matching takes place. The DQS server is not a standalone instance. It is essentially a set of user databases (DQS_MAIN, DQS_PROJECTS, DQS_STAGING_DATA) with a stored procedure-based API, much like the SSIS Catalog that was introduced in SQL Server 2012.

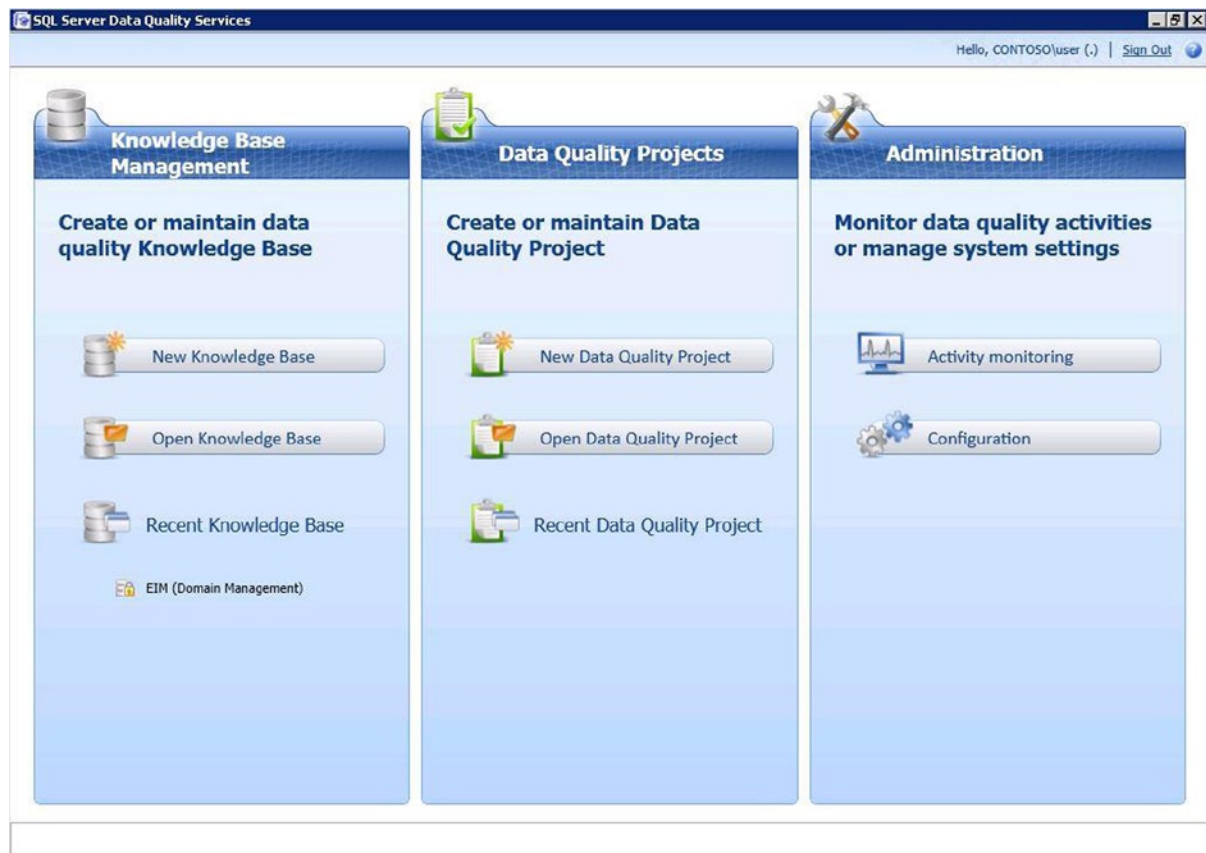


Figure 5-1. The Data Quality Client application

Using the Data Quality Client

The Data Quality Client application is used to build and manage your knowledge bases. It can also be used as a standalone tool for cleansing data. The tool is targeted toward data stewards and IT professionals who own and manage data within your organization. Users of the tool will fall into three different roles (shown in Table 5-1), which map to roles within the main DQS database. The functionality you can access through the tool will depend on what role you are currently assigned to.

Table 5-1. *Data Quality Services Roles*

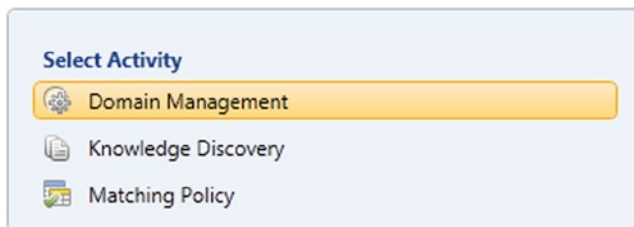
Name	SQL Role	Description
DQS KB Operator	dqs_kb_operator	User can edit and execute an existing data quality project.
DQS KB Editor	dqs_kb_editor	User can perform project functions and create and edit knowledge bases.
DQS Administrator	dqs_administrator	User can perform project and knowledge functions, as well as administer the system.

■ **Note** Members of the `sysadmin` role on the SQL Server instance on which DQS is hosted have the same level of permissions as a DQS Administrator by default. It is recommended that you still associate users with one of the three DQS roles.

Knowledge Base Management

The options under the Knowledge Base Management section allow you to create and maintain your knowledge bases. When creating a new knowledge base, you have the option to create an empty knowledge base, or to base it on an existing knowledge base, which will prepopulate the new knowledge base with the domains from the original. Knowledge bases can also be created from a DQS file (.dqs extension), allowing you to back up or share knowledge bases across systems.

You'll perform three main activities when interacting with your knowledge bases through this UI (shown in Figure 5-2). These activities are available after you've created a new knowledge base or have opened an existing one.

**Figure 5-2.** *Knowledge base management activities*

When doing Domain Management, you can verify and modify the domains within the knowledge base. This includes changing domain properties (shown in Figure 5-3), configuring online reference data, as well as viewing and modifying rules and values. You also have the option to export the knowledge base or individual domains to a DQS file, as well as to import new domains from a DQS file.

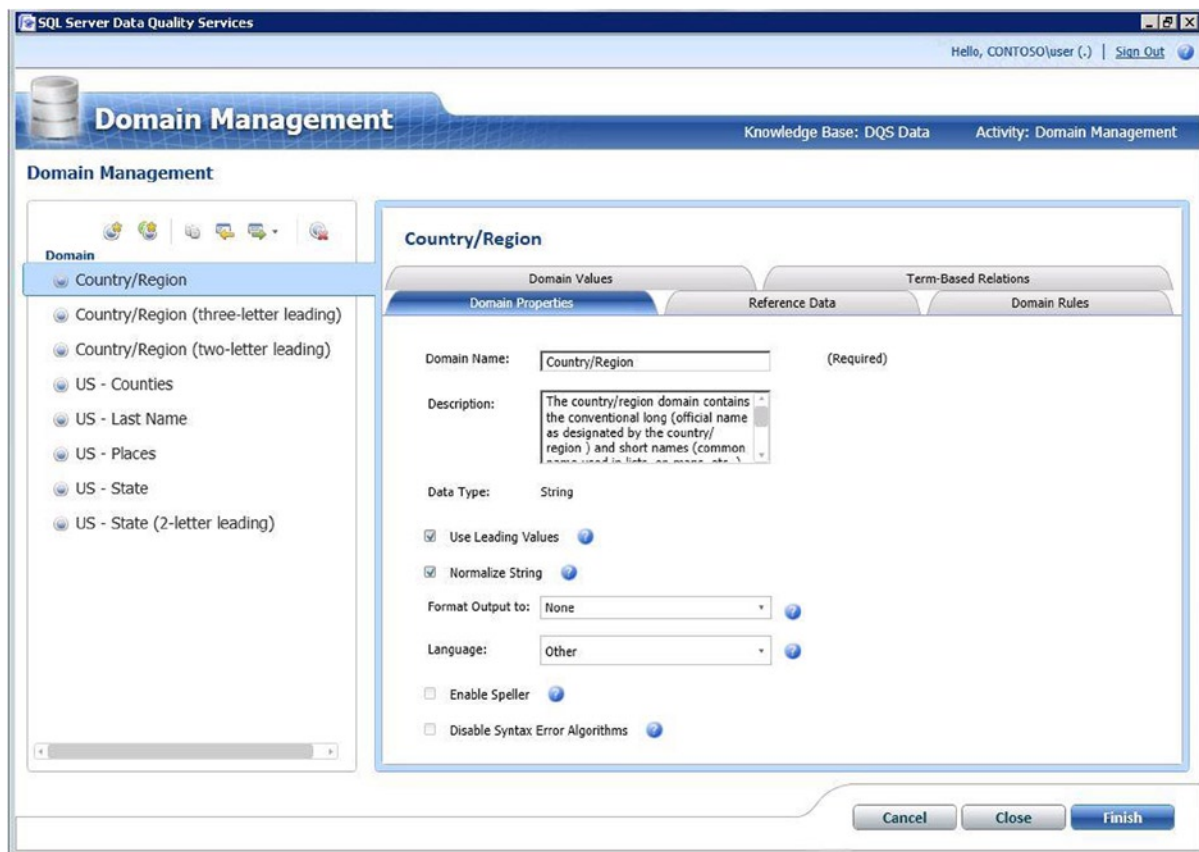


Figure 5-3. The Domain Properties tab of the Domain Management activity in the DQS Client

Knowledge Discovery is a computer-assisted process to build knowledge base information. You supply source data (from a SQL Server table or view or Excel file), and map the input columns to knowledge base domains. This data will be imported into DQS and stored as a set of known domain values.

The Matching Policy activity is used to prepare DQS for the data deduplication process. From this UI, a data steward can create a policy that contains one or more matching rules that DQS will use to determine how rows of data should be compared.

Data Quality Projects

A data quality project is one where you interactively cleanse or match your data. You'll select the source of your data (SQL Server or an Excel file, which you can upload through the client), and then map source columns to domains within your knowledge base. Figure 5-4 shows a data quality project that will attempt to cleanse the EnglishCountryRegionName and CountryRegionCode columns against domains from the default DQS knowledge base.

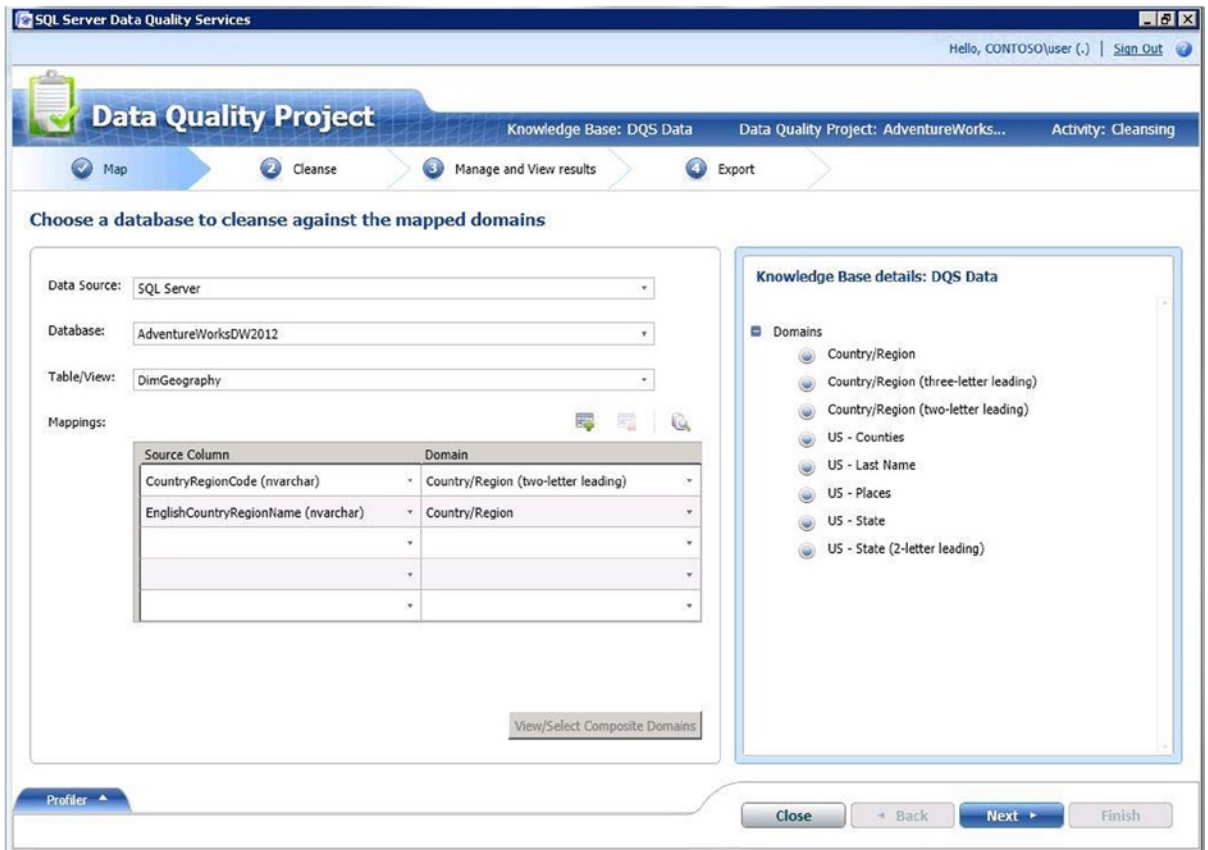


Figure 5-4. Creating a new data cleansing project

After you've mapped your columns to domains, DQS will process your data and provide you with the results of the cleansing operation. When you review the results, you have the option to approve or reject certain corrections, add new values to the list of known domain values, and specify correction rules. For example, as the data steward for your organization, you know that "Jack Ryan" and "John Ryan" are the same person. After approving the corrections, you can export the results to a SQL Server table, Excel file, or CSV file. DQS does not give you the option to correct the value in-place—you will need a separate process to update the original source data you examined.

At various times during the process you can save your data quality project. The project status is saved to the DQS server, allowing you to resume at a later point. This is especially useful when working with large sets of data that can take awhile to scan. It also allows you to come back to the correction results in case you need to do some research on what the correct values should be for a particular domain.

To manage your active data quality projects, click on the Open Data Quality Project button on the home page of the client. From here, you can see all projects that are currently in progress. Right clicking on a project gives you management options, such as renaming the project or deleting it if it is no longer needed.

Administration

The Administration section is available to users in the DQS Administrator's role. From here, you can monitor all activity on the DQS server (such as Domain Management and Cleansing projects), and set system wide configuration options. From these pages, you can set logging levels for various operations, as well as set the minimum confidence scores for suggestions and automatic corrections. If you are using online reference data from the Windows Azure Marketplace, you will configure your account information and service subscriptions from this page as well (as shown in Figure 5-5).

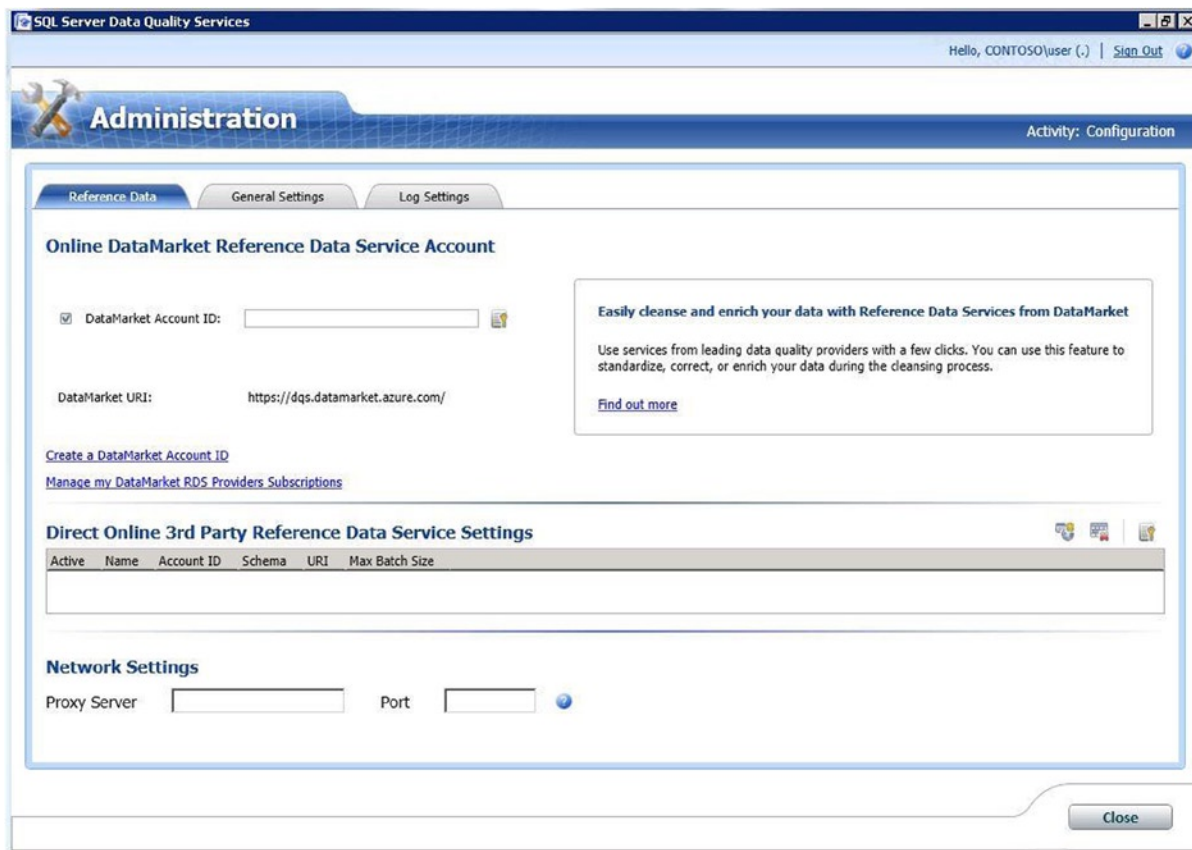


Figure 5-5. Configuration for online reference data in the Windows Azure Marketplace

Using the Default Knowledge Base

DQS comes with a default knowledge base containing domains related to cleansing and validation of countries and locations within the United States. Figure 5-6 shows the domain values for the “US – State” domain. In this figure, you can see that “Alabama” has synonyms defined for it—it will automatically correct “AL” to “Alabama,” and mark “Ala.” as an error.

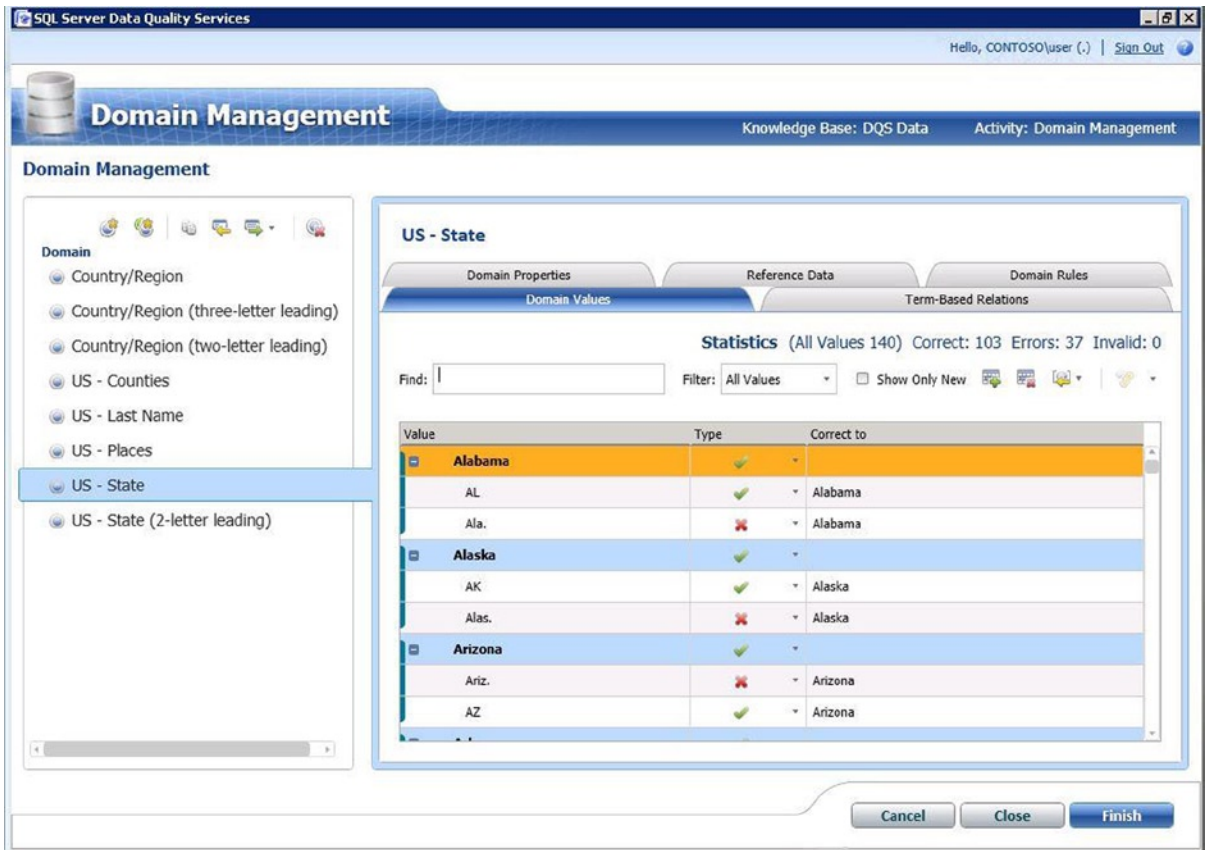


Figure 5-6. The US – State domain from the default DQS knowledge base

Online Reference Data Services

DQS has two types of data it will use to perform cleansing and matching operations; *local data* and *reference data*. Local data make up the values shown on the Domain Values page in the DQS Client—these are known values that are imported into DQS as part of the Knowledge Discovery process. The values are stored along with the knowledge base in the DQS_MAIN database. If these values change, you must update your domain with the new values. Reference data is not stored in the knowledge base—it is queried from an online reference data service. Using online reference data may impact performance, because your cleansing process will need to call out to an external system, but it requires less maintenance since you don't need to worry about keeping values in sync.

The online, Reference Data Services (RDS) that can be linked to your domains are configured on the Administration page in the DQS Client. There are two types of data providers: DataMarket providers from the Windows Azure Marketplace, and Direct Online third party providers. DataMarket providers require that you have a Windows Azure Marketplace Account ID and subscription to the data set you wish to use. The Direct Online provider option allows you to point to other third party web services that support the DQS provider interface.

Using DQS with SSIS

SQL Server 2014 comes with the DQS Cleansing transform for SSIS. This section describes how to configure and use this transform as part of your SSIS data flow. This section also contains information on a few open source extensions for DQS, which are available on CodePlex.

DQS Cleansing Transform

The DQS Cleansing transform is included with SSIS 2014 and can be found in the Data Flow toolbox (shown in Figure 5-7). It will appear under the Other Transforms section of the toolbox by default.

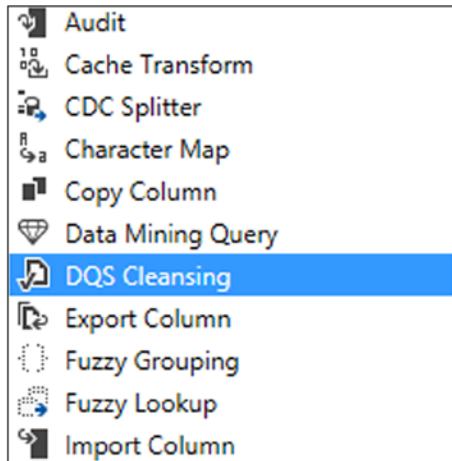


Figure 5-7. The DQS Cleansing transform

After adding the DQS Cleansing transform to your data flow, you can double click the component to bring up its editor UI. The first thing you need to set in the DQS Cleansing Transformation Editor is the Data Quality Connection Manager (as shown in Figure 5-8). This will point to a DQS installation residing on a SQL Server instance. Once the connection manager has been created, you select the knowledge base you want to use. Selecting the knowledge base you want to use will bring up its list of domains.

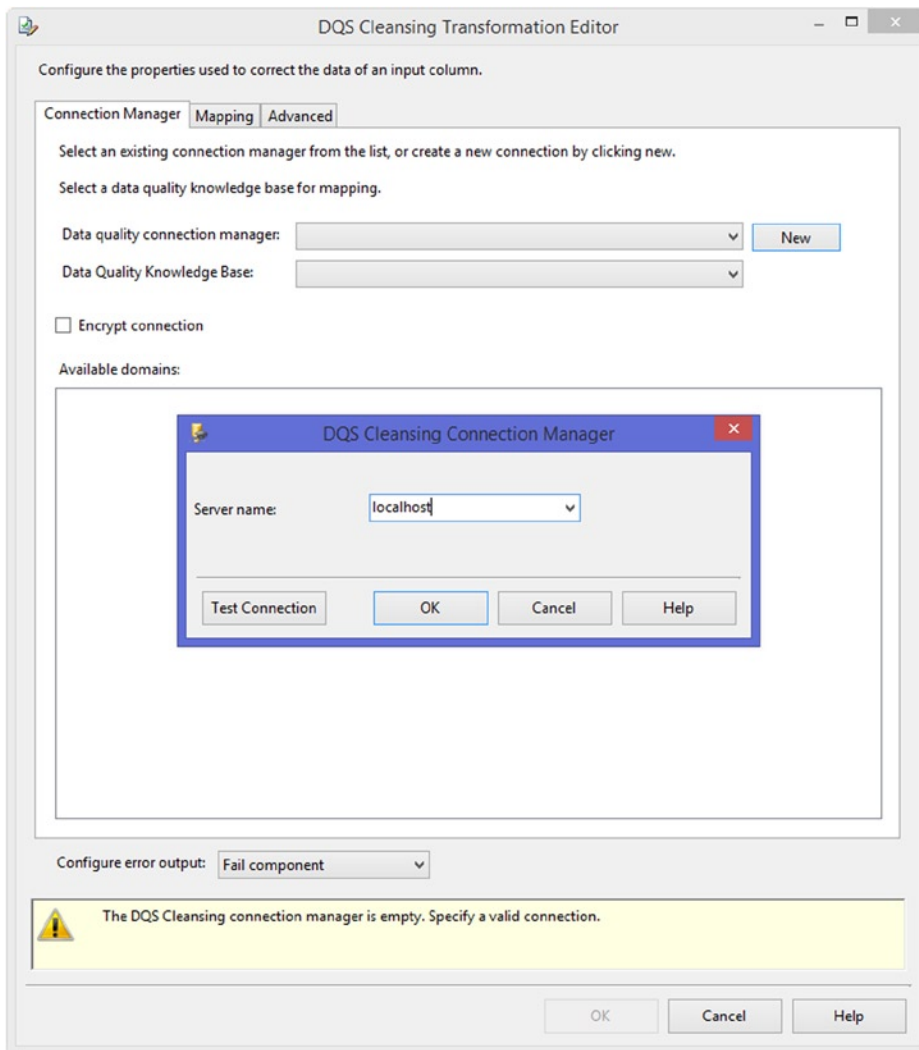


Figure 5-8. The DQS Connection Manager and Cleansing Transformation Editor

As mentioned earlier in the chapter, there are two types of domains in this list; regular domains (e.g., City, State, Zip), and composite domains, which are made up of two or more regular domains. When using the DQS Cleansing transform, you can map columns from your data flow to domains in the knowledge base. You can also make use of composite domains in two ways:

1. **A single (string) column:** For this to work, all values must appear in the same order as the domains do. So using the Company example from the beginning of this chapter, your column values would need to look like this: Microsoft Corporation, One Redmond Way, Redmond, WA, USA.
2. **Multiple columns:** Individual columns are always cleansed by the knowledge and rules stored within the DQS knowledge base. If you map a column to each domain of a composite domain, the row will also be cleansed using the composite domain logic.

Note There is currently no indicator in the DQS Cleansing transform UI to show when you've mapped columns to all domains within a composite domain. You need to double check that each domain is mapped; otherwise, each column will be validated and cleansed individually.

The Mapping tab (Figure 5-9) allows you to select the columns you want to cleanse and map them to domains in your knowledge base. Note that the Domain dropdown will automatically filter out columns with incompatible data types. For example, it won't show domains with a String data type if you are using a DT_I4 (four-byte signed integer) column. A domain can only be mapped once—if you have multiple columns for the same domain, you'll need to use two separate DQS Cleansing transforms in your data flow.

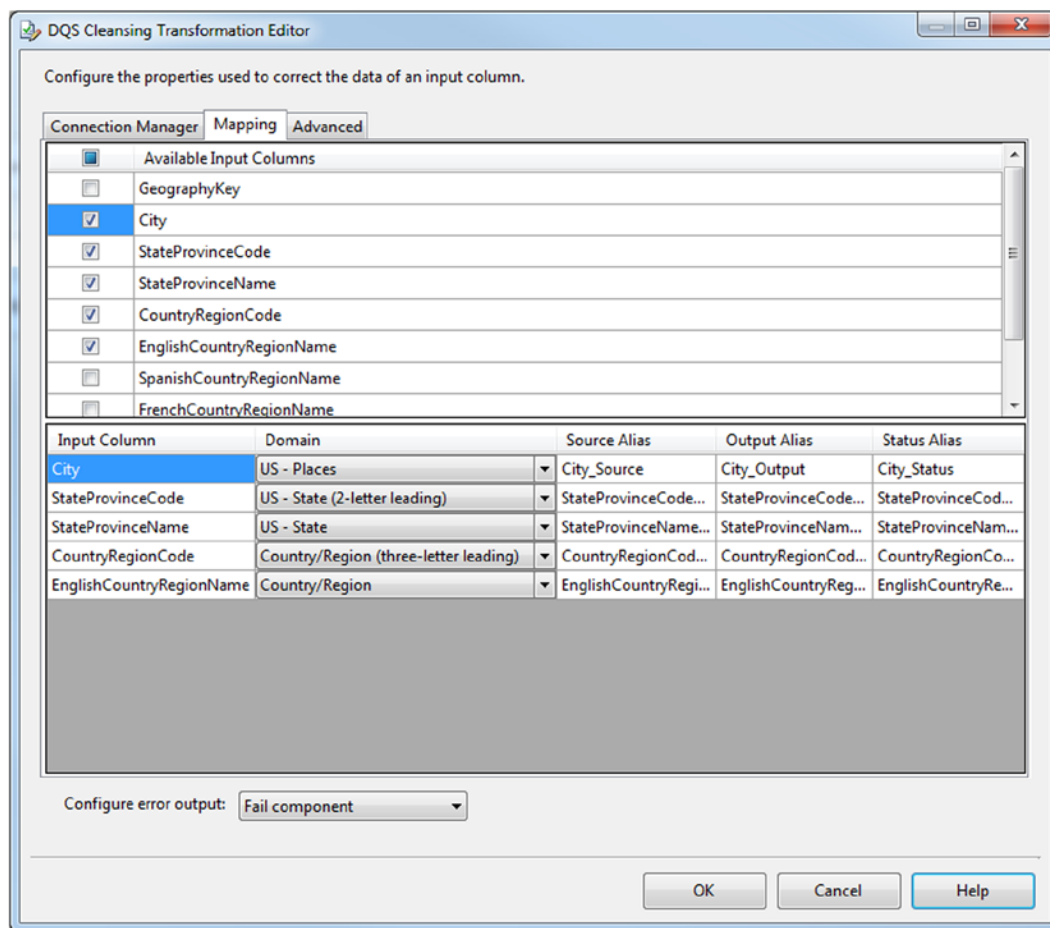


Figure 5-9. Mapping DQS knowledge base domains to columns in your data flow

Note If your data contains multiple columns with values from the same domain, consider using the Linked Domain feature when creating your knowledge base. For more information, see the "Create a Linked Domain" page in Books Online: <http://msdn.microsoft.com/en-us/library/hh479582.aspx>.

Each column you map causes at least three additional columns to be added to your data flow—Source, Output, and Status. More columns may be added, depending on the advanced options you select (more on that to follow). The list of columns created by the DQS Cleansing transform can be found in Table 5-2. Each additional column will be prefixed with the name of the original column by default and can be renamed on the Mapping tab. In addition to the columns that are prefixed with the name of the original, a Record Status column is added to record the overall status of the row. Details on how to handle the columns added by the DQS Cleansing transform are covered later in this chapter.

Table 5-2. *Additional Columns Created by the DQS Cleansing Transform*

Column	Default	Description
Record Status	Yes	The overall status of the record, based on the status of each mapped column. The overall status is based on the following algorithm: If one or more columns is - Invalid, the record status is Invalid. - Auto suggest, the record status is Auto suggest. - Corrected, the record status is Corrected. If all columns are Correct or New, then the record status will be Correct. If all columns are New, then the record status will be New. See Table 5-3 for possible Status column values.
_Source	Yes	This column contains the original value passed to the transform.
_Output	Yes	If the original value was modified during the cleansing process, this column contains the corrected value. If the value was not modified, this column contains the original value. When doing bulk cleansing through SSIS, downstream components will typically make use of this column.
_Status	Yes	The validation or cleansing status of the value. See Table 5-3 for possible values of the Status column.
_Confidence	No	This column contains a score that is given to any correction or suggestion. The score reflects to what extent the DQS server (or the relevant data source) has confidence in the correction/suggestion. Most ETL packages will want to include this field and use a conditional split to redirect values that do not meet the minimum confidence threshold so that they can be manually inspected.
_Reason	No	This column explains the reason for the column's cleansing status. For example, if a column was Corrected, the reason might be due to the DQS Cleansing algorithm, knowledge base rules, or a change due to standardization.
_Appended Data	No	This column is populated when there are domains attached to a reference data provider. Certain reference data providers will return additional information as part of the cleansing—not only values associated with the mapped domains. For example, when cleansing an address, the reference data provider might also return Latitude and Longitude values.
_Appended Data Schema	No	This column is related to the previously listed Appended Data setting. If the data source returned additional information in the Appended Data field, this column contains a simple schema that can be used to interpret that data.

The Advanced tab (as shown in Figure 5-10) has a number of different options, most of which add new columns to the data flow when selected. The Standardize output option is an exception to this. When enabled, DQS will modify the output values according to the domain settings defined in the DQS client application. You can see how the standardization settings are defined in the DQS Client on the Domain Management -> Domain Properties tab (shown earlier in Figure 5-3).

There are two kinds of standardization:

Reformatting operations: These include operations such as conversion to uppercase, to lowercase, and to capitalized words in a string.

Correction to a leading value: For example, if multiple values (or synonyms) are defined for a term, the current value will be replaced with the leading term (as defined in the KB).

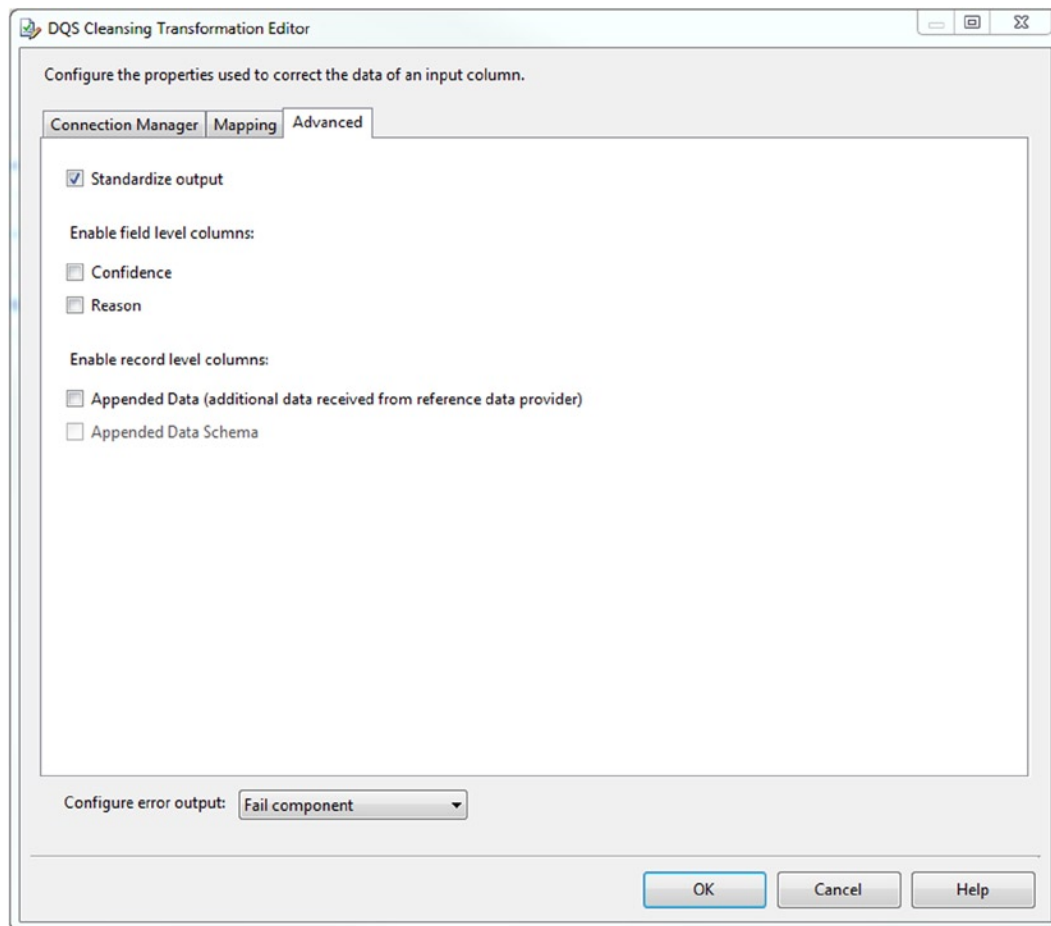


Figure 5-10. Advanced tab of the DQS Cleansing Transformation Editor

The DQS Cleansing transformation logs information events that indicate when it sends rows to the DQS server. There will be one event for each batch, and one event at the end, with a summary for all records. The messages contain details about how long the cleansing process took to process the batch, and the counts for each status. Listing 5-1 shows an example of what these messages look like. The transform processes data in 10,000 row chunks. The chunk size is currently hardcoded; there is no way to configure the size of the batch sent to the DQS server.

Listing 5-1. DQS Cleansing Transform Log Messages

```
[DQS Cleansing] Information: The DQS Cleansing component received 1000 records from the DQS server.
The data cleansing process took 7 seconds.
[DQS Cleansing] Information: DQS Cleansing component records chunk status count - Invalid: 0,
Autosuggest: 21, Corrected: 979, Unknown: 0, Correct: 0.
[DQS Cleansing] Information: DQS Cleansing component records total status count - Invalid: 0,
Autosuggest: 115, Corrected: 4885, Unknown: 0, Correct: 0.
```

DQS Extensions on CodePlex

A number of DQS extensions are available from CodePlex. These are listed in Table 5-3. They do not come with SQL Server 2014, but they are very useful for automating data cleansing scenarios. A brief description on how to use each one is included here. You can find more information on how to use them from their project pages on CodePlex.

Table 5-3. DQS Extensions on CodePlex

Extension	Description
DQS Matching	This transform allows you to do automated data deduplication within an SSIS data flow. It provides similar capabilities as the SSIS Fuzzy Grouping transform but also leverages the DQS matching policy defined within your knowledge base to give more accurate results.
DQS Domain Value Import	This destination component allows you to bulk load values into a DQS domain. It is useful for automation scenarios where your domain values are defined within an external system (such as Master Data Services).
Publish DQS Knowledge Base	This task is used to commit changes to your knowledge base (referred to as <i>publishing</i> in DQS terminology). The task is typically used in conjunction with the DQS Domain Value Import transform.

■ **Note** The CodePlex extensions for DQS were created by OH22 Data (<http://data.oh22.net/>) and are freely available. They are not officially supported by Microsoft. The extensions can be downloaded from <https://ssisdqsmatching.codeplex.com/> and <https://domainvalueimport.codeplex.com/>.

Cleansing Data in the Data Flow

The following section contains design patterns for cleansing data in the SSIS data flow using the DQS Cleansing transform. There are two key issues to keep in mind when cleansing data:

- The cleansing process is based on the rules within your knowledge base. The better the cleansing rules are, the more accurate your cleansing process will be. You may want to reprocess your data as the rules in your knowledge base improve.
- Cleansing large amounts of data can take a long time. See the “Performance Considerations” section later in this chapter for patterns that you can apply to reduce overall processing time.

Handling the Output of the DQS Cleansing Transform

The DQS Cleansing transform adds a number of new columns to the data flow (as described earlier in this chapter). The way you’ll handle the processed rows will usually depend on the status of the row, which is set in the Record Status column. A Conditional Split transformation can be used to redirect rows down the appropriate data flow path. Figure 5-11 shows what the Conditional Split transformation would look like with a separate output for each Record Status value. Table 5-4 contains a list of possible status values.

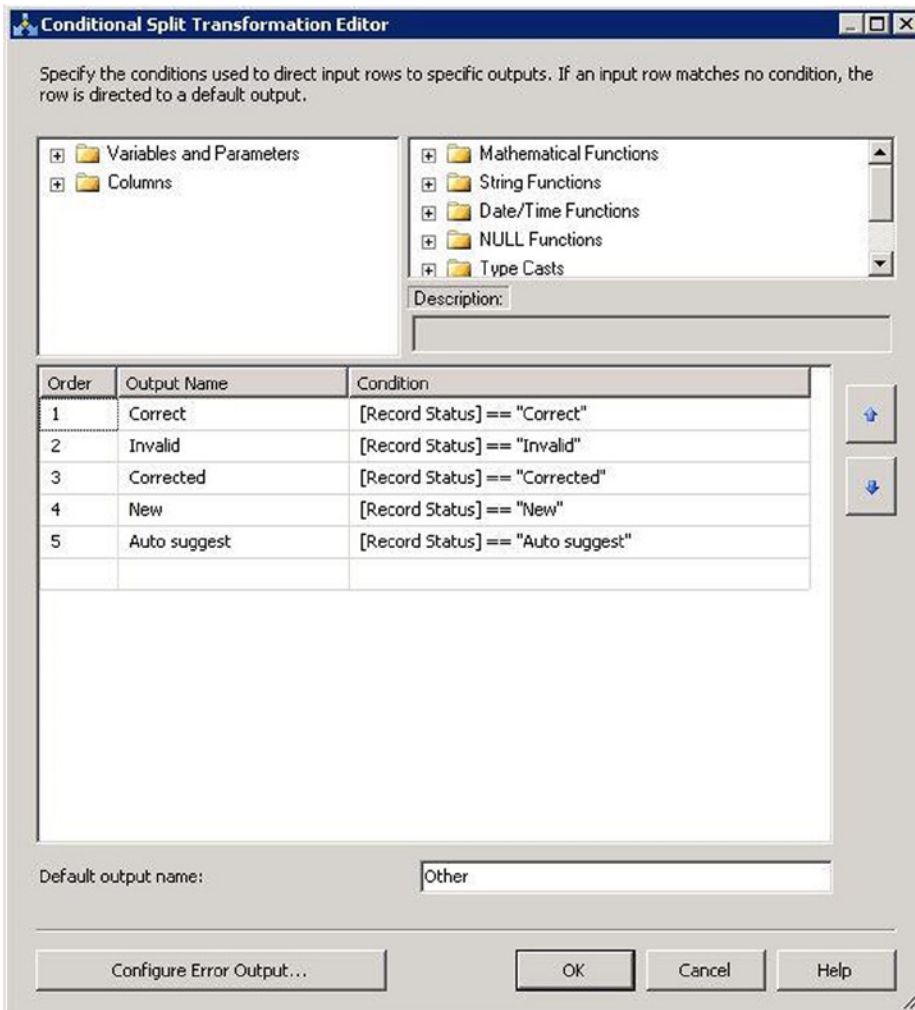


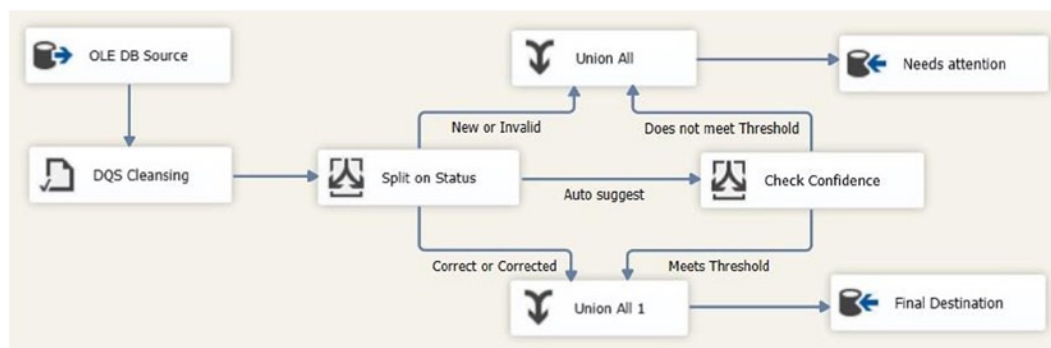
Figure 5-11. Conditional Split transformation configured to process the DQS Record Status

Table 5-4. Column Status Values

Option	Description
Correct	The value was already correct and needs no further modification. The Corrected column will contain the original value.
Invalid	The domain contained validation rules that marked this value as invalid.
Corrected	The value was incorrect, but DQS was able to correct it. The Corrected column will contain the modified value.
New	The value wasn't in the current domain and did not match any domain rules. DQS is unsure whether or not it is valid. The value should be redirected and manually inspected.
Auto suggest	The value wasn't an exact match, but DQS has provided a suggestion. If you include the Confidence field, you could automatically accept rows above a certain confidence level and redirect others to a separate table for later review.

■ **Note** The column status values are localized; the actual string will change depending on the language of your SQL Server installation. This might require you to add additional processing logic to your Conditional Split expressions if you expect your packages to run under different system locales. For more information about the status values, see the “Data Cleansing” page in books online: <http://msdn.microsoft.com/en-us/library/gg524800.aspx>.

The status values you handle and the downstream data flow logic you use will depend on the goals of your data cleansing process. Typically, you will want to split your rows into two paths. Correct, Corrected, and Auto suggest rows will go down a path that will update your destination table with the cleansed data values (found in the <column_name>_Output column). New and Invalid rows will usually go into a separate table so that someone can examine them later on and either correct the data (in the case of Invalid rows), or update the knowledge base (in the case of New rows) so that these values can be handled automatically in the future. You may wish to include a check against the confidence level (<column_name>_Confidence) of the Auto suggest rows to make sure it meets a minimum threshold. Figure 5-12 shows an SSIS data flow with logic to process rows from the DQS Cleansing transform.

**Figure 5-12.** Data Flow processing logic following a DQS Cleansing transform

■ **Note** Although the Confidence column's output by the DQS Cleansing transforms are numeric, they are output as `DT_WSTR(100)` columns (strings). To check the confidence level against a minimum threshold, you'll need to cast the value to a `DT_R4` (float) or `DT_R8` (double).

Performance Considerations

Data cleansing can be a CPU and memory intensive operation and may take some time to complete. Domains that rely on online reference data services may round trip incoming data to the Azure Data Marketplace, which will have a further impact on the time it takes to cleanse your data. As a result, when processing large amounts of data, you will typically want to reduce your dataset before passing it through the DQS Cleansing transform.

The DQS Cleansing transform sends incoming data to the DQS server (running within a SQL Server instance), where the actual cleansing operations are performed. Although this may offload a lot of the work being done by the SSIS machine, there may be some overhead in sending the data across the network to another server. Another thing to note is that the DQS Cleansing transform is an asynchronous component, which means it makes copies of data flow buffers at runtime. This can further impact the performance of your data flow and is another reason for only passing through the rows that need to be cleansed.

The following sections describe some package design tips that can be used to improve overall performance when cleansing data with the DQS Cleansing transform.

Parallel Processing

The DQS Cleansing transform sends its rows to the DQS server one batch at a time. This single threaded approach isn't ideal if you have a lot of spare CPU power on your system, so designing your packages in a way that allows DQS to send multiple batches to the server in parallel will give you a performance boost. You have two main options for parallel processing. First, you can split the incoming rows down multiple paths and have a separate DQS Cleansing transform on each path performing the same set of work. If your data set has a key or row that can be easily split using SSIS Expressions, you can use a Conditional Split transform. Otherwise, you can consider using a third party component like the Balanced Data Distributor. The second approach is to design your data flow in such a way that multiple instances of your package can be run in parallel. For this approach to work, you will need to partition your source query so that it pulls back a certain key range, and each instance of the package will work on a different range. This approach gives you a bit more flexibility, because you can dynamically control how many package instances you run in parallel by playing with the key ranges.

■ **Note** You might find that the DQS Client performs its cleansing operations faster than the DQS Cleansing transform in SSIS. This is because the client processes multiple batches in parallel by default, whereas the DQS Cleansing transform processes them one at a time. To get the same performance in SSIS as you do in the DQS Client, you'll need to add your own parallelism.

Tracking Which Rows Have Been Cleansed

You can track which rows have already been cleansed and when the cleansing operation was performed. This allows you to filter out rows that have already been cleansed so you don't need to process them a second time. By using a date value for this marker, you can also determine which rows need to be reprocessed if your knowledge base gets updated. Remember, as your knowledge base changes and your cleansing rules improve, you will get more accurate results each time data is processed by the DQS Cleansing transform.

To track when a row has been cleansed, add a new `datetime` column to your destination table (`DateLastCleansed`). A `NULL` or very early date value can be used to indicate that a row has never been processed. Alternatively, you can track dates in a separate table, linked to the original row with a foreign key constraint. Your SSIS package will contain the following logic:

1. Retrieve the date the DQS knowledge base was last updated using an Execute SQL task. This value should be stored in a package variable (`@[User::DQS_KB_Date]`).
2. Inside of a Data Flow task, retrieve the data to be cleansed with the appropriate source component. The source data should contain a `DateLastCleansed` column to track when the row was last processed with the DQS Cleansing transform.
3. Use a Conditional Split transform to compare the DQS knowledge base date against the date the row was last processed. The expression might look like this: `[DateLastCleansed] < @[User::DQS_KB_Date]`. Rows matching this expression will be directed to a DQS Cleansing transformation.
4. Handle the cleansed rows according to their status.
5. Use a Derived Column transform to set a new `DateLastCleansed` value.
6. Update the destination table with any corrected values and the new `DateLastCleansed` value.

Filtering Rows with the Lookup Transform

You can reduce the number of rows you need to cleanse by validating the data with a faster data flow component, such as the Lookup transform. Using one or more Lookup transforms, you can check if values exist in a reference table using quick, in-memory comparisons. Rows that match existing values can be filtered out. Rows with values that aren't found in the reference table can then be sent to Data Quality Services for cleansing. Prefiltering rows this way means you won't be able to take advantage of the standardized formatting that DQS provides, and this makes it difficult to do complex validation that involves relationships between multiple fields. This approach works best when you are working with a small number of unrelated fields that don't require any special formatting as part of the cleansing process.

To use this pattern, your data flow will use the following logic:

1. Retrieve the data containing the fields to be cleansed using a source component.
2. Set the component to Ignore failure when there are no matching entries.
3. Add a Lookup transform for each field you are going to cleanse. Each Lookup transform will use a SQL query that pulls in a unique set of values for that field and a static Boolean (bit) value. This static value will be used as a flag to determine whether the value was found. Since you are ignoring lookup failures, the flag value will be `NULL` if the lookup failed to find a match. Listing 5-2 shows what the query would look like for the `CountryRegionCode` field, coming from the `DimGeography` table.

Listing 5-2. Sample Lookup Query for the CountryRegionCode Field

```
SELECT DISTINCT CountryRegionCode, 1 as [RegionCodeFlag]
FROM [dbo].[DimGeography]
```

4. On the Columns tab, map the field to the related lookup column, and add the static flag value as a new column in your data flow (as shown in Figure 5-13).

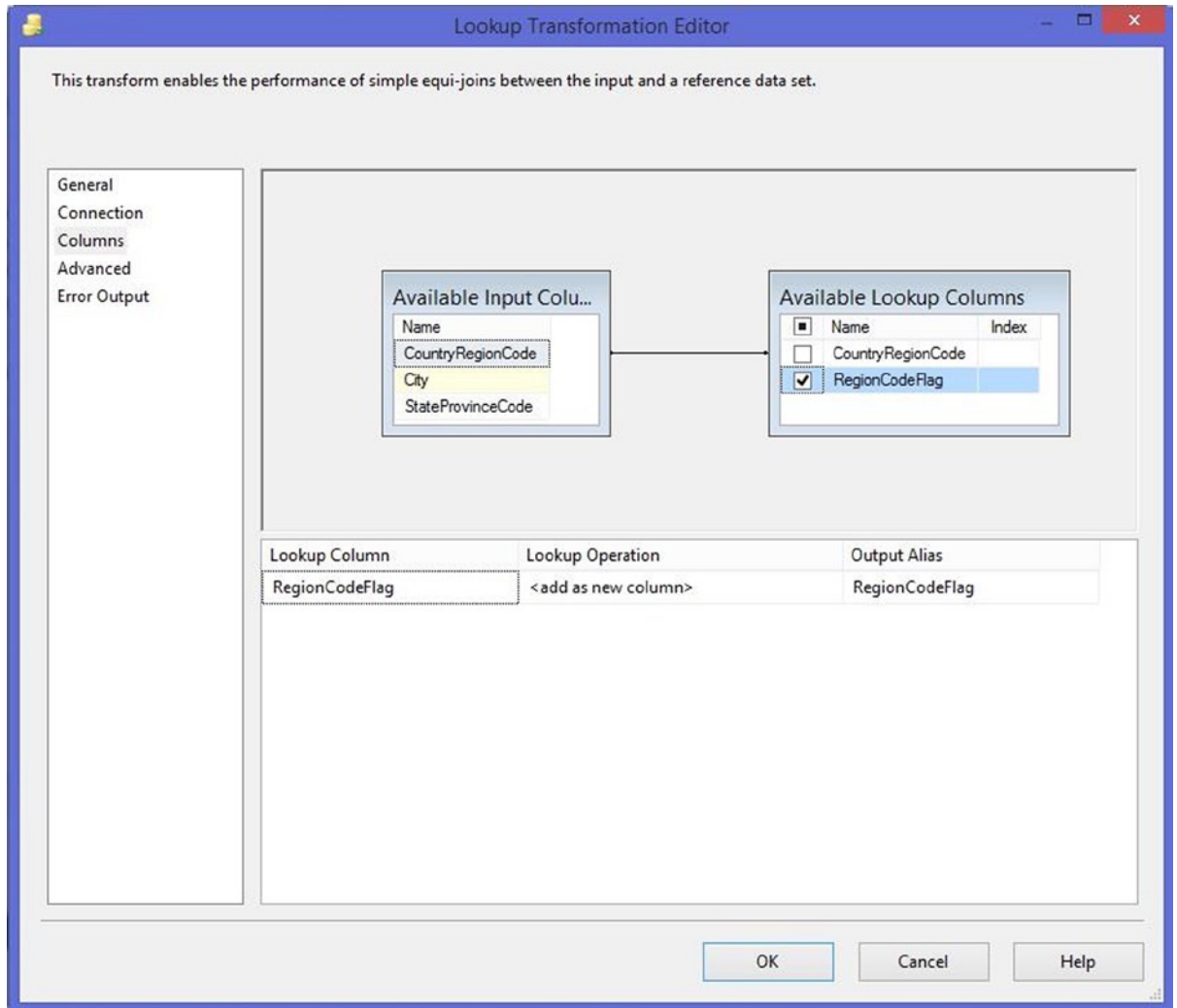


Figure 5-13. Column mapping for the Lookup transform

5. Repeat steps 3-4 for each field you will be cleansing. The Lookup transforms should be connected using the Lookup Match outputs.
6. Add a Conditional Split transform with a single expression that checks each of the flag fields. If any of the flag fields are NULL, the row should be sent to DQS for proper cleansing. For example, the expression to check the RegionCodeFlag for a NULL value would be `ISNULL([RegionCodeFlag])`.
7. Connect the Conditional Split output you created to the DQS Cleansing transform. Rows going to the Conditional Split's default output can be ignored (since their values were successfully validated using the Lookup transforms).
8. Complete the rest of the data flow based on the appropriate logic for handling the output of the DQS Cleansing transform.

Figure 5-14 shows a screenshot of a data flow to cleanse a single field using the preceding logic.

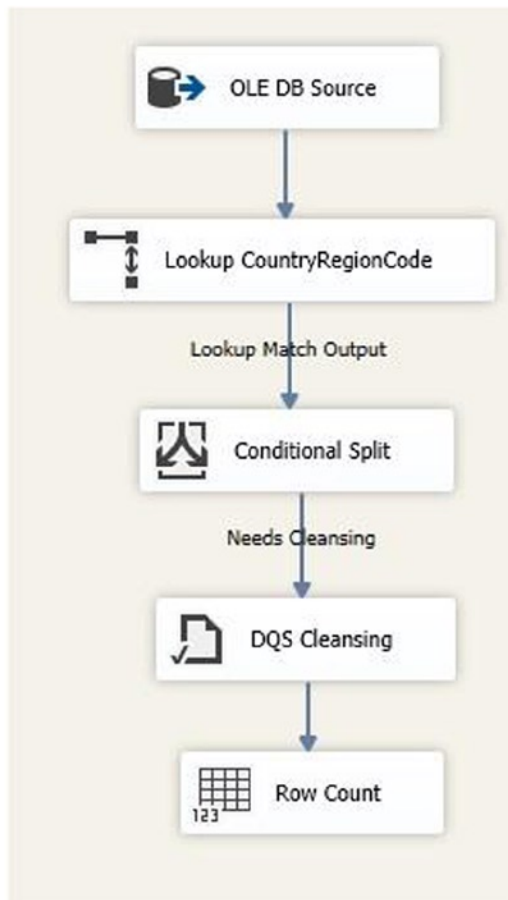


Figure 5-14. Completed data flow using Lookup transforms to prefilter rows

■ **Note** This approach works especially well when looking up key fields that are part of an entity in Master Data Services (MDS), another product that ships with SQL Server 2014. Using an MDS Subscription View, you can expose your dimension as a view that can be queried by a Lookup transform. For more information about Master Data Services, see the books online entry: <http://msdn.microsoft.com/en-us/library/ee633763.aspx>.

Approving and Importing Cleansing Rules

When a data flow with a DQS Cleansing transform is run, a cleansing project is created on the DQS server. This allows the KB editor to view the corrections performed by the transform and approve or reject rules. A new project is created automatically each time the package is run and can be viewed using the DQS client. When performing parallel cleansing with multiple DQS Cleansing transforms in a single data flow, a project will be created for each transform you are using.

Once correction rules have been approved by the KB editor, they can be imported into the knowledge base so they can be automatically applied the next time cleansing is performed. This process can be done with the following steps:

1. Run the SSIS package containing the DQS Cleansing transform.
2. Open the DQS client, and connect to the DQS server used by the SSIS package.
3. Click on the Open Data Quality Project button.
4. The newly created project will be listed on the left hand pane (as shown in Figure 5-15). The project's name will be generated using the name of the package, the name of the DQS Cleansing transform, a timestamp of when the package was executed, the unique identifier of the Data Flow task that contained the transformation, and another unique identifier for the specific execution of the package.

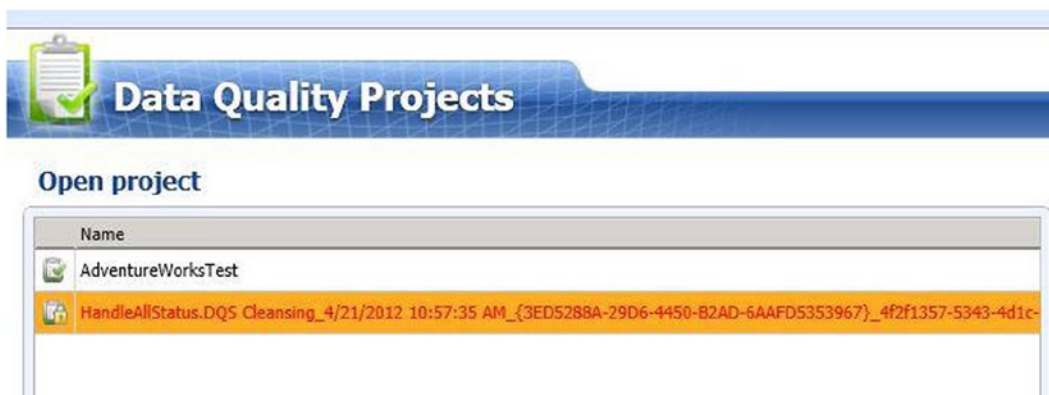


Figure 5-15. Running the DQS Cleansing transform will create a project on the DQS server

5. Selecting the project name will display details in the right hand pane (shown in Figure 5-16), such as the domains that were used in this cleansing activity.

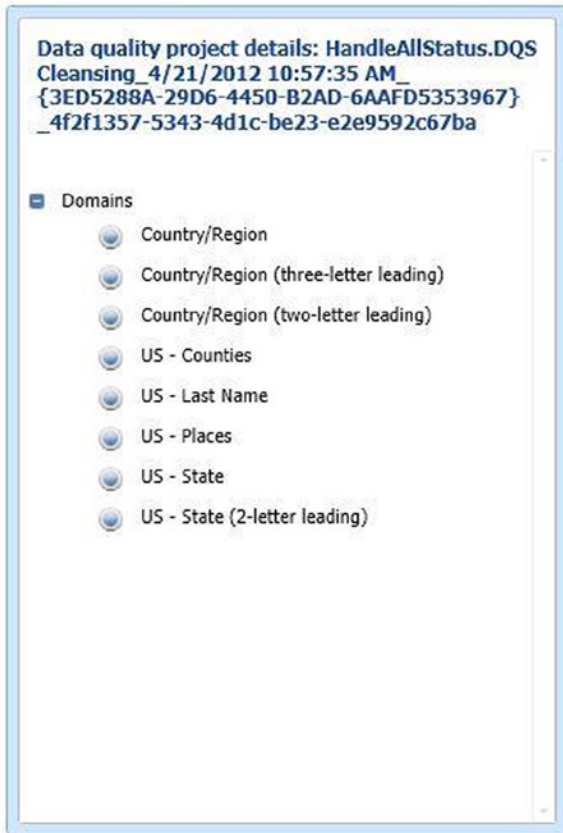


Figure 5-16. Selecting the project will display the domains used in this activity

6. Click Next to open the project.
7. Select the domains for which you would like to review the corrections.
8. Click the Approve or Reject radio buttons for each correction, or change the Correct to value for the entry.
9. Click the Next button when you have finished with the rules.
10. (Optional) Export the corrected data to SQL Server, CSV, or Excel. You will be able to skip this step in most scenarios, since your SSIS package will be responsible for handling the corrected data.
11. Click the Finish button to close the project.
12. From the home screen, select your knowledge base, and choose the Domain Management activity.

13. Select the domain you have defined new rules for.
14. Click the Domain Values tab.
15. Click the Import Values button, and select Import project values (as shown in Figure 5-17).

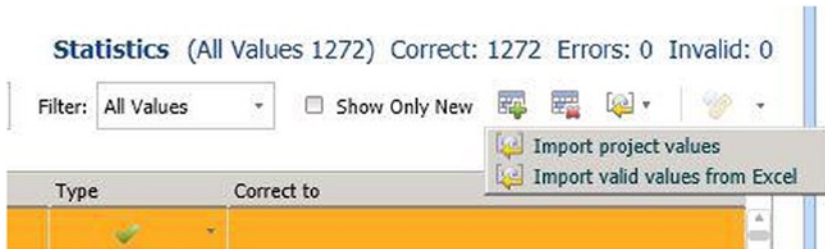


Figure 5-17. *Importing domain values from an existing project*

16. Repeat steps 13-15 for each domain you wish to update.
17. Click the Finish button to publish your knowledge base changes.
18. If you have modified any of the correction rules, you may want to re-run your SSIS package to pick up the new values.

Conclusion

This chapter described the new DQS Cleansing transform and how you can use it to take advantage of the advanced data cleansing functionality provided by Data Quality Services in SQL Server 2014. The design patterns detailed in this chapter will help you get the best possible performance while doing data cleansing with SSIS.



DB2 Source Patterns

In a previous chapter you learned about patterns related to SQL Server sources. In this chapter, we will move on to patterns that relate to sourcing data from the IBM DB2 database. DB2 describes a variety of databases, so it is essential to learn about the different databases we will discuss, as well as how to use each database as an Integration Services source.

As we described in Chapter 4, setting up a source entails four different objects: Connection Manager, Provider, Source Component, and a Source Component Query. While this remains true for the DB2 database, you need to take the additional first step of determining what type of database you own. DB2 has a number of types, providers, and ways to query data. As we look at the different patterns associated with each of these components, picture how they will work with other sources as well. Combining these steps will put you on the right path to pulling data from your DB2 database.

This chapter highlights patterns that you may find useful while connecting to a DB2 database, but it does not cover every possible scenario that you may run into in your environment.

DB2 Database Family

There are several different types of DB2 databases available on the market today. How you connect to the database depends on the DB2 version. DB2 separates its products into three groups:

DB2 for i: This version has gone through multiple names over the years, including DB2 for AS/400, iSeries, System I, and Power Systems. DB2 is included in this server, so people commonly refer to this version when they think of DB2.

DB2 for z/OS: This DB2 version is the main database available for the z/OS platform and is only available in 64-bit mode.

DB2 for LUW: This version of DB2 is a later addition to the DB2 family. The Linux, UNIX, and Windows (LUW) version comes in multiple editions, depending on the purpose of your database instance. More information on these editions can be found on IBM's website.

The different product types affect how you query data from Integration Services. As we walk you through setting up your connection, we will point out some of the differences you need to be aware of based on the product type. The first thing you need to do is pick a provider to use in your connection manager.

Selecting a DB2 Provider

The first step in pulling data from DB2 is to select a provider that can be used in your environment. There are two steps to accomplishing this:

1. Find the database version.
2. Pick the provider vendor.

Find the Database Version

The first step in selecting your DB2 provider is to learn what version you own. Combining the version information with the product type will help you choose what provider to use. If you're not sure what type of server you're working with, you have a couple of options. The first option is to use a DB2 administration tool to check the properties of your instance. For example, if you use Control Center, you can right-click on the instance name, and click the About menu option. This will show you something similar to Figure 6-1.

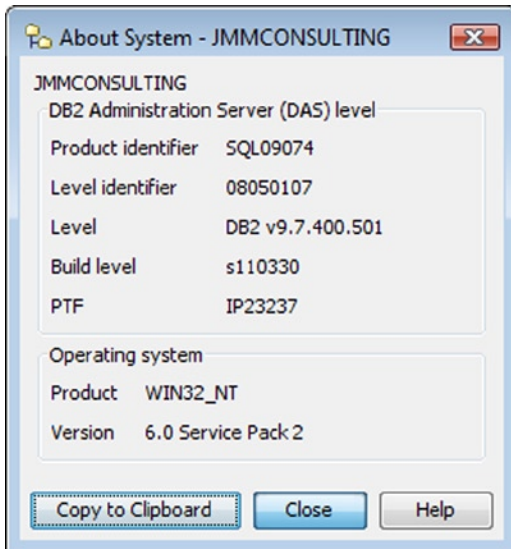


Figure 6-1. Control Center About System window showing database version and information

If you don't have access to connect directly to the instance, you can run a query against the database instead to pull the same information. A sample query that shows this information can be seen in Listing 6-1; the results are shown in Figure 6-2.

Listing 6-1. Sample Query to Show Database Version and Information

```
SELECT inst_name
, release_num
, service_level
, bld_level
, ptf
, fixpack_num
FROM TABLE (sysproc.env_get_inst_info())
```

Commands Query Results Access Plan					
Edits to these results are performed as positioned UPDATES and DELETES. Use the Tools Settings notebook to change the form of editing.					
INST_NAME	RELEASE_NUM	SERVICE_LEVEL	BLD_LEVEL	PTF	FIXPACK_NUM
DB2	08050107	DB2 v9.7.400.501	s110330	IP23237	4

Figure 6-2. Query results showing database version and information

Pick Provider Vendor

Although it is possible to use ODBC and ADO.NET to connect to a DB2 database, we will focus on OLE DB providers in this chapter to ensure that we can use the connection for all transformations. Here are two of the more common providers and when you would use each one.

IBM OLE DB Provider for DB2: IBM produces their own OLE DB provider, which can be used in applications such as Integration Services. This provider can be used for all versions and the latest products.

Microsoft OLE DB Provider for DB2 Version 4.0: Microsoft created a provider that uses OLE DB to connect to DB2. This provider can be used for all versions of DB2. See the latest documentation for which product numbers it supports.

Don't forget to make sure you've selected either the 32-bit or 64-bit version, based on the database server. Also ensure that the database version matches the supported version and product for the provider you want to use. We recommend using the provider most often used in your organization to facilitate ease of development and maintenance. If you are trying a provider for the first time, try the different versions to see what works best for you, since the performance and security discrepancies may vary per environment.

Connecting to a DB2 Database

For this chapter, we'll use the Microsoft OLE DB Provider for DB2. No matter what provider you choose, the next step is to make a connection to the DB2 database. To do this, you need to create a connection manager, select the correct provider, and fill out the appropriate server information.

Once you download your desired provider, you will install it on the server where you will develop and execute your Integration Services packages. If the provider has installed correctly, you can see it by opening up the Source Assistant. A correctly installed provider can be seen in Figure 6-3.

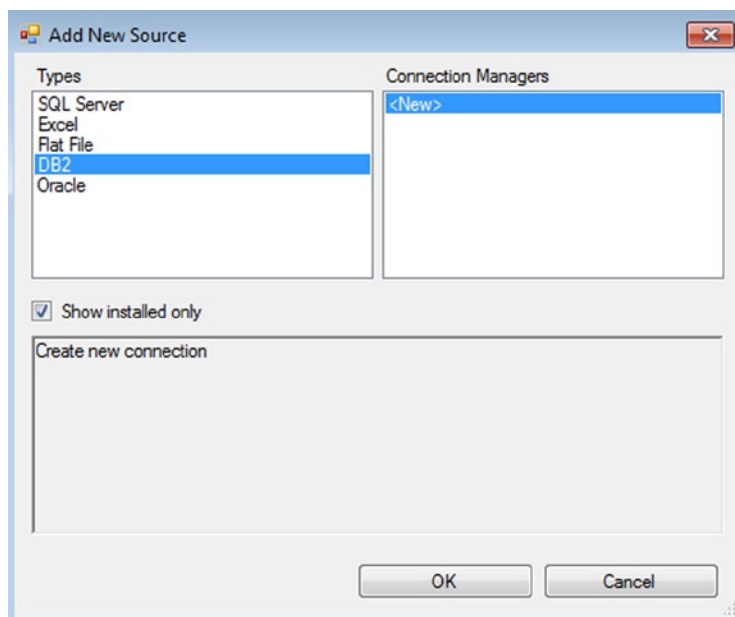


Figure 6-3. Source Assistant's Add New Source window

Begin by creating a shared OLE DB connection manager in the Solution Explorer of your package. In the provider drop-down list at the top of the Connection Manager window, change the provider to Microsoft OLE DB Provider for DB2, as seen in Figure 6-4.

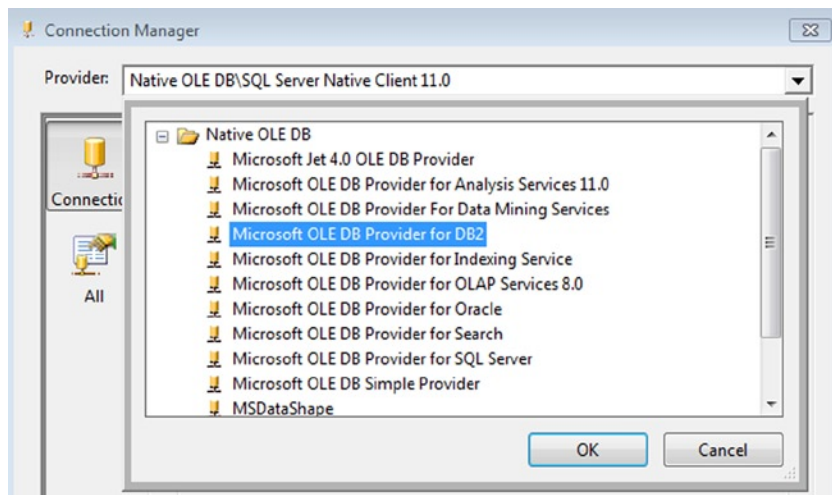


Figure 6-4. Connection Manager window Provider list

Next, you can add the name of the database instance, the correct authentication method, and the database you want to connect to. If you prefer, you can directly enter a connection string in the Connection property of the source.

■ **Note** If you ever have a question on the correct connection string to use, www.connectionstrings.com is a great resource to answer your question.

In addition to telling Integration Services how to connect to the DB2 database, you also need to tell Integration Services how to view the data. To store data, databases use an encoding scheme and character codeset. These are the two encoding schemes that you need to understand:

ASCII: The American Standard Code for Information Interchange is a 7-bit encoding scheme that contains 128 printable and non-printable characters.

EBCDIC: The Extended Binary Coded Decimal Interchange Code was created by IBM and is an 8-bit encoding scheme used in their mainframe servers.

Both DB2 for i and DB2 for z/OS use the EBCDIC encoding scheme, and DB2 for LUW uses the ASCII encoding scheme. Typically, the EBCDIC schemes use the codeset number 37, and the DB2 for LUW uses the ANSI-1252 codeset. Using the Microsoft OLE DB Provider for DB2, the next step is to modify the codeset for whichever version you are using.

Begin by clicking on the Data Links button next to the OLE DB Provider name in the Connection Manager, seen in Figure 6-5.

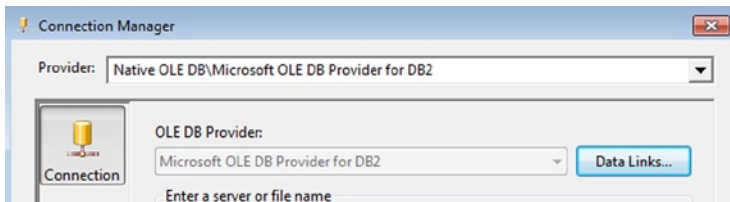


Figure 6-5. *Data Links button on the Connection Manager window*

The Data Link Properties window should open. On the Advanced tab, under the Host CCSID property, you can use the default value of EBCDIC - U.S./Canada [37] or change it to ANSI - Latin I [1252], as shown in Figure 6-6. In addition, you may find it necessary to check the Process Binary as Character option if you are seeing output that looks like data type names rather than your data.

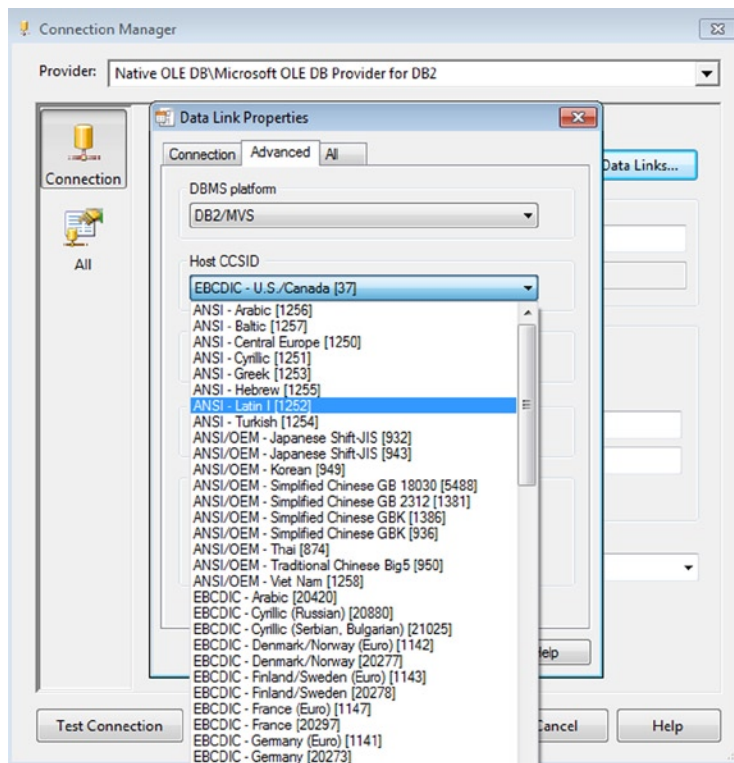


Figure 6-6. Data Link Properties window with Host CCSID list

Querying the DB2 Database

The final set of DB2 source patterns covers querying the DB2 database. Because the Integration Services package uses an OLE DB provider, it will also need an OLE DB source component. As with any other database, the source component should point to the DB2 connection manager already created. Once the package successfully connects to the database, it is time to query the database.

■ Note A number of companies provide alternatives to the Integration Services connection managers and source components. They provide a different interface and different functionality than the OLE DB source component. If you need additional functionality, such as EBCDIC to ASCII conversion, see aminoSoftware's Lysine EBCDIC source.

All source component queries are written in whatever brand of SQL the database uses. DB2's RDBMS-specific language is called SQL PL, and PL/SQL can also be used for later versions. If you receive an error message about syntax, be sure that your syntax matches the guidelines found on IBM's website: <http://publib.boulder.ibm.com/infocenter/db2luw/v9r7/index.jsp?topic=/com.ibm.db2.luw.apdv.plsql.doc/doc/c0053607.html>.

In certain cases, you may want to use parameters to limit the data returned from the database. Let's take a look at parameterizing your queries now.

DB2 Source Component Parameters

An important part of writing source queries is that doing so allows you to filter the data that enters the pipeline. There are a number of reasons why you would want to do this, including loading data incrementally, reusing the same package for different departments, or reducing the amount of data that is run at one time. When using the Microsoft OLE DB Provider for DB2, you need to set the Derive Parameters property to True. You find this property in the connection manager, by clicking the Data Links button, and on the resulting Data Link Properties window, choosing the All tab. When you click on Derive Parameters in the properties list, an Edit Property Value window appears, as shown in Figure 6-7.

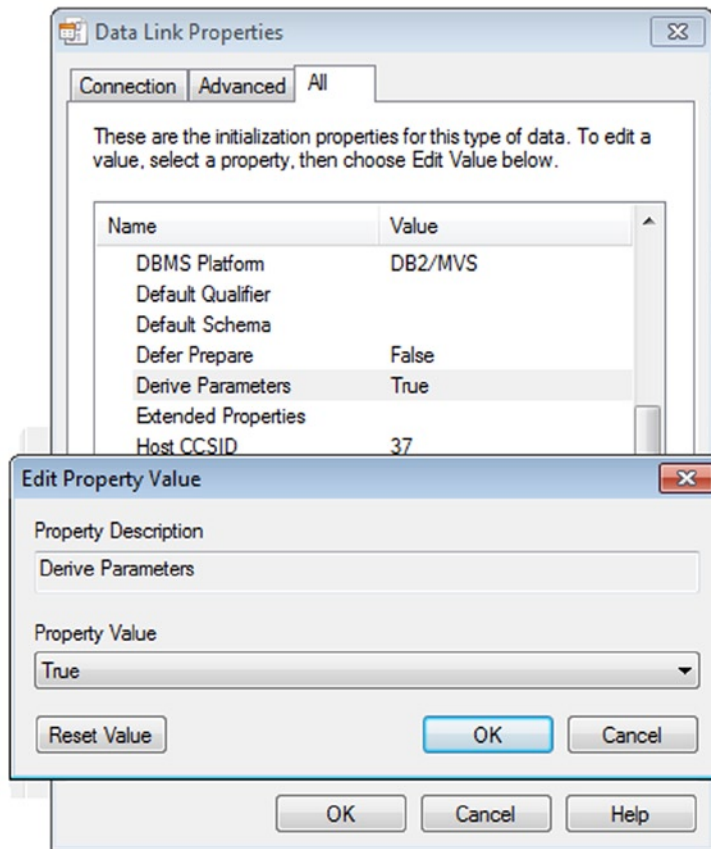


Figure 6-7. Edit Property Value window on the Data Link Properties window

Once the Derive Parameters property is set, you will write your query using question marks, similar to those in Listing 6-2. Put the query in the SQL command in the source component.

Listing 6-2. Sample Query to Illustrate DB2 Parameters

```
SELECT col1
      , col2
      , col3
FROM tab1
WHERE col4 = ?
```

Be sure to click the Parameters button next to the query in the source to assign variables to each parameter that you set. It is important that the list of variables in the Parameters window match the correct order in the query.

There are some scenarios where using query parameters won't work. Let's look at when you can't use query parameters and what to do instead.

DB2 Source Component Dynamic Queries

Parameterized queries will not work if the content of the source query needs to be changed for any reason. The table, schema, or column names could change as part of the query content. A typical example of this in DB2 is when you have different schemas in each environment. To fix this, we set an expression on a variable and use the variable in the SQLStatement property. Let's walk through the steps of setting this up.

Begin by creating two string variables: environment and query. Set the following properties on the query variable:

```
EvaluateAsExpression: True
Value: "select col1, col2 from" + @environment + ".tab1"
```

■ **Note** In Integration Services's editions prior to SQL Server 2012, expressions had a limit of 4000 characters. This restriction is now removed, allowing you to create strings as long as you need them to be.

In the OLE DB source component, change the query type to SQL Command from Variable, and pick the query variable that you just selected, as shown in Figure 6-8.

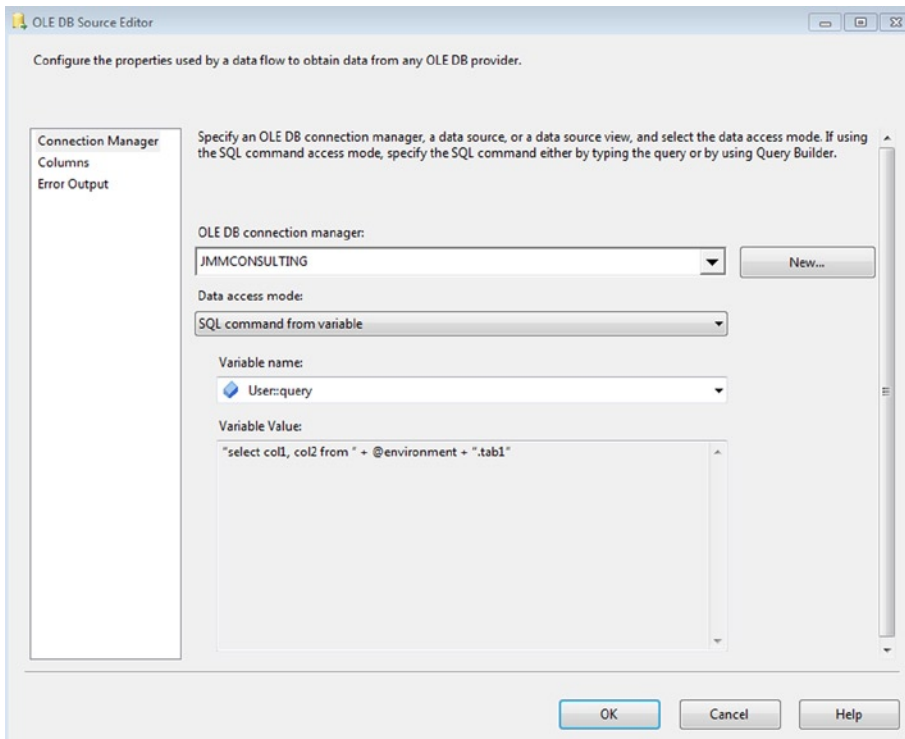


Figure 6-8. OLE DB Source Editor with dynamic query properties

When this package runs, use a package parameter to pass in the correct environment schema name. The expression on the query variable will be set to the new query and execute correctly. Make sure to set the `ValidateExternalMetadata` property on the OLE DB source to `False` to ensure that the package will validate successfully.

Summary

This chapter has covered many of the patterns necessary to connect to the different types of the IBM DB2 database. You've learned how to determine what type of DB2 database you own, how to pick the appropriate provider, and different ways of querying the database. Note that sometimes organizations go a different route when dealing with DB2: exporting the data into a file and then loading the file using SSIS. This is a perfectly valid option and might make sense for you if you have network latencies or problems with connectivity to your DB2 database. If you decide to go this route instead, you can learn how to load the data using Flat File source patterns, which we will discuss in Chapter 7.



Flat File Source Patterns

A common way to transfer data between systems is to export the source data to a flat file and then import the contents of this file into the destination database. Flat files come in all shapes, sizes, and types. There are no row-length limitations. File size is limited by the maximum size allowed by the operating system. When examining flat file types, there are two initial considerations: file format and schema. Common file formats of flat file sources include these:

- Comma-separated values (CSV)
- Tab-delimited file (TDF)
- Fixed-width file

In a flat file, as in a database, schema includes columns and data types. Schema options also allow for more exotic file format options such as “ragged right” and “variable-length rows” flat files.

In this chapter, we’ll examine a common pattern for loading a vanilla flat file source into SQL Server; then we’ll expand that pattern to load a variable-length row flat file source. We will next examine creating and consuming flat file header rows, which are found in some flat file formats. Finally, we will construct an extremely useful SSIS design pattern: Archive File.

Flat File Sources

Let’s begin with a simple flat file source. You can copy and paste the data below into a text editor such as Notepad and save it as `MyFlatFile.csv`:

```
RecordType,Name,Value
A,Simple One,11
B,Simple Two,22
C,Simple Three,33
A,Simple Four,44
C,Simple Five,55
B,Simple Six,66
```

The column names are in the first row. This is convenient, but you don’t always get column names in the first row—or anywhere inside a source flat file, for that matter.

Before leaving the setup phase of our demo project, let’s create a database named `StagingDB` that we can use as a destination. I use the re-executable T-SQL script in Listing 7-1 to create the `StagingDB` database.

Listing 7-1. Create StagingDB

```

use master
go

If Not Exists(Select name
              From sys.databases
              Where name='StagingDB')
begin
    print 'Creating StagingDB database'
    Create Database StagingDB
    print 'StagingDB database created'
end
Else
    print 'StagingDb database already exists.'
```

Execute this script on a server you can use for testing and development. Now you're all set to begin building the demo!

Moving to SSIS!

Open SQL Server Data Tools - Business Intelligence (SSDT-BI) and create a new SSIS project named **Chapter 7** and rename the initial package to **Chapter7.dtsx**. Drag a Data Flow task onto the control flow canvas and double-click it to open the Editing tab.

There are a couple ways to create a flat file source for the data flow: you can use the Source Assistant or you can expand Other Sources in the Data Flow SSIS Toolbox and configure a Flat File source adapter. Let's utilize the latter method: drag a Flat File source adapter from the Data Flow toolbox onto the data flow canvas and open the editor. Figure 7-1 shows the Connection Manager page for the Flat File Source Editor.

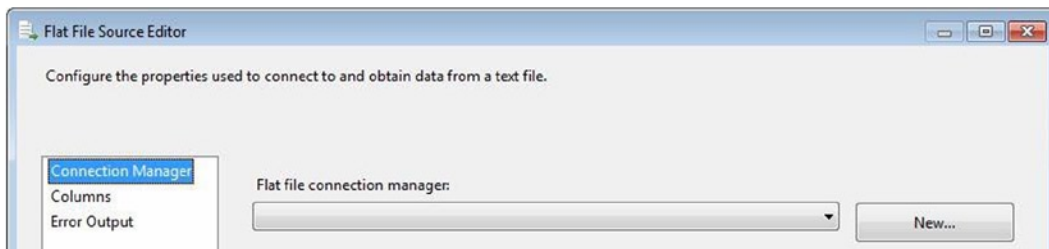


Figure 7-1. Flat File Source Editor Connection Manager configuration

Since there are no connection managers defined in this new SSIS project, click the New button beside the Flat File Connection Manager drop-down to open the Flat File Source Editor. On the General page, name the connection manager **My Flat File**. Click the Browse button beside the File Name textbox and navigate to the location where you saved **MyFlatFile.csv**.

Be aware that, by default, the File Name textbox shows only .txt files. Change the filter to *.csv in order to see and select **MyFlatFile.csv**.

As shown in Figure 7-2, check the Column Names in the First Data Row checkbox:

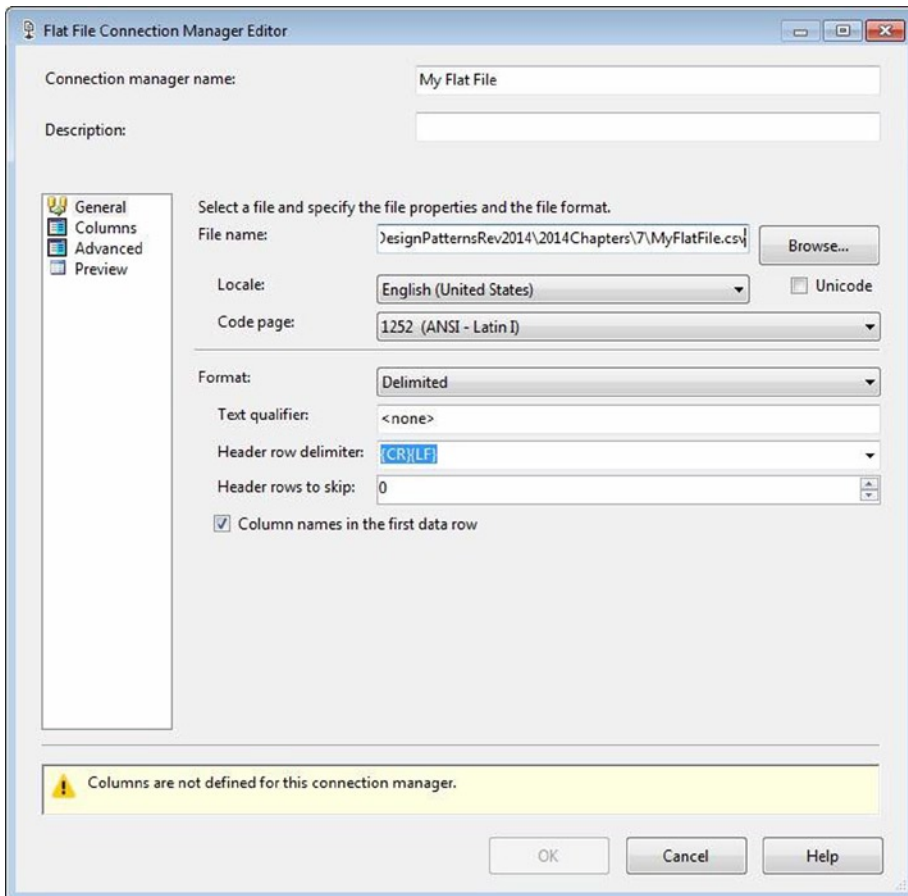


Figure 7-2. Configuring the My Flat File connection manager

Note the warning near the bottom of the window: Columns Are Not Defined for This Connection Manager. For a properly formatted simple CSV file, SSIS now has enough information to read the schema of the flat file and complete the mapping for the connection manager. Figure 7-3 shows the Columns page used to define the column and row delimiters.

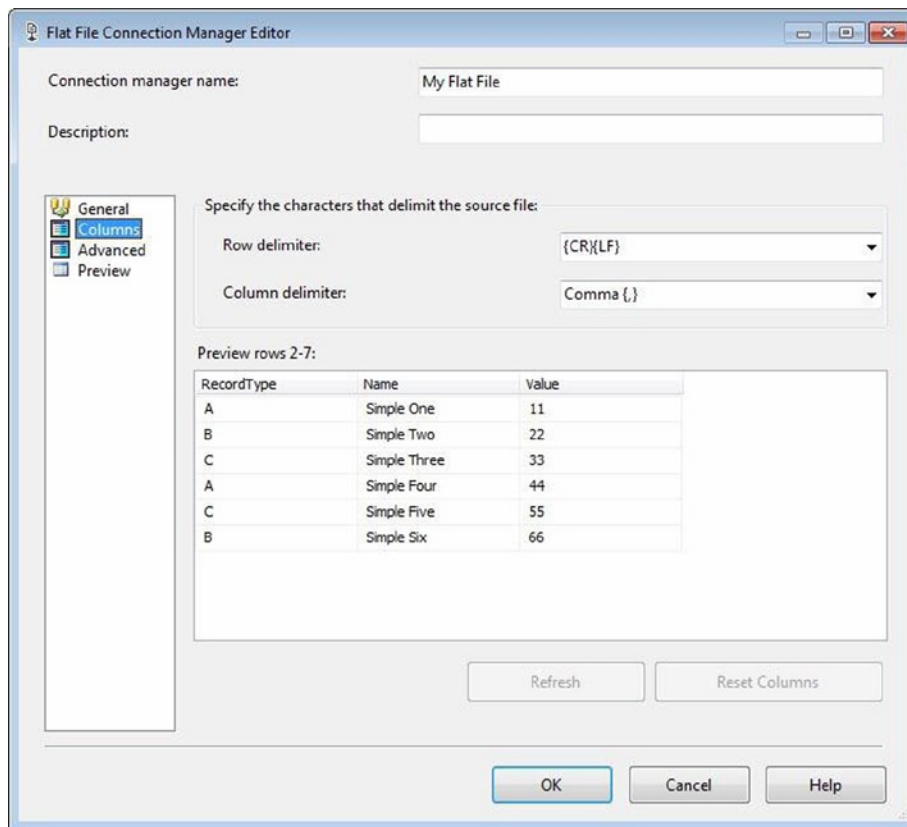


Figure 7-3. The Flat File Connection Manager Editor's Columns page

All data in a flat file are text by default.

Click the OK button to close the Flat File Connection Manager Editor, and then click the OK button to close the Flat File Source Editor.

Strong-Typing the Data

Why would you want to use strongly typed data? Consider the Value column in our example. Right now, Value is a DT_STR data type but the column contains numeric data. In fact, the numeric data is integers. In SQL Server, the INT data type consumes 4 bytes and covers the range from -2^{31} ($-2,147,483,648$) through $2^{31} - 1$ ($2,147,483,647$) according to Books Online. If you wanted to store the integer value $-2,147,483,600$ as text, this would consume at least 1 byte per character. In this case, that's a minimum of 11 bytes (not counting the commas), and it could be more bytes depending on the data type chosen. Converting these data to the DT_I4 data type allows me to store that value in 4 bytes. As an added bonus, the data are numeric, so sorts on this field will outperform sorts on a string data type.

Let's manipulate the data types provided by the Flat File connection manager and source adapter. Drag a Derived Column transformation onto the data flow canvas and connect a data flow path from the flat file source to the new Derived Column transformation. Double-click it to open the editor.

Expand the Type Casts virtual folder in the SSIS Expression Language functions provided in the listbox in the upper-right section of the Derived Column Editor. Drag a DT_STR type cast into the Expression cell of the first row of the Derived Column grid in the lower section of the editor. The Derived Column column of the grid defaults to "<add as new column>" but allows you to choose to replace the value in any of the rows flowing through the transformation.

You can make changes to the values as rows flow through the Derived Column transformation, but you cannot change the data type (which is precisely what you're going to do here), so you need to add a new column to the data flow. The default derived column name is *Derived Column n* , where n is a one-based array of columns configured in the transformation. Change the default derived column name to **strRecordType**. Return to the Expression cell and complete the DT_STR cast function by replacing the *«length»* placeholder text with the desired length of the field: 1. Next, replace the *«code_page»* placeholder with the number that matches your Window Code Page identifier. For U.S. English, this number is 1252. To complete the configuration, expand the Columns virtual folder in the Available Inputs listbox located in the upper-left section of the Derived Column Transformation Editor, and drag the RecordType column into the Expression cell to the right of the DT_STR cast function that you just configured.

When you click anywhere else in the editor, the logic of the transformation validates the expression. This has been happening all along; the text color was being changed to red in the Expression when an issue was encountered with the state of the expression. When you navigate off the Expression cell now, the expression (DT_STR, 1, 1252) [RecordType] should pass muster. The text should return to black to indicate a valid expression.

You can similarly create additional columns with casting expressions to manipulate the data types of the other fields moving through the data flow. Figure 7-4 shows how my example looks when I've completed editing the Derived Column transformation.

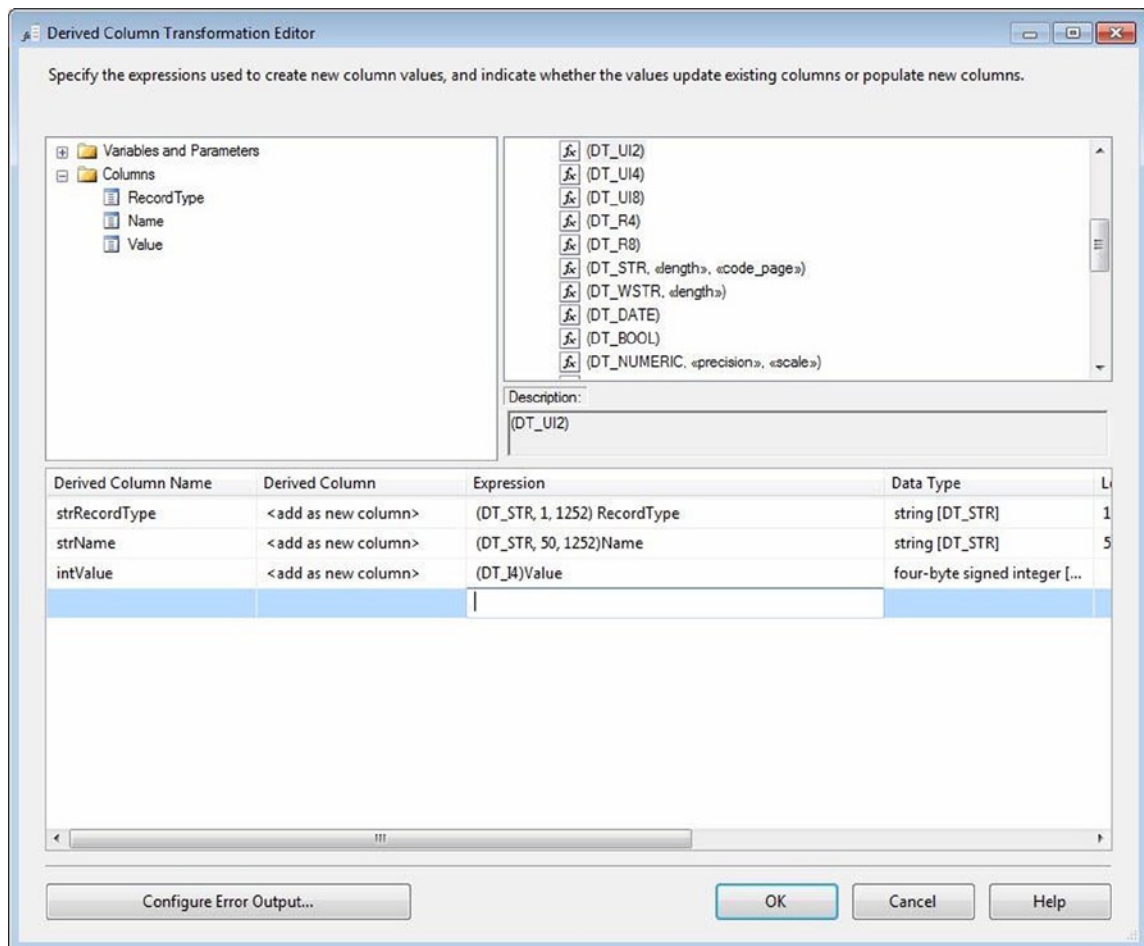


Figure 7-4. Derived Column transformation, configured

Introducing a Data-Staging Pattern

Data staging is an important concept. Every ETL developer has thoughts and opinions about the best way to stage data, and each thinks their way is best! (This is as it should be ... we want and need confident ETL developers). In my opinion, the data integration requirements drive the amount and type of staging.

For flat files, copying all the data into staging tables represents one pattern. Once the data are captured in a query-able format (a relational database), they can be manipulated and shaped by numerous transformations before they are loaded into the destination data warehouse or data mart.

Beyond flat files, staging supports a key tenet of the Extraction phase of any ETL solution: impact the source system-of-record as little as possible. Often, an important business driver for building an ETL solution in the first place is the difficulty of querying data in the system-of-record for reporting purposes. ETL's first goal is similar to that of the Hippocratic Oath: *"Primum non nocere"* (First, do no harm).

Staging requirements for some ETL lend themselves to storing a copy of all source data, whether from flat files or not. Other requirements allow for applying some transformative logic prior to staging. Which is the correct answer? "It depends." In my opinion, the very act of copying data from a text source and landing it in a relational database represents a transformation.

This, then, becomes a pattern of staging data: copying data straight from a flat file into a database. To that end, let's complete the example we've started.

Drag an OLE DB Destination adapter onto the data flow canvas and connect a data flow path from the Derived Column transformation to the OLE DB destination. Before you proceed, double-click on the data flow path to open its editor, and then click on the Metadata page. You'll see something that looks like Figure 7-5.

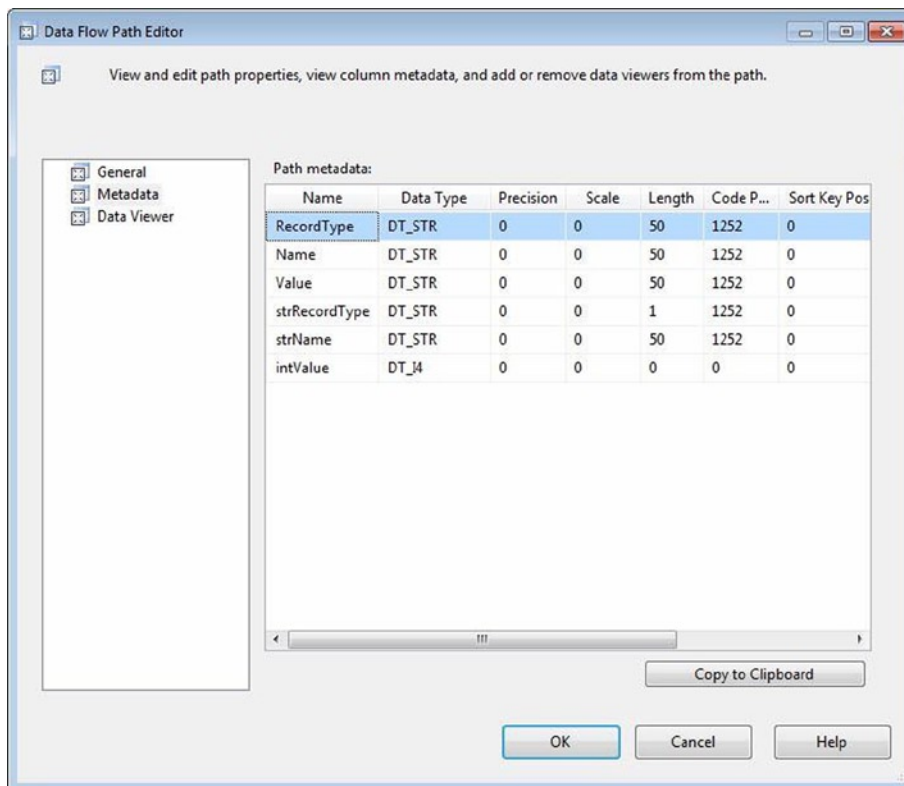


Figure 7-5. Inside the data flow path

I often describe the buffers inside the data flow as “table-ish.” It’s an adjective I made up, but it fits. This peek under the hood of a data flow path is evidence. We’ll come back to this diversion shortly. Click OK to close the Data Flow Path Editor.

Rename the OLE DB destination adapter **FlatFileDest**. Open the OLE DB Destination Editor and click the New button beside the OLE DB Connection Manager drop-down to configure an OLE DB connection manager. When the Configure OLE DB Connection Manager window displays, click the New button to create a new OLE DB connection manager. Add the name of your testing and development server/instance (the same server/instance you used earlier to create the StagingDB database) in the Server Name drop-down. In the Select or Enter a Database Name drop-down, select StagingDB. Click the OK button to complete the OLE DB connection manager configuration, and click the next OK button to select this new connection manager for use with the OLE DB destination adapter. Set the Data Access Mode property to Table or View – Fast Load and accept the default properties configured. Click the New button beside the Name of the Table or the View drop-down. The Create Table window displays the T-SQL data definition language (DDL) statement shown in Listing 7-2.

Listing 7-2. The CREATE TABLE Statement Generated from the Data Flow Path Metadata

```
CREATE TABLE [FlatFileDest] (
    [RecordType] varchar(50),
    [Name] varchar(50),
    [Value ] varchar(50),
    [strRecordType] varchar(1),
    [strName] varchar(50),
    [intValue] int
)
```

The name of the table the OLE DB destination is going to create is FlatFileDest—the name you gave the OLE DB destination adapter. Where did the column names come from? That’s right! From the data flow path metadata we viewed earlier. This functionality is pretty cool, when you think about it.

You don’t need all these columns to store your data in your StagingDB. Since you are using this table to stage data from the flat file, use the same column names you found in the source file. However, you should also use the strong data types you created in our Derived Column transformation. Fortunately for you, our naming convention makes these changes relatively painless. Simply delete the DDL for the first three columns (RecordType, Name, and Value), and then remove the first three letters of the remaining columns, which will rename them to RecordType, Name, and Value. Listing 7-3 shows the result.

Listing 7-3. The Modified CREATE TABLE Statement

```
CREATE TABLE [FlatFileDest] (
    [RecordType] varchar(1),
    [Name] varchar(50),
    [Value] int
)
```

When you click the OK button, the DDL statement is executed against StagingDB—creating the FlatFileDest table. That’s a good thing, because your OLE DB destination adapter is warning you that you need to complete Input-to-Output mappings (shown in Figure 7-6).

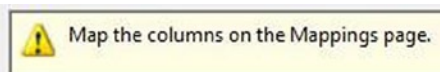


Figure 7-6. The columns have not been mapped

As shown in Figure 7-7, when you click on the Mappings page to begin this mapping, auto-mapping kicks in and finds it can auto-complete some of the mappings:

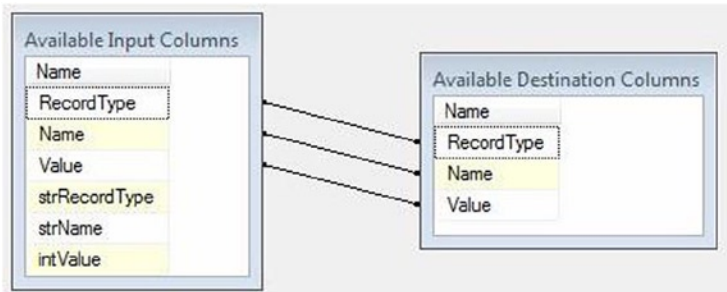


Figure 7-7. OLE DB destination auto-mapping

One issue is that these fields don't contain the data you want to load. You want to load the derived columns instead. There are a couple ways to correct the mapping, but I like dragging and dropping the fields I *want* mapped to the columns where I wish them to be mapped. Since mapping is, by definition, field-to-field, the existing (auto-) mappings will be overwritten by the new. Try it! Drag the `strRecordType` field from Available Input Columns to the `RecordType` cell in Available Output Columns. See? The old mapping is updated. Now map `strName` to `Name` and `intValue` to `Value`, as shown in Figure 7-8:

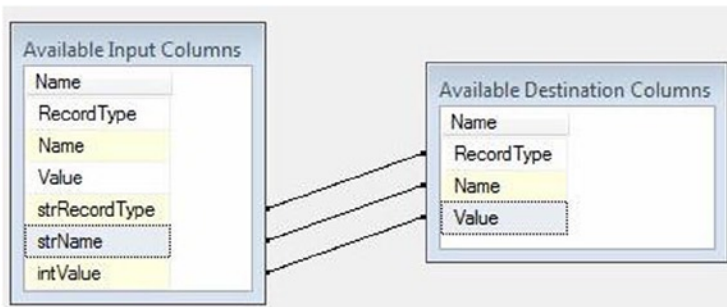


Figure 7-8. Overwriting the auto-mappings

Click the OK button; you're finished configuring the OLE DB destination adapter. Press the F5 key to execute the SSIS package in the SSDT debugger. Hopefully, your Data Flow task succeeds and appears as in Figure 7-9.

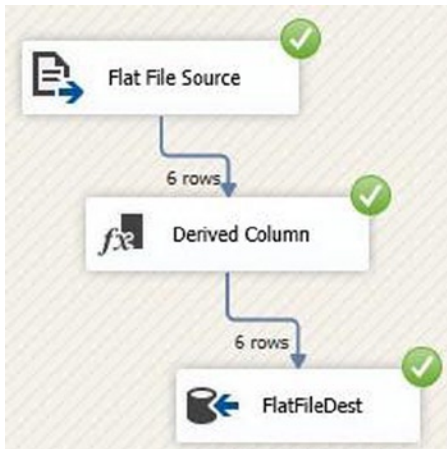


Figure 7-9. A successful data flow!

In this introductory section, I’ve introduced concepts of staging and we built a pattern to stage data from a flat file into a database table. Along the way, we delved into some data warehousing thinking and peeked under the hood of the Data Flow task. Next up: loading another format of flat file—one with variable-length rows.

Variable-Length Rows

A variable-length row flat file is a text source file. It can be a comma-separated values (CSV) file or a tab-delimited file (TDF). It can be a fixed-length file where columns are identified positionally or by ordinal. The major difference between a “normal” flat file and a variable-length row flat file is that the number of text positions is fixed in a normal flat file, and that number can change with each row in a variable-length flat file.

Let’s look at an example of a variable-length flat file:

RecordType,Name1,Value1,Name2,Value2,Name3,Value3

```
A,Multi One,11
B,Multi Two,22,Multi Two A,23
C,Multi Three,33,Multi Three A,34,Multi Three B,345
A,Multi Four,44
C,Multi Five,55,Multi Five A,56,Multi Five B,567
B,Multi Six,66,Multi Six A,67
```

There are seven potential columns: RecordType, Name1, Value1, Name2, Value2, Name3, and Value3. Not all rows contain seven values. In fact, the first row contains only three values:

```
A, Multi One,11
```

In this format, the RecordType is in the first column and this indicates how many columns of data to expect in the row. Rows of RecordType A contain three values, rows of RecordType B contain five values, and those of RecordType C contain seven values.

Reading into a Data Flow

It's typical to load data from a flat file into an SSIS data flow using a Flat File connection manager. Let's walk through configuring a Flat File connection manager for this file.

If you want to sing along, add a new SSIS package named `VariableLengthRows.dtsx` to your SSIS project. Add a Data Flow task to the control flow and open the Data Flow Editor (tab). Drag a Flat File source adapter onto the Data Flow task canvas and open its editor. Click the New button to create a new Flat File connection manager.

I named my Flat File connection manager **Variable-Length File**. I created a text file with the data from the example above and named it **VarLenRows.csv**. I saved it and browsed to that location for the File Name property. I also checked the Column Names in the First Data Row checkbox. When I click on the Columns page, the Flat File Connection Manager Editor appears as shown in Figure 7-10.

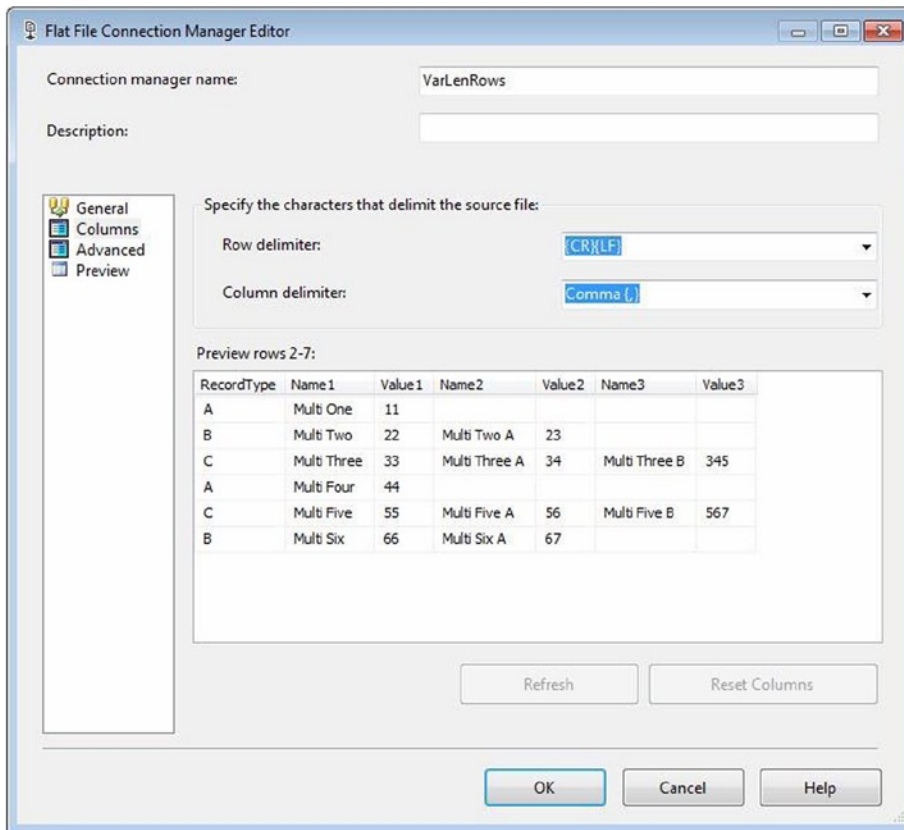


Figure 7-10. Configuring the Flat File connection manager for a flat file with variable-length rows

This behavior is different from earlier editions of SSIS. In previous versions, the Flat File connection manager would raise an error. I blogged about this in a post entitled *SSIS Design Pattern: Loading Variable-Length Rows* (http://sqlblog.com/blogs/andy_leonard/archive/2010/05/18/ssis-design-pattern-loading-variable-length-rows.aspx). That post inspired this chapter.

Splitting Record Types

Thanks to the new functionality in the SSIS 2014 version of the Flat File connection manager, we have all the data coming in as separate rows. But the data rows contain information of different types. The rows need to be filtered based on record type. I can hear you thinking, “Great, Andy. Now what?” I’m glad you asked! Now we need to parse the data as it flows through the Data Flow task. There are a couple ways to approach this, but I like the conditional split.

Drag a Conditional Split transformation onto the Data Flow task canvas and connect a data flow path from the Flat File source adapter to the conditional split. Open the editor for the transformation. In the Output Name column of the grid, enter **TypeA**. Into the corresponding Condition, drag (or type) the RecordType column, appending the text `== “A”` (note that the “A” is in double-quotes). Repeat this for each record type, `== “B”` and `== “C”`, as shown in Figure 7-11.

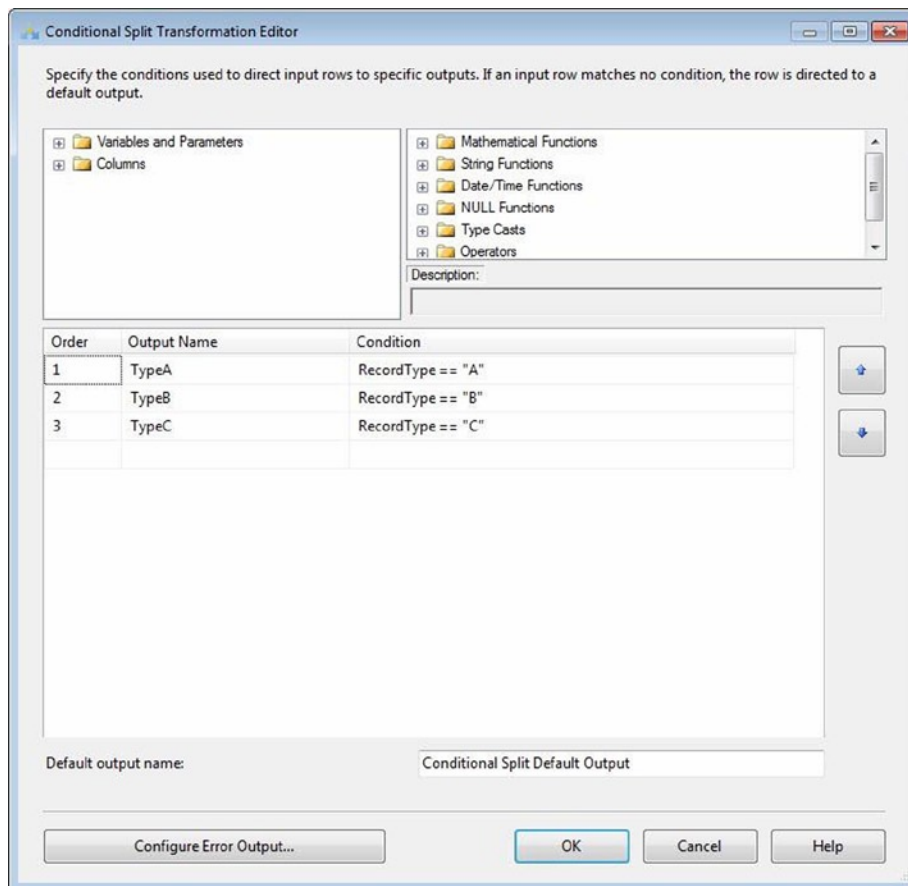


Figure 7-11. Configuring the Script component inputs

Click the OK button to close the Conditional Split Transformation Editor. It is important to note that this would have required a Script component in earlier versions of SSIS because the Flat File connection manager in previous versions couldn’t parse files containing rows with a varying number of columns.

Terminating the Streams

You can use several data flow components to terminate a data flow path. In a production environment, this would likely be an OLE DB destination adapter. In a development or test environment, you may want to terminate with a component that doesn't require database connectivity or database object creation.

You can use any component that doesn't require configuration to execute and succeed in the Data Flow task, such as a Derived Column or Multicast transformation. Here, I will use Multicast transformations to terminate the data flow path streams.

Drag three Multicast transformations onto the Data Flow task canvas. Connect an output from the Script component to the TypeA multicast. When prompted, select the TypeA output buffer for the Script component, as shown in Figure 7-12.

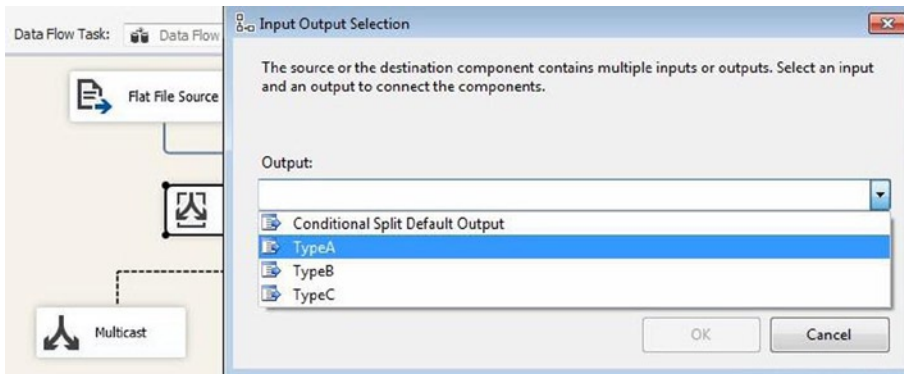


Figure 7-12. Terminating the TypeA output from the Script component

Repeat this process for TypeB and TypeC connections. When you've finished, your data flow could appear as shown in Figure 7-13.

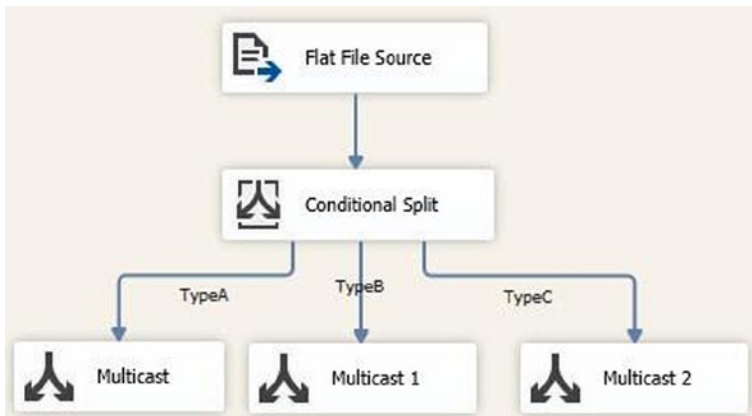


Figure 7-13. The Script component's outputs, terminated

Let's run it! Execution should succeed, and when it does, the results will be the green checkmarks that you see in Figure 7-14.

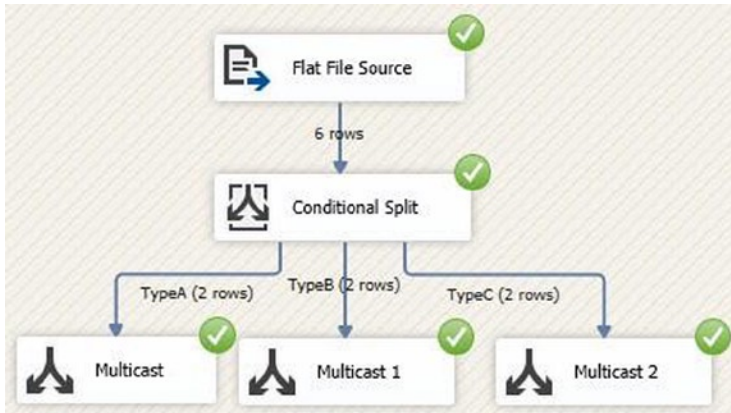


Figure 7-14. *Happiness is green checks*

This isn't the only way to address loading files of this format. It is one way, and it has the advantage of offering a lot of flexibility.

Header and Footer Rows

Header and footer rows are included in extract files from many sources. I regularly see these rows in flat files delivered from mainframe-based database systems.

A *header row* contains metadata about the contents of the extract file—a summary of information of the extract. At a minimum, it will include the extract date. Usually a field contains a row count for the extract. When you think about it, this is very practical instrumentation—it provides a way to check for the proper number of rows. This check can be performed immediately after the extract or immediately after loading the extract—both are valid uses of this metadata.

A *footer row* is identical in concept. The only difference is location: header rows appear first in a file; footer rows appear last. If you're including row counts to validate that the correct number of rows have been written or read, writing this information first is a good way to increase fault tolerance. Why? Imagine a failure: the write operation is interrupted or the extract ends abnormally. The header row may indicate 100 rows, for example, but only 70 rows of data follow. If the row count metadata is included in the header row, it's possible to calculate exactly how many data rows are missing. In contrast, a footer row would simply be missing. Although a missing footer would indicate that the extract had failed, that's *all* it would indicate. Having the row count metadata present would allow you to glean more and better information regarding the failure.

In this section, we will look at creating and consuming header and footer rows using SQL Server 2014 Integration Services.

Consuming a Footer Row

We begin by looking at how to consume a footer row. To start, create a file containing a footer row. My file looks like this:

```
ID,Name,Value
11,Andy,12
22,Christy,13
33,Stevie Ray,14
44,Emma Grace,15
55,Riley Cooper,16
5 rows, extracted 10/5/2011 10:22:12 AM
```

To demonstrate, create your own file and name it `MyFileFooterSource.csv`. Create a new SSIS package and rename it **ParseFileFooter.dtsx**. Add a Data Flow task and switch to the Data Flow tab. Add a Flat File source adapter and double-click it to open the Flat File Source Editor. On the Connection Manager page, click the New button to create a new Flat File connection manager and open the editor. Name the Flat File connection manager **My File Footer Source File** and set the File Path property to the location of `MyFileFooterSource.csv`. Check the Column Names in the First Data Row checkbox. Navigate to the Columns page to verify your configuration matches, as shown here in Figure 7-15.

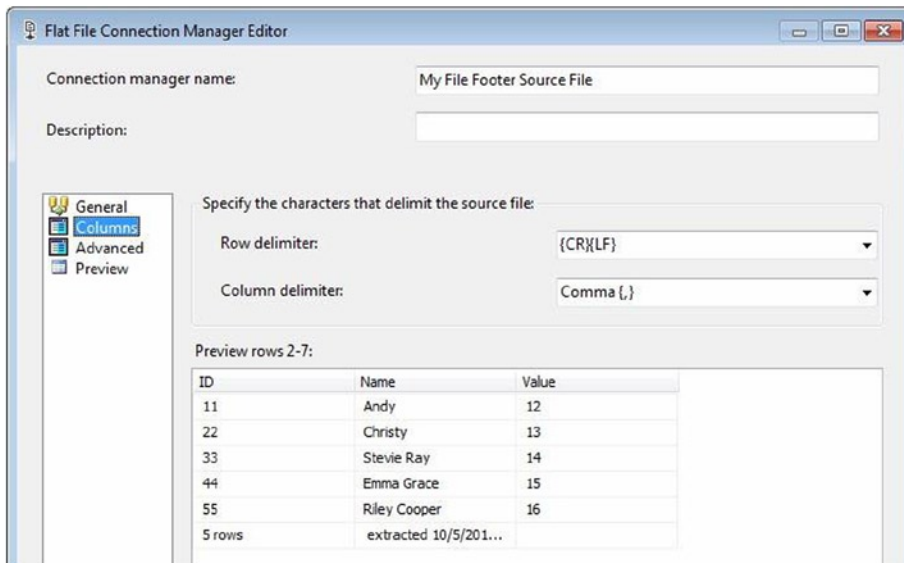


Figure 7-15. Flat File connection manager Columns page for a file containing a footer row

You can see the footer row contents in the preview shown in Figure 7-15. You may or may not be able to view the footer row on the Columns page of the Flat File Connection Manager Editor.

The next step is to separate the footer row from the data rows. To accomplish this, you will use a Conditional Split transformation to isolate the footer row. There are a *lot* of different ways to detect the footer row, but the trick is to pick something unique about that row. In the SSIS Expression Language expression I define in Figure 7-16, I search for the term "rows" in the ID column. This condition will work as long as there's *never a chance* that the term "rows" will legitimately show up in the ID column of the data rows. Never is a very long time.

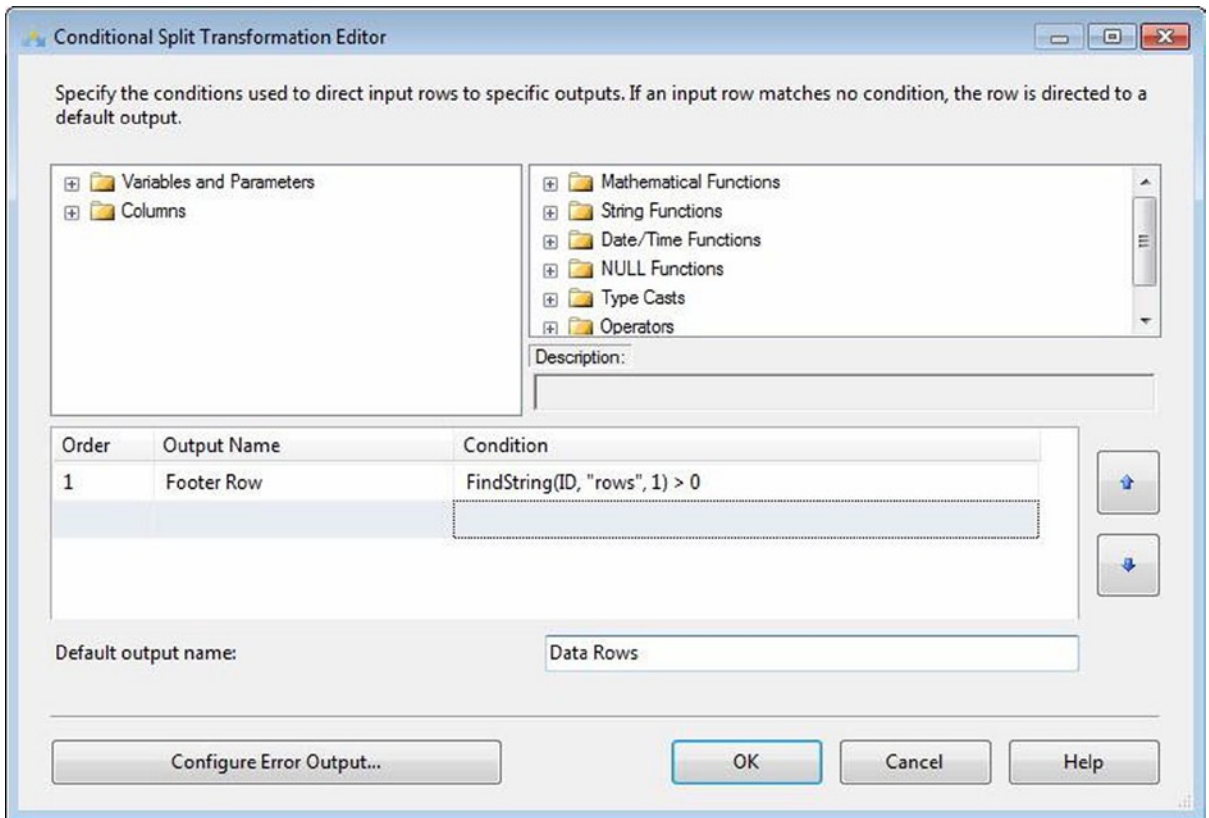


Figure 7-16. Configuring the Conditional Split transformation

To terminate the data rows pipeline—which flows from the Conditional Split transformation’s default output—I use a Derived Column transformation.

The footer row output requires more parsing. We send it to another Derived Column transformation named der Parse Footer.

■ **Note** Jamie Thomson wrote a great post entitled “SSIS: Suggested Best Practices and Naming Conventions” (http://sqlblog.com/blogs/jamie_thomson/archive/2012/01/29/suggested-best-practises-and-naming-conventions.aspx). I often use Jamie’s naming conventions.

We want the number of rows and the datetime of the extraction. I use the expressions in Figure 7-17 to parse the footer row count and footer extract datetime:

Derived Column Name	Derived Column	Expression	Data Type
FooterRowCount	<add as new column>	(DT_I4)TRIM(LEFT(ID,FINDSTRING(ID,"1")))	four-byte signed integer [DT_I4]
FooterExtractDateTime	<add as new column>	(DT_DBTimeStamp)TRIM(RIGHT(Name,LEN(Name) - FINDSTRING(Name,"2")))	database timestamp [DT_DBTIMESTAMP]

Figure 7-17. Parsing the row count and extract date

Now we have the footer row metadata in the data flow pipeline. We can terminate this branch of the pipeline using another Derived Column transformation, der Trash Destination Footer. Connect der Parse Footer to der Trash Destination Footer. Right-click the data flow path and click Enable Data Viewer. Execute the package in the debugger to view the contents of the footer row, as shown in Figure 7-18.

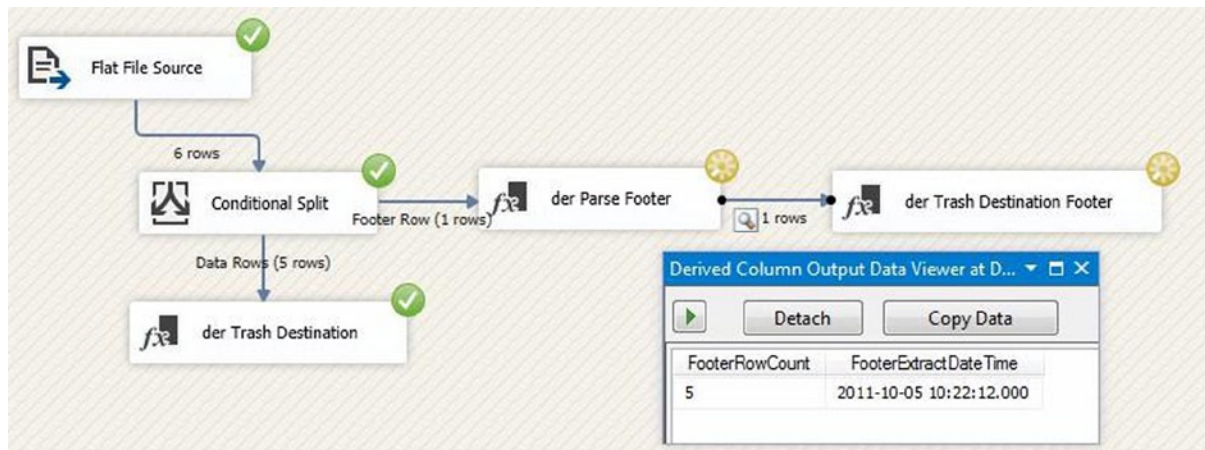


Figure 7-18. Footer row, parsed

You can see from Figure 7-18 that five (5) data rows exited the Conditional Split transformation. We can observe the contents of the footer row after parsing in the Data Viewer.

Consuming a Header Row

Header rows are even easier to read. Let's start with a look at the source file named `MyFileHeaderSource.csv`:

```
5 rows, extracted 10/5/2011 10:22:12 AM
```

```
-----
ID,Name,Value
11,Andy,12
22,Christy,13
33,Stevie Ray,14
44,Emma Grace,15
55,Riley Cooper,16
```

You can read header rows a few different ways. In this solution, we utilize one Flat File connection manager and one data flow to parse the header row of the data. We rely heavily on Script component logic for parsing and buffering operations.

Begin by creating a new SSIS package. I named mine `ParseFileHeader.dtsx`.

Add a Data Flow task and open the Data Flow Task Editor. Add a Flat File source adapter and open its editor. Use the New button to create a new Flat File connection manager aimed at `MyFileHeaderSource.csv`. Uncheck the Column Names in the First Data Row checkbox. Be sure to click the Advanced page of the Connection Manager Editor and change the names of Column 0 and Column 1 to ID and Name, respectively.

Close the Connection Manager and Source Adapter Editors and drag a Script component onto the data flow canvas. When prompted, select Transformation as the use of this Script component. Open the Script Component Editor and change the Name property to `scr Parse Header and Data`. Click the Input Columns page and select

both columns (ID and Name). Click on the Inputs and Outputs page. Rename Output 0 to **Header** and change the SynchronousInputID property to **None**. Expand the Header output and click the Output Columns virtual folder. Click the Add Column button, name it ExtractDateTime, and change the data type to **database timestamp [DT_DBTIMESTAMP]**. Click the Add Column button again, name this new column **RowCount**, and leave the data type set to the default (four-byte signed integer [DT_I4]).

Click the Add Output button and name this new output **Data**. Expand the output virtual folder and select the Output Columns virtual folder. As you did for the header output, create two columns with the following properties:

- ID, four-byte signed integer [DT_I4]
- Name, string [DT_STR]

Return to the Script page and set the ScriptLanguage property to Microsoft Visual Basic 2012. Click the Edit Script button. When the editor opens, add a variable declaration at the top of the class (see Listing 7-4).

Listing 7-4. Adding iRowNum Integer Variable

```
Public Class ScriptMain
    Inherits UserComponent
```

```
Dim iRowNum As Integer = 0
```

Replace the code in the Input0_ProcessInputRow subroutine with the code in Listing 7-5.

Listing 7-5. Building Input0_ProcessInputRow Subroutine

```
Public Overrides Sub Input0_ProcessInputRow(ByVal Row As Input0Buffer)
```

```
    ' increment rownum
    iRowNum += 1

    Select Case iRowNum
        Case 1
            ' parse
            Dim sTmpCount As String = Row.ID
            sTmpCount = Strings.Trim(Strings.Left(Row.ID, Strings.InStr(Row.ID, " ")))
            Dim sTmpDate As String = Row.Name
            sTmpDate = Strings.Replace(Row.Name, " extracted ", "")

            ' header row
            With HeaderBuffer
                .AddRow()
                .RowCount = Convert.ToInt32(sTmpCount)
                .ExtractDateTime = Convert.ToDateTime(sTmpDate)
            End With
        Case 2
            ' ignore
        Case 3
            'column names
```

```

    Case Else
      ' data rows
      With DataBuffer
        .AddRow()
        .ID = Convert.ToInt32(Row.ID)
        .Name = Row.Name
      End With
    End Select
End Sub

```

This script counts the rows flowing through the Script component and uses the number of the row to decide the disposition of the output row. A `Select Case` statement is driven by row number detection and each row increments the row number incrementor (`iRowNum`). The first row is the header row and it contains the extract metadata. The next two rows contain a scratch row of dashes and the column names, respectively. The remainder of the file contains data rows and is addressed in the `Select Case Else` condition of the `Select Case` statement.

Close the VSTA Projects Script Editor and click the OK button on the Script Component Editor. Terminate the Header and Data pipelines with the data flow component of your choice (I use Derived Column transformations named `der Header` and `der Data`).

Test the package by executing it in the debugger. Your results should be similar to those shown in Figure 7-19.

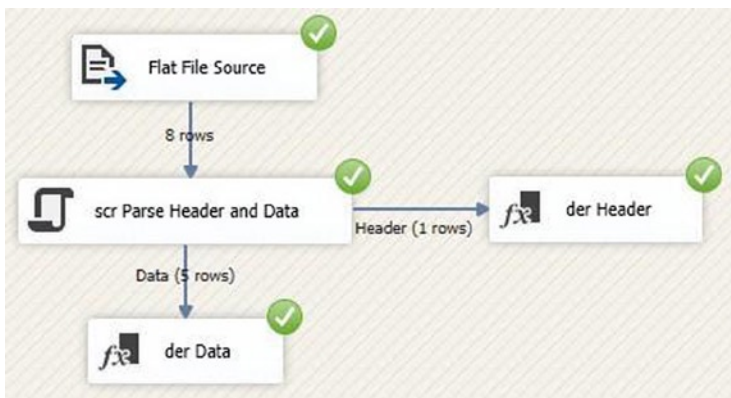


Figure 7-19. *Green checks rock!*

Producing a Footer Row

Let's look at producing a footer row and adding it to the data file. For this pattern, we will leverage project and package parameters. We will also leverage the Parent-Child pattern, which is discussed in detail in Chapter 16. We are not going to build the package that creates a flat file containing data. We will start with the assumptions that an extract file exists and we know the number of rows and the extract date. We will use parameters to transmit metadata from the parent package to the child package. Let's get started!

Create a new SSIS package and name it `WriteFileFooter.dtsx`. Click on the Parameters tab and add the following package parameters:

Name	Data Type	Value	Required
AmountSum	Decimal	0	False
DateFormat	String		True
Debug	Boolean	True	False
Delimiter	String	,	True
ExtractFilePath	String		True
LastUpdateDateTime	DateTime	1/1/1900	True
RecordCount	Int32	0	True

The parameters, when entered, appear as shown in Figure 7-20.

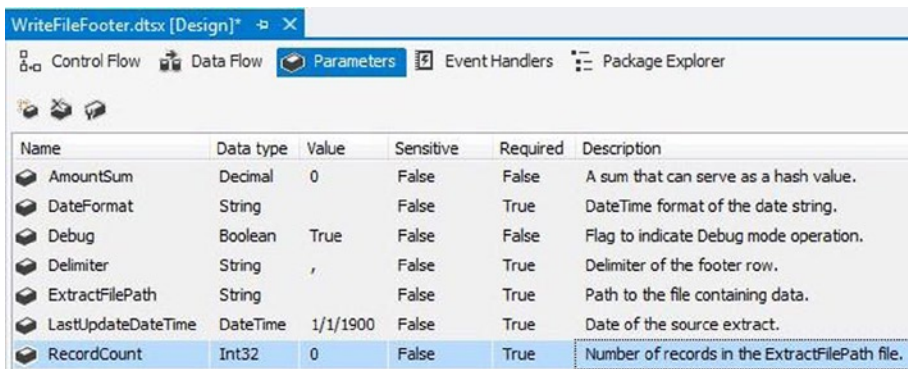


Figure 7-20. Parameters for the `WriteFileFooter.dtsx` package

The Sensitive property for each parameter is set to False. The Description is optional and available in the image.

We're going to do the heavy lifting in a Script task. Return to the control flow and drag a Script task onto the canvas. Change the name to **scr Append File Footer** and open the editor. On the Script page, click the ellipsis in the `ReadOnlyVariables` property's Value textbox. When the Select Variables window displays, select the following variables:

- `System::PackageName`
- `System::TaskName`
- `$Package::AmountSum`
- `$Package::DateFormat`
- `$Package::Debug`
- `$Package::Delimiter`
- `$Package::ExtractFilePath`
- `$Package::LastUpdateDateTime`
- `$Package::RecordCount`

The Select Variables window will not appear exactly as shown in Figure 7-21, but these are the variables you need to select for use inside the `scr Append File Footer` Script task.

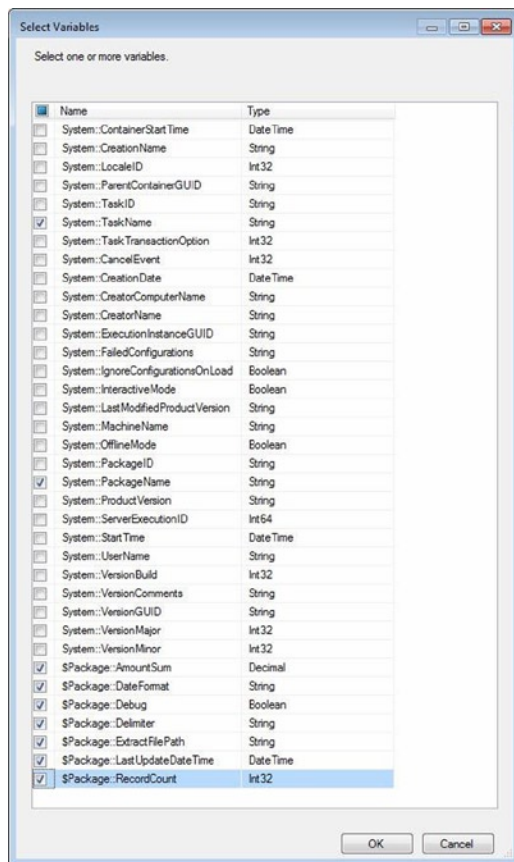


Figure 7-21. Selecting variables for the footer file

Click the OK button to close the Select Variables window. Set the ScriptLanguage property to Microsoft Visual Basic 2012. Click the Edit Script button to open the VstaProjects window. At the top of the ScriptMain.vb code window, you will find an Import region. Add the lines from Listing 7-6 to that region.

Listing 7-6. Adding Imports Statements to the VB.Net Script

```
Imports System.IO
Imports System.Text
```

Just after the partial class declaration, add the variable declaration for the bDebug variable. It's the Dim statement given in Listing 7-7.

Listing 7-7. Declaring the Debug Variable

```
Partial Public Class ScriptMain
    Inherits Microsoft.SqlServer.Dts.Tasks.ScriptTask.VSTARTScriptObjectModelBase

    Dim bDebug As Boolean
```

Replace the code in `Public Sub Main` with that from Listing 7-8.

Listing 7-8. Code for the `Main()` Subroutine

```
Public Sub Main()

    ' 1: detect Debug setting...
    bDebug = Convert.ToBoolean(Dts.Variables("Debug").Value)

    ' 2: declare and initialize variables...
    ' 2a: generic variables...
    Dim sPackageName As String = Dts.Variables("PackageName").Value.ToString
    Dim sTaskName As String = Dts.Variables("TaskName").Value.ToString
    Dim sSubComponent As String = sPackageName & "." & sTaskName
    Dim sMsg As String
    ' 2b: task-specific variables...
    Dim sExtractFilePath As String = Dts.Variables("ExtractFilePath").Value.ToString
    Dim iRecordCount As Integer = Convert.ToInt32(Dts.Variables("RecordCount").Value)
    Dim sAmountSum As String = Dts.Variables("AmountSum").Value.ToString
    Dim sDateFormat As String = Dts.Variables("DateFormat").Value.ToString
    Dim sDelimiter As String = Dts.Variables("Delimiter").Value.ToString
    Dim sLastUpdateDateTime As String = _
Strings.Format(Dts.Variables("LastUpdateDateTime").Value, sDateFormat) & _
"yyyy/MM/dd hh:mm:ss.fff")
    Dim sFooterRow As String
    Dim s As Integer = 0

    ' 3: log values...
    sMsg = "Package Name.Task Name: " & sSubComponent & ControlChars.CrLf & _
ControlChars.CrLf & _
    "Extract File Path: " & sExtractFilePath & ControlChars.CrLf & _
ControlChars.CrLf & _
    "Record Count: " & iRecordCount.ToString & ControlChars.CrLf & _
ControlChars.CrLf & _
    "Amount Sum: " & sAmountSum & ControlChars.CrLf & ControlChars.CrLf & _
    "Date Format: " & sDateFormat & ControlChars.CrLf & ControlChars.CrLf & _
    "Delimiter: " & sDelimiter & ControlChars.CrLf & ControlChars.CrLf & _
    "LastUpdateDateTime: " & sLastUpdateDateTime & ControlChars.CrLf & _
ControlChars.CrLf & _
    "Debug: " & bDebug.ToString
    Dts.Events.FireInformation(0, sSubComponent, sMsg, "", 0, True)
    If bDebug Then MsgBox(sMsg)

    ' 4: create footer row...
    sFooterRow = iRecordCount.ToString & sDelimiter & sAmountSum & sDelimiter & _
sLastUpdateDateTime
    ' 5: log...
    sMsg = "Footer Row: " & sFooterRow
    Dts.Events.FireInformation(0, sSubComponent, sMsg, "", 0, True)
    If bDebug Then MsgBox(sMsg)
```

```

' 6: check if the file is in use...
While FileInUse(sExtractFilePath)
' 6a: if file is in use, sleep for a second...
System.Threading.Thread.Sleep(1000)
' 6b: incrementor...
s += 1
' 6c: if incrementor reaches 10 (10 seconds),
If s > 10 Then
' exit the loop...
Exit While
End If 's > 10
End While 'FileInUse(sExtractFilePath)
' 7: log...
If s = 1 Then
sMsg = "File was in use " & s.ToString & " time."
Else ' s = 1
sMsg = "File was in use " & s.ToString & " times."
End If ' s = 1
Dts.Events.FireInformation(0, sSubComponent, sMsg, "", 0, True)
If bDebug Then MsgBox(sMsg)

' 8: if the file exists...
If File.Exists(sExtractFilePath) Then
Try
' 8a: open it for append, encoded as built, using a streamwriter...
Dim writer As StreamWriter = New StreamWriter(sExtractFilePath, True, _
Encoding.Default)
' 8b: add the footer row...
writer.WriteLine(sFooterRow)
' 8c: clean up...
writer.Flush()
' 8d: get out...
writer.Close()
' 8e: log...
sMsg = "File " & sExtractFilePath & " exists and the footer row has " & _
"been appended."
Dts.Events.FireInformation(0, sSubComponent, sMsg, "", 0, True)
If bDebug Then MsgBox(sMsg)
Catch ex As Exception
' 8f: log...
sMsg = "Issue with appending footer row to " & sExtractFilePath & _
" file: " & ControlChars.CrLf & ex.Message
Dts.Events.FireInformation(0, sSubComponent, sMsg, "", 0, True)
If bDebug Then MsgBox(sMsg)
End Try
Else
' 8g: log...
sMsg = "Cannot find file: " & sExtractFilePath
Dts.Events.FireInformation(0, sSubComponent, sMsg, "", 0, True)
If bDebug Then MsgBox(sMsg)

```



```

End If ' File.Exists(sExtractFilePath)

' 9: return success...
Dts.TaskResult = ScriptResults.Success

End Sub

```

Then add the function from Listing 7-9 after `Public Sub Main()`.

Listing 7-9. Code for the `FileInUse` Function

```

Function FileInUse(ByVal sFile As String) As Boolean

    If File.Exists(sFile) Then
        Try
            Dim f As Integer = FreeFile()
            FileOpen(f, sFile, OpenMode.Binary, OpenAccess.ReadWrite, _
OpenShare.LockReadWrite)
            FileClose(f)
        Catch ex As Exception
            Return True
        End Try
    End If
End Function

```

The script now builds the footer row and appends it to the Extract file. The first thing we do—at the comment labeled 1—is assign a value to the Debug variable. I use the Debug variable to control message boxes displaying variable values and other pertinent information. I describe why in Chapter 2 on execution patterns.

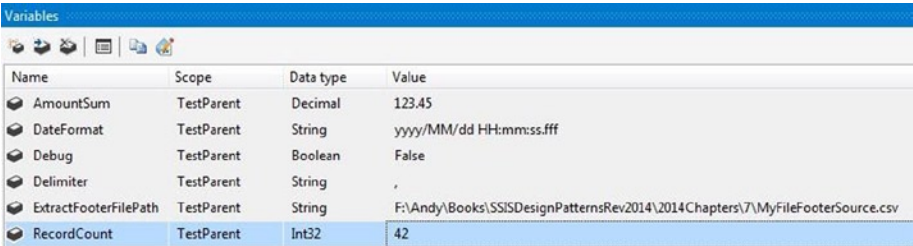
At comment 2, we declare and initialize variables. I break variables into two categories: generic and task-specific variables. At comment 3, we build a message in the variable `sMsg`. This message contains the values of each variable used in the script thus far. If we are running in Debug mode (if `bDebug` is `True`), the code displays a message box (via the `MsgBox` function) containing the contents of `sMsg`. Whether we’re running in Debug mode or not, I use the `Dts.Events.FireInformation` method to raise an `OnInformation` event, passing it the contents of `sMsg`. This means the information is always logged and is optionally displayed by a message box. I like options (a lot).

Comment 4 has us constructing the actual footer row and placing its text in the String variable `sFooterRow`. Note the delimiter is also dynamic. The String variable `sDelimiter` contains the value passed to the `WriteFileFooter` into the Package parameter named `$Package::Delimiter`. At comment 5, we log the contents of the footer row.

At comment 6, we initiate a check to make sure the extract file is not marked as “in use” by the operating system. There are many ways to detect the state of the file in the file system, so I created a Boolean function named `FileInUse` to encapsulate this test. If the function I created doesn’t work for you, you can construct your own. If the file is in use, the code initiates a While loop that sleeps the thread for one second. Each iteration through the loop causes the variable `s` (the incrementor in this example) to increment at comment 6b. If `s` exceeds ten, the loop exits. We will only wait 10 seconds for the file to be usable. Note that if the file remains in use at this juncture, we *still* move on. We’ll deal with the file-in-use matter later, but we will not hang ourselves in a potentially endless loop waiting for the file’s availability. We will instead fail. Whether the file is in use or not in use, the script logs its state at comment 7.

At comment 8, we check for the existence of the file and begin a Try-Catch. If the file doesn’t exist, I opt to log a status message (via `Dts.Events.FireInformation`) and continue (see comment 8g). The Try-Catch enforces the final test of the file’s usability. If the file remains in use here, the Catch fires and logs the status message at comment 8f. At 8f and/or 8g, you may very well decide to raise an error using the `Dts.Events.FireError` method. Raising an error causes the Script task to fail, and you may want this to happen. At comments 8a through 8d, we open the file, append the footer row, close the file, and clean up. At comment 8e, the code logs a status message. If anything fails when we are executing 8a through 8e, code execution jumps to the Catch block.

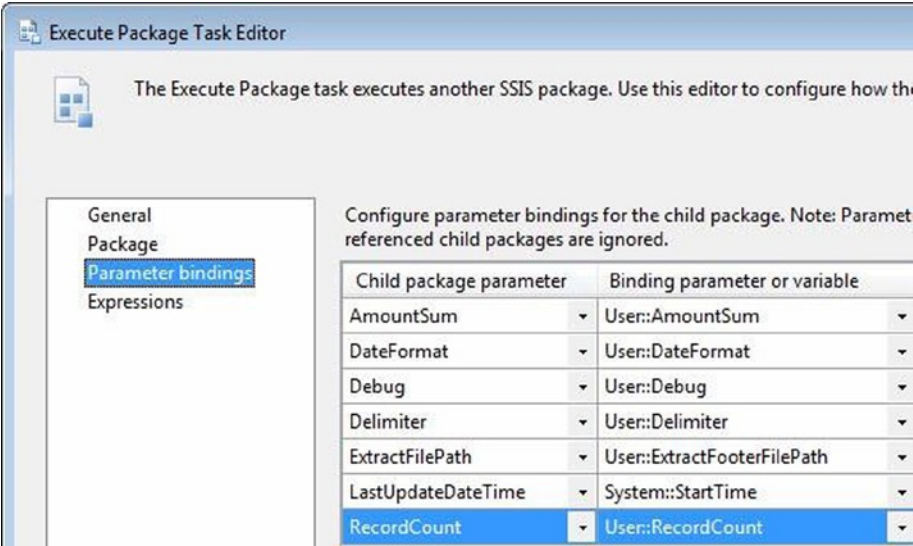
If all goes well, the code returns Success to the SSIS control flow via the `Dts.TaskResult` function (comment 9). The Script task does all the work in this pattern. Close the Script Task Editor. Click OK. Then save the package. I created a test package called `TestParent.dtsx` to test this package. The package has variables that align with the parameters of the `WriteFileFooter.dtsx` package, as shown in Figure 7-22.



Name	Scope	Data type	Value
AmountSum	TestParent	Decimal	123.45
DateFormat	TestParent	String	yyyy/MM/dd HH:mm:ss.fff
Debug	TestParent	Boolean	False
Delimiter	TestParent	String	,
ExtractFooterFilePath	TestParent	String	F:\Andy\Books\SSISDesignPatternsRev2014\2014Chapters\7\MyFileFooterSource.csv
RecordCount	TestParent	Int32	42

Figure 7-22. Variables in the `TestParent.dtsx` package

If you’re playing along at home, you should adjust the path of the `ExtractFooterFilePath` variable. I added a Sequence container named `seq Test WriteFileFooter` and included an Execute Package task named `ept Execute WriteFileFooter Package`. On the Package page of the Execute Package Task Editor, set the `ReferenceType` property to `Project Reference` and select `WriteFileFooter.dtsx` from the `PackageNameFromProjectReference` property drop-down. Map the `TestParent` package variables to the `WriteFileFooter` package parameters as shown in Figure 7-23.



Child package parameter	Binding parameter or variable
AmountSum	User::AmountSum
DateFormat	User::DateFormat
Debug	User::Debug
Delimiter	User::Delimiter
ExtractFilePath	User::ExtractFooterFilePath
LastUpdateDateTime	System::StartTime
RecordCount	User::RecordCount

Figure 7-23. Mapping package parameters

Execute `TestParent.dtsx` to test the functionality. The package executes successfully and the footer row is appended to the file as shown in Figure 7-24.

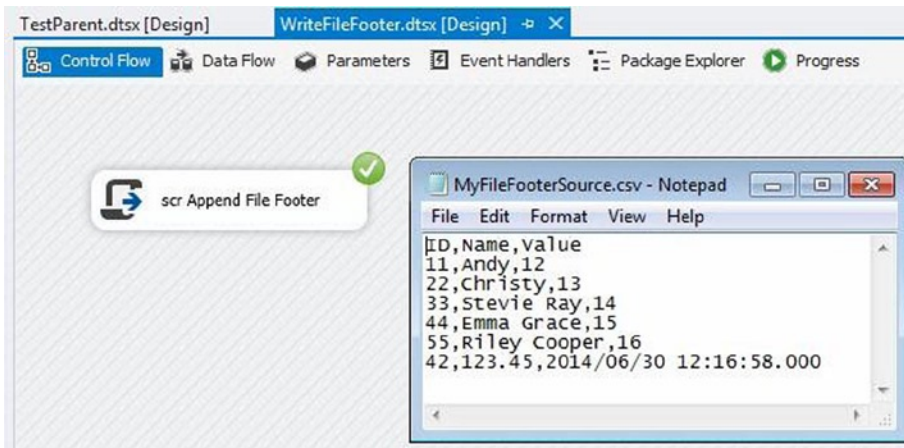


Figure 7-24. Mission accomplished

Producing a Header Row

Producing a header row is a very simple operation in SSIS 2014, provided you know the number of rows to be loaded in advance. You simply load the destination flat file with the header row in one Data Flow task, and then load the data rows to the same flat file in a subsequent Data Flow task. As we say in Farmville, Virginia: “dog simple.” There are some subtle complexities in this design, though.

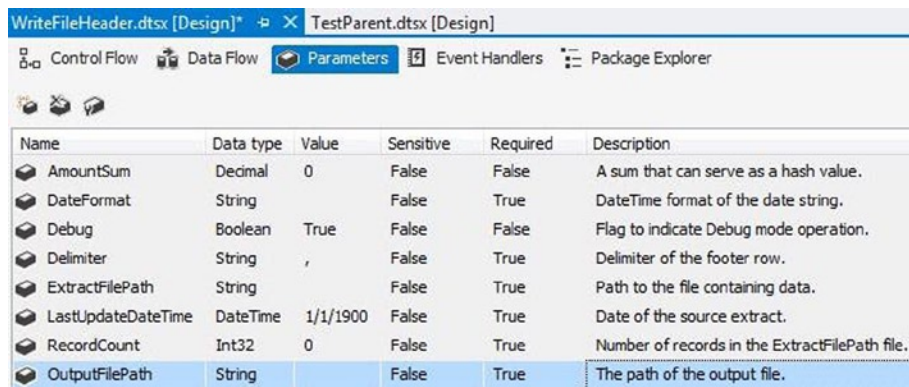
We’ll start with a simple file named `MyFileHeaderExtract.csv` that contains the following data:

```

ID,Name,Value
11,Andy,12
22,Christy,13
33,Stevie Ray,14
44,Emma Grace,15
55,Riley Cooper,16

```

Add a new SSIS package named `WriteFileHeader.dtsx` to your SSIS project. Add the package parameters shown in Figure 7-25.



Name	Data type	Value	Sensitive	Required	Description
AmountSum	Decimal	0	False	False	A sum that can serve as a hash value.
DateFormat	String		False	True	DateTime format of the date string.
Debug	Boolean	True	False	False	Flag to indicate Debug mode operation.
Delimiter	String	,	False	True	Delimiter of the footer row.
ExtractFilePath	String		False	True	Path to the file containing data.
LastUpdateDateTime	DateTime	1/1/1900	False	True	Date of the source extract.
RecordCount	Int32	0	False	True	Number of records in the ExtractFilePath file.
OutputFilePath	String		False	True	The path of the output file.

Figure 7-25. WriteFileHeader.dtsx parameters

Add two Data Flow tasks to the control flow. Name the first **dft Write Header Row** and the second **dft Write Data Rows**. Open the editor for dft Write Header Row and add a Script component named **srcr Build Header Row** to the Data Flow task. When prompted, configure the Script component to act as a source. Open the editor and set the ScriptLanguage property to Microsoft Visual Basic 2012. Set the ReadOnlyVariables property to reference the following:

- \$Package::AmountSum
- \$Package::Delimiter
- \$Package::LastUpdateDateTime
- \$Package::RecordCount

On the Inputs and Outputs page, ensure the SynchronousInputID property of Output 0 is set to None and add an output column named HeaderRow (String data type, 500 length) to Output 0. Click the Script page and the Edit Script button. Replace the code in the CreateNewOutputRows() subroutine with that from Listing 7-10.

Listing 7-10. Code for the CreateNewOutputRows Subroutine

```
Public Overrides Sub CreateNewOutputRows()

    ' create header row...          ' Get variable values...
    Dim iRecordCount As Integer = Me.Variables.RecordCount
    Dim sDelimiter As String = Me.Variables.Delimiter
    Dim dAmountSum As Decimal = Convert.ToDecimal(Me.Variables.AmountSum)
    Dim dtLastUpdateDateTime As DateTime = _
    Convert.ToDateTime(Me.Variables.LastUpdateDateTime)

    Dim sHeaderRow As String = iRecordCount.ToString & sDelimiter & _
                                dAmountSum.ToString & sDelimiter & _
    dtLastUpdateDateTime.ToString

    With Output0Buffer
        .AddRow()
        .HeaderRow = sHeaderRow
    End With
End Sub
```

Add a Flat File destination adapter and connect a data flow path from the Script component to it. Open the Flat File Destination Editor and click the New button beside the Flat File Connection Manager drop-down. When the Flat File Format window displays, select Delimited and click the OK button. Name the Flat File connection manager **Flat File Header Output** and supply (or select) a file path. On the Columns page, configure a landing column for the HeaderRow column from the src Build Header Row Script component. Click the OK button to return to the Flat File Destination Editor. Make sure the Overwrite Data in the File checkbox (in the Connection Manager page) is checked. It should be; this is the default. Click on the Mappings page and complete the destination configuration. This Data Flow task will construct and load the header row.

On the control flow, add a Success Precedence constraint from dft Write Header Row to the dft Write Data Rows Data Flow task. Open the editor for dft Write Data Rows and add a Flat File source adapter. Open the Flat File Source Editor and click the New button to create a new Flat File connection manager. When prompted, select Delimited. Name it **Extract File Input** and navigate to the MyFileHeaderExtract.csv file you created earlier. On the Columns page, delete the value in the Column Delimiter drop-down. To refresh the view, click the Refresh button. On the Advanced page, rename the column from ID,Name,Value to **Row** and set the OutputColumnWidth property to **500**. Click the OK buttons to close the Flat File Connection Manager Editor and the Flat File Source Editor.

Add a Flat File destination adapter and connect a data flow path from the Flat File source adapter to the Flat File destination adapter. Open the Flat File destination adapter and set its connection manager to the flat file header output. Be sure to uncheck the Overwrite the Data in the File checkbox on the Connection Manager page. On the Mappings page, map the Row column from the Available Input Columns to the HeaderRow in the Available Destination Columns. Close the Flat File Destination Editor.

Let's make these connection managers dynamic! Click the Extract File Input Flat File Connection Manager, and then press the F4 key to display properties. Click the Expressions property and click the ellipsis in the Value textbox. Click the drop-down in the Property column of the first row and click `ConnectionString`. In the corresponding Expression value textbox, click the ellipsis to display the Expression Builder. Expand the Variables and Parameters virtual folder in Expression Builder and drag `$Package::ExtractFilePath` into the Expression textbox. Click the OK button to close the Expression Builder. The Property Expressions Editor window will appear, as shown in Figure 7-26.



Figure 7-26. Dynamic ConnectionString property

Close the Property Expressions Editor. You have now assigned the ConnectionString property to the value passed to the ExtractFilePath package parameter passed to this package when it is called from another package. Repeat this process to dynamically assign the value of the `$Package::OutputFilePath` package parameter to the ConnectionString property of the Flat File Header Output Flat File Connection Manager.

To test this package, return to TestParent.dtsx. Let's add a couple of variables to use with the parameters we just mapped to connection manager expressions: ExtractHeaderFilePath and OutputPath. Supply a value for the OutputPath variable that represents the location of the file you want to create. (Note: this file may not exist!) Also, supply the path to the MyFileHeaderExtract.csv as the default value for the ExtractHeaderFilePath variable. On the control flow, add a Sequence container and rename it **seq Test WriteHeader**. Add an Execute Package task to the Sequence container and rename it **ept Execute WriteFileHeader Package**. Open the Execute Package Task Editor and configure a project reference to execute the WriteFileHeader.dtsx package. Configure the Parameter Bindings page as shown in Figure 7-27.

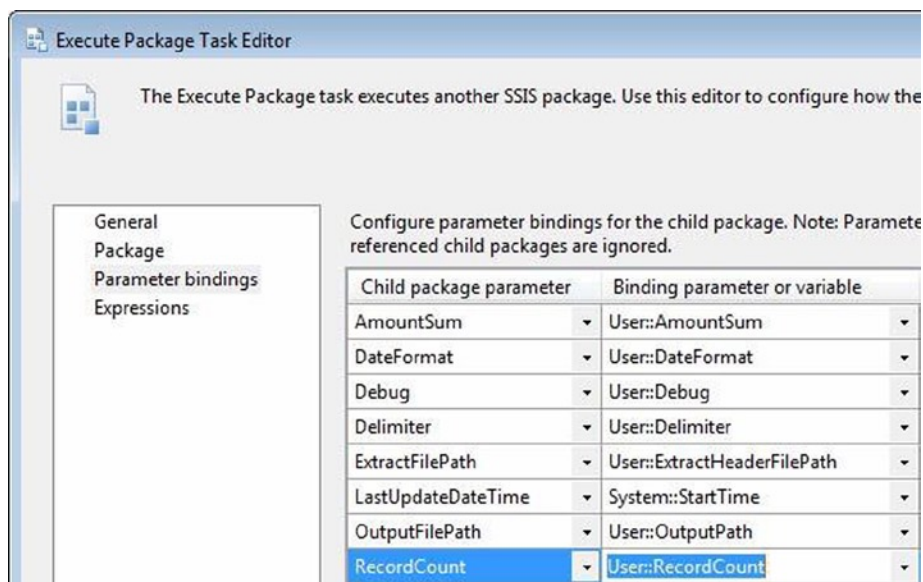


Figure 7-27. Parameter mapping in the Execute Package task

Close the Execute Package Task Editor and disable the seq Test WriteFileFooter Sequence container. Execute the package and observe the results. You should get results like those shown in Figure 7-28.

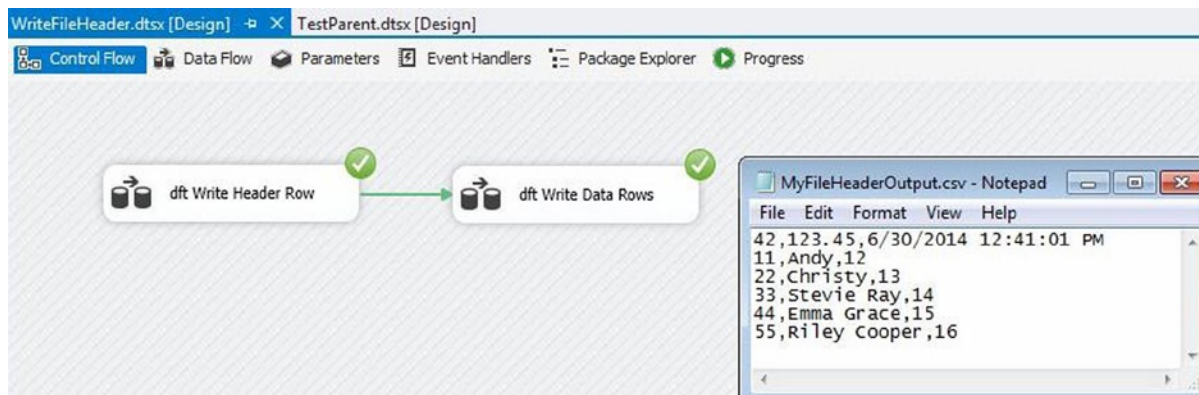


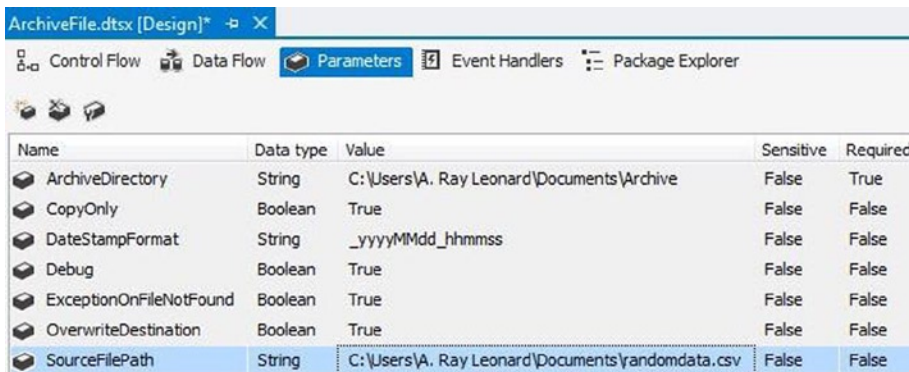
Figure 7-28. Success!

I like this pattern because it utilizes SSIS components without resorting to too much scripting. I don't like everything about this pattern, though. I need to know the number of rows before calling this package, which isn't hard to acquire—I can simply add a Row Count transformation to a data flow and count the rows as they are loaded into the extract file. But then I must *reload* the extract file, after the fact. For large files and scalability, I would attempt to ascertain the number of rows before loading the file and then integrate the functionality demonstrated in this package into the loader package. For smaller loads of data that will not scale, this package is acceptable.

The Archive File Pattern

The Archive File pattern is largely responsible for the book you are now reading. How? It was the first widely adopted design pattern package that I built. After re-using this pattern in several locations, I became convinced SSIS lent itself to design pattern-based architectures. Shortly after this realization, I discussed the idea over dinner in Bellevue, Washington with friends who work with SSIS and who also write books. We agreed design patterns offer interesting solutions to many data integration problems.

The ArchiveFile package is designed to copy a flat data file from one directory to another, appending a datetime stamp to the original file name. The full path of the original file is supplied in the SourceFilePath parameter, the format of the datetime stamp in the DateStampFormat parameter. The destination, or Archive, directory is supplied to the ArchiveDirectory parameter. Should the target file already exist, you can control overwrites of the destination file via the OverwriteDestination parameter. The package usually deletes the original file, but the CopyOnly parameter controls this function. If the SourceFilePath is not found, you can raise an error or simply log this condition. The ExceptionOnFileNotFound parameter controls whether the package raises an error if the source file is not found. Finally, the Debug parameter controls whether the package is being executed in Debug mode (something I cover in more detail in Chapter 2). The ArchiveFile package parameters, when configured, will appear as in Figure 7-29.



Name	Data type	Value	Sensitive	Required
ArchiveDirectory	String	C:\Users\A. Ray Leonard\Documents\Archive	False	True
CopyOnly	Boolean	True	False	False
DateStampFormat	String	_yyyyMMdd_hhmmss	False	False
Debug	Boolean	True	False	False
ExceptionOnFileNotFound	Boolean	True	False	False
OverwriteDestination	Boolean	True	False	False
SourceFilePath	String	C:\Users\A. Ray Leonard\Documents\randomdata.csv	False	False

Figure 7-29. ArchiveFile package parameters

Be sure you include default values for an existing folder for the ArchiveDirectory parameter and a path to a valid file for the SourceFilePath parameter. For all other parameter default values, use what I have supplied in Figure 7-29.

There are a couple ways to design this package. You can rely heavily on scripting or utilize the File System task. Which should you choose? When consulting, I ask questions to determine the comfort-level of those charged with maintaining the packages. Some data integration developers are comfortable with .NET coding; others are not. Since SSIS gives me a choice, I build packages so they are easily maintained by the team charged with maintenance.

In this package, I am choosing a hybrid of scripting and the File System task, leaning away from scripting. Let's add the following variables to the package:

- User::FormattedFileName [String]
- User::OkToProceed [Boolean]
- User::SourceFileDirectory [String]
- User::WorkingCopyFileName [String]

Add a Script task to the control flow and name it **scr Apply Format**. Open the editor and change the ScriptLanguage property to Microsoft Visual Basic 2012. Add the following variables and parameters to the ReadOnlyVariables property:

- System::TaskName
- System::PackageName
- \$Package::CopyOnly
- \$Package::DateStampFormat
- \$Package::Debug
- \$Package::ExceptionOnFileNotFound
- \$Package::SourceFilePath

Add the following variables and parameters to the ReadWriteVariables property:

- User::FormattedFileName
- User::OkToProceed
- User::SourceFileDirectory
- User::WorkingCopyFileName

Click the Edit Script button to open the VSTAProjects script editor. At the top of the ScriptMain.vb file, add the statement from Listing 7-11 to the Imports region.

Listing 7-11. Declaring a Reference to the System.IO Namespace

```
Imports System.IO
```

Replace the code in Public Sub Main() with that from Listing 7-12.

Listing 7-12. Code for the Main() Subroutine

```
Public Sub Main()

    ' 1: declare bDebug
    Dim bDebug As Boolean

    ' 2: detect Debug mode...
    bDebug = Convert.ToBoolean(Dts.Variables("Debug").Value)

    ' 3:variables declaration...
    Dim sPackageName As String = Dts.Variables("System::PackageName").Value.ToString
    Dim sTaskName As String = Dts.Variables("System::TaskName").Value.ToString
    Dim sSubComponent As String = sPackageName & "." & sTaskName
    Dim sDateStampFormat As String = _
        Dts.Variables("$Package::DateStampFormat").Value.ToString
    Dim sSourceFilePath As String = _
        Dts.Variables("$Package::SourceFilePath").Value.ToString
    Dim bExceptionOnFileNotFound As Boolean = _
        Convert.ToBoolean(Dts.Variables("ExceptionOnFileNotFound").Value)
```



```

Dim bCopyOnly As Boolean = Convert.ToBoolean(Dts.Variables("CopyOnly").Value)
Dim sFileName As String
Dim sBaseFileName As String
Dim sExtension As String
Dim sSourceFileDirectory As String
Dim sWorkingCopyFileName As String
Dim sFormattedFileName As String
Dim sMsg As String

' 4: work with the file
Try
    ' 4a: parse the source file directory...
    sSourceFileDirectory = Strings.Trim(Strings.Left(sSourceFilePath, _
Strings.InStrRev(sSourceFilePath, "\")))
    ' 4b: parse the filename...
    sFileName = Strings.Trim(Strings.Right(sSourceFilePath, _
Strings.Len(sSourceFilePath) - Strings.InStrRev(sSourceFilePath, "\")))
    ' 4c: parse the filepath minus the extension...
    sBaseFileName = Strings.Left(sSourceFilePath, Strings.InStrRev(sSourceFilePath, _
".")) - 1)
    ' 4d: build working copy file name...
    sWorkingCopyFileName = sSourceFileDirectory & "_" & sFileName

    ' 4e: parse extension...
    sExtension = Strings.Trim(Strings.Right(sSourceFilePath, _
Strings.Len(sSourceFilePath) - Strings.InStrRev(sSourceFilePath, ".")))
    ' 4f: apply formatting to filename and set the output value of FormattedFileName
    sFormattedFileName = sBaseFileName & _
Strings.Format(Date.Now, sDateStampFormat) & "." & sExtension
    ' 4g: assign external variables...
    Dts.Variables("User::FormattedFileName").Value = sFormattedFileName
    Dts.Variables("SourceFileDirectory").Value = sSourceFileDirectory
    Dts.Variables("WorkingCopyFileName").Value = sWorkingCopyFileName

    ' 4h: check for valid file...
    If File.Exists(sSourceFilePath) Then
        ' 4i: set OkToProceed flag...
        Dts.Variables("OkToProceed").Value = True
    Else
        ' 4j: if raising an exception on file not found...
        If bExceptionOnFileNotFound Then
            ' 4k: fire an error...
            Dts.Events.FireError(1001, sSubComponent, "cannot locate file " & _
sSourceFilePath, "", 0)
        End If
        ' 4l: set OkToProceed flag...
        Dts.Variables("OkToProceed").Value = False
        sMsg = "file " & sSourceFilePath & " not found."
        If bDebug Then MsgBox(sMsg, MsgBoxStyle.OkOnly, sSubComponent)
        ' 4m: log file not found, exception or not...
        Dts.Events.FireInformation(2001, sSubComponent, sMsg, "", 0, True)
    End If
End Try

```

```

Catch ex As Exception
    ' 4n: log error message...
    Dts.Events.FireError(1001, sSubComponent, ex.Message, "", 0)
End Try

    ' 5: log information
    sMsg = "DateStampFormat: " & sDateStampFormat & ControlChars.CrLf & _
ControlChars.CrLf & _
        "ExceptionOnFileNotFound: " & bExceptionOnFileNotFound.ToString & _
ControlChars.CrLf & ControlChars.CrLf & _
        "CopyOnly: " & bCopyOnly.ToString & ControlChars.CrLf & ControlChars.CrLf & _
        "OkToProceed: " & Dts.Variables("OkToProceed").Value.ToString & _
ControlChars.CrLf & ControlChars.CrLf & _
        "SourceFileDirectory: " & sSourceFileDirectory & ControlChars.CrLf & _
ControlChars.CrLf & _
        "FileName: " & sFileName & ControlChars.CrLf & ControlChars.CrLf & _
        "Extension: " & sExtension & ControlChars.CrLf & ControlChars.CrLf & _
        "BaseFileName: " & sBaseFileName & ControlChars.CrLf & ControlChars.CrLf & _
        "FormattedFileName: " & sFormattedFileName & ControlChars.CrLf & _
ControlChars.CrLf & _
        "WorkingCopyFileName: " & sWorkingCopyFileName & ControlChars.CrLf & _
ControlChars.CrLf

    If bDebug Then MsgBox(sMsg, MsgBoxStyle.OkOnly, sSubComponent)
    Dts.Events.FireInformation(2001, sSubComponent, sMsg, "", 0, True)
    ' 6: output
    Dts.TaskResult = ScriptResults.Success
End Sub

```

As in other scripts, we declare (Dim) a variable named `bDebug` to detect whether the package is executing in Debug mode at comments 1 and 2. At comment 3, the script declares the remainder of the variables used, assigning some values passed in from SSIS package variables and parameters. At comments 4a through 4c, the code picks the `Source File Path` variable apart, parsing the source directory, the filename with extension, and the filename without extension. At comments 4d through 4f, the filename extension is parsed and a filename for a “working copy” is created and formatted with the datetime stamp supplied from the SSIS package parameters. At comment 4g, the script assigns variable values to SSIS package variables. The code between comments 4h and 4m tests and responds to the existence of the source file. If an exception is encountered in any of the steps between comments 4a and 4m, the `Catch` block at comment 4n is executed and logs the exception as an error, which halts the execution of the Script task. The code at comment 5 builds, displays (if running in Debug mode), and logs a message containing the variable values inside the Script task. This is extremely useful information when troubleshooting. At comment 6, the script returns a `Success` result to the `Dts.TaskResult` object.

The remaining steps in the file archive process are as follows:

1. Create a working copy of the source file.
2. Rename the working copy to the **Formatted File Name** (including the datetime stamp).
3. Move the newly-renamed file to the archive directory.
4. Delete the original file (unless this is a `CopyOnly` operation).

If the `OkToProceed` (Boolean) package variable is set to `True` (this is accomplished in the script code at comment 4i), the remaining steps in the process are managed by File System tasks.

Drag four File System tasks onto the control flow canvas. Rename the first **fsys Copy Working File** and open its editor. Change the Operation property to **Copy File**. Set the `IsSourcePathVariable` property to True and the `SourceVariable` property to `$Package::SourceFilePath`. Set the `IsDestinationPathVariable` to True and set the `DestinationVariable` property to `User::WorkingCopyFileName`. Set the `OverwriteDestination` property to True. Close the File System Task Editor.

■ **Note** Because we did not set a value for `User::WorkingCopyFileName`, there will be a red X on the task. Do not worry about that.

Rename the second File System task **fsys Rename File** and open its editor. Set the Operation property to **Rename File**. Set the `IsSourcePathVariable` property to True and the `SourceVariable` property to `User::WorkingCopyFileName`. Set the `IsDestinationPathVariable` to True and set the `DestinationVariable` property to `User::FormattedFileName`. Set the `OverwriteDestination` property to True. Close the File System Task Editor.

Rename the third File System task **fsys Move File** and open its editor. Set the Operation property to **Move File**. Set the `IsSourcePathVariable` property to True and the `SourceVariable` property to `User::FormattedFileName`. Set the `IsDestinationPathVariable` to True and set the `DestinationVariable` property to `$Package::ArchiveDirectory`. Set the `OverwriteDestination` property to True. Close the File System Task Editor.

Rename the fourth File System task **fsys Delete Original File** and open its editor. Set the Operation property to **Delete File**. Set the `IsSourcePathVariable` property to True and the `SourceVariable` property to `$Package::SourceFilePath`. Close the File System Task Editor.

Use a Success Precedence constraint to connect the `scr Apply Format Script` task to the `fsys Copy Working File` File System task. Double-click the Precedence constraint to open the editor and set the `Evaluation Option` property to `Expression and Constraint`. Set the `Value` property to `Success` and the `Expression` property to `@[User::OkToProceed]`. This constraint will only fire if the `scr Apply Format Script` task completes execution successfully and sets the `OkToProceed` (Boolean) variable to True. Connect Success Precedence constraints between the `fsys Copy Working File` File System task and the `fsys Rename File` File System task, the `fsys Rename File` File System task and the `fsys Move File` File System task, and the `fsys Move File` File System task and the `fsys Delete Original File` File System task. Double-click the Precedence constraint between the `fsys Move File` File System task and the `fsys Delete Original File` File System task to open the editor. Set the `Evaluation Option` property to `Expression and Constraint`. Set the `Value` property to `Success` and the `Expression` property to `![@($Package::CopyOnly)]` (this equates to NOT `[!] $Package::CopyOnly`, or when `$Package::CopyOnly` is False). For the `fsys Delete Original File` File System task to fire, the `fsys Move File` File System task must succeed and the `$Package::CopyOnly` package parameter must be False. This makes sense, if you only want to copy the file to the archive directory; you don't want to delete the original.

In many versions of this design pattern, I also “variable-ize” the `OverwriteDestination` properties of the various File System tasks, managing these values on the Expressions pages by setting the `OverwriteDestinationFile` dynamic property expressions with Boolean package parameters. I do this because some enterprises have requirements regarding keeping or discarding data files regardless of whether they are temporary or not.

Your File System tasks may be marked with error indicators (red circles containing white Xs). Hovering over a task so marked will display the error. For example, in Figure 7-30 the error: “Variable ‘WorkingCopyFileName’ is used as a source or destination and is empty.”

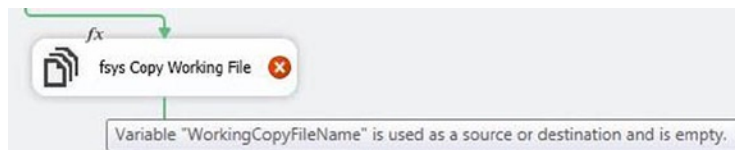


Figure 7-30. The *WorkingCopyFileName* Variable is empty

At issue is the content of the *WorkingCopyFileName* variable. The error is correct; the variable value is currently empty. However, since I wrote the code, I know that, in section 4d of the code listing, the script will populate the content of an internal string variable named *sWorkingCopyFile*. In section 4g of the code, the content of this internal variable will be assigned to the value of the SSIS package variable named *WorkingCopyFileName*. I know that, but the SSIS Package does not. It is doing its best to inform me of this issue. In fact, I cannot execute the package in its current state without raising an error message, as shown in Figure 7-31.

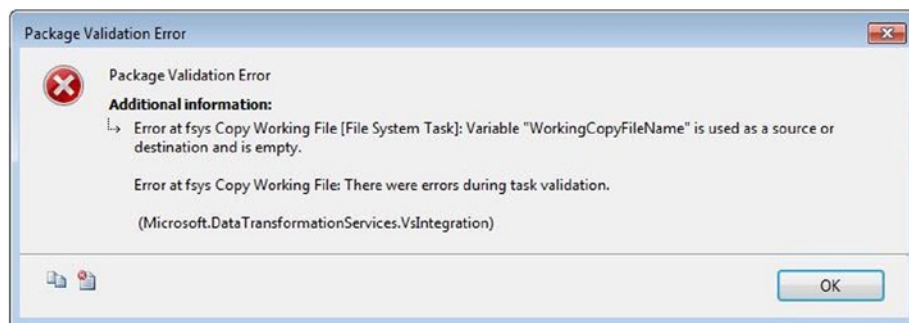


Figure 7-31. Package validation error

The validation is accurate. Now what? There's a clue in the error, near the very top. This is a Package Validation Error. To address this, click on the *fsys Copy Working File* File System task and press the F4 key to display Properties. In the Execution group of properties, at the top of the list, you will find the *DelayValidation* property. This property's default setting is *False* and that makes sense. There is a lot of design-time validation in SSIS and it is mostly a good thing. Change this property value to *True*. Change *DelayValidation* to *True* for the *fsys Rename File* and *fsys Move File* File System tasks as well.

Now, try executing the *ArchiveFile.dtsx* SSIS package. My results are shown in Figure 7-32.

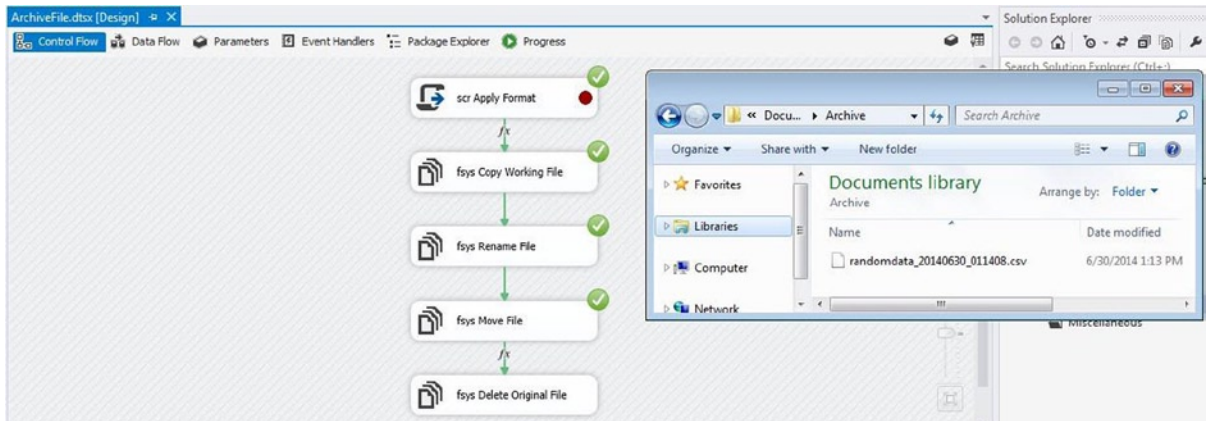


Figure 7-32. Successful execution of the *ArchiveFile.dtsx* SSIS package

Summary

In this chapter, we examined a common pattern for loading a basic flat source file into SQL Server, a pattern for loading variable-length rows, patterns for creating and consuming flat file header and footer rows, and an extremely useful SSIS design pattern for archiving flat files.



Loading a PDW Region in APS

SQL Server Parallel Data Warehouse (PDW) is Microsoft's massively parallel processing (MPP) offering and is available as part of Microsoft's Analytics Platform System (APS). APS is a turnkey solution focused on big data analytics. It offers two regions, or software options, to customers: PDW and HDInsight, which is Microsoft's 100% Apache Hadoop distribution. PDW is built upon the SQL Server platform, although it is a separate product with a build of SQL Server specifically designed to support MPP operations.

Massively Parallel Processing

As the name suggests, *massively parallel processing (MPP)* uses multiple servers working as one system, called an *appliance*, to achieve much greater performance and scan rates than in traditional SMP systems. SMP refers to *symmetric multiprocessing*; most database systems, such as all other versions of SQL Server, are SMP.

To obtain a better understanding of the difference between SMP and MPP systems, let's examine a common analogy. Imagine you are handed a shuffled deck of 52 playing cards and are asked to retrieve all of the queens. Even at your fastest, it would take you several seconds to retrieve the requested cards. Let's now take that same deck of 52 cards and divide it among ten people. No matter how quick you are, these ten people working together can retrieve all of the queens much faster than you can by yourself.

As you may have inferred, you represent the SMP system, and the ten people represent the MPP system. This divide-and-conquer strategy is why MPP appliances are particularly well suited for high-volume, scan-intensive data warehousing environments, especially ones that need to scale to hundreds of terabytes—or even petabytes!—of storage. There are other MPP appliance vendors available besides Microsoft; however, the ability to easily join unstructured data in HDInsight with structured data in PDW via PolyBase, close integration with the Microsoft business intelligence stack (SQL Server, Integration Services, Analysis Services, and Reporting Services), and a compelling cost-per-terabyte makes APS a natural progression for organizations that need to take their SQL Server data warehouse to the next level. In this chapter, we will walk through how to efficiently load data into a PDW region in an APS appliance using Integration Services. But before we do that, we will first explore the architecture of APS. APS could easily consume a book in its own right, so we will only be covering the most pertinent parts to ensure that you have the foundation necessary for efficiently loading data into PDW.

■ **Tip** Learn more about Microsoft's Analytics Platform System (APS), Parallel Data Warehouse (PDW), and HDInsight at <http://microsoft.com/aps>.

APS Appliance Overview

Generally, ETL developers and data integration engineers need not worry about the hardware specifications of the database systems they interact with. This isn't the case when you are loading MPP systems. Because MPP systems are designed to take advantage of distributed data and parallelized workloads, you will find tremendous performance benefits in designing ETL solutions that take advantage of the parallelization units of an MPP system. For that reason, we will briefly review general hardware and software architectures for Microsoft's APS appliance before we dive into load patterns.

Hardware Architecture

Microsoft has partnered and closely collaborated with Dell, HP, and Quanta to provide customers with a choice in hardware vendors. The hardware specifications of an APS appliance will vary depending on your chosen hardware vendor and capacity requirements, although there are some consistencies across the vendors.

■ **Note** For simplicity, we will examine APS configurations comprised only of PDW. It's worth noting, however, that both PDW and HDInsight can run side-by-side in the same appliance through the use of separate hardware regions dedicated to each application. Microsoft calls this configuration a *multi-region appliance*.

The *base rack* is the smallest unit available for purchase and contains everything required to run a SQL Server PDW region. Depending on the hardware vendor, the minimum configuration will include either two or three Compute nodes. Similarly, the *scale unit*—that is, the number of compute nodes required to increase the capacity of the appliance—is dependent on vendor and will either be two or three Compute nodes. For all base racks, irrespective of hardware vendor, the top of the base rack includes the following:

- Two redundant InfiniBand switches
- Two redundant Ethernet switches
- One active server, which contains VMs for
 - SQL Server PDW Control node
 - Management and Active Directory
 - Fabric Active Directory
 - Hyper-V Virtual Machine Manager
 - One passive spare server

The bottom of the base rack includes additional servers and storage. To add more storage or processing capacity, add scale units to the base rack until the base rack is full.

■ **Note** For simplicity and consistency, we will use depictions of Dell's hardware configurations. However, please do not mistake that as an endorsement on the part of the authors. We suggest you review the offerings of each hardware vendor and make the selection that best meets your requirements.

Once the base rack is full, you can continue to add near-linear scalability to your APS appliance by adding one or more expansion racks. All expansion racks, irrespective of hardware vendor, are comprised of the following:

- Two redundant InfiniBand switches
- Two redundant Ethernet switches
- Scale units of either two or three Compute nodes

The expansion rack is very similar to the base rack. Just like the base rack, the bottom of the expansion rack includes servers and storage in scale units of either two or three Compute nodes. In fact, the only difference between the base rack and the expansion rack is the absence of the active server and optional spare server at the top of the rack.

Software Architecture

In the previous section, we discussed that the primary difference between the base rack and the expansion rack is the active server at the top of the rack. Let's dig into that server in a little more detail.

The active server is comprised of four virtual machines (VMs) that provide functionality essential to the operation of the appliance. The four VMs are the PDW Control node (CTL), the Management and Active Directory (MAD), the Fabric Active Directory (AD), and Hyper-V Virtual Machine Manager (VMM). As an ETL developer or data integration engineer, you will only interact directly with the PDW Control node VM. Figure 8-1 depicts a nine-node rack and the four VMs that reside in the Active Server.

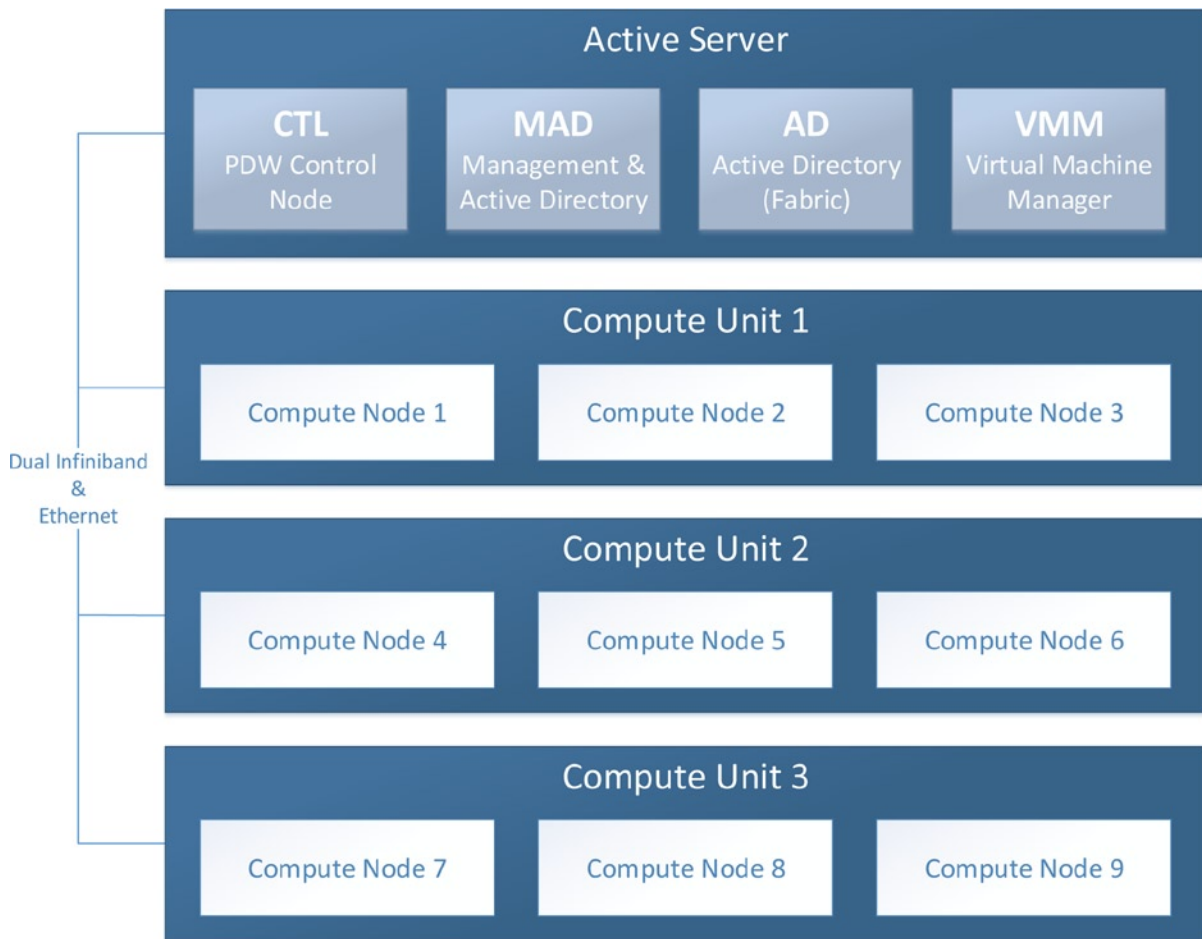


Figure 8-1. APS virtual machines

The PDW Control node VM provides the following critical functionality:

- Client connectivity and authentication
- System and database metadata
- SQL request processing
- Execution plan preparation
- Distributed execution orchestration
- Result aggregation

You can think of the Control node as the “brains” behind the distribution and parallelization of PDW. Although metadata resides on the Control node, no user data is persisted there. Rather, the Control node directs the loading and retrieval of user data to the appropriate Compute node and distribution. This is the first time we’ve introduced the term *distribution*, so if you’re not yet familiar with the term, don’t worry. We’ll cover distributions in the next section.

Shared-Nothing Architecture

At the core of PDW is the concept of *shared-nothing architecture*. In a shared-nothing architecture, a single logical table is broken up into numerous smaller physical pieces. The exact number of pieces depends on the number of Compute nodes in the PDW region. Within a single Compute node, each data piece is then split across eight (8) distributions. The number of distributions per Compute node cannot be configured and is consistent across all hardware vendors.

A distribution is the most granular physical level within PDW. Each distribution contains its own dedicated CPU, memory, and storage (LUNs), which it uses to store and retrieve data. Because each distribution contains its own dedicated hardware, it can perform load and retrieval operations in parallel with other distributions. This concept is what we mean by “shared-nothing.” There are numerous benefits that a shared-nothing architecture enables, such as more linear scalability. But perhaps PDW’s greatest power is its ability to scan data at incredible speeds.

Let’s do some math. Assume you have a PDW appliance with a base rack containing 9 Compute nodes, and you need to store a table with 1 billion rows. The data will be split across all 9 Compute nodes, and each Compute node will split its data across 8 distributions. Thus, the 1-billion-row table will be split into 72 distributions (9 Compute nodes $\times$ 8 distributions per Compute node). That means each distribution will store roughly 13,900,000 rows.

But what does this mean from the end user’s standpoint? Let’s look at a hypothetical situation. You are a user at a retail company and you have a query that joins two tables together: a Sales table with 1 billion rows, and a Customer table with 50 million rows. And, as luck would have it, there are no indexes available that will cover your query. This means you will need to *scan*, or read, every row in each table.

In an SMP system—where memory, storage, and CPU are shared—this query could take hours or days to run. On some systems, it might not even be feasible to attempt this query, depending on factors such as the server hardware and the amount of activity on the server. Suffice it to say, the query will take a considerable amount of time to return and will most likely have a negative impact on other activity on the server.

In PDW, these kinds of queries often return in minutes, and a well-designed schema, where the rows you’re joining are stored together on the same distribution, can even execute this query in seconds. This is because the architecture is *optimized* for scans; PDW *expects* to scan every row in the table. Remember how we said that every distribution has its own dedicated CPU, memory, and storage? When you submit the query to join 1 billion rows to 50 million rows, each distribution is performing a scan on its own Sales table of 13,800,000 rows and Customer table with 695,000 rows. These smaller volumes are much more manageable, and an idle distribution can handle this workload with ease. The data is then sent back to the Control node across an ultra-fast dual-InfiniBand channel to consolidate the results and return the data to the end user. It is this divide-and-conquer strategy that allows PDW to significantly outperform SMP systems.

Clustered Columnstore Indexes

Columnstore refers to the storing of data in a columnar—or column-oriented—storage format. In traditional RDBMS systems, data is stored using a *rowstore*—or row-oriented—storage format. Rowstores are generally well suited for transactional applications where the application is concerned with most or all columns for one or a small number of rows. Columnstores, on the other hand, are better suited for analytical applications, which are generally concerned with a subset of columns and a large portion, or even all, of the rows in the table.

For those of you familiar with indexing in SQL Server, you may think of a clustered columnstore index (CCI) as analogous to a clustered index in a row-oriented table. But unlike a clustered index, once a CCI has been defined, additional indexes such as nonclustered indexes may not be created on that table.

CCIs offer several performance improvements over traditional rowstores in PDW. Some customers have seen up to ten times the query performance and up to seven times the data compression improvements. Because of the compression and related performance improvements, Microsoft recommends CCI as the standard storage format for tables in PDW.

■ **Tip** Columnstores are available in both PDW and SQL Server Enterprise Edition. For more information on columnstore indexes, refer to the “Columnstore Indexes Described” topic on MSDN, or navigate to <http://msdn.microsoft.com/en-us/library/gg492088>.

Loading Data

We discussed in the previous section that PDW is able to query data very efficiently because of its shared-nothing architecture. For the same reason, PDW is also able to load data very efficiently. Let’s briefly discuss how PDW can perform data imports so efficiently.

As previously mentioned, the Control VM is the first stop for anything being written or retrieved from the appliance. The Control node determines which Compute nodes will be involved in the storage operation. Each Compute node then uses a hashing algorithm to determine where to store the data, down to the individual distribution and associated LUNs. This allows each distribution to load its data in parallel with other distributions. Again, dividing and conquering a large table import in parallel will be much faster than performing a single large import or performing several smaller imports serially.

Data can be imported from numerous platforms, including from Oracle, SQL Server, MySQL, and flat files. There are two primary methods of loading data into the PDW appliance: DWLoader and Integration Services. We will briefly discuss when to use DWLoader vs. Integration Services. After that, we will walk through an example of loading data from SQL Server using Integration Services.

DWLoader vs. Integration Services

DWLoader is a command-line utility that ships with PDW. Those familiar with SQL Server BCP (bulk copy program) will have an easy time learning DWLoader, because both utilities share a very similar syntax. One very common pattern for loading data into PDW from SQL Server is to

1. Export data from SQL Server to a flat file using BCP.
2. Store the flat file on the loading server.
3. Import the data file from the loading server to PDW using DWLoader.

This is a very efficient method for importing data, and it is very easy to generate scripts for table DDL, BCP commands, and DWLoader commands. For this reason, you may want to consider DWLoader for performing initial and incremental loading of the large quantity of small dimensional tables that often exist in data warehouses. Doing so can greatly speed up a data warehouse migration. This same load pattern can also be used with flat files generated from any system, not just SQL Server.

For your larger tables, you may instead want to consider Integration Services. Integration Services offers greater functionality and arguably more end-to-end convenience. This is because Integration Services is able to connect directly to the data source and load the data into the PDW appliance without having to stop at a file share. Another important distinction is that Integration Services can also perform transformations in flight, which DWLoader does not support.

It’s worth noting that each data flow within Integration Services is single-threaded and can bottleneck on I/O. Typically, a single-threaded Integration Services package will perform up to ten times slower than DWLoader. However, a multithreaded Integration Services package—similar to the one we will create shortly—can mitigate that limitation. For large tables requiring data type conversions, an Integration Services package with ten parallel data flows provides the best of both worlds: similar performance to DWLoader and all the advanced functionality that Integration Services offers.

You should consider a number of variables when deciding whether to use DWLoader or Integration Services. In addition to table size, both network speed and table design can have an impact. At the end of the day, most PDW implementations will likely use a combination of both tools. The best idea is to test the performance of each method in your environment and use the tool that makes the most sense for each table pattern.

ETL vs. ELT

Many Integration Services packages are designed using an extract, transform, and load (ETL) process. This is a practical model that strives to lessen the impact of moving data on the source and destination servers, which are traditionally more resource-constrained, by placing the burden of data filtering, cleansing, and other such activities on the (arguably more easy-to-scale) ETL server. Extract, load, and transform (ELT) processes, in contrast, place the burden on the destination server.

Although both models have their place and PDW supports both, ELT makes more sense with PDW from both a technical and a business perspective. On the technical side, PDW is able to utilize its massively parallel processing (MPP) power to more efficiently load and transform large volumes of data. From the business aspect, having more data co-located allows more meaningful data to be gleaned during the transformation process. Organizations with MPP systems often find that the ability to co-locate and transform large quantities of disparate data enables them to make the leap from reactive data marts (*How much of this product did we sell?*) to predictive data modeling (*How can we sell **more** of this product?*).

Just because you have decided on an ELT strategy does not necessarily mean your Integration Services package will not have to perform any transformations. In fact, many Integration Services packages may require data type transformations. Table 8-1 illustrates the data types supported in PDW and the equivalent Integration Services data types.

Table 8-1. Data Type Mappings for PDW and Integration Services

SQL Server PDW Data Type	Integration Services Data Type(s) That Map to the SQL Server PDW Data Type
BIT	DT_BOOL
BIGINT	DT_I1, DT_I2, DT_I4, DT_I8, DT_UI1, DT_UI2, DT_UI4
CHAR	DT_STR
DATE	DT_DBDATE
DATETIME	DT_DATE, DT_DBDATE, DT_DBTIMESTAMP, DT_DBTIMESTAMP2
DATETIME2	DT_DATE, DT_DBDATE, DT_DBTIMESTAMP, DT_DBTIMESTAMP2
DATETIMEOFFSET	DT_WSTR
DECIMAL	DT_DECIMAL, DT_I1, DT_I2, DT_I4, DT_I8, DT_NUMERIC, DT_UI1, DT_UI2, DT_UI4, DT_UI8
FLOAT	DT_R4, DT_R8
INT	DT_I1, DT_I2, DT_I4, DT_UI1, DT_UI2
MONEY	DT_CY
NCHAR	DT_WSTR
NUMERIC	DT_DECIMAL, DT_I1, DT_I2, DT_I4, DT_I8, DT_NUMERIC, DT_UI1, DT_UI2, DT_UI4, DT_UI8
NVARCHAR	DT_WSTR, DT_STR

(continued)

Table 8-1. (continued)

SQL Server PDW Data Type	Integration Services Data Type(s) That Map to the SQL Server PDW Data Type
REAL	DT_R4
SMALLDATETIME	DT_DBTIMESTAMP2
SMALLINT	DT_I1, DT_I2, DT_UI1
SMALLMONEY	DT_R4
TIME	DT_WSTR
TINYINT	DT_I1
VARBINARY	DT_BYTES
VARCHAR	DT_STR

- Also, it is worth noting that PDW does not currently support the following data types at the time of this writing:
- DT_DBTIMESTAMPOFFSET
 - DT_DBTIME2
 - DT_GUID
 - DT_IMAGE
 - DT_NTEXT
 - DT_TEXT

Any of these unsupported data types will need to be converted to a compatible data type using the Data Conversion transformation. We will walk through how to perform such a transformation in just a moment.

Data Import Pattern for PDW

Now that you have a basic understanding of the architecture and loading concepts of an APS appliance, we’re ready to get started with the pattern for importing data into PDW.

Prerequisites

All Integration Services packages will be executed from a loading server. A *loading server* is a non-appliance server that resides on your network and runs Windows Server 2008 R2 or newer. The loading server is connected to the APS appliance via Ethernet or InfiniBand, the latter of which is recommended for better performance. Additionally, the loading server will require permissions to access the Control node and must have Integration Services and the PDW destination adapter installed.

The requirements for the PDW destination adapter will vary depending on the version of PDW and Integration Services you are running. The PDW documentation outlines the various requirements and installation instructions in great detail. Please refer to the “Install Integration Services Destination Adapters (SQL Server PDW)” section of your PDW documentation for more information.

Once the PDW destination adapter is installed on the loading server, we’re ready to create mock data.

Preparing the Data

In preparation for moving data from SQL Server to PDW, you need to create a database in SQL Server and populate it with some test data. Execute the T-SQL code in Listing 8-1 from your favorite query editor, such as SQL Server Management Studio (SSMS), to create a new database called PDW_Example.

Listing 8-1. Example of T-SQL Code to Create a SQL Server Database

```
USE [master];
GO

/* Create a database to experiment with */
CREATE DATABASE [PDW_Example]
    ON PRIMARY
    (
        NAME = N'PDW_Example'
        , FILENAME = N'C:\Program Files\Microsoft SQL Server\MSSQL12.MSSQLSERVER\MSSQL\DATA\PDW_
Example.mdf'
        , SIZE = 1024MB
        , MAXSIZE = UNLIMITED
        , FILEGROWTH = 1024MB
    )
    LOG ON
    (
        NAME = N'PDW_Example_Log'
        , FILENAME = N'C:\Program Files\Microsoft SQL Server\MSSQL12.MSSQLSERVER\MSSQL\DATA\PDW_
Example_Log.ldf'
        , SIZE = 256MB
        , MAXSIZE = UNLIMITED
        , FILEGROWTH = 256MB
    );
GO
```

Please note that your database file path will vary depending on the details of your particular installation.

Next, create a table and populate it with some data. As we discussed before, Integration Services works best with large tables that can be multithreaded. One good example of this is a Sales Fact table that is partitioned by year. Listing 8-2 will provide the T-SQL code you need to create the table and partitioning dependencies.

Listing 8-2. Example of T-SQL Code to Create a Partitioned Table in SQL Server

```
USE PDW_Example;
GO

/* Create your partition function */
CREATE PARTITION FUNCTION example_yearlyDateRange_pf
(DATETIME) AS RANGE RIGHT
FOR VALUES('2013-01-01', '2014-01-01', '2015-01-01');
GO

/* Associate your partition function with a partition scheme */
CREATE PARTITION SCHEME example_yearlyDateRange_ps
AS PARTITION example_yearlyDateRange_pf ALL TO([Primary]);
GO
```

```

/* Create a partitioned fact table to experiment with */
CREATE TABLE PDW_Example.dbo.FactSales (
    orderID          INT IDENTITY(1,1)
    , orderDate      DATETIME
    , customerID     INT
    , webID          UNIQUEIDENTIFIER DEFAULT (NEWID())

    CONSTRAINT PK_FactSales
    PRIMARY KEY CLUSTERED
    (
        orderDate
    , orderID
    )
) ON example_yearlyDateRange_ps(orderDate);

```

■ **Note** Getting an error on the above syntax? Partitioning is a feature only available in SQL Server Enterprise and Developer Editions. If you are not using Enterprise or Developer Editions, you can comment out the partitioning in the last line

```

ON example_yearlyDateRange_ps(orderDate);

and replace it with

ON [Primary];

```

Next, you need to generate data using the T-SQL in Listing 8-3. This is the data you will be loading into PDW.

Listing 8-3. Example of T-SQL Code to Populate a Table with Sample Data

```

/* Declare variables and initialize with an arbitrary date */
DECLARE @startDate DATETIME = '2013-01-01';

/* Perform an iterative insert into the FactSales table */
WHILE @startDate < '2016-01-01'
BEGIN

    INSERT INTO PDW_Example.dbo.FactSales (orderDate, customerID)
    SELECT @startDate
        , DATEPART(WEEK, @startDate) + DATEPART(HOUR, @startDate);

    /* Increment the date value by hour; for more test data,
    replace HOUR with MINUTE or SECOND */
    SET @startDate = DATEADD(HOUR, 1, @startDate);

END;

```

This script will generate roughly 26,000 rows in the FactSales table spanning 3 years, although you can easily increase the number of rows generated by replacing HOUR in the DATEADD statement with MINUTE or even SECOND.

Now that you have a data source to work with, you are ready to start working on your Integration Services package.

Package Overview

Let's discuss what your package will do. You are going to configure a data flow that will move data from SQL Server to PDW. You will create a connection to your data source via an OLE DB Source. Because UNIQUEIDENTIFIER (also known as a GUID) is not yet supported as a data type in PDW, you will transform the UNIQUEIDENTIFIER to a Unicode string (DT_WSTR) using a data conversion. You will then configure the PDW destination adapter to load data into the APS appliance. Lastly, you will multithread the package, which takes advantage of PDW's parallelization to improve load performance.

One easy way to multithread is to create multiple data flows that execute in parallel for the same table. You can have up to ten simultaneous loads—ten data flows—for a table. The challenge with simultaneous loading, however, is to avoid causing too much contention on the source system. You can minimize contention by isolating each data flow to a separate, equal portion of the clustered index. Better yet, if you have SQL Server Enterprise Edition, you can isolate each data flow by loading by partition. This latter method is preferable and is the approach our example will use.

Now that you understand the general structure of the Integration Services package, let's create it.

The Data Source

If you have not already done so, create a new Integration Services project named **PDW_Example** (File ► New ► Project ► Integration Services Project).

Add a Data Flow task to the control flow designer surface. Name it **PDW Import**.

Add an OLE DB Source to the designer surface from the SSIS Toolbox. Edit the properties of the OLE DB Source by double-clicking on the icon. You should see the source editor shown in Figure 8-2.

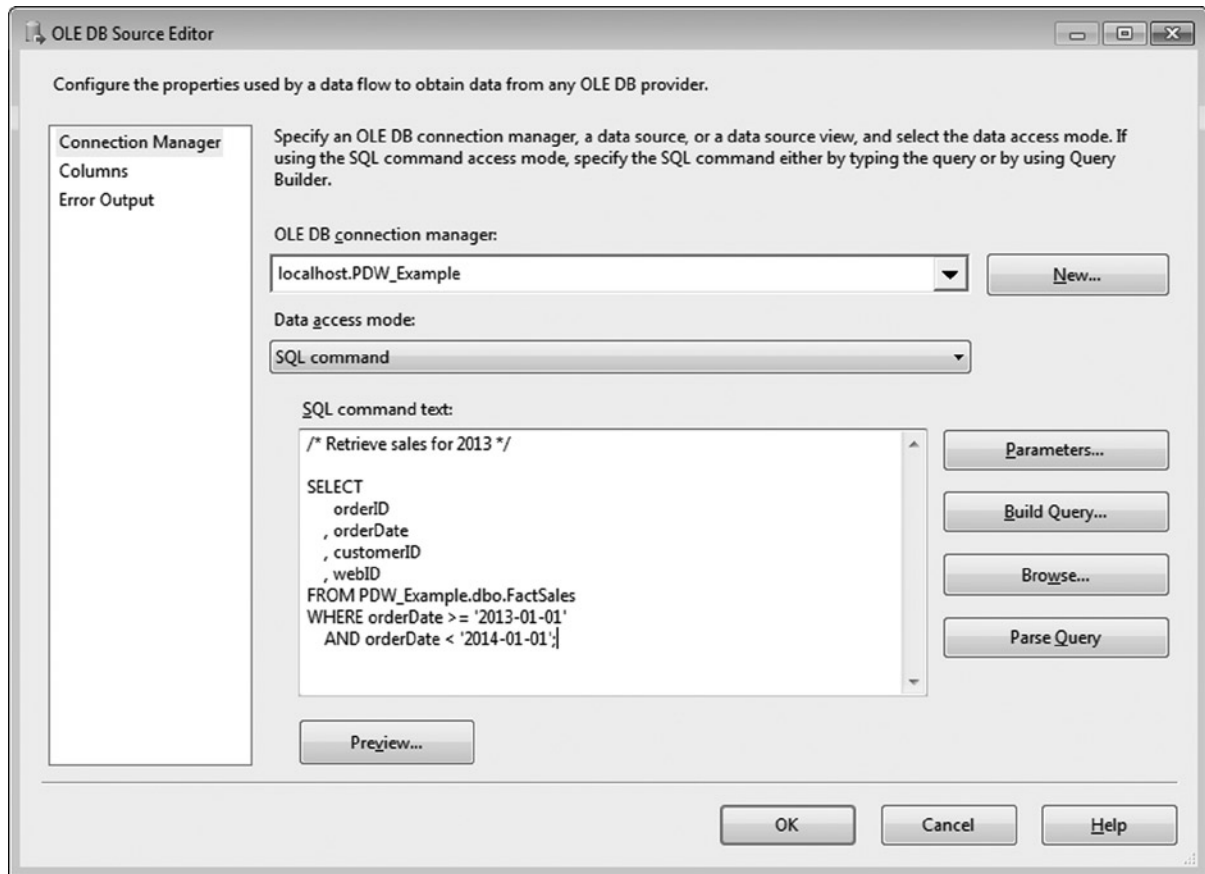


Figure 8-2. The OLE DB Source Editor

You will need to create an OLE DB Connection Manager that points to the PDW_Example database. Once this is done, change the data access mode to **SQL Command**; then enter the code in Listing 8-4.

Listing 8-4. Example SQL Command

```
/* Retrieve sales for 2013 */

SELECT
    orderID
    , orderDate
    , customerID
    , webID
FROM PDW_Example.dbo.FactSales
WHERE orderDate >= '2013-01-01'
    AND orderDate < '2014-01-01';
```

Your OLE DB Source Editor should look similar to Figure 8-2. Click Preview to verify the results, then click OK to close the editor.

This code is simple, but it's doing something pretty important. By searching the FactSales table on orderDate—the column specified as the partitioning key in Listing 8-2—SQL Server is able to perform partition elimination, which is important for minimizing I/O contention. This provides a natural boundary for each data flow that is both easy to understand and performs well. You can achieve a similar result even without partitioning FactSales by performing a sequential seek on the clustered index, orderDate. But what if FactSales was clustered on just orderID instead? You can apply the same principles and achieve good performance by searching for an evenly distributed number of sequential rows in each data flow. For example, if FactSales has 1,000,000 rows and we are using 10 data flows, each OLE DB Source should search for 100,000 rows (i.e., `orderID >= 1 and orderID < 100000`; `orderID >= 100000 and orderID < 200000`; and so on). These types of design considerations can have a significant impact on the overall performance of your Integration Services package.

■ **Tip** Not familiar with partitioning? Table partitioning is particularly well suited for large data warehouse environments and offers more than just the benefits briefly mentioned here. More information is available in the whitepaper, “Partitioned Table and Index Strategies Using SQL Server 2008,” at <http://msdn.microsoft.com/en-us/library/dd578580.aspx>.

The Data Transformation

As you may recall, the source table has a UNIQUEIDENTIFIER column that is stored as a CHAR(38) column in PDW. In order to load this data, we will need to transform the UNIQUEIDENTIFIER to a Unicode string. To do this, drag the Data Conversion icon from the SSIS Toolbox to the designer surface. Next, connect the blue data flow arrow from OLE DB Source to Data Conversion, as shown in Figure 8-3.

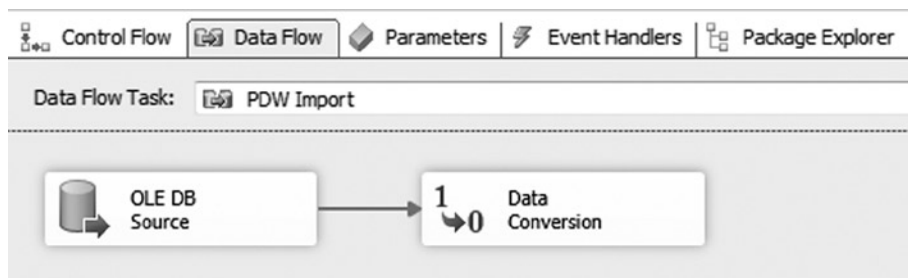


Figure 8-3. Connecting the OLE DB Source to the Data Conversion

Double-click on the Data Conversion icon to open the Data Conversion Transformation Editor. Click on the box to the left of webID; then edit its properties to reflect the following values:

Input Column: webID

Output Alias:

Data Type: string [DT_WSTR]

Length: 38

Confirm that the settings match those in Figure 8-4, then click OK.

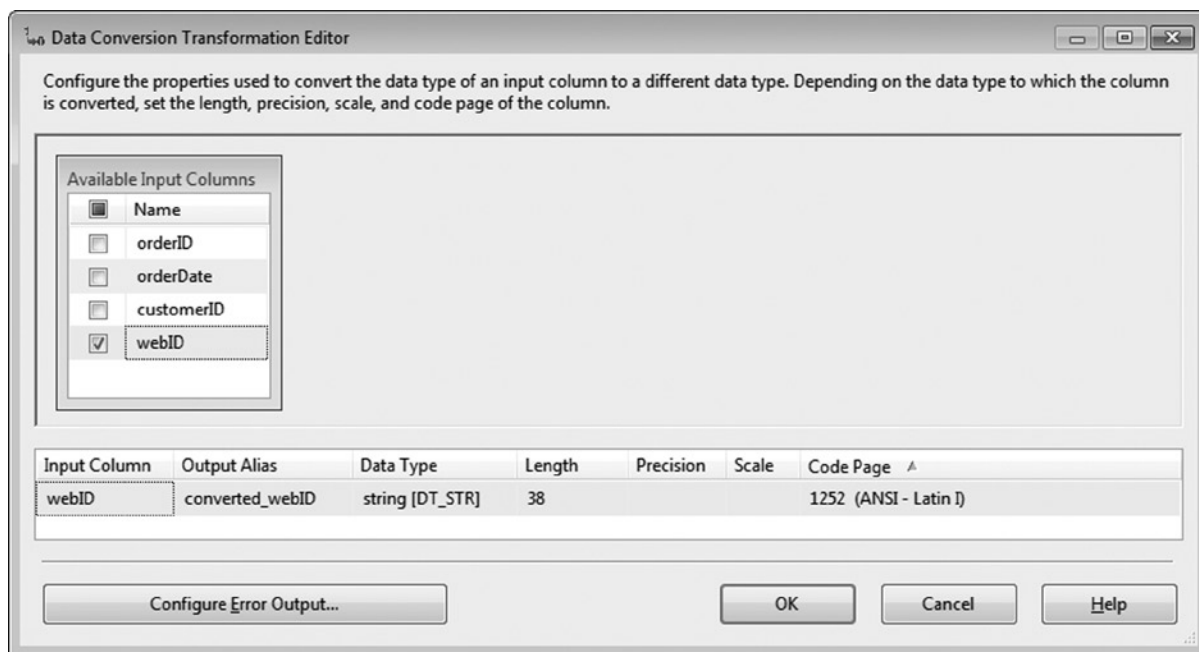


Figure 8-4. The Data Conversion Transformation Editor

■ **Tip** Wonder why we use a string length of 38 when converting a UNIQUEIDENTIFIER to a CHAR? This is because the global representation of a GUID is {00000000-0000-0000-0000-000000000000}. The curly brackets are stored implicitly in SQL Server for UNIQUEIDENTIFIER columns. Thus, during conversion, Integration Services materializes the curly brackets for export to the destination system. That is why, although a UNIQUEIDENTIFIER may *look* like it would only consume 36 bytes, it actually requires 38 bytes to store in PDW.

The Data Destination

The next few tasks, which prepare the PDW appliance for receiving FactSales data, will take place in SQL Server Data Tools—or just SSDT for short. Please refer to the section, “Install SQL Server Data Tools for Visual Studio (SQL Server PDW),” in the PDW documentation for more information on connecting to PDW from SSDT.

Before we go any further, we should discuss the use of a *staging database*. Although it is not required, Microsoft recommends that you use a staging database during incremental loads to reduce table fragmentation. When you use a staging database, the data is first loaded into a temporary table in the staging database before it is inserted into the permanent table in the destination database.

■ **Tip** Using a staging database? Make sure your staging database has enough space available to accommodate all tables being loaded concurrently. If you do not allocate enough space initially, don't worry; you'll still be okay—the staging database will autogrow. Your loads may just slow down while the autogrow is occurring. Also, your staging database will likely need to be larger when you perform the initial table loads during system deployment and migration. However, once your system becomes more mature and the initial ramp-up is complete, you can recover some space by dropping and re-creating a smaller staging database.

From within SSDT, execute the code in Listing 8-5 on your PDW appliance to create a staging database.

Listing 8-5. PDW Code to Run from SSDT to Create a Staging Database

```
CREATE DATABASE StageDB_Example
WITH
(
    AUTOGROW                = ON
    , REPLICATED_SIZE        = 1 GB
    , DISTRIBUTED_SIZE       = 5 GB
    , LOG_SIZE               = 1 GB
);
```

PDW introduces the concept of replicated and distributed tables. In a *distributed table*, the data is split across all nodes using a distribution hash specified during table creation. In a *replicated table*, the full table data exist on every Compute node. When used correctly, a replicated table can often improve join performance. As a hypothetical example, consider a small DimCountry dimension table with 200 rows. DimCountry would likely be replicated, whereas a much larger FactSales table would be distributed. This design allows any joins between FactSales and DimCountry to take place locally on each node. Although you would essentially be creating ten copies of DimCountry, one on each Compute node, the performance benefit of a local join outweighs the minimal cost of storing duplicate copies of such a small table.

Let's take another look at the CREATE DATABASE code in Listing 8-5. REPLICATED_SIZE specifies space allocation for replicated tables *on each Compute node*, whereas DISTRIBUTED_SIZE specifies space allocation for distributed tables *across the appliance*. That means StageDB_Example actually has 16GB of space allocated: 10GB for replicated tables (10 Compute nodes with 1GB each), 5GB for distributed tables, and 1GB for the log.

All data is automatically compressed using page-level compression during the PDW load process. This is not optional, and the amount of compression will vary greatly from customer to customer and table to table. If you have SQL Server Enterprise or Developer Editions, you can execute the statement in Listing 8-6 to estimate compression results.

Listing 8-6. Code to Estimate Compression Savings

```
/* Estimate compression ratio */
EXECUTE sp_estimate_data_compression_savings
'dbo', 'FactSales', NULL, NULL, 'PAGE';
```

Example results of sp_estimate_data_compression_savings are shown in Figure 8-5.

```

1  /* Estimate compression ratio */
2  EXECUTE sp_estimate_data_compression_savings
3  'dbo', 'FactSales', NULL, NULL, 'PAGE';

```

	object_name	schema_name	index_id	partition_number	size_with_current_compression_setting(KB)	size_with_requested_compression_setting(KB)
1	FactSales	dbo	1	1	0	0
2	FactSales	dbo	1	2	376	328
3	FactSales	dbo	1	3	376	328
4	FactSales	dbo	1	4	376	328

Figure 8-5. Example compression savings

You can generally use 2:1 as a rough estimate. With a 2:1 compression ratio, the 5GB of distributed data specified in Listing 8-5 actually stores 10GB of uncompressed SQL Server data.

You still need a place in PDW to store the data you're importing. Execute the code in Listing 8-7 in SSDT to create the destination database and table for FactSales.

Listing 8-7. PDW Code to Create the Destination Database and Table

```

CREATE DATABASE PDW_Destination_Example
WITH
(
    REPLICATED_SIZE          = 1 GB
    , DISTRIBUTED_SIZE       = 5 GB
    , LOG_SIZE                = 1 GB
);

CREATE TABLE PDW_Destination_Example.dbo.FactSales
(
    orderID                  INT
    , orderDate               DATETIME
    , customerID              INT
    , webID                   CHAR(38)
)
WITH
(
    DISTRIBUTION = HASH (orderID)
    , CLUSTERED COLUMNSTORE INDEX
);

```

Now that the destination objects are created, we can return to the Integration Services package. Drag the SQL Server PDW Destination from the Toolbox to the Data Flow pane. Double-click on the SQL Server PDW Destination, illustrated in Figure 8-6, to edit its configuration.



Figure 8-6. The SQL Server PDW Destination

Next, click on the down arrow next to Connection Manager and select <Create New Connection...>, as shown in Figure 8-7.

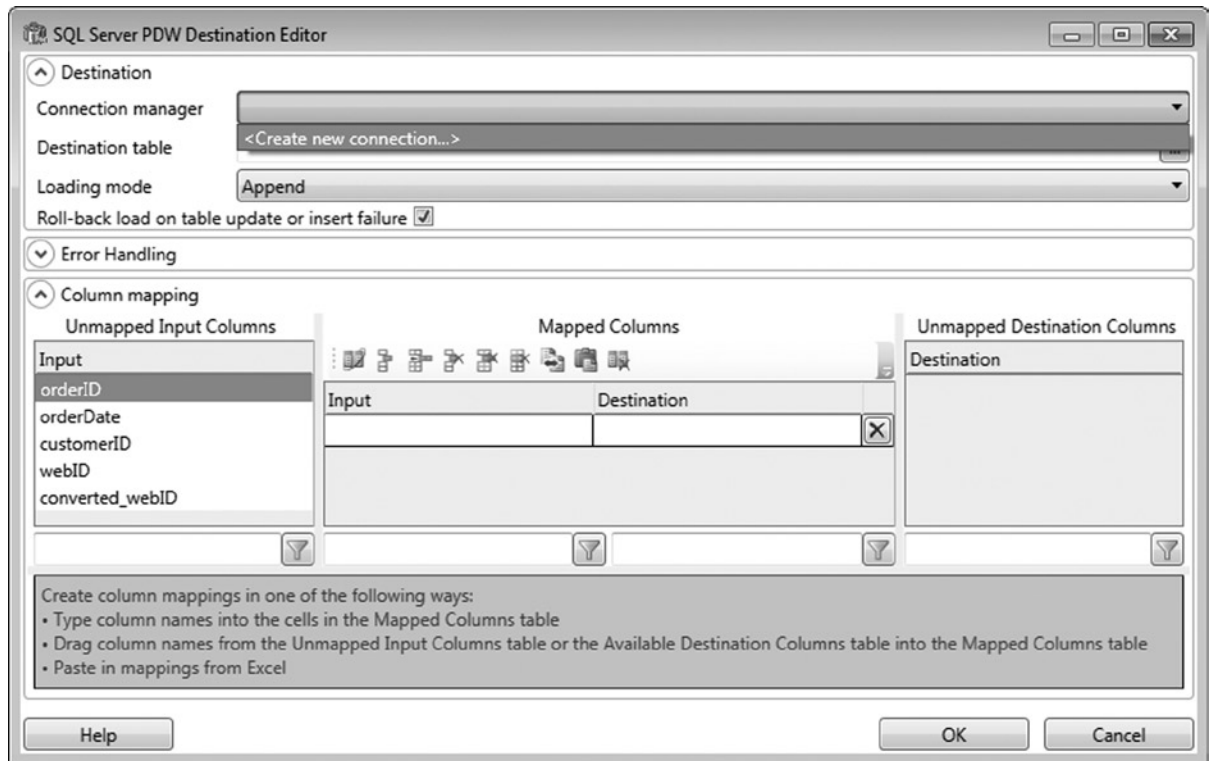


Figure 8-7. The SQL Server PDW Destination Editor

Enter your connection information in the SQL Server PDW Connection Manager Editor using these items:

Server: The IP address of the Control node on your appliance (Best practice is to use the clustered IP address to support Control node failover.)

User: Your login name for authenticating to the appliance

Password: Your login password

Destination Database: PDW_Destination_Example

Staging Database: StageDB_Example

Let's discuss a few best practices relating to this connection information. First, you should specify the IP address of the Control node cluster instead of the IP address of the active Control node server. Using the clustered IP address will allow your connection to still resolve without manual intervention in the event of a Control node failover.

Secondly, PDW supports both SQL Server and Active Directory authentication. When using SQL Server authentication, best practice is to use an account other than sa. Doing so will improve the security of your PDW appliance.

Lastly, as we previously discussed, Microsoft recommends the use of a staging database for data loads. The staging database is selected in the Staging Database Name drop-down. This tells PDW to first load the data to a temporary table in the specified staging database before loading the data into the final destination database. This is optional, but loading directly into the destination database will increase fragmentation.

When you are done, your SQL Server PDW Connection Manager Editor should resemble Figure 8-8. Click on Test Connection to confirm your information was entered correctly, then click OK to return to the SQL Server PDW Destination Editor.

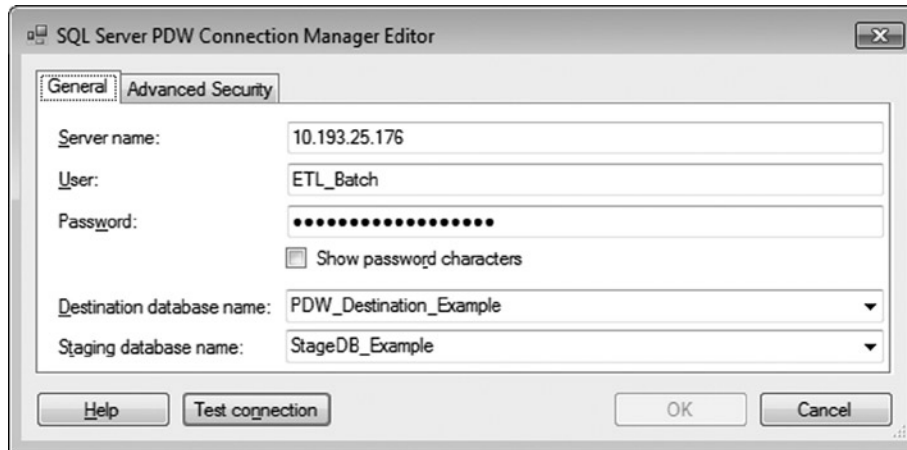


Figure 8-8. The SQL Server PDW Connection Manager Editor

■ **Note** If the staging database is not specified, SQL Server PDW will perform the load operation directly within the destination database, which can lead to high levels of table fragmentation.

Clicking on the Destination Table field will bring up a model for Select Destination Table. Click on FactSales. There are four loading modes available:

Append: Inserts the rows at the end of existing data in the destination table. This is the mode you are probably most used to.

Reload: Truncates the table before load.

Upsert: Performs a MERGE on the destination table, where new data is inserted and existing data is updated. You will need to specify one or more columns that will be used to join the data.

FastAppend: As its name implies, FastAppend is the fastest way to load data into a destination table. The trade-off is that it does not support rollback; in the case of a failure, you are responsible for removing any partially inserted rows. FastAppend will also bypass the staging database, causing high levels of fragmentation.

Let's take a moment to discuss how to use these modes with two common load patterns. If you are performing regular, incremental loads on a large table (say, updating a transactional sales table with the previous day's orders), you should load the data directly using Append, since no transformations are required. Now let's say you're loading the same data, but you plan to instead transform the data and load into a mart before deleting the temporary data. This second example would be better suited to the FastAppend mode. Or, to say it more concisely, use FastAppend any time you are loading into an empty, intermediate working table.

There is one last option we need to discuss. Underneath the Loading Mode is a checkbox for Roll-Back Load on Table Update or Insert Failure. In order to understand this option, you need to understand a little about how data is loaded into PDW. When data is loaded using the Append, Reload, or Upsert modes, PDW performs a two-phase load. In Phase 1, the data is loaded into the staging database. In Phase 2, PDW performs an INSERT/SELECT of the sorted data into the final destination table. By default, data is loaded in parallel on all Compute nodes, but it is loaded serially within a Compute node to each distribution. This is necessary in order to support rollback. Roughly 85–95% of the load process is spent in Phase 1. When Roll-Back Load on Table Update or Insert Failure is deselected, each distribution is loaded in parallel instead of serially during Phase 2. So, in other words, deselecting this option will improve performance but only affects 5–15% of the overall process. Also, deselecting this option removes PDW's ability to roll back; in the event of a failure during Phase 2, you would be responsible for cleaning up any partially inserted data.

Because of the potential risk and minimal gain, it is best practice to deselect this option only when you are loading to an empty table. FastAppend is unaffected by this option because it always skips Phase 2 and loads directly into the final table, which is why FastAppend also does not support rollback.

■ **Tip** Roll-Back Load on Table Update or Insert Failure is also available in DWLoader using the `-m` option.

Return to the PDW Destination Editor and select Append in the Loading Mode field. Because the destination table is currently empty, deselect the Roll-Back Load on Table Update or Insert Failure option to receive a small, risk-free performance boost.

You are almost done with your first data flow. All you have left to do is to map your data. Drag the blue arrow from the Data Conversion box to the SQL Server PDW Destination box, and then double-click on SQL Server PDW Destination. Map your input and destination columns. Make sure to map `webID` to your transformed `converted_webID` column. Click OK.

You have now successfully completed your first data flow connecting SQL Server to PDW. All you have left is to multithread the package.

Multithreading

You have completed the data flow for 2013, but you still need to create identical data flows for 2014 and 2015. You can do this easily by using copy and paste.

First, click on the Control Flow tab and rename the first Data Flow **SalesMart 2013**. Then, copy and paste the first data flow and rename it **SalesMart 2014**.

Double-click on SalesMart 2014 to return to the Data Flow designer, then double-click on the OLE DB Source. Replace the SQL Command with the code in Listing 8-6.

Listing 8-6. SQL Command for 2014 Data

```

/* Retrieve sales for 2014 */
SELECT
    orderID
    , orderDate
    , customerID
    , webID
FROM PDW_Example.dbo.FactSales
WHERE orderDate >= '2014-01-01'
      AND orderDate < '2015-01-01';

```

Return to the Control Flow tab and copy the SalesMart 2013 data flow again. Rename it **Sales Mart 2015**. Using the code in Listing 8-7, replace the SQL command in the OLE DB Source.

Listing 8-7. SQL Command for 2015 Data

```

/* Retrieve sales for 2015 */
SELECT
    orderID
    , orderDate
    , customerID
    , webID
FROM PDW_Example.dbo.FactSales
WHERE orderDate >= '2015-01-01'
      AND orderDate < '2016-01-01';

```

You are now ready to execute the package! Press F5 or navigate to Debug ► Start Debugging. Your package should execute, and you should see a successful result.

Limitations

ETL developers and data integration engineers should be aware of the following limitations and behaviors in PDW when developing ETL solutions:

- PDW is limited to a maximum of ten (10) active, concurrent loads. This number applies to the appliance as a whole, regardless of the number of Compute nodes, and cannot be changed.
- Each PDW destination adapter defined in an Integration Services package counts as one concurrent load. An Integration Services package with more than ten PDW destination adapters defined will not execute.
- As previously mentioned, not all SQL Server data types are supported in PDW. Refer to the list in Table 8-1 under “ETL vs. ELT,” or in your PDW documentation, for more details.

A small amount of overhead is associated with each load operation. This overhead is a necessary part of operating in a distributed environment. For example, the loading server needs to communicate with the Control node to identify which Compute nodes should receive data. Because this overhead is incurred on every single load, transactional load patterns (such as singleton inserts) should be avoided. PDW performs at its best when data is loaded in large, incremental batches. You will see much better performance loading 10 files with a 100,000 rows each, or a single file with 1,000,000 rows, than loading 1,000,000 rows individually.

Summary

We've covered a lot of material in this chapter. You have learned about the architectures of Microsoft's Analytics Platform System (APS) and SQL Server Parallel Data Warehouse (PDW). You've learned about the differences between SMP and MPP systems and why MPP systems are better suited for large analytical workloads. You have learned about different methods for loading PDW and ways to improve load performance. You have also discovered some best practices along the way. Lastly, you walked through a step-by-step exercise to parallelize loading data from SQL Server into PDW using SSIS.



XML Patterns

XML is a popular format for exchanging data between systems. SSIS provides an XML Source adapter, but because of the flexible nature of XML, it can sometimes be tricky to get your data to fit into the tabular format that the SSIS data flow expects. This chapter describes the formats that work best with the XML Source and two alternative patterns for reading XML data with SSIS.

Using the XML Source

Like most Data Flow components, the XML Source component requires column metadata to be set at design time. This is done using an XML schema file (.xsd). The XML Source component uses the XML structure defined in the schema to create one or more outputs, and it also uses the element and attribute data types to set the column metadata. Changing the schema file will refresh the component's metadata and may cause validation errors if you have already mapped some of its outputs.

If you don't already have an XML schema defined for your document, SSIS can generate one for you. Click the Generate Schema button on the XML Source editor UI, and the component will infer the schema from the current document. Note that although this schema is guaranteed to work with the current XML file, it might not work for others if optional elements or values are longer than expected. You may need to modify the generated schema file by hand to ensure that the `minOccurs` and `maxOccurs` attribute values are correct for each element and that the data types were set correctly.

The XML Source is easiest to use when your input file has a simple element/subelement structure. Listing 9-1 shows an example of that structure.

Listing 9-1. Simple XML Format Using Elements

```
<root>
  <node>
    <subnode>value</subnode>
    <anothersubnode>1</anothersubnode>
  </node>
  <node>
    <subnode>value</subnode>
    <anothersubnode>2</anothersubnode>
  </node>
</root>
```

Alternatively, the XML Source works well when values are listed as attributes, as shown in Listing 9-2. This format is similar to the output you would get from a `SELECT ... FROM XML RAW` statement in SQL Server.

Listing 9-2. Simple XML Format Using Attributes

```
<root>
  <row CustomerID="1" TerritoryID="1" AccountNumber="AW00000001" />
  <row CustomerID="2" TerritoryID="1" AccountNumber="AW00000002" />
</root>
```

Dealing with Multiple Outputs

The XML samples in Listings 9-1 and 9-2 will produce a single output in the XML Source. If your XML format has multiple levels of nested elements, the XML Source will start to produce more than one output. These outputs will be linked by automatically generated `_Id` columns, which you may need to join further downstream using a Merge Join transform.

■ Note This pattern works well if you have a single level of nested XML elements that you need to join. If you have multiple levels of XML elements and you need to join more than two of the XML Source outputs, you'll need to use the Sort transform.

Listing 9-3 contains an XML document with customer information. We'll use this document as our example for the remainder of the chapter.

Listing 9-3. Sample XML Document

```
<?xml version="1.0" encoding="utf-8"?>
<Extract Date="2011-07-04">
  <Customers>
    <Customer Key="11000">
      <Name>
        <FirstName>Jon</FirstName>
        <LastName>Yang</LastName>
      </Name>
      <BirthDate>1966-04-08</BirthDate>
      <Gender>M</Gender>
      <YearlyIncome>90000</YearlyIncome>
    </Customer>
    <Customer Key="11001">
      <Name>
        <FirstName>Eugene</FirstName>
        <LastName>Huang</LastName>
      </Name>
      <BirthDate>1965-05-14</BirthDate>
      <Gender>M</Gender>
      <YearlyIncome>60000</YearlyIncome>
    </Customer>
  </Customers>
</Extract>
```

The XML Source component will generate a separate output for each nested XML element. There will be three outputs for the XML document in Listing 9-3: Customers, Customer, and Name. Each output contains the elements and attributes that were defined in the schema, as well as an `<element_name>_Id` column, which acts as a primary key for the row. Outputs generated for child elements will contain an `_Id` column for their parent element's output, which allows the data to be joined later in your data flow if needed.

Figure 9-1 shows the outputs and column names generated by the XML Source component for the XML document in Listing 9-3.

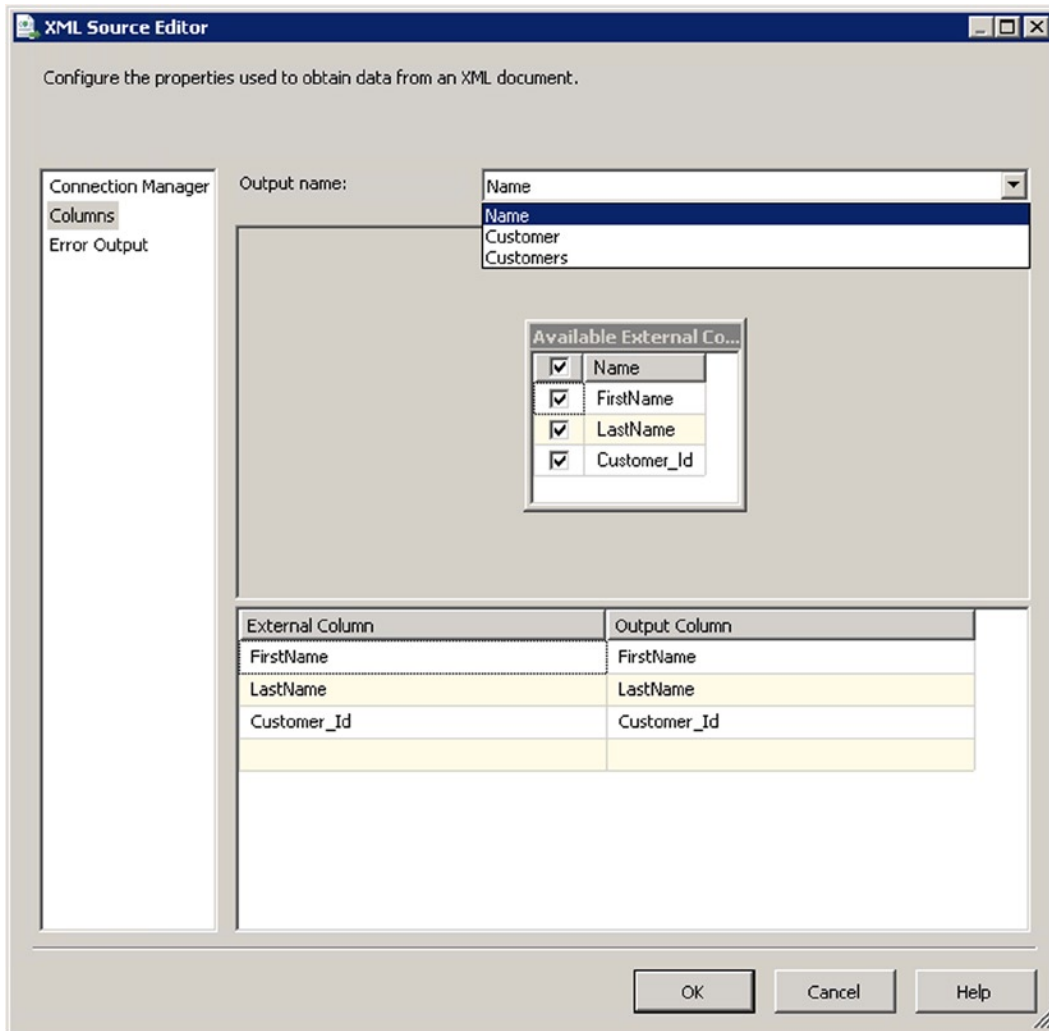


Figure 9-1. Outputs and columns generated for the Name element

Note The XML Source will not pick up any attribute values found on the root element of the document. To include this value in your output, you'll need to reformat the document to include a new root element node.

Figure 9-2 shows the schema of the destination table we will be storing the customer data in. As you can see, the table wants all of the columns shown in a single row, which means we'll have to merge the Customer and Name outputs before we can insert the data. The Merge Join transform is well suited for this, but it requires that both of its inputs are sorted the same way. We could add a Sort transform on each path before the Merge Join, but performing a sort can adversely affect performance and we should try to avoid doing so.

Customers			
	Column Name	Data Type	Allow Nulls
	[Key]	int	☐
	FirstName	nvarchar(50)	☒
	LastName	nvarchar(50)	☒
	BirthDate	date	☒
	Gender	nvarchar(1)	☒
	YearlyIncome	money	☒
			☐

Figure 9-2. Customers database table schema

Although the XML Source component doesn't set any sort information on the columns it produces, the output is already sorted on the generated `_Id` columns. To get the Merge Join to accept these inputs without using the Sort transform, we'll have to manually set the `IsSorted` and `SortKeyPosition` properties using the Advanced Editor for XML Source component, as follows:

1. Right-click the XML Source component and select Show Advanced Editor.
2. Select the Input and Output Properties tab.
3. Select the Name output and set the `IsSorted` property to True, as shown in Figure 9-3.

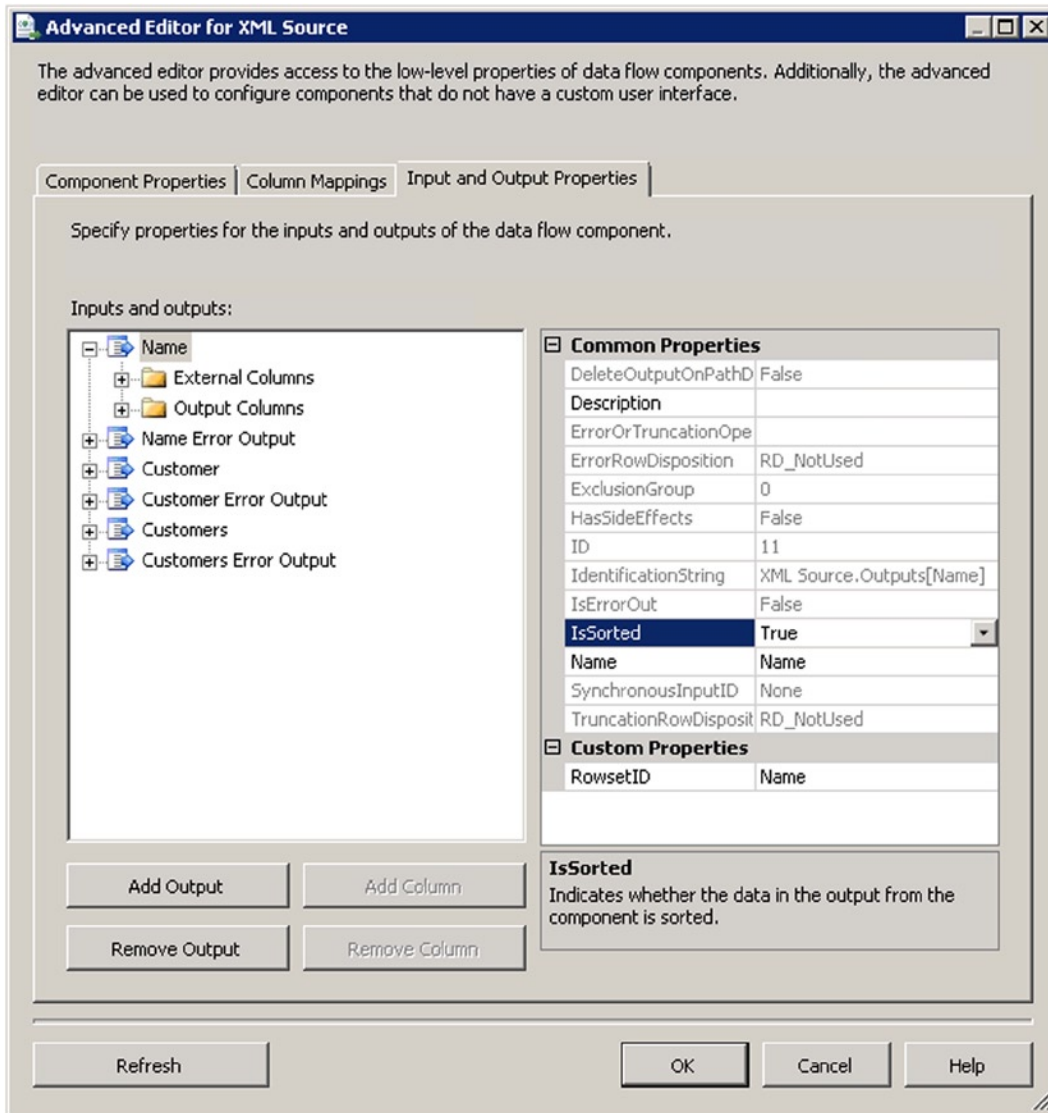


Figure 9-3. Setting the *IsSorted* property value in the Advanced Editor for XML Source

4. Expand the Name output and then expand the Output Columns folder.
5. Select the Customer_Id field and set the SortKeyPosition property to 1, as shown in Figure 9-4.

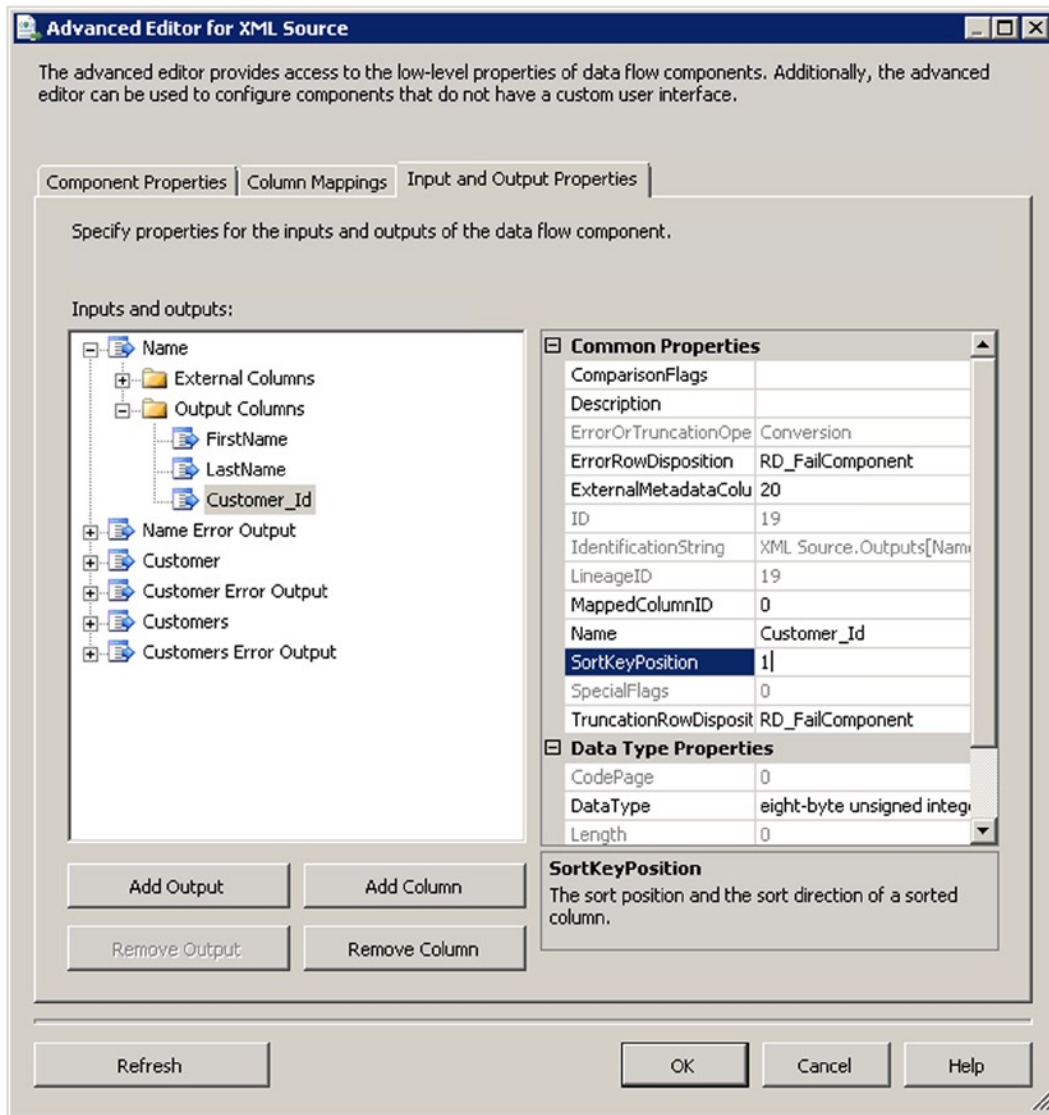


Figure 9-4. Setting the SortKeyPosition property value in the Advanced Editor for XML Source

6. Repeat steps 3–5 for the Customer output.
7. Click OK to save the changes and return to the designer.

By setting the `SortKeyPosition` value for the `Customer_Id` columns in the Name and Customer outputs, we've told SSIS that the rows will be sorted. We can now map both outputs directly to the Merge Join transform (as shown in Figure 9-5), and select the columns we want for our destination table (as shown in Figure 9-6).

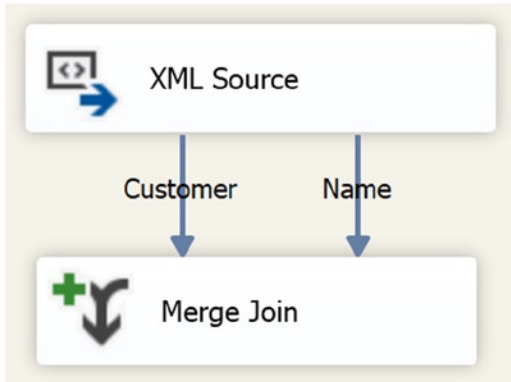


Figure 9-5. Connecting the XML Source component to a Merge Join transform

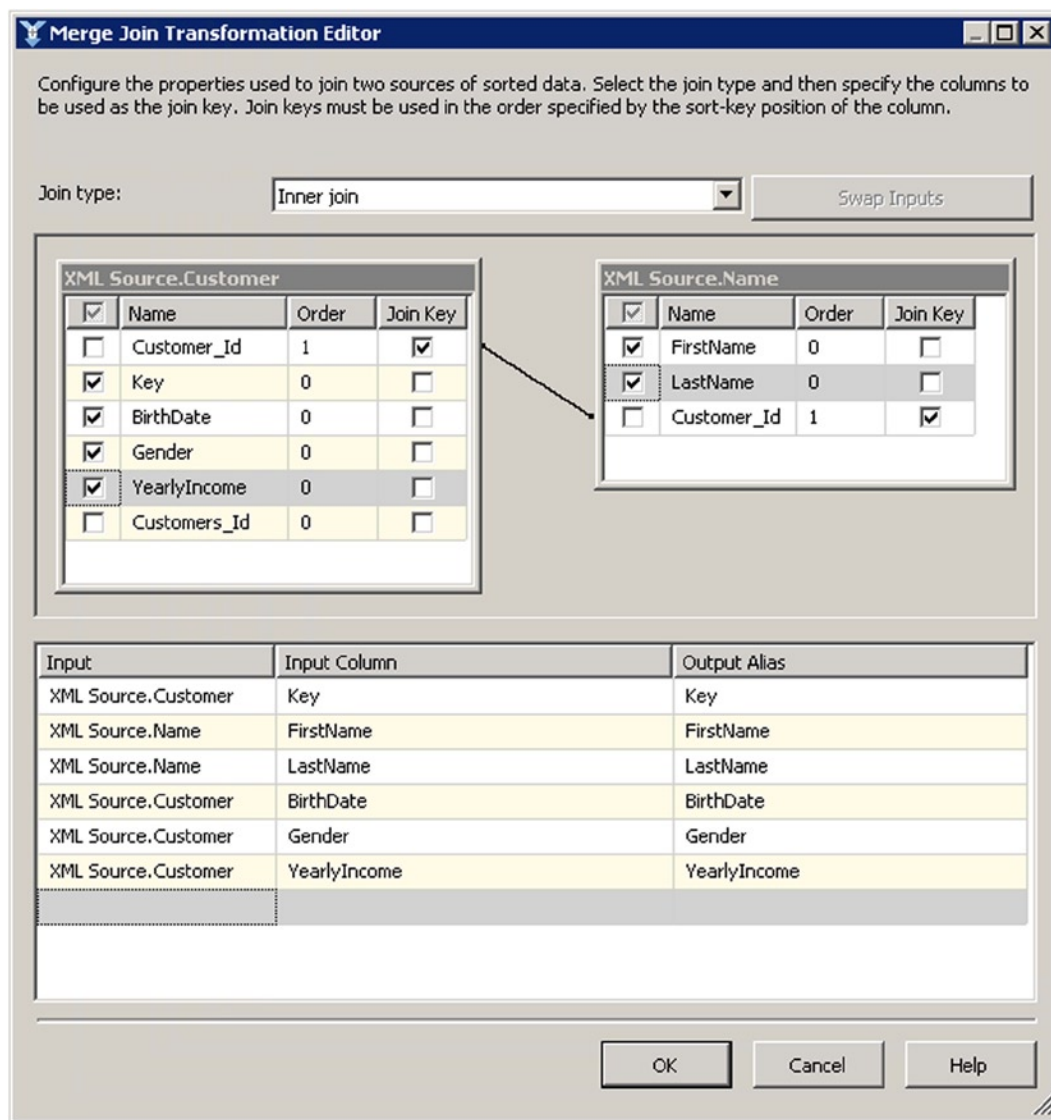


Figure 9-6. Mapping columns from the Merge Join transform

Making Things Easier with XSLT

You can simplify the handling of complex XML documents by preprocessing a source file with XSLT. Using XSLT, you can shape the XML to a format that closely resembles the destination schema, remove the fields that you don't need to capture, or transform it into a simple format that is easily handled by the XML Source component.

The sample XML from Listing 9-3 produced three separate outputs for the XML Source component. To insert the data into our destination, we had to merge the outputs into the format we wanted. Using the XSLT script in Listing 9-4, we can “flatten” or denormalize the data so that the XML Source component will have a single output.

Listing 9-4. XSLT Script to Simplify Our XML Sample

```

<?xml version="1.0" encoding="utf-8"?>
<xsl:stylesheet version="1.0" xmlns:xsl="http://www.w3.org/1999/XSL/Transform">
<xsl:output method="xml" indent="yes"/>
<xsl:template match="/Extract">
  <Customers>
    <xsl:for-each select="Customers/Customer">
      <Customer>
        <Key>
          <xsl:value-of select="@Key"/>
        </Key>
        <FirstName>
          <xsl:value-of select="Name/FirstName"/>
        </FirstName>
        <LastName>
          <xsl:value-of select="Name/LastName"/>
        </LastName>
        <BirthDate>
          <xsl:value-of select="BirthDate"/>
        </BirthDate>
        <Gender>
          <xsl:value-of select="Gender"/>
        </Gender>
        <YearlyIncome>
          <xsl:value-of select="YearlyIncome"/>
        </YearlyIncome>
      </Customer>
    </xsl:for-each>
  </Customers>
</xsl:template>
</xsl:stylesheet>

```

We can apply this XSLT using an XML task. Here is the process for doing that:

1. Save your XSLT script to a file.
2. Add an XML task to your package.
3. Double-click the task to open the editor.
4. Set the `OperationType` property to XSLT and the `SaveOperationResult` property to True.
5. Set the `SecondOperandType` and `SecondOperand` properties to point to your XSLT script file.
6. Enter the appropriate connection information for `SourceType`, `Source`, `DestinationType`, and `Destination`, similar to what is shown in Figure 9-7.

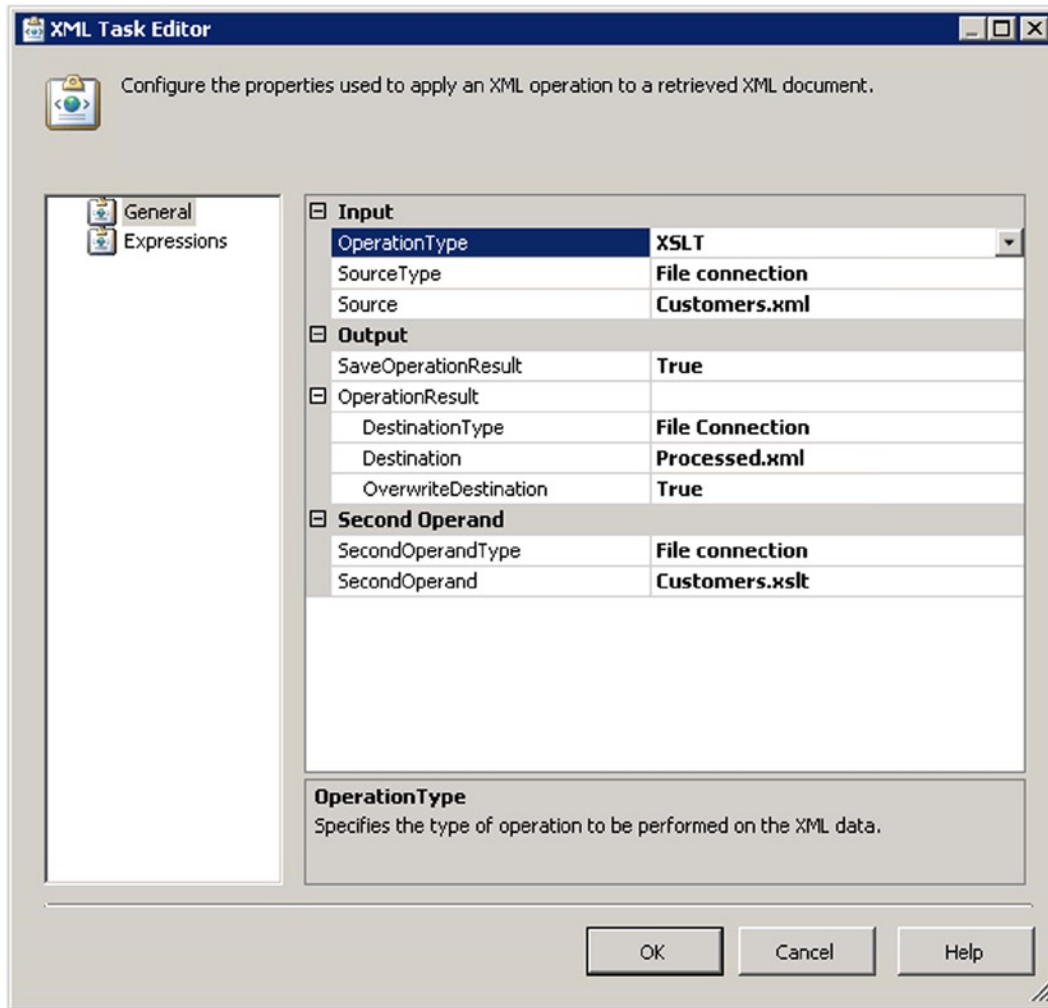


Figure 9-7. XML task configuration

Note The XML task was updated in the SQL Server 2012 release to use the latest .NET XML technologies. The performance of applying XSLT scripts is much better than in previous versions.

Listing 9-5 shows us what our sample XML document looks like after applying the XSLT script from Listing 9-4. We can see that all of the fields we need to extract are now under a single parent element. The schema for this new XML format gives us a single output with the XML Source component, removing the need to join outputs later on in the data flow.

Listing 9-5. Simplified XML Document

```

<?xml version="1.0" encoding="utf-8"?>
<Customers>
  <Customer>
    <Key>11000</Key>
    <FirstName>Jon</FirstName>
    <LastName>Yang</LastName>
    <BirthDate>1966-04-08</BirthDate>
    <Gender>M</Gender>
    <YearlyIncome>90000</YearlyIncome>
  </Customer>
  <Customer>
    <Key>11001</Key>
    <FirstName>Eugene</FirstName>
    <LastName>Huang</LastName>
    <BirthDate>1965-05-14</BirthDate>
    <Gender>M</Gender>
    <YearlyIncome>60000</YearlyIncome>
  </Customer>
</Customers>

```

Using a Script Component

An alternative to processing an XML document with the XML Source is to use a script component. This pattern requires some custom coding, but it gives you full control over the way the data is output. The .NET Framework provides a number of ways to parse and load an XML document, each with their own strengths and performance characteristics. This section describes two separate patterns for processing XML documents with a script component.

The first pattern uses the XML Schema Definition Tool (Xsd.exe) to generate a set of .NET classes that can be used by an SSIS script component. It uses the `XmlSerializer` class to convert the source XML document into a set of easy-to-use .NET objects. Although `XmlSerializer` is not the fastest way to process an XML document in .NET, the strongly typed classes allow for code that is straightforward and easy to maintain. This approach is a recommended alternative to the XML Source when you're working with complex XML documents that can easily fit into memory (for example, smaller than 100MB).

The second pattern uses a combination of LINQ to XML and the `XmlReader` class to process XML documents in a streaming manner. This approach is more sensitive to changes to the XML format; it may be harder to maintain, but it will output scripts that use the `XmlSerializer` class. This pattern is recommended when you are processing very large XML documents or when performance is critical.

Configuring the Script Component

The script components in both patterns are configured the same way, but they will contain different code. Both will use a file connection manager to locate the source XML file at runtime, and both will define the same set of output columns. Use the following steps to configure your script component:

1. Add a file connection manager to your package.
2. Set Usage Type to Existing File and set the path to our XML source file as shown in Figure 9-8.

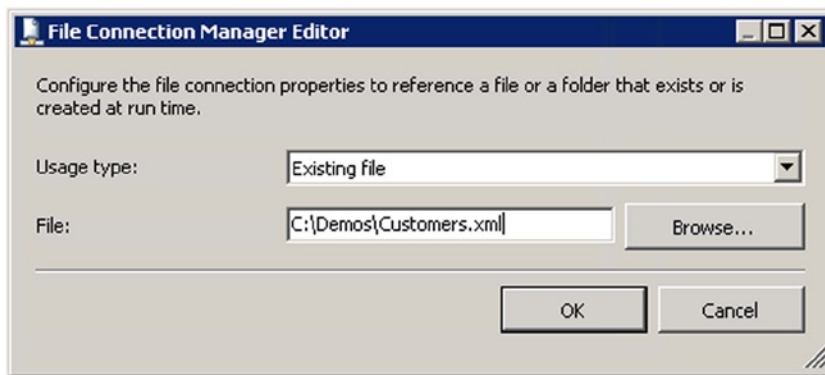


Figure 9-8. *Configure the File Connection Manager editor*

3. Add a data flow to your package, and drag a Script Component transform from the toolbox.
4. Select Source from the Select Script Component Type dialog as shown in Figure 9-9.

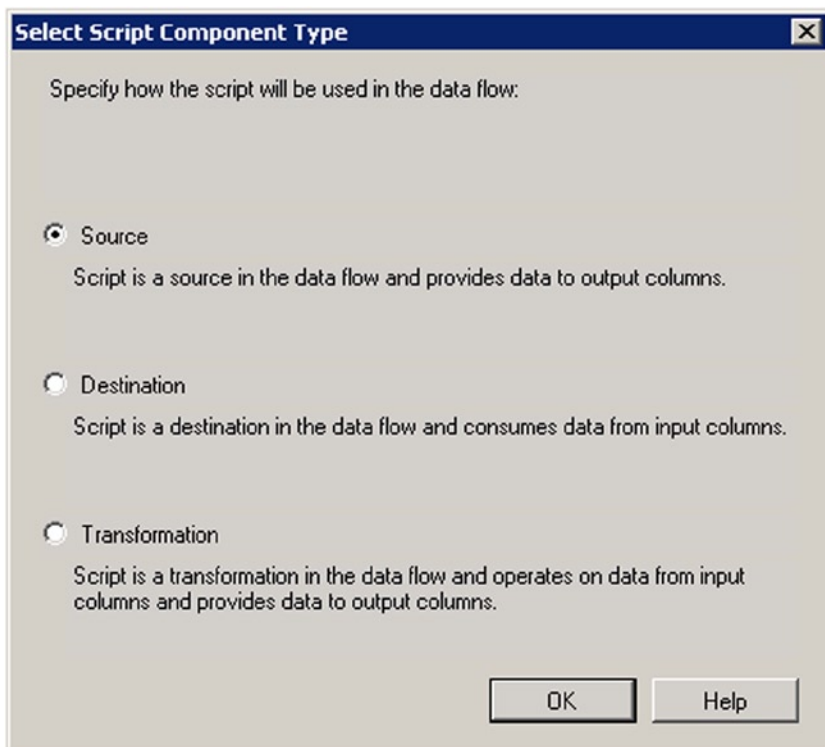


Figure 9-9. *Creating a new script component source*

5. Double-click the component to bring up the Script Transform editor.
6. Click the Inputs and Outputs page.
7. Rename your output from Output 0 to something more meaningful. Since our sample XML is outputting a set of Customers, that is the name we'll use for this example.
8. Define the columns as you'd like them to be output. Make sure the data types for the columns match what has been defined in your schema. Table 9-1 shows the column-to-data-type mappings you would use for the sample defined in Listing 9-3. Your column definition should look similar to Figure 9-10 when you are done.

Table 9-1. *Column Data Type Mapping for the XML Sample*

Column	Data Type
Key	DT_UI4
FirstName	DT_WSTR(255)
LastName	DT_WSTR(255)
BirthDate	DT_DBTIMESTAMP
Gender	DT_WSTR(255)
YearlyIncome	DT_UI4

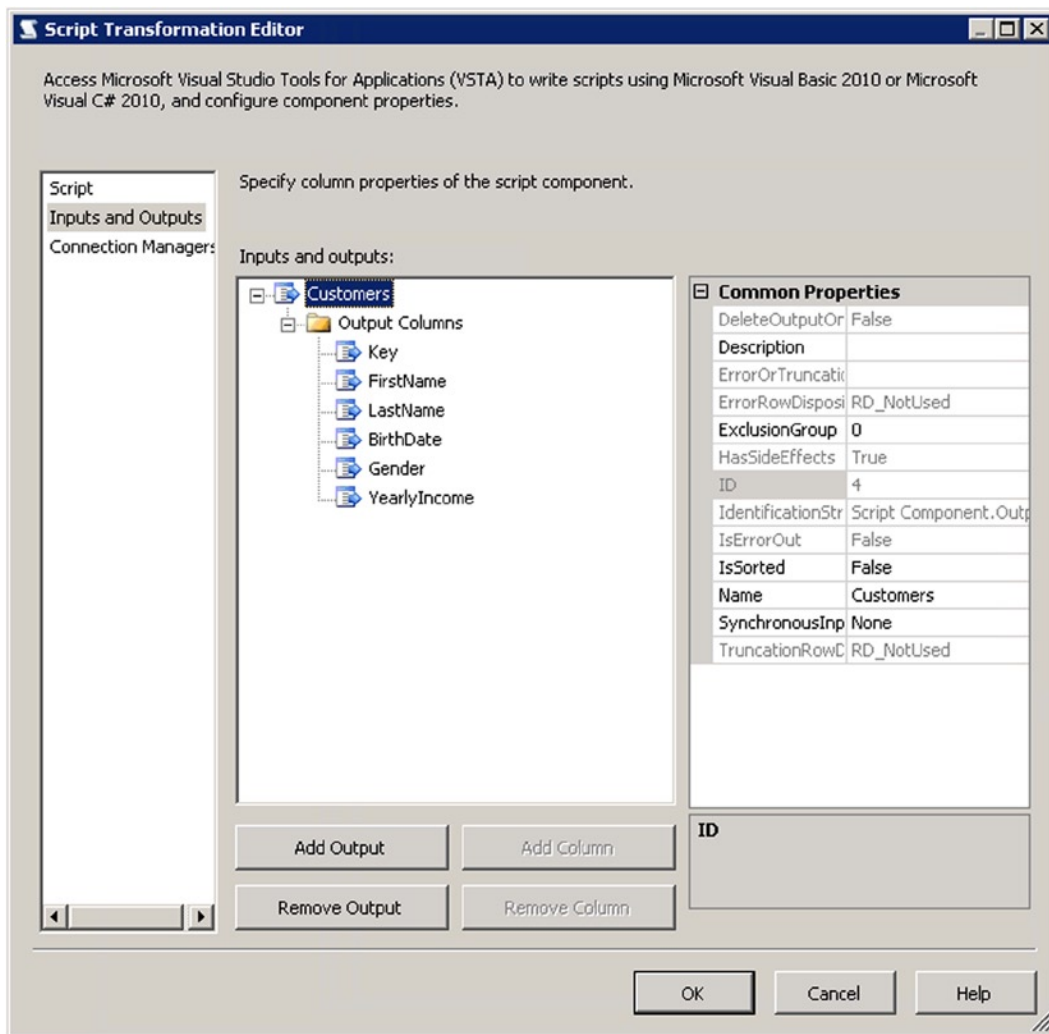


Figure 9-10. Configured output columns

9. Go to the Connection Managers page and add a reference to the file connection manager you created in step 1.
10. Give the connection manager a meaningful name, such as **CustomerFile**. The page will look similar to Figure 9-11.

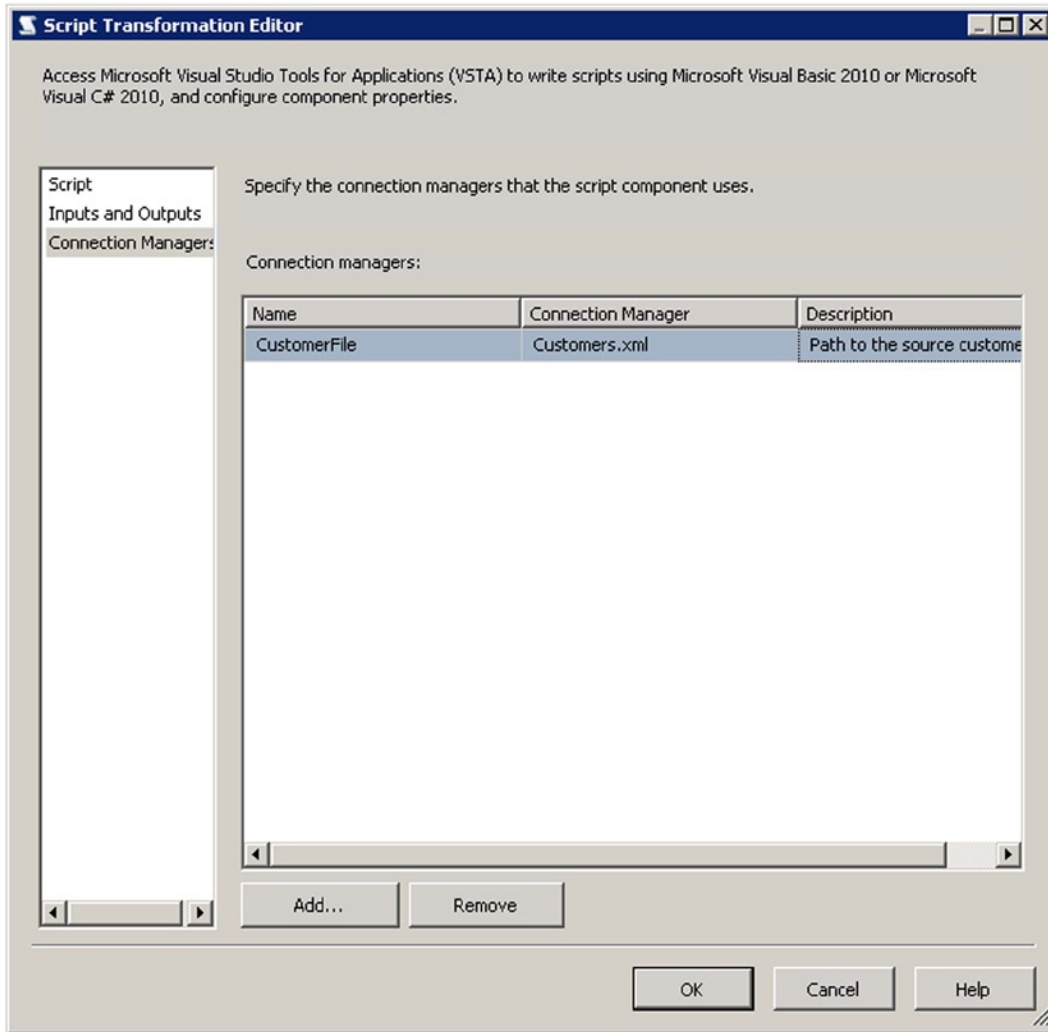


Figure 9-11. Configured connection manager

11. Go to the Script page and click the Edit Script button to launch the VSTA script editor.

We'll be adding code to the `PreExecute` and `CreateNewOutputRows` methods of the `ScriptMain` class, as well as overriding two additional methods from the base class: `AcquireConnection` and `ReleaseConnection`.

The `AcquireConnection` method will retrieve the path to our XML file from the connection manager we configured in step 10.

The `PreExecute` method will verify that the file actually exists, and it will raise an error if it is not found.

The `CreateNewOutputRows` method does the majority of the script's work. It is responsible for extracting data from our source document and outputting it to the data flow. The code that goes in here will depend on which pattern you select.

Finally, the `ReleaseConnection` method will release the file connection, indicating to the runtime that we are finished with it.

■ **Note** Although you don't need to call `ReleaseConnection` for a file connection manager, it's good to get into the habit of calling `ReleaseConnection` anytime you have a matching call to `AcquireConnection`. Certain connection managers, such as the OLE DB connection manager, will leave the underlying database connections open and keep resources in memory until the connection object has been released.

Listing 9-6 shows the code that we will be using for both script component patterns.

Listing 9-6. Full Source Code Listing

```
using System;
using System.Data;
using Microsoft.SqlServer.Dts.Pipeline Wrapper;
using Microsoft.SqlServer.Dts.Runtime Wrapper;
using System.IO;
using System.Xml.Serialization;
using System.Xml;

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

    public override void AcquireConnections(object Transaction)
    {
        // Call the base class
        base.AcquireConnections(Transaction);

        // The file connection manager's AcquireConnection() method returns us the path as a string.
        pathToXmlFile = (string)Connections.CustomerFile.AcquireConnection(Transaction);
    }

    public override void PreExecute()
    {
        // Call the base class
        base.PreExecute();

        // Make sure the file path exists
        if (!File.Exists(pathToXmlFile))
        {
            string errorMessage = string.Format("Source XML file does not exist. Path: {0}", pathToXmlFile);
            bool bCancel;
```

```

        ComponentMetaData.FireError(0, ComponentMetaData.Name, errorMessage, string.Empty, 0,
                                   out bCancel);
    }

    public override void CreateNewOutputRows()
    {
        // TODO - This is where we will load our XML document
    }

    public override void ReleaseConnections()
    {
        // Call the base class
        base.ReleaseConnections();

        // Release our connection
        Connections.CustomerFile.ReleaseConnection(pathToXmlFile);
    }
}

```

Once your script component is configured, you can plug in the `CreateNewOutputRows` logic from one of the following patterns.

Processing XML with `XmlSerializer`

To process the XML file using the `XmlSerializer` class, we'll use the XML Schema Definition tool to generate a set of .NET classes from our XML schema file. From the command line, we'll specify that we want to generate classes (/classes), the language we'd like to use (in this example, we'll use C#, but VB could be used as well), the namespace of the resulting class, and the path to our schema file. We'll use the schema file (`Customer.xsd`) for the customer data XML from Listing 9-3. The command line and `xsd.exe` output is shown in Listing 9-7.

Listing 9-7. XML Schema Definition Tool Command Line

```
C:\demos>xsd.exe /classes /language:CS /namespace:DesignPatterns.Samples Customer.xsd
```

```

Microsoft (R) Xml Schemas/DataTypes support utility
[Microsoft (R) .NET Framework, Version 2.0.50727.3038]
Copyright (C) Microsoft Corporation. All rights reserved.
Writing file 'Customer.cs'.

```

Note The XML Schema Definition tool is part of the Windows SDK. On most machines, it will be found in the `C:\Program Files (x86)\Microsoft SDKs\Windows\v7.0A\Bin` directory. For more information on the XML Schema Definition tool, see its MSDN entry at <http://msdn.microsoft.com/en-us/library/x6c1kb0s.aspx>.

The resulting `Customer.cs` file will have the classes we'll use in our script component. When used with the `XmlSerializer` class, we can read the entire XML Source file into an easy-to-manipulate set of objects.

Before we begin writing the `CreateNewOutputRows` logic, we'll need to include the `Customer.cs` file that we generated using `xsd.exe`. To do this, perform the following steps from within the VSTA script editor environment:

1. Right-click the project node in the Solution Explorer (this will start with "sc_", followed by a string of numbers) and choose **Add Existing Item**.
2. Browse to the `Customer.cs` file that you generated with `xsd.exe`.
3. Open `main.cs` from the Solution Explorer.
4. Include the namespace for the `Extract` class by adding `using DesignPatterns.Samples;` to the top of the file.

Once the file has been added to your project, you can write the code in `CreateNewOutputRows` that will read and manipulate the XML data. The source code for the `CreateNewOutputRows` function is in Listing 9-8.

Listing 9-8. Script Logic for Using the `XmlSerializer` Class

```
public override void CreateNewOutputRows()
{
    // Load our XML document
    Extract extract = (Extract) new XmlSerializer(typeof(Extract)).Deserialize(XmlReader.
        Create(pathToXmlFile));

    // Output a new row for each Customer in our file
    foreach (ExtractCustomer customer in extract.Customers)
    {
        CustomersBuffer.AddRow();

        CustomersBuffer.Key = customer.Key;
        CustomersBuffer.FirstName = customer.Name.FirstName;
        CustomersBuffer.LastName = customer.Name.LastName;
        CustomersBuffer.BirthDate = customer.BirthDate;
        CustomersBuffer.Gender = customer.Gender;
        CustomersBuffer.YearlyIncome = customer.YearlyIncome;
    }
}
```

Processing XML with `XmlReader` and LINQ to XML

This pattern makes use of the `XmlReader` class to stream in an XML document, as well as LINQ to XML functionality to extract the values you want to keep. It is ideal for processing large XML documents, since it does not require the entire document to be read into memory. It is also well suited for scenarios where you want to extract certain fields from the XML document and ignore the rest.

■ **Note** The idea for this pattern came from a post from SQL Server MVP, Simon Sabin. Sample code and other great SSIS content can be found on his blog at <http://sqlblogcasts.com/blogs/simons/>.

The key to this pattern is the use of the `XmlReader` class. Instead of using the `XDocument` class to read our source XML file (which is the typical approach when using LINQ to XML), we'll create a special function that returns the XML

as a collection of XElements. This allows us to make use of the LINQ syntax while taking advantage of the streaming functionality provided by XmlReader.

Before adding the code, you'll need to add the following namespaces to your using statements:

- System.Collections.Generic
- System.Linq
- System.Xml.Linq

Listing 9-9 contains the code for the XmlReader function (StreamReader), as well as the CreateNewOutputRows logic to consume the XML document.

Listing 9-9. Script Logic for Using the XmlReader Class

```
public override void CreateNewOutputRows()
{
    foreach (var xdata in (
        from customer in StreamReader(pathToXmlFile, "Customer")
        select new
        {
            Key = customer.Attribute("Key").Value,
            FirstName = customer.Element("Name").Element("FirstName").Value,
            LastName = customer.Element("Name").Element("LastName").Value,
            BirthDate = customer.Element("BirthDate").Value,
            Gender = customer.Element("Gender").Value,
            YearlyIncome = customer.Element("YearlyIncome").Value,
        })
    {
        try
        {
            CustomersBuffer.AddRow();
            CustomersBuffer.Key = Convert.ToInt32(xdata.Key);
            CustomersBuffer.FirstName = xdata.FirstName;
            CustomersBuffer.LastName = xdata.LastName;
            CustomersBuffer.BirthDate = Convert.ToDateTime(xdata.BirthDate);
            CustomersBuffer.Gender = xdata.Gender;
            CustomersBuffer.YearlyIncome = Convert.ToDecimal(xdata.YearlyIncome);
        }
        catch (Exception e)
        {
            string errorMessage = string.Format("Error retrieving data. Exception message: {0}",
                                                e.Message);

            bool bCancel;
            ComponentMetaData.FireError(0, ComponentMetaData.Name, errorMessage, string.Empty, 0,
                                        out bCancel);
        }
    }
}

static IEnumerable<XElement> StreamReader(String filename, string elementName)
{
    using (XmlReader xr = XmlReader.Create(filename))
```

```

{
    xr.MoveToContent();
    while (xr.Read())
    {
        while (xr.NodeType == XmlNodeType.Element && xr.Name == elementName)
        {
            XElement node = (XElement)XElement.ReadFrom(xr);
            yield return node;
        }
    }
    xr.Close();
}
}

```

Conclusion

The XML Source component lets you process XML documents from an SSIS data flow without writing any code. Although it can handle most XML schemas, it tends to work best with simple XML documents. When dealing with complex XML formats, consider using XSLT to reformat the source document to a format that is easily parsed. If you don't mind writing and maintaining .NET code, consider using one of the script component patterns described in this chapter when you need more control over how a document is parsed, are working with large XML documents, or have specific performance requirements.



Expression Language Patterns

The expression language in SSIS might appropriately be referred to as the glue that holds the product together. Expressions in SSIS provide a relatively simple and easy-to-use interface to allow data developers to introduce dynamic logic into the ETL infrastructure. When you think through the various moving parts within Integration Services, you'll likely discover that you can manipulate all of them in one way or another by using expressions.

Expressions provide a fast, effective, and, dare I say, fun way to solve specific ETL challenges. In this chapter, we'll look into some of the basics of the expression language and I'll describe a few instances where SSIS expressions are ideal (and a few where they might not be) for effectively solving difficult ETL problems.

Getting to Know the Expression Language

Before we dive into the design patterns surrounding the SSIS expression language, let's spend a little time defining and becoming familiar with the nuances of the language. Reviewing the language-specific patterns can help you get up to speed and use the language correctly.

What Is the Expression Language?

The *SSIS expression language* is an interpreted language that has been built into the SSIS runtime environment. This specialized language is used to craft scalar-valued snippets of code (individually referred to as *expressions*) that you may use at various points within the SSIS environment.

The SSIS designer exposes dozens of interfaces where expressions can be used in place of hard-coded values, allowing the BI professional to leverage that flexibility to create dynamic and reusable elements within SSIS. Conceptually, it's not unlike the product-specific dialects that exist in other Microsoft development environments. For example, when you are developing reports in SSDT or BIDS that you want to deploy to SQL Server Reporting Services, you can use Visual Basic for Applications (VBA) code to generate dynamic behavior during report execution and rendering.

As you explore the expression language, you'll find it to be a very powerful addition to the natural capabilities of SQL Server Integration Services. It has a rich library of functionality that will be familiar to both developers and DBAs. Among the functional domains of the SSIS expression language are

- A full complement of mathematical functions and operators
- An impressive set of string functions that may be used in comparisons, analysis, and value manipulation
- Common date and time functionality, including date part extraction, date arithmetic, and comparison

The expression language serves two different roles within the package life cycle:

Evaluation: You can use expressions to determine whether a specified condition is true and to change the behavior of the package accordingly. When it is used as part of the control flow, an expression used as an evaluation may check a certain value and dynamically alter the execution path based on the results of that comparison. Within the data flow, expressions allow you to evaluate data a row at a time to determine how to proceed in the ETL.

Assignment: In addition to using expressions as decision-making elements, you can use SSIS expressions to programmatically modify data during package execution. Typically you might use them as expression-based property settings and to transform in-process data within the data flow.

Expressions in SSIS may derive their comparisons or assignments from several fronts. Built-in system variables permit visibility into software environmental data such as package and container start times, machine environment information, package versioning metadata, and more. You can interrogate and manipulate the values of user-defined SSIS variables by using expressions and access values of package parameters to be leveraged elsewhere during package execution. In the data flow, expressions may interact with running data at the cellular level.

Expressions are value-driven at runtime. Unlike settings that are generally only configurable at design time (think data flow column definitions), expressions will calculate their values when the package is actually executed. Furthermore, a single expression may be evaluated many times (perhaps with a different result each time) during the execution life cycle of the package. Consider the case of the ForEach Loop, a container that loops through a specified set of objects or values until it reaches the end of said collection. Expressions that are manipulated within the loop may be updated dozens or even hundreds of times during this process.

Why Use Expressions?

The ability to use expressions is one of the greatest strengths of SQL Server Integration Services. Simply put, expressions help to fill in the small gaps. The expression language isn't a tool in itself, but rather, it is an interface that helps other SSIS tools more effectively perform their respective functions. That's all well and good, but in the interests of simplicity, why would an ETL developer choose to use expressions instead of other languages such as T-SQL, C#, or VB.NET? Here are a few compelling reasons to employ the expression language in your SSIS packages:

Simplicity: Expressions language can be used to quickly add flow logic or make small changes to in-pipeline data in the data flow. You can often handle small ETL changes that might otherwise be relegated to a Script task or component inline without needing to introduce extra code to the package.

Consistency: Use of the expression language can lead to a consistent approach to data or program flow challenges. For example, if your ETL requires that you convert blank strings to NULLs, the approach and syntax would otherwise be different for flat files, Access databases, and relational database sources. By applying the expression language to the same task, you reduce the amount of distinct code you have to write by relying on the built-in string manipulation functions in the expression language.

Maintenance scope: By applying the design pattern of using the expression language for cleansing needs, you can eliminate much of the sleuth work you need to do to track down and change cleansing rules as your business expectations change. Using expressions in the package itself provides you with a single point of maintenance rather than forcing you to inspect the upstream data sources each time you need to make a change.

I've done a number of presentations for novice SSIS developers, and when I bring up the topic of the expression language, one question almost always seems to come up: "Where do I use this expression language stuff?" My answer: "Everywhere!" Part of the beauty of expressions is that they can be used almost anywhere within SSIS packages. You can employ expressions on the control flow in precedence constraints. It's convenient to make your SSIS package variables dynamic by replacing their static values with expressions. You can leverage expressions within the data flow to manipulate data and even control the execution path. The bottom line is that you can manipulate many of the common properties of packages, tasks, constraints, and data flow elements by using expressions.

Although its syntax may seem unusual, the expression language isn't difficult to learn. Anyone with logical scripting experience (even if that experience is limited to T-SQL) can quickly pick up on the basics and should be able to master the language with a reasonable amount of practice.

Language Essentials

Even for those who have experience scripting in other Microsoft development environments, the first exposure to the SSIS expression language can be a little unsettling. The syntax and functionality are unlike any other language, either interpreted or compiled. It appears to be a strange hybrid of several languages and is certainly a dialect all of its own.

Developers who have spent time using the C-style languages (C, C++, C#, Java) will recognize some of the syntactical nuances within the expression language:

- Case-sensitive column and variable names
- Case-sensitive string comparisons
- Double-equal (==) comparison operators
- Simplified conditional (if/then/else) operators

Similarly, anyone experienced in T-SQL will find a great deal of familiar behavior within the SSIS expression language:

- Case-insensitive function names
- Date arithmetic and string manipulation functions much like those in T-SQL

The SSIS expression language is quite powerful, with its wide variety of functions and operators. With native behavior including equality tests, type casts, string manipulation, and date arithmetic, the use of expressions within SSIS packages can help to overcome ETL challenges both large and small.

Limitations

As useful as the expression language is, there are a few key limitations to its use. Bear in mind that these are relatively minor hang-ups; the SSIS expression language is not intended to be a full-featured programming language, but rather a lightweight tool to supplement the behavior of existing SSIS tasks and components. Among some of the challenges are the following:

Expressions are limited to single-value statements: This almost goes without saying since it's an expression language and not a programming language. Still, it's worth mentioning that you can't, for example, use a single expression to iterate through a list or process a string character-by-character.

No IntelliSense: Unlike other scripting/expression environments, there is no built-in IntelliSense within the native expression editors. Although the expression editor in SSIS does have field, variable, and function lists, the convenience and coding reliability of IntelliSense has not yet made it into the product.

No error handling: This limitation is most visible when you attempt to change data type or length. Because there is no `try/catch` or `TryParse()` behavior found in the .NET-based languages, you cannot, for example, attempt to cast a text value to a number and programmatically handle any type cast errors in the same expression.

No comments allowed: The fact that there is no provision for code comments can be a significant downside when you are using lengthy or complex expressions. Any comments documenting the purpose of the expression have to be done peripherally—for example, on the data flow or control flow surface as an SSIS annotation.

- **Complex statements can be difficult:** Simple assignments or comparisons are easy to do and are usually easy to understand after the fact. However, introducing even a moderate amount of complexity to an expression can make for a lengthy and convoluted statement. Consider the case of a multiconditional `If` statement. In most other dialects, one could simply perform an `If/Then/Else If` operation to account for more than one test condition. However, the expression language doesn't have such behavior, so to build such logic you need to nest conditional operators. Listing 10-1 shows how you might easily address four possible conditions in a CASE operation in Transact-SQL. By contrast, Listing 10-2 shows a similar example using the expression language (note that I manually wrapped the text to fit it on the page). Although the result of the operation is the same, the latter has conditional operators nested two levels deep and is more difficult to develop and maintain.

Listing 10-1. Multiconditional Evaluation in T-SQL

```
SELECT CASE WHEN @TestCase = 3 THEN 'Test case = Solid'
          WHEN @TestCase = 2 THEN 'Test case = Liquid'
          WHEN @TestCase = 1 THEN 'Test case = Gas'
          ELSE 'Unknown test case' END [TestCaseType]
```

Listing 10-2. Multiconditional Evaluation in the Expression Language

```
(TestCase == 1) ? "Test case = Gas" : (TestCase == 2 ? "Test case = Liquid" : (TestCase == 3 ?
"Test case = Solid" : "Unknown Test Case"))
```

Despite its minor shortcomings, the SSIS expression language remains an integral part of the product, and as you'll see later in this chapter, it has some very practical uses in a well-designed ETL ecosystem.

Putting the Expression Language to Work

Now that you understand what the expression language is (and is not), let's talk about some design patterns where you might use it.

Package Expressions

Although not as common as other uses, it is possible to use SSIS expressions to configure package-level properties. Here are a handful of properties that may be set at the package level by using expressions:

- `Disable`
- `DisableEventHandlers`
- `CheckpointFileName`

- MaxConcurrentExecutables
- DelayValidation
- Description

Consider the example of `MaxConcurrentExecutables`, which defines how many executables (packages, tasks, etc.) can run concurrently. By setting this property through an expression, the ETL developer would be able to dynamically control this value based on any criteria visible through an expression.

Although these properties are configurable by using expressions, it's far more common to find package-level options set by using package parameters (with later versions of SSIS) or package configurations (SQL Server 2008 and earlier). It is usually best to share common values across package ancestries using parameters or configurations, which allow you greater flexibility and easier maintenance. I expose this particular design pattern more for the purpose of identifying it as an antipattern than for defining parameters for its use. Unless there's some business case or regulation dictating otherwise, it's a better long-term solution to externalize these values rather than rely on expressions.

Variable Expressions

As shown in Figure 10-1, you can configure each variable with a static value in the Value field or define a value expression that will be evaluated at runtime. Note that the variable window was improved starting in SQL Server 2012—in older versions, static values were shown in the Variable window, but you had to use the Properties window to view or alter an expression for a variable.

Name	Scope	Data type	Value	Expression	
 vOutputDirectory	Merge Vendor...	String	d:\data\	"d:\\data\\" + @[User::vClientName]	...
 vNumberOfFiles	Merge Vendor...	Int32	0		...
 vClientName	Merge Vendor...	String		@{\$Package::pClientName}	...

Figure 10-1. Expressions with variables

In practice, I often see expressions applied to variable values, and then the resulting variable is used as a property on a task or component (as opposed to it being used as an expression to set the property directly). I'm a fan of this design pattern for one simple reason: reusability. It's not uncommon for components to share certain properties, and building expressions on each of those shared properties for every applicable component is both redundant and unnecessary. For those properties that will be shared across multiple tasks or components, it's far easier to centralize the expression logic into a variable and then use that variable to set the shared properties. This approach allows for faster development as well as easier maintenance should the logic require changes down the road.

When using this design pattern, don't forget that you can also "stack" variable values. In the expression statement, you can leverage other variables to set the value of the current variable.

Connection Managers

One of the most practical and common places to use SSIS expressions is the Connection Managers tray. Generally speaking, it's typically preferable to store dynamic connection properties not in expressions, but rather as parameters, particularly when you're dealing with structured data. Because of the sensitive and frequently changing nature of connection metadata (server names, user names, and passwords), most ETL professionals choose to externalize those settings to keep them stored securely and externally to the package so they can be globally changed (rather than modified package by package).

One recurring exception to this pattern is connections that interact with the file system. There are several cases where using expressions helps to lighten the load of processing file-based sources or destinations:

- When working with flat files, the files are to be named according to the current date and/or time stamp (such as Medicare_2014_06_01.txt, for example).
- The files are expected to be filed in the file system according to the date (such as D:\Data\2014\06\01\Medicare.txt).
- A scheduled job loads a text file that always has the same file name, but a copy of each day's file needs to be saved without overwriting the file processed on the previous day.

For these cases, you can use a little dab of expression language to dynamically build directory paths and file names that your connections in SSIS will use. For this example, let's assume that you're generating a flat text file from within your package, and you want to use a dynamic file name based on the current date. By setting the `ConnectionString` property from within the Properties window of the instance of the Flat File Connection Manager, you can manipulate the runtime value of the file name. As shown in Figure 10-2, you're specifying the base file name and then appending the elements of the current date to build a customized file name.

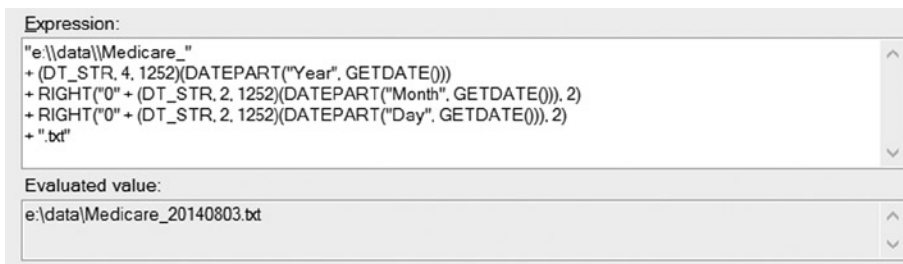


Figure 10-2. Dynamic file name using expression

Note that the pattern just discussed could be further extended to include elements of time (hours/minutes/seconds) should your ETL requirements include a restraint for that level of granularity.

Since we're not going in-depth into all the syntactical elements of the expression language, I'll just point out a couple of things I've done here:

- Because the backslash (\) is a special character in the expression language, I have to "escape" it (to negate its status as a special character) by using a double backslash (\\) when I include it as a string literal.
- Using `(DT_STR, n, 1252)`, I'm converting the integer value returned by the `DATEPART` function to ASCII text. In this case, I'm using code page 1252 with a maximum length of either 2 or 4 depending on the component of the date element.
- Using the `RIGHT` function, I'll pad any single-digit month or day value with zeroes (e.g., so that "3" becomes "03") to maintain consistency.

Remember that this pattern is highly flexible. It can be utilized with almost any file connection, whether it's to be used as a source or a destination. You're not limited to just flat file connections here either; you can extend this logic to some of the other SSIS connections as well. I've used this same design pattern when dealing with FTP data as both a source and destination. By embedding the same logic within the properties of an FTP source, you can programmatically "walk" the directory structure of a remote server when it is in a known and predictable format such as this.

Project-Level Connection Managers

When working with projects in the catalog deployment model, you can expose connection information across multiple packages in the same project by way of project connection managers. When you are using older versions of SSIS (2008 or earlier), or when you are working in the package deployment model in later versions of SSIS, any connection manager defined within a package is independent of those in other packages. Starting with SSIS in SQL Server 2012, however, you now have the ability to attach a connection manager to your workspace at the project level. These are accessible to all packages within the same project.

We'll not go deeply into the new deployment model in this chapter, but it is important to point out how the use of expressions impacts project connection managers. Because they are attached to the project and not one particular package, the properties of these shared connections are common to all packages in the project. As such, any property setting on these project connection managers—including the use of expressions—would be immediately reflected in all packages in the project. This is a welcome and much needed improvement to the way packages interact with one another, but for those of us who have worked with previous versions of SSIS, it's a bit of a paradigm shift. Don't get caught off guard when an expression applied to a project connection in one package gets applied to the other packages in the project!

Control Flow

Within the control flow, there are a couple of different ways to implement SSIS expressions. First, each of the tasks and containers will expose several properties that are configurable using expressions. In addition, the paths between them (known as *precedence constraints*) allow ETL developers to customize the decision path when moving from one task/container to another.

Conditional Execution Through Expressions and Constraints

The essential function of the control flow is to manage the execution of package elements. By using precedence constraints, you can design a package so that tasks and containers are fired in the proper order and with the correct dependencies intact. For a simple example of this, think about a package that truncates and then loads a staging table. You can perform both of these tasks in the same package, but without a precedence constraint to cause the insert operation to occur after the TRUNCATE TABLE execution, you run the risk of inadvertently *loading and then deleting* the same data.

You can configure precedence constraints to manage flow-based successful completion of the preceding task (the default behavior), or you may set them to cause the task to execute only if the preceding task fails. In addition, you can set the constraint to *Completion*, allowing the downstream task to fire when the upstream task is finished, regardless of its outcome. Tasks may have multiple precedence constraints, and you may set these so that any or all of them must be satisfied before the task to which they are attached will execute. Figure 10-3 shows a fairly typical use of precedence constraints; note that the unlabeled arrows represent Success constraints, and the others are labeled as to their purpose. The dashed lines indicate that the task is configured to execute upon completion of *either* of the preceding tasks.

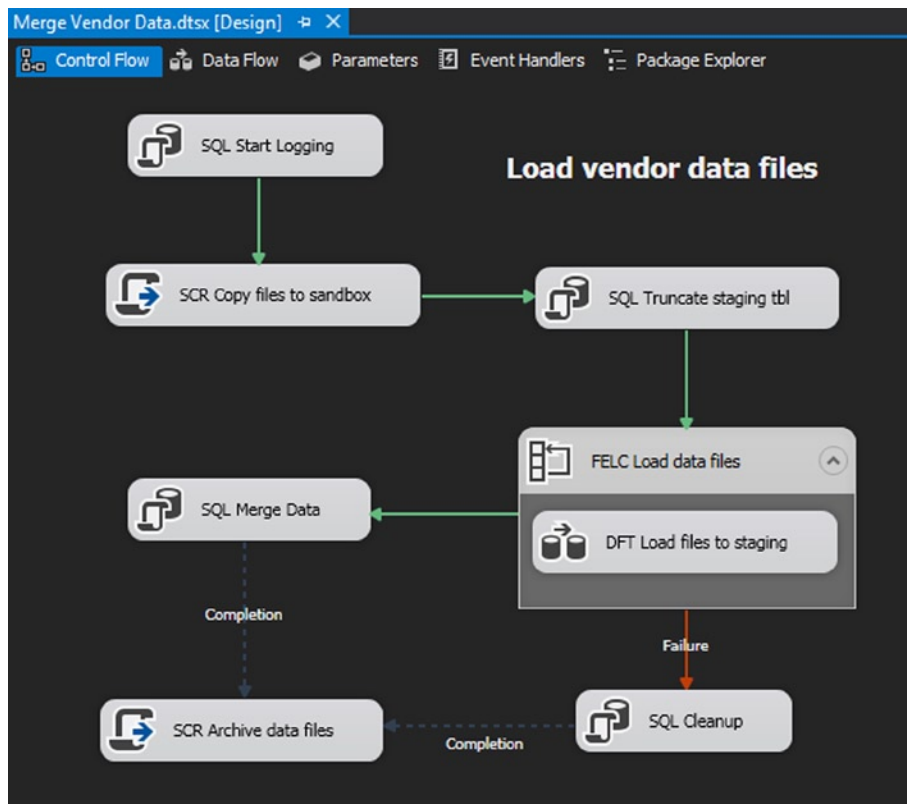


Figure 10-3. Precedence constraints

As useful as precedence constraints are, the domain of variability that they address is fairly limited: the only conditions that can be tested are whether a task completed as well as the success or failure of said task. In the brief example shown in Figure 10-3, you can probably infer that I'm downloading one or more files from an external source, loading the data from those files into staging tables, and then merging (upserting) the data into a database table. Although there's nothing technically wrong here, there is room for improvement. For example, what happens if there are no files to be processed? In the example shown, the truncation of the staging table, the loop through the file system to find the downloaded files (even if none exist), and the merge operation will all be executed even if there are no files to process.

In the first job I ever held, I was responsible for, among other things, gathering stray shopping carts from the store parking lot and bringing them back inside. My boss once told me, "This job requires an excessive amount of walking, so do what you can to save steps." All these years later, that advice still holds true. Why run through extra steps when you can simply skip past them if they are not needed? For the previous example, you can include a relatively simple expression to bypass the execution of the majority of the package when no files are found to process. Saving those steps saves CPU cycles, disk I/O, and other resources.

Precedence constraints also have the ability to use expressions to enforce proper package flow. In Figure 10-4, you'll see that the evaluation operation is set to Expression to enforce both the execution value of the prior task as well as the value defined in the Expression box. For illustration purposes, assume that you've populated an SSIS variable to store the number of files downloaded in the Script task operation, and you're using the expression to confirm that at least one file was processed. From here, you can either type the expression into the window manually or use the ellipsis button to open the Expression Editor (note that in earlier versions (2008 and earlier) of the product, you will have to enter the expression by hand without the benefit of the Expression Editor).

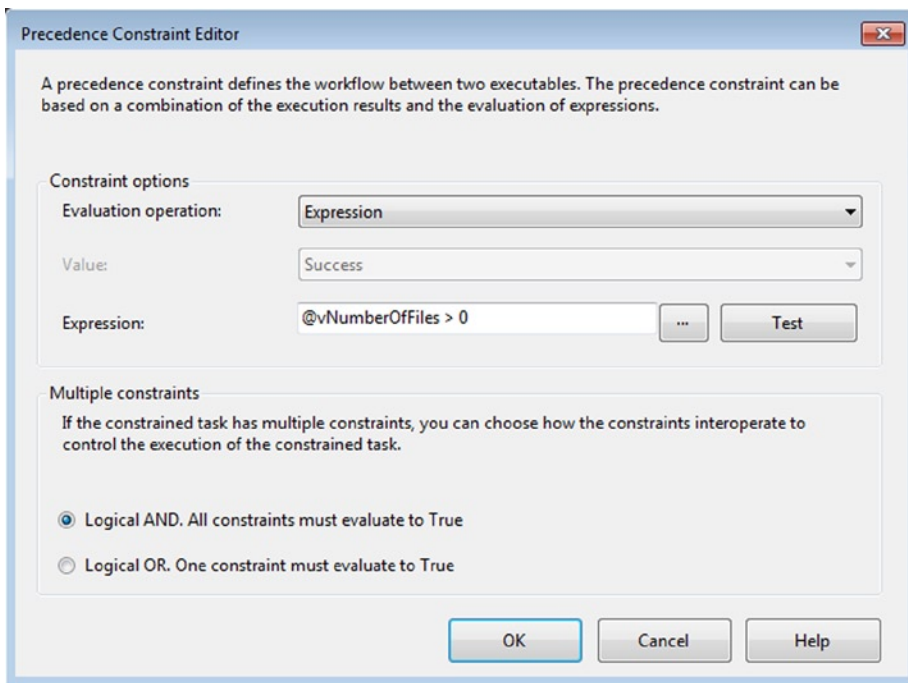


Figure 10-4. *Precedence Constraint Editor*

Refer back to the original package; you'll see that the precedence constraint between the first Script task and the truncation SQL task now reflects the presence of an expression in the constraint (Figure 10-5).

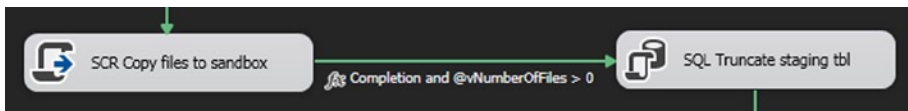


Figure 10-5. *Expression notation in precedence constraint*

It's worth noting that the example in Figure 10-5 shows a non-standard notation on the constraint. By default, only the function icon (fx) will appear when you are using an expression as part of a constraint. Assuming that the expression is not a lengthy one, I typically change the ShowAnnotation option of the constraint to ConstraintOptions, which will include the expression itself on the label of the constraint. This is an easy reminder of the expression used in the constraint, and it doesn't require opening the properties window to see the expression.

Task-Level Expressions

In addition to the control flow uses of expressions, most every task and container in SSIS has its own properties that can be configured using expressions. The options for configuration using expressions will vary from one executable to the next, but there are elements common to most tasks and containers:

- Description
- Disable
- DisableValidation
- TransactionOption
- FailPackageOnFailure
- FailParentOnFailure

A common design pattern using a task-level expression is to employ the `SqlStatementSource` property of the Execute SQL task. In most cases, you can use this task combined with query parameters to create dynamic statements in T-SQL. However, some language constructs (such as subqueries) don't always work well with parameters, exposing a need to build the SQL string in code. By using an expression instead of static text for the `SqlStatementSource` property, the ETL developer can have complete control of the T-SQL statement when query parameters don't fit.

■ **Note** There was a limit on string size in SQL Server 2008 Release 2 that's been greatly relaxed in 2012.

Data Flow Expressions

As we move from the control flow into the data flow, we find the more traditional use of expressions as part of our ETL strategy. Like the higher-level executables, we find that every component in the data flow is affected either directly or indirectly by SSIS expressions.

Data Cleansing

Lightweight data cleansing is one of the most common uses of the expression language within the data flow of SSIS. Most frequently used within the derived column transformation, expressions can be used for certain cleanup tasks, including these:

- Changing the case of data
- Grabbing a substring from within a longer string
- Trimming extraneous space characters
- Replacing inappropriate characters (such as removing letters from text)
- Changing data length or type

Often, you can minimize the need for data cleansing in the data flow simply through well-designed query statements in the extraction from the various data sources. However, sometimes cleanup at the source is just not an option. Many sources of data are nonrelational: consider text files and web services as data sources, for example, which generally do not have the option of cleaning up the data before its arrival into the SSIS space. Sometimes even relational sources fit in this box: I've encountered a number of scenarios where the only interface to the data was through a

predefined stored procedure that could neither be inspected nor changed by the ETL developer. For cases such as these where source cleansing is not possible, using expressions within the data flow is a good second-level defense.

One design pattern that I use frequently is to trim out extra whitespace and convert blank strings to NULL values. As shown in the following, such an operation could be performed with a single, relatively simple expression:

```
(LEN(TRIM([Street_Address])) > 0) ? TRIM([Street_Address]) : (DT_WSTR, 100) NULL(DT_WSTR, 100)
```

Regarding data cleansing using the expression language, I will offer a brief word of caution: if you find yourself needing to do complex, multistep cleansing operations within your SSIS packages, consider using some other means to do the heavy lifting. As I mentioned earlier, the expression language is best suited for lightweight data cleansing; because complex expressions can be difficult to develop and debug, you might instead use a richer tool such as the Script task or Script component, or perhaps Data Quality Services, for these advanced transformations.

Branching

Sometimes you will find that you need to create forks in the road with ETL data flow. There are several reasons why you might need to create such branches within your data flow:

Different outputs: For data that exists in a single data flow but is bound for different destinations, creating branches is an effective solution. I recall a case I worked on several years back when we were building a system to distribute data to several financial vendors. Each of these vendors required the same type of data, but each vendor would get a different “slice” of the data based on several criteria, and each required the data in a slightly different format. Rather than design multiple end-to-end data flows that would essentially duplicate much of the logic, I created a single package that employed a conditional split transformation to split the data flow based on a specified condition, and from there, the data branched out to the respective outputs.

Inline cleansing: A very common ETL scenario in SSIS is to split “good” data from “bad” data within a single data flow, attempt to clean the bad data, and then merge the cleansed data with the good data. This allows you to leave intact any data that does not require cleansing, which may help to conserve processing resources.

Disparate data domains: In cases where data is structurally similar but syntactically different, you might want to employ branching to handle the data differently within your data flow. Consider the example of geographical address data: although they both describe a physical address, you might need to process domestic addresses differently than you would handle international addresses. By using branching tools such as the conditional split, various address types from a single source type can be handled within one data flow task.

Varying metadata: Although relatively rare, there will be the occasion where a source may contain rows with varying metadata. Consider a text file with a ragged structure in which some rows are missing columns at the end of the line. By splitting the data based on the absence of certain columns, you can account for the metadata differences inline.

Figure 10-6 exposes this design pattern by showing the use of expression logic to break apart a data stream into multiple outputs. In this case, you are processing a billing file by using comparison expressions within the conditional split transformation (see the callout) to determine whether each row is paid on time, not yet due, or past due, and then you’re sending it to the appropriate output accordingly.

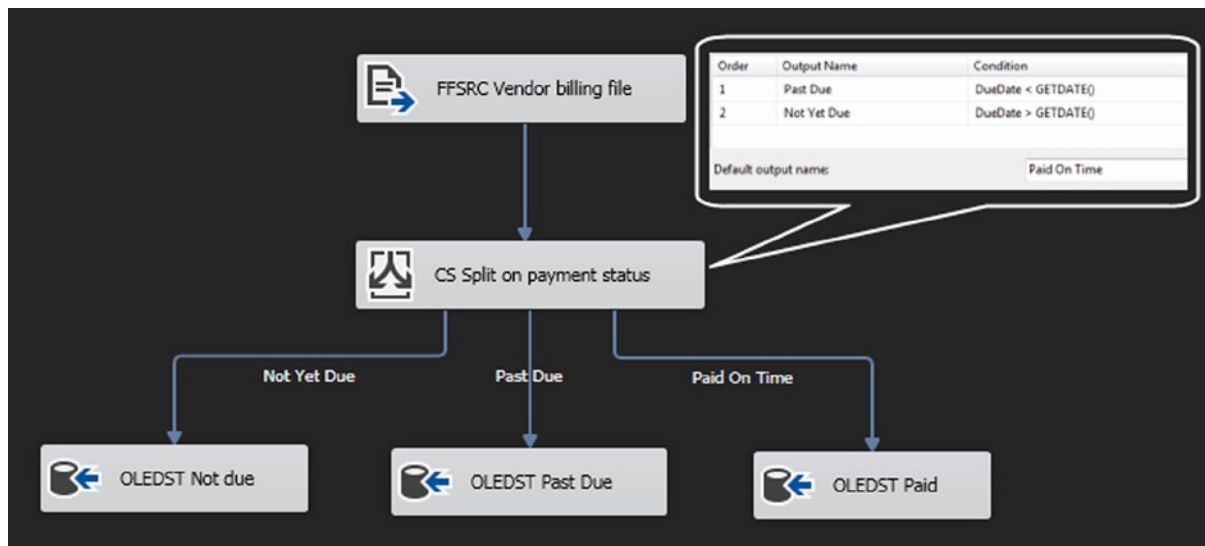


Figure 10-6. Using expressions to define multiple paths

One interesting caveat regarding the application of expressions within the data flow is the way in which SSIS exposes component-level expressions. Although the expression language is very useful within the pipeline of the data flow, most components do not actually expose properties that can be set using expressions. For those that do allow expressions on certain properties, these expressions are surfaced as elements of the data flow itself and will appear as part of the options in the Data Flow Properties window while working in the control flow.

As shown in Figure 10-7, you are using the expression properties of the data flow to access the ADO.NET data source within that data flow. As you can see, the identifier in the Property column shows that this expression belongs to the data source within the data flow, allowing you to set the `SqlCommand` property of that source. It's useful to note here that I used the ADO.NET source purposefully in this example. Since this source does not currently allow the use of parameters, setting the `SqlCommand` property is often an acceptable substitute for dynamically retrieving data from a relational database using this component.

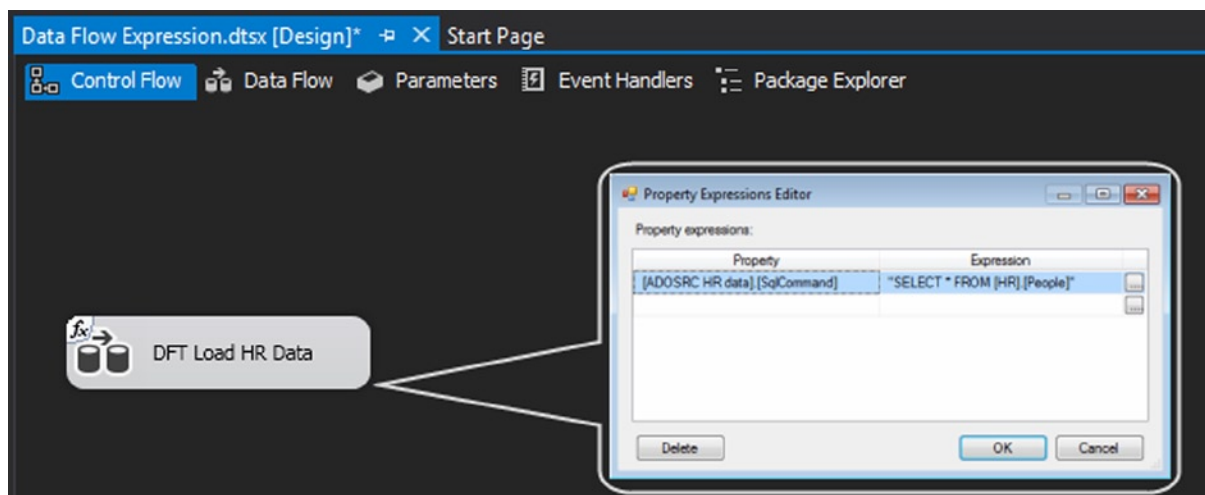


Figure 10-7. Data flow expression

Application of Business Rules

Although they share some of the same methods, the applying business rules differs conceptually from data cleansing. For the most part, data cleansing is considered to be universal: misspelled words, inconsistent casing, extraneous spacing, and the NULL-versus-blank-versus-zero quandary are all common problems that must be dealt with in almost every ETL process. Business rules, on the other hand, are specific use cases in which data is manipulated, extrapolated, or discarded based on custom logic that is specific to the business at hand. These rules may be general enough to apply to an entire industry (healthcare billing workflows, for example) or as specific as the arrangement of data to suit the preferences of an individual manager.

Generally speaking, the use of expressions to apply business logic works best when limited to a small number of simple business rule cases. As mentioned earlier, the expression language is not ideal for multiple test conditions and therefore may not be ideally suited for multifaceted and complex business rules. For enterprise-level business rule application, consider other tools in SSIS, such as the Script component or Execute SQL task (for operations that can be performed at the relational database level), or perhaps a separate tool such as SQL Server Data Quality Services or Master Data Services.

CHOOSING BETWEEN COMPLEX EXPRESSIONS AND OTHER TOOLS

In my experience, the majority of uses of SSIS expressions involve short, simple expressions. Interrogating the value of a variable, modifying the contents of an existing column, comparing two values, and other similar operations tend to require relatively brief and uncomplicated logic as an SSIS expression. However, there are many cases where a short-and-sweet expression just won't get it done.

In these cases of more complicated logic, is an SSIS expression still the best choice? In some instances, the answer is no. As mentioned earlier, there are instances in the ETL cycle where the expression language is ill-suited to solve the problem. In cases where the logic required involves complexity that exceeds that which is practical or convenient for the SSIS expression language, a common pattern is to engage a separate tool to address the problem at hand. Some of the other methods for handling these complex logical scenarios are as follows:

- **Data source component:** Especially when working with relational source data, it can be simpler and faster (both in design time and runtime) to build the necessary logic into the source component instead of using an expression in SSIS.
- **Execute SQL task:** Sometimes it's easier to load the data to a relational store and then perform transformation and cleansing there rather than doing it inline within the SSIS package. This methodology differs slightly from that of traditional ETL and is typically branded as ELT (extract/load/transform). Using this model, you could use the Execute SQL task to transform the data once it has been loaded from the source to the relational database in which it will be transformed.
- **Script task:** When working in the control flow, you can substitute an instance of the Script task in place of an overly complex SSIS expression. When using a Script task for this purpose, you get the added benefits of IntelliSense, error handling, multistep operations, and the ability to include comments in your code.
- **Script component:** Replaces complex expressions within the data flow, for the same reasons as stated previously. In addition, the Script component may be used as a source, transformation, or destination in the data flow surface, giving you even greater control of the manipulation of data than by strictly using expressions.

- **Custom task/component:** If you find yourself reusing the same complex logic in many packages, consider creating a custom task or component that you can distribute to multiple packages without having to copy and paste script code to each package.
- **Third-party task/component:** Sometimes it's easier to buy (or borrow) than to build. There are hundreds, perhaps even thousands, of third-party tasks and components designed to extend the native behaviors of SSIS. In fact, many of these tools are freely available—often with the underlying source code in case you need to further customize the behavior of the tool.

There are no hard-and-fast rules defining when an expression may not be the best solution. However, there are a few design patterns that I tend to follow when deciding whether to use an expression or some other tool when applying dynamic logic in my SSIS packages. Typically, I will avoid using expressions in situations where

- **The expression will be exceptionally lengthy:** If the logic required in an expression would exceed more than a few hundred characters, a script or other tool is often a better choice.
- **The expression requires more than three levels of nesting:** Especially in cases where If/Then/Else logic is required, there's frequently a need to respond to more than one condition (if/then/elseif/then/else), and unfortunately, the only way to accomplish this in the SSIS expression language is by nesting conditional operators.
- **Complex string interrogation or manipulation is required:** Simple string manipulation is easy enough through SSIS expressions with the use of well-known functions such as SUBSTRING, REPLACE, LEFT/RIGHT, UPPER/LOWER, and REVERSE. However, more advanced operations (extracting text from the middle of a string, replacing multiple patterns of character(s), extracting numbers embedded in text, etc.) usually requires overly complex expressions. Further, some text operations such as regular expression (RegEx) matching are not natively supported in the SSIS expression language.
- **The logic requires a volatile type cast:** Because the SSIS expression language has no error handling in itself, a conversion that is prone to failure (text to number, Unicode to ASCII, moving from a larger to smaller capacity of the same type) may cause an undesired interruption in your package flow. Often, I'll wrap these into a Script task or Script component using a TryParse() method or a Try-Catch block, which allows a greater amount of flexibility in the event of a type cast failure.

The bottom line is that not every ETL challenge within SSIS should be solved using expressions. The expression language was intended as a lightweight solution, and used in that context, it is an outstanding supplement to the product line. Try to think of SSIS expressions as spackle; small, light, elegant, and used pervasively, but in small doses. As effective as spackle is, a building contractor would never think of building an entire house using only spackle. As with any tool, expressions in SSIS are best used in proper context and should not be considered as a one-size-fits-all solution to every problem.

Conclusion

ETL can be hard. Often, it's not the big design problems but the small “how do I . . . ?” tactical questions that collectively cause the most friction during SSIS development. The SSIS expression language was designed for these types of questions. Its small footprint, somewhat familiar syntax, and extensive usability across the breadth of SSIS make it an excellent addition to the capabilities within Integration Services. Used properly, it can help to address a variety of problem domains and hopefully ease the burden on the ETL developer.



Data Warehouse Patterns

SQL Server Integration Services is an excellent all-purpose ETL tool. Because of its versatility, it is used by DBAs, developers, BI professionals, and even business principals in many different scenarios. Sometimes it's a dump truck, used for the wholesale movement of enormous amounts of data. Other times it's more like a scalpel, carving out with precision just the right amount of data.

Though it is a great tool in other areas, SSIS truly excels when used as a data warehouse ETL tool. It would be hard to argue that data warehousing isn't its primary purpose in life. From native slowly changing dimension (SCD) elements to recently added CDC processing tasks and components, SSIS has all the hooks it needs to compete with data warehouse ETL tools at much higher price points.

In this chapter, we'll discuss design patterns applicable to loading a data warehouse using SQL Server Integration Services. From incremental loads to error handling and general workflow, we'll investigate methodologies and best practices that can be applied in SSIS data warehouse ETL.

Incremental Loads

Anyone who has spent more than ten minutes working in the data warehouse space has heard the term *incremental load*. Before we demonstrate design patterns for performing incremental loads, let's first touch on the need for an incremental load.

What Is an Incremental Load?

As the name implies, an *incremental load* is one that processes a partial set of data based only on what is new or changed since the last execution. Although many consider incremental loads to be purely time based (for example, grabbing just the data processed on the prior business day), it's not always that simple. Sometimes, changes are made to historical data that should be reflected in downstream systems, and unfortunately it's not always possible to detect when changes were made. (As an aside, those ubiquitous "last update timestamp" fields are notorious for being wrong.)

Handling flexible data warehouse structures that allow not only inserts, but updates as well, can be a challenging proposition. In this chapter, we'll surface a few design patterns in SSIS that you can use when dealing with changing data in your data warehouse.

Why Incremental Loads?

Imagine you are hired by a small shoe store to create a system through which the staff can analyze their sales data. Let's say the store averages 50 transactions a day, amounting to an average of 100 items (2 items per transaction). If you do simple math on the row counts generated by the business, you'll end up with fewer than 200 rows of data per day being sent to the analysis system. Even over the course of the entire year, you're looking at less than 75,000 rows of data. With such a small volume of data, why would you want to perform an incremental load? After all, it would be almost as efficient to simply dump and reload the entire analytical database each night rather than try to calculate which rows are new, changed, or deleted.

In a situation like the one just described, the best course of action might be to perform a full load each day. However, in the real world, few, if any, systems are so small and simple. In fact, it's not at all uncommon for data warehouse developers to work in environments that generate millions of rows of data per day. Even with the best hardware money can buy, attempting to perform a daily full load on that volume of data simply isn't practical. The incremental load seeks to solve this problem by allowing the systematic identification of data to be moved from transactional systems to the data warehouse. By selectively carving out only the data that requires manipulation—specifically, the rows that have been inserted, updated, or deleted since the last load—we can eliminate a lot of unnecessary and duplicate data processing.

The Slowly Changing Dimension

When you consider the application of incremental data loads in a data warehouse scenario, there's no better example than the slowly changing dimension (SCD). The nature of dimensional data is such that it often does require updates by way of manipulating existing rows in the dimension table (SCD Type 1) or expiring the current record and adding a new row for that value, thus preserving the history for that dimension attribute (SCD Type 2).

Although the slowly changing dimension is certainly not the only data warehouse structure to benefit from an incremental load, it is one of the most common species of that animal. As such, we'll focus mostly on SCD structures for talking points around incremental loads.

Incremental Loads of Fact Data

Some data warehouse scenarios require the changing of fact data after it has been loaded to the data warehouse. This scenario is typically handled through a secondary load with a negating entry and a delta record, but some designs require the original fact record to be corrected.

In such cases, the same methodology used for SCD data may also apply to fact data. However, pay careful consideration to performance when applying SCD methods to fact data. Fact data is exponentially more voluminous than dimension data and typically involves millions, and sometimes billions, of records. Therefore, apply the SCD pattern to fact data only if you must. If there's any flexibility at all in the data warehouse (DW) design, use the delta record approach instead.

Incremental Loads in SSIS

Microsoft SQL Server, and SSIS specifically, have several tools and methodologies available for managing incremental data loads. This section discusses design patterns around the following vehicles:

- Native SSIS components (Lookup transform, conditional split, etc.)
- Slowly Changing Dimension Wizard

- MERGE statement in T-SQL
- Change data capture (CDC)

Each of these tools is effective when used properly, though some are better suited than others for different scenarios. We'll now step through the design patterns with each of these.

Native SSIS Components

The first incremental load pattern we'll explore is that of using native components within SSIS to perform the load. Through the use of lookups, conditional splits, and OLE DB command components, we can create a simple data path through which we can process new and changed data from our source system(s).

Employing this design pattern is one of the most common ways to perform an incremental load using SSIS. Because all of the components used in this pattern have been around since SSIS was introduced in 2005, it's a very mature and time-tested methodology. Of all of the incremental methodologies we'll explore, this is certainly the most flexible. When properly configured, it can perform reasonably well. This is also the design pattern with the fewest external dependencies; almost any data can be used as a source, it does not require database engine features such as CDC to be enabled, and it does not require any third-party components to be installed.

The Moving Parts

When you are using this design pattern, the most common operations you will perform will include the following steps:

1. Extract data from the data source(s). If more than one source is used, they can be brought together using the appropriate junction component (Merge, Merge Join, or Union All transforms).
2. Using the Lookup transform, join the source data with the target table based on the business key(s).
3. Route changed rows to the target table. Unmatched rows from the step 2 can then be routed directly to the target table using the appropriate database destination component.
4. For the source rows that have a business key match in the target table, compare the other values that may change. If any of those source values differs from the value found for that row in the destination table, those rows in the destination table will be updated with the new values.

Figure 11-1 shows a typical data flow design for these operations. In the first callout, you can see that we need to set the option on the Lookup transform to Redirect Rows to No Match Output. When you use this setting, any source rows that are not resolved to an existing row in the destination table are sent down an alternate path—in this case, directly to the destination table.

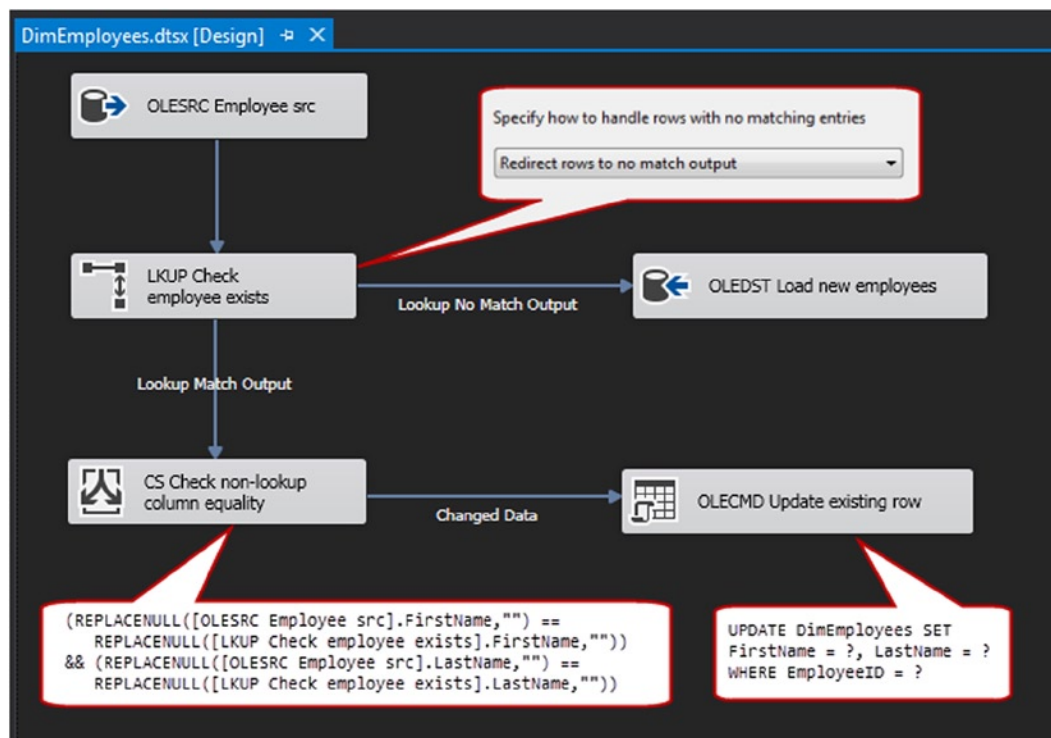


Figure 11-1. Incremental load using atomic SSIS components

Next, we'll apply the Conditional Split transform to the matched rows. As shown in the snippet on the lower left of Figure 11-1, we'll use a bit of the SSIS expression language to compare equivalent columns between source and destination. Any rows with an exact match will not go through any further processing (though you could send them to a Row Count transform if you want to capture the count of source rows with no action taken).

Finally, the rows that matched the business key but not the subsequent attribute values will be sent to the OLE DB Command transform. The code on the lower right in Figure 11-1 shows a snippet of the SQL code in which we perform a parameterized update of each row in the destination table. It is important to note that the OLE DB Command transform performs a row-by-row update against the target table. You can typically leverage this pattern as shown because most incremental operations are heavy on new rows and have far fewer updates than inserts. However, if a particular implementation requires the processing of a very large amount of data, or if your data exploration indicates that the number of updates is proportionally high, consider modifying this design pattern to send those rows meant for update out to a staging table where they can be more efficiently processed in a set-based operation.

Typical Uses

As mentioned previously, this incremental load design pattern is quite useful and mature. This pattern fits especially well when you are dealing with nonrelational source data, or when it's not possible to stage incoming data before processing. This is often the go-to design for incremental loads, and it fits most of this type of scenario reasonably well.

Make sure you keep in mind that, because we're performing the business key lookup and the column equivalency tests within SSIS, some resource costs are associated with getting the data into the SSIS memory space and then performing said operations against the data. Therefore, if a particular implementation involves a very large amount of data, and the source data is already in a SQL Server database (or could be staged there), another design pattern such as the T-SQL MERGE operation (to be covered shortly) might be a better option.

The following subsections describe components of the incremental load pattern and configuration options for each.

Lookup Caching Options

When performing lookup operations, you want to consider the many options available for managing lookup caching. Depending on the amount of data you're dealing with, one of the following caching design patterns may help reduce performance bottlenecks.

Table Cache

A table cache for lookups is populated prior to executing a Data Flow task that requires the lookup operation. SSIS can create and drop the table as needed using Execute SQL tasks. It can be populated via the Execute SQL task or a Data Flow task. Most of the time, the data I need to build a table cache is local to the destination and contains data from the destination, so I often use T-SQL in an Execute SQL task to populate it.

You can maintain the table cache by using truncate-and-load. However, for larger sets of lookup data, you may wish to consider maintaining the table cache using incremental load techniques. This may sound like overkill, but when you need to perform a lookup against a billion-row table (it happens, trust me), the incremental approach starts to make sense.

Cache Transformation and Cache Connection Manager

If you find you need to look up the same data in multiple Data Flow tasks, consider using the Cache transform along with the Cache connection manager. The Cache connection manager provides a memory-resident copy of the data supplied via the Cache transform. The cache is loaded prior to the first Data Flow task that will consume the lookup data, and the data can be consumed directly by a Lookup transform. Precaching data in this manner supports lookups, but it also provides a way to "mark" sets of rows for other considerations such as loading. Later in this chapter, we will explore late-arriving data and discuss patterns for managing it. One way to manage the scenario of data continuing to arrive after the load operation has started is to create a cache of primary and foreign keys that represent completed transactions, and then join to those keys in Data Flow tasks throughout the load process. Will you miss last-second data loading in this way? Yes, you will. But your data will contain complete transactions. One benefit of executing incremental loads with table caches is the ability to execute the load each month, week, evening, or every five minutes; only complete transactions that have arrived since the last load executed will be loaded.

If you find you need to use the same lookup data across many SSIS packages (or that the cache is larger than the amount of available server RAM), the Cache connection manager can persist its contents to disk. The Cache connection manager makes use of the new and improved RAW file format, a proprietary format for storing data directly from a Data Flow task to disk, completely bypassing connection managers. Reads and writes are very fast as a result, and the new format persists column names and data types.

Load Staging

Another pattern worth mentioning here is load staging. Consider the following scenario: a data warehouse destination table is large and grows often. Since the destination is used during the load window, dropping keys and indexes is not an option to improve load performance. All related data must become available in the data warehouse at roughly the same time to maintain source-transactional consistency. By nature, the data does not lend itself to partitioning. What to do?

Consider *load staging*, where all the data required to represent a source transaction is loaded into stage tables on the destination. Once these tables are populated, you can use Execute SQL tasks to insert the staged rows into the data warehouse destination table. If timed properly, you may be able to use a bulk insert to accomplish the load. Often, data loads between tables in the same SQL Server instance can be accomplished more efficiently using T-SQL rather than the buffered SSIS Data Flow task. How can you tell which will perform better? Test it!

The Slowly Changing Dimension Wizard

The Slowly Changing Dimension (SCD) Wizard is another veteran of the SSIS incremental load arsenal. As its name implies, it is designed specifically for managing SCD elements in a data warehouse. However, its use is certainly not limited to dimension processing.

The SCD Wizard has been a part of SSIS ever since the product's introduction in 2005. At first glance, it is the natural choice for handling slowly changing dimension data in SSIS. This tool is built right into Integration Services, and it is designed specifically for the purpose of SCD processing.

The SCD Wizard is surfaced as a transformation in SSIS and is leveraged by connecting the output of a data source (the incoming data) to the input of the SCD transformation. Editing the SCD transformation will kick off the wizard, as shown in Figure 11-2.

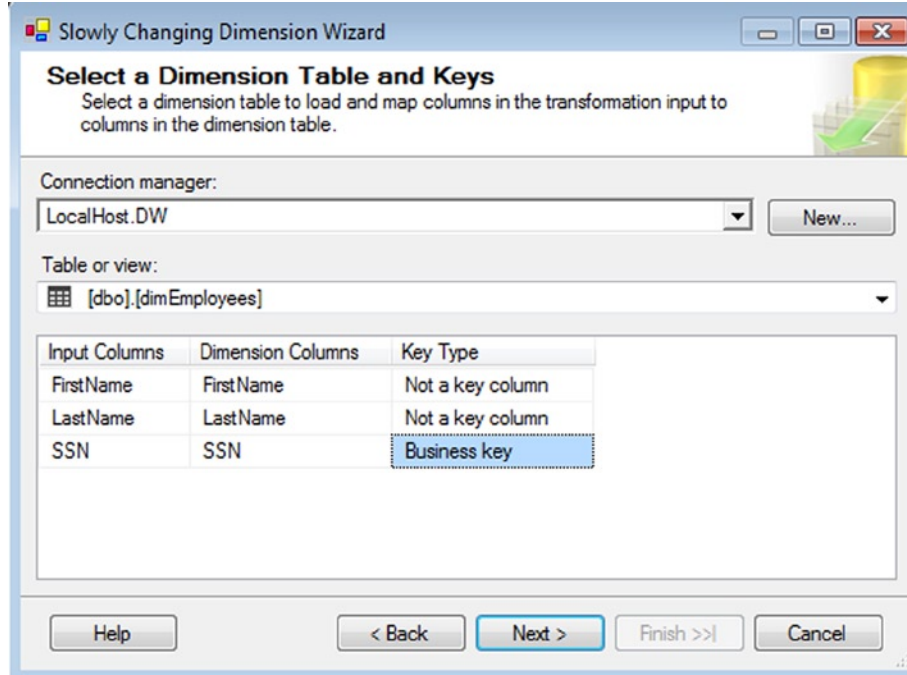


Figure 11-2. The SCD Wizard showing alignment of columns

From there, the wizard will guide you through the selection of the necessary elements of the SCD configuration, including the following:

- Which columns should be engaged as part of the SCD, along with the option to handle changes as simple updates (Type 1) or historical values (Type 2).
- How to identify current vs. expired rows, if any Type 2 columns are present; you can specify either a flag or a date span to indicate currency of SCD records.
- How to handle inferred members (discussed in more depth shortly).

When you've completed the SCD Wizard, several new elements are automatically added to the data flow. Figure 11-3 shows an example of the transforms and destinations added when using a combination of Type 1 and Type 2 fields, fixed-attribute (static) fields, and inferred member support. The SCD Wizard adds only those components pertinent to the design specified, so the final result may look a bit different than the example in this figure, depending on how the wizard is configured.

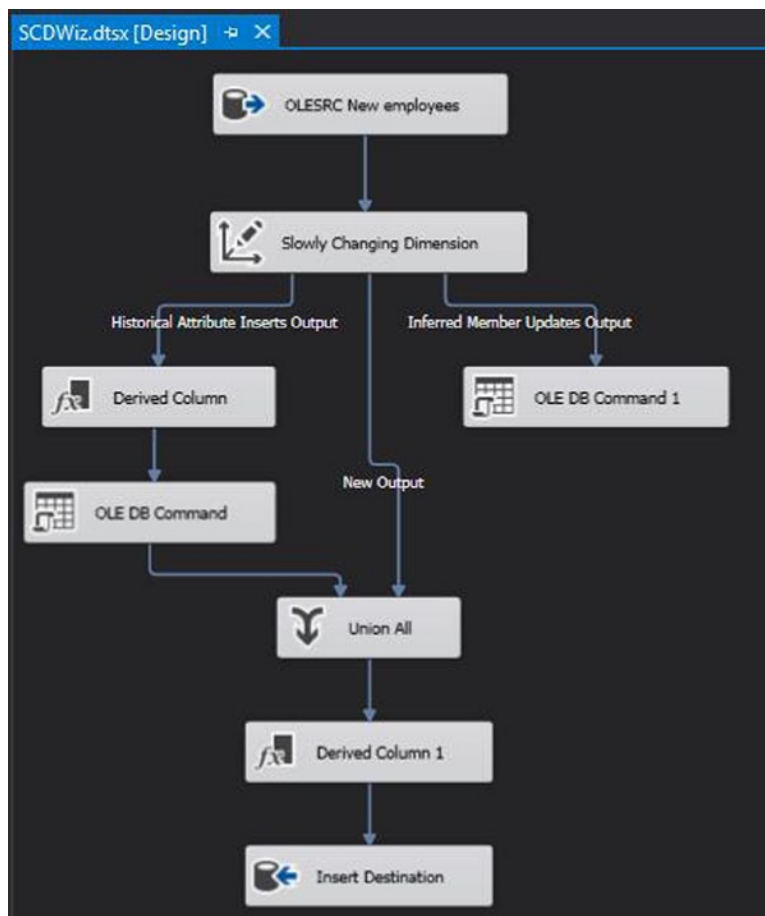


Figure 11-3. SCD Wizard output

Of all of the SCD design patterns, the SCD Wizard is arguably the easiest to use for simple SCD scenarios, and it offers the fastest turnaround at design time. For small, simple dimensions, the wizard can be an effective tool for managing changing data.

However, the SCD Wizard does have some significant drawbacks.

Performance: The wizard performs reasonably well against small sets of data. However, because many of the operations are performed on a row-by-row basis, leveraging this transform against sizeable dimensions can cause a significant performance bottleneck. Some data warehouse architects have strong feelings against the SCD Wizard, and this is often the chief complaint.

Destructive changes: As I mentioned, when you run the wizard, all of the required transforms and destinations are created automatically. Similarly, if you reconfigure the SCD transform (for example, changing a column type from a Type 1 to a Type 2 historical), the existing design elements are removed and added back to the data flow. As a result, any changes that you've made to that data path will be lost if you make a change to the SCD transform.

No direct support for auditing: Although you can add your own auditing logic, it's not a native function of this component. Further, because any changes to the SCD transformation will delete and re-create the relevant elements of the data flow, you'll have to reconfigure that auditing logic if any changes are made to the SCD transform.

As a result of these shortcomings (especially the performance implications), the SCD Wizard doesn't see a lot of action in the real world. I don't want to beat up on this tool too much because it does have some usefulness, and I don't necessarily recommend that you avoid it completely. However, like any specialty tool, it should be used only where appropriate. For small sets of SCD data that do not require complex logic or specialized logging, it can be the most effective option for SCD processing.

The MERGE Statement

Although technically not a function of SSIS, the MERGE statement has become such a large part of incremental data loads that any discussion focused on data warehousing design patterns would not be complete without this tool being covered.

A Little Background

Prior to version 2008, there were no native upserts (UPDATE/INSERT) in Microsoft SQL Server. Any operation that blended updates and inserts required either the use of a cursor (which often performs very poorly) or two separate queries (usually resulting in duplicate logic and redundant effort). Other relational database products have had this capability for years—in fact, it has been present in Oracle since version 9i (around 2001). Naturally, SQL Server professionals were chomping at the bit for such capabilities.

Fortunately, they got their wish with the release of SQL Server 2008. That version featured the debut of the new MERGE statement as part of the T-SQL arsenal. MERGE allows three simultaneous operations (INSERT, UPDATE, and DELETE) against the target table.

The anatomy of a MERGE statement looks something like this:

1. Specify the source data.
2. Specify the target table.
3. Choose the columns on which to join the source and target data.
4. Indicate the column alignment between the two sets of data to be used for determining whether matched records are different in some way.
5. Define the logic for cases when the data is changed, or if it only exists in either the source or the destination.

The new MERGE capabilities are useful for DBAs and database developers alike. However, for data warehouse professionals, MERGE was a game changer in terms of managing SCD data. Not only did this new capability provide an easier way to perform upsert operations, but it also performed very well.

MERGE in Action

Since there are no native (read: graphical) hooks to the MERGE statement in Integration Services, the implementation of MERGE in an SSIS package is done through an Execute SQL task.

To explore this design pattern, let's first examine the typical flow for using the T-SQL MERGE functionality within a data warehouse SSIS load package. Again, we'll use the SCD scenario as a basis for exploration, but much of the same logic would apply to other uses of MERGE.

As part of an SCD MERGE upsert process, our SSIS package would contain tasks to perform the following functions:

1. Remove previously staged data from the staging table.
2. Load the staging table from the source system.
3. Clean up the staged data (if required).
4. Execute the MERGE statement to upsert the staged data to the dimension table.
5. Log the upsert operation (optional).

A typical control flow design pattern is shown in Figure 11-4.

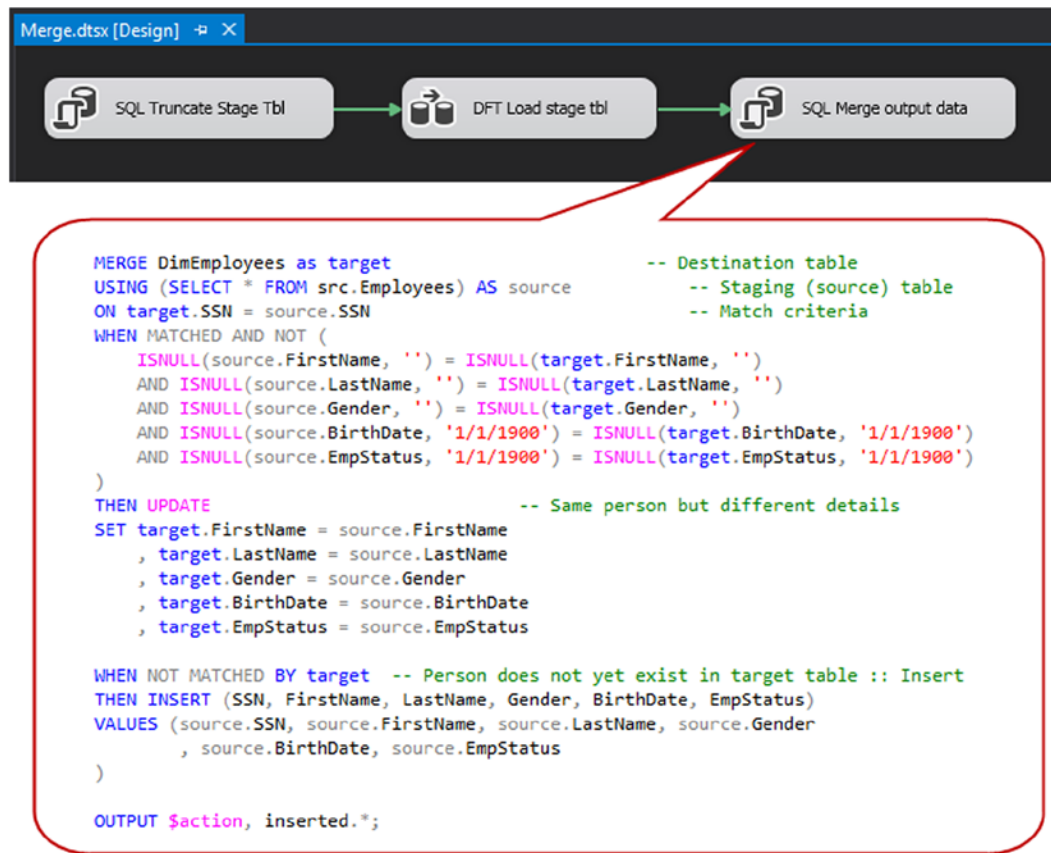


Figure 11-4. Using MERGE against a SCD table

Also note the large callout in Figure 11-4 with the T-SQL code used for the MERGE statement. In the interest of maintaining focus, I won't try to provide comprehensive coverage of the MERGE statement here, but I'll point out a couple of the high points:

- The ON clause (third line) indicates the field on which we join the source data with the destination data. Note that we can use multiple fields on which to align the two sets.
- The ten-line code block following WHEN MATCHED AND NOT . . . indicates which of the fields will be checked to see if the data in the destination table differs from the source data. In this case, we're checking ten different fields, and we'll process an update to the destination if any of those fields is different between the source and destination. Also note the liberal use of the ISNULL() function against the destination table. This is recommended to ensure that rows in the destination table containing NULL values are not inadvertently skipped during the MERGE.
- In the code block immediately following, we update rows in the target table that have a valid match against the source but have one or more values that differ between the two.

- In the code block beginning with `WHEN NOT MATCHED BY target...`, any source rows not matched to the specified key column(s) of an existing dimension record are written as new rows to that dimension table.
- Finally, we use the `OUTPUT` clause to select the action description and insert data. We can use the output of this to write to our auditing table (more on that momentarily).

You'll notice that we're handling this dimension processing as a Type 1 dimension, in which we intentionally overwrite the previous values and do not maintain a historical record of past values. It is possible to use the `MERGE` command to process Type 2 dimensions that retain historical values, or even those with a mixture of Type 1 and Type 2 attributes. In the interest of brevity, I won't try to cover the various other uses of `MERGE` as it applies to slowly changing dimensions, but I think you'll find that it's flexible enough to handle most Type 1 and Type 2 dimensions.

It is also worth noting that you can also use the `MERGE` statement to delete data in addition to performing inserts and updates. It's not as common to delete data in data warehouse environments as it is in other settings, but you may occasionally find it necessary to delete data from the target table.

Auditing with MERGE

As with other data warehouse operations, it's considered a best practice to audit, at a minimum, the row counts of dimensional data that is added, deleted, or changed. This is especially true for `MERGE` operations. Because multiple operations can occur in the same statement, it's important to be able to track those operations to assist with troubleshooting, even if comprehensive auditing is not used in a given operation.

The `MERGE` statement does have a provision for auditing the various data operations. As shown in the example in Figure 11-4, we can use the `OUTPUT` clause to select out of the `MERGE` statement the `INSERT`, `UPDATE`, or `DELETE` operations. This example shows a scenario where the data changes would be selected as a result set from the query, which could subsequently be captured into a package variable in SSIS and processed from there. Alternatively, you could modify the `OUTPUT` clause to insert data directly into an audit table without returning a result set to SSIS.

Change Data Capture (CDC)

Along with the `MERGE` capability, another significant incremental load feature first surfaced in SQL Server 2008: change data capture (CDC). CDC is a feature of the database engine that allows the collection of data changes to monitored tables.

Without jumping too far off track, here's just a bit about how CDC works. CDC is a supply-side incremental load tool that is enabled first at the database level and then implemented on a table-by-table basis via capture instances. Once CDC is enabled for a table, the database engine uses the transaction log to track all DML operations (`INSERTs`, `UPDATEs`, and `DELETEs`), logging each change to the change table for each monitored table. The change table contains not only the fact that there was a change to the data, but it also maintains the history of those changes. Downstream processes can then consume just the changes (rather than the entire set of data) and process the, `INSERTs`, `UPDATEs`, and `DELETEs` in any dependent systems.

CDC in Integration Services

SSIS can consume CDC data in a couple of different ways. First, using common native SSIS components, you can access the change table to capture the data changes. You can keep track of which changes have been processed by SSIS by capturing and storing the log sequence number (LSN) by using a set of system stored procedures created when CDC is enabled.

The manual methods are still valid; however, new to SSIS in SQL Server 2012 is an entirely new set of tools for interfacing with CDC data. Integration Services now comes packaged with a new task and two new components that help to streamline the processing of CDC data:

CDC Control task: This task is used for managing the metadata around CDC loads. Using the CDC Control task, you can track the start and end points of the initial (historical) load, as well as retrieve and store the processing range for an incremental load.

CDC Source: The CDC Source is used to retrieve data from the CDC change table. It receives the CDC state information from the CDC Control task by way of an SSIS variable, and it will selectively retrieve the changed data using that marker.

CDC Splitter: The CDC Splitter is a transform that will branch the changed data out into its various operations. Effectively a specialized Conditional Split transform, it will use the CDC information received from the CDC Source and send the rows to the Insert, Update, Delete, or Error path accordingly.

For the purposes of reviewing CDC capabilities as part of an SSIS incremental load strategy, we'll stick with the new task and components present in SSIS 2012. In systems using SQL Server 2008, know that you can meet the same objectives by employing the manual extraction and LSN tracking briefly described previously.

Change Detection in General

Detecting changes in data is a subsistence of data integration. Much has been written on the topic from sources too numerous to list. Although CDC provides handy change detection in SQL Server, it was possible (and necessary!) to achieve change detection before the advent of CDC. It is important to note that CDC is not available in all editions of SQL Server; it is also not available in other relational database engines.

Checksum-Based Detection

One early pattern for change detection was using the Transact-SQL Checksum function. Checksum accepts a string as an argument and generates a numeric hash value. But Checksum performance has proven less than ideal, generating the same number for different string values. Steve Jones blogged about this behavior in a post entitled “SQL Server Encryption—Hashing Collisions” (www.sqlservercentral.com/blogs/steve_jones/2009/06/01/sql-server-encryption-hashing-collisions/). Michael Coles provided rich evidence to support Steve's claims in the post's comments (www.sqlservercentral.com/blogs/steve_jones/2009/06/01/sql-server-encryption-hashing-collisions/#comments). In short, the odds of collision with Checksums are substantial, and you should not use the Checksum function for change detection.

So, what can you use?

Detection via Hashbytes

One good alternative to the Checksum function is the Hashbytes function. Like Checksum, Hashbytes provides value hashing for a string value or variable. Checksum returns an integer value; Hashbytes returns a binary value. Checksum uses an internal algorithm to calculate the hash; Hashbytes uses standard encryption algorithms. The sheer number of values available to each function is one reason Hashbytes is a better choice. Checksum's int data type can return +/-231 values, whereas Hashbytes can return +/-2127 values for MD2, MD4, and MD5 algorithms and +/-2159 values for SHA and SHA1 algorithms.

Brute Force Detection

Believe it or not, a “brute force” value comparison between sources and destinations remains a viable option for change detection. How does it work? You acquire the destination values by way of either a second Source component or a Lookup transform in the SSIS Data Flow task. You match the rows in the source and destination by using an alternate (or business) key—a value or combination of values that uniquely identifies the row in both source and destination—and then compare the non-key column values in the source row to the non-key values in the destination row.

Remember, you are attempting to isolate changes. It is assumed that you have separated the new rows—data that exists in the source and not in the destination—and perhaps you have even detected deleted rows that exist in the destination but are no longer found in the source. Changed and unchanged rows remain. Unchanged rows are just that: the alternate keys align, as does every other value in each source and destination column. Changed rows, however, have identical alternate keys and one or more differences in the source and destination columns. Comparing the column values and accounting for the possibility of NULLs remains an option.

Historical Load

There should be a separate process to populate the historical data for each of the tracked CDC tables. This historical load is designed to be executed just once, and it would load data to the destination system from as far back as is required. As shown in Figure 11-5, two CDC Control tasks are required. The first one (configured as shown in the callout) is used to set the beginning boundary of the data capture. With this information, the CDC Control task writes the CDC state to the specified state table. The second CDC Control task marks the end point of the initial load, updating the state table so the appropriate starting point will be used on subsequent incremental loads. Between these two tasks sits the data flow, which facilitates the historical load of the target table.

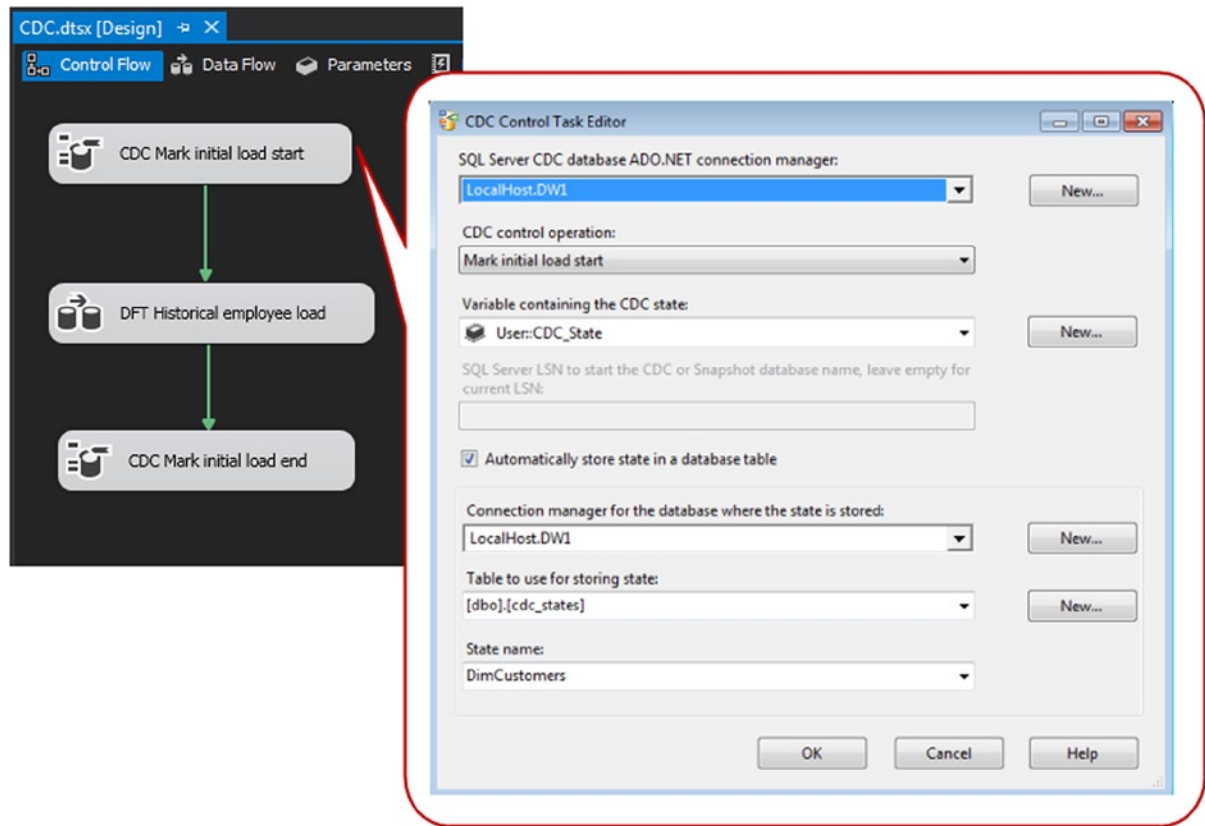


Figure 11-5. CDC Control task for an initial historical load

Incremental Load

Because of the inherent differences between historical loads and incremental loads, it's almost always preferable to create separate packages (or package groups) for each of these. Although there are similarities, there are enough differences to justify separating the logic into different sandboxes.

For the control flow elements of the historical load, this incremental load pattern will also use two CDC Control tasks with a data flow between them. We'll need to slightly change the configuration of these tasks so that we retrieve and then update the current processing range of the CDC operation. As shown in Figure 11-6, we'll set the operation for the first of these as Get Processing Range, which would be followed by Update Processing Range after the incremental load is complete.

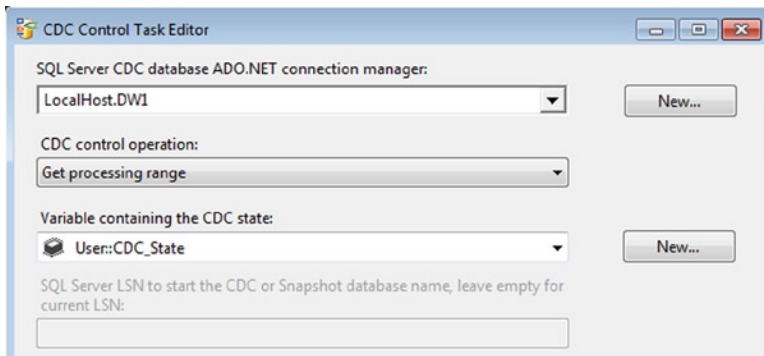


Figure 11-6. Get processing range for CDC incremental load

Note The CDC Control task is a versatile tool that includes several processing modes to handle various CDC phases, including dealing with a snapshot database or quiescence database as a source. A complete listing of the processing modes can be found here: <http://msdn.microsoft.com/en-us/library/hh231079.aspx>.

Within the data flow, the CDC Source should be set with the table from which to capture, the capture instance, and the processing mode. In this case, we're going to use "Net" to retrieve the net changes to the CDC table. See Figure 11-7.

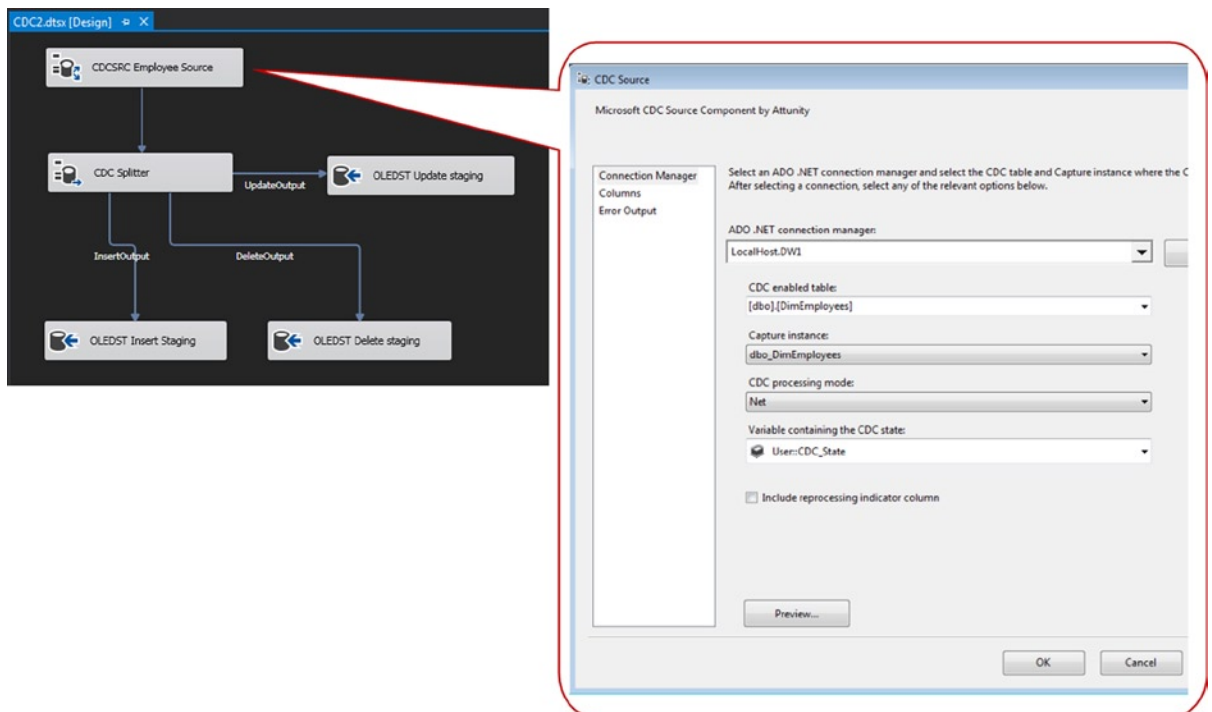


Figure 11-7. CDC Source

The CDC Splitter breaks apart the data stream and sends rows out to the insert, update, and delete outputs. From there, we'll write the update and delete rows to a staging table so we can process those as high-performance, set-based operations. The insert records can go directly into the output table.

It's worth mentioning here that several CDC processing modes are available through the CDC Source component. The example in Figure 11-7 illustrated the use of the Net setting, which is the most common mode in most data warehouse scenarios. However, depending on the ETL requirements and source system design, you may opt for one of the other processing modes, as follows:

All: Lists each change in the source, not just the Net result

All with Old Values: Includes each change plus the old values for updated records

Net with Update Mask: Is for monitoring changes to a specific column in the monitored table

Net with Merge: Is similar to Net, but the output is optimized for consumption by the T-SQL MERGE statement

Typical Uses

CDC represents a shift in the incremental load methodology. The other methods described here apply a downstream approach to incremental loading, with a minimally restrictive extraction from the source and a decision point late in the ETL flow. CDC, on the other hand, processes the change logic further upstream, which can help lighten the load on SSIS and other moving parts in the ETL.

If CDC is in place (or could be implemented) in a source system, it's certainly worth considering using this design pattern to process incremental loads. It can perform well, can reduce network loads due to processing fewer rows from the source, and requires fewer resources on the ETL side. It's not appropriate for every situation, but CDC can be an excellent way to manage incremental loads in SQL Server Integration Services.

Keep in mind that the use of CDC as a design pattern isn't strictly limited to Microsoft SQL Server databases. CDC may also be leveraged against CDC-enabled Oracle database servers.

Data Errors

Henry Wadsworth Longfellow once wrote, "Into each life some rain must fall." The world of ETL is no different, except that rain comes in the form of errors, often as a result of missing or invalid data. We don't always know when they're going to occur. However, given enough time, something is going to go wrong: late-arriving dimension members, packages executed out of order, or just plain old bad data. The good news is that there are data warehousing design patterns that can help mitigate the risk of data anomalies that interrupt the execution of Integration Services packages.

To address patterns of handling missing data, we're going to concentrate mostly on missing dimension members, as this is the most frequent cause of such errors. However, you can extend some of the same patterns to other elements that are part of or peripheral to data warehousing.

Simple Errors

The vast majority of errors can and should be handled inline, or simply prevented before they occur. Consider the common case of data truncation: you have a character type field that's expected to contain no more than 50 characters, so you set the data length accordingly. Months later, you get a late-night phone call (most likely when you're on vacation or when you're out at a karaoke bar with your fellow ETL professionals) informing you that the SSIS package has failed because of a truncation error. Yep, the party's over.

We've all been bitten before by the truncation bug or one of its cousins—the invalid data type error, the unexpected NULL/blank value error, or the out-of-range error. In many cases, however, these types of errors can be handled through defensive ETL strategies. By using tasks and components that detect and subsequently correct or redirect nonconforming rows, we can handle minor data errors such as this without bubbling up a failure that stops the rest of the ETL from processing.

Missing Data

With respect to data warehousing, a more common example of handling errors inline is the case of late-arriving dimension data. As shown in Figure 11-8, the typical pattern is to load the dimensions first, followed by a load of the fact tables. This helps to ensure that the fact records will find a valid dimension key when the former is loaded to the data warehouse.

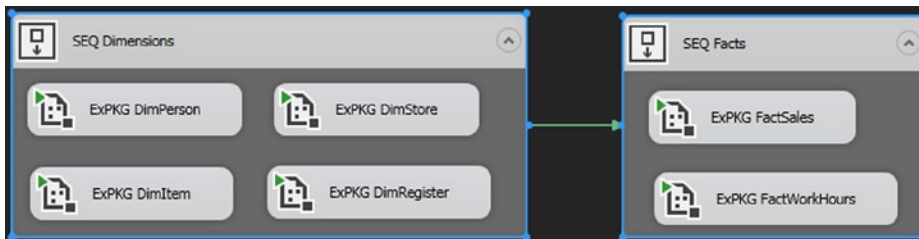


Figure 11-8. Typical data warehouse methodology of loading dimensions, then facts

However, this pattern breaks down when you attempt to process fact records that reference dimension data that does not yet exist in the data warehouse. Consider the case of holiday retail sales: because things happen so quickly at the retail level during the end-of-year holiday season, it's not uncommon for last-minute items to appear at a store's dock with little or no advance notice. Large companies (retailers included) often have multiple systems used for different purposes that may or may not be in sync, so a last-minute item entered in the point-of-sale (POS) system may not yet be loaded in the sales forecasting system. As a result, an attempt to load a data warehouse with both POS and forecasting data may not fit this model because we would have fact data from the sales system that do not yet have the required dimension rows from the forecasting system.

At this point, if the decision is made to handle this issue inline, there are a few different methodologies we can use. These are described next.

Use the Unknown Member

The fastest and simplest pattern to address the issue of missing dimension members is to push fact records with missing dimension data into the fact table while marking that dimension value as unknown. In this case, the fact records in question would be immediately available in the data warehouse; however, all of the unknowns would, by default, be grouped together for analytical purposes. This pattern generally works best when the fact record alone does not contain the required information to create a unique new dimension record.

This design pattern is fleshed out in Figure 11-9. Using the No Match output of the Lookup transform, we're sending the fact records not matched to an existing [Item] dimension member to an instance of the Derived Column transform, which sets the value of the missing dimension record to the unknown member for that dimension (in most cases, represented by a key value of -1). The matched and unmatched records are then brought back together using the Union All transform.

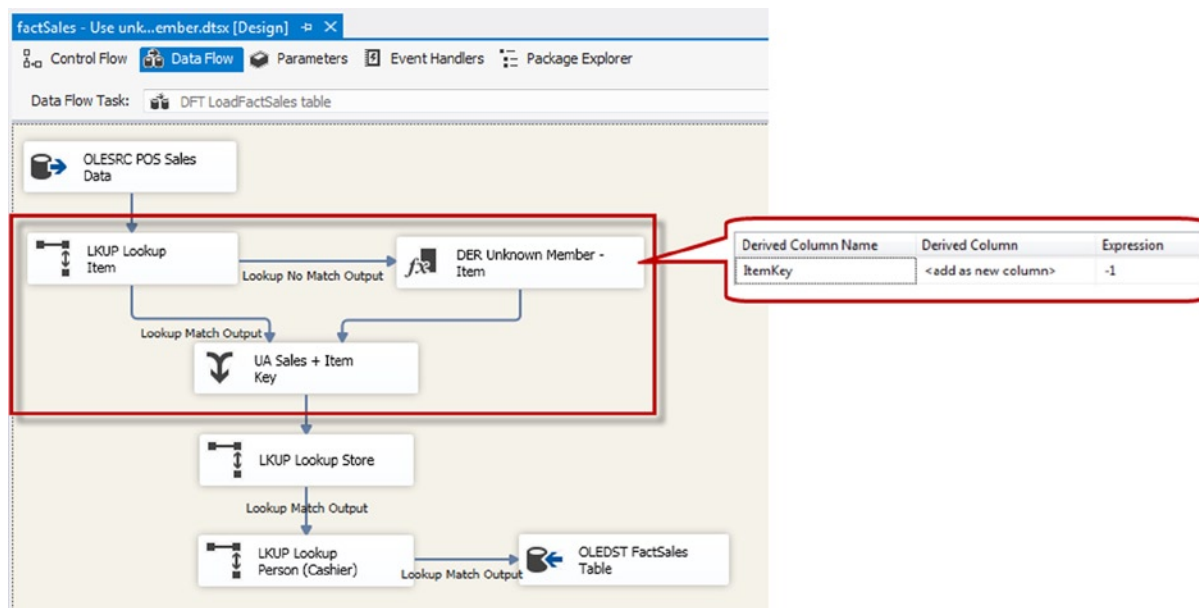


Figure 11-9. Using Unknown Member for missing dimension member

It is important to note that this design pattern should also include a supplemental process (possibly consisting of just a simple SQL statement) to periodically attempt to match these modified facts with their proper dimension records. This follow-up step is required to prevent the fact data in question from being permanently linked to the unknown member for that dimension.

Add the Missing Dimension Member

Using this design pattern, you can add missing dimension records on the fly as part of the fact package using as much dimension data as is provided by the fact data. In this scenario, the fact records are immediately available in the data warehouse, just like they were for the previous design pattern, but this methodology has the added benefit of matching the fact record to its proper dimension member. In most cases, this allows you to immediately associate the correct dimension data rather than group the unmatched data into the unknown member bucket.

Like the previous pattern, this method does come with a couple of caveats. First of all, the fact record must contain all of the information to 1) satisfy the table constraints (such as NOT NULL restrictions) on the dimension table, and 2) create a uniquely identifiable dimension row using the business key column(s). Also, since we're deriving the newly added dimension member from the incoming fact records, in most cases, it can be reasonably assumed that the incoming fact data will not completely describe the dimension member. For that reason, you should also complement this design pattern with a process that attempts to fill in the missing dimension elements (which may already be addressed as part of a comprehensive slowly changing dimension strategy).

As shown in Figure 11-10, here we use a methodology similar to the previous example. However, instead of simply assigning the value of the unknown member using the Derived Column transform, we leverage an instance of the OLE DB Command transform to insert the data for the missing dimension record in the fact table into the dimension table. The SQL statement is shown in the callout, and in the properties of the OLE DB command, we map the placeholders (indicated by question marks) to the appropriate values from the fact record.

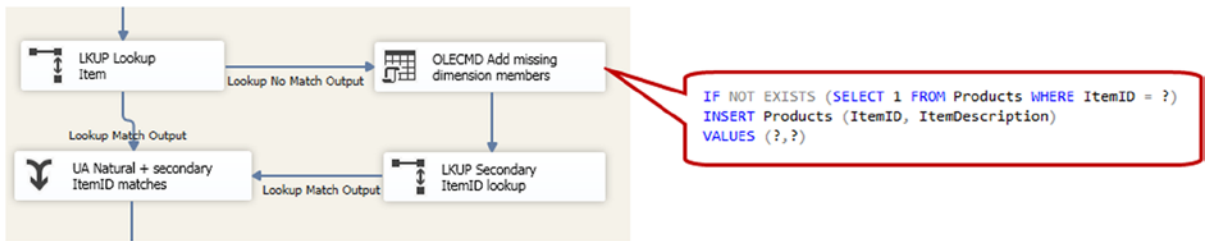


Figure 11-10. Adding the missing dimension member

After adding the missing member, we send those rows to a secondary ItemID lookup, which will attempt (successfully, unless something goes terribly wrong) to match the previously unmatched data with the newly added dimension records in the DimItem table. It is important that you remember to set the cache mode to either Partial Cache or No Cache when using a secondary lookup in this manner. The default lookup cache setting (Full Cache) will buffer the contents of the Item dimension table before the data flow is initiated, and as a result, none of the rows added during package execution would be present in this secondary lookup. To prevent all of these redirected fact rows from failing the secondary Item dimension lookup, use one of the non-default cache methods to force the package to perform an on-demand lookup to include the newly added dimension values.

Regarding the secondary lookup transformation methodology, you might wonder if the second lookup is even necessary. After all, if we perform the insert in the previous step (OLE DB command), couldn't we just collect the new Item dimension key value (a SQL Server table identity value, in most cases) using that SQL statement? The answer is a qualified yes, and in my experience, that is the simpler of the two options. However, I've also found that some ETL situations—in particular, the introduction of parallel processes performing the same on-the-fly addition of dimension members—can cloud the issue of collecting the identity value of the most recently inserted record. From that perspective, I lean toward using the secondary lookup in cases such as this.

Triage the Lookup Failures

For missing dimension records, the most common approaches typically involve one of the preceding. However, on occasion it will be necessary to delay the processing of fact records that do not match an existing dimension record. In cases such as this, you'll need to create a triage table that will store the interim records until they can be successfully matched to the proper dimension.

As shown in Figure 11-11, we're adding a couple of additional components to the ETL pipeline for this design pattern. At the outset, we need to use two separate sources of data: one to bring in the new data from the source system, and the other for reintroducing previously triaged data into the pipeline. Further into the data flow, the example shows that we are redirecting the unmatched fact records to another table rather than trying to fix the data inline.

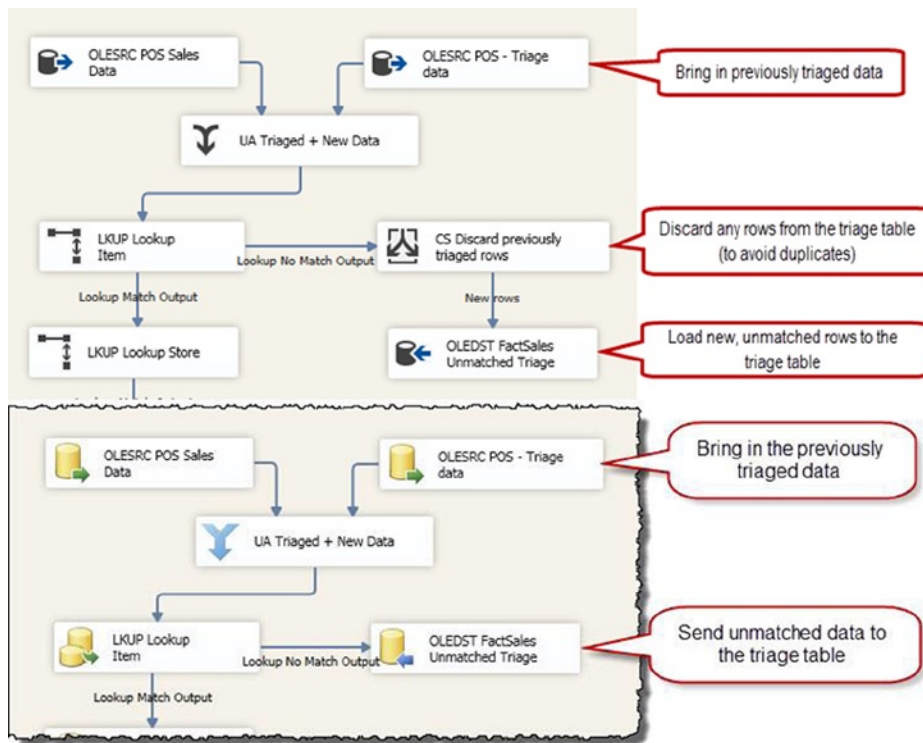


Figure 11-11. Use a triage table to store unmatched fact data

As an aside, this pattern could be modified to support manual intervention for correcting failed lookups as well. If the business and technical requirements are such that the unmatched fact data must be reviewed and corrected by hand (as opposed to a systematic cleanup), you could eliminate the triage source so that the triage data is not reintroduced into the data flow.

Coding to Allow Errors

Although it may sound like an oxymoron, it's actually a common practice to code for known errors. In my experience, the nature of most ETL errors dictates that the package execution can continue, and any errors or anomalies that cannot be addressed inline are triaged, as shown in the previous example (with the appropriate notification to the person responsible, if manual intervention is required), for later resolution. However, there are situations where the data warehouse ETL process should be designed to fail if error conditions arise.

Consider the case of financial data. Institutions that store or process financial data are subject to frequent and comprehensive audits, and if inconsistent data appeared in a governmental review of the data, it could spell disaster for the organization and its officers. Even though a data warehouse may not be subject to the same to-the-penny auditing scrutiny as OLTP systems, there is still an expectation of consistency when matters of money and governmental regulation are involved. In the case of a data warehouse load where nonconforming data is encountered, quite possibly the best thing to occur is that the package fails gracefully, rolling back any changes made as part of the load.

Fail Package on Error

Extending the financial data example mentioned previously, let's examine the design pattern to facilitate a failure in the event of a lookup error. In reality, this is the default behavior. As shown in Figure 11-12, we use the default setting of Fail Component on the lookup components, which stops the execution of the package if a row is encountered that cannot be matched to either the GL Account or GL Subaccount lookup.

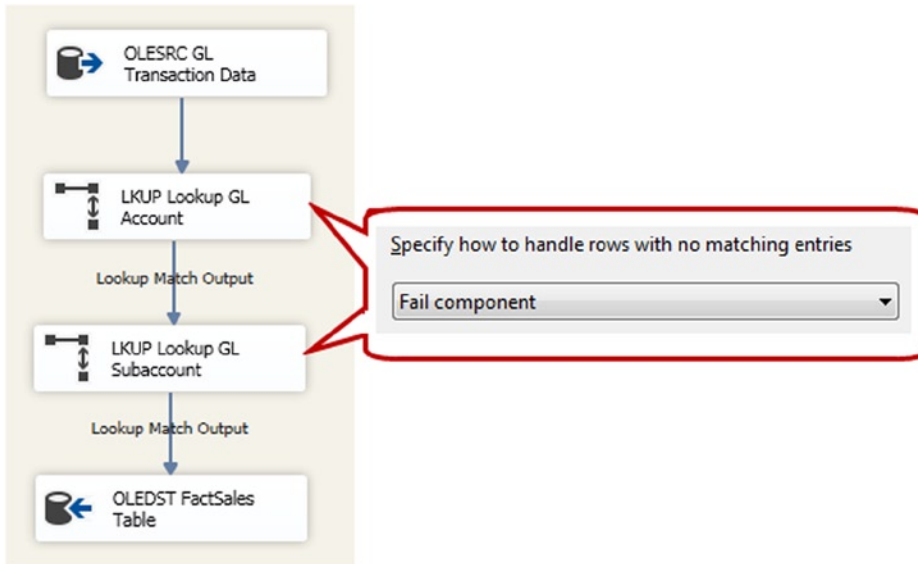


Figure 11-12. Allow the package to fail upon error

It's worth noting here that to properly use this design pattern, you must include some logic to roll back changes due to a partial load. If you encounter a fact row that does not match one of the lookup components, the package will still fail; however, you may commit rows preceding the errored row that have already been sent to the destination to that table before the failure and can cause all the data to be in an inconsistent state.

There are several ways to accomplish this rollback behavior:

SSIS transactions: Integration Services natively has the ability to wrap tasks and containers into transactions and, in theory, will reverse any durable changes to the relational database in the event of an error in the scope of that transaction. In practice, however, I've found that using the SSIS transaction functionality can be challenging even on a good day. Leveraging transactions directly in SSIS requires a lot of things to align at once: all involved systems must support DTC transactions, the DTC service must be running and accessible at each of those endpoints, and all relevant tasks and connections in SSIS must support transactions.

Explicit SQL transactions: This method involves manually creating a transaction in SSIS by executing the T-SQL command to engage a relational transaction on the database engine. Using this method, you essentially create your own transaction container by explicitly declaring the initialization and COMMIT point (or ROLLBACK, in the event of an error) using the Execute SQL task. On the database connection, you'll need to set the `RetainSameConnection` property to `True` to ensure that all operations reuse the same connection and, by extension, the same transaction. Although this method does require some additional work, it's the more straightforward and reliable of the two transactional methods of strategic rollback.

Explicit cleanup: This design pattern involves creating your own environment-specific cleanup routines to reverse the changes due to a failed partial load, and it typically does not engage database transactions for rollback purposes. This method requires the most effort in terms of development and maintenance, but it also allows you the greatest amount of flexibility if you need to selectively undo changes made during a failed package execution.

Unhandled Errors

I'm certain that the gentleman who came up with Murphy's Law was working as a data warehouse ETL developer. Dealing with data from disparate systems is often an ugly process! Although we can defensively code around many common issues, eventually some data anomaly will introduce unexpected errors.

To make sure that any error or other data anomaly does not cause the ETL process to abruptly terminate, it's advisable to build in some safety nets to handle any unexpected errors.

Data Warehouse ETL Workflow

Most of what has been covered in this chapter so far has been core concepts about the loading of data warehouses. I'd like to briefly change gears and touch on the topic of SSIS package design with respect to workflow. Data warehouse ETL systems tend to have a lot of moving parts, and the workload to develop those pieces is often distributed to multiple developers. Owing to a few lessons learned the hard way, I've developed a workflow design pattern of package atomicity.

Dividing Up the Work

I've told this story in a couple of presentations I've done, and it continues to be amusing (to me, anyway) to think about ever having done things this way. The first production SSIS package of any significance that I deployed was created to move a large amount of data from multiple legacy systems into a new SQL Server database. It started off rather innocently; it initially appeared that the ETL logic would be much simpler than what it eventually became, so I wrapped everything into a single package.

In the end, the resulting SSIS package was enormous. There were 30, maybe 40, different data flows (some with multiple sources/destinations and complex transformation logic) and dozens of other helper tasks intermingled. The resulting `.dtsx` file size was about 5MB just for the XML metadata! Needless to say, every time I opened this package in Visual Studio, it would take several minutes to run through all of the validation steps.

This extremely large SSIS package worked fine, and technically there was nothing wrong with the design. Its sheer size did bring to light some challenges that are present in working with large, do-everything packages, and as a result of that experience, I reengineered my methodology for atomic package design.

One Package = One Unit of Work

With respect to data warehouse ETL, I've found that the best solution in most cases is to break apart logical units of work into separate packages in which each package does just one thing. By splitting up the workload, you can avoid a number of potential snags and increase your productivity as an ETL developer. Some of the considerations for smaller SSIS packages include the following:

Less time spent waiting on design-time validation: SQL Server Data Tools has a rich interface that provides, among other things, a near real-time evaluation of potential metadata problems in the SSDT designer. If, for example, a table that is accessed by the SSIS package is changed, the developer will be presented with a warning (or an error, if applicable) in SSDT indicating that metadata accessed by the package has changed. This constant metadata validation is beneficial in that it can help to identify potential problems before they are pushed out for testing. There's also a performance cost associated with this. The length of time required for validation increases as the size of the package increases, so naturally keeping the packages as small as practical will cut down on the amount of time you're drumming your fingers on your desk waiting for validation to complete.

Easier testing and deployment: A single package that loads, say, ten dimensions has a lot of moving parts. When you are developing each of the dimensions within the package, there is no easy way to test just one element of the package (apart from manually running it in the SSDT designer, which isn't a completely realistic test for a package that will eventually be deployed to the server). The only realistic test for such a package is to test the entire package as a server-based execution, which may be overkill if you're only interested in one or two changed properties. Further, it's not uncommon for organizations with formal software testing and promotion procedures to require that the entire thing be retested, not just the new or changed elements. By breaking up operations into smaller units, you can usually make testing and deployment less of a burden because you are only operating on one component at a time.

Distributed development: If you work in an environment where you are the only person developing SSIS packages, this is less of a concern. However, if your shop has multiple ETL developers, those do-everything packages are quite inconvenient. Although it's much easier in SQL Server 2012 than in previous releases to compare differences between versions of the same package file, it's still a mostly manual process. By segmenting the workload into multiple packages, it's much easier to farm out development tasks to multiple people without having to reconcile multiple versions of the same package.

Reusability: It's not uncommon for the same logic to be used more than once in the same ETL execution, or for that same logic to be shared among multiple ETL processes. When you encapsulate these logical units of work into their own packages, it's much easier to share that logic and avoid duplicate development.

It is possible to go overboard here. For example, if, during the course of the ETL execution, you need to drop or disable indexes on 20 tables, you probably don't need to create a package for each index! Break operations up into individual packages, but be realistic about what constitutes a logical unit of work.

These aren't hard and fast rules, but with respect to breaking up ETL work into packages, here are a few design patterns that I've found work well when populating a data warehouse:

- Each dimension has a separate package.
- Each fact table has a separate package.

- Staging operations (if used) each have their own package.
- Functional operations unrelated to data movement (for example, dropping or disabling indexes on tables before loading them with data) are separated as well. You can group some of these operations together in common packages where appropriate; for example, if you truncate tables and drop indexes in a staging database, those operations typically reside in the same package.

Further, it's often a good practice to isolate in separate packages any ETL logic that is significantly different in terms of scope or breadth of data. For example, a historical ETL process that retrieves a large chunk of old data will likely have different performance expectations, error-handling rules, and so on than a more frequently executed package that collects just the most recent incremental data. As such, by creating a separate package structure to address those larger sets of data, you help avoid the issue of trying to force a single package to handle these disparate scenarios.

Conclusion

SQL Server Integration Services isn't just another ETL tool. At its root, it is ideally suited for the unique challenges of data warehouse ETL. This chapter has shown some specific methodologies for how to leverage SSIS against common and realistic DW challenges.



OData Source

The Open Data (OData) protocol is a data access protocol that provides a standardized way to create and consume data APIs via the web. The technology builds upon common protocols and methodologies, such as HTTP, REST, AtomPub, and JSON. The initial versions were defined by Microsoft as open standard, and OData has since been picked up by an OASIS committee with sponsors from many industry giants, such as IBM, Red Hat, and SAP AG. OASIS approved the latest version of the standard (Version 4) in February 2014.

Microsoft has embraced OData as the primary way to expose data APIs for many of its products. SharePoint, Project Online, and many Azure services expose OData APIs for retrieving data, and performing CRUD operations. WCF Data Services (previously known as ADO.NET Data Services) provides a quick and easy way for anyone to develop their own OData services based on an Entity Data Model (EDM) and the ADO.NET Entity Framework. Given the proliferation of the technology, it made sense for Microsoft to add support for OData within SSIS.

This chapter describes the newly added OData Source for SSIS.

■ **Note** The OData Source is not included with the SQL Server 2014 or SQL Server 2012 installation package and must be downloaded separately. The SQL Server 2012 version is available as a download from the Microsoft Download Center: www.microsoft.com/en-us/download/details.aspx?id=42280. The SQL Server 2014 version is part of the SQL Server 2014 feature pack, which can be found at www.microsoft.com/en-us/download/details.aspx?id=42295.

Understanding the OData Protocol

The OData protocol is a fairly extensive protocol and exposes a wide range of capabilities. This section describes the key features of the OData protocol that an SSIS developer should be aware of. If you are interested in learning more about the protocol, the full specification and documentation is available from the www.odata.org web site.

■ **Note** The SSIS OData Source for SQL Server 2012 and 2014 supports version 3 of the OData protocol, and not version 4. More information about the OData protocol can be found at www.odata.org/.

There are two main features of OData from an SSIS perspective: the metadata document and the data you get back from accessing an OData resource.

The metadata document is an EDM description of the OData service that describes the entity sets exposed by the service. The SSIS OData Source uses the metadata document to figure out the schema of the source data. This document is located at a standard location for each service—`http://<base_url>/$metadata`. Table 12-1 shows a mapping of OData and EDM concepts.

Table 12-1. *OData v3 Concepts Mapped to Relational Terms*

OData	Relational	Comments
Entity set	Table or view	<i>Entity sets</i> (also known as collections or feeds) define the objects you'll be receiving in SSIS. When using the Collection mode, the OData Source provides a drop-down box allowing you to select one of the entity sets defined by the service.
Entity	Row of data	Defines the schema for the rows of data within an entity set.
Operation	Stored procedure or function	The OData Source UI does not expose a way to select operations, but it is possible to invoke an operation by specifying its resource path.
Navigation property	Foreign key relationship	Navigation properties define relationships between entities. The OData Source does not support these directly, but you can access subentities by entering their resource path.

At runtime, SSIS retrieves data from the OData service using standard HTTP URLs. That means you can see the same data being returned to SSIS by entering the URL into a web browser, which can be very useful for debugging purposes. The OData Source has two modes: when accessing the service using the *Collection mode*, SSIS will automatically generate the resource URL for you; when using the *Resource Path mode*, you will supply the full URL.

Data Type Mappings

The OData Source will attempt to map the fields of complex types defined within the service's metadata document to SSIS data flow types. When configuring the OData Source, the fields in the entity or resource collection or entity set you reference will be added as external metadata columns for the component. Table 12-2 shows the EDM to SSIS data type mappings. Given the flexibility of the Entity Data Model, you may find that certain complex types will not work with the SSIS OData Source.

Table 12-2. EDM Data Type Mapping to SSIS Types

EDM Type	CLR Type	SSIS Type
Edm.Binary	byte[]	DT_BYTES
Edm.Boolean	bool	DT_BOOL
Edm.DateTime	DateTime	DT_DBTIMESTAMP
Edm.DateTimeOffset	DateTimeOffset	DT_DBTIMESTAMPOFFSET
Edm.Decimal	decimal	DT_NUMERIC
Edm.Double	double	DT_R8
Edm.Guid	Guid	DT_GUID
Edm.Int16	Int16	DT_I2
Edm.Int32	Int32	DT_I4
Edm.Int64	Int64	DT_I8
Edm.SByte	sbyte	DT_I1
Edm.Single	float	DT_R4
Edm.String	string	DT_WSTR
Edm.Time	TimeSpan	DT_DBTIME2

Query Options

OData supports multiple query options that can be included with a request. Query options start with a dollar sign (\$) and are added to the query string portion of the URL, following the resource path. These options are typically used to pass filters to the service or to limit the number of rows or columns in the result set. Some OData services may not support all of the available query options, so sometimes you will need to experiment. Table 12-3 shows some of the more commonly used query options you will use when working with SSIS. You can use the query options in SSIS by specifying them in the Query Options box in the OData Source Editor, or by directly setting the Query property value on the component.

Table 12-3. OData Query Options

Option	Description
\$select	Comma-separated list used to specify which columns should be returned by the service. Equivalent to the SELECT statement in T-SQL.
\$filter	Restricts the result set to rows that match the specified filter expression. Equivalent to the WHERE clause in T-SQL.
\$orderby	Specifies the order in which results are returned from the service. Equivalent to the ORDER BY clause in T-SQL.
\$top	Limits the number of rows returned from the service. Equivalent to the TOP expression in T-SQL.
\$skip	Indicates that the service should ignore the first <i>N</i> rows of the result set, where <i>N</i> is specified by the \$skip argument.
\$format	This option allows you to override the default format of the response. Valid options are json, atom, and xml. SSIS will try to use the json format by default, since it is the most concise and tends to offer the best performance (especially when the service supports JSON with minimal metadata—also known as JSON light).

Configuring the OData Connection Manager

The OData Connection Manager is where you specify the service document URL (i.e., the root URL of the OData service) and authentication information. The connection manager supports three authentication types: Windows, Basic (username and password), and Microsoft Online Services Authentication. You can configure Windows and Basic authentication settings on the main OData Connection Manager Editor page (shown in Figure 12-1). Microsoft Online Services Authentication is meant to be used with SharePoint Online and requires some additional configuration. The process is described in the following section.

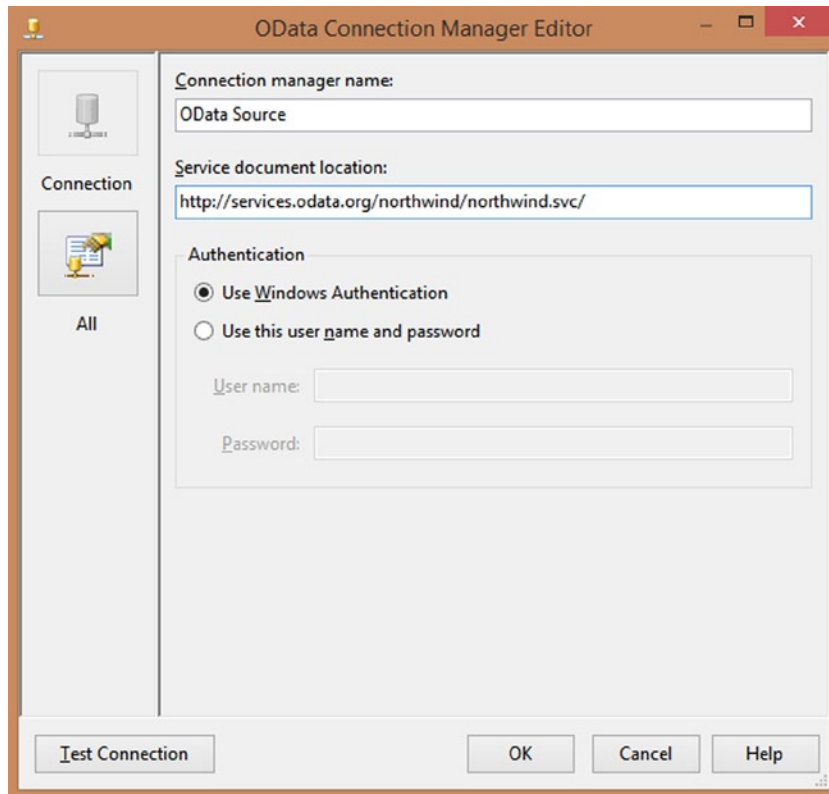


Figure 12-1. The OData Connection Manager Editor

Enabling Microsoft Online Services Authentication

SharePoint provides OData endpoints for the objects that it stores, such as its lists and documents, which can now be accessed in SSIS through the OData Source. Although you can configure different types of authentication for the on-premises versions of SharePoint, SharePoint Online (and other Microsoft services that are a part of Office 365, like Project Online) uses a special type of authentication. The OData Connection Manager supports these online services, but you may need to install an additional SharePoint SDK component to make it work.

You can follow these steps to connect to SharePoint Online:

1. Click the All button in the OData Connection Manager Editor (shown in Figure 12-1) to bring up the full property page.
2. Under the Security group, change the Integrated Security property from SSPI to False.
3. If the Microsoft Online Services Authentication property stays disabled, then you will need to install the SharePoint Service 2013 Client Components SDK from the Microsoft Download Center (search for the SDK, or follow this link: <http://www.microsoft.com/en-us/download/details.aspx?id=35585>). After installing the SDK, save the SSIS package and restart SQL Server Data Tools. You should now be able to change the Microsoft Online Services Authentication property to True (as shown in Figure 12-2).

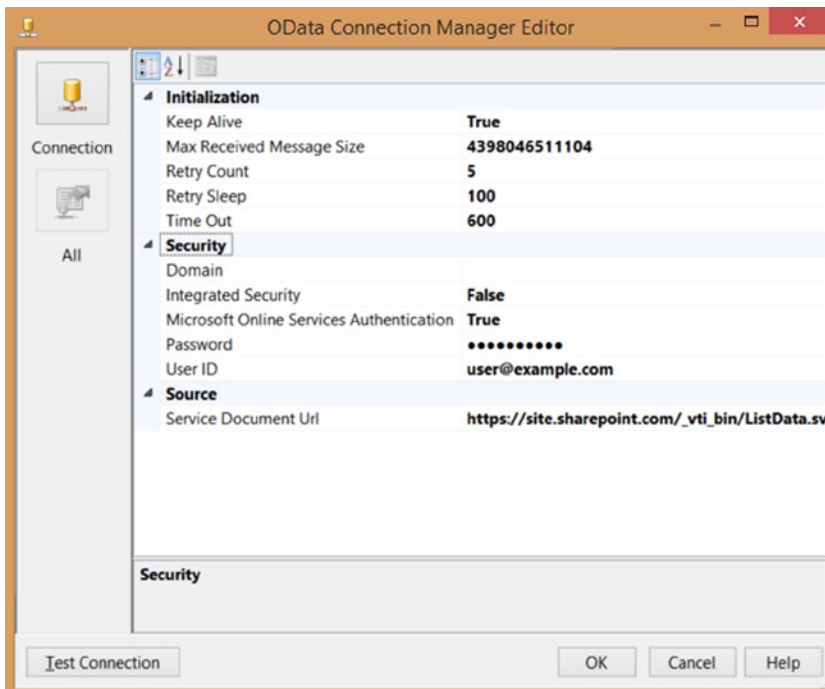


Figure 12-2. The OData Connection Manager Editor showing all properties

4. Enter the user ID and password you use when connecting to SharePoint Online. You may also need to provide a domain value, depending on how your site administrator has configured your Active Directory integration with Office 365.
5. You should now be able to successfully connect to your SharePoint Online site. Enter the Service Document URL (e.g., `https://<mysite>.sharepoint.com/_vti_bin/ListData.svc`) and click the Test Connection button.

■ **Note** More information about the OData endpoints offered by SharePoint Online can be found in Books Online at <http://msdn.microsoft.com/en-us/library/ff521587.aspx>.

Configuring the Source Component

You can bring up the OData Source Editor by double-clicking on the component in the data flow. The editor is similar to what you'd expect from other SSIS data flow components and is shown in Figure 12-3.

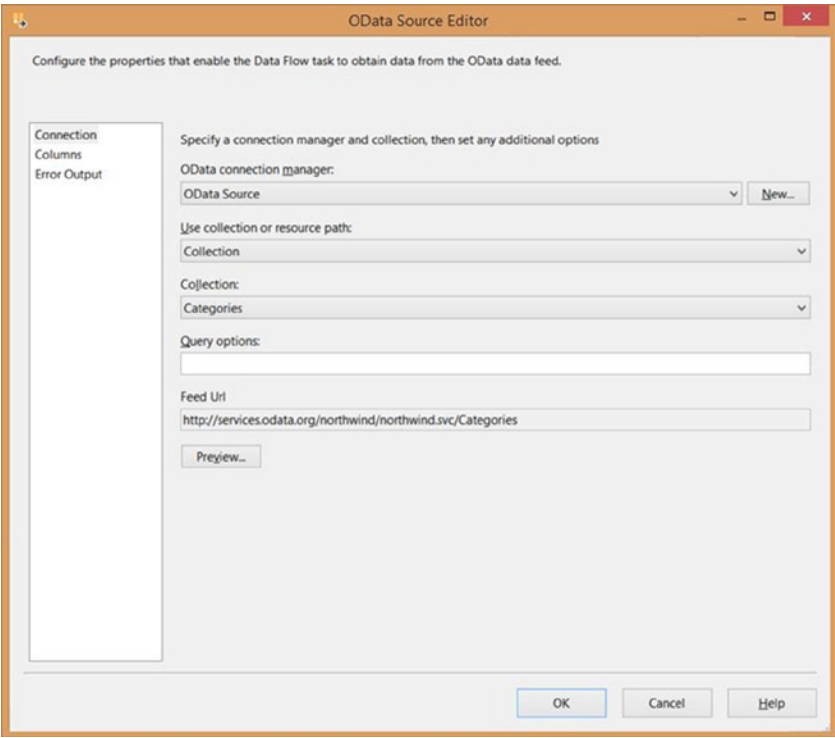


Figure 12-3. The OData Source Editor

A description of the fields you can configure on the Connection page can be found in Table 12-4. The table lists the field name as it is described in the editor UI, as well as the matching property name when setting the value through the Advanced Editor (shown later in Figure 12-6) or the Properties window.

Table 12-4. OData Source Fields

Field	Property	Description
OData Connection Manager	N/A	The connection manager you will use for this Source component. This property must be set before you can set the other values. Clicking the New button will bring up a dialog that lets you create a new connection manager.
Use Collection or Resource Path	UseResourcePath	This field determines whether you are accessing a specific collection from the OData service or providing your own resource path. This is set to Collection by default. More information about these access modes can be found later in this section.

(continued)

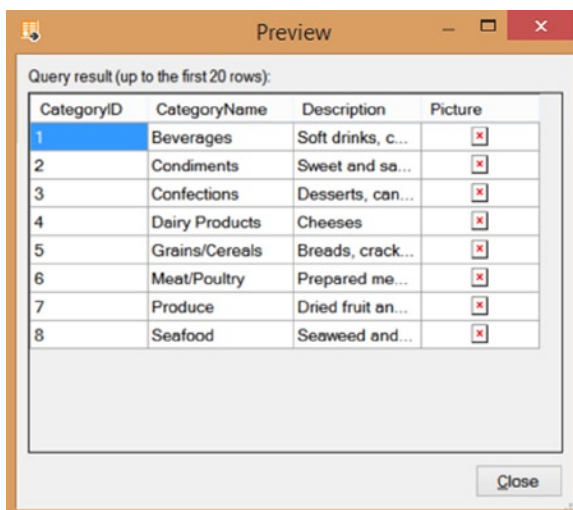
Table 12-4. (continued)

Field	Property	Description
Collection	CollectionName	This box lets you select a specific collection from the OData feed.
Resource Path	ResourcePath	When using the Resource Path access mode, this field lets you enter a full path of the specific OData resource you want to access.
Query Options	Query	This field lets you append different query options, such as \$top or \$filter, to your request. Table 12-3 lists the set of query options you'd commonly include in a request. Note that the values should be URL encoded—the OData Source does not automatically encode these values for you.
Feed Url	N/A	This is a read-only field that shows the URL that the OData Source will use to access the server. It is built based on the values you set for the other fields in this dialog. This field can be useful for debugging purposes.

There are two main ways to access the OData service using the OData Source: by selecting a collection, or by configuring a resource path.

Accessing the service using a collection is the default behavior of the OData Source. You will typically use this option when you want to read one of the top level entity sets exposed by the service. The Collection field will give you a drop-down list of the available entity sets, similar to how the OLE DB Source lists the set of available tables and views.

You would use the resource path when you are doing more than just accessing a single entity set. For example, you want to run an action or operation defined by the service, or you want to access a subentity through a navigation property. In these cases, you would provide the full URL in the Resource Path field. Note that you are expected to apply any URL encoding to the value yourself—the component does not automatically do this for you. Also note that the OData Source will not do any special validation for these values—it will simply try accessing the URL at runtime and rely on the service's \$metadata document to determine how to interpret the results. Using the Preview button on the OData Source Editor dialog (shown in Figure 12-4) is very helpful in these cases because it will let you quickly validate whether the request will succeed and that you are getting the data you want.

**Figure 12-4.** The Preview dialog from the OData Source Editor

After configuring the collection or resource path value, you can select the columns you want to retrieve for your data flow by clicking on the Columns page (shown in Figure 12-5). This dialog uses the standard SSIS external column dialog and should be familiar to SSIS developers. Note that unselected columns on this page won't be added to your data flow, but they will still be pulled down by the request to the service. If you are only selecting a few columns from a given feed, you might want to limit the columns upfront using the \$select query option. More information about query options can be found in Table 12-3.

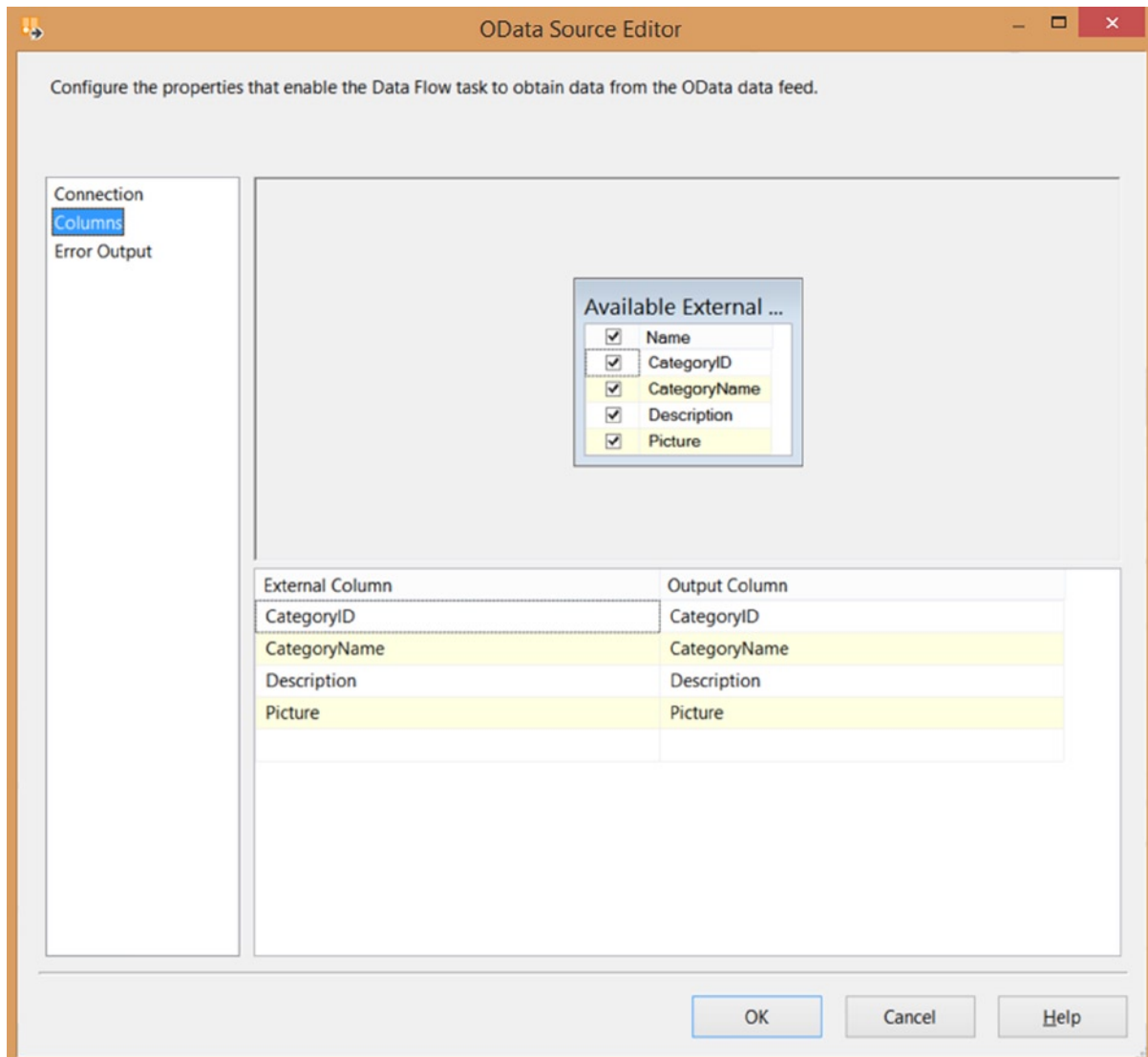


Figure 12-5. The OData Source Editor's Columns page

Overriding Data Types

The OData Source allows you to manually override data types on output columns for certain types. This comes in handy when you are working with OData services that don't specify max length values for String and Binary fields. SSIS will map to DT_NTEXT and DT_IMAGE types (respectively), which can have negative implications to performance. You can change the default data type for a column by using the Advanced Editor (as shown in Figure 12-6).

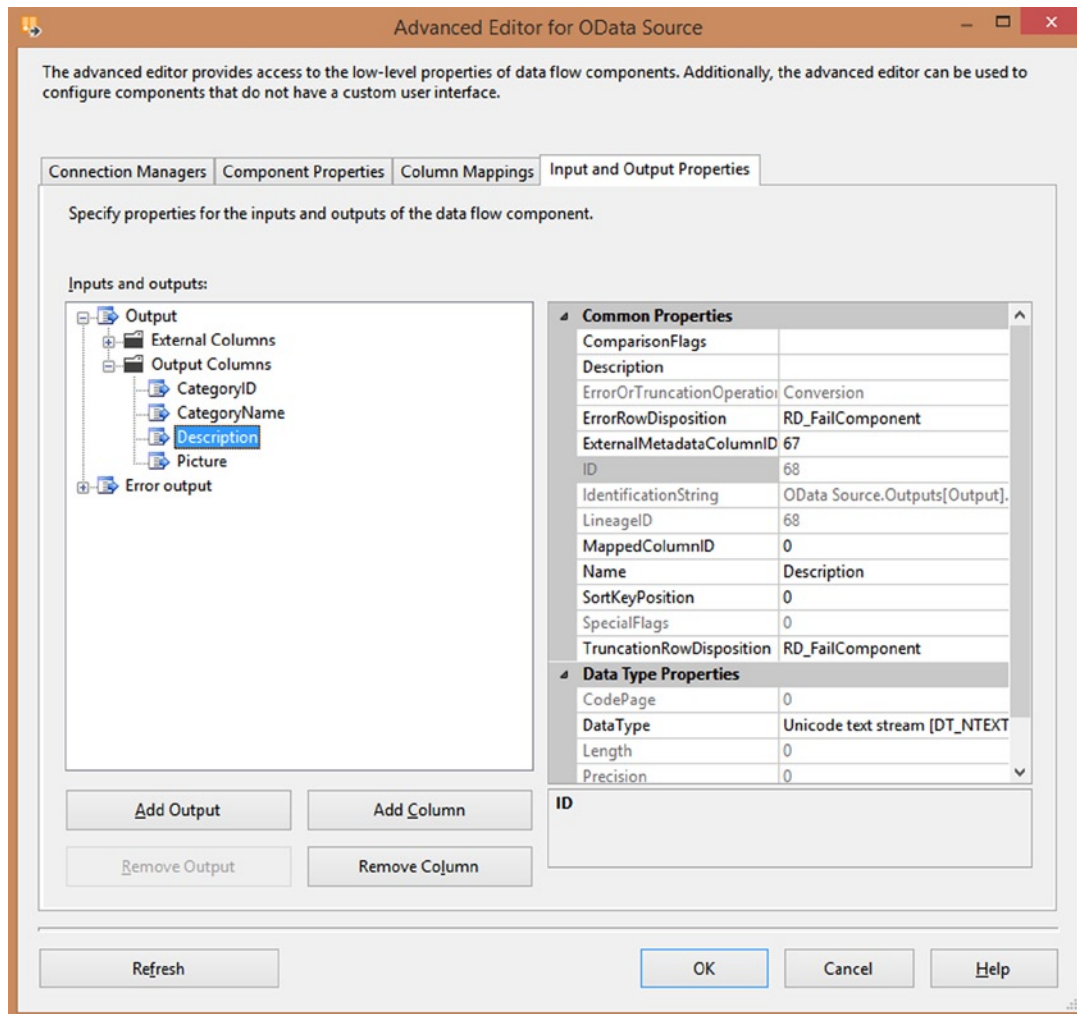


Figure 12-6. The Advanced Editor for the OData Source

Here's how to modify the default data types:

1. Add an OData Source to your data flow and configure its connection manager and the data you want to access.
2. Click OK to save the settings.
3. Right-click on the OData Source component on the data flow's design surface and select Show Advanced Editor.
4. Select the Input and Output Properties tab.
5. Expand the Output folder.
6. Expand the Output Columns folder.
7. Select the column you want to modify.
8. Select the DataType property.
9. Change the value to the new SSIS data type and the Length value, if appropriate, for the type.

You will only be able to change the DataType property for certain types. Table 12-5 contains a list of data types that are interchangeable. Changing the value for an unsupported data type will result in an error message (COMException 0xC020837D).

Table 12-5. *Data Types That Can Be Changed*

Data Type	Compatible Type
DT_NTEXT	DT_WSTR
DT_IMAGE	DT_BYTES
DT_DBTIMESTAMP2	DT_DBTIMESTAMP

Conclusion

This chapter described the new OData Source component in SSIS. OData has seen a steady increase in adoption over the past few years, and the OData Source provides an out-of-the-box solution for reading data from a variety of sources, such as SharePoint, and the Windows Azure Storage APIs.



Slowly Changing Dimensions

Processing slowly changing dimensions (SCDs) is a common ETL operation when you are dealing with data warehouses. The SSIS data flow has a SCD transform, which provides a wizard that outputs a set of transforms you need to handle the multiple steps of processing SCDs. Although the built-in SCD transform can be useful, it is not ideal for all data loading scenarios. This chapter describes how to make the most of the SCD transform and provides a couple of alternative patterns you can use.

■ **Note** There are many different types of SCD, but this chapter will focus on the two most common types: Type 1 and Type 2. For more information about the different types of SCDs, see the Wikipedia entry at http://en.wikipedia.org/wiki/Slowly_changing_dimensions.

The Slowly Changing Dimension Transform

To best understand the SCD transform, let's consider two key scenarios it was designed for:

A small number of change rows: You are performing change data capture (CDC) at the source, or as close to the source as possible. Unless you are dealing with a very active dimension, most SCD processing batches will contain a small number of rows.

Large dimension: You are working against large dimensions but are only processing a small number of change rows. You will want to avoid operations that cause full table scans of your dimension.

Because of these target scenarios, the SCD transform does not cache the existing dimension data (like a Lookup transform does) and performs all of its comparisons row by row against the destination table. Although this allows the transform to avoid full scans of the destination dimension and reduces memory usage, it does affect performance when you are processing a large number of rows. If your scenario does not match one of these, you might want to consider using one of the other patterns in this chapter. If it does match, or if you prefer using out-of-the-box components over third-party solutions (or you would just like to avoid hand-crafted SQL statements required for the Merge pattern), consider applying the optimizations listed at the end of this pattern.

Running the Wizard

Unlike other SSIS data flow components, when you drop the SCD transform onto the design surface in Business Intelligence Development Studio (BIDS), a wizard pops up and walks you through the steps of setting up your SCD processing.

The first page of the wizard (Figure 13-1) allows you to select the dimension you'll be updating and the column or columns that make up the business key (also known as the natural key).

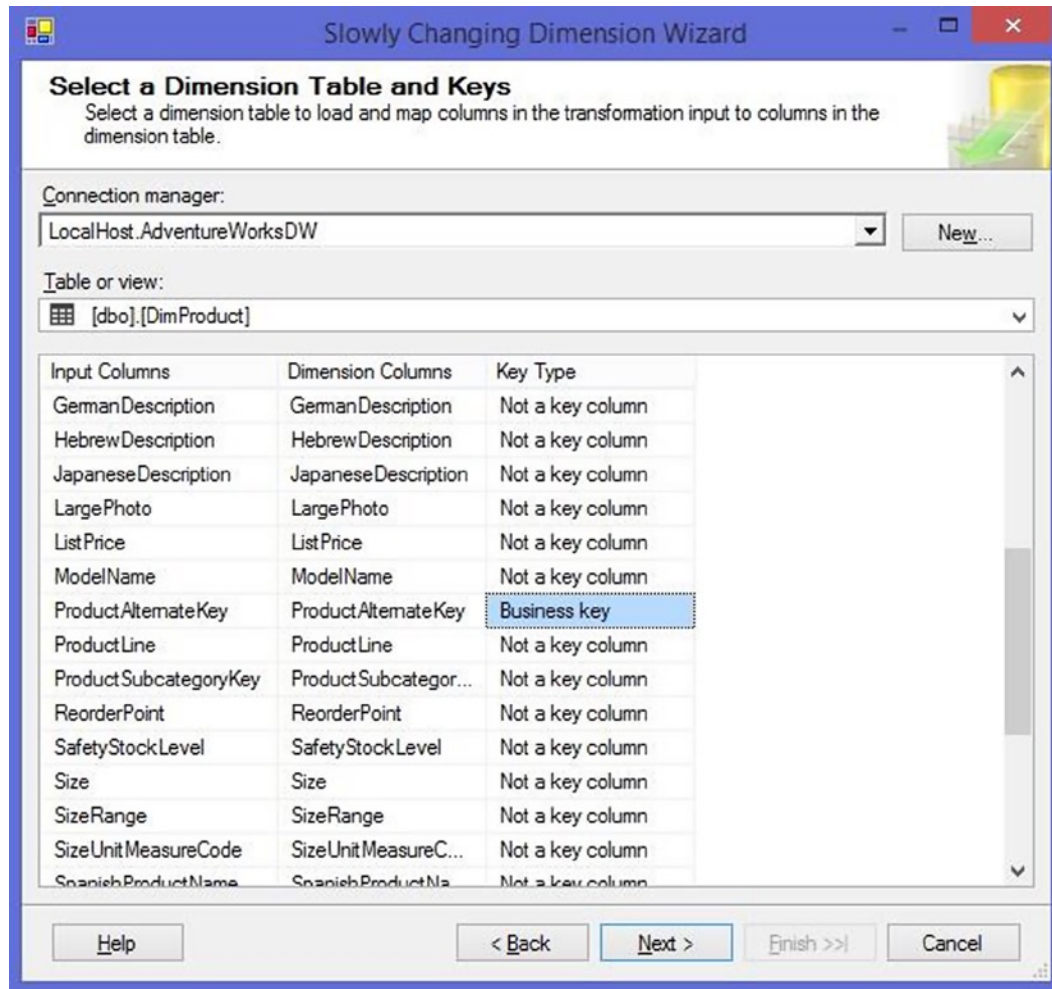


Figure 13-1. Selecting the dimension table and keys in the Slowly Changing Dimension Wizard

On the next page (Figure 13-2), you specify the columns that you'll be processing and determine how you'd like the wizard to treat them. You have the three choices (as shown in Table 13-1).

Slowly Changing Dimension Columns
Manage the changes to column data in your slowly changing dimensions by setting the change type for dimension columns.

Fixed Attribute
Select this type when the value in a column should not change. Changes are treated as errors.

Changing Attribute
Select this type when changed values should overwrite existing values. This is a Type 1 change.

Historical Attribute
Select this type when changes in column values are saved in new records. Previous values are saved in records marked as outdated. This is a Type 2 change.

Select a change type for slowly changing dimension columns:

Dimension Columns	Change Type
ArabicDescription	Changing attribute
ChineseDescription	Changing attribute
Class	Fixed attribute
Color	Changing attribute
DaysToManufacture	Historical attribute
DealerPrice	Historical attribute
EnglishDescription	Changing attribute
EnglishProductName	Historical attribute
FinishedGoodsFlag	Fixed attribute
FrenchDescription	Changing attribute
FrenchProductName	Historical attribute
GermanDescription	Changing attribute
HebrewDescription	Changing attribute
JapaneseDescription	Changing attribute
LargePhoto	Changing attribute

Remove

Help < Back Next > Finish >>| Cancel

Figure 13-2. Selecting the dimension table and keys in the Slowly Changing Dimension Wizard

Table 13-1. *Column Change Types*

Change Type	Dimension Type	When to Use
Fixed attribute	—	Fixed attributes are columns that should not change or that require special handling when you make changes. By default, a change on one of these columns is treated as an error.
Changing attribute	Type 1	When a change is made to a changing attribute column, existing records are updated to reflect the new value. These are typically columns that aren't used as part of business logic or time-sensitive reporting queries, such as a product description.
Historical attribute	Type 2	Historical attributes are columns for which you need to maintain history. These are frequently numeric columns that are used in time-sensitive reporting queries, such as a Sales Price, or Weight.

On this page, you should not map columns that will not be updated as part of your SCD processing, such as foreign keys to other dimension tables or columns related to the tracking of historical changes—for example, Start and End Date columns, an expiry flag, or the surrogate key. The SCD transform does not support Large Object (LOB) columns (columns that would be treated as DT_IMAGE, DT_TEXT, and DT_NTEXT types in the SSIS data flow), so these columns should be handled separately and also should not be mapped here.

The next pages of the wizard allow you to configure options for how you'd like to handle fixed attributes (Figure 13-3), as well as Type 1 and Type 2 changes (Figure 13-4). When dealing with historical attributes, the wizard knows how to generate the logic needed to update the dimension in two different ways: by using a single column or by using Start and End Date columns to indicate whether the record is current or expired. If your table is using a combination of these, or some other method of tracking the current record, you will need to update the generated transforms to contain this logic.

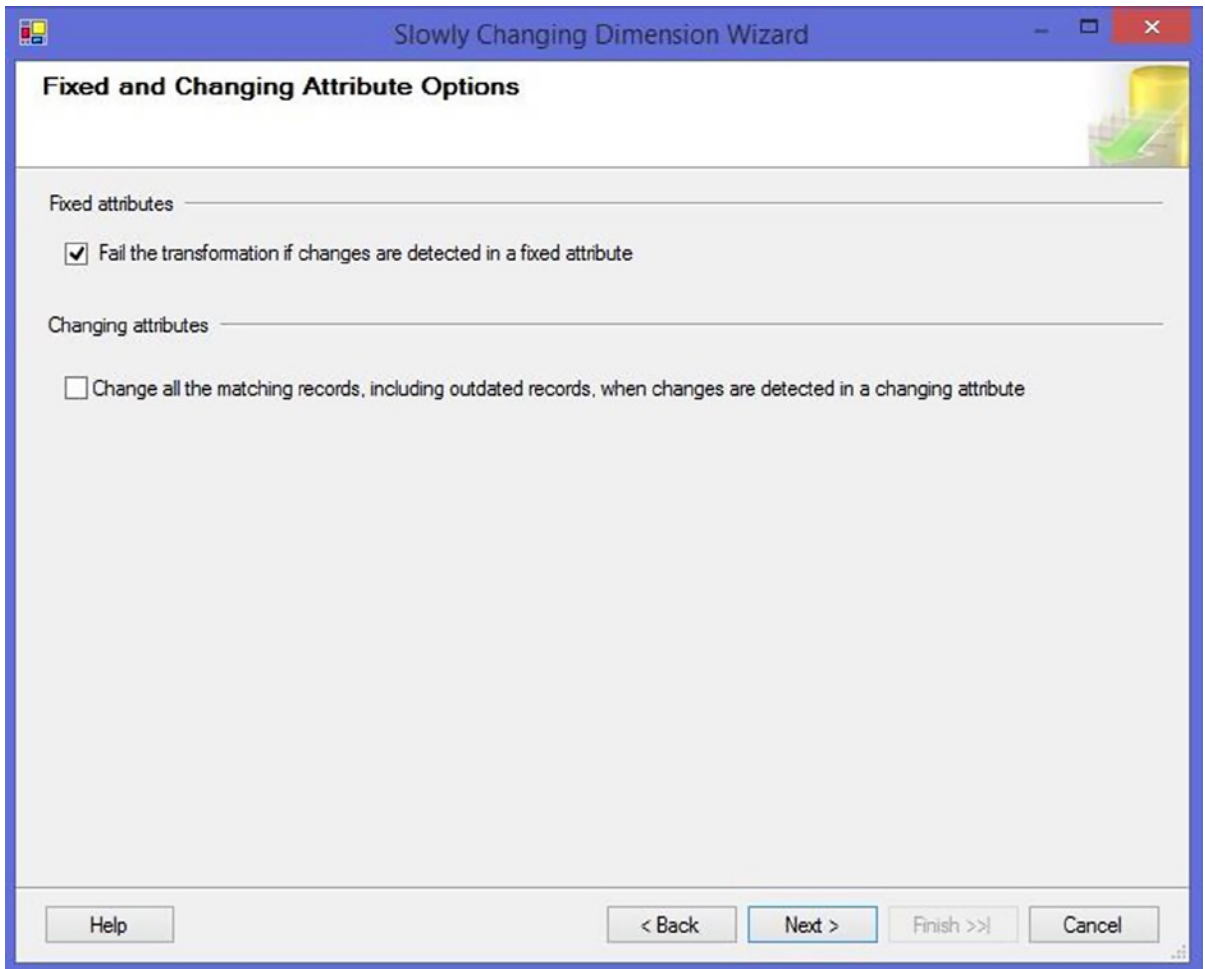


Figure 13-3. Fixed and changing attribute options

Slowly Changing Dimension Wizard

Historical Attribute Options
You can record historical attributes using a single column or start and end date columns.

☐ Use a single column to show current and expired records

Column to indicate current record:

Value when current:

Expiration value:

☒ Use start and end dates to identify current and expired records

Start date column:

End date column:

Variable to set date values:

Help < Back Next > Finish >> Cancel

Figure 13-4. Historical attribute options

The final page of the wizard (Figure 13-5) lets you enable support for inferred members. An inferred member is created with minimal information—typically just the business and surrogate keys. It's expected that the remaining fields will be populated in subsequent loading of the dimension data. Although the wizard enables inferred member support by default, for most forms of SCD processing, you will not need it.

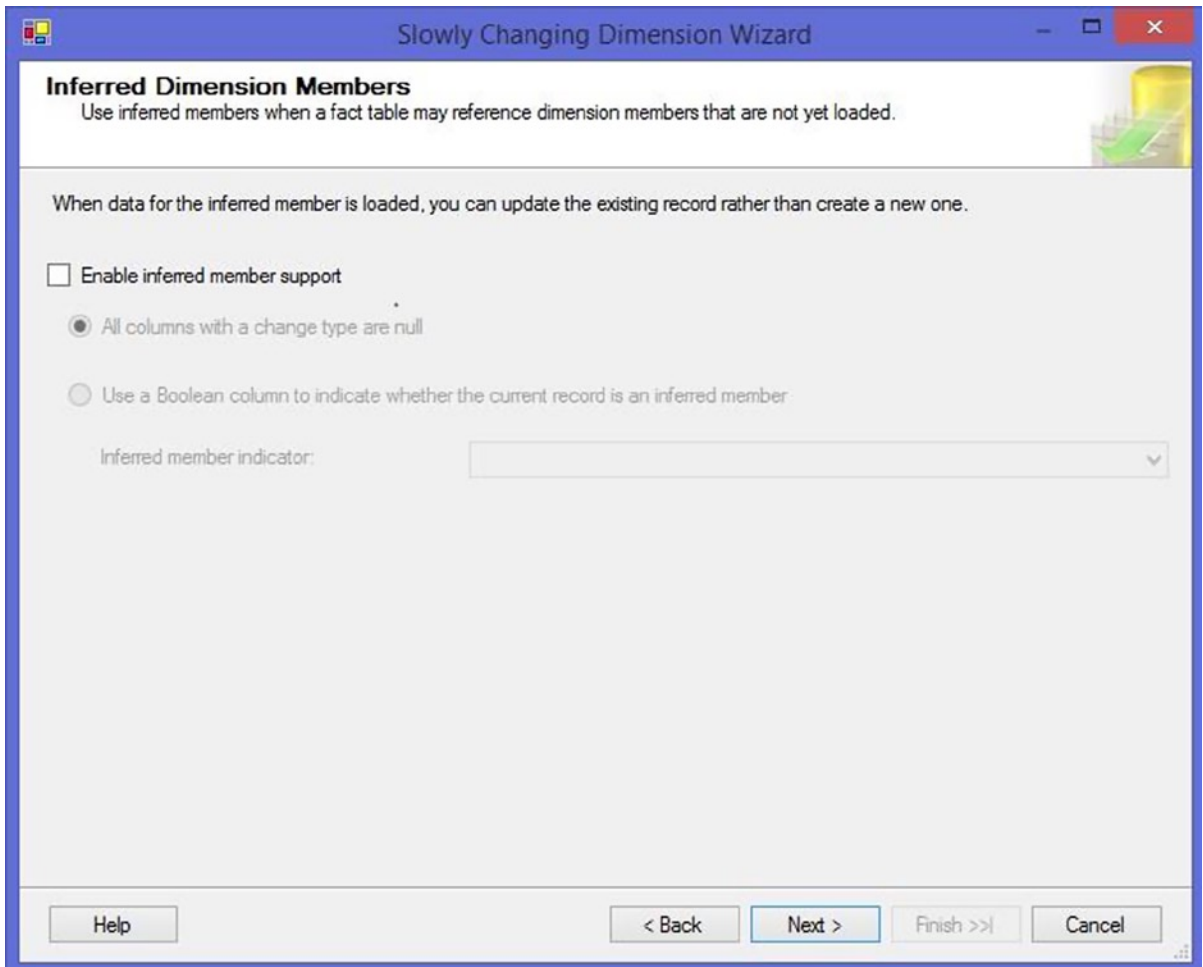


Figure 13-5. *Inferred dimension members*

Using the Transformations

When the wizard completes, it will output a number of different data flow components in addition to the main Slowly Changing Dimension component (Figure 13-6). The main component checks incoming data against the destination table and sends incoming rows down one of its outputs if the record is new or modified. Records without any changes are ignored. The components connected to these outputs will be configured according to the options you selected in the wizard dialogs.

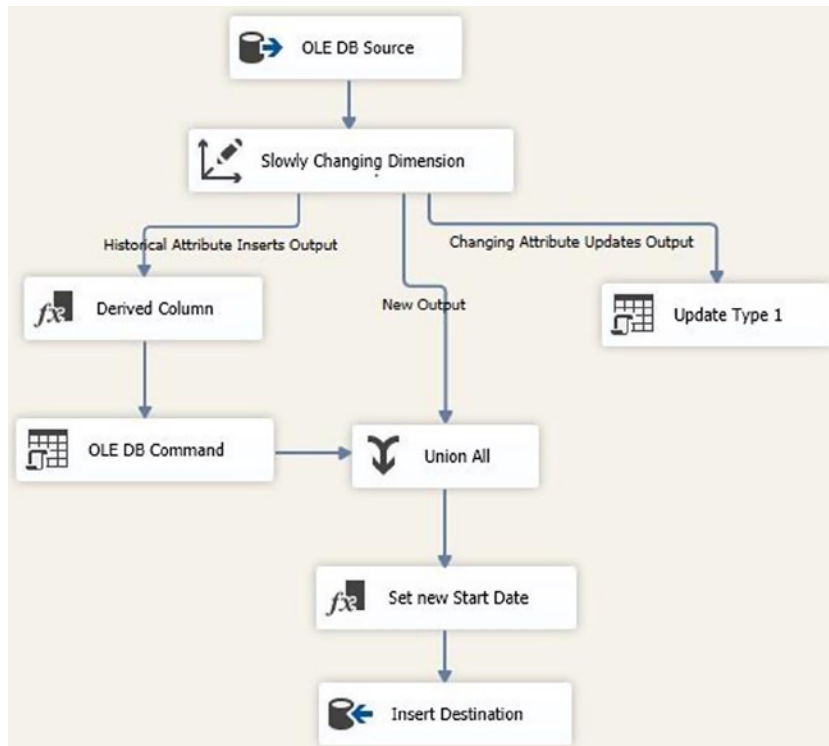


Figure 13-6. Wizard output for Type 1 and Type 2 changes, and no inferred member support

You can further customize the SCD processing logic by modifying these components. Double-clicking the main Slowly Changing Dimension transform relaunches the wizard. The wizard remembers your settings from the previous run; however, it will overwrite any changes or customizations you’ve made to the existing transforms. This includes any layout and resizing changes you might have performed.

■ **Note** When you re-run the Slowly Changing Dimension Wizard, the default options selected in the UI are not inferred from the components. Instead, they are persisted as part of the package in <designTime> elements. If you have a deployment process that removes package layout information, note that you will also lose your choices in the wizard.

Optimizing Performance

The components output from the Slowly Changing Dimension Wizard are not configured for optimal performance. By changing some settings and moving to a set-based pattern, you can drastically improve the performance of your SCD processing.

The Slowly Changing Dimension Transform

The main transform does not cache any row results from the reference dimension, so every incoming row results in a query against the database. By default, the wizard will open a new connection to the database on each query. For a gain in performance (as well as lower resource usage), you can set the `RetainSameConnection` property of the connection manager used by the wizard to `True` so that the same connection is reused on each query.

OLE DB Command Transforms

The wizard will output two (or three, if you're processing inferred members) OLE DB Command transforms. These transforms perform row-by-row updates, which greatly degrade performance. You will get a big performance boost by placing these rows in staging tables and performing the updates in a single batch once the data flow completes.

OLE DB Destination

Since the main Slowly Changing Dimension transform and the destination use the same connection manager, the destination component will have the Fast Load option disabled by default to avoid deadlocking your data flow. If you are processing a small number of rows (for example, a single data flow buffer's worth of data), you can enable Fast Load on the destination component for an immediate performance gain. To avoid deadlocking issues when processing a larger number of rows, consider using a staging table. Bulk load the data into a temporary staging table and update the final destination once the data flow is complete using an `INSERT INTO ... SELECT` statement.

Third-Party SCD Components

A couple of popular third-party alternatives to the SCD transform are available. Both have similar architectures and usage patterns but offer different capabilities.

- **The Table Difference component is available through CozyRoc.com:** This transform takes in the source and destination tables as inputs and does row-by-row comparisons in memory. It has three outputs—New, Updated, and Deleted. It can also be used to do general-purpose table comparisons in addition to SCD processing.

■ **Note** For more information about the Table Difference component, please see the CozyRoc web page at www.cozyroc.com/ssis/table-difference.

- **The Dimension Merge SCD component is available through PragmaticWorks.com:** This component was designed to handle dimension loading as per the Kimball method. Like the Table Difference component, it takes in the source and destination dimension tables and does the comparisons in memory. Also like the Table Difference component, it does not modify the destination table directly. It will apply row updates in memory and provides a number of outputs to feed your destination tables.

■ **Note** For more information about the Dimension Merge SCD component, please see the Pragmatic Works web page at <http://pragmaticworks.com/Products/Business-Intelligence/TaskFactory/Features.aspx#TSDimensionMergeSCD>.

The main draw of these components is their performance. Since the transforms take both source and destination tables into memory, they are able to do fast in-memory comparisons, without multiple queries to the destination server. They also provide additional functionality over the SCD transform, such as detecting deleted rows, and can be easier to maintain because all of the logic is contained within a single component.

However, bringing in both the source and destination dimension tables means that you're doing a full table scan of the destination (and typically the source, as well). Since the data flow does not end until all sources are done reading their rows, the entire destination dimension will be read, even if you are only processing a small number of changed source rows. Although the third-party components will perform well in many cases, you should consider if they will be ideal for your scenario.

Merge Pattern

SQL Server 2008 introduced support for the T-SQL MERGE statement. This statement will perform insert, update, and delete operations on a destination table based on the results of a join with a source table. It is very efficient and provides a good alternative for SCD processing.

■ **Note** For more information about MERGE, please see the Books Online entry “Using MERGE in Integration Services Packages” at <http://technet.microsoft.com/en-us/library/cc280522.aspx>.

There are three steps to using MERGE from within SSIS:

1. Stage the data in a data flow.
2. Optimize the staging table (optional).
3. Run the MERGE statement(s) using an Execute SQL task.

MERGE allows a single statement to be run when it detects a row has been updated (“matched” using the MERGE terminology) and when a row is new (“not matched”). Since Type 1 and Type 2 changes require different types of processing, we'll use two MERGE statements to complete the SCD processing (as shown in Figure 13-7).

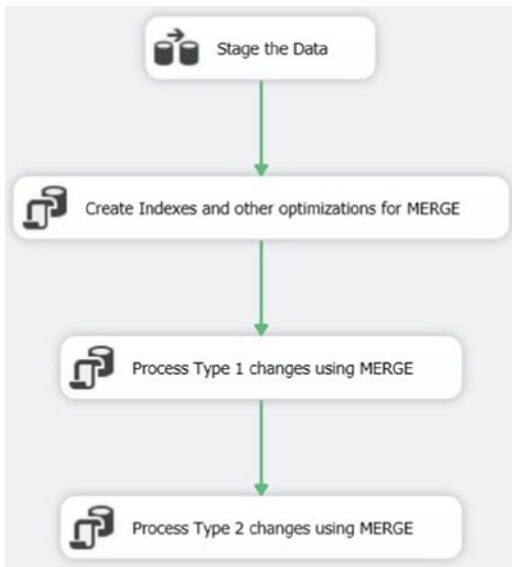


Figure 13-7. Control flow for the Merge pattern

Handling Type 1 Changes

Listing 13-1 shows the first MERGE statement we'll run to update all of our Type 1 columns in the destination table. The ON () section specifies the keys that we'll be matching on (in this case, the business key for the table). In the WHEN MATCHED section, we include DEST.EndDate is NULL to ensure that we are only updating the current record (this is optional—in many cases, you do want to update all records and not just the current one). The THEN UPDATE section contains the list of our Type 1 columns that we want to update.

Listing 13-1. MERGE Statement for Type 1 Columns

```

MERGE INTO [DimProduct] AS DEST
USING [Staging] AS SRC
ON (
    DEST.ProductAlternateKey = SRC.ProductAlternateKey
)
WHEN MATCHED AND DEST.EndDate is NULL -- update the current record
THEN UPDATE SET
    DEST.[ArabicDescription] = SRC.ArabicDescription
    ,DEST.[ChineseDescription] = SRC.ChineseDescription
    ,DEST.[EnglishDescription] = SRC.EnglishDescription
    ,DEST.[FrenchDescription] = SRC.FrenchDescription
    ,DEST.[GermanDescription] = SRC.GermanDescription
    ,DEST.[HebrewDescription] = SRC.HebrewDescription
    ,DEST.[JapaneseDescription] = SRC.JapaneseDescription
    ,DEST.[ThaiDescription] = SRC.ThaiDescription
    ,DEST.[TurkishDescription] = SRC.TurkishDescription
    ,DEST.[ReorderPoint] = SRC.ReorderPoint
    ,DEST.[SafetyStockLevel] = SRC.SafetyStockLevel
;
  
```

Handling Type 2 Changes

Since the MERGE statement allows a single statement for each action, updating Type 2 columns is a little more challenging. Remember, for Type 2 changes, you need to perform two operations: 1) mark the current record as expired, and 2) insert the new record as current. To accomplish this, you'll use the MERGE statement inside of a FROM clause and use its OUTPUT to feed an INSERT INTO statement (as shown in Listing 13-2).

Listing 13-2. MERGE Statement for Type 2 Columns

```
INSERT INTO [DimProduct] ([ProductAlternateKey],[ListPrice],[EnglishDescription],[StartDate])
SELECT [ProductAlternateKey],[ListPrice],[EnglishDescription],[StartDate]
FROM (
    MERGE INTO [DimProduct] AS FACT
    USING [Staging] AS SRC
    ON ( FACT.ProductAlternateKey = SRC.ProductAlternateKey )
    WHEN NOT MATCHED THEN
    INSERT VALUES (
        SRC.ProductAlternateKey
        ,SRC.ListPrice
        ,SRC.EnglishDescription
        ,GETDATE()      -- StartDate
        ,NULL           -- EndDate
    )
    WHEN MATCHED AND FACT.EndDate is NULL
    THEN UPDATE SET FACT.EndDate = GETDATE()
    OUTPUT $Action Action_Out
        ,SRC.ProductAlternateKey
        ,SRC.ListPrice
        ,SRC.EnglishDescription
        ,GETDATE() StartDate
) AS MERGE_OUT
WHERE MERGE_OUT.Action_Out = 'UPDATE'
```

Conclusion

There are many ways to process SCDs in SSIS. Although the built-in SCD transform can get you up and running quickly, it may not perform as well as the alternatives. You may prefer using the Merge pattern due to its overall performance, but having to maintain the SQL statements may be an inhibitor in the long run. If you prefer a visual designer experience, consider trying one of the third-party component options.

Table 13-2 summarizes the advantages and disadvantages described in this chapter.

Table 13-2. *Slowly Changing Dimension Processing Patterns*

Pattern	Use For
Slowly Changing Dimension transform	• Quick prototyping • Processing a small number of rows • Very large dimensions
Third-party components	• Full or historical dimension loads • Small-medium sized dimensions • Non-SQL Server destinations
Merge pattern	• Best overall performance • Cases in which you don't mind hand-crafting SQL statements



Loading the Cloud

It is 2014 and cloud technology is becoming ubiquitous. As more applications are hosted in various cloud service providers, the desire to locate their associated data in the cloud increases. Thanks to forethought and good engineering, Microsoft SQL Server is well-positioned to assist. The user experience when interacting with Microsoft Azure SQL Database (MASD) databases is nearly identical to interacting with local servers or servers on the enterprise network. Make no mistake, this is by design—and it is good design.

In this chapter, we will consider SSIS design patterns used to integrate data in the cloud. These patterns are useful when connecting to any repository that shares cloud technology characteristics. Because interacting with data in the cloud is similar to interacting with data that are more local, the patterns aren't revolutionary. "So why write a chapter about loading the cloud?" Excellent question.

First, the cloud is here to stay—the djinni will not fit back into the bottle. The more we, as data professionals, learn about using the cloud, the better. Second, the cloud offers interesting challenges to data integration; these are challenges to be addressed and solved by the data integration developer. Loading the cloud isn't just about interacting with buzz-worthy technology. It is an opportunity to design good architecture.

Interacting with the Cloud

For the purposes of this chapter, *the cloud* will refer to containers of data or data repositories that

- Reside off-enterprise-premises
- Are hosted by a third party
- Are outside of the physical enterprise domain

I understand these points are subject to debate. I will not debate them here. This definition will likely not survive the years following this writing (2012) and revision (2014). And even now, there is ambiguity and a blurring of lines between what is and is not considered "in the cloud."

For demonstration, I am using data collected from my local weather station in Farmville, Virginia. The weather data are exposed and available at AndyWeather.com. AndyWeather.com is hosted by a large hosting company that provides remotely-accessible SQL Server database connectivity. As such, the data are stored in the cloud (according to my definition).

I also host weather data using a Microsoft Azure SQL Database: the same data, stored in a different location. "Why?" The simple answer: fault tolerance. Fault tolerance is the same reason DBAs perform database backups and test database restores. The difference between a technician and an engineer—or a developer and an architect—is that a technician builds systems that succeed whereas engineers build systems that don't fail. It's about mindset. Technicians get it working and stop there. Engineers consider many ways the system can fail and try to fix these things before they break.

Incremental Loads to Azure SQL Database

An *incremental load* is one in which only new or updated rows (and sometimes deleted) rows are loaded or changed. Incremental loads can be contrasted with the *truncate-and-load* pattern, where the existing data in the destination is deleted and all data is reloaded from the source. Sometimes truncate-and-load is most efficient. As data scale—especially at the destination—truncate-and-load performance often suffers. How do you know which will perform best? Test, measure, rinse, and repeat.

One benefit to truncate-and-load is simplicity. It is difficult to mess up a simple mechanism. Incremental loads introduce complexity, and change detection is the first place complexity enters the solution.

Change Detection

Change detection is functionality designed to separate rows that have never been sent from the source (New Rows) to the destination from rows that have been sent. Change detection also separates source rows that exist in the destination into rows that have changed in the source (Changed Rows) and rows that remain unchanged since they were last loaded or updated in the destination (Unchanged Rows). Change detection can also encompass rows that have been deleted from the destination that need to be deleted (or “soft-deleted”) from the source. We will ignore the Deleted Rows use case in this chapter.

We will consider using change detection to determine and discriminate between Unchanged Rows, Changed Rows, and New Rows. The block diagram for an incremental load pattern is shown in Figure 14-1.

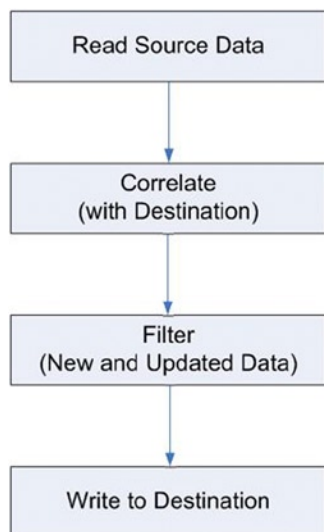


Figure 14-1. Incremental load block diagram

New Rows (Only)

You would think detecting new rows would be simple, regardless of the technology behind the destination database. You would be right. But when loading MASD, there is an economic consideration. As I write this in 2011 (and revise it in 2014), uploads are free. As a data provider, you are charged when your data is consumed. How does this fact impact your incremental load?

Here is where good data integration architecture and design comes in. Part of the job of the data integration architect is to know your data. This goes beyond knowing there is a column in the CleanTemperature table that contains the Average Dew Point for a given hour. Knowing your data means that you understand how and when it changes—or even if it changes. Certain types of data, like weather data, do not get updated after they are captured and recorded.

If you have read Tim Mitchell’s description of a typical incremental load pattern in Chapter 11, you will note a Lookup transform configured to connect to the destination and read data. In MASD, you will pay for that data read. There are almost 30,000 rows present in my MASD AndyWeather database. If I load all the rows from my source into a Data Flow task and use a Lookup transform to “join” between my source and MASD, I pay for reading rows that haven’t changed. What’s more, I know they *will never change*. Since weather data is never updated once recorded, there will simply be no Changed Rows.

Each hour, a few rows of new data are generated for each subject area in the AndyWeather database. If I use a Lookup, I load all 30,000 rows for no good reason—and I pay for the privilege. No thank you.

To limit the amount of data read, and thereby lower the total cost of the solution, I could execute a query that selects a “marker” indicating the latest or last row loaded. For this, I could select the maximum value from a field containing the date and time the table was last loaded; something like `Max(LastUpdatedDateTime)` or even a data integration lineage or metadata field like `Max(LoadDate)`. I could similarly select another marker such as the `Max(ID)` from an integer column reliably maintained by a sequence, identity, trigger, or other mechanism. The reliability of the mechanism represents the maximum confidence a data integration architect can place in the value. I will demonstrate building an incremental loader using an identity column maintained on the source data.

Before I do, I wish to point out that Chapter 11 contains a great section on the incremental load design pattern. There is a discussion of another solution I will not touch upon: change data capture. I encourage you to review Chapter 11 before you complete your data integration design.

Building the Cloud Loader

To work through the demonstration that follows, you will need a MASD account and database. Creating the account and database is beyond the scope of this book, but you can learn more at www.windowsazure.com/en-us/home/features/sql-azure. Once MASD is set up and configured, create a database. In this database, create a table named `dbo.LoadMetrics` using the T-SQL shown in Listing 14-1.

Listing 14-1. Creating the LoadMetrics Table

```
Create Table dbo.LoadMetrics
(ID int identity(1,1)
  Constraint PK_LoadMetrics_ID Primary Key Clustered
,LoadDate datetime
  Constraint DF_LoadMetrics_LoadDate Default(GetDate()))
,MetricName varchar(25)
,MetricIntValue int)
```

The LoadMetrics table will hold the last ID loaded for each table in the cloud destination. We will write this row once, and read and update it each load cycle. Accessing this table in this manner is the simplest and least processor-intensive manner in which to acquire the information we seek: the value of the last ID column loaded for a particular table. Why store this value in a table? Why not simply execute a `Max(ID)` select statement on the data table? Again, MASD charges for reads and not writes. Billing may change—it has in the past. What if we’re billed according to cycles or execution plans? You never know.

While connected to the MASD instance, create a table to hold your data. My data table will hold temperature information collected from my weather station in Farmville, Virginia. The table I use contains temperature- and humidity-related data and is shown in Listing 14-2.

Listing 14-2. Creating the CleanTemperature Table

```

Create Table dbo.CleanTemperature
  (ID int identity(1,1)
    Constraint PK_Cleantemperature_ID Primary Key Clustered
  ,MeasDateTime datetime
  ,MinT real
  ,MaxT real
  ,AvgT real
  ,MinH smallint
  ,MaxH smallint
  ,AvgH smallint
  ,ComfortZone smallint
  ,MinDP real
  ,MaxDP real
  ,AvgDP real
  ,MinHI varchar(7)
  ,MaxHI varchar(7)
  ,AvgHI varchar(7)
  ,LoadDate datetime
  ,LowBatt bit
  ,SensorID int)

```

Once the cloud tables have been created, we can begin work on an SSIS loader.

Locally, create a new SSIS solution and project named **CloudLoader**. Rename the default SSIS package **SimpleCloudLoader.dtsx**. Add a Sequence container and rename it **SEQ Pre-Load Operations**. Add an Execute SQL task by placing it into the Sequence container and rename it **Get AWCleanTempMaxID From AndyWeather**. Set the **ResultSet** property to **Single Row** and change the **ConnectionType** property to **ADO.Net**. Create the ADO.Net connection manager using information from your MASD account. To acquire the latest ID from the **LoadMetrics** table, I use the following query.

```

IF NOT EXISTS
(
  Select *
  From dbo.LoadMetrics
  Where MetricName = 'CleanTempMaxID'
)
BEGIN
  INSERT INTO dbo.LoadMetrics (LoadDate, MetricName,MetricIntValue)
  VALUES (NULL,'CleanTempMaxID',NULL);
END
Select Coalesce(MetricIntValue, 0) As CleanTempMaxID
From dbo.LoadMetrics
Where MetricName = 'CleanTempMaxID'

```

On the Result Set page, I store the value in an SSIS variable of an **Int32** data type named **AWCleanLoadMaxID**.

Add another Execute SQL task into the Sequence container and rename it **Get CleanTempMaxID from the local table**. Configure the connection to your source database and table. For me, it's a local default instance of SQL Server hosting the WeatherData database and the `dbo.CleanTemperature` table. I use the following T-SQL to extract the current maximum value from the table, configuring a single row result set to push this value into the `CleanTempLocalMaxID` SSIS variable (Int32 data type).

```
Select Max(ID) As CleanTempLocalMaxID
From dbo.CleanTemperature
```

Add a Data Flow task outside the SEQ Pre-Load Operations Sequence container and rename it **Load Windows Azure SQL Database**. Connect an `OnSuccess` precedence constraint between the Sequence container and the Data Flow task. Open the Data Flow Task Editor and add an OLE DB source adapter. Connect the OLE DB source adapter to a local source database you wish to load in the cloud and write a query to pull the latest data from the desired table. In my case, I am pulling data from my `dbo.CleanTemperature` table. To accomplish the load, I use the source query shown in Listing 14-3.

Listing 14-3. WeatherData Source Query

```
SELECT ID
      ,MeasDateTime
      ,MinT
      ,MaxT
      ,AvgT
      ,MinH
      ,MaxH
      ,AvgH
      ,ComfortZone
      ,MinDP
      ,MaxDP
      ,AvgDP
      ,MinHI
      ,MaxHI
      ,AvgHI
      ,LoadDate
      ,LowBatt
      ,SensorID
FROM dbo.CleanTemperature
WHERE ID Between ? And ?
```

Click the Parameters button and map `Parameter0` and `Parameter1` to the `AWCleanLoadMaxID` and `CleanTempLocalMaxID` variables as shown in Figure 14-2.

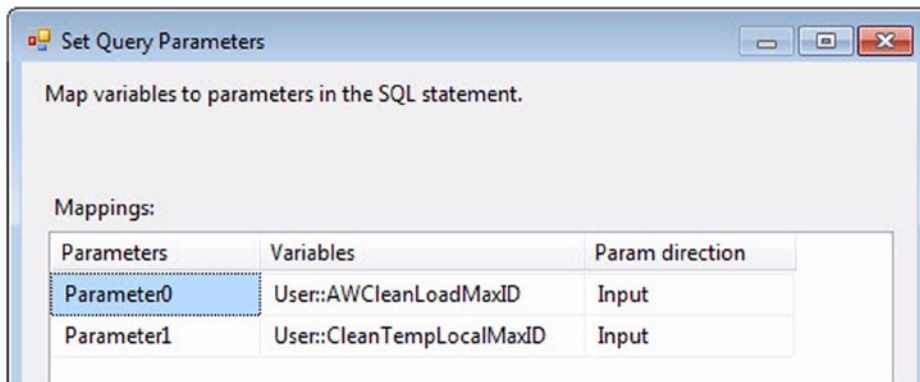


Figure 14-2. Mapping the *SQLAzureCleanLoadMaxID* variable to *Parameter0*

The question marks in the source query shown in Listing 14-3 are replaced with the values stored in the respective mapped variables. This query will only return rows where the ID is greater than the value stored in the cloud. Why do we grab the maximum ID from the source table before the load? In a word, latency. In the WeatherData database, the latency is minimal. But think about loading highly active systems—latency can be an issue. For example, suppose several transactions per second are entering the source table and it takes a few seconds to load the destination. If we wait until the load is complete to capture the source table's Max ID value, that value will likely include data we didn't load. The technical term for that is “bad.” So we design the package to grab the Max ID value before the load starts and only load rows between the last ID loaded into MASD and the Max ID value captured at the start of the SSIS package. And we never miss a row.

Returning to the demo package, add an ADO.Net destination adapter and rename it **Windows Azure SQL Database ADO NET Destination**. Connect a data flow path from the OLE DB source adapter to the ADO.Net destination adapter. Why an ADO.Net destination? MASD allows only ADO.Net and ODBC connections—there is no support at the time of this writing for OLE DB connections to MASDs.

Connect a data flow path between the source and destination adapters, edit the destination, and map the columns.

The last step is to update the LoadMetrics table in the MASD database. To accomplish this update, add an Execute SQL task to the control flow and rename it appropriately and descriptively. I named mine **Update AndyWeather LoadMetrics Table** and configured it to use ADO.Net to connect to my MASD database. My query looks like this one shown in Listing 14-4.

Listing 14-4. Updating the Azure SQL Database LoadMetrics Table

```
Update dbo.LoadMetrics
Set MetricIntValue = (@MaxID + 1)
, LoadDate = GetDate()
Where MetricName = 'CleanTempMaxID'
```

Map the value of CleanTempLocalMaxID into the @MaxID parameter on the Parameter Mapping page. And that's it. This script makes the current maximum ID the minimum ID of the next load.

Conclusion

In this chapter we examined aspects of the architecture and design for loading cloud destinations. We designed a sound solution after weighing the architectural and economic considerations.



Logging and Reporting Patterns

An essential part of managing any application is knowing what happens during the day-to-day usage of the application. This theme holds especially true in ETL solutions in which the data being manipulated can be used for reporting and analysis. Administrators satisfy this need through logging and reporting of the executions, errors, and statuses of the applications, which fits perfectly into the management framework concept.

The past few chapters have discussed how to set up other pieces of the management framework, including how to execute parent-child packages and how to implement centralized custom logging. This chapter will describe how to use the built-in logging in Integration Services to report on all aspects of an Integration Services application.

Integration Services provides two primary methods to help satisfy the logging and reporting need.

- Package logging and reporting
- Catalog logging and reporting

Let's walk through how to set up each of these methods and then utilize patterns that best highlight these methods.

Package Logging and Reporting

The package logging and reporting method has been around since the first edition of Integration Services. This method is characterized by setting up logging during development of the package. A logging provider can log to different outputs, including SQL Server tables, text files, and more. The log information is stored in one object, such as one file or the `sysssislog` table.

Each log can be restricted to store only certain types of events, such as `OnError`, `OnPreExecute`, and `OnVariableValueChanged`. An administrator can then look at the logs to see what happened during the execution of the package. Once the package has been deployed to the server, you cannot change the type or amount of logging that occurs.

Package logging is the best and only option when you're using Integration Services 2005 or 2008 or Integration Services 2012 or 2014 in package deployment mode. In the project deployment model, you can use package logging on a regular basis to keep track of errors that may occur or to ensure that packages are executing when expected. For more in-depth logging and reporting, you will want to use catalog logging and reporting, which we will discuss later in this chapter.

Let's take a look at setting up package logging and then how to use the output.

Setting Up Package Logging

To set up logging at a package level, you will go to the package itself and turn on logging. Each package needs to be set up separately to log to the database. You can do this by right-clicking on the package and selecting the Logging option or by going to the SSIS menu at the top of the SQL Server Data Tools (SSDT) and selecting the Logging option.

Within the logging menu, which you can see in Figure 15-1, you will decide what type of logging you want to use. Among the options are text files, XML files, and SQL Server tables. Once you have decided on the type of logging, you will select which events you want to log and at what level you want to log these events. If you select events at the highest package level, you will be able to see all events for all lower containers, too.

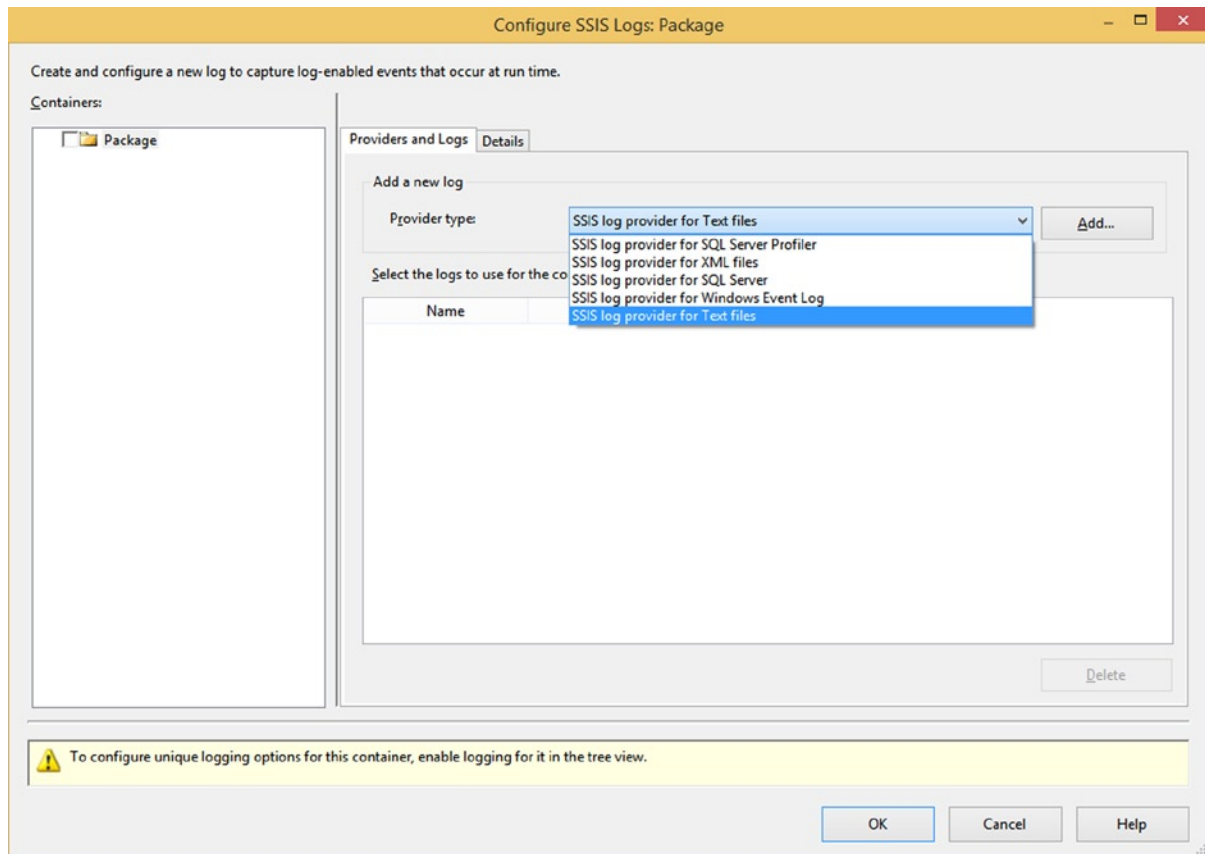


Figure 15-1. The SSIS logging menu

When the package runs, Integration Services creates a new table, if one is not already available, and stores the logging information within it. The table `sysssislog` contains the data for all recorded events.

Reporting on Package Logging

Once you've run the package with logging, you'll want to know what happened! The table that contains all of the information you need is called `sysssislog`. By default, it will be created in the `msdb` database on the server of the connection manager you selected in the logging menu; however, you can change the database by specifying it directly in the connection manager.

Let's take a look at the data in the table once we've run the package by running the following SQL query:

```
select * from msdb.dbo.sysssislog
```

This statement returns results similar to those in Figure 15-2.

	id	event	computer	operator	source	sourceid	executionid	starttime	endtime
1	1	OnPreValidate	2012RC0	2012RC0\MVP	Ch19_Reporting	D3F185D9-B64A-4293-9789-306712FEDE5C	E3961A32-67D4-4978-B077-86E92AF4FD20	2012-02-06 23:44:21.000	2012-02-06 23:44:21.000
2	2	OnPostValidate	2012RC0	2012RC0\MVP	Ch19_Reporting	D3F185D9-B64A-4293-9789-306712FEDE5C	E3961A32-67D4-4978-B077-86E92AF4FD20	2012-02-06 23:44:21.000	2012-02-06 23:44:21.000
3	3	OnPreExecute	2012RC0	2012RC0\MVP	Ch19_Reporting	D3F185D9-B64A-4293-9789-306712FEDE5C	A14EB62C-2ECC-4B93-86A5-3F2E02390CD4	2012-02-06 23:44:22.000	2012-02-06 23:44:22.000
4	4	OnPostExecute	2012RC0	2012RC0\MVP	Ch19_Reporting	D3F185D9-B64A-4293-9789-306712FEDE5C	A14EB62C-2ECC-4B93-86A5-3F2E02390CD4	2012-02-06 23:44:23.000	2012-02-06 23:44:23.000
5	5	DiagnosticEx	2012RC0	2012RC0\MVP	Ch19_Reporting	D3F185D9-B64A-4293-9789-306712FEDE5C	A14EB62C-2ECC-4B93-86A5-3F2E02390CD4	2012-02-06 23:44:23.000	2012-02-06 23:44:23.000
6	6	PackageStart	2012RC0	2012RC0\MVP	Ch19_Reporting	D3F185D9-B64A-4293-9789-306712FEDE5C	A14EB62C-2ECC-4B93-86A5-3F2E02390CD4	2012-02-06 23:44:23.000	2012-02-06 23:44:23.000
7	7	DiagnosticEx	2012RC0	2012RC0\MVP	Ch19_Reporting	D3F185D9-B64A-4293-9789-306712FEDE5C	A14EB62C-2ECC-4B93-86A5-3F2E02390CD4	2012-02-06 23:44:23.000	2012-02-06 23:44:23.000
8	8	OnPreExecute	2012RC0	2012RC0\MVP	Ch19_Reporting	D3F185D9-B64A-4293-9789-306712FEDE5C	A14EB62C-2ECC-4B93-86A5-3F2E02390CD4	2012-02-06 23:44:24.000	2012-02-06 23:44:24.000
9	9	OnPostExecute	2012RC0	2012RC0\MVP	Ch19_Reporting	D3F185D9-B64A-4293-9789-306712FEDE5C	A14EB62C-2ECC-4B93-86A5-3F2E02390CD4	2012-02-06 23:44:24.000	2012-02-06 23:44:24.000
10	10	DiagnosticEx	2012RC0	2012RC0\MVP	Ch19_Reporting	D3F185D9-B64A-4293-9789-306712FEDE5C	A14EB62C-2ECC-4B93-86A5-3F2E02390CD4	2012-02-06 23:44:24.000	2012-02-06 23:44:24.000

Figure 15-2. Results from the SSIS log table

Design Pattern: Package Executions

Although it is possible to use the information in the table directly, you can also combine the information to make it a little more readable. If you want to see the package executions and how long each package took to run, you can use the query in Listing 15-1:

Listing 15-1. Query to Return Package Durations

```
select ssis.source
       , min(starttime) as package_start
       , max(endtime) as package_end
       , DATEDIFF(ms, min(starttime), max(endtime)) as duration_ms
from msdb.dbo.sysssislog ssis
where event in ('PackageStart', 'PackageEnd')
group by ssis.source, ssis.executionid
```

Catalog Logging and Reporting

The catalog logging and reporting method was introduced in Integration Services 2012 and is the best logging method to use if it is available. It can be used only if you have set up the project deployment model type. The nice thing about this type of logging is you don't need to prepare anything in the package to utilize it. Let's jump right into how to set up the logging and design patterns to report on that data.

Setting Up Catalog Logging

As I mentioned earlier, the benefit of catalog logging is that you don't need to modify the package at all to use the logging output. The only preparation you need to do is to make sure your package is set to the project deployment type and deploy the package to the SSIS Catalog.

To begin setting up catalog logging and reporting, you will create an SSIS Catalog. You can do this by connecting to the database instance. If Integration Services is installed, you will see a node entitled Integration Services Catalogs. If you create a new catalog named SSISDB, it will look like Figure 15-3.

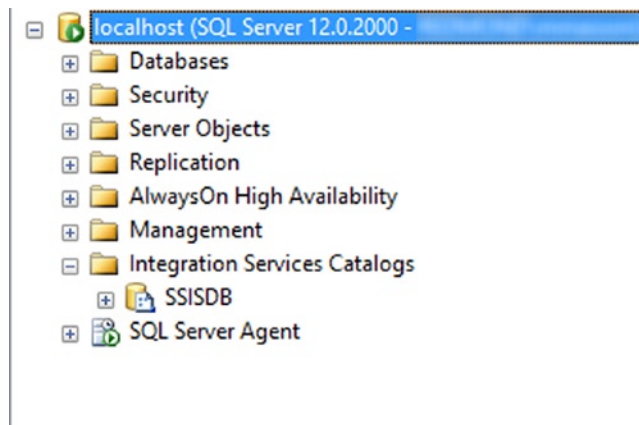


Figure 15-3. The SSISDB catalog

At this point, you are ready to deploy your package. First, though, you should make sure the project is set to use the project deployment model. You can do this by right-clicking on the project. If you see the option for Convert to Package Deployment Model, as shown in Figure 15-4, you are in this mode.

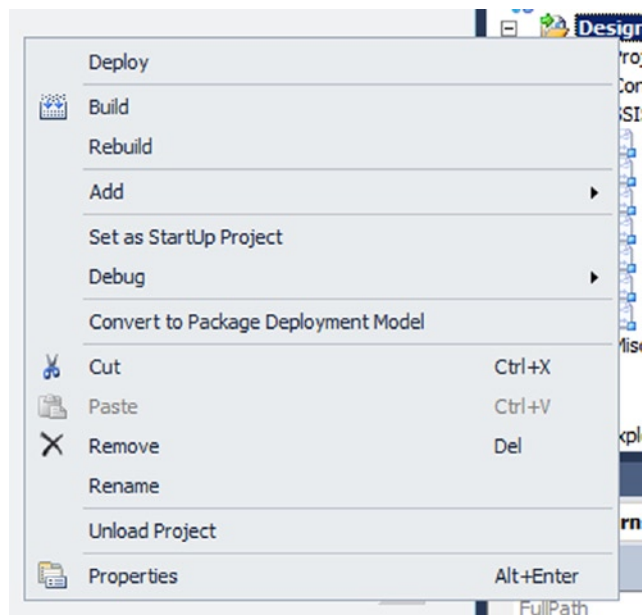


Figure 15-4. A project in project deployment mode

Finally, you will deploy the package to the SSIS Catalog. This stores the package in the SSISDB database and allows for some default and some configurable logging.

Next, we will look at the tables where the information for both types of logging is stored.

Catalog Tables

When a package runs, all of the information is stored in a set of tables that reside in the SSISDB database on the same server where the Integration Services package was deployed. Although there is a series of internal tables, you will do most of your reporting from the catalog views. Figure 15-5 shows a database diagram of the SSIS internal tables, whereas Figure 15-6 shows a list of the SSIS catalog views.

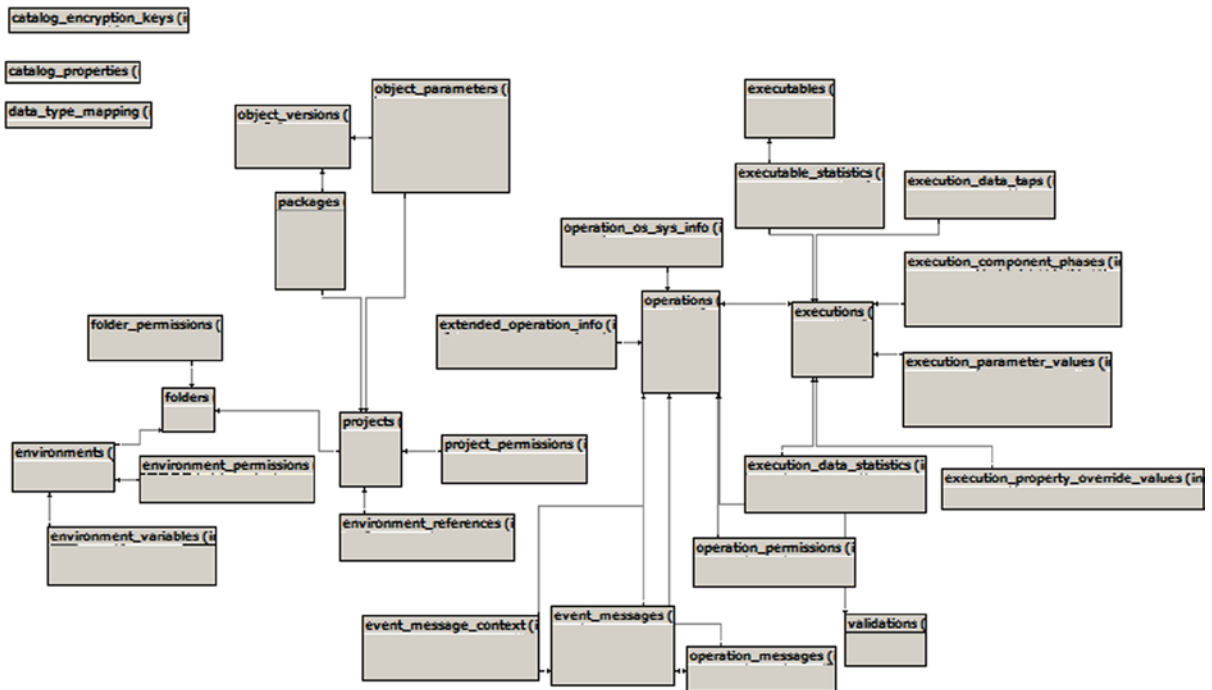


Figure 15-5. SSIS catalog internal tables

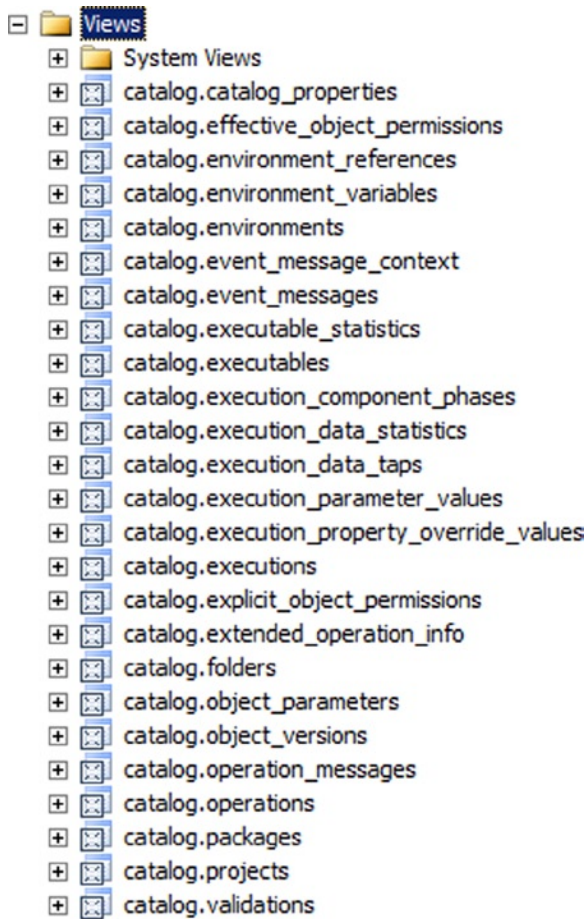


Figure 15-6. SSIS catalog views

Changing Logging Levels After the Fact

Even after the package has been deployed to the Integration Services server, you can change the amount of logging that occurs. But why would you want to do this? If you initially set up your package with a defined set of logging events, you will see only that set of data. However, you may want to include more events if you are doing more advanced troubleshooting or if you have a specific error you need to track down. On the other hand, you may want to increase the performance of a package by reducing the number of events that are recorded.

Modifying logging at the package level is not a best practice. By opening up the package to change even the slightest item, you increase the risk of a breaking change, whether by mistyping a value or choosing an unavailable logging option. In some organizations, modifications to logging made at the package level may even result in the package having to go through the change control process again. Ideally, you want to make logging changes in an external location without touching the package at all.

In Integration Services 2012, you can choose from four different logging levels, as described in Table 15-1.

Table 15-1. *SSIS Logging Levels*

Logging Level ID	Level	Events	Notes
0	None	None	Captures enough information to say whether the package succeeded or failed but does not log any messages to the [operation_messages] view.
1	Basic	OnPreValidate OnPostValidate OnPreExecute OnPostExecute OnInformation OnWarning OnError	Captures similar information to what is displayed on the console by default when a package is run with dtexec.
2	Performance	OnWarning OnError	This level is used to track the performance information for the run (how long it took to run each task or component, how many rows were processed, etc.) but does not log all of the events captured by the Basic log level.
3	Verbose	All events	Captures all log events, including performance and diagnostic events; can introduce some overhead on performance.

Design Patterns

Now that you know how to set up and log information, let's walk through the following design patterns:

1. Changing the logging level
2. Utilizing existing reports
3. Creating new reports

Changing the Logging Level

Now that you know what the different logging levels are and when you would use each one, let's walk through changing the logging level. You can do this in either of two ways: through the execution interface or through a command-line execution.

To modify the logging level through the execution interface, you will connect your Integration Services Catalog, right-click on the desired package, and select **Execute**. On **Execute Package** screen, you will see the **Logging Level** option on the **Advanced** tab. By default, the option is set to **Basic**, as shown in Figure 15-7. Alternatively, you can modify this value to another logging level to see more or less in the logging tables.

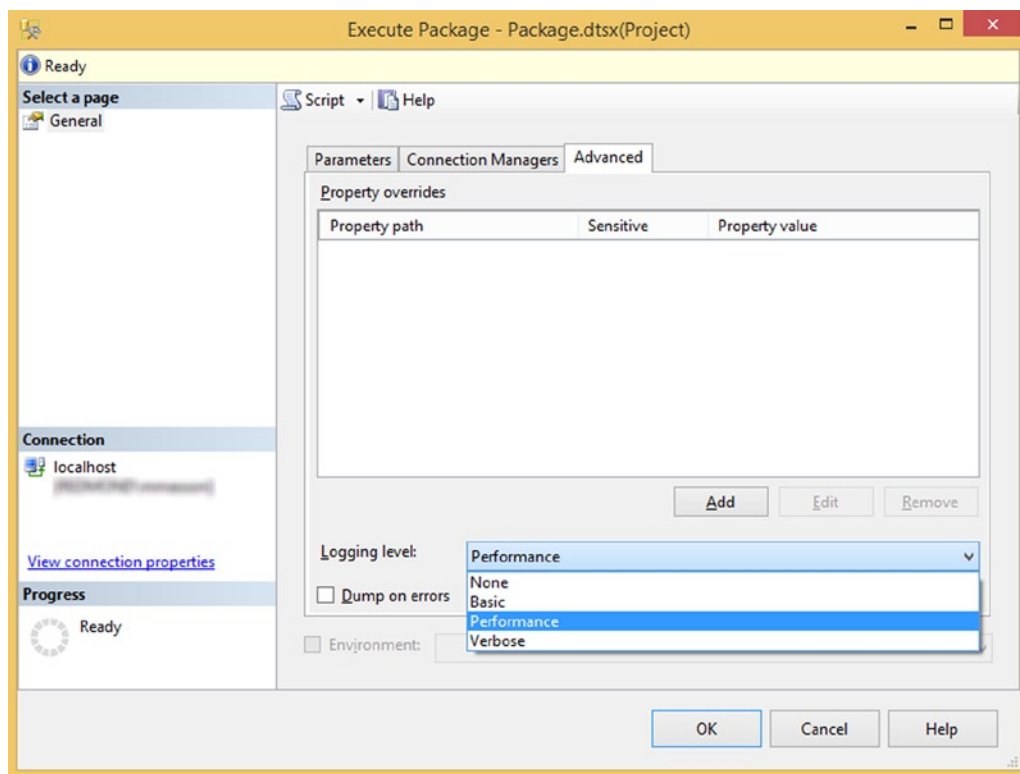


Figure 15-7. *Execute Package screen*

The other option is to modify logging through the command line. All packages can be executed through the command line, and you can set a logging level associated with an individual execution.

■ **Note** Much of the functionality associated with administering Integration Services packages can be accessed through a command-line interface. By using the command line, you can integrate your Integration Services administration with your other maintenance tasks.

Run the following code in Listing 15-2 to change the logging level to log all Verbose records for a new execution.

Listing 15-2. Statement to Modify the Logging Level for an Execution

```
DECLARE @execution_id INT
EXECUTE [catalog].[create_execution]
    @folder_name = 'DesignPatterns'
    ,@project_name = 'DesignPatterns'
    ,@package_name = 'Ch15_Reporting.dtsx'
    ,@reference_id = null
    ,@use32bitruntime = false
    ,@execution_id = @execution_id OUTPUT
```

```
EXECUTE [catalog].[set_execution_parameter_value]
    @execution_id
    ,@object_type = 50
    ,@parameter_name = 'LOGGING_LEVEL'
    ,@parameter_value = 3 --Verbose

EXECUTE [catalog].[start_execution]
    @execution_id
```

Once you've done this, you can see the output from the newly set logging level by running the query in Listing 15-3.

Listing 15-3. Query to Return All Messages

```
select * from catalog.event_messages
where operation_id =
    (select max(execution_id) from catalog.executions)
```

Using the Existing Reports

Our next design pattern is an important one: use what is provided to you. Included in the SSIS Catalog are reports that use the logging information we have just discussed. The information in these reports includes an in-depth view of all of your packages' executions. These reports are a great start for you to see when your packages run, if any errors occur, and potential trouble areas for you to investigate.

Figure 15-8 shows all of the reports available to you. You can access all reports through the Management Studio interface and the Integration Services Catalog node.

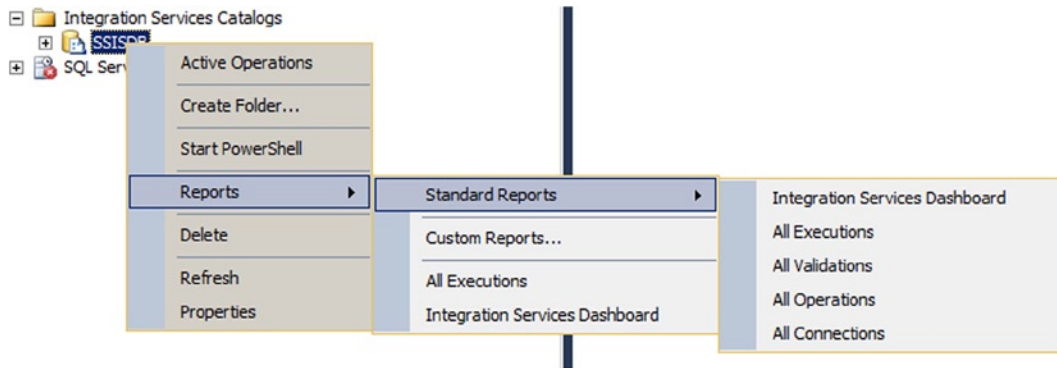


Figure 15-8. Available catalog reports

If you are looking at a specific execution, you will always want to start with the Overview report, which can be run by selecting the Overview link on any of the provided reports. In fact, at the end of an execution through the interface, you will be asked if you want to see this report. If you select yes, you will see something similar to Figure 15-9.

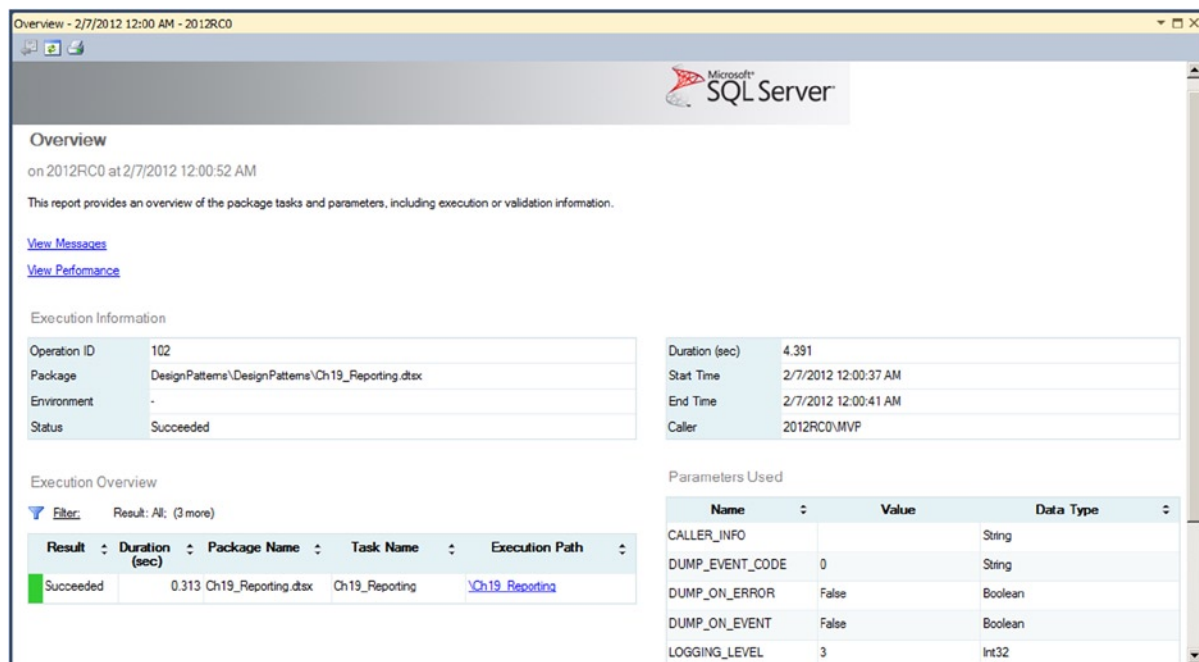


Figure 15-9. An overview report

Creating New Reports

Now that you've seen the reports that are available to you without doing any work, you may be perfectly happy. If not, you may want to dig into the data a little deeper. You can create new reports by looking at the catalog views that were described earlier. Particular reasons why you may want to do this include

1. Seeing the longest-running executions
2. Finding out why a package failed
3. Understanding the inner workings of a particular component

Let's start with the first reason. This report is interesting because it uses the main output view, but based on the query and transformations, it becomes a helpful little tool. Listing 15-4 shows the query that lists the five longest-running packages over the past day.

Listing 15-4. Query for Five Longest-Running Packages

```
select top 5 e.execution_id, e.package_name, DATEDIFF(ms, start_time, end_time) as duration_ms
from catalog.executions e
where e.start_time > DATEADD(dd, -1, getdate())
order by duration_ms desc
```

The second reason you may want a new report is to see why a package failed. You will use an additional view for this information, the `catalog.event_messages` view. Restricting the data on both the executions and the `event_messages` view will ensure that you get only packages that failed entirely and see only the events that caused them to fail. This query can be seen in Listing 15-5.

Listing 15-5. Failed-Packages Query

```

select e.execution_id, e.package_name, em.*
from catalog.executions e
inner join catalog.event_messages em on e.execution_id=em.operation_id
where e.status = 4 and em.event_name = 'OnError'

```

The final reason is to understand the inner workings of a particular component. You can see the individual steps that occurred during the execution of each component in the data flow. For example, the query in Listing 15-6 returns each step that occurs in the execution of the sources, transformations, and destinations and how long each step takes.

Listing 15-6. Query to Return Component Phases and Times

```

select subcomponent_name, phase
      , DATEDIFF(ms, start_time, end_time) as duration_ms
from catalog.execution_component_phases
where package_name = 'Ch16_Reporting.dtsx'
      and task_name = 'Data Flow Task'

```

Once you have your desired query, you can either run it directly from Management Studio or embed it into a Reporting Services report to make it look like the standard reports available in the solution. To make the report through Management Studio, you can store the folders in your local Documents folder, under the structure SQL Server Management Studio\Custom Reports. To access them, you will then select the Custom Reports option under the Reports menu on the Integration Services node, as shown in Figure 15-10.

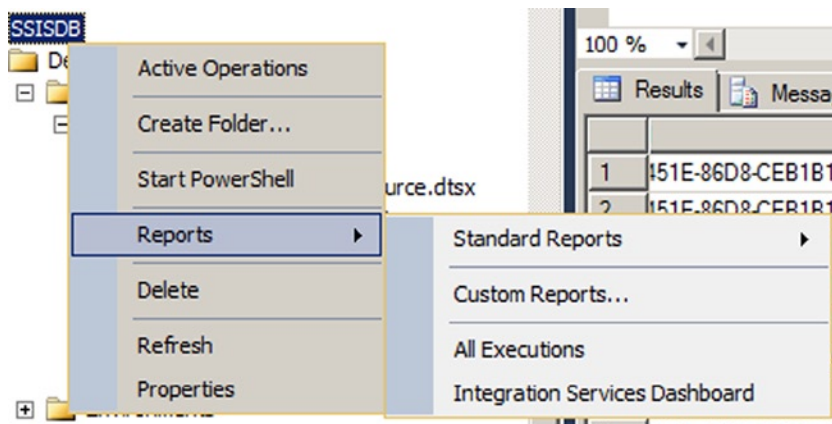


Figure 15-10. Selection of custom reports

Summary

This chapter has discussed many ways to monitor your Integration Services packages. Whether you are using an older version of the tool or the latest and greatest, you will be able to understand the internal workings of the package by following the design patterns described here. Discussions of both package logging and reporting and catalog logging and reporting have shown you how to modify the types of events you log and how to retrieve that information.



Parent-Child Patterns

In earlier versions of Integration Services, the data movement platform did not include a management framework, which is the implementation of the execution, logging, and deployment of the Integration Services packages. To try to fill this hole, developers created their own management framework to use in their organizations. As with any custom solution, the developers found that the management framework needed to be cared for and upgraded when new versions or new packages were introduced into the system.

Previous chapters have covered ETL instrumentation, focusing on metadata collection and validation. The metadata we discussed included key information you need to manage your packages. This chapter covers parent-child patterns, where an Integration Services package can execute another package from within its own execution. These patterns are a critical part of implementing an ETL framework.

Integration Services 2014 contains its own management framework, which includes logging and execution through the SSIS Catalog. In this and subsequent chapters, we will show you how to use the available framework and enhance it to provide more information while still working around the issues we discuss.

The following are the three parent-child patterns we'll cover in this chapter:

- The master package pattern
- The dynamic child package pattern
- The child-to-parent variable pattern

Using these patterns, you can implement the Integration Services management functionality out of the box.

Master Package Pattern

When setting up a framework, one of the first things you want to do is find a way to organize how your packages execute. Your organization could include parallel versus serial processing, conditional execution, and categorical batching. And although you could have some of this organization occur in a job scheduler such as SQL Server Agent or Tivoli, wouldn't it be easier if you could manage your package execution in an environment you already know?

Luckily for you, Integration Services already provides this ability! By using the workflow designer and the Execute Package task, you can execute other packages, creating a parent-child package relationship. When you use the parent-child relationship to execute a series of packages, you call this a master package. There are two steps you need to complete in order to set up one child package for your master package:

1. Assign the child package.
2. Configure the parameter binding.

Assign the Child Package

Once you have created your initial package, begin by using the Execute Package task from the SSIS Toolbox. Drag the task to the control flow, and open the task to see multiple menus that you can modify. Let's begin by configuring the Package page, as shown in Figure 16-1.

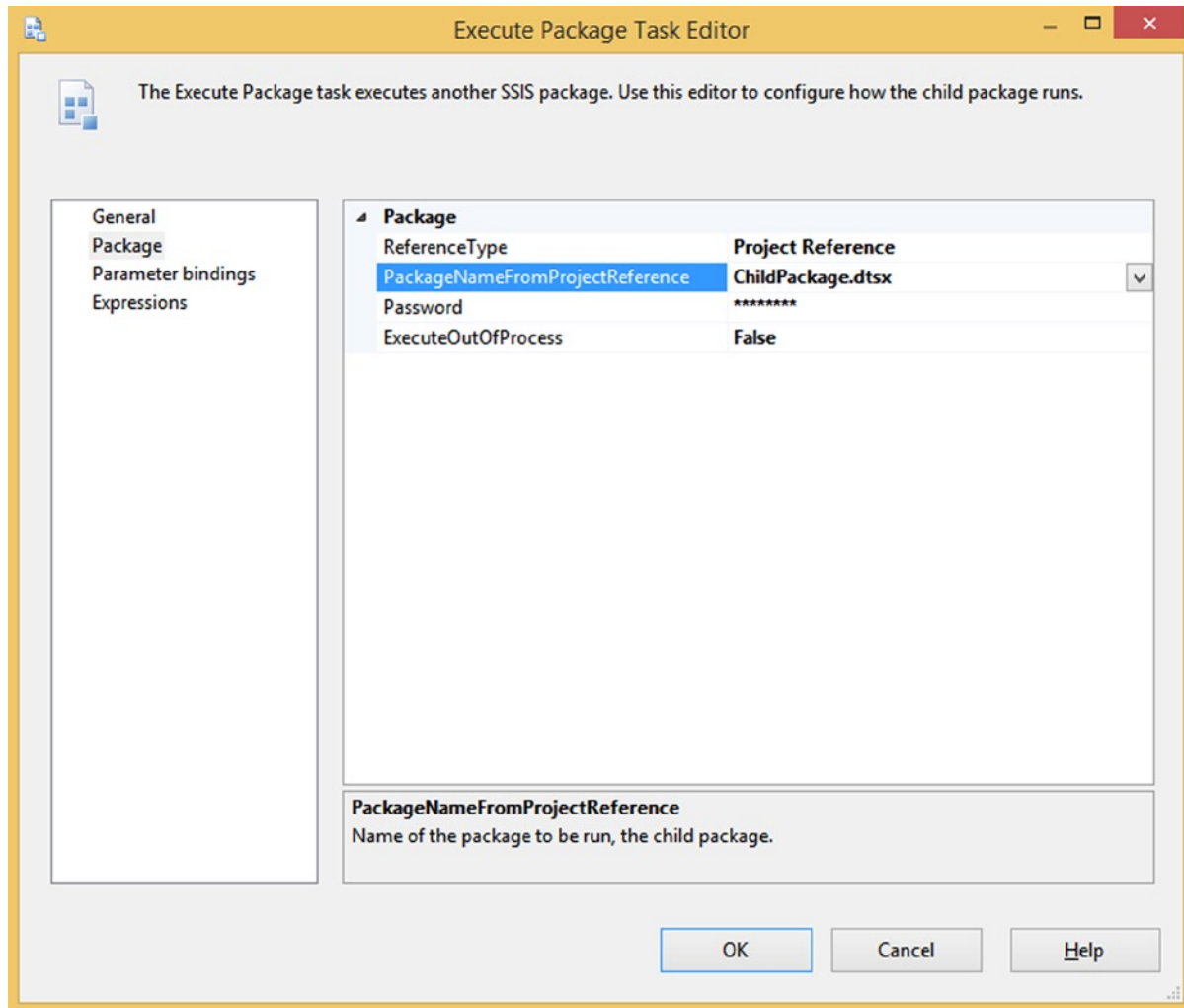


Figure 16-1. Execute Package Task Editor Package page

This is where you set up the package that you want to execute. A new addition to the Execute Package task is the `ReferenceType` property, which enables developers to use the master package to run a package that is included in this project or a package that is external to the project. For this example, you will just use an existing package in your solution.

At this point, you could click the OK button and have a perfectly acceptable master package. Before you do that, however, you should delve into passing information between the packages using the parameters in the next menu, Parameter Bindings.

Configure Parameter Binding

Just calling a child package isn't very exciting. What is exciting is tying the child package into something that the master package is doing! You do this through parent package parameters. This option can only be used if you are using a child package from the same project as the master package. Once you complete the setup for your package parameters, you should see the screen shown in Figure 16-2.

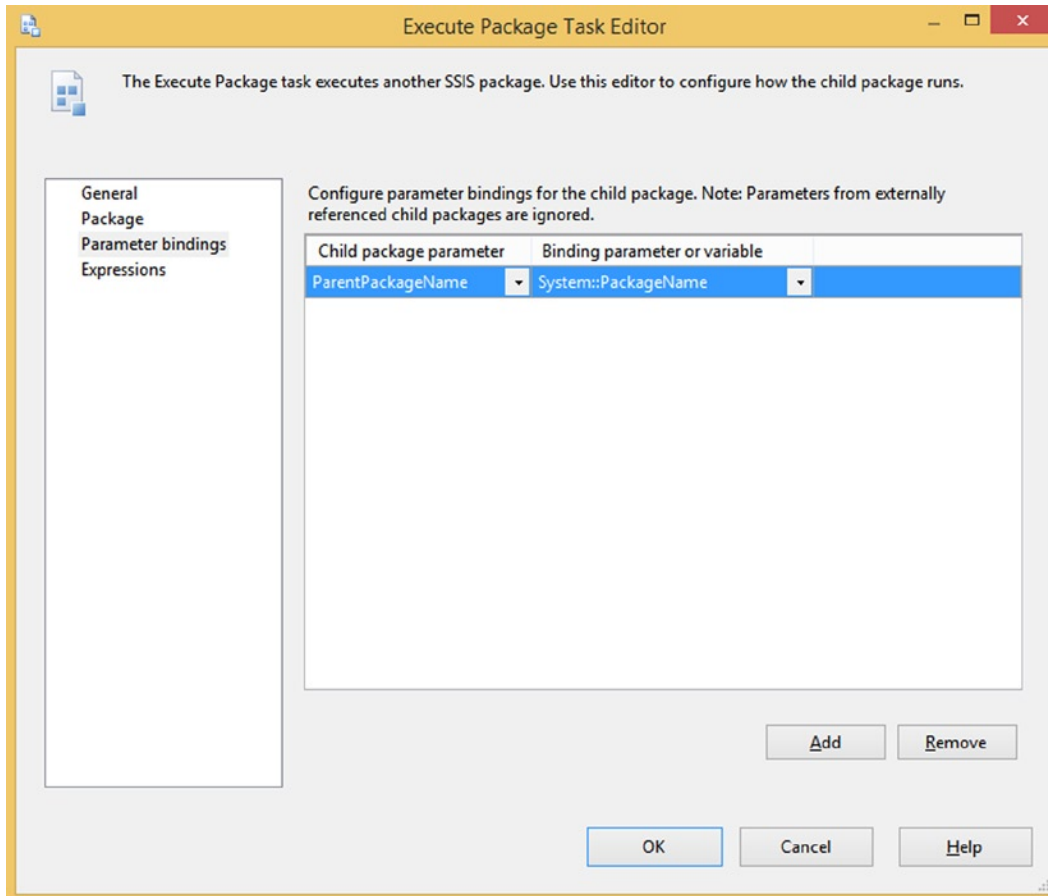


Figure 16-2. Execute Package Task Editor Parameter Bindings page

To achieve the result shown in Figure 16-2, you need to look at the Execute Package Task Editor and go to the Parameter Bindings page. Click the Add button to set up a parameter. For the Child Package Parameter, you can either select a parameter that has already been created or add your own parameter, in case you have not created the child package's parameter yet. Keep in mind that this will not automatically create the variable in the child package. That is up to you! Next, you will assign either a parameter or variable from the master package to be stored in the child parameter. In the scenario shown in Figure 16-2, we are storing the name of the parent package in a parameter in the child package, which we could then use to record the package that called the child package.

If you want to test the package, you can create a Script task in the child package using the code shown in Listing 16-1. Make sure to put the `$Package::ParentPackageName` parameter in the `ReadOnlyVariables` property. If everything is mapped correctly, when you run the package, you should see the name of the parent package in a message box, as shown in Figure 16-3.

Listing 16-1. Visual Basic Code to Display the Parent Package Name

```
Public Sub Main()  
    MsgBox("The name of the parent package is: " & _  
        Dts.Variables("$Package::ParentPackageName").Value.ToString)  
    Dts.TaskResult = ScriptResults.Success  
End Sub
```

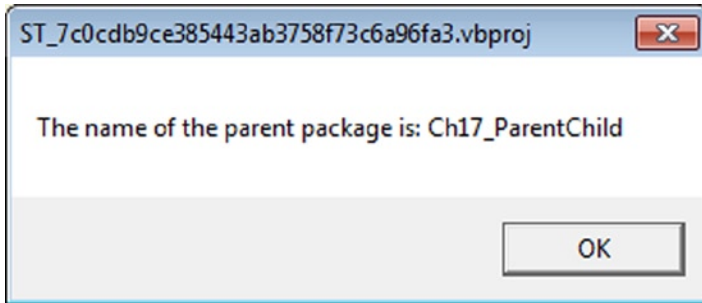


Figure 16-3. Message box showing the name of the parent package

Now that you have a working parent-child package, let's take it to the next level by creating a dynamic child package.

Dynamic Child Package Pattern

One of the nice things about Integration Services is the flexibility it provides if you want to do something a little different. For example, if you are not sure exactly which packages need to run, you can create a master package that has a dynamic child package that will only execute the desired packages. This is a great idea if you have a series of files coming in, but you're not sure which files come in at a certain time. Your end goal is to create a package that looks like Figure 16-4. Let's walk through an example of creating the master package and a list of the dynamic packages that you want to execute.

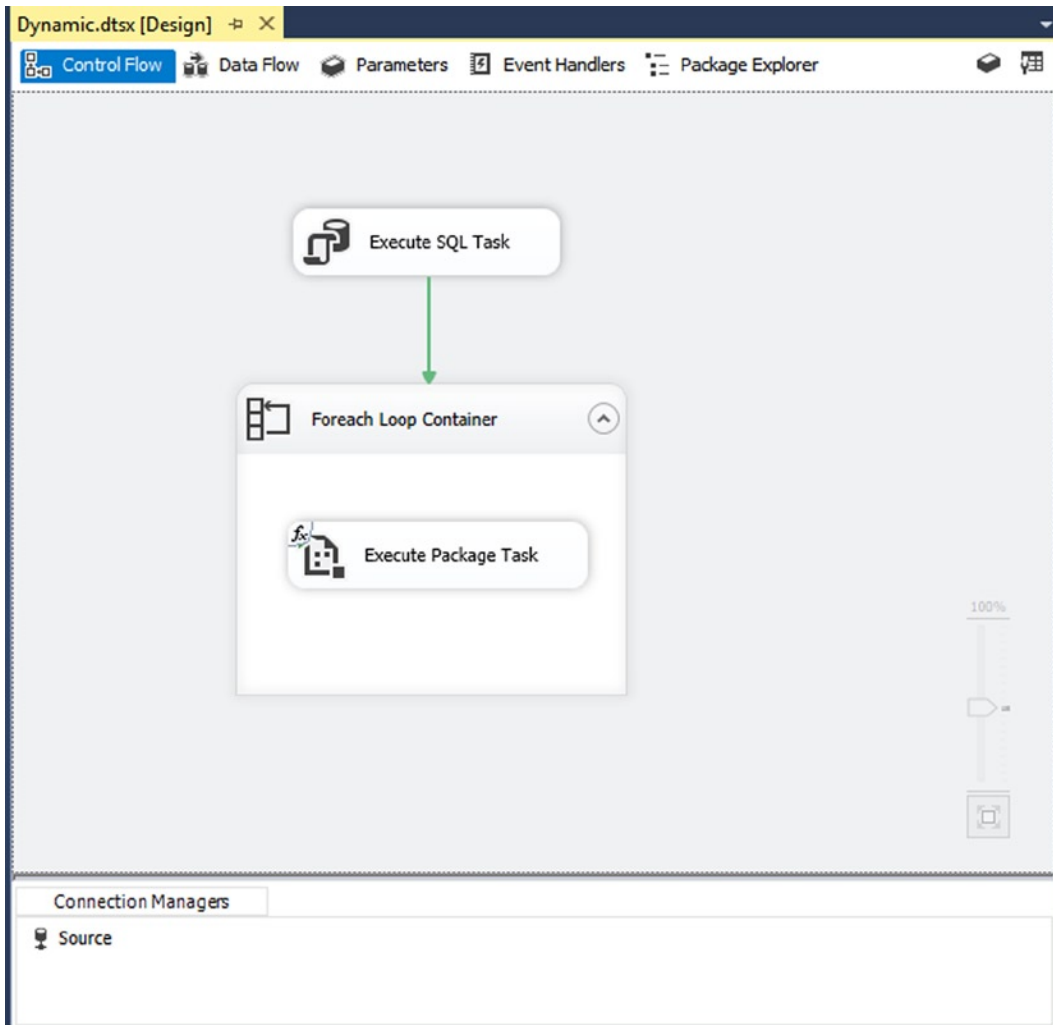


Figure 16-4. Completed dynamic child package pattern package

To create the table that contains the package names, run the Create and Insert statements found in Listing 16-2. Either create a database named `DesignPatterns`, or modify the script to run against some other database that you have available and can use for experimenting.

Listing 16-2. T-SQL Code to Create and Populate a Package List Table

```

USE [DesignPatterns]
GO

CREATE TABLE [dbo].[PackageList](
    [ChildPackageName] [varchar](50) NULL
)
GO

INSERT INTO [dbo].[PackageList] ([ChildPackageName])
VALUES ('ChildPackage.dtsx')
GO

INSERT INTO [dbo].[PackageList] ([ChildPackageName])
VALUES ('ChildPackage2.dtsx')
GO

```

Now you will create the master package. Starting with a blank SSIS package, create a variable that is scoped to the package level. (In this chapter's example, the SSIS package is named `Dynamic.dtsx`). The variable should be named `packageListObject` and have a data type of `Object`. You do not need to provide a value for the variable. Secondly, add a variable, also scoped to the package level, named `packageName` with a data type of `String`. Set the value of this variable to the same name as one of the packages in your project (i.e., `ChildPackage.dtsx`) so it can be used for design-time configuration.

Next, add an `Execute SQL` task in the control flow. Use the query in the `Execute SQL` task shown in Listing 16-3 against the database you just created for your table. (Hint: You'll need a `Connection Manager` named `Source` that points to the `DesignPatterns` database).

Listing 16-3. T-SQL Code to Query the Package List Table

```

SELECT [ChildPackageName] FROM [dbo].[PackageList]

```

In addition to the SQL query, ensure the `ResultSet` property is set to return a `Full Result Set` and store it in the variable you just created called `packageListObject`. This property page can be seen in Figure 16-5.

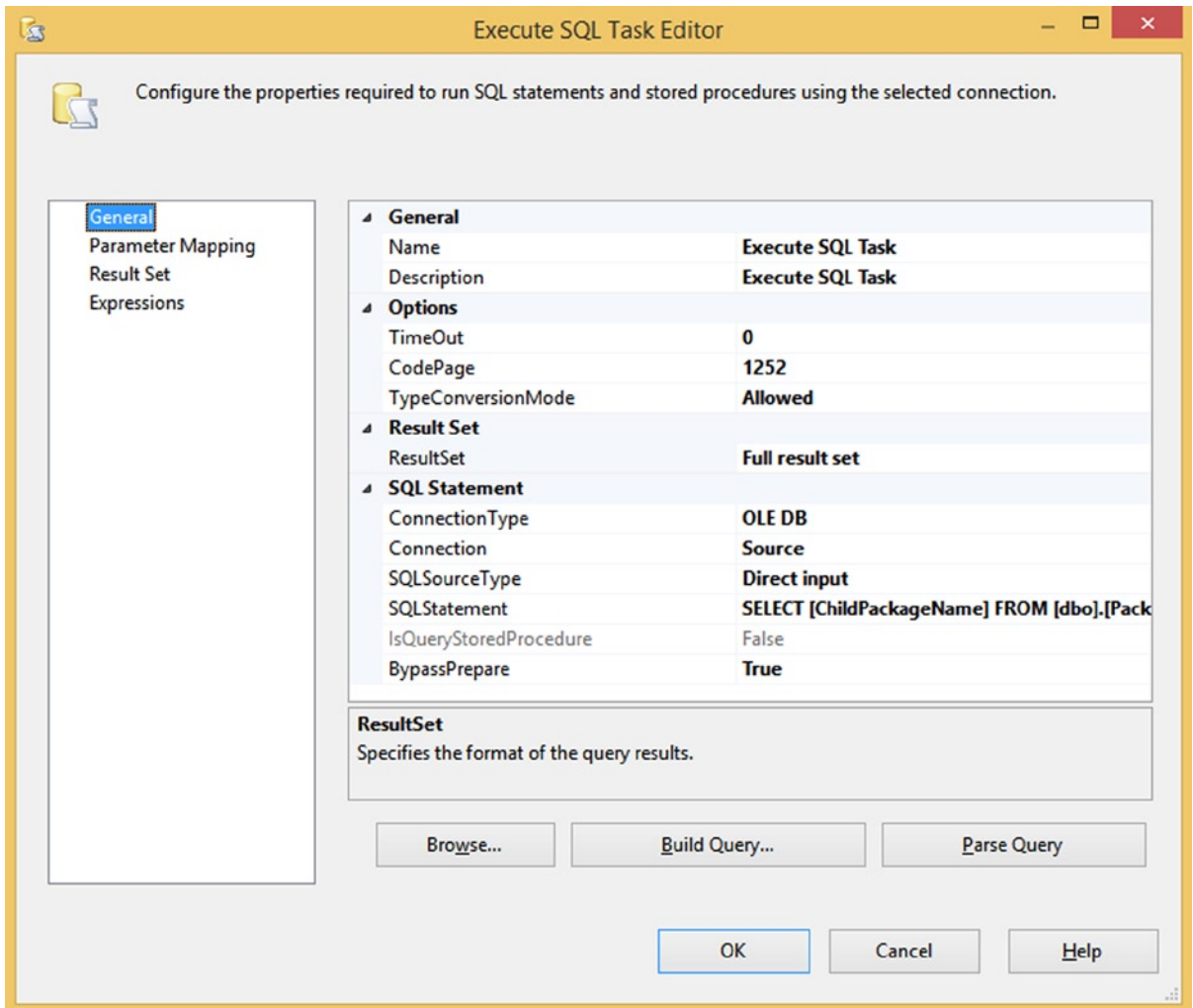


Figure 16-5. Execute SQL Task Editor General page

Attach a ForEach Loop container to the Execute SQL task. This is where you will execute the package. Within the Collection page of the ForEach Loop container, set the enumerator to use Foreach ADO Enumerator, which will loop through the variable object. The ADO object source variable field should contain @[User::packageListObject]. This screen can be seen in Figure 16-6.

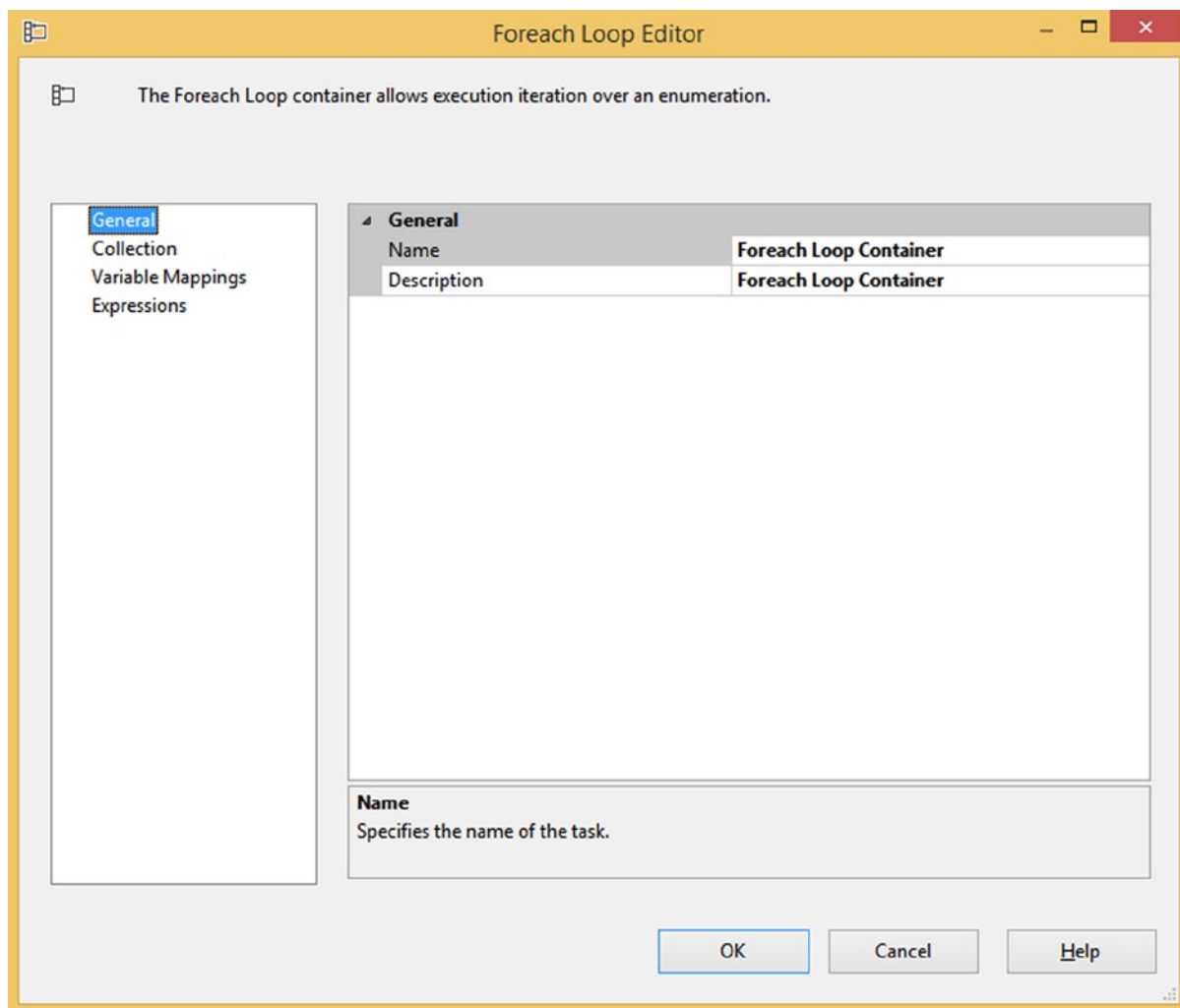


Figure 16-6. Foreach Loop Editor General page that enumerates through each row in the `packageListObject` variable

Then you need to tell Integration Services what to do with the value it retrieves when enumerating through the object list. On the Variable Mappings page, set the variable to `@[User::packageName]` and the Index to 0. This will put each value into the variable.

Finally, you're at a point where you can add the part that executes the package. Similar to the creation of the master-child package, you want to use an Execute Package task. Begin by setting the `DelayValidation` property to True, which allows you to decide what package to run at runtime.

Rather than walk through the same steps as you did in the master-child package, go directly to the Expressions page in the Execute Package Task Editor. This is where you set up the dynamic portion of the package. Set the `PackageName` property to use the expression `@[User::packageName]`. The final Expressions page should look like Figure 16-7.

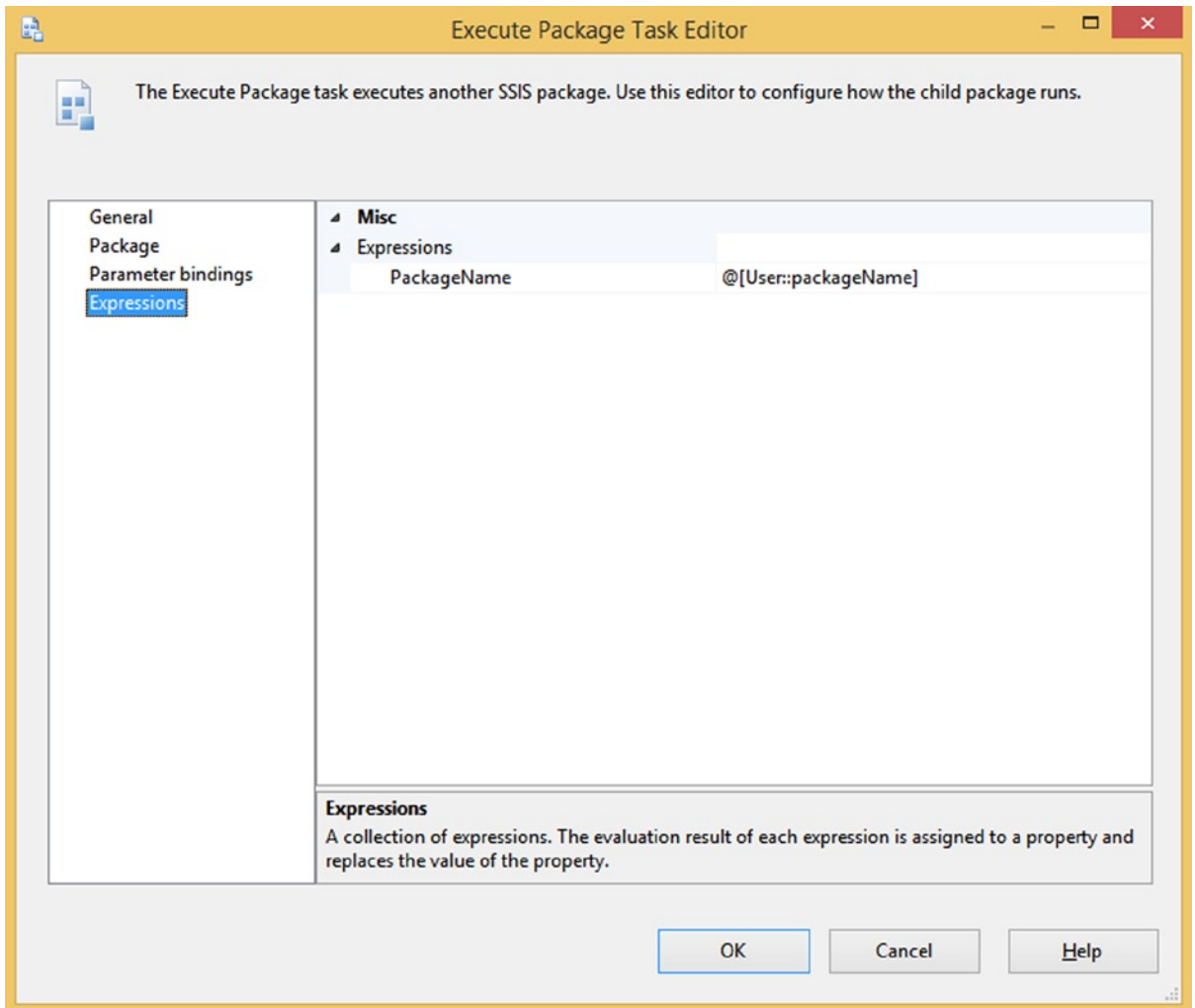


Figure 16-7. *Execute Package Task Editor Expressions page*

When the package runs, it will loop through each row in the `PackageList` table; set the `PackageName` property of the Execute SQL task to the current row, and execute only the packages that you need. Keep in mind that this will always run the child packages serially unless you create multiple loops and specifically code your master package to handle parallelism.

Next, we will describe how a child package can send information back to the parent package in the child-to-parent variable pattern.

Child-to-Parent Variable Pattern

Parent-child patterns are an essential part of a management framework. For example, you could use the master package pattern to group similar packages together and make sure they are executed in the correct order. You could also use the dynamic child package pattern to run a variable number of packages. To ensure that you store all of this information, it is vital that you pass important information between packages, not only from the parent to the child, but also from the child back to the parent. Although this feature is not readily known, it is possible to do this using the Script task. Let's use your existing packages to show how to pass the name of a file from the child package to its parent.

The first step is to create a variable in the parent package. In this example scenario, you are going to create a variable named `ChildFileName` of data type String that is scoped at the package level. Attached to the Execute Package task you created previously in this chapter, you'll add a Script task. Add the `ChildFileName` variable as a `ReadOnly` variable, and add the code in Listing 16-4 inside the Visual Basic script.

Listing 16-4. Visual Basic Script to Display the Child File Name

```
Public Sub Main()
    MsgBox("The name of the child file is: " & _
        Dts.Variables("User::ChildFileName").Value.ToString)
    Dts.TaskResult = ScriptResults.Success
End Sub
```

Next, modify your child package. In the Script task, add the variable `User::ChildFileName` to the `ReadWriteVariables` property list. You will have to manually type this in, as it will not display in the menu. Thus, your full, read-only variables list appears as:

```
$Package::ParentPackageName,User::ChildFileName
```

Then add the line of code found in Listing 16-5 to the Visual Basic Script task.

Listing 16-5. Visual Basic Script to Set the Child File Name Value

```
Dts.Variables("User::ChildFileName").Value = "SalesFile.txt"
```

Once you run it, the package will finish with the figure seen in Figure 16-8.

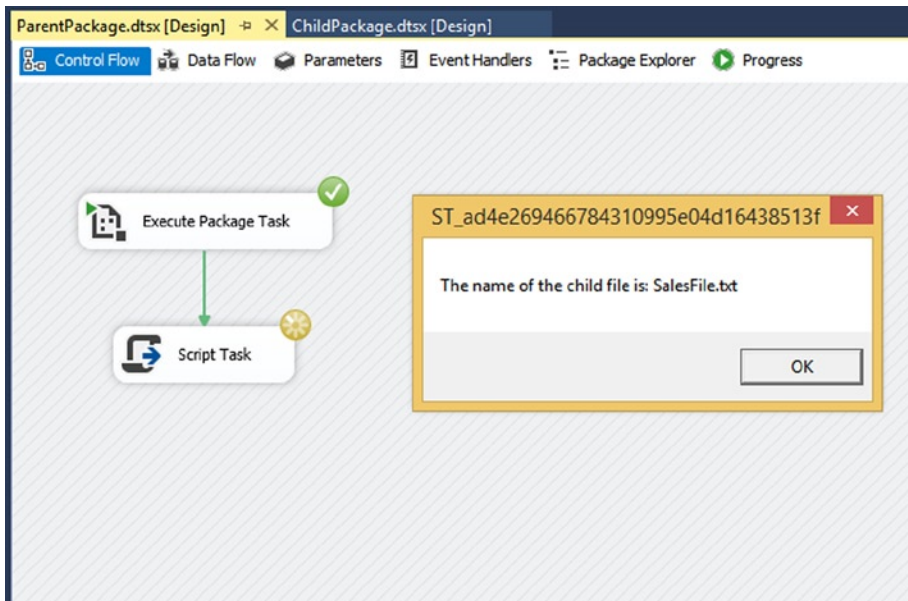


Figure 16-8. *Child-to-parent variable pattern execution*

The passing of variable values from child to parent succeeds because of how containers work in Integration Services. Inside of a package, any child container, such as a Sequence container, can access its parent's properties. Likewise, any child task, such as an Execute SQL task, can access its parent's properties. This paradigm allows you to use variables and properties without having to re-create them for every object in your package. When you add a child package using the Execute Package task, you add another layer to the parent-child hierarchy and allow the child package to set the parent package's variable.

Conclusion

SQL Server enthusiasts everywhere embraced Integration Services when it was first introduced as part of SQL Server 2005. The latest edition of Integration Services has been enhanced to make ETL developers even more excited than before. Integration Services 2012 added the basis for a management framework and the ability to create parent-child relationships, as this chapter discussed. We also discussed master package patterns and management frameworks.



Configuration

SQL Server 2012 introduced a new, parameter-based configuration model for SSIS. This new model is meant to simplify the configuration process and make it easier for users to identify where values are coming from at runtime. Although 2005/2008-style package configurations are still supported in SQL Server 2012 and SQL Server 2014, the two configuration models are not meant to be mixed. In fact, the menu option to use them will only appear when you are using the file deployment model and on packages that have been upgraded from previous versions. New packages created in SQL Server 2014 will use the new parameter model by default.

This chapter describes the new parameter model and how it can be used to configure package properties at runtime. We'll look at how parameters are exposed in the SSIS Catalog and how you can set parameter values as part of your build process using Visual Studio configurations. Finally, we'll look at design patterns that you can use to augment the functionality provided by the built-in parameter model, providing dynamic runtime configuration.

Parameters

SSIS parameters allow packages to define an explicit contract, much like function parameters do in programming languages like C#. Unlike package configurations, parameters are exposed to the callers, like SQL Server Agent, or the Execute Package task, so users are able to see exactly what a package needs to run. Parameters are essentially read-only package variables in a special namespace. They follow the same type system as package variables and will appear in all of the same UIs that variables do (for example, for setting property expressions). You'll make use of parameter values through expressions or by reading them in a Script task. Parameter values are set before package execution begins, and their value cannot be changed while the package is running.

Parameters can be defined at the package level and at the project level. Package-level parameters are visible only to tasks and components within that package—much like package variables. Package parameters are defined in the `$Package` namespace. Parameters defined at the project level are global—all packages within the project are able to make use of them. Project parameters are defined in the `$Project` namespace.

Figure 17-1 shows the new Parameters tab in SQL Server Data Tools for Business Intelligence (SSDT-BI), which displays parameters defined at the package level. In addition to the standard properties you'd find on a package variable (such as Name, Type, and Value), the Parameters tab expose three new properties: Description, Sensitive, and Required.

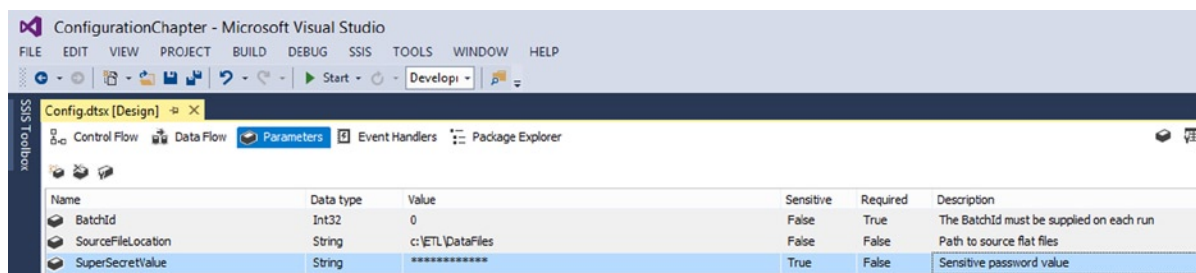


Figure 17-1. Package-level parameters are created and displayed on their own tab in SSDT-BI

The Description field provides an easy way for the SSIS package developers to document the arguments for their packages. It's recommended that you provide descriptions for your parameters, especially in cases where the person running or configuring the packages is not the same person who developed them.

If a parameter is marked as Sensitive, its value will be stored in an encrypted format within the package (or it will not be stored at all, depending on the package's `ProtectionLevel` setting). Its value will also be masked when it is displayed in the UI and will not be displayed in execution logs. Sensitive parameters can only be used in expressions for properties that are marked as Sensitive (such as the Password property of a connection manager). Sensitive parameter values can also be retrieved in a Script task or script component with the `Variable.GetSensitiveValue()` method.

Parameters that are marked as Required must have their values specified at runtime. All parameters (and variables) need values set at design time for validation purposes. Required parameters will not use this design time value when the package runs—a new value must be specified by the caller (i.e., SQL Server Agent or the parent Execute Package task). If a parameter's Required property is set to False, the parameter becomes optional—its design time value will be used if no other value is supplied. Parameters that have no logical default value (such as a BatchID or path to an input file) should be marked as Required.

Project-level parameters can be found by accessing the new node in the Solution Explorer (as shown in Figure 17-2). Project parameters appear in their own node because they are stored in a separate file (`Project.params`) within the solution directory. Double-clicking this node brings up the same parameter designer used for package parameters with all of the same properties and options.

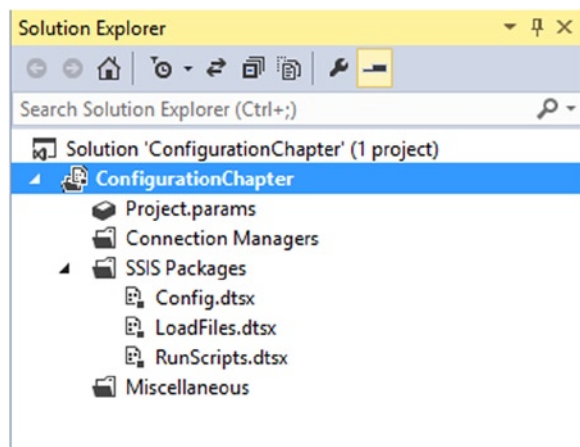


Figure 17-2. Project-level parameters can be found in the `Project.params` node in Solution Explorer

Configuring Your Package Using Parameters

Parameter values are used in your package via SSIS Expressions. Expressions can be set on most task properties, variables, and certain component properties in a Data Flow task. To set an expression on a task, open the Property Expressions Editor dialog (shown in Figure 17-3) by clicking on the expression's property in a task's Properties window.

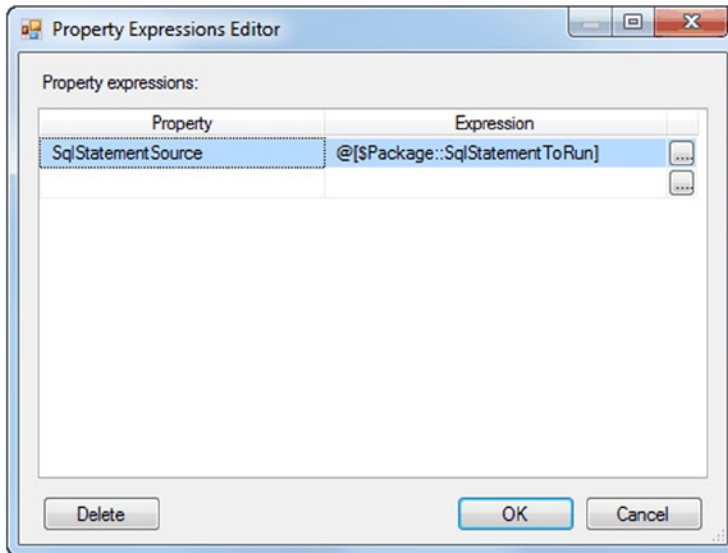


Figure 17-3. The Property Expressions Editor dialog shows all properties that have expressions set on them

Expressions can be set on variables directly from the Variables window (as shown in Figure 17-4). In SQL Server 2012, adding an expression to a variable automatically sets its `EvaluateAsExpression` property to `True`—in previous versions of the product, you had to perform this step yourself. You can disable expression evaluation for a variable by setting this property back to `False`.

Name	Scope	Data type	Value	Expression
 ConnectionString	1 Designer	String	Data Source=.;Initial Catalog=Test;Provide...	
 DatabaseName	1 Designer	String	Test Value	
 CurrentDate	1 Designer	DateTime	9/30/2011 5:12 PM	GETDATE()
 IntVariable	1 Designer	Int32	0	
 SqlStatement	1 Designer	String	SELECT 1+1	

Figure 17-4. Expressions can be set directly from the Variables window in SQL Server 2012. Variables that have an expression set on them appear with a special icon

Figure 17-4 also shows a new feature in SQL Server 2012—expression adorners. The icons for tasks, connection managers, and variables will change if any of the object's properties are set via expression, providing a visual way for a developer to identify which parts of a package are being set dynamically.

Setting expressions for data flow components is less straightforward than setting them on tasks. The main differences are that the expressions are set on the Data Flow task itself and that not all component properties are expressible. Figure 17-5 shows how expressible properties on a Lookup transform “bubble up” and appear as properties on the Data Flow task.

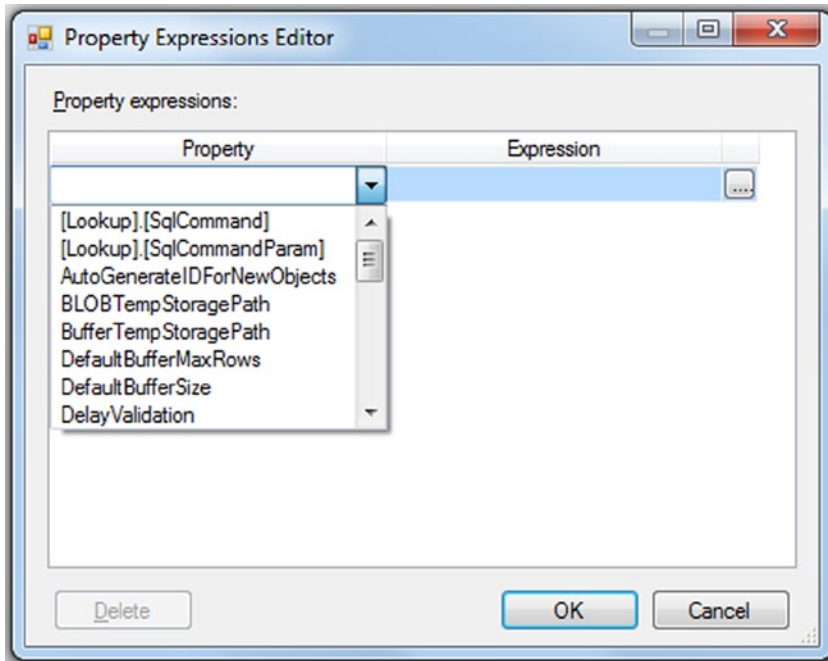


Figure 17-5. Expressible data flow component properties will show up as properties on the Data Flow task

Package and project-level parameters will appear in all of the UIs that display the list of available variables. On the Expression Builder dialog (Figure 17-6), all parameters appear under the Variables and Parameters folder.

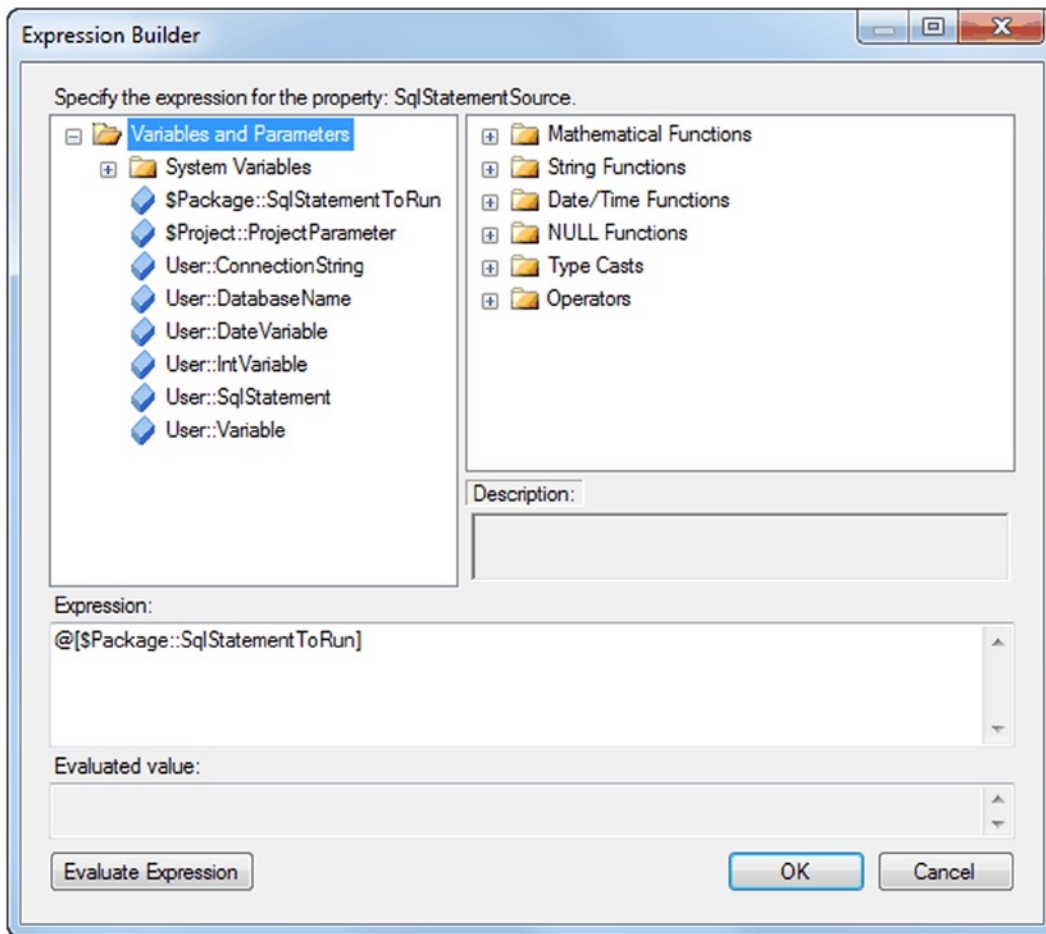


Figure 17-6. Parameters appear alongside variables in the Expression Builder dialog

Certain tasks and data flow components are able to make use of variable and parameter values without the use of expressions. For example, the OLE DB Source provides a “SQL command from variable” data access mode that allows you to set the source query from a variable. Parameters can be used instead of variables for all such properties.

Using the Parametrize Dialog

SSIS provides a Parameterize UI (shown in Figure 17-7) that acts as a shortcut for making use of parameters in your packages. From this UI, you can create a new parameter or use one that already exists. To launch the Parameterize UI, right-click on the task, container, or control flow, and select Parameterize from the context menu. When you click OK, SSIS will automatically add an expression to the selected property.

Parameterize - Mark File as Processed

Specify the property to parameterize and the parameter to use. An expression referencing the parameter will be created on the property.

Property: SqlStatementSource

☐ Do not use parameter

☐ Use existing parameter

☒ Create new parameter

Name: MarkFileAsProcessed_SqlStatementSource

Description:

Value: UPDATE [PackageConfiguration] SET [processed] = 1 WHERE id = ?

Scope:

☒ Package

☐ Project

☐ Sensitive

☐ Required

Help OK Cancel

Figure 17-7. The Parameterize UI is a shortcut for making use of parameters in your package

Creating Visual Studio Configurations

You can use Visual Studio configurations to create multiple sets of parameter values within SSDT-BI. Switching between configurations allows you to easily change parameter values during development and also allows you to build multiple versions of your project deployment file with different default parameter values. Visual Studio configurations are a way for developers to maintain their own settings in multideveloper or team environments.

When you first create a project within SSDT-BI, you will have a default configuration called Development. You can create additional configurations from the Configuration Manager dialog (shown in Figure 17-8). You can launch the Configuration Manager dialog from the Solution Configurations combo box on the Standard toolbar or by right-clicking on the project node in the Solution Explorer, selecting Properties, and clicking the Configuration Managers button. To create a new configuration, select the <New...> option from the Active Solution Configuration dropdown.

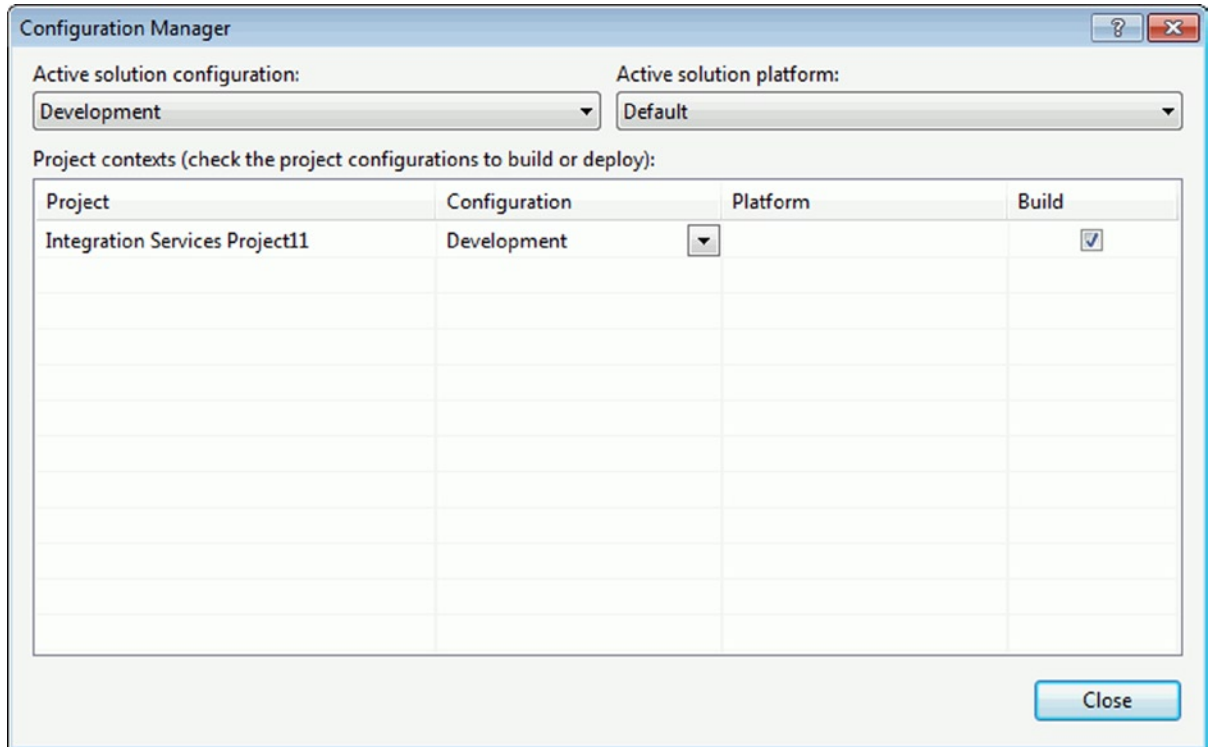


Figure 17-8. Visual Studio configurations can be managed from the Configuration Manager dialog

To add a parameter to a configuration, click the Add Parameters to Configurations button on the package Parameters tab. Figure 17-9 shows the Manage Parameter Values dialog that will be displayed when you add package parameters to configurations. Clicking the Add button allows you to select a parameter—once a parameter is added, it will appear in all configurations in the solution. The Remove button will remove the selected parameter from configurations (which means it will always have the same default value at design time). The Sync button will apply the same value to all configurations—use this button as a shortcut when you’re sure that the parameter’s default value should change across all configurations. Currently you can only add package parameters and project parameters to Visual Studio configurations, but they are configured from separate dialogs. To manage project-level parameters, click the Add Parameters to Configurations button from the Project Parameters designer (Project.params). To manage a connection manager’s settings with Visual Studio configurations, you will first need to parameterize the Connection Manager. Shared connection managers cannot be configured using Visual Studio configurations.

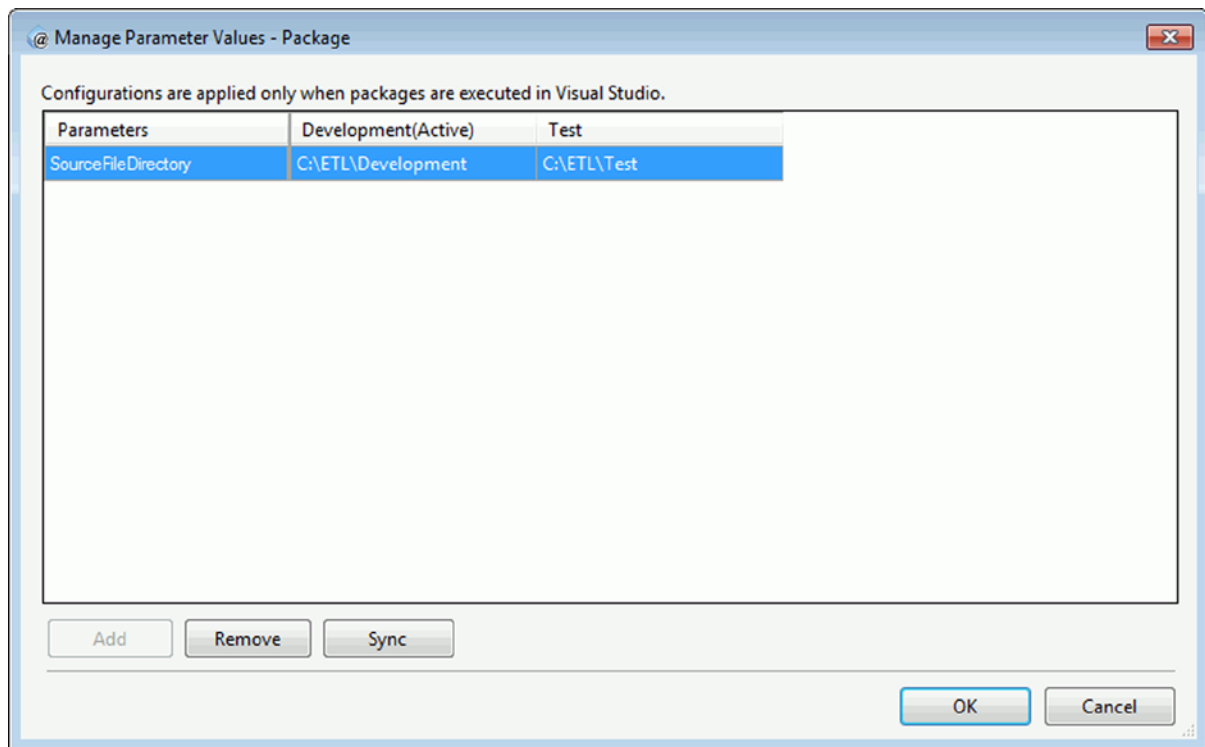


Figure 17-9. The Manage Parameter Values dialog displays all parameters currently set via configurations

■ **Note** When a parameter is controlled by Visual Studio configurations, its value is saved out to the Visual Studio project file (.dtproj). Be sure to save the project file after making updates to your configurations to make sure that you do not lose the changes.

Specifying Entry-Point Packages

SQL Server 2012 introduces another new concept for SSIS—the *entry-point package*. This feature allows the package developer to indicate that special attention should be paid to certain packages. This is very useful in projects that contain a small number of master packages that run a number of child packages. Note that packages that are not marked as entry-point packages can still be run—the setting is meant to be a hint for the person configuring parameter values in the SSIS Catalog. Most SSIS UIs in SQL Server Management Studio (SSMS) allow you to quickly filter out parameters on non-entry-point packages, allowing you to view only the parameters they need to set.

Packages are marked as entry points by default. To remove this setting, right-click on the package name in the Solution Explorer, and unselect the Entry-Point Package option.

Connection Managers

Most connection managers will require some form of configuration, and in SQL Server 2012, all connection manager properties are configurable when packages are run through the SSIS Catalog. Since these properties are already exposed, in most cases, you will not need to expose additional parameters for your connection managers. However, you may encounter some scenarios where parameterized connection managers will be beneficial. Note that any connection manager property that is set via expression will not be exposed through the SSIS Catalog, which prevents a DBA from accidentally overriding property values that are set at runtime.

■ **Note** In previous versions of SQL Server Integration Services, it was common for child packages to configure connection managers with variable values from the parent package. You may wish to keep this pattern in SQL Server 2014 if the connection string is determined at runtime; however, in many cases, you'll want to use shared connection managers instead.

Parameters can be set on connection managers using property expressions. The most common property to set via expression is the `ConnectionString`, because many connection managers derive their properties by parsing the `ConnectionString` value. When configuring connection managers, be sure to set expressions on either the `ConnectionString` or individual properties—the order in which expressions are resolved cannot be guaranteed, and certain properties may be overwritten when the `ConnectionString` is applied.

To parameterize a shared connection manager, open one of the packages in the project and right-click on the shared connection manager's name in the Connection Managers area of the design surface. Note that since shared connection managers are declared at the project level, you can only use project-level parameters or static strings in any property expressions on shared connection managers. The expression dialog will not give you the option to use package parameters or variables.

Parameter Configuration on the Server

Parameters were designed to make it easier for the person scheduling and running SSIS packages. In many environments, this is typically a DBA or IT operations person—not the person who originally developed the package. By including descriptions with the parameters, an ETL developer can create self-documenting packages, making it very easy for whoever is configuring the package to see exactly what it needs to run.

This section describes how to configure packages through the SSIS Catalog and how to surface parameters through SSMS. It covers how to set default parameter values after a project is deployed, the various package execution options, and how the built-in reporting functionality in SQL Server 2012 makes it easier to determine the exact configuration values set when the package was run.

Default Configuration

Default values for all parameters and connection managers are saved within the SSIS project deployment file (.ispac) when the file is built. These become the default values for the project once it is deployed to the SSIS Catalog. To change the default configuration, right-click on the project name (or individual package names) and select **Configure** within the SSMS Object Explorer (as shown in Figure 17-10).

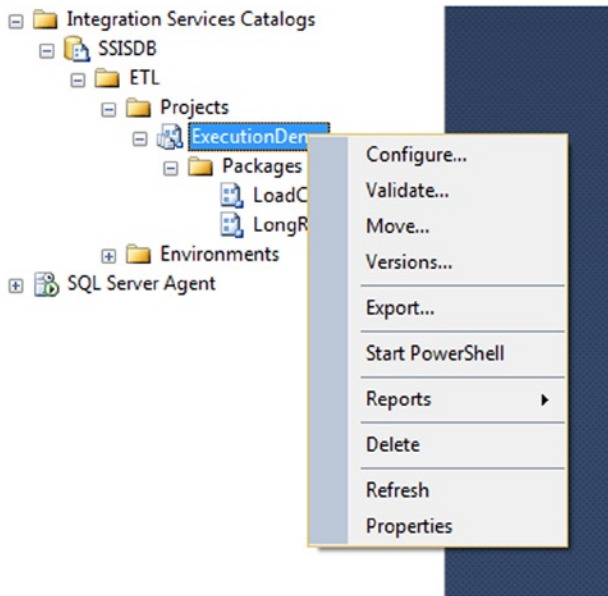


Figure 17-10. The default configuration for a project can be changed through SSMS after the project is deployed

Figure 17-11 shows the parameter configuration dialog in SSMS. Through this dialog, you can set default values for all parameters and connection manager properties for packages within this project. The Scope dropdown allows you to filter your view of the parameters and connection managers. The default view will display entry-point packages only, but you can also view parameters for individual packages and for the entire project. To change the value for a parameter or connection manager property, click the ellipses button at the end of the row. You will have three options when you go to change a value: use the project default, set a literal value, or use a server environment variable. For more information about environments, see the next section.

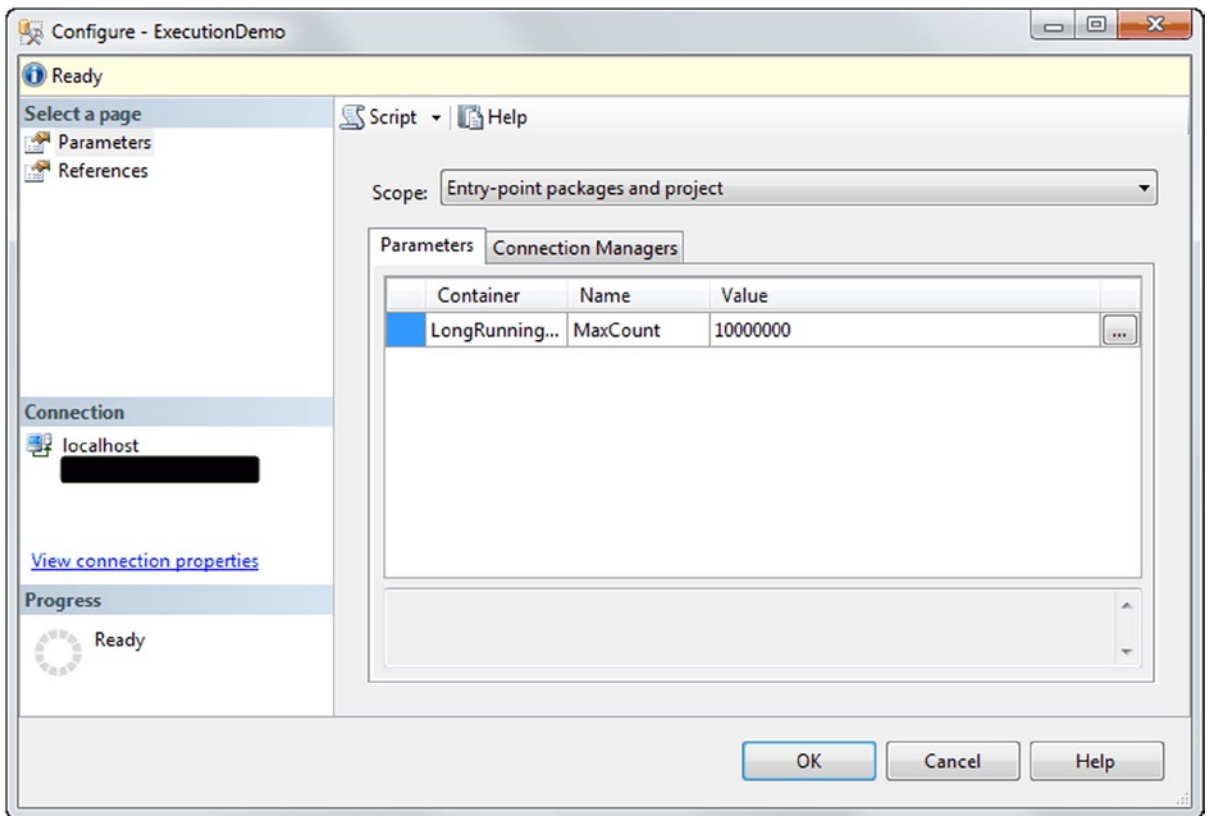


Figure 17-11. Parameter configuration dialog

Server Environments

Server environments contain a set of variables—essentially name-value pairs—that you can map to parameters and connection manager properties within your project. When you run a package through the SSIS Catalog, you can select an environment to run it in. When a value is mapped to a server environment variable, its value will be determined by the environment it is currently running in.

Before you can map a value to a server environment variable, you must associate the environment with the project. Figure 17-12 shows the References page of the project Configure dialog, which allows you to associate a project with one or more environments.

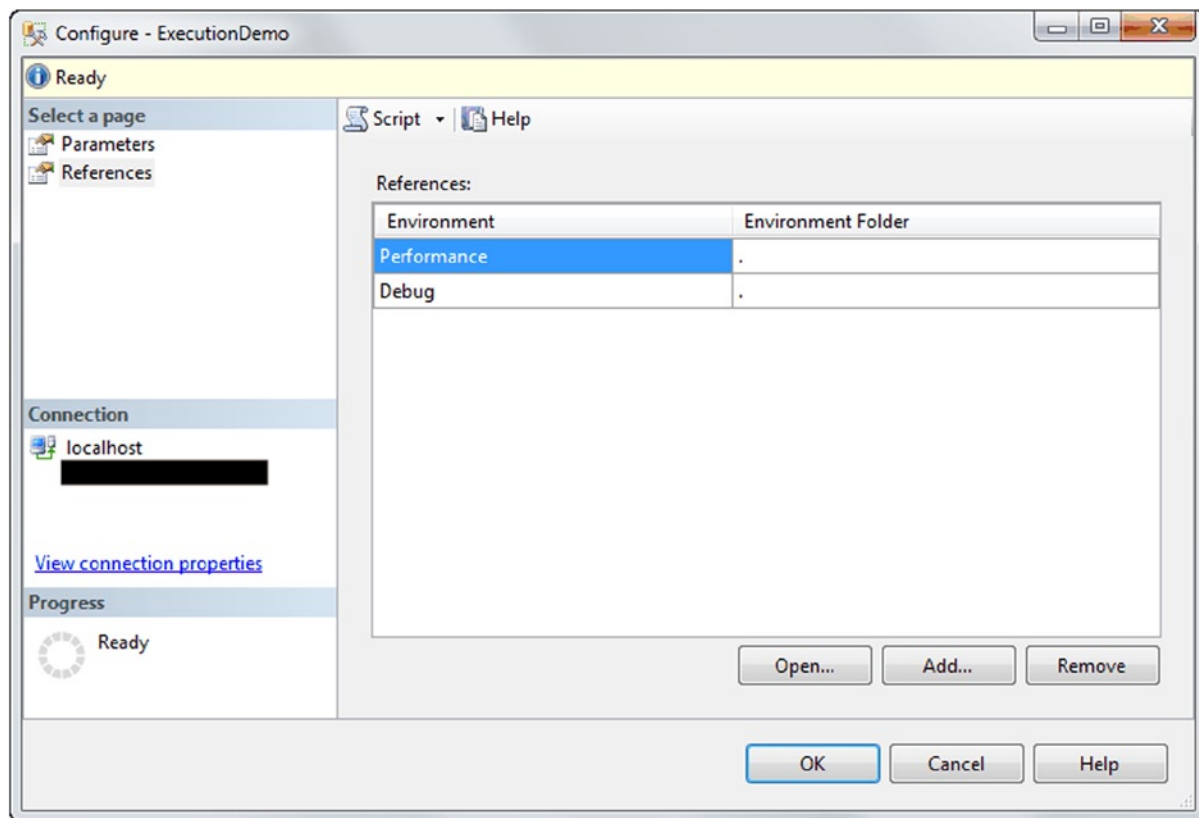


Figure 17-12. The References page of the Configure dialog lets you associate a project with environments

Like projects, environments are contained within a folder in the SSIS Catalog. A project may reference an environment in any folder in the catalog—references are not limited to the current folder only. If you plan to use environments throughout your projects, you might consider creating a separate folder as an area to store all of the common environments.

Environments support row-level security. Like projects and folders, you can configure which users or roles have access to individual environments. Users will not be able to see environments they do not have access to.

Once a project has been associated with one or more server environments, you are able to map parameter and connection manager values to variables contained within those environments.

■ **Note** Environments can contain any number of server variables, and two environments might not contain variables with the same name. If a parameter or connection manager value is mapped to a server variable, only environments that contain a variable with that name (and matching data type!) will be available when you go to select the environment to run the package in.

Default Parameter Values Using T-SQL

Default parameter values and connection manager properties can be set through the SSIS Catalog's T-SQL API. This allows a DBA to automate the setting of parameter values after a deployment or after a project is moved to a new SSIS Catalog. An easy way to create a script is to make the changes through the parameter configuration UI, and then click the Script button. Listing 17-1 shows the T-SQL used to set default values for a two items: a package parameter (MaxCount) is set to 100, and a connection manager property (CM.SourceFile.ConnectionString) is set to 'C:\Demos\Data\RaggedRight.txt'.

Listing 17-1. Setting Parameter Values Using T-SQL

```
DECLARE @var sql_variant = N'C:\Demos\Data\RaggedRight.txt'
EXEC [SSISDB].[catalog].[set_object_parameter_value]
    @object_type=20,
    @parameter_name=N'CM.SourceFile.ConnectionString',
    @object_name=N'ExecutionDemo',
    @folder_name=N'ETL',
    @project_name=N'ExecutionDemo',
    @value_type=V,
    @parameter_value=@var
GO

DECLARE @var bigint = 100
EXEC [SSISDB].[catalog].[set_object_parameter_value]
    @object_type=30,
    @parameter_name=N'MaxCount',
    @object_name=N'LongRunning.dtsx',
    @folder_name=N'ETL',
    @project_name=N'ExecutionDemo',
    @value_type=V,
    @parameter_value=@var
GO
```

■ **Note** For more information, see the `set_object_parameter_value` stored procedure entry in Books Online: [http://msdn.microsoft.com/en-us/library/ff878162\(sql.110\).aspx](http://msdn.microsoft.com/en-us/library/ff878162(sql.110).aspx).

Package Execution Through the SSIS Catalog

Default values for parameters and connection manager properties can be overridden when a package is executed. The Execute Package UI in SSMS (shown in Figure 17-13) allows you to specify the values to use for that specific execution of the package. Project and package-level parameters are displayed on the Parameters tab and shared connection managers and package-level connection managers are shown in the Connection Managers tab. The Advanced tab allows you to override property values that were not exposed as parameters. This feature—called Property Overrides—allows a DBA to make a quick configuration change to a value within a package without having to redeploy the entire project. The functionality is similar to using the `/Set` command line option with the DTEXEC utility.

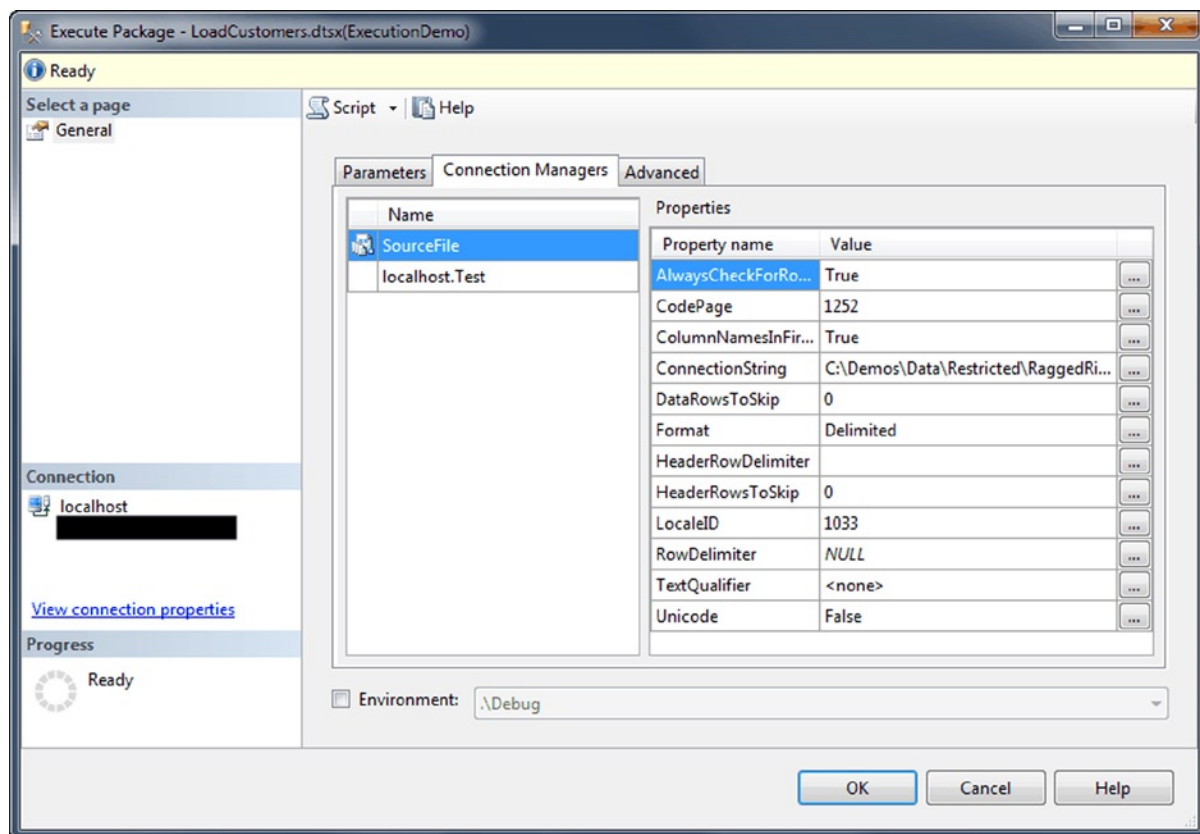


Figure 17-13. Interactive package execution through SSMS

The Execute Package UI also has a Script menu which allows you to script out the creation of a package execution to T-SQL. Listing 17-2 provides an example of creating a new package execution and overriding a number of settings. This procedure involves a number of steps:

1. Create a new execution instance using `[catalog].[create_execution]`.
2. Override parameter or connection manager values using `[catalog].[set_execution_parameter_value]`.
3. Set property overrides using `[catalog].[set_execution_property_override_value]`.
4. Start the package execution using `[catalog].[start_execution]`.

Listing 17-2. Running a Package Using T-SQL

```

-- Create the package execution
DECLARE @exec_id bigint
EXEC [SSISDB].[catalog].[create_execution]
    @execution_id=@exec_id OUTPUT,
    @package_name=N'LoadCustomers.dtsx',
    @folder_name=N'ETL',
    @project_name=N'ExecutionDemo',
    @use32bitruntime=0

-- Set a new value for the AlwaysCheckForRowDelimiters property of the
-- SourceFile connection manager
EXEC [SSISDB].[catalog].[set_execution_parameter_value]
    @execution_id=@exec_id,
    @object_type=20,
    @parameter_name=N'CM.SourceFile.AlwaysCheckForRowDelimiters',
    @parameter_value=0

-- Set the logging level for this execution
EXEC [SSISDB].[catalog].[set_execution_parameter_value]
    @execution_id=@exec_id,
    @object_type=50,
    @parameter_name=N'LOGGING_LEVEL',
    @parameter_value=1

-- Create a property override for the MaxConcurrentExecutables property
EXEC [SSISDB].[catalog].[set_execution_property_override_value]
    @exec_id,
    @property_path=N'\Package.Properties[MaxConcurrentExecutables]',
    @property_value=N'1',
    @sensitive=0

-- Start the package execution
EXEC [SSISDB].[catalog].[start_execution] @exec_id

-- Return the execution ID
SELECT @exec_id

GO

```

The Integration Services job steps in SQL Server Agent have been enhanced in SQL Server 2012 to support running packages stored in an SSIS Catalog. The user interface is the same as when you run a package interactively through SSMS, and it provides the same configuration options. Alternatively, you can run SSIS packages using the T-SQL job step. However, since this step does not support the use of proxy accounts, you will be limited to running the packages as the SQL Server Agent service account.

Parameters with DTEXEC

The command prompt utility to run SSIS packages (DTEXEC) has been updated to support projects and parameters. DTEXEC is able to run packages stored within an SSIS project file (.ispac) as well as start a server-based execution of a package stored within an SSIS Catalog (local or remote). Both modes use different command-line switches to set parameter values and are described in separate sections in the following pages.

Note When working with individual SSIS package files (.dtsx), DTEXEC behaves the same as it did in previous versions of SQL Server. For more information on the various command-line options for DTEXEC, see its entry in Books Online: <http://msdn.microsoft.com/en-us/library/ms162810.aspx>.

Projects on the File System

Although the new project deployment model is primarily meant to be used with the SSIS Catalog, it is possible to run packages within a project file using DTEXEC. Packages run this way are executed locally by the DTEXEC process. Individual parameter values can be set using the /Set option, and /ConfigFile can be used to set a number of parameter values from a 2005/2008-style XML configuration file. Table 17-1 provides a summary of the options related to running packages from projects stored on the file system.

Table 17-1. DTEXEC Command-Line Options for Using Project Files (.ispac)

Parameter	Description
Proj[ect]= <i>path_to_project</i>	This option provides the path to the SSIS project file (.ispac). Example: /Proj c:\demo\project.ispac
Pack[age]= <i>package_name</i>	The name of the package within the project file you want to run. The value should include the .dtsx extension. Example: /Pack MyPackage.dtsx
Set= <i>parameter_name;value</i>	This option allows you to set a value for a parameter within the project. The syntax is similar to what you'd use to override package variable values on the command line. Use the \$Project namespace to set values for parameters defined at the project scope, and \$Package for parameters defined at the package scope. Example: /Set=\Package.Variables[\$Project::IntParameter];1
Conf[igFile]= <i>path_to_file</i>	This option allows you to set multiple parameter values from an XML configuration file. The syntax for each parameter value is similar to what is used for the /Set option. Example: /Conf parameters.xml

Listing 17-3 provides an example of running a package (MyPackage.dtsx) contained within a project file (project.ispac). It sets the values for two parameters—BatchNumber, an integer parameter defined at the project level, and HostName, a string parameter defined at the package level.

Listing 17-3. Running Packages Within a Project File Using DTEXEC

```
dtexec.exe /Project c:\demo\project.ispac /Package MyPackage.dtsx /Set
\Package.Variables[$Project::BatchNumber];432 /Set
\Package.Variables[$Package::HostName];localhost
```

■ **Note** Although the syntax for setting parameter values is similar to setting values for variables and other package properties, there is one key difference. To set parameter values, you should not include the name of the property—you only specify the name of the parameter itself.

Projects in the SSIS Catalog

DTEXEC has been extended in SQL Server 2012 to support running packages contained within an SSIS Catalog. Unlike other execution modes, when running a package from a catalog, the execution takes place on the SSIS Catalog's server and not by the DTEXEC process. In this mode, you will use the `/IServer` command-line option to specify the path to the package you want to run, the `/Parameter` option to set parameter values, and the `/EnvReference` option if you wish to run your package in a specific server environment. Table 17-2 contains a full list of command-line options for SSIS Catalog-based execution with DTEXEC.

Table 17-2. DTEXEC Command-Line Options for the SSIS Catalog

Parameter	Description
Ser[ver]= <i>server_instance</i>	The name of the SQL instance containing the SSIS Catalog. If this option is not specified, the default instance on the localhost is assumed. Example: /Ser ETLSERVER1
IS[Server]= <i>path_to_package</i>	The path of the package in the SSIS Catalog. This will contain the name of the catalog (SSISDB), the folder name, the project name, and the name of the package you want to run. This option cannot be used with the <code>/DTS</code> , <code>/SQL</code> , or <code>/FILE</code> options. Example: /IS \SSISDB\MyFolder\ETLProject\MyPackage.dtsx
Par[ameter]= <i>name[(type)];value</i>	Set a value for the given parameter. Include the namespace of the parameter along with the name to distinguish parameter scope (<code>\$Project</code> for project level parameters, <code>\$Package</code> for package level parameters, <code>\$CM</code> for connection manager properties, and <code>\$ServerOption</code> for server specific options). If the namespace is not included, the parameter is assumed to be at the package scope. Example: /Par \$Project::BatchNumber;432
Env[Reference]= <i>environment_id</i>	This option allows you to specify a server environment to use when running a package. Any parameter values that have been bound to server environment variables will be resolved automatically. To get the ID for an environment, query for its name in the <code>[catalog].[environments]</code> view in SSISDB. Example: /Env 20

Listing 17-4 provides an example of running a package (`MyPackage.dtsx`) contained within a project (`ETLProject`) in a folder (`MyFolder`) on a remote SSIS Catalog server (`ETLServer`). It sets the values for two parameters—`BatchNumber`, an integer parameter defined at the project level, and `HostName`, a string parameter defined at the package level. It also sets the `SYNCHRONIZED` server option to `True`, which tells `DTEXEC` to run in a synchronous mode—more details on synchronous vs. asynchronous execution will come in the following pages.

Listing 17-4. Running Packages Within an SSIS Catalog Using `DTEXEC`

```
C:\>dtexec.exe /Ser ETLServer /IS \SSISDB\MyFolder\ETLProject\MyPackage.dtsx /Par
$Project::BatchNumber;432 /Par $Package::HostName;localhost /Par
"$ServerOption::SYNCHRONIZED(Boolean)";True
```

```
Microsoft (R) SQL Server Execute Package Utility
Version 12.0.2000.8 for 64-bit
Copyright (C) Microsoft Corporation. All rights reserved.
```

```
Started: 4:46:44 PM
```

```
Execution ID: 4.
```

```
To view the details for the execution, right-click on the Integration Services Catalog, and open the
[All Executions] report
```

```
Started: 4:46:44 PM
```

```
Finished: 4:49:45 PM
```

```
Elapsed: 3 seconds
```

■ **Note** You must use Windows Authentication to connect to your SQL Server instance when you are running packages contained in an SSIS Catalog. The `/User` and `/Password` command-line options cannot be used with the `/ISServer` option. If you need to impersonate another user account, you can use the `RunAs` DOS command with `DTEXEC`.

When you run an SSIS Catalog package with `DTEXEC`, it will run in an asynchronous mode by default. This means that the process will return immediately and will not tell you whether the package actually ran successfully. To get synchronous execution behavior (e.g., the same that you would get when running packages from the file system or MSDB), you need to include the `/Par "$ServerOption::SYNCHRONIZED(Boolean)";True` command-line switch. When synchronous execution is used, the `DTEXEC` process will not return until the package has finished running.

Another difference between the SSIS Catalog and other forms of `DTEXEC` execution is that the events that occur while the package is running are not displayed on the command line. Listing 17-4 shows a sample output from running a package in the SSIS Catalog—as you can see, there is only a single message telling you the server execution ID and pointing you to the catalog reports.

Dynamic Configurations

Parameters on an entry-point package allow a user to specify values, but they require that the values be known before the package starts running. There may be times where you'll need to determine configurations at runtime or dynamically pull in values from other sources (such as an external file or a database table). The following sections provide design patterns that you can use to augment the capabilities provided by the parameter model.

Configuring from a Database Table

The SSIS Catalog provides a central location for package configuration values, but your environment may already have alternative locations that store metadata that your packages need at runtime. This pattern shows you how to retrieve values from a database table using an Execute SQL task and how to configure properties within the package using property expressions. For this example, you'll be reading a directory and file name from a database, storing the values in variables, and then using them to dynamically set the `ConnectionString` for a flat file connection manager.

Creating the Database Table

Listing 17-5 shows the SQL for the table that you will be reading your configuration values from. Each row in the table is a new flat file that you will want to process with this package. The two main columns you are interested in are `directory` and `name`—the `id` column is a surrogate key to uniquely identify each row in the table, and the `processed` column lets us easily filter out files that have already been processed. Sample values are shown in Table 17-3.

Listing 17-5. SQL Definition of the Table Our Package Will Read Its Configuration Values From

```
CREATE TABLE [dbo].[PackageConfiguration]
(
    [id] int IDENTITY(1,1) NOT NULL,
    [directory] nvarchar(255) NOT NULL,
    [name] nvarchar(255) NOT NULL,
    [processed] bit NOT NULL
)
```

Table 17-3. Sample Rows from the PackageConfiguration Table

ID	Directory	Name	Processed
1	C:\ETL\Development	File1.txt	False
2	C:\ETL\Development	File2.txt	False
3	C:\ETL\Test	File1.txt	False

Retrieving Configuration Values with an Execute SQL Task

You will retrieve the list of files you need to process from the `PackageConfiguration` table you created using an Execute SQL task. You will store the result set in a package variable, and then loop through each row with a `ForEach` loop container. You will use the `processed` field to mark the files that have already been processed—you will set the `processed` value to `True` once you have successfully loaded the file.

■ **Note** This example assumes that all of the flat files listed in the `PackageConfiguration` table have the same schema. It does not cover the logic needed to actually load the flat file into the database—it is meant to illustrate the pattern that you'd use as a template for processing a number of items in a loop.

Setting up the package takes the following steps:

1. Add four package variables.
 - **FileID (Int32):** The row ID for the file you are currently processing
 - **Directory (String):** The directory containing the flat file you need to process
 - **FileName (String):** The name of the file you are processing
 - **FilesToProcess (Object):** The result set of the Execute SQL task
2. Add an Execute SQL task to your package; name it **Retrieve File List**.
3. Double-click the task to open its editor.
4. Ensure the `ConnectionType` is OLE DB.
5. Click on the Connection dropdown and select New connection....
6. Click New and configure the connection manager to point to the database containing the `PackageConfiguration` table.
7. Select all of the files that have not been processed from the `PackageConfiguration` table (as shown in Listing 17-6).

Listing 17-6. Query to Pull Out All Entries in the Configuration Table That Have Not Been Processed Yet

```
SELECT * FROM [dbo].[PackageConfiguration] WHERE [processed] = 0
```

8. Set the `ResultSet` value to Full Result Set. This means that the Execute SQL task will retrieve the values as an ADO Recordset that can be processed by the Foreach loop. Note that you could also use an ADO.NET connection manager here, which would cause the results to be returned as an ADO.NET data table.
9. Click on the Result Set tab.
10. Click Add, and use these mappings:
 - a. **Result Name:** 0
 - b. **Variable Name:** `User::FilesToProcess`
11. Click OK to save the changes to the Execute SQL task.
12. Add a Foreach loop container to your package.
13. Connect the Execute SQL task to the Foreach loop container.
14. Add a Data Flow task inside of the Foreach loop container.
15. Add a new Execute SQL task inside of the Foreach loop container.
16. Connect the Data Flow task to the Execute SQL task.
17. Double-click the Execute SQL task to open its editor.
18. Set the connection to the same connection manager you created in step 5.

19. Listing 17-7 shows the SQLStatement to mark a row in the table as processed. Note that the statement contains a parameter marker (the question mark). You will map a variable value to this parameter in the next step.

Listing 17-7. SQL Statement to Mark the File as Processed

```
UPDATE [dbo].[PackageConfiguration] SET [processed] = 1 WHERE id = ?
```

20. Click the Parameter Mapping tab.
21. Click Add, and use these mappings:
 - a. **Variable name:** User::FileID
 - b. **Data Type:** LONG
 - c. **Parameter Name:** 0
22. Click OK to save the changes to the Execute SQL task.
23. Add a new Flat File connection manager, and point it to an existing flat file.
24. Right-click on the Flat File connection manager, and select Properties.
25. Select the Expression property, and bring up the Property Expression editor.
26. Set an expression on the ConnectionString property that makes use of the variable values retrieved from the PackageConfiguration table. Listing 17-8 provides an example of the expression.

Listing 17-8. Expression to Set the Path to the Input File on the ConnectionString Property

```
@[User::Directory] + "\\\" + @[User::FileName]
```

Your package should now look like Figure 17-14.

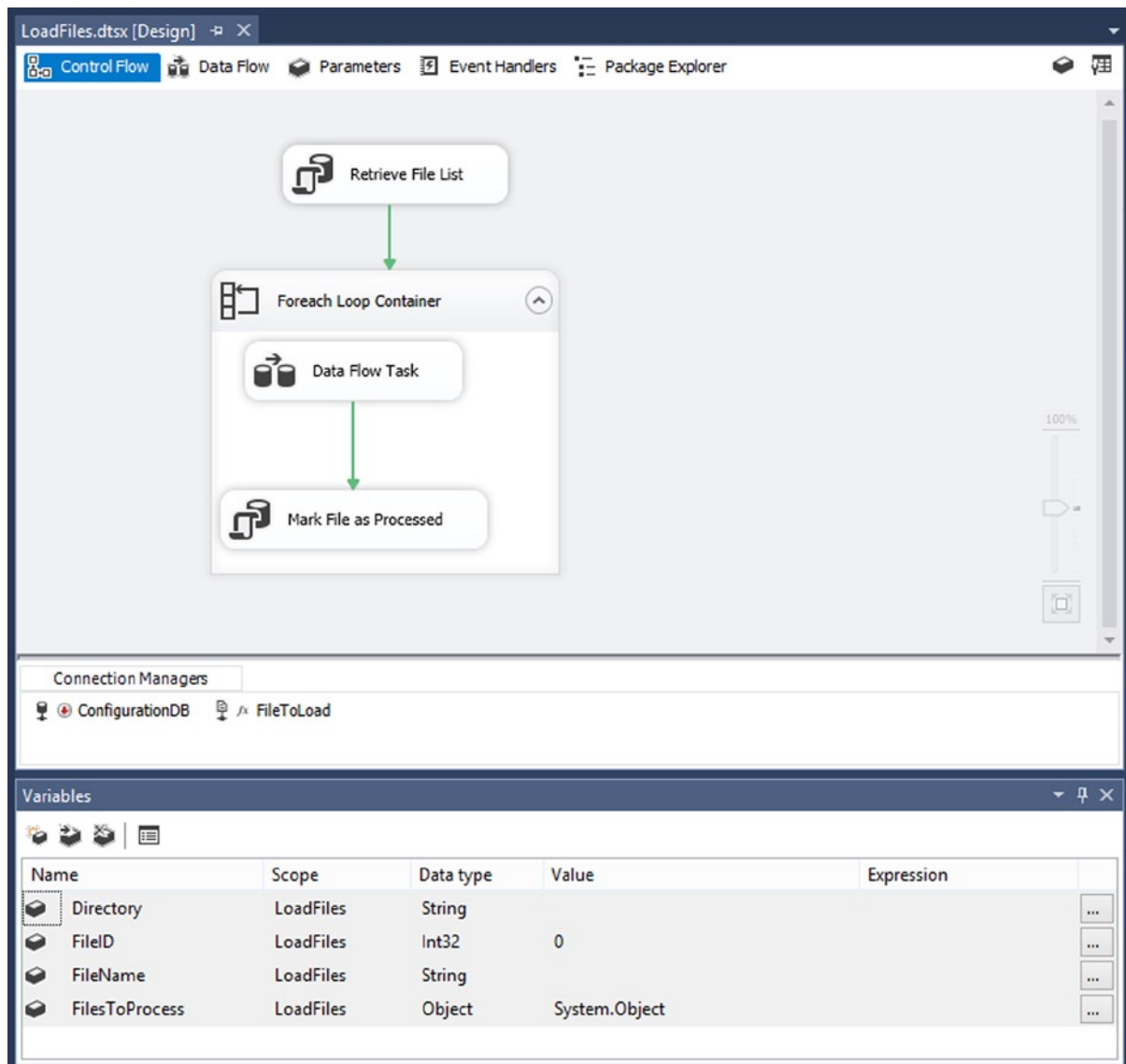


Figure 17-14. Package configured for Execute SQL task–based dynamic configurations

Setting Values Using a Script Task

An alternative to retrieving your configuration with an Execute SQL task and setting package properties through expressions is to use a Script task. This approach can be useful if your values aren't coming from a database or they require additional processing logic—for example, if they are coming from an encrypted source. From within a Script task, you can easily read values from external configuration files (such as an XML file) and access shared configuration resources that might be used by other, non-SSIS parts of your data integration solution. The Script task is able to read and modify package properties at runtime, including the variable values and all connection manager properties.

Listing 17-9 provides sample code of a Script task that sets a connection manager's `ConnectionString` at runtime.

Listing 17-9. Sample Code to Set Package Properties Using a Script Task

```
public void Main()
{
    // TODO: This would be set from an external configuration file
    const string SourceSystemConnectionString = "...";

    Dts.TaskResult = (int)ScriptResults.Success;

    if (Dts.Connections.Contains("SourceSystem"))
    {
        ConnectionManager cm = Dts.Connections["SourceSystem"];
        cm.ConnectionString = SourceSystemConnectionString;
    }
    else
    {
        // The expected connection manager wasn't found - log and set an error status
        Dts.Events.FireError(0, "Script Task",
            "Could not find the SourceSystem connection manager",
            string.Empty, 0);

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

Dynamic Package Executions

In this approach, you will use the same table from Listing 17-5, but instead of reading the configuration values with an SSIS package, you'll use T-SQL to create dynamic package executions on the SSIS Catalog. The code in Listing 17-10 implements the following steps:

1. Declare script variables. Note that in a real-world script, these values would be set through parameters or from an external source.
2. Read the list of files to process from the `PackageConfiguration` table, and store the results in a table variable (`@FileList`).
3. Loop through the list of files. For each file, the code will do the following:
 - a. Retrieve the ID and parameter values from the table variable.
 - b. Create a new SSIS Catalog package execution.
 - c. Set the parameter `Directory` and `FileName` parameter values.
 - d. Start the execution.
 - e. Update the `PackageConfiguration` table to mark that the file has been processed.

Listing 17-10. Dynamic Package Execution Script

```

DECLARE @FolderName NVARCHAR(50) = N'ExecutionDemo'
DECLARE @ProjectName NVARCHAR(50) = N'ETL'
DECLARE @DirectoryParameter NVARCHAR(50) = N'Directory'
DECLARE @FileNameParameter NVARCHAR(50) = N'FileName'
DECLARE @PackageName NVARCHAR(100) = N'LoadCustomers.dtsx'

DECLARE @FileList TABLE
(
    RowNum smallint,
    Id int,
    Directory nvarchar(255),
    Name nvarchar(255)
)

INSERT INTO @FileList (RowNum, Id, Directory, Name)
    SELECT ROW_NUMBER() OVER (ORDER BY id), id, Directory, Name
    FROM [dbo].[PackageConfiguration]
    WHERE processed = 0

DECLARE @maxCount int = (SELECT MAX(RowNum) FROM @FileList)
DECLARE @count int = (SELECT MIN(RowNum) FROM @FileList)

WHILE (@count <= @maxCount)
BEGIN

    DECLARE @Id NVARCHAR(255) = (SELECT Id FROM @FileList WHERE RowNum = @count)
    DECLARE @DirectoryValue NVARCHAR(255) = (SELECT Directory FROM @FileList WHERE RowNum = @count)
    DECLARE @NameValue NVARCHAR(255) = (SELECT Name FROM @FileList WHERE RowNum = @count)

    -- Create the package execution
    DECLARE @exec_id bigint
    EXEC [SSISDB].[catalog].[create_execution]
        @execution_id = @exec_id OUTPUT,
        @package_name = @PackageName,
        @folder_name = @FolderName,
        @project_name = @ProjectName

    -- Set the Directory parameter value
    EXEC [SSISDB].[catalog].[set_execution_parameter_value]
        @execution_id = @exec_id,
        @object_type = 20,
        @parameter_name = @DirectoryParameter,
        @parameter_value = @DirectoryValue

    -- Set the File Name parameter value
    EXEC [SSISDB].[catalog].[set_execution_parameter_value]
        @execution_id = @exec_id,
        @object_type = 20,
        @parameter_name = @FileNameParameter,
        @parameter_value = @NameValue

```

```

-- Start the package execution
EXEC [SSISDB].[catalog].[start_execution] @exec_id

-- Return the execution ID
SELECT N'Started package execution ' + CONVERT(nvarchar(20), @exec_id)

-- Mark the file as processed
DECLARE @UpdateSql nvarchar(1024) = N'UPDATE [dbo].[PackageConfiguration] SET processed = 1
WHERE id = ' + CONVERT(nvarchar(20), @Id)
EXEC sp_sqlexec @UpdateSql

SET @count = @count + 1

END

```

Conclusion

This chapter has covered some of the usage patterns for the new parameter model, as well as some dynamic configuration scenarios. Although the configuration patterns and best practices that were commonly used in SQL 2005 and 2008 continue to work in the latest version of SSIS, most users will see a benefit in migrating to the new model. The clarity of the parameter model was designed to help everyone involved with an SSIS solution life cycle, from those who develop the packages to those who deploy and schedule them.



Deployment

Great strides were made towards simplifying the deployment process for Integration Services projects in SQL Server 2012. Projects within Visual Studio can now target two different deployment models – the Package Deployment Model, which is similar to what was used in previous versions of the product, and the Project Deployment Model, which was designed for the new SSIS Catalog.

This chapter will focus on patterns associated with the new Project Deployment Model and server based deployment. While the Project Model and SSIS Catalog are the recommended way to do deployment, organizations upgrading from previous versions may already have package execution frameworks that rely on file system based deployment.

■ **Note** SSIS package deployment was not changed between SQL Server 2012 and SQL Server 2014.

Project Deployment Model

The new Project Model is the default target when creating SSIS projects in SQL Server 2014. With this model, packages and other project items such as Shared Connection Managers are bundled into a single file with an .ispac extension during the project's Build phase. This file can then be deployed to the SSIS Catalog using the Deployment Wizard, or executed directly using dtexec.exe.

If your project is targeting the Package Deployment Model, you can convert to the Project Deployment Model within Visual Studio. Right click on the project name in the Solution Explorer window, and select Convert to Project Deployment Model (as shown in Figure 18-1). Converting to the Project Deployment Model brings up the Project Conversion Wizard. The wizard helps you convert to the new model by updating Execute Package Tasks to use Project References, and changing Configurations to Parameters.

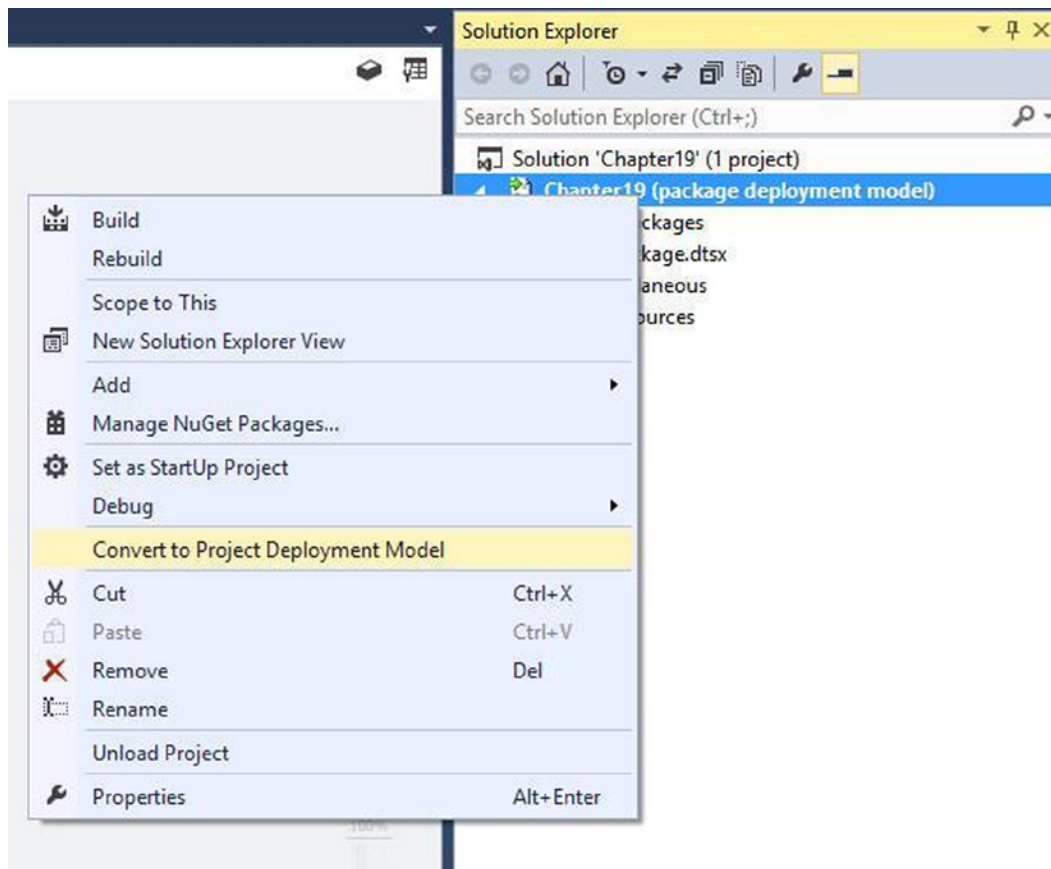


Figure 18-1. SSDT-BI provides an option to convert to the Project Deployment Model

Integration Services projects in the Project Deployment Model can make use of new features such as Parameters, Shared Connection Managers, and Project References. Project References allow the Execute Package Task to locate child packages without the use of connection managers, and greatly simplify the deployment process.

SSIS Catalog

The SSIS Catalog was added in SQL Server 2012, and is the recommended deployment target for Integration Services projects. Deployment to the catalog is typically done using the SSIS Deployment Wizard, which can be launched from within SSDT-BI, SSMS, double clicking an SSIS project file (.ispac) from windows explorer, or by running ISDeploymentWizard.exe.

To launch the Deployment Wizard from SSDT-BI, right click on the project in the Solution Explorer and select the Deploy option. The wizard will automatically load your project file, putting you on the Select Destination page (as show in Figure 18-2).

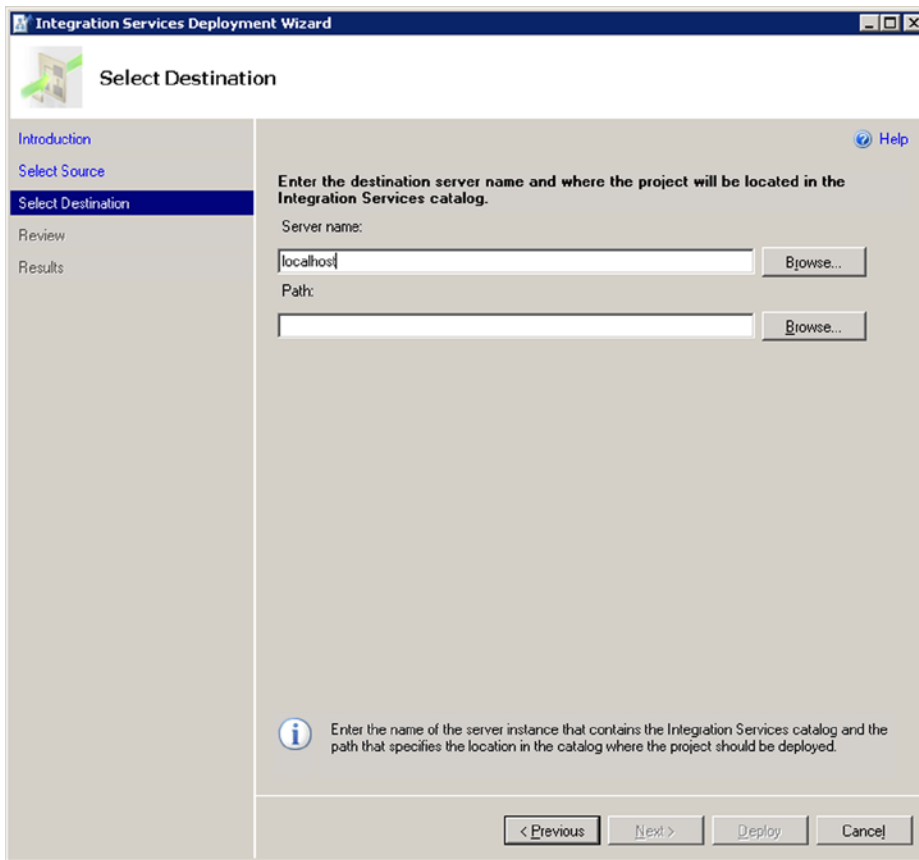


Figure 18-2. The Integration Services Deployment Wizard can be launched from SSDT-BI

Note The Deployment Wizard is typically used to deploy files to the SSIS Catalog, but it can also be used to move projects between servers. To do this, choose the Integration Services catalog option on the Select Source page.

The Deployment Wizard allows you to select the server and folder you wish to deploy the project to. On the final page, the project file is sent to the server and stored in the SSIS Catalog. Note that, during deployment, the wizard indicates that it is changing the project's protection level (Figure 18-3). During this phase, sensitive data within the project is decrypted, and the project is converted to the Server Storage protection level. The server relies on database encryption to protect packages and parameter values – these tables are automatically encrypted in the SSIS Catalog.

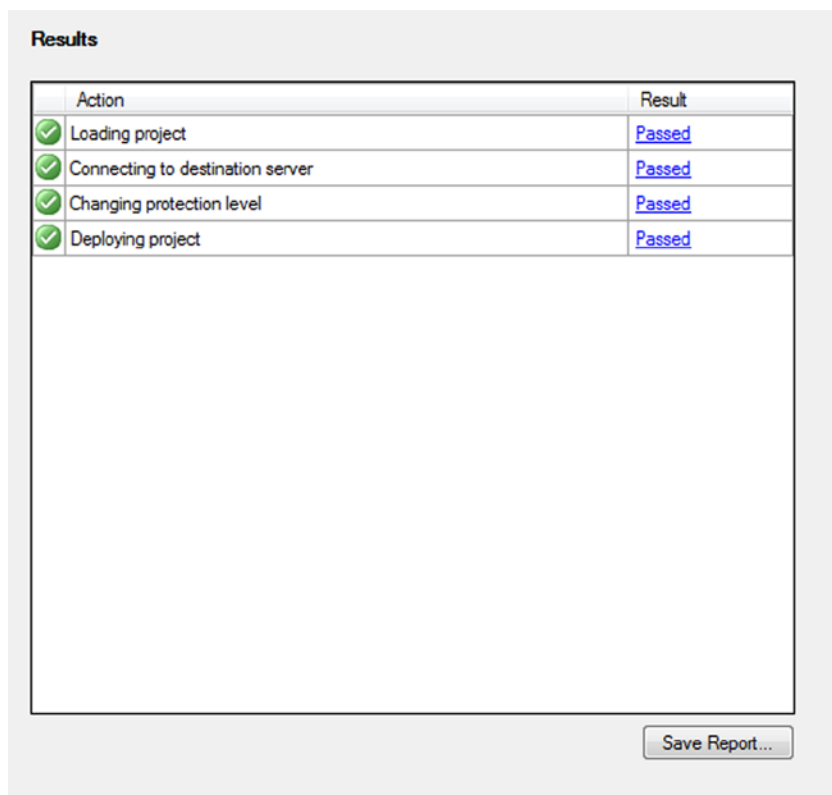


Figure 18-3. The Deployment Wizard status page

■ **Note** More information about package protection levels and secure deployments can be found in Books Online at <http://msdn.microsoft.com/en-us/library/bb522558.aspx>.

Deployment Methods

This section describes the different deployment methods supported by the SSIS Catalog. The method you choose will depend on your environment, and what the people doing the deployment – whether they are developers, ETL operators, or DBAs – are most comfortable with. The deployment methods described here include:

- Deployment from the command line
- Deployment using custom code
- Deployment using PowerShell
- Deployment using SQL

Deployment from the Command Line

The Deployment Wizard (ISDeploymentWizard.exe) provides a command line interface, which allows you to deploy to the SSIS Catalog without a UI. This is very useful for deploying from scripts, or as part of a batch process. Table 18-1 shows the list of supported parameters. Listing 18-1 provides an example command line that deploys a project (C:\ETL\Project.ispac) to a folder named MyFolder on a local SSIS Catalog.

Table 18-1. Integration Services Deployment Wizard command line parameters

Parameter	Short Version	Description
Silent[+ -]	S	When this option is true, the deployment will be done in a UI-less mode (command line only). Use this option when deploying from batch files. The default value is '-', which will display the UI. Example: /Silent+
SourceType:{File Server}	ST	This option specifies whether the source project comes from the file system, or another SSIS Catalog. The default value is "File." Example: /SourceType:File
SourcePath:path_to_project	SP	The path to the .ispac file being deployed (when using the File source), or the path to the project name (when using the Server source). Example: /SourcePath:C:\ETL\project.ispac
SourceServer:server_instance	SS	The name of the server instance when the SourceType is set to Server. Example: /SourceServer:localhost\SQL1
ProjectPassword:password	PP	If the source .ispac file is password protected, this parameter can be used to supply the password. Note that specifying a password on the command line is not recommended, as other users on the system might be able to see the arguments. If your project file is using password encryption, consider specifying the password in the response file (see the @<file> option for more information)
DestinationServer:server_instance	DS	The name of the server instance you are deploying to. Example: /DestinationServer:localhost
DestinationPath:path	DP	The path you want to deploy the project to on the destination server. The format of the path is "/<catalog>/<folder>/<project>". Example: /DestinationPath:/SSISDB/MyFolder/Project
@<file>		This option allows you to specify all of your command line arguments in a text file, instead of entering them directly on the command line. Example: @arguments.txt

Listing 18-1. Deploying a project from the command line

```
ISDeploymentWizard.exe /Silent /SourcePath:"C:\ETL\Project.ispac" /DestinationServer:"localhost" /
DestinationPath:"/SSISDB/MyFolder/Project"
```

■ **Note** When the Deployment Wizard is run in interactive (UI) mode, the Review page displays the equivalent parameters to do a command line based deployment. This can be a handy shortcut – simply copy the command line arguments into a batch file to perform automatic deployments in the future.

Deployment Using Custom Code

The SSIS Catalog has a managed .NET API called the Management Object Model (or MOM). This API allows you to programmatically perform that same management tasks that would normally be done through SQL Server Management Studio (SSMS), including Folder creation, and deployment of projects.

Listing 18-2 provides a sample C# application that makes use of the MOM to create a new Folder in an SSIS Catalog, and deploys a project to it. The core functionality can be found in the `Microsoft.SqlServer.Management.IntegrationServices` assembly, which is installed with SSMS and found in the Global Assembly Cache (GAC), along with all of its dependencies.

■ **Tip** If you can't find the `Microsoft.SqlServer.ManagedDTS.dll` file, then look in the .Net Framework 4.0 Global Assembly Cache (GAC) directory, which often in a default install will be `C:\Windows\Microsoft.NET\assembly`.

Listing 18-2. Deploying a project using the Management Object Model

```
using System;
using System.Collections.Generic;
using System.Linq;
using System.Text;
using Microsoft.SqlServer.Management.IntegrationServices;
using Microsoft.SqlServer.Management.Smo;
using Microsoft.SqlServer.Dts.Runtime;
using System.IO;

class Program
{
    const string ProjectFileLocation = @"C:\ETL\Project.ispac";

    static void Main(string[] args)
    {
        // Connect to the default instance on localhost
        var server = new Server("localhost");
        var store = new IntegrationServices(server);
```

```

// Check that we have a catalog
if (store.Catalogs.Count == 0)
{
    Console.WriteLine("SSIS catalog not found on localhost.");
}

// Get the SSISDB catalog - note that there should only
// be one, but the API may support multiple catalogs
// in the future
var catalog = store.Catalogs["SSISDB"];

// Create a new folder
var folder = new CatalogFolder(catalog,
                               "MyFolder",
                               "Folder that holds projects");

folder.Create();

// Make sure the project file exists
if (!File.Exists(ProjectFileLocation))
{
    Console.WriteLine("Project file not found at: {0}",
                      ProjectFileLocation);
}

// Load the project using the SSIS API
var project = Project.OpenProject(ProjectFileLocation);

// Deploy the project to the folder we just created
folder.DeployProject(project);
}
}

```

Deployment Using PowerShell

The SSIS Management Object Model (MOM) is accessible via PowerShell, which makes it possible to fully automate your deployment (and other management tasks) using PowerShell scripts. Listing 18-3 shows the PowerShell version of the simple deployment application from Listing 18-2.

Listing 18-3. Deploying a project using PowerShell

```

# Variables
$ProjectFilePath = "C:\ETL\Project.ispac"
$ProjectName = "Project"
$FolderName = "MyFolder"

# Load the IntegrationServices Assembly
$loadStatus = [Reflection.Assembly]::Load("Microsoft.SqlServer.Management.IntegrationServices,
Version=12.0.0.0, Culture=neutral, PublicKeyToken=89845dc8080cc91")

# Store the IntegrationServices Assembly namespace to avoid typing it every time
$ISNamespace = "Microsoft.SqlServer.Management.IntegrationServices"

```

```

Write-Host "Connecting to server ..."

# Create a connection to the server
$sqlConnectionString = "Data Source=localhost;Initial Catalog=master;Integrated Security=SSPI;"
$sqlConnection = New-Object System.Data.SqlClient.SqlConnection $sqlConnectionString

# Create the Integration Services object
$integrationServices = New-Object $ISNamespace".IntegrationServices" $sqlConnection
$catalog = $integrationServices.Catalogs["SSISDB"]

Write-Host "Creating Folder" $FolderName "..."

# Create a new folder
$folder = New-Object $ISNamespace".CatalogFolder" ($catalog, $FolderName, "This is a folder
description")
$folder.Create()

Write-Host "Deploying" $ProjectName "project ..."

# Read the project file, and deploy it to the folder
[byte[]] $projectFile = [System.IO.File]::ReadAllBytes($ProjectFilePath)
$project = $folder.DeployProject($ProjectName, $projectFile)

Write-Host "All done."

```

Deployment Using SQL

If you prefer to do all of your database management and deployments using T-SQL, the SSIS Catalog exposes a full management interface through a set of Views and Stored Procedures. Listing 18-4 provides a sample that loads a project file in binary format, deploys it to a folder using the [catalog].[deploy_project] stored procedure, and then queries the status of the deployment from the [catalog].[operations] view.

Listing 18-4. Deploying a project using the SQL API

```

use SSISDB

DECLARE @ProjectBinary as varbinary(max)
DECLARE @OperationID as bigint

-- load the project file
SET @ProjectBinary =
(
    SELECT *
    FROM OPENROWSET
    (
        BULK 'C:\ETL\Project.ispac',
        SINGLE_BLOB
    ) as BinaryData
)

```



```

-- deploy the project
EXEC [catalog].[deploy_project]
    'MyFolder',      -- folder
    'Project',       -- project name
    @ProjectBinary,  -- binary data
    @OperationID out -- operation id

--
-- Get the status of the last deployment
--

DECLARE @LastDeployment_id bigint;
SET @LastDeployment_id =
(
    SELECT MAX(operation_id)
    FROM   [catalog].[operations]
    WHERE  operation_type = 101 -- deploy
)

SELECT [object_name], start_time, end_time, [status], [value] =
    case
        when [status] = 1 then N'Created'
        when [status] = 2 then N'Running'
        when [status] = 3 then N'Canceled'
        when [status] = 4 then N'Failed'
        when [status] = 5 then N'Pending'
        when [status] = 6 then N'Unexpected Termination'
        when [status] = 7 then N'Succeeded'
        when [status] = 8 then N'Stopping'
        when [status] = 9 then N'Completed'
    end
FROM   [catalog].[operations]
WHERE  [operation_id] = @LastDeployment_id

```

Package Deployment Model

SSIS projects created in SQL Server 2012 will default to the Project Deployment Model, but some users may want to continue using the Package Deployment Model from SQL Server 2005 and 2008. You can convert from the Project Deployment Model to the Package Deployment Model in Visual Studio by right clicking on the project name in Solution Explorer, and selecting Convert to Package Deployment Model (as shown in Figure 18-4). Projects that were originally created in previous versions of SQL Server will automatically start off in the Package Deployment Model when you open them in SSDT-BI.

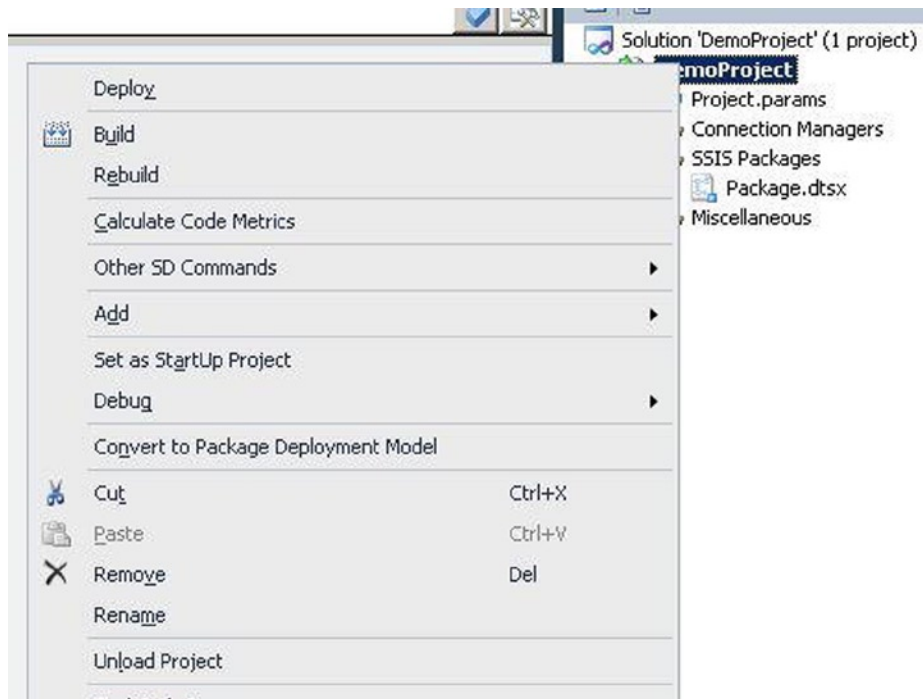


Figure 18-4. Converting to the Package Deployment Model

■ **Note** When using the Package Deployment Model, you will not be able to use some of the new functionality introduced in SQL Server 2012, such as Parameters and Project References. If any of your packages are using these features, SSDT-BI will not let you convert to the Package Deployment Model.

Table 18-2 lists the deployment locations you would use with the Package Deployment Model, and briefly describes the advantages of each approach.

Table 18-2. *Deployment locations when using the Package Deployment Model*

Location	Notes
File System	• Mirrors the structure you have when developing in SSDT-BI • Doesn't require database permissions • Deployment is a simple file copy
SQL Server (MSDB)	• Backup and maintenance part of regular SQL functionality • Finer control over package access and security • Deploys through DTSInstall.exe (legacy deployment wizard), the SSIS object model, or dtutil.exe
Package Store (SSIS Service)	• Provides a façade over the File System and MSDB storage locations, allowing you to change the physical location of a package, yet keep the same logical path • Manages multiple storage locations from a single place • Deploys through SSMS, the SSIS object model, or dtutil.exe • Requires special DCOM permission configuration for access

■ **Note** You cannot use the Package Store interface to manage packages deployed to the SSIS Catalog. The service is only able to interact with packages stored in MSDB (the 2005, and 2008 deployment model), and is there to continue supporting users who have not migrated to the new Project Deployment Model. It may be depreciated in the future.

Conclusion

The deployment process for SSIS packages was greatly simplified in SQL Server 2012. Although the deployment model used in SQL Server 2005 and 2008 (now called the Package Deployment Model) is still fully supported, moving to the new Project Deployment Model is highly recommended for new data integration projects. SSIS provides a number of ways to deploy to the SSIS Catalog, providing the flexibility you need to fit the deployment process into your environment.



Business Intelligence Markup Language

You likely purchased this book to learn how to be a more productive SQL Server Integration Services (SSIS) developer. I applaud your desire and decision, and I sincerely hope the information contained herein has provided ideas and information to help you be more productive. I am always on the lookout for ways to become a better data integration developer. Specifically, I seek out ways to improve code quality and reduce the amount of time required to build solutions. Those goals are what motivated me to begin practicing patterns-based development in the first place, which eventually led to the idea for this book.

Business Intelligence Markup Language—or Biml—represents SSIS packages using XML. By storing metadata that describes SSIS packages in XML, Biml approaches data integration development from the perspective of a domain-specific language. Biml provides another means to materialize SSIS design patterns—something other than an SSIS package library containing template packages. Regardless of which mechanism is used, storing design patterns facilitates code production at a consistent and repeatable quality. That may sound innocuous, but I assure you it is important; and it is one of the primary reasons to use design patterns in the first place.

Biml is a complex language. You would do well to gain an understanding of domain-specific languages, XML, and .NET development before diving into Biml development proper. I will not delve into the underlying architecture of Biml in this chapter. I will show you some of the mechanisms and direct you to the Biml documentation websites: the Biml Language (www.varigence.com/Documentation/Language/Index) and the Biml Api (www.varigence.com/Documentation/Api/Index). I believe this is enough to whet your appetite while I am demonstrating the power of Biml.

A Brief History of Business Intelligence Markup Language

In early 2007, the Microsoft Customer Service and Support (CSS) business incubated a new approach to building business intelligence (BI) solutions. As the organization responsible for managing all frontline customer support interactions, CSS has significant analytical and predictive needs—across data from a wide variety of sources. To accelerate the development of its internal solutions, CSS began developing the Vulcan project, which used an XML-based markup language to describe a subset of SISS packages. This created a model where business intelligence solutions could be developed more rapidly and iteratively by globally distributed teams of BI developers.

After a period of significant success building new BI capabilities, the CSS and the SQL Server product team decided to publish the source code for the Vulcan project on CodePlex to enable customers to try the technology and begin building a community around it (<http://vulcan.codeplex.com>). Feedback from customers showed that they recognized that the approach was powerful and promising, but that the implementation reflected the project's status as an internal tool used to accelerate an operational delivery team. Without documentation and training resources, usability considerations, and additional features, the cost of adopting Vulcan was prohibitive for all but the most determined customers.

In late 2008, Scott Currie, who worked with the Vulcan technology in CSS, founded Varigence, Inc. Varigence created the Business Intelligence Markup Language (Biml), along with tools to enable its design and development. Although Biml didn't directly use any code or technology from Vulcan, the approach taken by the Vulcan project inspired the Varigence team to build Biml as an XML-based markup language with rapid, iterative global team development capabilities in mind.

Biml is now available in proprietary products and open source projects and has been published as an open language specification. Varigence has developed a Biml compiler that enables a wide variety of automation and multitargeting capabilities. Additionally, Varigence offers an Integrated Development Environment (IDE) for Biml called Mist. Mist enables rapid and visual design and debugging features for Biml. The open source BIDS Helper project includes Biml functionality, enabling anyone to write and execute Biml code for free.

In this chapter, we will leverage the free Biml functionality included with BIDS Helper to dynamically generate SSIS packages.

■ **Note** An object containing Business Intelligence Markup Language is a *Biml file*. Biml files are “executed” to generate SSIS packages.

Building Your First Biml File

Before we get started with Business Intelligence Markup Language, you will need to download and install the latest version of BIDS Helper from <http://bidshelper.codeplex.com>. Once it is installed, create a new SSIS solution and project named **Biml2014**. In Solution Explorer, right-click the project name and click Add New Biml File. The new file, `BimlScript.biml`, will be created and assigned to the Miscellaneous virtual folder in Solution Explorer. Double-click the file to open it in the editor.

The file begins with the most basic Biml construct, as shown in Listing 19-1.

Listing 19-1. Initial Biml Code

```
<Biml xmlns="http://schemas.varigence.com/biml.xsd">
</Biml>
```

Add XML so that your Biml file reads as shown in Listing 19-2. As you type, note IntelliSense auto-indents XML tags to produce well-formed code that is easier to read.

Listing 19-2. Biml After Adding Package XML Metadata

```
<Biml xmlns="http://schemas.varigence.com/biml.xsd">
  <Packages>
    <Package Name="TestBimlPackage" ConstraintMode="Parallel">
    </Package>
  </Packages>
</Biml>
```

Save the file, right-click `BimlScript.biml` in Solution Explorer, and then click Generate SSIS Packages. Figure 19-1 shows that a new SSIS package named `TestBimlPackage.dtsx` is created in the project and file system. The package shows up in Solution Explorer as part of this project.

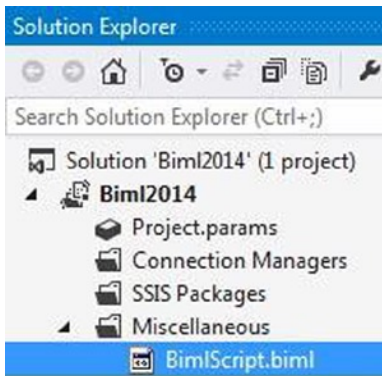


Figure 19-1. *TestBimlPackage.dtsx*

Let's return to the `BimlScript.biml` file and add a task. Create a new XML node beneath the `<Package>` tag named **Tasks**. Between the `<Tasks>` and `</Tasks>` tags, add a new node named **ExecuteSQL**.

■ **Tip** If you are not seeing IntelliSense with Biml, follow this link: <http://bidshelper.codeplex.com/wikipage?title=Manually%20Configuring%20Biml%20Package%20Generator&referringTitle=xcopy%20deploy> for Biml IntelliSense configuration instructions.

Add an attribute to the `ExecuteSQL` root node named `Name` and set its value to **"Test Select"**. Create a new XML node between the `<ExecuteSQL>` and `</ExecuteSQL>` tags named **DirectInput**. Between the `<DirectInput>` and `</DirectInput>` tags, add the T-SQL statement **Select 1 As One**. If you are playing along at home, your `BimlScript.biml` file should look like Listing 19-3.

Listing 19-3. Biml After Adding Initial Metadata Describing an Execute SQL Task

```
<Biml xmlns="http://schemas.varigence.com/biml.xsd">
  <Packages>
    <Package Name="TestBimlPackage" ConstraintMode="Parallel">
      <Tasks>
        <ExecuteSQL Name="Test Select">
          <DirectInput>Select 1 As One</DirectInput>
        </ExecuteSQL>
      </Tasks>
    </Package>
  </Packages>
</Biml>
```

To test, save the file and generate the SSIS package from `BimlScript.biml` in Solution Explorer. (Generate the package by right-clicking on the Biml file and selecting **Generate SSIS Packages** from the context-sensitive menu). Do you get an error similar to that displayed in Figure 19-2? You should get such an error.

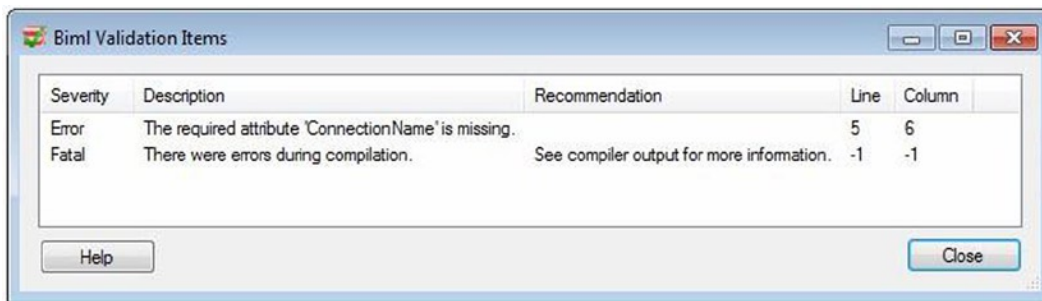


Figure 19-2. Missing *ConnectionName* attribute

The Business Intelligence Markup Language engine includes validation functionality and it caught the error in Figure 19-2. You can invoke a validation from Solution Explorer; simply right-click *BimlScript.biml* and then click Check Biml For Errors.

To fix the error, we need to add a *ConnectionName* attribute to the *ExecuteSQL* tag. But we don't have a connection specified at this time. To create a connection, return to the top of *BimlScript.biml* and add a new line just after the *Biml* tag and before the *Packages* tag. On this line, add the *Connections* XML node. Inside the *<Connections>* and *</Connections>* tags, add a *Connection* XML node. A *Connection* XML node requires two attributes, *Name* and *ConnectionString*. I created a connection to the *AdventureWorks2012* database on the default instance of the local SQL Server. Once the *Connection* metadata is configured, I added a *ConnectionName* attribute to the *ExecuteSQL* tag. My *BimlScript.biml* file now contains the code listed in Listing 19-4.

Listing 19-4. Biml After Adding Connection Metadata

```
<Biml xmlns="http://schemas.varigence.com/biml.xsd">
  <Connections>
    <Connection Name="AdventureWorks2012" ConnectionString=
"Data Source=.;Initial Catalog=AdventureWorks2012;Provider=
SQLNCLI10.1;Integrated Security=SSPI;Auto Translate=False;" />
  </Connections>
  <Packages>
    <Package Name="TestBimlPackage" ConstraintMode="Parallel">
      <Tasks>
        <ExecuteSQL Name="Test Select" ConnectionName="AdventureWorks2012">
          <DirectInput>Select 1 As One</DirectInput>
        </ExecuteSQL>
      </Tasks>
    </Package>
  </Packages>
</Biml>
```

Let's test by regenerating the *TestBimlPackage.dtsx* SSIS package from *BimlScript.biml*. When you attempt to generate the SSIS package, you see a dialog that confirms you would like to overwrite the existing *TestBimlPackage.dtsx* SSIS package. When you confirm this intention, the *TestBimlPackage.dtsx* SSIS package is regenerated from the metadata contained in the updated *BimlScript.biml* file. Open the *TestBimlPackage.dtsx* SSIS package: it should appear as shown in Figure 19-3.

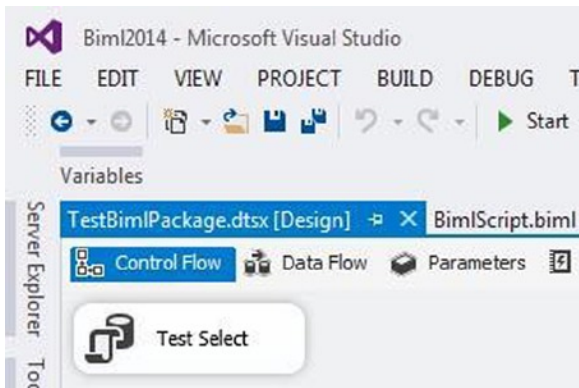


Figure 19-3. A Biml-generated SSIS package

Building a Basic Incremental Load SSIS Package

The incremental load pattern is fundamental in data integration solutions; especially Extract, Transform, and Load (ETL) solutions. Biml provides a mechanism for codifying the incremental load pattern in a repeatable fashion.

Creating Databases and Tables

Let's prepare for this demo by building a couple databases and tables. Execute the T-SQL statements from Listing 19-5 to build and populate the test databases and tables.

Listing 19-5. Building and Populating Demo Databases and Tables

```
Use master
Go

If Not Exists(Select name
              From sys.databases
              Where name = 'SSISIncrementalLoad_Source')
CREATE DATABASE [SSISIncrementalLoad_Source]

If Not Exists(Select name
              From sys.databases
              Where name = 'SSISIncrementalLoad_Dest')
CREATE DATABASE [SSISIncrementalLoad_Dest]
Go
Use SSISIncrementalLoad_Source
Go

If Not Exists(Select name
              From sys.tables
              Where name = 'tblSource')
CREATE TABLE dbo.tblSource
```



```
(ColID int NOT NULL
,ColA varchar(10) NULL
,ColB datetime NULL constraint df_ColB default (getDate()))
,ColC int NULL
,constraint PK_tblSource primary key clustered (ColID))
```

```
Use SSISIncrementalLoad_Dest
Go
```

```
If Not Exists(Select name
                From sys.tables
                Where name = 'tblDest')
```

```
CREATE TABLE dbo.tblDest
(ColID int NOT NULL
,ColA varchar(10) NULL
,ColB datetime NULL
,ColC int NULL)
```

```
If Not Exists(Select name
                From sys.tables
                Where name = 'stgUpdates')
```

```
CREATE TABLE dbo.stgUpdates
(ColID int NULL
,ColA varchar(10) NULL
,ColB datetime NULL
,ColC int NULL)
```

```
Use SSISIncrementalLoad_Source
Go
```

```
-- insert an "unchanged", a "changed", and a "new" row
INSERT INTO dbo.tblSource
(ColID,ColA,ColB,ColC)
VALUES
(0, 'A', '1/1/2007 12:01 AM', -1),
(1, 'B', '1/1/2007 12:02 AM', -2),
(2, 'N', '1/1/2007 12:03 AM', -3)
```

```
Use SSISIncrementalLoad_Dest
Go
```

```
-- insert a "changed" and an "unchanged" row
INSERT INTO dbo.tblDest
(ColID,ColA,ColB,ColC)
VALUES
(0, 'A', '1/1/2007 12:01 AM', -1),
(1, 'C', '1/1/2007 12:02 AM', -2)
```

The T-SQL statements in Listing 19-5 create two databases; SSISIncrementalLoad_Source and SSISIncrementalLoad_Dest. A table named tblSource is created in the SSISIncrementalLoad_Source database and is populated with three rows. Another table named tblDest is created in the SSISIncrementalLoad_Dest database and is populated with two rows.

The configuration created by Listing 19-5 is a basic setup for an incremental load. ColID is the business key. This value should never change and should also uniquely identify the row in the Source and Destination systems. The character values in ColA of the Source and Destination tables indicate clues to the type of row. The A row is present and identical in both the Source and Destination tables. It is an *Unchanged* row. The row with a ColID value of 1 contains the ColA value B in the Source and the ColA value C in the Destination table. This row has *Changed* in the Source since it was initially loaded into the Destination table. The row with a ColID value of 2 exists only in the Source. It is a *New* row.

Adding Metadata

In this section, we will

- Add metadata that defines the connection managers used in the incremental load SSIS design pattern.
- Add a new Biml file to the Biml project and rename it **IncrementalLoad.biml**.
- Add a Connections XML node just after the <Biml> tag.
- Add two Connection XML nodes configured to connect with the SSISIncremental_Source and SSISIncremental_Dest databases.

Your code should appear as shown in Listing 19-6.

Listing 19-6. Configured Connections for IncrementalLoad.biml

```
<Biml xmlns="http://schemas.varigence.com/biml.xsd">
  <Connections>
    <Connection Name="SSISIncrementalLoad_Source" ConnectionString=
      "Data Source=(local);Initial Catalog=SSISIncrementalLoad_Source;Provider=
        SQLNCLI11.1;Integrated Security=SSPI;" />
    <Connection Name="SSISIncrementalLoad_Dest" ConnectionString=
      "Data Source=(local);Initial Catalog=SSISIncrementalLoad_Dest;Provider=
        SQLNCLI11.1;OLE DB Services=1;Integrated Security=SSPI;" />
  </Connections>
</Biml>
```

Add a Packages node between the </Connections> and </Biml> tags. Just after, add a Package XML node followed by a Tasks node. Immediately thereafter, add an ExecuteSQL node configured as shown in Listing 19-7.

Listing 19-7. Configured Packages, Package, Tasks, and ExecuteSQL Nodes

```
<Packages>
  <Package Name="IncrementalLoadPackage" ConstraintMode=
    "Parallel" ProtectionLevel="EncryptSensitiveWithUserKey">
    <Tasks>
      <ExecuteSQL Name="Truncate stgUpdates" ConnectionName="SSISIncrementalLoad_Dest">
        <DirectInput>Truncate Table stgUpdates</DirectInput>
      </ExecuteSQL>
    </Tasks>
  </Package>
</Packages>
```

The Execute SQL task defined in the Biml in Listing 19-7 will truncate a staging table that will hold rows that have been changed in the Source table since being loaded into the Destination table.

Specifying a Data Flow Task

After the `</ExecuteSQL>` tag, add a Dataflow XML node. Include a Name attribute and set the value of the Name attribute to "Load tblDest". Inside the `<Dataflow>` tag, add a PrecedenceConstraints node. Place an Inputs node inside the `<PrecedenceConstraints>` tag, and an Input node that includes an OutputPathName attribute with the value "Truncate stgUpdates.Output" inside the `<Inputs>` tag, as shown in Listing 19-8.

Listing 19-8. Adding a Precedence Constraint from the Truncate stgUpdates Execute SQL Task to the "Load tblDest" Data Flow Task

```
<Dataflow Name="Load tblDest">
  <PrecedenceConstraints>
    <Inputs>
      <Input OutputPathName="Truncate stgUpdates.Output" />
    </Inputs>
  </PrecedenceConstraints>
</Dataflow>
```

This code defines an OnSuccess precedence constraint between the Truncate stgUpdates Execute SQL task and the Load tblDest Data Flow task.

Adding Transforms

We are now ready to add metadata that define transforms, which are the heart of a Data Flow task. In this section, we will design an incremental load that includes an OLE DB Source adapter, a Lookup transform, a Condition Split transform, and a couple of OLE DB Destination adapters.

To begin, Add a Transformations node just after the `</PrecedenceConstraints>` tag. Inside the `<Transformations>` tags, add an OLEDBSource tag with following the attribute and value pairs:

- Name: tblSource Source
- ConnectionName: SSISIncrementalLoad_Source

Inside the `<OLEDBSource>` tag, add an ExternalTableInput node with a Table attribute whose value is "dbo.tblSource". This metadata constructs an OLE DB Source adapter named "tblSource Source" that connects to the SSISIncrementalLoad_Source connection defined earlier inside the `<Connections>` tag. The OLE DB Source adapter will connect to the table "dbo.tblSource" as specified in the ExternalTableInput tag. The Dataflow XML node will now appear, as shown in Listing 19-9.

Listing 19-9. The Dataflow Node Containing an OLE DB Source Adapter

```
<Dataflow Name="Load tblDest">
  <PrecedenceConstraints>
    <Inputs>
      <Input OutputPathName="Truncate stgUpdates.Output" />
    </Inputs>
  </PrecedenceConstraints>
  <Transformations>
    <OLEDBSource Name="tblSource Source" ConnectionName="SSISIncrementalLoad_Source">
      <ExternalTableInput Table="dbo.tblSource" />
    </OLEDBSource>
  </Transformations>
</Dataflow>
```

To continue, add a Lookup XML node immediately after the `</OleDbSource>` tag. Include the following attribute and value pairs in the `<Lookup>` tag:

- Name: Correlate
- OleDbConnectionName: SSISIncrementalLoad_Dest
- NoMatchBehavior: RedirectRowsToNoMatchOutput

The Name attribute sets the name of the Lookup transform. The OleDbConnectionName instructs Biml to use the connection manager defined in the `<Connections>` tag in the listing. The NoMatchBehavior attribute is configured to redirect nonmatching rows to the NoMatch output of the Lookup transform.

Continue configuring the metadata that define the Lookup transform by adding a DirectInput node immediately after the `<InputPath>` tag. Enter the following T-SQL statement between the `<DirectInput>` and `</DirectInput>` tags.

```
SELECT ColID, ColA, ColB, ColC FROM dbo.tblDest
```

Add an Inputs node immediately following the `</DirectInput>` tag. Inside the `<Inputs>` tag, add a Column node. Include the following attribute name and value pairs.

- SourceColumn: ColID
- TargetColumn: ColID

The preceding metadata provides the mapping between the Available Input columns and Available Lookup columns on the Columns page of the Lookup transform.

Add an Outputs node immediately following the `</Inputs>` tag. Inside the `<Outputs>` tag, add three Column nodes with the following attribute name and value pairs.

1.
 - a. SourceColumn: ColA
 - b. TargetColumn: Dest_ColA
2.
 - a. SourceColumn: ColB
 - b. TargetColumn: Dest_ColB
3.
 - a. SourceColumn: ColC
 - b. TargetColumn: Dest_ColC

The preceding metadata “selects” the columns returned from the Lookup transform’s Available Lookup columns on the Columns page. Once added, the Lookup transform metadata should appear as shown in Listing 19-10.

Listing 19-10. Transformations Including Lookup Metadata

```
<Transformations>
  <OleDbSource Name="tblSource Source" ConnectionName="SSISIncrementalLoad_Source">
    <ExternalTableInput Table="dbo.tblSource" />
  </OleDbSource>
  <Lookup Name="Correlate" OleDbConnectionName="SSISIncrementalLoad_Dest"
    NoMatchBehavior="RedirectRowsToNoMatchOutput">
```

```

<InputPath OutputPathName="tblSource Source.Output" />
<DirectInput>SELECT ColID, ColA, ColB, ColC FROM dbo.tblDest</DirectInput>
<Inputs>
  <Column SourceColumn="ColID" TargetColumn="ColID" />
</Inputs>
<Outputs>
  <Column SourceColumn="ColA" TargetColumn="Dest_ColA" />
  <Column SourceColumn="ColB" TargetColumn="Dest_ColB" />
  <Column SourceColumn="ColC" TargetColumn="Dest_ColC" />
</Outputs>
</Lookup>
</Transformations>

```

Immediately following the </Lookup> tag, add an OleDbDestination XML node with the following attribute name and value pairs.

- Name: tblDest Destination
- ConnectionName: SSISIncrementalLoad_Dest

Inside the <OleDbDestination> tag, add an InputPath node with an OutputPathName attribute set to the value "Correlate.NoMatch". After the <InputPath> tag, add an ExternalTableOutput node with a Table attribute set to the value "dbo.tblDest".

The preceding metadata defines an OLE DB Destination adapter and configures it to connect the Lookup transform's NoMatch output to the SSISIncrementalLoad_Dest connection defined earlier.

Add a ConditionalSplit XML node immediately after the </OleDbDestination> tag. Add an attribute called Name and set its value to "Filter". Inside the <ConditionalSplit> tags, add an InputPath XML node with an OutputPathName attribute set to "Correlate.Match". Now we need to add a conditional output path. Immediately following the <InputPath> tag, add an OutputPaths node, followed in turn by an OutputPath node containing a Name attribute set to "Changed Rows". Inside the <OutputPaths> tags, create an Expression node. Between the <Expression> and </Expression> tags, add the following SSIS expression.

```
(ColA != Dest_ColA) || (ColB != Dest_ColB) || (ColC != Dest_ColC)
```

Once this step is complete, the Transformations XML should appear as shown in Listing 19-11.

Listing 19-11. Transformations Node Including an OLE DB Source, Lookup, Conditional Split, and one OLE DB Destination

```

<Transformations>
  <OleDbSource Name="tblSource Source" ConnectionName="SSISIncrementalLoad_Source">
    <ExternalTableInput Table="dbo.tblSource" />
  </OleDbSource>
  <Lookup Name="Correlate" OleDbConnectionName="SSISIncrementalLoad_Dest"
    NoMatchBehavior="RedirectRowsToNoMatchOutput">
    <InputPath OutputPathName="tblSource Source.Output" />
    <DirectInput>SELECT ColID, ColA, ColB, ColC FROM dbo.tblDest</DirectInput>
    <Inputs>
      <Column SourceColumn="ColID" TargetColumn="ColID" />
    </Inputs>
    <Outputs>
      <Column SourceColumn="ColA" TargetColumn="Dest_ColA" />
      <Column SourceColumn="ColB" TargetColumn="Dest_ColB" />
      <Column SourceColumn="ColC" TargetColumn="Dest_ColC" />
    </Outputs>
  </Lookup>
  <ConditionalSplit Name="Filter">
    <InputPath OutputPathName="Correlate.Match">
      <OutputPaths>
        <OutputPath Name="Changed Rows">
          <Expression>
            (ColA != Dest_ColA) || (ColB != Dest_ColB) || (ColC != Dest_ColC)
          </Expression>
        </OutputPath>
      </OutputPaths>
    </InputPath>
  </ConditionalSplit>
  <OleDbDestination Name="tblDest Destination" ConnectionName="SSISIncrementalLoad_Dest">
    <InputPath OutputPathName="Correlate.NoMatch">
      <ExternalTableOutput Table="dbo.tblDest" />
    </InputPath>
  </OleDbDestination>
</Transformations>

```

```

    </Outputs>
  </Lookup>
  <OleDbDestination Name="tblDest Destination" ConnectionName="SSISIncrementalLoad_Dest">
    <InputPath OutputPathName="Correlate.NoMatch" />
    <ExternalTableOutput Table="dbo.tblDest" />
  </OleDbDestination>
  <ConditionalSplit Name="Filter">
    <InputPath OutputPathName="Correlate.Match"/>
    <OutputPaths>
      <OutputPath Name="Changed Rows">
        <Expression>(ColA != Dest_ColA) || (ColB != Dest_ColB) ||
          (ColC != Dest_ColC)</Expression>
      </OutputPath>
    </OutputPaths>
  </ConditionalSplit>
</Transformations>

```

The conditional split metadata most recently added configures a single output named "Changed Rows" and assigns an SSIS expression designed to detect changes in rows that exist in both the Source and Destination tables.

The final component in our Data Flow task is an OLE DB Destination adapter designed to stage rows that will be updated *after* the data flow completes execution. Immediately following the </ConditionalSplit> tag, add an OleDbDestination node with the following attribute name and value pairs.

- Name: stgUpdates
- ConnectionName: SSISIncrementalLoad_Dest

Inside the <OleDbDestination> tag, add a new node named InputPath with an attribute named OutputPathName and the value set to "Filter.Changed Rows". Immediately thereafter, add a node named ExternalTableOutput that includes a Table attribute set to "dbo.stgUpdates". This metadata defines an OLE DB Destination adapter that connects the Changed Rows output of the conditional split named Filter to a table named dbo.stgUpdates in the database defined by the SSISIncrementalLoad_Dest connection defined previously.

The complete Data Flow task metadata is shown in Listing 19-12.

Listing 19-12. The Completed Dataflow XML Node

```

<Dataflow Name="Load tblDest">
  <PrecedenceConstraints>
    <Inputs>
      <Input OutputPathName="Truncate stgUpdates.Output" />
    </Inputs>
  </PrecedenceConstraints>
  <Transformations>
    <OleDbSource Name="tblSource Source" ConnectionName="SSISIncrementalLoad_Source">
      <ExternalTableInput Table="dbo.tblSource" />
    </OleDbSource>
    <Lookup Name="Correlate" OleDbConnectionName="SSISIncrementalLoad_Dest"
      NoMatchBehavior="RedirectRowsToNoMatchOutput">
      <InputPath OutputPathName="tblSource Source.Output" />
      <DirectInput>SELECT ColID, ColA, ColB, ColC FROM dbo.tblDest</DirectInput>
    </Lookup>
    <Column SourceColumn="ColID" TargetColumn="ColID" />
  </Transformations>

```

```

    <Outputs>
      <Column SourceColumn="ColA" TargetColumn="Dest_ColA" />
      <Column SourceColumn="ColB" TargetColumn="Dest_ColB" />
      <Column SourceColumn="ColC" TargetColumn="Dest_ColC" />
    </Outputs>
  </Lookup>
  <OleDbDestination Name="tblDest Destination" ConnectionName="SSISIncrementalLoad_Dest">
    <InputPath OutputPathName="Correlate.NoMatch" />
    <ExternalTableOutput Table="dbo.tblDest" />
  </OleDbDestination>
  <ConditionalSplit Name="Filter">
    <InputPath OutputPathName="Correlate.Match"/>
    <OutputPaths>
      <OutputPath Name="Changed Rows">
        <Expression>(ColA != Dest_ColA) || (ColB != Dest_ColB) ||
          (ColC != Dest_ColC)</Expression>
      </OutputPath>
    </OutputPaths>
  </ConditionalSplit>
  <OleDbDestination Name="stgUpdates" ConnectionName="SSISIncrementalLoad_Dest">
    <InputPath OutputPathName="Filter.Changed Rows" />
    <ExternalTableOutput Table="dbo.stgUpdates" />
  </OleDbDestination>
</Transformations>
</Dataflow>

```

There remains one more Execute SQL task to complete our incremental load SSIS package. This task will update the Destination table by applying the rows stored in the "dbo.stgUpdates" table using a single Update T-SQL statement. Applying the updates in this fashion is generally faster than updating each row individually.

To continue developing the demo code, add an ExecuteSQL XML node immediately following the </Dataflow> tag with the following attribute name and value pairs.

- Name: Apply stgUpdates
- ConnectionName: SSISIncrementalLoad_Dest

Immediately following the <ExecuteSQL> tag, add a PrecedenceConstraints node, followed by an Inputs node. Inside the <Inputs> tag add an Input node containing an attribute named OutputPathName set to the value "Load tblDest.Output". Add a DirectInput node immediately following the </PrecedenceConstraints> tag. Inside the <DirectInput> tags, add the following T-SQL statement.

```

Update Dest
Set Dest.ColA = Upd.ColA
  ,Dest.ColB = Upd.ColB
  ,Dest.ColC = Upd.ColC
From tblDest Dest
Join stgUpdates Upd
  On Upd.ColID = Dest.ColID

```

Believe it or not, that's it! If your Biml looks like Listing 19-13, you should have compilable metadata.

Listing 19-13. The Complete IncrementalLoad.biml Listing

```
<Biml xmlns="http://schemas.varigence.com/biml.xsd">
  <Connections>
    <Connection Name="SSISIncrementalLoad_Source" ConnectionString=
      "Data Source=(local);Initial Catalog=SSISIncrementalLoad_Source;Provider=
      SQLNCLI11.1;Integrated Security=SSPI" />
    <Connection Name="SSISIncrementalLoad_Dest" ConnectionString=
      "Data Source=(local);Initial Catalog=SSISIncrementalLoad_Dest;Provider=
      SQLNCLI11.1;OLE DB Services=1;Integrated Security=SSPI;" />
  </Connections>
  <Packages>
    <Package Name="IncrementalLoadPackage" ConstraintMode=
      "Parallel" ProtectionLevel="EncryptSensitiveWithUserKey">
      <Tasks>
        <ExecuteSQL Name="Truncate stgUpdates" ConnectionName="SSISIncrementalLoad_Dest">
          <DirectInput>Truncate Table stgUpdates</DirectInput>
        </ExecuteSQL>
        <Dataflow Name="Load tblDest">
          <PrecedenceConstraints>
            <Inputs>
              <Input OutputPathName="Truncate stgUpdates.Output" />
            </Inputs>
          </PrecedenceConstraints>
          <Transformations>
            <OleDbSource Name="tblSource Source"
              ConnectionName="SSISIncrementalLoad_Source">
              <ExternalTableInput Table="dbo.tblSource" />
            </OleDbSource>
            <Lookup Name="Correlate" OleDbConnectionName="SSISIncrementalLoad_Dest"
              NoMatchBehavior="RedirectRowsToNoMatchOutput">
              <InputPath OutputPathName="tblSource Source.Output" />
              <DirectInput>SELECT ColID, ColA, ColB, ColC FROM dbo.tblDest</DirectInput>
              <Inputs>
                <Column SourceColumn="ColID" TargetColumn="ColID" />
              </Inputs>
              <Outputs>
                <Column SourceColumn="ColA" TargetColumn="Dest_ColA" />
                <Column SourceColumn="ColB" TargetColumn="Dest_ColB" />
                <Column SourceColumn="ColC" TargetColumn="Dest_ColC" />
              </Outputs>
            </Lookup>
            <OleDbDestination Name="tblDest Destination"
              ConnectionName="SSISIncrementalLoad_Dest">
              <InputPath OutputPathName="Correlate.NoMatch" />
              <ExternalTableOutput Table="dbo.tblDest" />
            </OleDbDestination>
            <ConditionalSplit Name="Filter">
              <InputPath OutputPathName="Correlate.Match"/>
              <OutputPaths>
```



```

        <OutputPath Name="Changed Rows">
            <Expression>(ColA != Dest_ColA) || (ColB != Dest_ColB) ||
                (ColC != Dest_ColC)</Expression>
        </OutputPath>
    </OutputPaths>
</ConditionalSplit>
<OleDbDestination Name="stgUpdates" ConnectionName="SSISIncrementalLoad_Dest">
    <InputPath OutputPathName="Filter.Changed Rows" />
    <ExternalTableOutput Table="dbo.stgUpdates" />
</OleDbDestination>
</Transformations>
</Dataflow>
<ExecuteSQL Name="Apply stgUpdates" ConnectionName="SSISIncrementalLoad_Dest">
    <PrecedenceConstraints>
        <Inputs>
            <Input OutputPathName="Load tblDest.Output" />
        </Inputs>
    </PrecedenceConstraints>
    <DirectInput>
        Update Dest
        Set Dest.ColA = Upd.ColA
        ,Dest.ColB = Upd.ColB
        ,Dest.ColC = Upd.ColC
        From tblDest Dest
        Join stgUpdates Upd
        On Upd.ColID = Dest.ColID
    </DirectInput>
</ExecuteSQL>
</Tasks>
</Package>
</Packages>
</Biml>

```

We are now ready to test!

Testing the Biml

Testing the Biml will consist of generating the SSIS package, then executing it. We will look at the data to see if the incremental load executed as expected. To begin, I have prepared a T-SQL Reset Rows script shown in Listing 19-14.

Listing 19-14. Resetting the Incremental Load Source and Destination Values

```

Use SSISIncrementalLoad_Source
Go

```

```

TRUNCATE TABLE dbo.tblSource

```

```
-- insert an "unchanged" row, a "changed" row, and a "new" row
INSERT INTO dbo.tblSource
(ColID,ColA,ColB,ColC)
VALUES
(0, 'A', '1/1/2007 12:01 AM', -1),
(1, 'B', '1/1/2007 12:02 AM', -2),
(2, 'N', '1/1/2007 12:03 AM', -3)

Use SSISIncrementalLoad_Dest
Go

TRUNCATE TABLE dbo.stgUpdates
TRUNCATE TABLE dbo.tblDest
```

```
-- insert an "unchanged" row and a "changed" row
INSERT INTO dbo.tblDest
(ColID,ColA,ColB,ColC)
VALUES
(0, 'A', '1/1/2007 12:01 AM', -1),
(1, 'C', '1/1/2007 12:02 AM', -2)
```

Listing 19-15 contains the test script we will use to examine and compare the contents of the Source and Destination.

Listing 19-15. Test Script for the IncrementalLoad.dtsx SSIS Package

```
Use SSISIncrementalLoad_Source
Go

SELECT TableName = 'tblSource'
       ,ColID
       ,ColA
       ,ColB
       ,ColC
FROM dbo.tblSource
Go

Use SSISIncrementalLoad_Dest
Go

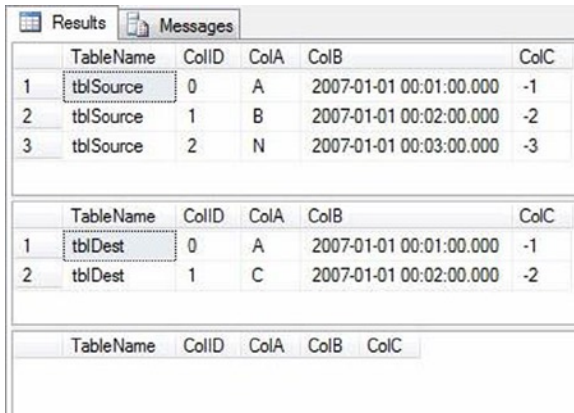
SELECT TableName = 'tblDest'
       ,[ColID]
       ,[ColA]
       ,[ColB]
       ,[ColC]
FROM [dbo].[tblDest]
```

```

SELECT TableName = 'stgUpdates'
       , [ColID]
       , [ColA]
       , [ColB]
       , [ColC]
FROM [dbo].[stgUpdates]
Go

```

Executing the test script after executing the reset script yields the results pictured in Figure 19-4.



	TableName	ColID	ColA	ColB	ColC
1	tblSource	0	A	2007-01-01 00:01:00.000	-1
2	tblSource	1	B	2007-01-01 00:02:00.000	-2
3	tblSource	2	N	2007-01-01 00:03:00.000	-3

	TableName	ColID	ColA	ColB	ColC
1	tblDest	0	A	2007-01-01 00:01:00.000	-1
2	tblDest	1	C	2007-01-01 00:02:00.000	-2

TableName	ColID	ColA	ColB	ColC

Figure 19-4. Pre-SSIS-Package-Execution results of test script

Return to Solution Explorer in SQL Server Data Tools. Right-click `IncrementalLoad.biml` and click `Generate SSIS Packages`. If you receive no error, your Biml is sound and you should see an SSIS package named `IncrementalLoadPackage.dtsx` in the SSIS Packages virtual folder in Solution Explorer. If the SSIS package opens with no errors, press the F5 key to execute it in the Debugger. If all is as it should be, you should see results similar to those shown in Figure 19-5.

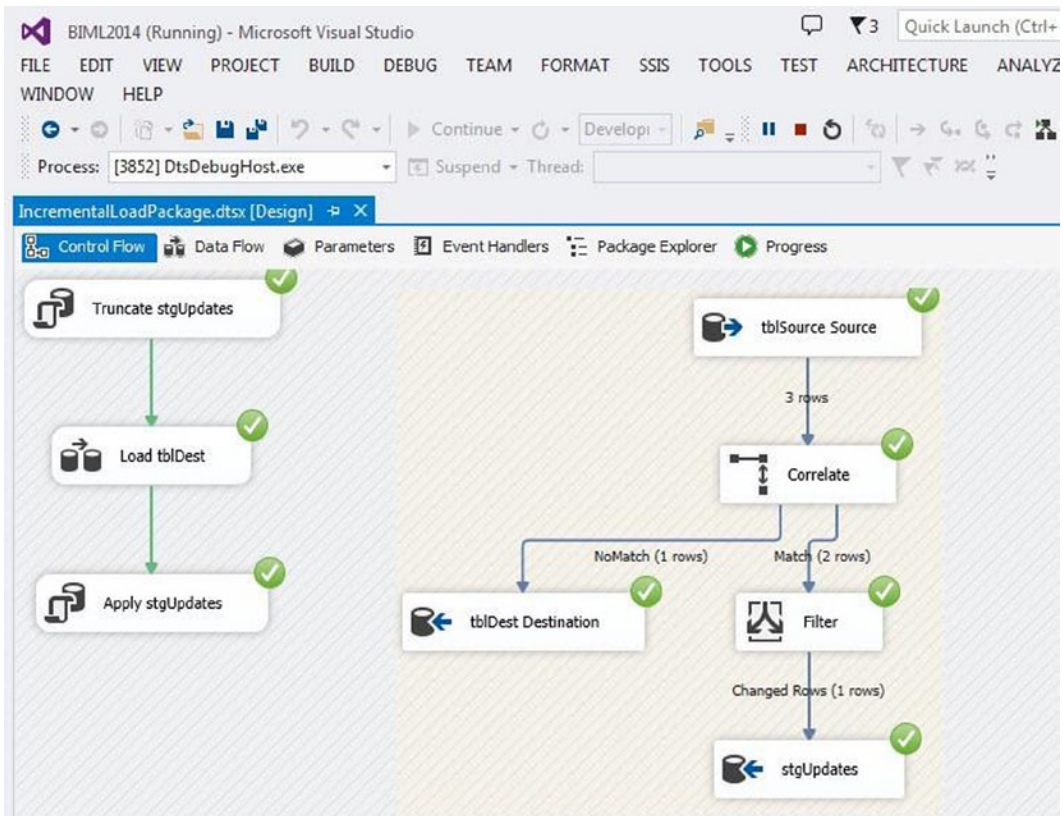


Figure 19-5. Debug execution for *IncrementalLoadPackage.dtsx*

Executing the test script now returns evidence that `SSISIncrementalLoad_Dest.dbo.tblDest` has received the updates loaded from `SSISIncrementalLoad_Source.dbo.tblSource`, as shown in Figure 19-6.

Results					
	TableName	ColID	ColA	ColB	ColC
1	tblSource	0	A	2007-01-01 00:01:00.000	-1
2	tblSource	1	B	2007-01-01 00:02:00.000	-2
3	tblSource	2	N	2007-01-01 00:03:00.000	-3
	TableName	ColID	ColA	ColB	ColC
1	tblDest	0	A	2007-01-01 00:01:00.000	-1
2	tblDest	1	B	2007-01-01 00:02:00.000	-2
3	tblDest	2	N	2007-01-01 00:03:00.000	-3
	TableName	ColID	ColA	ColB	ColC
1	stgUpdates	1	B	2007-01-01 00:02:00.000	-2

Figure 19-6. Results of a successful execution of *IncrementalLoadPackage.dtsx*

By examining the results and comparing them to Figure 19-4, we can see `SSISIncrementalLoad_Dest.dbo.tblDest` has been updated to match `SSISIncrementalLoad_Source.dbo.tblSource`. We can also see the updated row, with `ColID` equal to 1, was sent to the `SSISIncrementalLoad_Dest.dbo.stgUpdates` table.

Cool. But just wait: this is about to get awesome.

Using Biml as an SSIS Design Patterns Engine

Let's do something really cool and interesting with Biml. Using the `IncrementalLoad.biml` file as a template, and applying .NET integration—known as `BimlScript`—found in the Biml library supplied to BISD Helper, we are going to add flexibility and versatility to a new Biml file that will build an incremental load SSIS package between all the tables in a source and staging database. This is an example of the capital “E” in ETL; this is an extraction SSIS design pattern.

Note This pattern requires that the Source and Stage tables exist prior to expanding the Biml file to create the SSIS packages. Even with this caveat, which can be addressed, automated, and overcome, I believe this example demonstrates the power and game-changing attributes of Biml.

Let's begin by adding new tables to the `SSISIncrementalLoad_Source` database and creating—and populating—a new database named `SSISIncrementalLoad_Stage`. First, add new tables to `SSISIncrementalLoad_Source` by executing the T-SQL script shown in Listing 19-16.

Listing 19-16. Adding and Populating New `SSISIncrementalLoad_Source` Tables

```
USE SSISIncrementalLoad_Source
GO

-- Create Source1
If Not Exists(Select name
              From sys.tables
              Where name = 'Source1')
CREATE TABLE dbo.Source1
(ColID int NOT NULL
,ColA varchar(10) NULL
,ColB datetime NULL
,ColC int NULL
,constraint PK_Source1 primary key clustered (ColID))
Go

-- Load Source1
INSERT INTO dbo.Source1
(ColID,ColA,ColB,ColC)
VALUES
(0, 'A', '1/1/2007 12:01 AM', -1),
(1, 'B', '1/1/2007 12:02 AM', -2),
(2, 'C', '1/1/2007 12:03 AM', -3),
(3, 'D', '1/1/2007 12:04 AM', -4),
(4, 'E', '1/1/2007 12:05 AM', -5),
(5, 'F', '1/1/2007 12:06 AM', -6)
```

```

-- Create Source1
If Not Exists(Select name
              From sys.tables
              Where name = 'Source2')
CREATE TABLE dbo.Source2
(ColID int NOT NULL
,Name varchar(25) NULL
,Value int NULL
,constraint PK_Source2 primary key clustered (ColID))
Go

-- Load Source2
INSERT INTO dbo.Source2
(ColID,Name,Value)
VALUES
(0, 'Willie', 11),
(1, 'Waylon', 22),
(2, 'Stevie Ray', 33),
(3, 'Johnny', 44),
(4, 'Kris', 55)

-- Create Source3
If Not Exists(Select name
              From sys.tables
              Where name = 'Source3')
CREATE TABLE dbo.Source3
(ColID int NOT NULL
,Value int NULL
,Name varchar(100) NULL
,constraint PK_Source3 primary key clustered (ColID))
Go

-- Load Source3
INSERT INTO dbo.Source3
(ColID,Value,Name)
VALUES
(0, 101, 'Good-Hearted Woman'),
(1, 202, 'Lonesome, Onry, and Mean'),
(2, 303, 'The Sky Is Crying'),
(3, 404, 'Ghost Riders in the Sky'),
(4, 505, 'Sunday Morning, Coming Down')

```

The T-SQL in Listing 19-16 creates and populates three new tables.

- `dbo.Source1`
- `dbo.Source2`
- `dbo.Source3`

Execute the T-SQL shown in Listing 19-17 to build and populate the `SSISIncrementalLoad_Stage` database.

Listing 19-17. Building and Populating the SSISIncrementalLoad_Stage Database

```

Use master
Go

If Not Exists(Select name
              From sys.databases
              Where name = 'SSISIncrementalLoad_Stage')
  Create Database SSISIncrementalLoad_Stage
Go

Use SSISIncrementalLoad_Stage
Go

CREATE TABLE dbo.tblSource(
    ColID int NOT NULL,
    ColA varchar(10) NULL,
    ColB datetime NULL,
    ColC int NULL
)

CREATE TABLE dbo.stgUpdates_tblSource(
    ColID int NOT NULL,
    ColA varchar(10) NULL,
    ColB datetime NULL,
    ColC int NULL
)
Go

INSERT INTO dbo.tblSource
    (ColID,ColA,ColB,ColC)
VALUES
    (0, 'A', '1/1/2007 12:01 AM', -1),
    (1, 'B', '1/1/2007 12:02 AM', -2),
    (2, 'N', '1/1/2007 12:03 AM', -3)
Go

CREATE TABLE dbo.Source1(
    ColID int NOT NULL,
    ColA varchar(10) NULL,
    ColB datetime NULL,
    ColC int NULL
)

CREATE TABLE dbo.stgUpdates_Source1(
    ColID int NOT NULL,
    ColA varchar(10) NULL,
    ColB datetime NULL,
    ColC int NULL
)
Go

```

```

INSERT INTO dbo.Source1
(ColID,ColA,ColB,ColC)
VALUES
(0, 'A', '1/1/2007 12:01 AM', -1),
(1, 'Z', '1/1/2007 12:02 AM', -2)
Go

CREATE TABLE dbo.Source2(
    ColID int NOT NULL,
    Name varchar(25) NULL,
    Value int NULL
)

CREATE TABLE dbo.stgUpdates_Source2(
    ColID int NOT NULL,
    Name varchar(25) NULL,
    Value int NULL
)
Go

INSERT INTO dbo.Source2
(ColID,Name,Value)
VALUES
(0, 'Willie', 11),
(1, 'Waylon', 22),
(2, 'Stevie', 33)
Go

CREATE TABLE dbo.Source3(
    ColID int NOT NULL,
    Value int NULL,
    Name varchar(100) NULL
)

CREATE TABLE dbo.stgUpdates_Source3(
    ColID int NOT NULL,
    Value int NULL,
    Name varchar(100) NULL
)
Go

INSERT INTO dbo.Source3
(ColID,Value,Name)
VALUES
(0, 101, 'Good-Hearted Woman'),
(1, 202, 'Are You Sure Hank Done It This Way?')
Go

```

Let's continue by adding a new Biml file to the Biml project. Rename this file `GenerateStagingPackages.biml`. Before the `<Biml>` tag, add the code snippet shown in Listing 19-18.

Listing 19-18. Adding .NET Namespaces and Initial Method Calls to Biml

```
<#@ import namespace="System.Data" #>
<#@ import namespace="Varigence.Hadron.CoreLowerer.SchemaManagement" #>
<# var connection = SchemaManager.CreateConnectionNode("SchemaProvider", "Data Source=(local);Initial
Catalog=SSISIncrementalLoad_Source;Provider=SQLNCLI11.1;Integrated Security=SSPI;"); #>
<# var tables = connection.GenerateTableNodes(); #>
```

The code in Listing 19-18 imports the `System.Data` and `Varigence.Hadron.CoreLowerer.SchemaManagement` namespaces into the Biml file. A variable named `connection` is created and assigned the value of a `SchemaManager ConnectionNode` object, which is aimed at the `SSISIncrementalLoad_Source` database. The `connection` variable supports another variable named `tables`. The `tables` variable is populated from a call to the `connection` variable's `GenerateTableNodes()` method, which populates `tables` with the list of tables found in the `SSISIncrementalLoad_Source` database.

After the `<Biml>` tag, add a `Connections` XML node that contains two `Connection` child nodes so that your Biml file now appears as shown in Listing 19-19.

Listing 19-19. Adding Connections to the `GenerateStagingPackages.biml` File

```
<#@ import namespace="System.Data" #>
<#@ import namespace="Varigence.Hadron.CoreLowerer.SchemaManagement" #>
<# var connection = SchemaManager.CreateConnectionNode("SchemaProvider",
    "Data Source=(local);Initial Catalog=SSISIncrementalLoad_Source;Provider=
    SQLNCLI11.1;Integrated Security=SSPI;"); #>
<# var tables = connection.GenerateTableNodes(); #>
<Biml xmlns="http://schemas.varigence.com/biml.xsd">
<Connections>
    <Connection Name="SSISIncrementalLoad_Source" ConnectionString=
        "Data Source=(local);Initial Catalog=SSISIncrementalLoad_Source;Provider=
        SQLNCLI11.1;Integrated Security=SSPI;" />
    <Connection Name="SSISIncrementalLoad_Stage" ConnectionString=
        "Data Source=(local);Initial Catalog=SSISIncrementalLoad_Stage;Provider=
        SQLNCLI11.1;OLE DB Services=1;Integrated Security=SSPI;" />
</Connections>
```

As in the `IncrementalLoad.biml` file we designed in the last section, the `Connection` nodes are the templates for SSIS Connection Managers in the SSIS package. Next, add a `Package` node immediately after the `</Connections>` tag. Here we will make a crucial modification to this Biml file and its capability. We begin a C# loop here that spans all but the last two lines of this Biml file. Your Biml file should now include the code from Listing 19-20, immediately after the `</Connections>` tag.

Listing 19-20. Adding the Packages Node and Starting a Loop

```
<Packages>
    <# foreach (var table in tables) { #>
```

The loop defined in Listing 19-20 will drive the Biml engine as it creates an SSIS package for each table found in the `SSISIncrementalLoad_Source` database. Because we are using the SSIS incremental load design pattern as the template for this package, this Biml file will construct an incremental load SSIS package for each of these tables.

The variables defined above are used later in the Biml file. Immediately after these variable declarations, add the `Package` node shown in Listing 19-21.

Listing 19-21. The Package Node with .NET Replacements

```
<Package Name="IncrementalLoad_<#=table.Name#>" ConstraintMode=
  "Linear" ProtectionLevel="EncryptSensitiveWithUserKey">
```

This Biml code, like much in this Biml file, is copied from the `IncrementalLoad.biml` file and modified to accept .NET overrides from the `ForEach` loop. Each SSIS package generated when this Biml is expanded will be named consistently: `IncrementalLoad_<Source Table Name>`.

Also note the `ConstraintMode` attribute of the `Package` node is set to "Linear". In the `IncrementalLoad.biml` file, this was set to "Parallel". The differences are subtle but powerful. First, the Biml compiler will automatically create precedence constraints for you. Specifically, it will create an `OnSuccess` precedence constraint in the control flow from one task to the next, based on the order they appear in the Biml file. This functionality makes scripting and simple file authoring extremely quick. Second, you can eliminate `InputPath` nodes in the Data Flow task because the `InputPath` will connect to the default output path of the transform that appears directly before it.

Immediately following the `<Package>` tag, add a `Tasks` node, followed by an `ExecuteSQL` node configured as shown in Listing 19-22.

Listing 19-22. Adding Tasks and the Truncate Staging Table Execute SQL Task

```
<Tasks>
  <ExecuteSQL Name="Truncate stgUpdates_<#=table.Name#>"
    ConnectionName="SSISIncrementalLoad_Stage">
    <DirectInput>Truncate Table stgUpdates_<#=table.Name#></DirectInput>
  </ExecuteSQL>
```

Again, note the generic naming of the `Execute SQL` task that performs the truncate operation on the staging table. The name of the Source table will replace the `<#=table.Name#>` placeholder when the Biml file is expanded. It will be named differently for each table in the Source database, but it will also be descriptive and accurate.

In the next listing (Listing 19-23), I am simply going to show you the Biml for the incrementally loading Data Flow task. Each component includes .NET code where necessary to make the Biml generic enough to respond to different Source table schemas.

Listing 19-23. The Generic Data Flow Task

```
<Dataflow Name="Load <#=table.Name#>">
  <Transformations>
    <OleDbSource Name="<#=table.Name#> Source" ConnectionName="SSISIncrementalLoad_Source">
      <DirectInput>SELECT <#=table.GetColumnList()#> FROM
<#=table.SchemaQualifiedName#></DirectInput>
    </OleDbSource>
    <Lookup Name="Correlate" OleDbConnectionName="SSISIncrementalLoad_Stage"
NoMatchBehavior="RedirectRowsToNoMatchOutput">
      <DirectInput>SELECT <#=table.GetColumnList()#> FROM dbo.<#=table.Name#></DirectInput>
    <Inputs>
      <# foreach (var keyColumn in table.Keys[0].Columns) { #>
      <Column SourceColumn="<#=keyColumn.Column#>" TargetColumn="<#=keyColumn.Column#>" />
      <# } #>
    </Inputs>
    <Outputs>
      <# foreach (var col in table.Columns) { #>
      <Column SourceColumn="<#=col#>" TargetColumn="Dest_<#=col#>" />
      <# } #>
    </Outputs>
```

```

</Lookup>
<ConditionalSplit Name="Filter">
  <OutputPaths>
    <OutputPath Name="Changed Rows">
      <# string exp = "";
        foreach (var colex in table.Columns) { exp += "(" + colex + " != Dest_"
          + colex + ") || "; } #>
      <Expression><#=exp.Substring(0, exp.Length - 4)#></Expression>
    </OutputPath>
  </OutputPaths>
</ConditionalSplit>
<OleDbDestination Name="stgUpdates_<#=table.Name#>"
  ConnectionName="SSISIncrementalLoad_Stage">
  <InputPath OutputPathName="Filter.Changed Rows" />
  <ExternalTableOutput Table="dbo.stgUpdates_<#=table.Name#>" />
</OleDbDestination>
<OleDbDestination Name="<#=table.Name#> Destination"
  ConnectionName="SSISIncrementalLoad_Stage">
  <InputPath OutputPathName="Correlate.NoMatch" />
  <ExternalTableOutput Table="dbo.<#=table.Name#>" />
</OleDbDestination>
</Transformations>
</Dataflow>

```

The Biml/.NET code shown in Listing 19-24 dynamically generates an incrementally loading Data Flow task, given the caveats listed near the beginning of this section. Let's complete the Biml file by creating a generic template for the final Execute SQL task that performs the set-based update for Changed Rows between the staging table and destination, shown in Listing 19-24.

Listing 19-24. The Generic Apply Staged Updates Execute SQL Task

```

<ExecuteSQL Name="Apply stgUpdates_<#=table.Name#>"
  ConnectionName="SSISIncrementalLoad_Stage">
  <# string upd = "Update Dest Set ";
    foreach (var colex in table.Columns.Where(column =>
!table.Keys[0].Columns.Select(keyColumn => keyColumn.Column).Contains(column))) {
      upd = upd + "Dest." + colex + " = Upd." + colex + ",";
    }
    var updc = upd.Substring(0,upd.Length-1) + " From " + table.SchemaQualifiedName +
" Dest Join [" + table.Schema.Name + "].[" + stgUpdates_ + table.Name + "] Upd On Upd." +
table.Keys[0].Columns[0].Column + " = Dest." + table.Keys[0].Columns[0].Column;#>
  <DirectInput><#=updc#></DirectInput>
</ExecuteSQL>

```

The final instructions contained in the Biml file are shown in Listing 19-25 and close out the Tasks, Package, Packages, loop, and Biml nodes.

Listing 19-25. Closing out the Last Nodes and Loop in the Biml File

```
</Tasks>
</Package>
<# } #>
</Packages>
</Biml>
```

Time for a Test

In Solution Explorer, right-click the `GenerateStagingPackages.biml` file and click **Generate SSIS Packages**. If all goes as planned, your Solution Explorer window should appear similar to that shown in Figure 19-7.

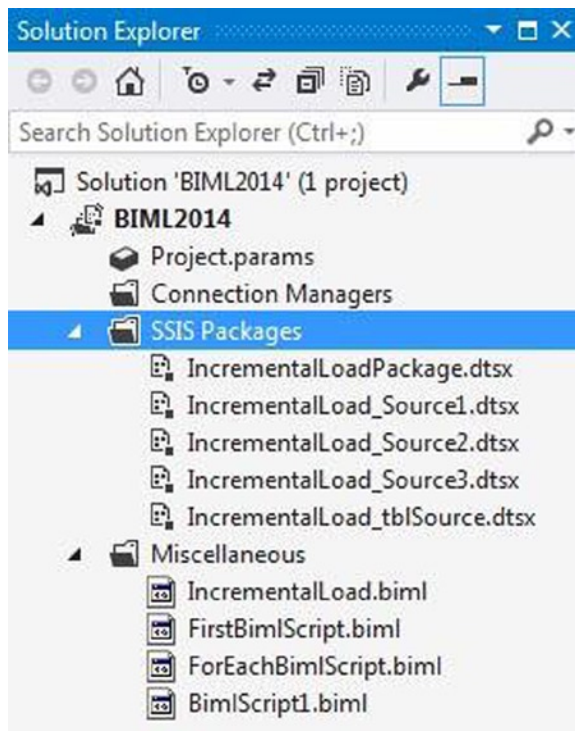


Figure 19-7. Four SSIS packages from one Biml file!

Conduct further testing by executing (and re-executing) each of the four SSIS packages created by the Biml expansion. When I execute the SSIS package named `IncrementalLoad_tblSource.dtsx`, I see results (shown in Figure 19-8) remarkably similar to those observed earlier (in Figure 19-5).

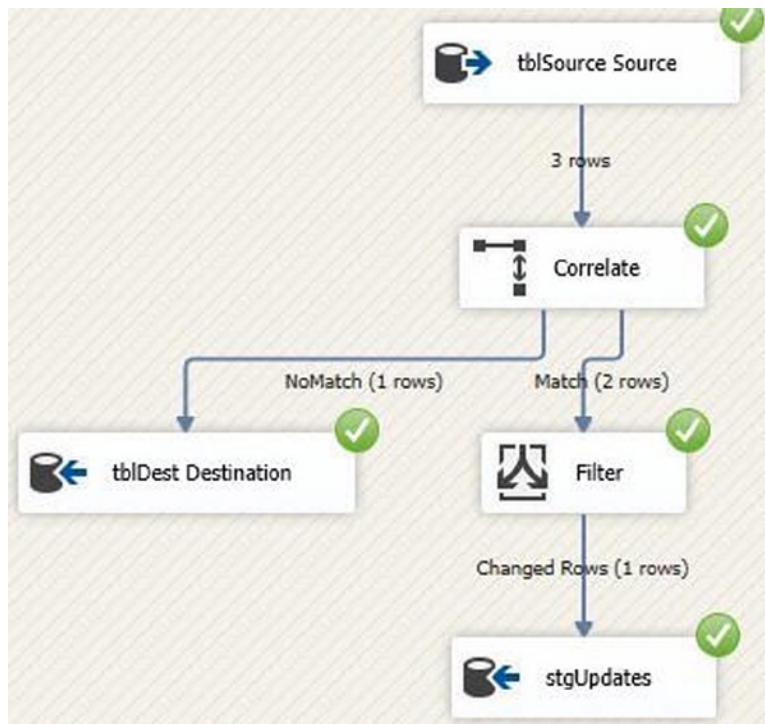


Figure 19-8. Dynamically-built incremental load SSIS package

Testing will reveal the other SSIS packages perform similarly.

Conclusion

In this chapter, we have taken a brief tour of some of the functionality of Business Intelligence Markup Language. We have demonstrated its usefulness as a domain-specific language for generating SSIS design patterns, focusing on the incremental load pattern. In the final example, we demonstrated how Biml's integrated .NET functionality, BimlScript, can be used to create a patterns-based approach to building four SSIS packages using a tried-and-true data integration pattern (incremental load). On my machine, the four packages were generated in a matter of seconds. I assure you, based on testing, Biml can produce hundreds of incremental load SSIS packages in a matter of minutes. This is game-changing technology, because generating hundreds of SSIS packages—even using templates and patterns—can easily consume data integration developer-months.



Biml and SSIS Frameworks

Elegant solutions lead to more elegant solutions. This is one feature of elegant solutions, and Business Intelligence Markup Language (Biml) is an elegant solution. Biml facilitates the rapid development of SSIS packages. As impressive as this is, Biml can do more – much more. We will not be able to explore Biml’s complete feature set in this volume, but I would like to examine some additional functionality.

In this chapter, we will build on the example file from the Business Intelligence Markup Language chapter, adding some additional files and functionality.

Using Biml with an SSIS Framework

Appendix A shows the source code for building a file system-based SSIS Framework. Because SSIS 2014 supports backwards-compatibility via the Package Deployment Model, you can use this Framework in all versions of SSIS (to date). The examples in this chapter are designed to interact with the Framework described in Appendix A.

■ **Note** To complete the examples that follow you will need to implement the Framework in Appendix A and complete the examples in the chapter titled *Business Intelligence Markup Language*.

Adding SSIS Package Metadata to the Framework

Open the solution created in the Business Intelligence Markup Language chapter. The solution should be named “Biml2014.sln.” Add a new Biml file and rename it “FrameworkSQLGen.biml.” Add a second new Biml file and rename it “i-FrameworkSQLGen.biml.” I will use the i-FrameworkSQLGen.biml file to demonstrate parameterizing a Biml solution.

Add the following BimlScript to i-FrameworkSQLGen.biml, as shown in Listing 20-1:

Listing 20-1. Manually Adding SSIS Application Metadata to the Framework

```
<#
string applicationName = "Staging_Application";
string packageFolder = "I:\\Andy\\Projects\\BIML Demo\\";
string filename = @"G:\out\MyFile.sql";
string connectionString = "Data Source=(local);Initial Catalog=SSISIncrementalLoad_Source;Provider=SQLNCLI11.1;Integrated Security=SSPI;";
#>
```

The BimlScript shown in Listing 20-1 is a code fragment containing variable declarations with initial values. The `applicationName` variable is a string that contains the name of the SSIS Application I will be adding to the SSIS Framework. The `packageFolder` string contains the location of the SSIS packages I generated in the Business Intelligence Markup Language chapter example code. The `filename` string variable contains the full path to the SQL file that will be generated by the `FrameworkSQLGen.biml` file. The `connectionString` variable contains connection information to the source database used by this solution and the example in the Business Intelligence Markup Language chapter.

Open the `FrameworkSQLGen.biml` file and add the code shown in Listing 20-2:

Listing 20-2. Starting the `FrameworkSQLGen.biml` File

```
<#@ template language="C#" tier="1" #>
<#@ include file="i-FrameworkSQLGen.biml" #>
<#@ import namespace="System.Data" #>
<#@ import namespace="System.Text" #>
<#@ import namespace="System.IO" #>
<#@ import namespace="Varigence.Hadron.CoreLowerer.SchemaManagement" #>
<# var connection = SchemaManager.CreateConnectionNode("SchemaProvider", connectionString); #>
<# var tables = connection.GenerateTableNodes();
    int e = 10;
    StringBuilder sb = new StringBuilder();
```

The template directive on line 1 identifies the .Net language I will use in this file (C#) and the “tier” of this file.

■ **Note** Learn more about Biml tiers and the language constructs in Reeves Smith’s excellent article “Stairway to Biml Level 5 - Biml Language Elements” at [SQLServerCentral.com](http://www.sqlservercentral.com/articles/BIML/111305/) (<http://www.sqlservercentral.com/articles/BIML/111305/>).

The “include file” directive is next and points to the `i-FrameworkSQLGen.biml` I built first. Including this file means the variables defined in `i-FrameworkSQLGen.biml` are available in the `FrameworkSQLGen.biml` file. I next include four namespaces: `System.Data`, `System.IO`, `System.Text`, and `Varigence.Hadron.CoreLowerer.SchemaManagement`. The first three refer to .Net libraries; the fourth supplies a reference to a Biml assembly.

The C# application logic begins in earnest starting with Line 7 with the declaration and initialization of the variable named “connection.” Connection is set to the `SchemaManager` object’s `ConnectionNode` object via the `CreateConnectionNode` method. The `ConnectionNode` is aimed at the `ConnectionString` supplied from the variable “connectionString” contained in the `i-FrameworkSQLGen.biml` include file.

Another C# variable named “tables” is declared and initialized using the connection variable’s `GenerateTableNodes` method. The `GenerateTableNodes` method creates a collection of table objects ready for consumption by BimlScript.

Two additional variables are declared and initialized: an `int` variable named “e” which will be used to enumerate the execution order of packages in the Framework, and a `StringBuilder` variable named “sb” which will be used to build the Transact-SQL command to load the Framework metadata for the SSIS Application.

The Transact-SQL script for adding an SSIS Application to the Framework is listed in Figure A-7 in Appendix A and is shown in Listing 20-3.

Listing 20-3. Manually Adding SSIS Application Metadata to the Framework

Use SSISConfig
Go

```
declare @ApplicationName varchar(255) = 'SSISApp1'
declare @ApplicationID int

/* Add the SSIS First Application */
If Not Exists(Select ApplicationName
              From cfg.Applications
              Where ApplicationName = @ApplicationName)
begin
    print 'Adding ' + @ApplicationName
    exec cfg.AddSSISApplication @ApplicationName, @ApplicationID output
    print @ApplicationName + ' added.'
end
Else
begin
    Select @ApplicationID = ApplicationID
    From cfg.Applications
    Where ApplicationName = @ApplicationName
    print @ApplicationName + ' already exists in the Framework.'
end
print ''
```

Place this code into the FrameworkSQLGen.biml after the code in Listing 2. I need to load this Transact-SQL into the stringbuilder variable (sb) and I need to make it generic by leveraging the variables in the include file. Accomplish this by loading the code into the stringbuilder as shown in Listing 20-4:

Listing 20-4. Adding SSIS Application Metadata to the Stringbuilder

```
sb.AppendLine("Use SSISConfig");
sb.AppendLine("Go");
sb.AppendLine();
sb.AppendLine("declare @ApplicationName varchar(255) = '" + applicationName + "'");
sb.AppendLine("declare @ApplicationID int");
sb.AppendLine();
sb.AppendLine("/* Add the SSIS First Application */");
sb.AppendLine("If Not Exists(Select ApplicationName");
sb.AppendLine("                From cfg.Applications");
sb.AppendLine("                Where ApplicationName = @ApplicationName) ");
sb.AppendLine(" begin");
sb.AppendLine("  print 'Adding ' + @ApplicationName");
sb.AppendLine("  exec cfg.AddSSISApplication @ApplicationName, @ApplicationID output");
sb.AppendLine("  print @ApplicationName + ' added.'");
sb.AppendLine(" end");
sb.AppendLine("Else");
sb.AppendLine(" begin");
sb.AppendLine("  Select @ApplicationID = ApplicationID");
sb.AppendLine("  From cfg.Applications");
sb.AppendLine("  Where ApplicationName = @ApplicationName");
sb.AppendLine("  print @ApplicationName + ' already exists in the Framework.'");
```



```
sb.AppendLine(" end");
sb.AppendLine("print ' '");
sb.AppendLine();
```

The Transact-SQL for building parameters to manage package metadata is found in the first sections of Listings A-5 and A-9 and is shown here in Listing 20-5:

Listing 20-5. SSIS Framework Package Parameters

```
declare @ExecutionOrder int = 10
declare @ApplicationID int = 1
declare @PackageID int = 1
declare @ApplicationName varchar(255) = 'SSISApp1'
declare @PackageFolder varchar(255) =
'F:\Andy\Projects\PublicFramework_PackageDeployment_2014\SSISConfig2014\'
declare @PackageName varchar(255) = 'Child1.dtsx'
```

Add to the stringbuilder Transact-SQL to declare the parameters that will be used to insert SSIS package metadata into the Framework, omitting some and adding some, as shown in Listing 20-6:

Listing 20-6. Adding SSIS Package Parameters to the Stringbuilder

```
sb.AppendLine("declare @ExecutionOrder int");
sb.AppendLine("declare @PackageID int");
sb.AppendLine("declare @PackageFolder varchar(255) = '" + packageFolder + "'");
sb.AppendLine("declare @PackageName varchar(255)");
sb.AppendLine();
sb.AppendLine("/* Add Packages */");
sb.AppendLine();
```

The real work of this BimlScript is done inside a loop, shown in Listing 20-7:

Listing 20-7. Starting the Loop to Load SSIS Framework Package Metadata

```
foreach (var table in tables)
{
    string tbl = "IncrementalLoad_" + table.Name + ".dtsx";
```

The foreach loop definition creates a new variable named table that will represent each Biml table in the tables collection populated earlier. The tbl string variable is created for use identifying the SSIS package by name.

The remainder of Listings A-5 and A-9 in Appendix A are consolidated and shown in Listing 20-8:

Listing 20-8. Adding SSIS Framework Package Metadata to the Stringbuilder

```
sb.AppendLine("/* " + tbl + " */");
sb.AppendLine();
sb.AppendLine("Set @PackageName = '" + tbl + "'");
sb.AppendLine("Set @ExecutionOrder = " + e.ToString());
sb.AppendLine("If Not Exists(Select PackageFolder + PackageName");
sb.AppendLine("                From cfg.Packages");
sb.AppendLine("                Where PackageFolder = @PackageFolder");
sb.AppendLine("                And PackageName = @PackageName)");
sb.AppendLine(" begin");
sb.AppendLine("  print 'Adding ' + @PackageFolder + @PackageName");
```

```

sb.AppendLine(" exec cfg.AddSSISPackage @PackageName, @PackageFolder, @PackageID output");
sb.AppendLine(" end");
sb.AppendLine("Else");
sb.AppendLine(" begin");
sb.AppendLine("     Select @PackageID = PackageID");
sb.AppendLine("     From cfg.Packages");
sb.AppendLine("     Where PackageFolder = @PackageFolder");
sb.AppendLine("     And PackageName = @PackageName");
sb.AppendLine("     print @PackageFolder + @PackageName + ' already exists in the Framework.'");
sb.AppendLine(" end");
sb.AppendLine();
sb.AppendLine("         If Not Exists(Select AppPackageID");
sb.AppendLine("             From cfg.AppPackages");
sb.AppendLine("             Where ApplicationID = @ApplicationID");
sb.AppendLine("             And PackageID = @PackageID");
sb.AppendLine("             And ExecutionOrder = @ExecutionOrder)");
sb.AppendLine(" begin");
sb.AppendLine("     print 'Adding ' + @ApplicationName + '.' + @PackageName + ' to Framework with ExecutionOrder ' + convert(varchar, @ExecutionOrder)");
sb.AppendLine(" exec cfg.AddSSISAppPackage @ApplicationID, @PackageID, @ExecutionOrder");
sb.AppendLine("     print @PackageName + ' added and wired to ' + @ApplicationName");
sb.AppendLine(" end");
sb.AppendLine("Else");
sb.AppendLine(" print @ApplicationName + '.' + @PackageName + ' already exists in the Framework with ExecutionOrder ' + convert(varchar, @ExecutionOrder)");
sb.AppendLine();
e += 10;
}

```

This code inside the loop executes for every package contained in the tables collection populated from the call against the connection object's `GenerateTableNodes()` method. Each iteration begins with a comment to identify the name of the SSIS package followed by the setting of the `PackageName` and `ExecutionOrder` parameter values. `PackageName` is set to the name of the SSIS Package contained in the string variable `tbl`; `ExecutionOrder` is set to the string version of the integer value contained in the `e` variable.

Next, a conditional statement which checks for the existence of a package in the Framework is added to the `StringBuilder` variable, "sb." If the package metadata is not present, it is added by executing the `cfg.AddSSISPackage` stored procedure which returns the `PackageID` parameter. If the package metadata is present, a statement populating the `PackageID` parameter is executed.

The next portion of the Transact-SQL added to the stringbuilder is a similar section which associates SSIS Package and Application metadata. This is accomplished by adding rows to the `cfg.AppPackages` "bridge" table – or informing the user that this relationship already exists.

The final statement of the loop increments the value of the variable `e` by 10.

The last section of `BimlScript` writes the contents of the stringbuilder to the file specified in the `i-FrameworkSQLGen.biml` file with the filename variable, as shown in Listing 20-9:

Listing 20-9. Writing the Transact-SQL in the Stringbuilder to the Output File

```

using (StreamWriter outfile = new StreamWriter(filename))
{
    outfile.Write(sb.ToString());
}
#>

```

Executing the Biml File

Let's test by generating the Transact-SQL file, as shown in Figure 20-1:



Figure 20-1. Execute the FrameworkSQLGen.biml File

Right-click the FrameworkSQLGen.biml file and click Generate SSIS Packages. The file with the filename MyFile.sql should be generated in the location contained in the i-FrameworkSQLGen.biml include file. Open that file in SQL Server Management Studio (SSMS) and execute the SQL statements. Your output should appear similar to that shown in Figure 20-2:

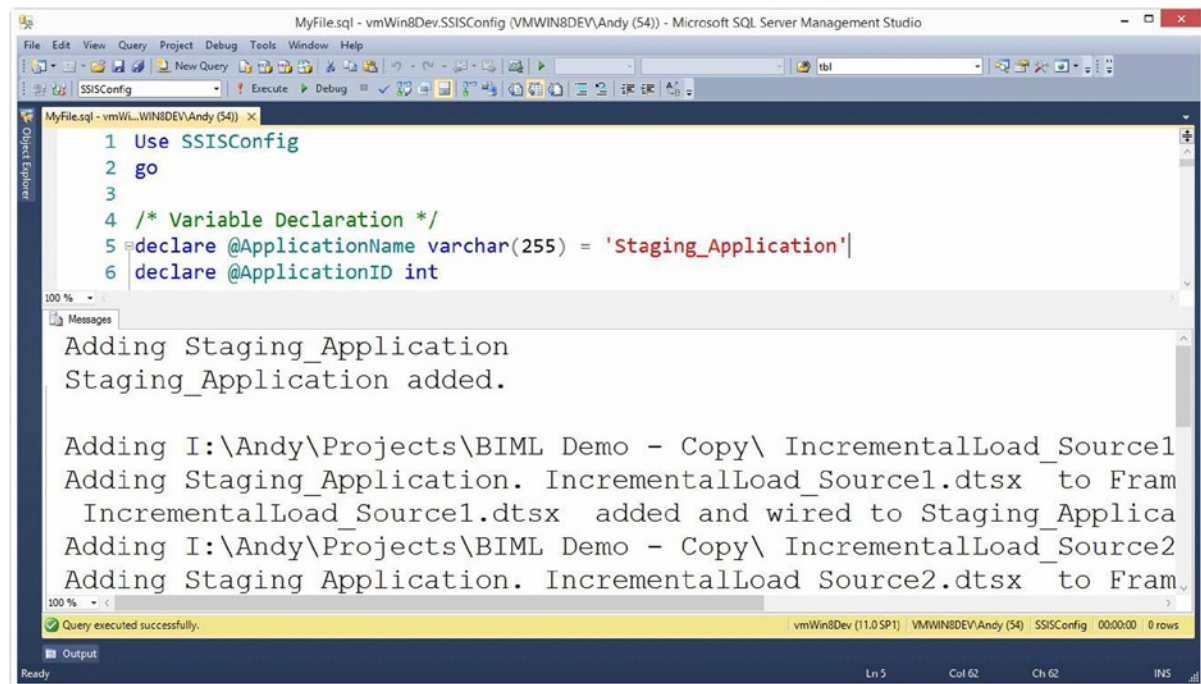


Figure 20-2. Execute the SQL File Generated by the FrameworkSQLGen.biml File

Generating the SSIS Command-Line

We can do even more using BimlScript, though. We can generate the command-line to execute the SSIS Application using the Framework's Parent.dtsx package.

Open the i-FrameworkSQLGen.biml file and add a couple more variables, as shown in Listing 20-10:

Listing 20-10. Adding Variables to the i-FrameworkSQLGen.biml File

```
string FrameworkParentPkgPath = @"E:\Projects\SSISConfig2014\SSISConfig2014\Parent.dtsx";
string CmdFilename = @"G:\out\MyCmd.cmd";
```

Your file locations will be different from those shown in Listing 20-10.

Add a new Biml file to your solution and rename it “GenerateExecCmd.biml.” Add the following statements to GenerateExecCmd.biml, shown in Listing 20-11:

Listing 20-11. Building the GenerateExecCmd.biml File

```
<#@ template language="C#" tier="2" #>
<#@ include file="i-FrameworkSQLGen.biml" #>
<#@ import namespace="System.Text" #>
<#@ import namespace="System.IO" #>
<# StringBuilder sb = new StringBuilder();
    sb.AppendLine("dtexec /FILE " + FrameworkParentPkgPath + " /SET
        \\Package.Variables[ApplicationName].Properties[Value];" + applicationName);
    using (StreamWriter outfile = new StreamWriter(CmdFilename))
    {
        outfile.Write(sb.ToString());
    }
#>
```

Right-click the GenerateExecCmd.biml file in Solution Explorer and click “Generate SSIS Packages.” Open Windows Explorer and browse to the folder you supplied for the CmdFilename variable in the i_FrameworkSQLGen.biml file. Open the file that was just generated and you should observe a string similar to the one shown in Listing 20-12:

Listing 20-12. The Command Line Generated by the GenerateExecCmd.biml File

```
dtexec /FILE E:\Projects\SSISConfig2014\SSISConfig2014\Parent.dtsx /SET \Package.
Variables[ApplicationName].Properties[Value];Staging_Application
```

Because this command line is contained in a “cmd” file, you should be able to double-click it watch it execute in a command prompt window, similar to that pictured in Figure 20-3:

```

Command Prompt

Cleanup: 80% complete
End Progress
Progress: 2014-09-19 20:24:58.31
Source: LoadtblSource
Cleanup: 100% complete
End Progress
Progress: 2014-09-19 20:24:58.32
Source: Apply stgUpdates_tblSource
Executing query "
Update Dest Set Dest.Cola = Upd.Cola, Dest...": 100% complete
End Progress
Progress: 2014-09-19 20:24:58.33
Source: Log Package Success
Executing query "log.LogPackageSuccess".: 100% complete
End Progress
Progress: 2014-09-19 20:24:58.33
Source: Log Application Success
Executing query "log.LogApplicationSuccess".: 100% complete
End Progress
DTExec: The package execution returned DTSEXEC_SUCCEEDED (0).
Started: 8:24:54 PM
Finished: 8:24:58 PM
Elapsed: 3.641 seconds
C:\Users\Andy>

```

Figure 20-3. Executing the Command Line Generated by *GenerateExecCmd.biml*

How cool is that?

Summarizing

In this chapter, we have extended our knowledge regarding the functionality of Business Intelligence Markup Language. We have demonstrated how to use Biml to dynamically generate Transact-SQL statements and execution command lines.



Evolution of an SSIS Framework

SSIS frameworks are the next logical step after SSIS design patterns because frameworks comprise many patterns. At a minimum, an SSIS framework should provide package execution control and monitoring. That sounds simple, but I assure you, it is not. Execution control requires a working knowledge of the Integration Services runtime API. Understanding the subtleties of tight—and loose—coupling is not merely helpful, it can make or ruin your day (or data integration solution).

Why would anyone need an SSIS framework if SSIS 2012 and 2014 include the SSIS Catalog? That is an excellent question. The SSIS 2012 and 2014 Catalogs utilize the project deployment model—the default for SSIS projects developed in SQL Server Data Tools - Business Intelligence (SSDT-BI). But SSIS 2012 and 2014 also include the package deployment model so they support upgrading legacy SSIS projects to SSIS 2012 or 2014. In addition, there are use cases for using the SSIS Catalog for execution and monitoring and also for using a custom framework and the package deployment model. As a data integration architect, I am very grateful to the Microsoft SSIS Team for both options.

In this appendix, I will walk you through designing and building a custom, serial SSIS framework that I've named the SSIS Execution and Monitoring Framework. This framework will work with SSIS 2012/2014's package deployment model, complete with a SQL Server Reporting Services solution. Building an SSIS framework is an advanced task. I will help you build it from the ground up using many of the design patterns covered earlier in this book.

Starting in the Middle

We begin at the heart of execution control with the parent-child pattern. You need to start by creating a new SSIS solution and project named **SSISConfig2014**. Rename the default **Package.dtsx** to **Child1.dtsx**. Open the Child1 SSIS package and add a Script task to the control flow. Rename the Script task **Who Am I?** and open the Script task's editor. On the Script page, set the ScriptLanguage property to Microsoft Visual Basic 2012. Click the ellipsis in the ReadOnlyVariables property value textbox and add the **System::TaskName** and **System::PackageName** variables. Open the script editor and add the following code in **Sub Main()**.

Listing A-1. Sub Main from Who Am I? Script Task in Child1.dtsx Package

```
Public Sub Main()  
  
    Dim sPackageName As String = Dts.Variables("PackageName").Value.ToString  
    Dim sTaskName As String = Dts.Variables("TaskName").Value.ToString  
  
    MsgBox("I am " & sPackageName, , sTaskName)  
  
    Dts.TaskResult = ScriptResults.Success  
End Sub
```

The code shown in Listing A-1 pops up a message box that informs an observer of the name of the package from which the message box originates. This is reusable code. You can copy and paste this Script task into any SSIS package and it will perform the same way each time.

Now close the editor and execute the `Child1.dtsx` package in the SSDT debugger. When I execute the package, I see a message box similar to the one shown in Figure A-1.

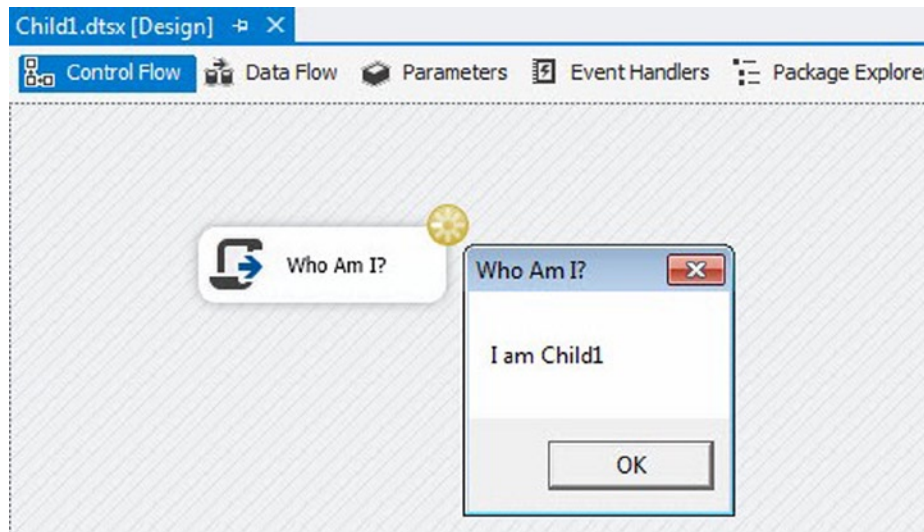


Figure A-1. Message box from `Child1.dtsx`

`Child.dtsx` will be your first test package. We will use `Child1.dtsx` going forward to conduct tests of the SSIS Execution and Monitoring Framework.

Before we proceed, change the deployment model for the SSIS from project deployment model—the default—to package deployment model. To accomplish the conversion, right-click the SSIS project in Solution Explorer and click `Convert to Package Deployment Model`, as shown in Figure A-2.

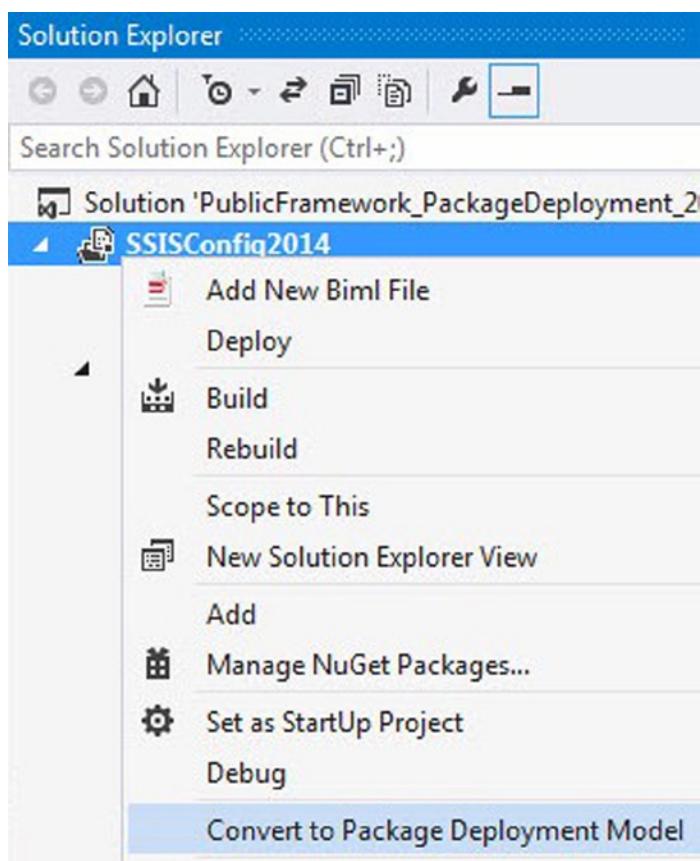


Figure A-2. Converting the project to the package deployment model

You will need to click the OK button on the dialog to acknowledge that you understand that this will change the features that are available for you to use in SSIS. Once the conversion is complete, you will see a result pane informing you that the project Child1.dtsx was converted to the package deployment model. The project in Solution Explorer will also indicate that the non-default deployment model has been selected, as shown in Figure A-3.

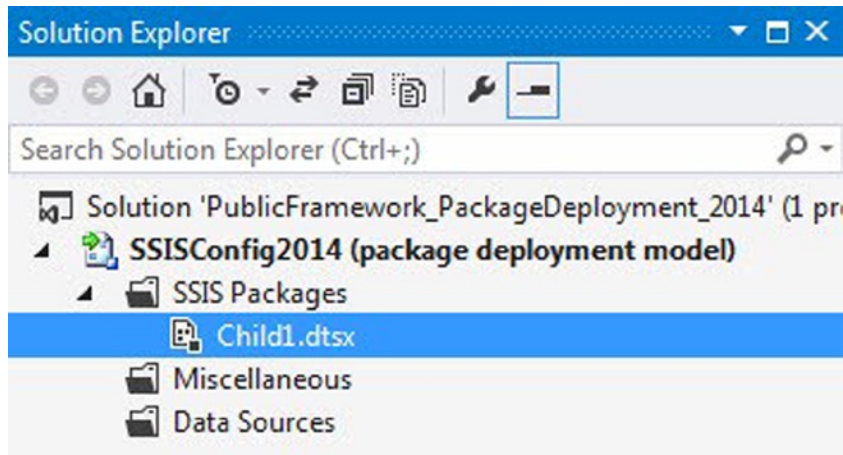


Figure A-3. Package deployment model

Add a new SSIS package and rename it **Parent.dtsx**. Add an Execute Package task to the control flow of **Parent.dtsx**. Rename the Execute Package task **Execute Child Package** and open the editor. On the Package page, set the Location property to **File System** and click the drop-down for the Connection property value. Click <New connection...> to configure a new file connection manager. Set the File Connection Manager Editor's Usage Type property to **Existing File**. Browse to the location of the **SSISConfig2014** project and select **Child1.dtsx**. Click the OK button to close the File Connection Manager Editor and click OK again to close the Execute Package Task Editor. Note the file connection manager that was created when you were configuring the Execute Package task. It is named **Child1.dtsx**; rename it **Child.dtsx**.

Test the **Parent.dtsx** package by executing it in the SSDT debugger. If all goes as planned, **Child1.dtsx** will execute and display the message box shown in Figure A-1. Acknowledge the message box and stop the debugger.

This is the parent-child pattern in action. You can improve upon the parent-child with a little metadata. How? I'm glad you asked. First, add an SSIS variable named **ChildPackagePath (String)**. Click on the **Child.dtsx** connection manager, and then press F4 to display properties. The **ConnectionString** property of the file connection manager is the path to the file. Select the **ConnectionString** property, copy it to the clipboard, and then paste it into the **Value** property of the **ChildPackagePath** SSIS variable. Return to the properties of the file connection manager named **Child.dtsx** and click the ellipsis in the **Value** textbox of the **Expressions** property. When the **Property Expressions Editor** displays, select **ConnectionString** from the **Property** drop-down, as shown in Figure A-4.

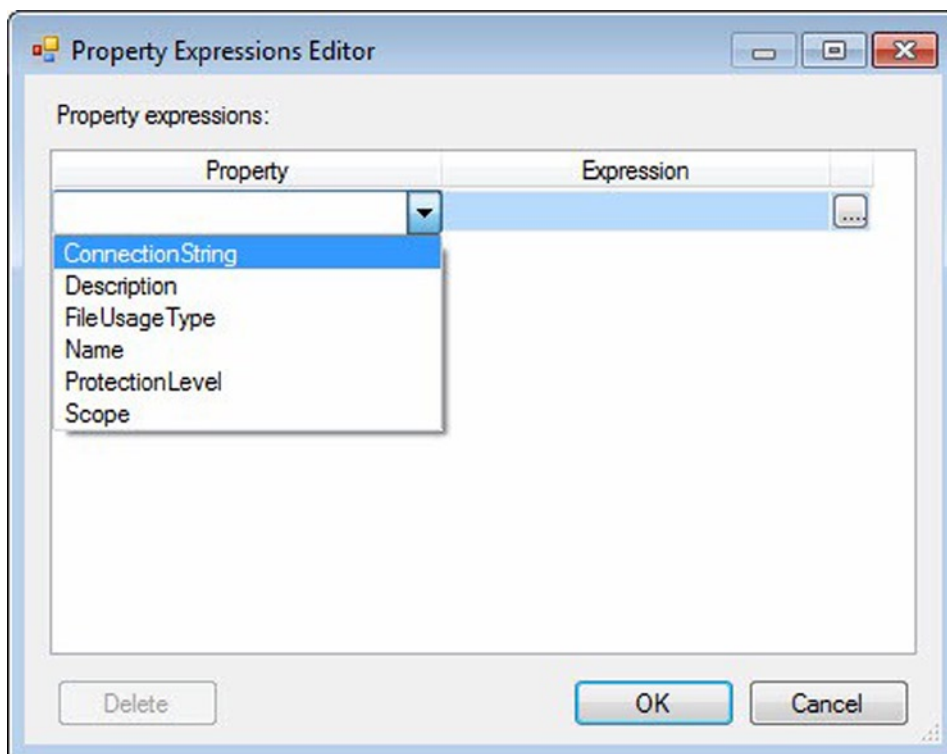


Figure A-4. The file connection manager *Property Expressions Editor*

Click the ellipsis in the Expression textbox beside `ConnectionString`. Expand the Variables and Parameters virtual folder in the upper left of the Expression Builder. Drag the variable `User::ChildPackagePath` from the virtual folder to the Expression textbox and click the Evaluate Expression button, as shown in Figure A-5.

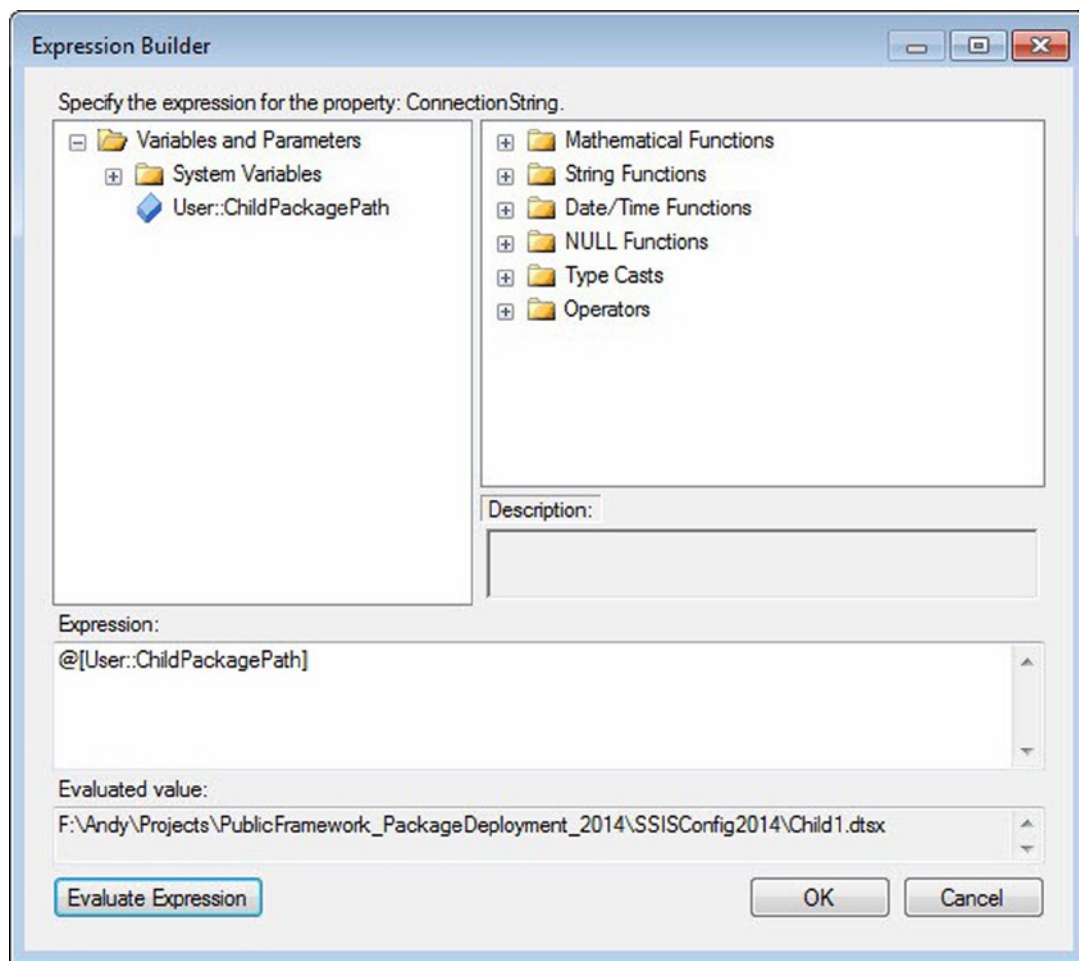


Figure A-5. Assigning the *User::ChildPackagePath* variable to the *ConnectionString* expression

Click the OK button to close the Expression Builder and then click the OK button to close the Property Expressions Editor. At this point, the *ConnectionString* property of the *Child.dtsx* file connection manager is managed by the *User::ChildPackagePath* SSIS variable. You can test this functionality by creating a second test child package. Fortunately, creating a second test child package is relatively simple.

In Solution Explorer, right-click the *Child1.dtsx* SSIS package and then click Copy. Right-click the SSIS Packages virtual folder and click Paste. Change the name of the new package from *Child1 1.dtsx* to **Child2.dtsx**.

Return to the *Parent.dtsx* package and change the value of the *ChildPackagePath* variable, substituting **Child2.dtsx** for *Child1.dtsx*. Execute *Parent.dtsx* in the SSDT debugger and observe the results, as shown in Figure A-6.



Figure A-6. Executing *Child2.dtsx* in the parent-child pattern

Pretty cool, huh? We're just getting started!

Let's create a database to hold package metadata. Open SQL Server Management Studio (SSMS) and execute the T-SQL script shown in Listing A-2.

Listing A-2. Creating the SSISConfig Database

```
Use master
go

/* SSISConfig database */
If Not Exists(Select name
              From sys.databases
              Where name = 'SSISConfig')
begin
    print 'Creating SSISConfig database'
    Create Database SSISConfig
    print 'SSISConfig database created'
end
Else
    print 'SSISConfig database already exists.'
print ''
go
```

The script in Listing A-2 is re-executable. Plus, it informs the person executing the script about its actions via Print statements. The first time you execute this script, you will see the following messages in the SSMS Messages tab:

```
Creating SSISConfig database
SSISConfig database created
```

The second time—and each subsequent time—you execute the same script, you will see this message:

SSISConfig database already exists.

Writing re-executable T-SQL is not always feasible, but when feasible, it is a good idea. Now that we have the database, we'll build a table to hold SSIS package metadata. Listing A-3 contains T-SQL for such a table.

Listing A-3. Building the cfg Schema and cfg.Packages Table

```
Use SSISConfig
go

/* cfg schema */
If Not Exists(Select name
              From sys.schemas
              Where name = 'cfg')
begin
    print 'Creating cfg schema'
    declare @sql varchar(100) = 'Create Schema cfg'
    exec(@sql)
    print 'Cfg schema created'
end
Else
    print 'Cfg schema already exists.'
print ''

/* cfg.Packages table */
If Not Exists(Select s.name + '.' + t.name
              From sys.tables t
              Join sys.schemas s
                On s.schema_id = t.schema_id
              Where s.name = 'cfg'
                And t.name = 'Packages')
begin
    print 'Creating cfg.Packages table'
    Create Table cfg.Packages
    (
        PackageID int identity(1,1)
        Constraint PK_Packages
        Primary Key Clustered
        ,PackageFolder varchar(255) Not Null
        ,PackageName varchar(255) Not Null
    )
    print 'Cfg.Packages created'
end
Else
    print 'Cfg.Packages table already exists.'
print ''
```

The script in Listing A-3 creates a schema named `cfg` if one doesn't already exist; it then creates a table named `cfg.Packages`, which contains three columns:

- `PackageID` is an identity column that serves as the primary key.
- `PackageFolder` is a `VarChar(255)` column that holds the path to the folder containing the SSIS package.
- `PackageName` is a `VarChar(255)` column that contains the name of the SSIS package.

I recently began identifying the stored procedures, functions, and views that support such a repository as a database programming interface (DPI), and not as an application programming interface, or API, because databases are *not* applications. Here, you should begin building the `SSISConfig` DPI with a stored procedure to load data into the `cfg.Packages` table, as shown in Listing A-4.

Listing A-4. The `cfg.AddSSISPackages` Stored Procedure

```

/* cfg.AddSSISPackage stored procedure */
If Exists(Select s.name + '.' + p.name
          From sys.procedures p
          Join sys.schemas s
            On s.schema_id = p.schema_id
          Where s.name = 'cfg'
            And p.name = 'AddSSISPackage')
begin
    print 'Dropping cfg.AddSSISPackage stored procedure'
    Drop Procedure cfg.AddSSISPackage
    print 'Cfg.AddSSISPackage stored procedure dropped'
end
print 'Creating cfg.AddSSISPackage stored procedure'
print ''
go

Create Procedure cfg.AddSSISPackage
    @PackageName varchar(255)
    ,@PackageFolder varchar(255)
    ,@PkgID int output
As

    Set NoCount On

    declare @tbl table (PkgID int)

    If Not Exists(Select PackageFolder + PackageName
                  From cfg.Packages
                  Where PackageFolder = @PackageFolder
                    And PackageName = @PackageName)
    begin
        Insert Into cfg.Packages
            (PackageName
            ,PackageFolder)
        Output inserted.PackageID Into @tbl
        Values (@PackageName, @PackageFolder)
    end

```

```

Else
    insert into @tbl
    (PkgID)
    (Select PackageID
     From cfg.Packages
     Where PackageFolder = @PackageFolder
     And PackageName = @PackageName)

    Select @PkgID = PkgID From @tbl
go
print 'Cfg.AddSSISPackage stored procedure created.'
print ''

```

Note the `cfg.AddSSISPackage` stored procedure returns an integer value that represents the identity column—`PackageID`—from the `cfg.Packages` table. You will use this integer value later. Once this stored procedure is in place, you can use the T-SQL script in Listing A-5 to add the packages in the project.

Listing A-5. Adding the Packages to the `cfg.Packages` Table

```

/* Variable Declaration */
declare @PackageFolder varchar(255) = F:\Andy\Projects\PublicFramework_PackageDeployment_2014\
SSISConfig2014\F:\Andy\Projects\
PublicFramework_PackageDeployment_2014\SSISConfig2014\
declare @PackageName varchar(255) = 'Child1.dtsx'
declare @PackageID int

/* Add the Child1.dtsx SSIS Package*/
If Not Exists(Select PackageFolder + PackageName
              From cfg.Packages
              Where PackageFolder = @PackageFolder
              And PackageName = @PackageName)
begin
    print 'Adding ' + @PackageFolder + @PackageName
    exec cfg.AddSSISPackage @PackageName, @PackageFolder, @PackageID output
end
Else
begin
    Select @PackageID = PackageID
    From cfg.Packages
    Where PackageFolder = @PackageFolder
    And PackageName = @PackageName
    print @PackageFolder + @PackageName + ' already exists in the Framework.'
end

set @PackageName = 'Child2.dtsx'
/* Add the Child2.dtsx SSIS Package*/
If Not Exists(Select PackageFolder + PackageName
              From cfg.Packages
              Where PackageFolder = @PackageFolder
              And PackageName = @PackageName)

```

```

begin
    print 'Adding ' + @PackageFolder + @PackageName
    exec cfg.AddSSISPackage @PackageName, @PackageFolder, @PackageID output
end
Else
begin
    Select @PackageID = PackageID
    From cfg.Packages
    Where PackageFolder = @PackageFolder
        And PackageName = @PackageName
    print @PackageFolder + @PackageName + ' already exists in the Framework.'
End

```

We now have enough to test the next step of my SSIS Execution and Monitoring Framework, so let's returning to SSDT. Add an Execute SQL task to the control flow and rename it **Get PackageMetadata**. Open the editor and change the ResultSet property to Single row. Change the ConnectionType property to ADO.Net. Click the drop-down in the Connection property and click <New connection...>. Configure an ADO.Net connection to the SSISConfig database. Set the SQLStatement property to the following T-SQL script:

```

Select PackageFolder + PackageName
From cfg.Packages
Where PackageName = 'Child1.dtsx'

```

On the Result Set page, add a resultset. Set the Result Name to 0 and the Variable Name to User::ChildPackagePath. Connect a precedence constraint between the Get Package Metadata Execute SQL task and the Execute Child Package Execute Package task. Execute the Parent.dtsx package to test it. What happens? The Get Package Metadata Execute SQL task runs a query that returns the full path to the Child1.dtsx package stored in the SSISConfig.cfg.Packages table. The returned path is sent into the ChildPackagePath variable. Remember, this variable controls the Child.dtsx file connection manager, which is used by the Execute Package task.

Alter the query in the Get Package Metadata Execute SQL task to return Child2.dtsx and retest.

Introducing SSIS Applications

An SSIS application is a collection of SSIS packages that execute in a prescribed order. Let's start by adding a couple of tables and supporting stored procedures to the SSISConfig database.

First, create a table named cfg.Applications, and a stored procedure to add them applications to the table, in SSISConfig using the T-SQL in Listing A-6.

Listing A-6. Building cfg.Applications and cfg.AddSSISApplication

```

/* cfg.Applications table */
If Not Exists(Select s.name + '.' + t.name
              From sys.tables t
              Join sys.schemas s
                On s.schema_id = t.schema_id
              Where s.name = 'cfg'
                And t.name = 'Applications')

```



```

begin
print 'Creating cfg.Applications table'
Create Table cfg.Applications
(
    ApplicationID int identity(1,1)
    Constraint PK_Applications
    Primary Key Clustered
    ,ApplicationName varchar(255) Not Null
    Constraint U_Applications_ApplicationName
    Unique
)
print 'Cfg.Applications created'
end
Else
print 'Cfg.Applications table already exists.'
print ''

/* cfg.AddSSISApplication stored procedure */
If Exists(Select s.name + '.' + p.name
    From sys.procedures p
    Join sys.schemas s
    On s.schema_id = p.schema_id
    Where s.name = 'cfg'
    And p.name = 'AddSSISApplication')
begin
print 'Dropping cfg.AddSSISApplication stored procedure'
Drop Procedure cfg.AddSSISApplication
print 'Cfg.AddSSISApplication stored procedure dropped'
end
print 'Creating cfg.AddSSISApplication stored procedure'
print ''
go

Create Procedure cfg.AddSSISApplication
    @ApplicationName varchar(255)
    ,@AppID int output
As

Set NoCount On

declare @tbl table (AppID int)

If Not Exists(Select ApplicationName
    From cfg.Applications
    Where ApplicationName = @ApplicationName)
begin
    Insert Into cfg.Applications
    (ApplicationName)
    Output inserted.ApplicationID into @tbl
    Values (@ApplicationName)
end

```

```

Else
    insert into @tbl
    (AppID)
    (Select ApplicationID
     From cfg.Applications
     Where ApplicationName = @ApplicationName)

    Select @AppID = AppID from @tbl
go
print 'cfg.AddSSISApplication stored procedure created.'
print ''

```

Note the `cfg.AddSSISApplication` stored procedure returns an integer value that represents the identity column—`ApplicationID`—from the `cfg.Applications` table. Add an SSIS application to the table using the following T-SQL in Listing A-7.

Listing A-7. Adding an SSIS Application

```

declare @ApplicationName varchar(255) = 'SSISApp1'
declare @ApplicationID int

/* Add the SSIS First Application */
If Not Exists(Select ApplicationName
              From cfg.Applications
              Where ApplicationName = @ApplicationName)
begin
    print 'Adding ' + @ApplicationName
    exec cfg.AddSSISApplication @ApplicationName, @ApplicationID output
    print @ApplicationName + ' added.'
end
Else
begin
    Select @ApplicationID = ApplicationID
    From cfg.Applications
    Where ApplicationName = @ApplicationName
    print @ApplicationName + ' already exists in the Framework.'
end
print ''

```

The script in Listing A-7 uses the `cfg.AddSSISApplication` stored procedure to add the `SSISApp1` SSIS application to the `cfg.Applications` table in the `SSISConfig` database.

A Note About Relationships

An SSIS application is a collection of SSIS packages that execute in a prescribed order, so it is pretty obvious that the relationship between an SSIS application and SSIS packages is one-to-many. What may not be as obvious is the relationship between SSIS packages and SSIS applications. Herein is a key benefit of choosing patterns-based development: code reusability, specifically in reference to the SSIS package code. Consider the archive file pattern from the end of Chapter 7 on flat file design patterns. In an enterprise that loads data from dozens or hundreds of flat file sources, this package may be called many times by different SSIS applications. So the relationship between SSIS

packages and SSIS applications is *also* one-to-many. If you do the set math, these relationships combine to create a many-to-many relationship between the applications' and packages' tables. This means you need a bridge or resolver table between them to create mappings between SSIS applications and SSIS packages.

I call this table `cfg.AppPackages`. Listing A-8 contains the T-SQL script that creates `cfg.AppPackages` and a stored procedure with which it is loaded.

Listing A-8. Creating `cfg.AppPackages` and `cfg.AddSSISApplicationPackage`

```
/* cfg.AppPackages table */
If Not Exists(Select s.name + '.' + t.name
              From sys.tables t
              Join sys.schemas s
                On s.schema_id = t.schema_id
              Where s.name = 'cfg'
                And t.name = 'AppPackages')
begin
  print 'Creating cfg.AppPackages table'
  Create Table cfg.AppPackages
  (
    AppPackageID int identity(1,1)
    Constraint PK_AppPackages
    Primary Key Clustered
    ,ApplicationID int Not Null
    Constraint FK_cfgAppPackages_cfgApplications_ApplicationID
    Foreign Key References cfg.Applications(ApplicationID)
    ,PackageID int Not Null
    Constraint FK_cfgAppPackages_cfgPackages_PackageID
    Foreign Key References cfg.Packages(PackageID)
    ,ExecutionOrder int Null
  )
  print 'Cfg.AppPackages created'
end
Else
  print 'Cfg.AppPackages table already exists.'
print ''

/* cfg.AddSSISApplicationPackage stored procedure */
If Exists(Select s.name + '.' + p.name
          From sys.procedures p
          Join sys.schemas s
            On s.schema_id = p.schema_id
          Where s.name = 'cfg'
            And p.name = 'AddSSISApplicationPackage')
begin
  print 'Dropping cfg.AddSSISApplicationPackage stored procedure'
  Drop Procedure cfg.AddSSISApplicationPackage
  print 'Cfg.AddSSISApplicationPackage stored procedure dropped'
end
print 'Creating cfg.AddSSISApplicationPackage stored procedure'
go
```

```

Create Procedure cfg.AddSSISApplicationPackage
    @ApplicationID int
    ,@PackageID int
    ,@ExecutionOrder int = 10
As

    Set NoCount On

    If Not Exists(Select AppPackageID
                  From cfg.AppPackages
                  Where ApplicationID = @ApplicationID
                    And PackageID = @PackageID)

    begin
        Insert Into cfg.AppPackages
        (ApplicationID
        ,PackageID
        ,ExecutionOrder)
        Values (@ApplicationID, @PackageID, @ExecutionOrder)
    end
go
print 'cfg.AddSSISApplicationPackage stored procedure created.'
print ''

```

To create the mappings between SSIS applications and SSIS packages, you need the IDs of each. Executing the following queries returns the information you need:

```

Select * from cfg.Applications
Select * from cfg.Packages

```

You will now use that information to execute the `cfg.AddSSISApplicationPackage` stored procedure, building `SSISApp1` in the metadata of the `SSISConfig` database and assigning it `Child1.dtsx` and `Child2.dtsx`—in that order. I use the T-SQL script shown in Listing A-9 to accomplish the mapping.

Listing A-9. Coupling the Child1 and Child2 SSIS Packages to the SSISApp1 SSIS Application

```

declare @ExecutionOrder int = 10
declare @ApplicationID int = 1
declare @PackageID int = 1
declare @ApplicationName varchar(255) = 'SSISApp1'
declare @PackageFolder varchar(255) = 'F:\Andy\Projects\PublicFramework_PackageDeployment_2014\
SSISConfig2014\'
declare @PackageName varchar(255) = 'Child1.dtsx'

If Not Exists(Select AppPackageID
              From cfg.AppPackages
              Where ApplicationID = @ApplicationID
                And PackageID = @PackageID
                And ExecutionOrder = @ExecutionOrder)

begin
    print 'Adding ' + @ApplicationName + '.' + @PackageName
      + ' to Framework with ExecutionOrder ' + convert(varchar, @ExecutionOrder)

```

```

exec cfg.AddSSISApplicationPackage @ApplicationID, @PackageID, @ExecutionOrder
print @PackageName + ' added and wired to ' + @ApplicationName
end
Else
print @ApplicationName + '.' + @PackageName
+ ' already exists in the Framework with ExecutionOrder '
+ convert(varchar, @ExecutionOrder)

/*Child2.dtsx */
set @PackageName = 'Child2.dtsx'
set @ExecutionOrder = 20
set @PackageID = 2

If Not Exists(Select AppPackageID
              From cfg.AppPackages
              Where ApplicationID = @ApplicationID
                And PackageID = @PackageID
                And ExecutionOrder = @ExecutionOrder)
begin
print 'Adding ' + @ApplicationName + '.' + @PackageName
+ ' to Framework with ExecutionOrder ' + convert(varchar, @ExecutionOrder)
exec cfg.AddSSISApplicationPackage @ApplicationID, @PackageID, @ExecutionOrder
print @PackageName + ' added and wired to ' + @ApplicationName
end
Else
print @ApplicationName + '.' + @PackageName
+ ' already exists in the Framework with ExecutionOrder '
+ convert(varchar, @ExecutionOrder)

```

One note about the T-SQL script shown in Listing A-9. This is not the way I would load this metadata into production (or even test) environments. I would not re-declare the ApplicationName, PackageFolder, PackageName, ApplicationID, and PackageID variables; rather, I would reuse these values from the previous T-SQL scripts. I alluded to this earlier when I mentioned we will use the ApplicationID and PackageID values later. I will provide a full T-SQL Metadata Load script later in this appendix.

Retrieving SSIS Applications in T-SQL

We now have SSIS application metadata stored in the SSISConfig database. Awesome, now what? It's time to build a stored procedure to return the SSIS package metadata we want for a given SSIS application. Listing A-10 contains the T-SQL data definition language (DDL) script to build such a stored procedure named `cfg.GetSSISApplication`.

Listing A-10. Creating the `cfg.GetSSISApplication` Stored Procedure

```

/* cfg.GetSSISApplication stored procedure */
If Exists(Select s.name + '.' + p.name
          From sys.procedures p
          Join sys.schemas s
            On s.schema_id = p.schema_id
          Where s.name = 'cfg'
            And p.name = 'GetSSISApplication')

```

```

begin
    print 'Dropping cfg.GetSSISApplication stored procedure'
    Drop Procedure cfg.GetSSISApplication
    print 'Cfg.GetSSISApplication stored procedure dropped'
end
print 'Creating cfg.GetSSISApplication stored procedure'
go

Create Procedure cfg.GetSSISApplication
    @ApplicationName varchar(255)
As

Select p.PackageFolder + p.PackageName As PackagePath
    , ap.ExecutionOrder
    , p.PackageName
    , p.PackageFolder
    , ap.AppPackageID
From cfg.AppPackages ap
Inner Join cfg.Packages p on p.PackageID = ap.PackageID
Inner Join cfg.Applications a on a.ApplicationID = ap.ApplicationID
Where ApplicationName = @ApplicationName
Order By ap.ExecutionOrder
go
print 'Cfg.GetSSISApplication stored procedure created.'
print ''

```

The `cfg.GetSSISApplication` stored procedure shown in Listing A-10 accepts a single parameter—`ApplicationName`—and uses this value to look up the SSIS packages associated with the SSIS application of that name. Note the columns that are returned:

- PackagePath
- ExecutionOrder
- PackageName
- PackagePath

Also note that the SSIS packages are returned in the order specified by `ExecutionOrder`.

Now test the stored procedure using the existing metadata in the `SSISConfig` database by executing the following T-SQL statement:

```
exec cfg.GetSSISApplication 'SSISApp1'
```

My results appear as shown in Figure A-7.



	PackagePath	ExecutionOrder	PackageName
1	F:\Andy\Projects\PublicFramework_PackageDeployment_2014\SSISConfig2014\Child1.dtsx	10	Child1.dtsx
2	F:\Andy\Projects\PublicFramework_PackageDeployment_2014\SSISConfig2014\Child2.dtsx	20	Child2.dtsx

Figure A-7. Results of `cfg.GetSSISApplication` statement

Figure A-7 shows the results of the stored procedure statement execution—a result containing two rows of data—and this data represents the SSIS package’s metadata associated with the SSIS application named `SSISApp1` in the `SSISConfig` database.

That was a lot of work! Fortunately, we will not need to repeat most of it. When you want to add SSIS packages and associate them with SSIS applications in the future, the script will look like the T-SQL shown in Listing A-11.

Listing A-11. The Complete T-SQL Script for Adding `SSISApp1` and Associated SSIS Packages

```
Use SSISConfig
go

/* Variable Declaration */
declare @PackageFolder varchar(255) = 'F:\Andy\Projects\PublicFramework_PackageDeployment_2014\
SSISConfig2014\'
declare @PackageName varchar(255) = 'Child1.dtsx'
declare @PackageID int
declare @ExecutionOrder int = 10

declare @ApplicationName varchar(255) = 'SSISApp1'
declare @ApplicationID int

/* Add the SSIS First Application */
If Not Exists(Select ApplicationName
              From cfg.Applications
              Where ApplicationName = @ApplicationName)
begin
    print 'Adding ' + @ApplicationName
    exec cfg.AddSSISApplication @ApplicationName, @ApplicationID output
    print @ApplicationName + ' added.'
end
Else
begin
    Select @ApplicationID = ApplicationID
    From cfg.Applications
    Where ApplicationName = @ApplicationName
    print @ApplicationName + ' already exists in the Framework.'
end
print ''
```

```

/* Add the Child1.dtsx SSIS Package*/
If Not Exists(Select PackageFolder + PackageName
              From cfg.Packages
              Where PackageFolder = @PackageFolder
              And PackageName = @PackageName)
begin
    print 'Adding ' + @PackageFolder + @PackageName
    exec cfg.AddSSISPackage @PackageName, @PackageFolder, @PackageID output
end
Else
begin
    Select @PackageID = PackageID
    From cfg.Packages
    Where PackageFolder = @PackageFolder
    And PackageName = @PackageName
    print @PackageFolder + @PackageName + ' already exists in the Framework.'
end

If Not Exists(Select AppPackageID
              From cfg.AppPackages
              Where ApplicationID = @ApplicationID
              And PackageID = @PackageID
              And ExecutionOrder = @ExecutionOrder)
begin
    print 'Adding ' + @ApplicationName + '.' + @PackageName
+ ' to Framework with ExecutionOrder ' + convert(varchar, @ExecutionOrder)
    exec cfg.AddSSISApplicationPackage @ApplicationID, @PackageID, @ExecutionOrder
    print @PackageName + ' added and wired to ' + @ApplicationName
end
Else
    print @ApplicationName + '.' + @PackageName
+ ' already exists in the Framework with ExecutionOrder '
+ convert(varchar, @ExecutionOrder)

/*Child2.dtsx */
set @PackageName = 'Child2.dtsx'
set @ExecutionOrder = 20

If Not Exists(Select PackageFolder + PackageName
              From cfg.Packages
              Where PackageFolder = @PackageFolder
              And PackageName = @PackageName)
begin
    print 'Adding ' + @PackageFolder + @PackageName
    exec cfg.AddSSISPackage @PackageName, @PackageFolder, @PackageID output
end
Else
begin
    Select @PackageID = PackageID
    From cfg.Packages

```



```

Where PackageFolder = @PackageFolder
And PackageName = @PackageName
print @PackageFolder + @PackageName + ' already exists in the Framework.'
end

If Not Exists(Select AppPackageID
              From cfg.AppPackages
              Where ApplicationID = @ApplicationID
                And PackageID = @PackageID
                And ExecutionOrder = @ExecutionOrder)
begin
  print 'Adding ' + @ApplicationName + '.' + @PackageName
+ ' to Framework with ExecutionOrder ' + convert(varchar, @ExecutionOrder)
  exec cfg.AddSSISApplicationPackage @ApplicationID, @PackageID, @ExecutionOrder
  print @PackageName + ' added and wired to ' + @ApplicationName
end
Else
  print @ApplicationName + '.' + @PackageName
+ ' already exists in the Framework with ExecutionOrder '
+ convert(varchar, @ExecutionOrder)

```

Retrieving SSIS Applications in SSIS

Return to SSDT-BI and open the editor for the Get Package Metadata Execute SQL task. Change the ResultSet property from Single Row to Full Result Set and change the SQLStatement property to `cfg.GetSSISApplication`. Set the IsQueryStoredProcedure property to True. On the Parameter Mapping page, click the Add button. Click the drop-down in the Variable Name column and select <New variable...> (you will probably need to scroll up to find <New variable...>). In the Add Variable window, make sure the Container property is set to Parent. Change the Name property to **ApplicationName**. The Namespace should be **User** and the Value Type property should be set to String. For the Value property, enter **SSISApp1**. The Add Variable window should appear as shown in Figure A-8.

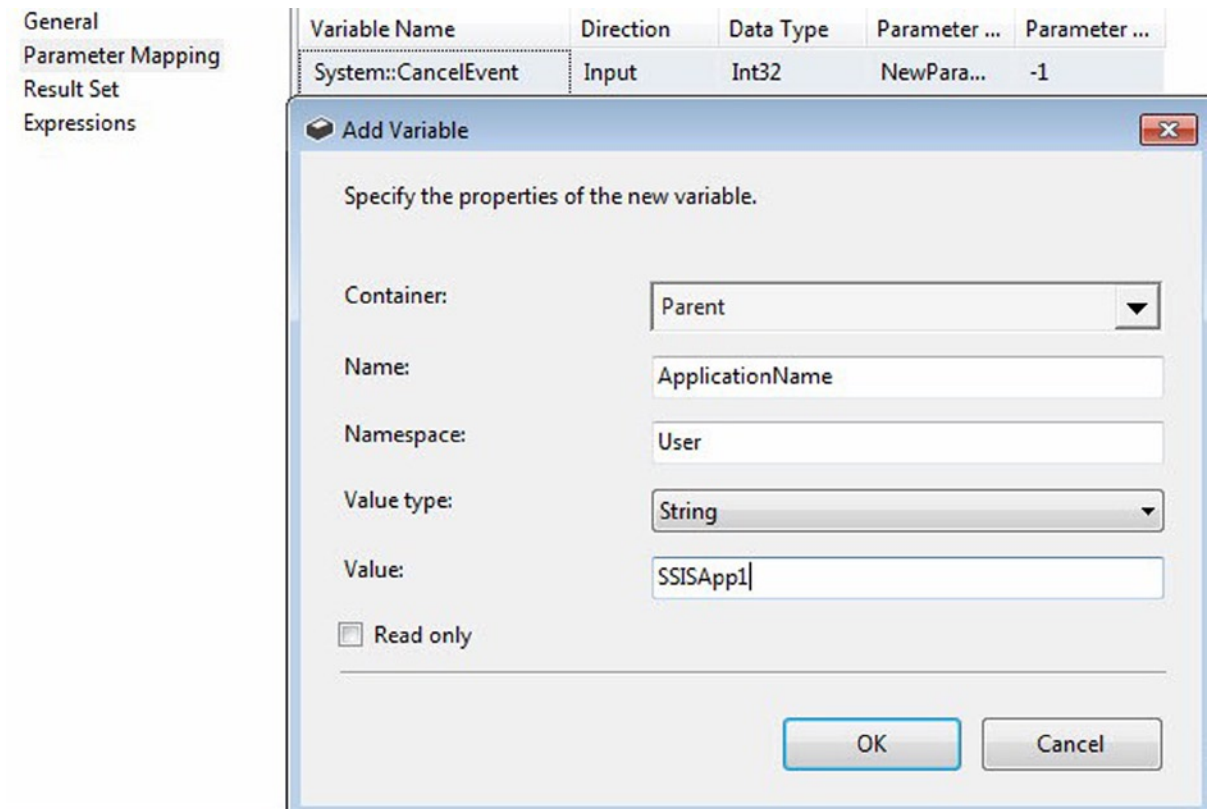


Figure A-8. Adding the *ApplicationName* variable

Click the OK button to close the Add Variable window and change the Data Type of the *ApplicationName* variable to String. Change the Parameter Name to *ApplicationName*. Navigate to the Result Set page and change the 0 result name variable from *User::ChildPackagePath* to a new variable with the following settings:

- Container: Parent
- Name: Packages
- Namespace: User
- Value Type: Object

Click the OK button to close the Add Variable window, and click the OK button to close the Execute SQL Task Editor. Delete the precedence constraint between the Get Package Metadata Execute SQL task and the Execute Child Package Execute Package task. Drag a Foreach Loop container onto the control flow and then drag the Execute Child Package Execute Package task inside it. Add a precedence constraint from the Get Package Metadata Execute SQL task to the new Foreach Loop container, and rename the Foreach Loop container **Foreach Child Package**. Open the Foreach Child Package Foreach Loop container's editor and navigate to the Collection page. Change the Enumerator to Foreach ADO Enumerator. In the ADO Object Source Variable drop-down, select the *User::Packages* variable. Accept the default Enumeration Mode: Rows in the First Table.

Navigate to the Variable Mappings page in the Foreach Loop Editor. Click on the Variable drop-down and select the *User::ChildPackagePath* variable. The Index property will default to 0—do not change it.

The changes we just made accomplish the following:

1. Execute the `cfg.GetSSISApplications` stored procedure in the `SSISConfig` database, passing it the value contained in the `ApplicationName` variable.
2. Push the full result set returned by the stored procedure execution into an SSIS object variable named `Packages`.
3. Configure a Foreach Loop to point at each row stored in the `Packages` variable in the order returned.
4. Push the value contained in the first column (Column 0) of the row to which the Foreach Loop points into the `User::ChildPackagePath` variable.

When the value of the `ChildPackagePath` variable changes, the `ConnectionString` property of the `Child.dtsx` file connection manager is dynamically updated, aiming the connection manager at the path contained in `User::ChildPackagePath`.

Click the OK button to close the Foreach Loop Container Editor and execute the `Parent.dtsx` SSIS package in the SSDT debugger. When you do this, you get two message boxes. The first states “I am Child1” and the second appears as shown in Figure A-9.

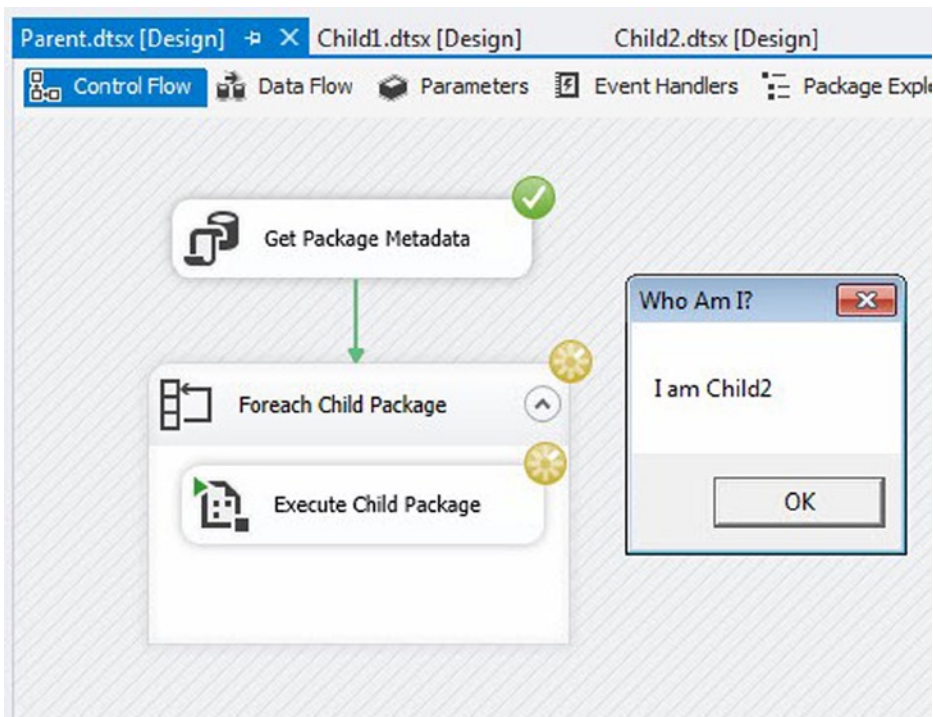


Figure A-9. Executing a test serial SSIS framework

This code, as it stands, composes an SSIS execution framework. The database contains the metadata and the parent package executes the SSIS packages. Monitoring is next.

Monitoring Execution

Most experienced business intelligence developers will tell you to start with the reports and work your way back to the source data. The source data in this particular case is information collected from the data integration process. What kind of information? Things like start and end execution times, execution status, and error and event messages.

Instance data is recorded for each SSIS application and SSIS package execution. Each entry represents an execution, and there are two tables that hold these entries: `Log.SSISAppInstance` holds execution metrics about SSIS application instances, and `Log.SSISPackageInstance` holds execution metrics for SSIS child package instances. When an SSIS application starts, a row is inserted into the `Log.SSISAppInstance` table. When the SSIS application completes, the row is updated. `Log.SSISPackageInstance` works the same way for each SSIS package in an SSIS application. An SSIS application instance is logically comprised of an application ID and a start time. An SSIS package instance is comprised of an application instance ID, application package ID, and a start time.

Error and event logging is relatively straightforward. You store a description of the error or event, the time it occurred, and the instance IDs. That's what the reports will reflect, and that's all there is to logging.

Building Application Instance Logging

Let's return to SSMS to build the tables and stored procedures to support logging. You need to execute the T-SQL script shown in Listing A-12 to build the instance tables and stored procedures.

Listing A-12. Building the Application Instance Tables and Stored Procedures

```
/* log schema */
If Not Exists(Select name
              From sys.schemas
              Where name = 'log')
begin
    print 'Creating log schema'
    declare @sql varchar(100) = 'Create Schema [log]'
    exec(@sql)
    print 'Log schema created'
end
Else
    print 'Log schema already exists.'
print ''

/* log.SSISAppInstance table */
If Not Exists(Select s.name + '.' + t.name
              From sys.tables t
              Join sys.schemas s
                On s.schema_id = t.schema_id
              Where s.name = 'log'
              And t.name = 'SSISAppInstance')
begin
    print 'Creating log.SSISAppInstance table'
    Create Table [log].SSISAppInstance
    (
        AppInstanceID int identity(1,1)
        Constraint PK_SSISAppInstance
        Primary Key Clustered
        ,ApplicationID int Not Null
```

```

Constraint FK_logSSISAppInstance_cfgApplication_ApplicationID
Foreign Key References cfg.Applications(ApplicationID)
,StartDate datetime Not Null
    Constraint DF_cfgSSISAppInstance_StartDateTime
    Default(GetDate())
,EndDate datetime Null
,[Status] varchar(12) Null
)
print 'Log.SSISAppInstance created'
end
Else
    print 'Log.SSISAppInstance table already exists.'
print ''

/* log.LogStartOfApplication stored procedure */
If Exists(Select s.name + '.' + p.name
    From sys.procedures p
    Join sys.schemas s
    On s.schema_id = p.schema_id
    Where s.name = 'log'
    And p.name = 'LogStartOfApplication')
begin
    print 'Dropping log.LogStartOfApplication stored procedure'
    Drop Procedure [log].LogStartOfApplication
    print 'Log.LogStartOfApplication stored procedure dropped'
end
print 'Creating log.LogStartOfApplication stored procedure'
go

Create Procedure [log].LogStartOfApplication
    @ApplicationName varchar(255)
As

declare @ErrMsg varchar(255)
declare @AppID int = (Select ApplicationID
    From cfg.Applications
    Where ApplicationName = @ApplicationName)

If (@AppID Is Null)
begin
    set @ErrMsg = 'Cannot find ApplicationName ' + Coalesce(@ApplicationName, '<NULL>')
    raiserror(@ErrMsg,16,1)
    return-1
end

Insert Into [log].SSISAppInstance
(ApplicationID, StartDateTime, Status)
Output inserted.AppInstanceID
Values
(@AppID, GetDate(), 'Running')
go
print 'Log.LogStartOfApplication stored procedure created.'
print ''

```

```

/* log.LogApplicationSuccess stored procedure */
If Exists(Select s.name + '.' + p.name
          From sys.procedures p
          Join sys.schemas s
            On s.schema_id = p.schema_id
          Where s.name = 'log'
          And p.name = 'LogApplicationSuccess')
begin
    print 'Dropping log.LogApplicationSuccess stored procedure'
    Drop Procedure [log].LogApplicationSuccess
    print 'Log.LogApplicationSuccess stored procedure dropped'
end
print 'Creating log.LogApplicationSuccess stored procedure'
go

Create Procedure [log].LogApplicationSuccess
    @AppInstanceID int
As

    update log.SSISAppInstance
    set EndDateTime = GetDate()
      , Status = 'Success'
    where AppInstanceID = @AppInstanceID
go
print 'Log.LogApplicationSuccess stored procedure created.'
print ''

/* log.LogApplicationFailure stored procedure */
If Exists(Select s.name + '.' + p.name
          From sys.procedures p
          Join sys.schemas s
            On s.schema_id = p.schema_id
          Where s.name = 'log'
          And p.name = 'LogApplicationFailure')
begin
    print 'Dropping log.LogApplicationFailure stored procedure'
    Drop Procedure [log].LogApplicationFailure
    print 'Log.LogApplicationFailure stored procedure dropped'
end
print 'Creating log.LogApplicationFailure stored procedure'
go

Create Procedure [log].LogApplicationFailure
    @AppInstanceID int
As

    update log.SSISAppInstance
    set EndDateTime = GetDate()
      , Status = 'Failed'
    where AppInstanceID = @AppInstanceID
go
print 'Log.LogApplicationFailure stored procedure created.'
print ''

```

Now let's return to SSDT and add application instance logging to the `Parent.dtsx` package. Drag a new Execute SQL task to the control flow and rename it **Log Start of Application**. Set the `ResultSet` property to Single Row. Set the `ConnectionType` property to ADO.Net and `Connection` to the `SSISConfig` connection manager. Set the `SQLStatement` property to `log.LogStartOfApplication` and the `IsQueryStoredProcedure` property to True. Navigate to the Parameter Mapping page and add a new parameter: mapping the `User::ApplicationName` SSIS variable to the `ApplicationName` parameter for the `log.LogStartOfApplication` stored procedure. Change the `Data Type` property of the `ApplicationName` parameter to String. On the Result Set page, add a new result named 0 and map it to a new Int32 variable named `AppInstanceID`. Set the default value of the `AppInstanceID` variable to 0. Close the Execute SQL Task Editor and connect a precedence constraint from the Log Start of Application Execute SQL task to the Get Package Metadata Execute SQL task.

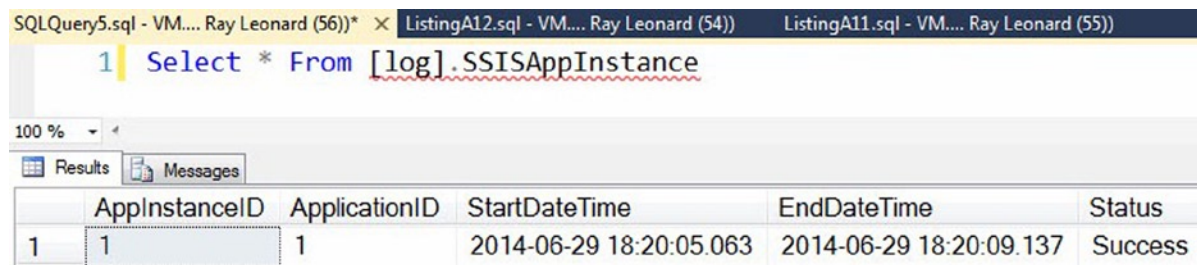
Drag another Execute SQL task onto the control flow beneath the Foreach Child Package Foreach Loop container and rename it **Log Application Success**. Open the editor, change the `ConnectionType` property to ADO.Net, and set the `Connection` property to the `SSISConfig` connection manager. Enter `log.LogApplicationSuccess` in the `SQLStatement` property and set the `IsQueryStoredProcedure` property to True. Navigate to the Parameter Mapping page and add a mapping between the `User::AppInstanceID` SSIS variable and the `Int32 AppInstanceID` parameter for the `log.LogApplicationSuccess` stored procedure. Close the Execute SQL Task Editor and connect a precedence constraint from the Foreach Child Package Foreach Loop container to the Log Application Success Execute SQL task.

What did you just accomplish? You added SSIS application instance logging to the control flow of the `Parent.dtsx` SSIS package. To test this, you need to execute `Parent.dtsx` in the SSDT debugger.

Once execution completes, execute the following query to observe the logged results:

```
Select * From [log].SSISAppInstance
```

When I execute this query, I get the results that are shown in Figure A-10.

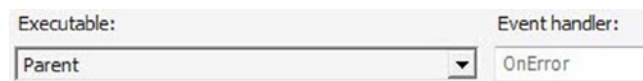


	AppInstanceID	ApplicationID	StartDateTime	EndDateTime	Status
1	1	1	2014-06-29 18:20:05.063	2014-06-29 18:20:09.137	Success

Figure A-10. Observing the results of querying the application instance log

What happens when an SSIS application fails? You want to update the `log.SSISAppInstance` row with an `EndDateTime` and set the Status to Failed. For this, you will use an Execute SQL task configured to execute the `log.LogApplicationFailure` stored procedure. The question is: Where? The answer is: the `Parent.dtsx` package's `OnError` event handler.

In SSDT, click the Event Handlers tab on `Parent.dtsx`. In the Executable drop-down, select `Parent`; in the Event Handler drop-down, select `OnError` as shown in Figure A-11.



Executable:	Event handler:
Parent	OnError

Figure A-11. Configuring the Parent package's OnError event handler

Click the Click Here to Create an “OnError” Event Handler for Executable “Parent” link on the surface of the event handler to create the OnError event handler for the Parent.dtsx package. I could walk you through building another Execute SQL task to log the SSIS application failure; however, it’s easier and simpler to just copy the Log Application Success Execute SQL task from the bottom of the control flow and paste it into the Parent.dtsx OnError event handler. When you do, change the name to **Log Application Failure** and the SQLStatement property to log.LogApplicationFailure.

You are now ready to test, but you have no real way to test the application failure unless you modify a package—and that just seems tragic. You’re likely going to need to test errors after this, too. So why not build an ErrorTest.dtsx SSIS package and add it to the SSIS application? I like this plan!

Create a new SSIS package and rename it ErrorTest.dtsx. Add a Script task to the control flow and rename it **Succeed or Fail?** Change the ScriptLanguage property to Microsoft Visual Basic 2012. Open the editor and add the System::TaskName and System::PackageName variables to the ReadOnlyVariables property. Open the Script Editor and add the code shown in Listing A-13 to SubMain().

Listing A-13. Code to Succeed or Fail SSIS Package

```
Public Sub Main()

    Dim sPackageName As String = Dts.Variables("PackageName").Value.ToString
    Dim sTaskName As String = Dts.Variables("TaskName").Value.ToString
    Dim sSubComponent As String = sPackageName & "." & sTaskName

    Dim iResponse As Integer = MsgBox("Succeed Package?", MsgBoxStyle.YesNo, sSubComponent)
    If iResponse = vbYes Then
        Dts.TaskResult = ScriptResults.Success
    Else
        Dts.TaskResult = ScriptResults.Failure
    End If

End Sub
```

Unit-test by executing ErrorTest.dtsx in the SSDT debugger, as shown in Figure A-12.

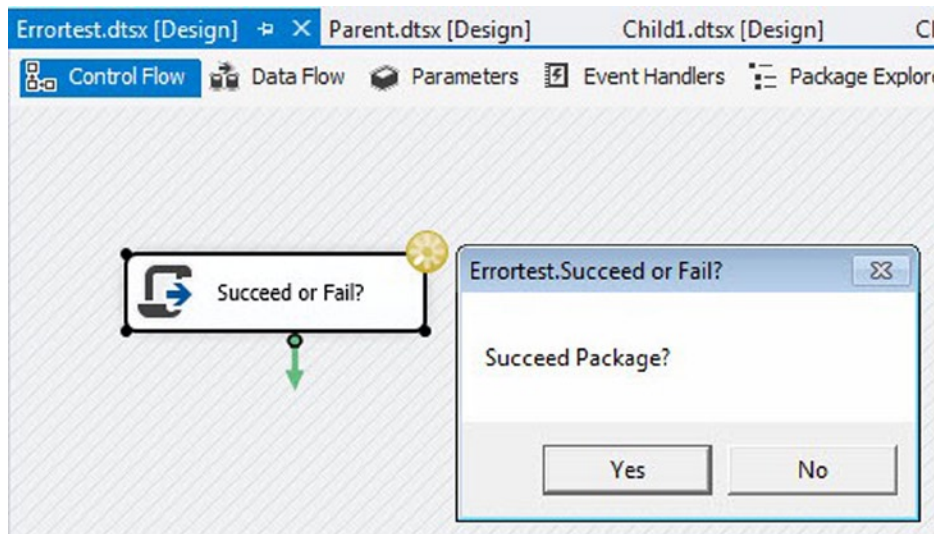


Figure A-12. Unit-testing the ErrorTest.dtsx SSIS package

To add this SSIS package to the SSISApp1 SSIS application, append the T-SQL script in Listing A-14 to the T-SQL script in Listing A-11.

Listing A-14. Append This T-SQL Script to Listing A-11 to Add the ErrorTest.dtsx SSIS Package to the SSISApp1 SSIS Application

```
/*ErrorTest.dtsx */

/* Variable Declaration */
declare @PackageFolder varchar(255) = 'F:\Andy\Projects\PublicFramework_PackageDeployment_2014\
SSISConfig2014\'
declare @PackageName varchar(255) = 'Child1.dtsx'
declare @PackageID int
declare @ExecutionOrder int = 10

declare @ApplicationName varchar(255) = 'SSISApp1'
declare @ApplicationID int

/* Add the SSIS First Application */
If Not Exists(Select ApplicationName
              From cfg.Applications
              Where ApplicationName = @ApplicationName)
begin
    print 'Adding ' + @ApplicationName
    exec cfg.AddSSISApplication @ApplicationName, @ApplicationID output
    print @ApplicationName + ' added.'
end
Else
begin
    Select @ApplicationID = ApplicationID
    From cfg.Applications
    Where ApplicationName = @ApplicationName
    print @ApplicationName + ' already exists in the Framework.'
end
print ''

set @PackageName = 'ErrorTest.dtsx'
set @ExecutionOrder = 30

If Not Exists(Select PackageFolder + PackageName
              From cfg.Packages
              Where PackageFolder = @PackageFolder
                And PackageName = @PackageName)
begin
    print 'Adding ' + @PackageFolder + @PackageName
    exec cfg.AddSSISPackage @PackageName, @PackageFolder, @PackageID output
end
Else
begin
    Select @PackageID = PackageID
    From cfg.Packages
```

```

Where PackageFolder = @PackageFolder
And PackageName = @PackageName
print @PackageFolder + @PackageName + ' already exists in the Framework.'
end

If Not Exists(Select AppPackageID
              From cfg.AppPackages
              Where ApplicationID = @ApplicationID
                And PackageID = @PackageID
                And ExecutionOrder = @ExecutionOrder)
begin
  print 'Adding ' + @ApplicationName + '.' + @PackageName
+ ' to Framework with ExecutionOrder ' + convert(varchar, @ExecutionOrder)
  exec cfg.AddSSISApplicationPackage @ApplicationID, @PackageID, @ExecutionOrder
  print @PackageName + ' added and wired to ' + @ApplicationName
end
Else
  print @ApplicationName + '.' + @PackageName
+ ' already exists in the Framework with ExecutionOrder '
+ convert(varchar, @ExecutionOrder)

```

Now open `Parent.dtsx` and execute it in the SSDT debugger. Once prompted by the `ErrorTest.dtsx` message box, click the No button to cause the `ErrorTest.dtsx` to fail. This should cause the `Parent.dtsx` package `OnError` event handler to fire, as shown in Figure A-13.

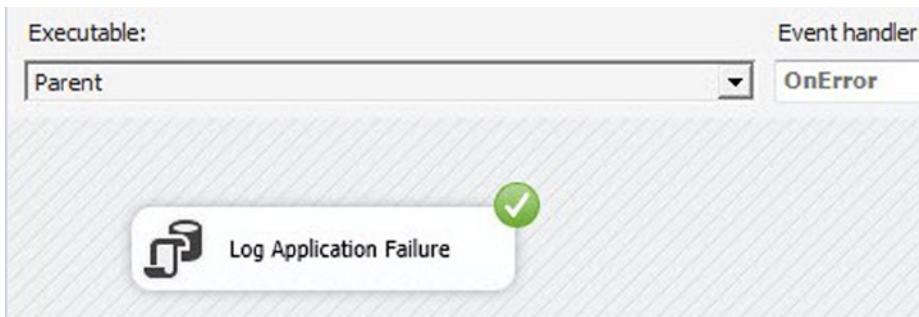


Figure A-13. *I have mixed emotions about successful OnError event handlers*

A couple successful and failed executions later, the `log.SSISAppInstance` table contains the rows shown in Figure A-14

	AppInstanceID	ApplicationID	StartDateTime	EndDateTime	Status
1	1	1	2014-06-29 18:20:05.063	2014-06-29 18:20:09.137	Success
2	2	1	2014-06-29 18:31:55.310	2014-06-29 18:32:20.267	Failed
3	3	1	2014-06-29 18:34:25.557	2014-06-29 18:34:38.683	Failed
4	4	1	2014-06-29 18:34:48.180	2014-06-29 18:34:55.200	Success

Figure A-14. Successes and failures of SSISApp1

That's a wrap on application instance logging! Next, let's build out child package instance logging.

Building Package Instance Logging

Package instance logging works like application instance logging, only on a different scale. An application instance consists of an application ID and an execution start time. A package instance consists of an application package ID, an application instance ID, and an execution start time.

Let's start by creating the log.SSISPkgInstance table and stored procedures. Listing A-15 contains these database objects.

Listing A-15. Building the Package Instance Logging Table and Stored Procedures

```

/* log.SSISPkgInstance table */
If Not Exists(Select s.name + '.' + t.name
              From sys.tables t
              Join sys.schemas s
                On s.schema_id = t.schema_id
              Where s.name = 'log'
                And t.name = 'SSISPkgInstance')
begin
  print 'Creating log.SSISPkgInstance table'
  Create Table [log].SSISPkgInstance
  (
    PkgInstanceID int identity(1,1)
    Constraint PK_SSISPkgInstance Primary Key Clustered
    ,AppInstanceID int Not Null
    Constraint FK_logSSISPkgInstance_logSSISAppInstance_AppInstanceID
    Foreign Key References [log].SSISAppInstance(AppInstanceID)
    ,AppPackageID int Not Null
    Constraint FK_logSSISPkgInstance_cfgAppPackages_AppPackageID
    Foreign Key References cfg.AppPackages(AppPackageID)
    ,StartDateTime datetime Not Null
    Constraint DF_cfgSSISPkgInstance_StartDateTime
    Default(GetDate())
    ,EndDateTime datetime Null
    ,[Status] varchar(12) Null
  )

```

```

    print 'Log.SSISPackageInstance created'
end
Else
    print 'Log.SSISPackageInstance table already exists.'
print ''

/* log.LogStartOfPackage stored procedure */
If Exists(Select s.name + '.' + p.name
          From sys.procedures p
          Join sys.schemas s
            On s.schema_id = p.schema_id
          Where s.name = 'log'
            And p.name = 'LogStartOfPackage')
begin
    print 'Dropping log.LogStartOfPackage stored procedure'
    Drop Procedure [log].LogStartOfPackage
    print 'Log.LogStartOfPackage stored procedure dropped'
end
print 'Creating log.LogStartOfPackage stored procedure'
go

Create Procedure [log].LogStartOfPackage
    @AppInstanceID int
    ,@AppPackageID int
As

declare @ErrMsg varchar(255)

Insert Into log.SSISPackageInstance
    (AppInstanceID, AppPackageID, StartDateTime, Status)
Output inserted.PkgInstanceID
Values
    (@AppInstanceID, @AppPackageID, GetDate(), 'Running')
go
print 'Log.SSISPackageInstance stored procedure created.'
print ''

/* log.LogPackageSuccess stored procedure */
If Exists(Select s.name + '.' + p.name
          From sys.procedures p
          Join sys.schemas s
            On s.schema_id = p.schema_id
          Where s.name = 'log'
            And p.name = 'LogPackageSuccess')
begin
    print 'Dropping log.LogPackageSuccess stored procedure'
    Drop Procedure [log].LogPackageSuccess
    print 'Log.LogPackageSuccess stored procedure dropped'
end
print 'Creating log.LogPackageSuccess stored procedure'
go

```

```

Create Procedure [log].LogPackageSuccess
    @PkgInstanceID int
As

    update log.SSISPackageInstance
    set EndDateTime = GetDate()
      , Status = 'Success'
    where PkgInstanceID = @PkgInstanceID
go
print 'Log.LogPackageSuccess stored procedure created.'
print ''

/* log.LogPackageFailure stored procedure */
If Exists(Select s.name + '.' + p.name
          From sys.procedures p
          Join sys.schemas s
            On s.schema_id = p.schema_id
          Where s.name = 'log'
            And p.name = 'LogPackageFailure')
begin
    print 'Dropping log.LogPackageFailure stored procedure'
    Drop Procedure [log].LogPackageFailure
    print 'Log.LogPackageFailure stored procedure dropped'
end
print 'Creating log.LogPackageFailure stored procedure'
go

Create Procedure [log].LogPackageFailure
    @PkgInstanceID int
As

    update log.SSISPackageInstance
    set EndDateTime = GetDate()
      , Status = 'Failed'
    where PkgInstanceID = @PkgInstanceID
go
print 'Log.LogPackageFailure stored procedure created.'
print ''

```

The log.SSISPackageInstance table will hold the SSIS package instance data. Log.LogStartofPackage inserts a row into the SSISPackageInstance table; log.LogPackageSuccess updates the row with an EndDateTime and a 'Success' status, while log.LogPackageFailure updates the record with an EndDateTime and a 'Failed' status.

In Parent.dtsx, open the editor for the Foreach Child Package Foreach Loop container. Navigate to the Variable Mappings page and add a new variable. Configure the following settings in the Add Variable window:

- Container: Parent
- Name: AppPackageID
- Namespace: User
- Value Type: Int32
- Value: 0

Click the OK button to close the Add Variable window. The AppInstanceID—which exists in the dataset inside the User::Packages SSIS variable—is returned from executing the `cfg.GetSSISApplication` stored procedure. The AppPackageID column is returned as the fifth column. Therefore, the AppPackageID variable's Index column on the Variable Mappings page of the Foreach Child Package Foreach Loop container should be set to 4 (the fifth value in a 0-based array). Click the OK button to close the Foreach Child Package Foreach Loop Container Editor.

Add an Execute SQL task to the Foreach Child Package Foreach Loop container. Rename the new Execute SQL task **Log Start of Package**. Open the editor and set the ResultSet property to Single Row. Set the ConnectionType property to ADO.Net and the Connection to the SSISConfig connection manager. Set the SQLStatement property to `log.LogStartOfPackage` and the IsQueryStoredProcedure property to True. Navigate to the Parameter Mapping page and add two new parameters:

- Variable Name: User::AppInstanceID
- Direction: Input
- Data Type: Int32
- Parameter Name: AppInstanceID
- Variable Name: User::AppPackageID
- Direction: Input
- Data Type: Int32
- Parameter Name: AppPackageID

On the Result Set page, add a new result named **0** and map it to a new Parent-level Int32 variable named PkgInstanceID, with a default value of 0. Close the Execute SQL Task Editor. Connect a precedence constraint from the Log Start of Package Execute SQL task to the Execute Child Package Execute Package task.

Add two more Execute SQL tasks to the Foreach Child Package Foreach Loop container. Rename the first **Log Package Success**, set the connection properties from the ADO.Net connection manager used to connect to the SSISConfig database, the SQLStatement property to `log.LogPackageSuccess`, and the IsQueryStoredProcedure property to True. On the Parameter Mapping page, add a parameter and map the User::PkgInstanceID variable to the PkgInstanceID parameter for the `log.LogStartofPackage` stored procedure. Connect a precedence constraint (OnSuccess) from the Execute Child Package Execute Package task to the Log Package Success Execute SQL task.

Rename the second Log Package Failure, set the connection properties from the ADO.Net connection manager used to connect to the SSISConfig database, set the SQLStatement property to `log.LogPackageFailure`, and set the IsQueryStoredProcedure property to True. On the Parameter Mapping page, add a parameter and map the User::PkgInstanceID variable to the PkgInstanceID parameter for the `log.LogStartofPackage` stored procedure. Connect a precedence constraint (OnFailure) from the Execute Child Package Execute Package task to the Log Package Failure Execute SQL task.

Test the package instance logging by running a few test executions. Allow one to succeed and the other to fail. When you check the Application and Package Instance tables, the results should appear as shown in Figure A-15.

	AppInstanceID	ApplicationID	StartDateTime	EndDateTime	Status
1	8	1	2014-06-29 18:47:24.023	2014-06-29 18:47:28.943	Success
2	7	1	2014-06-29 18:47:15.030	2014-06-29 18:47:20.157	Failed
3	6	1	2014-06-29 18:46:58.347	2014-06-29 18:47:04.320	Failed
4	5	1	2014-06-29 18:46:46.060	2014-06-29 18:46:52.307	Success

	PkgInstanceID	AppInstanceID	AppPackageID	StartDateTime	EndDateTime	Status
1	12	8	5	2014-06-29 18:47:26.930	2014-06-29 18:47:28.920	Success
2	11	8	2	2014-06-29 18:47:24.863	2014-06-29 18:47:26.833	Success
3	10	8	1	2014-06-29 18:47:24.053	2014-06-29 18:47:24.853	Success
4	9	7	5	2014-06-29 18:47:18.250	2014-06-29 18:47:20.223	Failed
5	8	7	2	2014-06-29 18:47:16.003	2014-06-29 18:47:18.227	Success
6	7	7	1	2014-06-29 18:47:15.060	2014-06-29 18:47:15.987	Success
7	6	6	5	2014-06-29 18:47:01.483	2014-06-29 18:47:04.363	Failed

Figure A-15. Examining the application and package instance logs

You can tell by examining the Application Instance and Package Instance log tables that AppInstanceID8 started at 6:47:24 PM 29 Jun 2014. You can also see three SSIS packages—with PkgInstanceID's 10, 11, and 12—were executed as part of the SSIS application. Each package succeeded, and the SSIS application succeeded as well. You also know AppInstanceID7 started at 6:14:15 PM 29 Jun 2014 and executed PkgInstanceID's 7, 8, and 9. PkgInstanceID's 7 and 8 succeeded, but PkgInstanceID 9 failed; failing the SSIS application.

Cool? Cool. Let's move to error and event logging.

Building Error Logging

Instrumenting data integration processes to capture and preserve error and exception metadata is the most important and useful type of logging. Exceptions and errors are going to happen. SSIS provides a fairly robust model for capturing and reporting errors as long as you realize you can mostly ignore the error codes. The error descriptions, however, are mostly good. So it balances out.

Before I demonstrate how to capture error messages in SSIS, let's discuss why. I used to manage a team of data integration developers. The team ranged in size from 28 to 40 developers, and I built very large ETL solutions for US state government interests. Part of my job was to figure out best practices. Having all SSIS packages log error data in the same format to the same location is a best practice. But how do you do this with 40 developers? Have you ever tried to get 40 developers to do the same thing the same way? It's like herding cats. The problem was half of them thought they were smarter than me; and half of those were correct in thinking that. But this wasn't the kind of problem that required deep thinking; this required strategy. So what's the best strategy for getting every developer to build the exact same kind of log for every SSIS package every time? You guessed it: don't let them. Take error logging completely out of their hands.

Soon after learning how to use the Execute Package task, I learned events "bubble" from child to parent packages. For the purposes of error logging, this means I can capture and record any error at the parent package. Even better, it means I can do this *with no code in the child package*. Problem solved.

Let's take a look at how to implement this functionality into an SSIS framework. First, let's add a table and a stored procedure to record and preserve errors, as shown in Listing A-16.

Listing A-16. Building the Error Logging Table and Stored Procedure

```

/* log.SSISerrors table */
If Not Exists(Select s.name + '.' + t.name
              From sys.tables t
              Join sys.schemas s
                On s.schema_id = t.schema_id
              Where s.name = 'log'
                And t.name = 'SSISerrors')
begin
  print 'Creating log.SSISerrors table'
  Create Table [log].SSISerrors
  (
    ID int identity(1,1)
      Constraint PK_SSISerrors Primary Key Clustered
    ,AppInstanceID int Not Null
  Constraint FK_logSSISerrors_logSSISAppInstance_AppInstanceID
  Foreign Key References [log].SSISAppInstance(AppInstanceID)
    ,PkgInstanceID int Not Null
  Constraint FK_logSSISerrors_logPkgInstance_PkgInstanceID
  Foreign Key References [log].SSISPkgInstance(PkgInstanceID)
    ,ErrorDateTime datetime Not Null
      Constraint DF_logSSISerrors_ErrorDateTime
      Default(GetDate())
    ,ErrorDescription varchar(max) Null
    ,SourceName varchar(255) Null
  )
  print 'Log.SSISerrors created'
end
Else
  print 'Log.SSISerrors table already exists.'
print ''

/* log.LogError stored procedure */
If Exists(Select s.name + '.' + p.name
          From sys.procedures p
          Join sys.schemas s
            On s.schema_id = p.schema_id
          Where s.name = 'log'
            And p.name = 'LogError')
begin
  print 'Dropping log.LogError stored procedure'
  Drop Procedure [log].LogError
  print 'Log.LogError stored procedure dropped'
end
print 'Creating log.LogError stored procedure'
go

```



```

Create Procedure [log].LogError
    @AppInstanceID int
    ,@PkgInstanceID int
    ,@SourceName varchar(255)
    ,@ErrorDescription varchar(max)
As

insert into log.SSISerrors
(AppInstanceID, PkgInstanceID, SourceName, ErrorDescription)
Values
    (@AppInstanceID
    ,@PkgInstanceID
    ,@SourceName
    ,@ErrorDescription)
go
print 'Log.LogError stored procedure created.'
print ''

```

Each row in the `log.SSISerrors` table contains an `AppInstanceID` and `PkgInstanceID` for identification purposes. Why both? It is designed to capture and preserve errors that originate in both the parent and child packages. An error in the `Parent.dtsx` package will have a `PkgInstanceID` of 0. The remaining columns capture metadata about the error proper: the date and time the error occurred (`ErrorDateTime`), the error message (`ErrorDescription`), and the SSIS task from which the error originated (`SourceName`).

■ **Caution** Adding a row to the `log.SSISerrors` table with a `PkgInstanceID` of 0 will actually raise a foreign key constraint violation at this time, but I will address this matter later in the appendix.

It is important to note that error events are “raised” by SSIS tasks. When an error event is instantiated, its fields are populated with information such as the error description and source name (the name of the task raising the error). These data do not change as the event navigates, or bubbles, inside the SSIS package execution stack. When the event arrives at the `Parent.dtsx` package in the framework, it will contain the name of the task that originated the error (`SourceName`) and the description of the error from that task (`ErrorDescription`).

When the error bubbles to the `Parent.dtsx` package, you will call the `log.LogError` stored procedure to populate the `log.SSISerrors` table. In SSDT, return to the `Parent.dtsx` package’s `OnError` event handler that we configured earlier. Add an `Execute SQL` task and rename it **Log Error**. Open the editor and configure the `ConnectionType` and `Connection` properties to connect to the `SSISConfig` database via ADO.Net. Set the `SQLStatement` property to `log.LogError` and the `IsQueryStoredProcedure` property to `True`. Navigate to the `Parameter Mapping` page and add the following parameters:

- Variable Name: `User::AppInstanceID`
- Direction: Input
- Data Type: `Int32`
- Parameter Name: `AppInstanceID`
- Variable Name: `User::PkgInstanceID`
- Direction: Input
- Data Type: `Int32`

- Parameter Name: PkgInstanceID
- Variable Name: System::SourceName
- Direction: Input
- Data Type: String
- Parameter Name: SourceName
- Variable Name: System::ErrorDescription
- Direction: Input
- Data Type: String
- Parameter Name: ErrorDescription

We created the AppInstanceID and PkgInstanceID SSIS variables earlier in this appendix. You are now using the two variables from the System namespace—SourceName and ErrorDescription—which are two of the fields populated when an error event is first raised by the originating task.

Once these parameters are mapped, close the Execute SQL Task Editor and connect a precedence constraint from the Log Error Execute SQL task to the Log Application Failure Execute SQL task, as shown in Figure A-16.

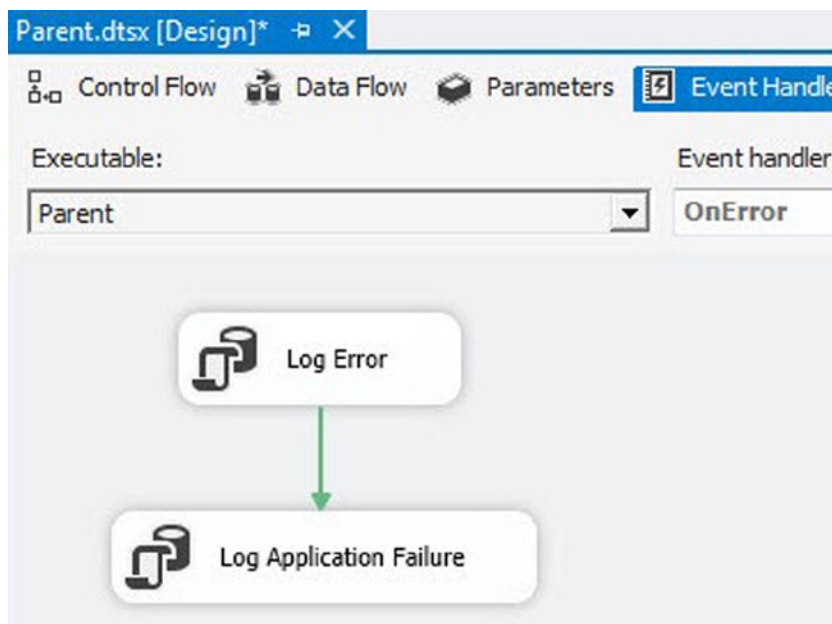


Figure A-16. Adding the Log Error Execute SQL task to the Parent Package OnError event handler

Test the new error logging functionality by running Parent.dtsx in the SSDT debugger. When prompted from the ErrorTest.dtsx package, click the No button to generate an error. In SSMS, execute the following query to examine the error metadata:

```
Select * From log.SSISErrors
```

The results should appear similar to those shown in Figure A-17.

ID	AppInstanceID	PkgInstanceID	ErrorDateTime	ErrorDescription	SourceName
1	9	15	2014-06-29 18:59:42.017	The script returned a failure result.	Succeed or Fail?

Figure A-17. Error metadata in the log.SSISerrors table

As you can see from the preceding image (and hopefully your own code if you are following along at home), error logging can make trouble shooting SSIS issues much simpler.

Event logging is very similar to error logging in SSIS. Part of the reason is that SSIS reuses the object model for the OnError event handler in the OnInformation event handler.

Let's begin by adding another table and stored procedure to the SSISConfig database. The T-SQL script in Listing A-17 accomplishes this task.

Listing A-17. Building the Event Logging Table and Stored Procedure

```

/* log.SSISEvents table */
If Not Exists(Select s.name + '.' + t.name
              From sys.tables t
              Join sys.schemas s
                On s.schema_id = t.schema_id
              Where s.name = 'log'
                And t.name = 'SSISEvents')
begin
    print 'Creating log.SSISEvents table'
    Create Table [log].SSISEvents
    (
        ID int identity(1,1)
        Constraint PK_SSISEvents Primary Key Clustered
    ,AppInstanceID int Not Null
        Constraint FK_logSSISEvents_logSSISAppInstance_AppInstanceID
        Foreign Key References [log].SSISAppInstance(AppInstanceID)
    ,PkgInstanceID int Not Null
        Constraint FK_logSSISEvents_logPkgInstance_PkgInstanceID
        Foreign Key References [log].SSISPkgInstance(PkgInstanceID)
    ,EventDateTime datetime Not Null
        Constraint DF_logSSISEvents_ErrorDateTime
        Default(GetDate())
    ,EventDescription varchar(max) Null
    ,SourceName varchar(255) Null
    )
    print 'Log.SSISEvents created'
end
Else
    print 'Log.SSISEvents table already exists.'
print ''

```

```

/* log.LogEvent stored procedure */
If Exists(Select s.name + '.' + p.name
          From sys.procedures p
          Join sys.schemas s
            On s.schema_id = p.schema_id
          Where s.name = 'log'
            And p.name = 'LogEvent')
begin
  print 'Dropping log.LogEvent stored procedure'
  Drop Procedure [log].LogEvent
  print 'Log.LogEvent stored procedure dropped'
end
print 'Creating log.LogEvent stored procedure'
go

Create Procedure [log].LogEvent
  @AppInstanceID int
  ,@PkgInstanceID int
  ,@SourceName varchar(255)
  ,@EventDescription varchar(max)
As

  insert into [log].SSISEvents
  (AppInstanceID, PkgInstanceID, SourceName, EventDescription)
  Values
  (@AppInstanceID
  ,@PkgInstanceID
  ,@SourceName
  ,@EventDescription)
go
print 'Log.LogEvent stored procedure created.'
print ''

```

With the exception of the column names, the log.SSISEvents table is precisely the same design as the log.SSISErrors table. Return to SSDT and copy the Log Error Execute SQL task from the Parent.dtsx OnError event handler. Change the Event Handler drop-down from OnError to OnInformation and create the OnInformation event handler by clicking the link. Next, paste the contents of the clipboard onto the OnInformation event handler surface. Open the editor and change the name of the task to **Log Event**. Edit the SQLStatement property to read **log.LogEvent**. On the Parameter Mapping page, change the ErrorDescription parameter name from ErrorDescription to **EventDescription**. Close the Execute SQL Task Editor and you are done.

But what about all that “Error” stuff in the parameter mapping? The OnInformation event handler message is conveyed via an SSIS variable named System::ErrorDescription. *That is not a typo.* You might expect it to be InformationDescription, but it’s not, which makes less work for me (and you).

If you execute Parent.dtsx now to test the new event logging functionality, then you won’t see any events logged. Bummer. How do you get events from SSIS? Several tasks provide information via OnInformation events. The Data Flow task, for example, provides lots of helpful metadata about rows read from sources and written to destinations, and lookup cache sizes, rows, and time to populate. You can also inject OnInformation events into the execution stream using a Script task.

I like to include Script tasks that summarize the information I have about SSIS applications and packages in SSIS framework parent packages. Let’s add those now.

Drag a Script task onto the Parent .dtsx package's control flow and rename it **Log Application Variables**. Open the editor and change the ScriptLanguage to Microsoft Visual Basic 2012. Add the following variables to the ReadOnlyVariables property:

- System::TaskName
- System::PackageName
- User::AppInstanceID
- User::ApplicationName

Edit the script and place the code shown in Listing A-18 in Sub Main().

Listing A-18. Raising an Information Event from a Script Task

```
Public Sub Main()

    Dim sPackageName As String = Dts.Variables("PackageName").Value.ToString
    Dim sTaskName As String = Dts.Variables("TaskName").Value.ToString
    Dim sSubComponent As String = sPackageName & "." & sTaskName
    Dim sApplicationName As String = Dts.Variables("ApplicationName").Value.ToString
    Dim iAppInstanceID As Integer = _
        Convert.ToInt32(Dts.Variables("AppInstanceID").Value)

    Dim sMsg As String = "ApplicationName: " & sApplicationName & vbCrLf & _
        "AppInstanceID: " & iAppInstanceID.ToString
    Dts.Events.FireInformation(1001, sSubComponent, sMsg, "", 0, True)

    Dts.TaskResult = ScriptResults.Success
End Sub
```

The purpose of the script is the `Dts.Events.FireInformation` call near the end. The first argument for this function is the `InformationCode`. Depending on the nature and purpose of the SSIS framework, I may or may not enter a value (other than 0) here. The `SubComponent` argument is next and I usually construct a string identifying the names of the package and task. The description argument follows, and this contains the message I want to inject into the log. `SSISEvents` table. The next two arguments are help-related—I usually blank and zero them, respectively. The last argument is `FireAgain`, and I am uncertain if it does anything (anymore); I always set it to `True`.

Close the script editor and the Script Task Editor. Connect a precedence constraint from the Log Start of Application Execute SQL task to the Log Application Variables Script task and another precedence constraint from the Log Application Variables Script task to the Get Package Metadata Execute SQL task.

Drag another Script task into the Foreach Child Package Foreach Loop container and rename it **Log Package Variables**. Open the editor and change the ScriptLanguage to Microsoft Visual Basic 2012. Add the following variables to the ReadOnlyVariables property:

- System::TaskName
- System::PackageName
- User::PkgInstanceID
- User::ChildPackagePath
- User::AppPackageID

Edit the script and place the code shown in Listing A-19 in Sub Main().

Listing A-19. Raising an Information Event from a Script Task

```
Public Sub Main()

    Dim sPackageName As String = Dts.Variables("PackageName").Value.ToString
    Dim sTaskName As String = Dts.Variables("TaskName").Value.ToString
    Dim sSubComponent As String = sPackageName & "." & sTaskName
    Dim sChildPackagePath As String = Dts.Variables("ChildPackagePath").Value.ToString
    Dim iAppPackageID As Integer = Convert.ToInt32(Dts.Variables("AppPackageID").Value)
    Dim iPkgInstanceID As Integer = _ Convert.ToInt32(Dts.Variables("PkgInstanceID").Value)

    Dim sMsg As String = "ChildPackagePath: " & sChildPackagePath & vbCrLf & _
        "AppPackageID: " & iAppPackageID.ToString & vbCrLf & _
        "PkgInstanceID: " & iPkgInstanceID.ToString
    Dts.Events.FireInformation(1001, sSubComponent, sMsg, "", 0, True)

    Dts.TaskResult = ScriptResults.Success
End Sub
```

Close the script editor and the Script Task Editor. Connect a precedence constraint from the Log Start of Package Execute SQL task to the Log Start of Package Execute SQL task.

If you execute Parent.dtsx now, you may get a foreign key constraint error when you try to log the application variables. Why? PkgInstanceID is set to a default value, 0, and there is no 0 row in the log.SSISPkgInstance table. Let's remedy that now with the following script shown in Listing A-20.

Listing A-20. Adding 0 ID Rows to Selected Tables in the SSISConfig Database

```
/* Add "0" rows */
If Not Exists(Select ApplicationID
              From cfg.Applications
              Where ApplicationID = 0)
begin
    print 'Adding 0 row for cfg.Applications'
    Set Identity_Insert cfg.Applications ON
    Insert Into cfg.Applications
    (ApplicationID
    ,ApplicationName)
    Values
    (0
    ,'SSIS Framework')
    Set Identity_Insert cfg.Applications OFF
    print '0 row for cfg.Applications added'
end
Else
    print '0 row already exists for cfg.Applications'
print ''

If Not Exists(Select PackageID
              From cfg.Packages
              Where PackageID = 0)
begin
    print 'Adding 0 row for cfg.Packages'
```

```

Set Identity_Insert cfg.Packages ON
Insert Into cfg.Packages
(PackageID
,PackageFolder
,PackageName)
Values
(0
,'\'
,'parent.dtsx')
Set Identity_Insert cfg.Packages OFF
print '0 row for cfg.Packages added'
end
Else
print '0 row already exists for cfg.Packages'
print ''

If Not Exists(Select AppPackageID
              From cfg.AppPackages
              Where AppPackageID = 0)
begin
print 'Adding 0 row for cfg.Packages'
Set Identity_Insert cfg.AppPackages ON
Insert Into cfg.AppPackages
(AppPackageID
,ApplicationID
,PackageID
,ExecutionOrder)
Values
(0
,0
,0
,10)
Set Identity_Insert cfg.AppPackages OFF
print '0 row for cfg.AppPackages added'
end
Else
print '0 row already exists for cfg.AppPackages'
print ''

If Not Exists(Select AppInstanceID
              From [log].SSISAppInstance
              Where AppInstanceID = 0)
begin
print 'Adding 0 row for cfg.Packages'
Set Identity_Insert [log].SSISAppInstance ON
Insert Into [log].SSISAppInstance
(AppInstanceID
,ApplicationID
,StartDateTime
,EndDateTime
,[Status])

```

```

Values
(0
,0
,'1/1/1900'
,'1/1/1900'
,'Unknown')
Set Identity_Insert [log].SSISAppInstance OFF
print '0 row for log.SSISAppInstance added'
end
Else
print '0 row already exists for log.SSISAppInstance'
print ''

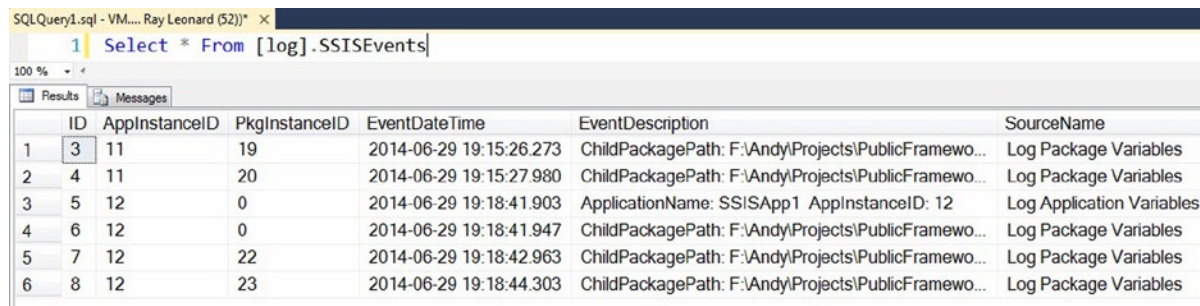
If Not Exists(Select PkgInstanceID
               From [log].SSISPkgInstance
               Where PkgInstanceID = 0)
begin
print 'Adding 0 row for cfg.Packages'
Set Identity_Insert [log].SSISPkgInstance ON
Insert Into [log].SSISPkgInstance
(PkgInstanceID
,AppInstanceID
,AppPackageID
,StartDateTime
,EndDateTime
,[Status])
Values
(0
,0
,0
,'1/1/1900'
,'1/1/1900'
,'Unknown')
Set Identity_Insert [log].SSISPkgInstance OFF print '0 row for log.SSISPkgInstance added'
end
Else
print '0 row already exists for log.SSISPkgInstance'
print ''

```

Now that these event-generating Script tasks are in place, test-execute the `Parent.dtsx` package and then observe the `log.LogEvents` table by executing the following T-SQL in SSMS:

```
Select * From [log].SSISEvents
```

My results appear as shown in Figure A-18.



ID	AppInstanceID	PkgInstanceID	EventDateTime	EventDescription	SourceName
1	3	11	2014-06-29 19:15:26.273	ChildPackagePath: F:\Andy\Projects\PublicFrameworko...	Log Package Variables
2	4	11	2014-06-29 19:15:27.980	ChildPackagePath: F:\Andy\Projects\PublicFrameworko...	Log Package Variables
3	5	12	2014-06-29 19:18:41.903	ApplicationName: SSISApp1 AppInstanceID: 12	Log Application Variables
4	6	12	2014-06-29 19:18:41.947	ChildPackagePath: F:\Andy\Projects\PublicFrameworko...	Log Package Variables
5	7	12	2014-06-29 19:18:42.963	ChildPackagePath: F:\Andy\Projects\PublicFrameworko...	Log Package Variables
6	8	12	2014-06-29 19:18:44.303	ChildPackagePath: F:\Andy\Projects\PublicFrameworko...	Log Package Variables

Figure A-18. SSIS framework events!

Viewing the log.SSISEvents table in SSMS is disappointing. The data is accurate and SSMS is doing its job, but the user experience could be better for this type of data. Fortunately, SQL Server 2014 ships with SQL Server Reporting Services, which provides a better user experience! Let's look at building reports to display this data.

Reporting Execution Metrics

SQL Server Reporting Services (SSRS) allows us to create reports that display SSIS framework metadata and metrics in a more user-friendly format. We can add visualizations to the reports that will assist in identifying the status of SSIS applications and SSIS packages.

Open a new instance of SSDT-BI and create a new Report Server project named **PublicFrameworkReports_PackageDeployment_2014**. In Solution Explorer, right-click Shared Data Source and click Add New Data Source. When the Shared Data Source Properties window displays, set the Name property to **SSISConfig** and click the Edit button to configure the connection to your instance of the SSISConfig database. When I configure the shared data source, it appears as shown in Figure A-19.

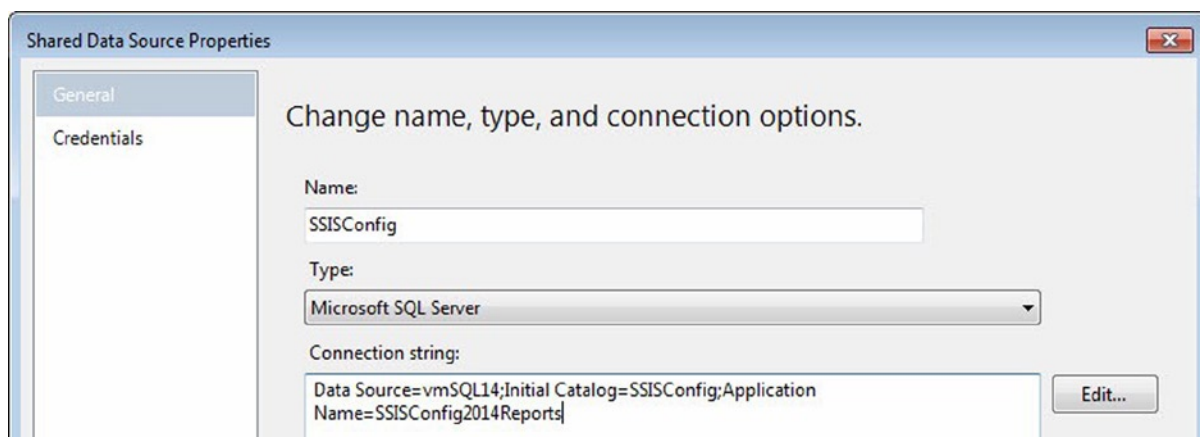


Figure A-19. Configuring the SSISConfig shared data source

You're now ready to build reports! Let's begin by creating a report to display application instance data.

Before we jump into report development, we will create supporting objects in the SSISConfig database. Listing A-21 contains the T-SQL script required to build the rpt schema and the rpt.ReturnAppInstanceHeader stored procedure.

Listing A-21. Creating the rpt Schema and rpt.ReturnAppInstanceHeader Stored Procedure

```

/* rpt schema */
If Not Exists(Select name
              From sys.schemas
              Where name = 'rpt')
begin
    print 'Creating rpt schema'
    declare @sql varchar(100) = 'Create Schema rpt'
    exec(@sql)
    print 'Rpt schema created'
end
Else
    print 'Rpt schema already exists.'
print ''

/* rpt.ReturnAppInstanceHeader stored procedure */
If Exists(Select s.name + '.' + p.name
          From sys.procedures p
          Join sys.schemas s
            On s.schema_id = p.schema_id
          Where s.name = 'rpt'
          And p.name = 'ReturnAppInstanceHeader')
begin
    print 'Dropping rpt.ReturnAppInstanceHeader stored procedure'
    Drop Procedure rpt.ReturnAppInstanceHeader
    print 'Rpt.ReturnAppInstanceHeader stored procedure dropped'
end
print 'Creating rpt.ReturnAppInstanceHeader stored procedure'
go

Create Procedure rpt.ReturnAppInstanceHeader
    @ApplicationName varchar(255) = NULL
As

    Select a.ApplicationID
           ,ap.AppInstanceID
           ,a.ApplicationName
           ,ap.StartDateTime
           ,DateDiff(ss,ap.StartDateTime,Coalesce(ap.EndDateTime,GetDate())) As RunSeconds
           ,ap.Status
    From log.SSISAppInstance ap
    Join cfg.Applications a
      On ap.ApplicationID = a.ApplicationID
    Where a.ApplicationName = Coalesce(@ApplicationName,a.ApplicationName)
    Order by AppInstanceID desc

go

```

```
print 'Rpt.ReturnAppInstanceHeader stored procedure created.'
print ''
```

Return to SSDT, right-click the Reports virtual folder in Solution Explorer, and click Add New Report. If the welcome screen displays, then click the Next button. On the Select the Data Source screen, select the shared data source named SSISConfig and click the Next button. The Design the Query window displays next; add `rpt.ReturnAppInstanceHeader` to the Query String textbox and click the Next button. Select Tabular on the Select the Report Type page and click the Next button. When the Design the Table page displays, multiselect all the columns listed in the Available Fields listbox and click the Details button. The Report Wizard will appear as shown in Figure A-20.

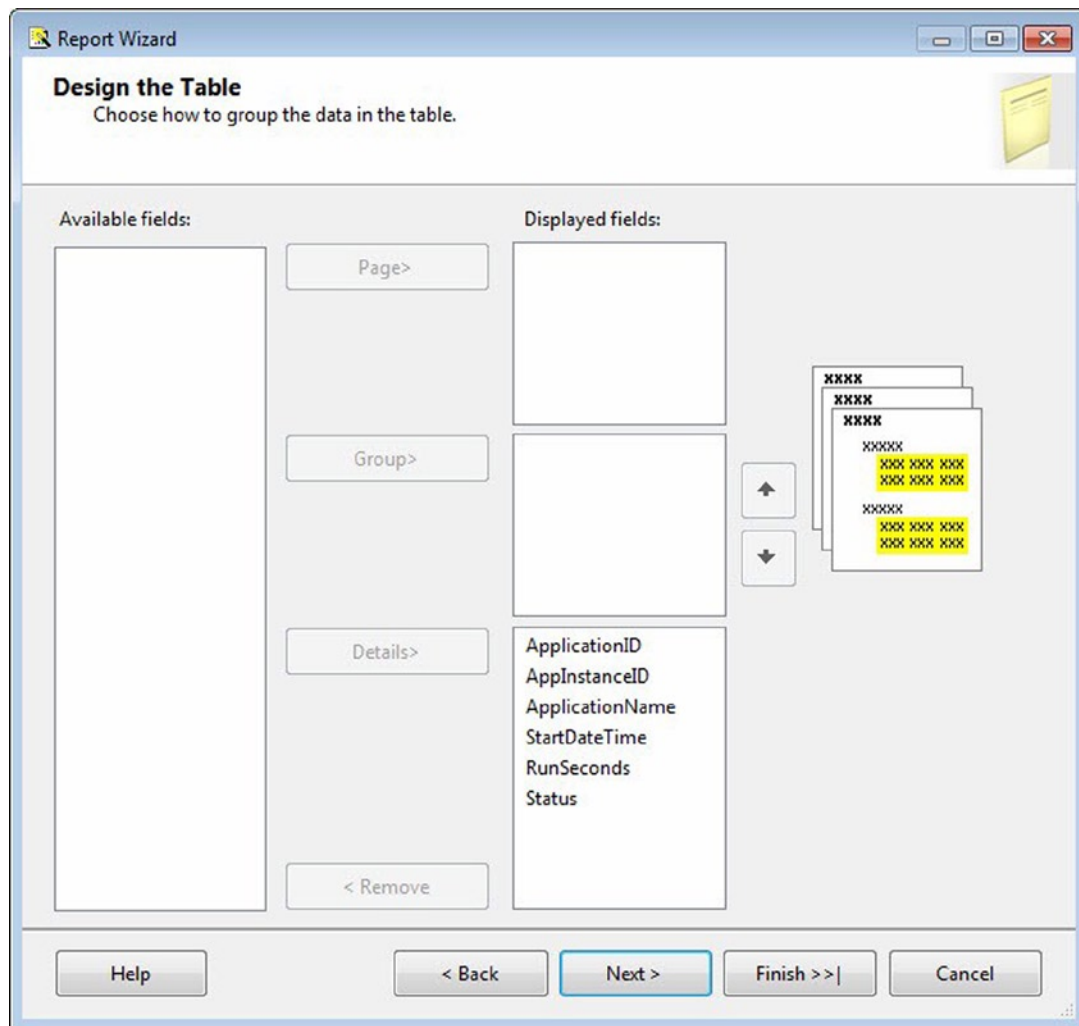


Figure A-20. Selecting all available fields as details

Click the Next button. Select a theme on the Choose the Table Style page and click the Next button. On the Completing the Wizard page, enter **Application Instance** in the Report Name property textbox and click the Finish button.

The SSRS Report Wizard will generate the report, but it doesn't manage stored procedures effectively. You need to change this so you get the maximum performance out of the reports. Click View ► Report Data to display the Report Data sidebar. Expand the Datasets virtual folder. Right-click DataSet1 and click Dataset Properties. When the Dataset Properties window displays, rename the dataset rpt_ReturnAppInstanceHeader (the Dataset Name property does not like periods). Copy rpt.ReturnAppInstanceHeader out of the Query property and click the Stored Procedure option under Query Type. Paste rpt.ReturnAppInstanceHeader into the Select or Enter Stored Procedure Name drop-down. The Dataset Properties window should appear similar to what is shown in Figure A-21.

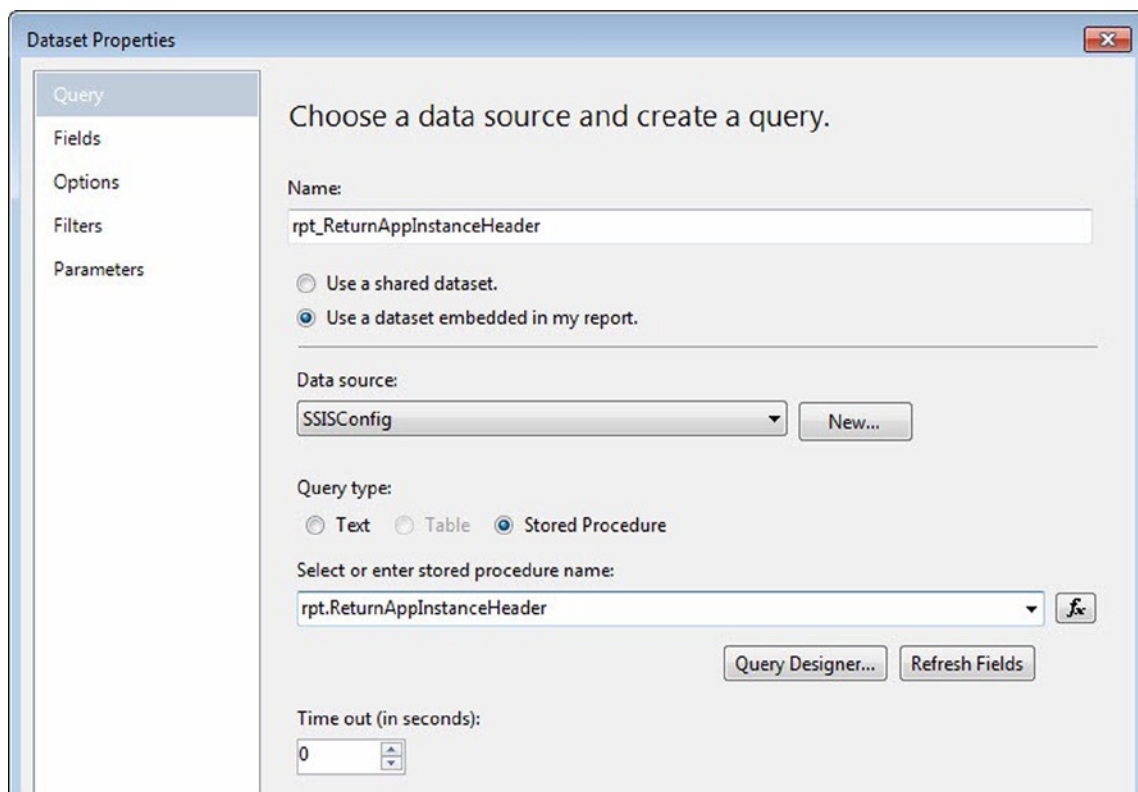


Figure A-21. Configuring the dataset to use the rpt.ReturnAppInstanceHeader stored procedure

Click the OK button to close the Dataset Properties window. If you click the Preview tab, the report will prompt you for an application name as shown in Figure A-22.

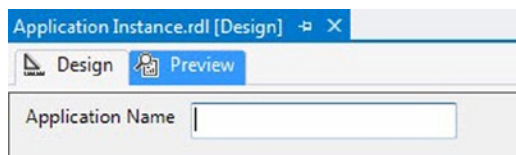
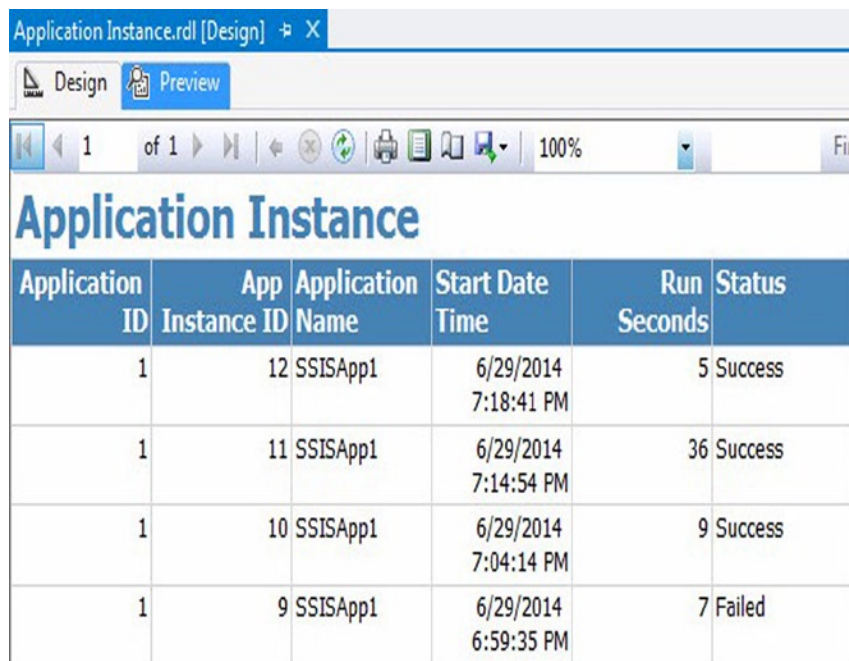


Figure A-22. Prompting for application name

Type **SSISApp1** in the textbox and click the View Report button in the upper-right corner. We don't want the user to supply an SSIS application each time they use the report, so let's configure the report parameter named @ApplicationName. Click the Design tab and return to the Report Data sidebar. Expand the Parameters virtual folder. Double-click @ApplicationName to open the Report Parameter Properties window. On the General page, check the Allow Null Value checkbox and change the Select Parameter Visibility option to Hidden. On the Default Values page, select the Specify Values option and click the Add button. A (Null) row will be added to the Value grid, which is what we want. Click the OK button to close the Report Parameter Properties window.

Test the changes by clicking the Preview tab. The report should display all application instance rows stored in the database, as shown in Figure A-23.



Application ID	App Instance ID	Application Name	Start Date Time	Run Seconds	Status
1	12	SSISApp1	6/29/2014 7:18:41 PM	5	Success
1	11	SSISApp1	6/29/2014 7:14:54 PM	36	Success
1	10	SSISApp1	6/29/2014 7:04:14 PM	9	Success
1	9	SSISApp1	6/29/2014 6:59:35 PM	7	Failed

Figure A-23. Displaying the application instance data

We do not want to see the 0 rows displayed in these reports. Modify the rpt.ReturnAppInstanceHeader stored procedure to eliminate these records from the returned results by executing the T-SQL shown in Listing A-22.

Listing A-22. Updating the rpt.ReturnAppInstanceHeader Stored Procedure

```
/* rpt.ReturnAppInstanceHeader stored procedure */
If Exists(Select s.name + '.' + p.name
          From sys.procedures p
          Join sys.schemas s
            On s.schema_id = p.schema_id
          Where s.name = 'rpt'
          And p.name = 'ReturnAppInstanceHeader')
```

```

begin
    print 'Dropping rpt.ReturnAppInstanceHeader stored procedure'
    Drop Procedure rpt.ReturnAppInstanceHeader
    print 'Rpt.ReturnAppInstanceHeader stored procedure dropped'
end
print 'Creating rpt.ReturnAppInstanceHeader stored procedure'
go

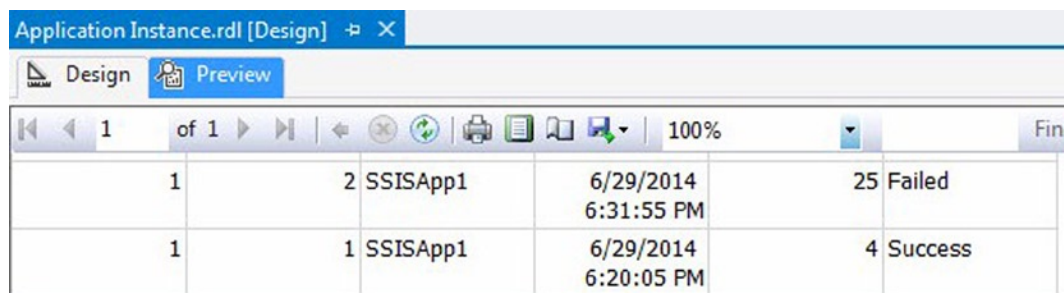
Create Procedure rpt.ReturnAppInstanceHeader
    @ApplicationName varchar(255) = NULL
As

    Select a.ApplicationID
           ,ap.AppInstanceID
           ,a.ApplicationName
           ,ap.StartDateTime
           ,DateDiff(ss,ap.StartDateTime,Coalesce(ap.EndDateTime,GetDate())) As RunSeconds
           ,ap.Status
    From log.SSISAppInstance ap
    Join cfg.Applications a
        On ap.ApplicationID = a.ApplicationID
    Where a.ApplicationName = Coalesce(@ApplicationName,a.ApplicationName)
    And a.ApplicationID > 0
    Order by AppInstanceID desc

go
print 'Rpt.ReturnAppInstanceHeader stored procedure created.'
print ''

```

Refresh the Application Instance report preview and it now appears as shown in Figure A-24.



ApplicationID	AppInstanceID	ApplicationName	StartDateTime	RunSeconds	Status
1	2	SSISApp1	6/29/2014 6:31:55 PM	25	Failed
1	1	SSISApp1	6/29/2014 6:20:05 PM	4	Success

Figure A-24. Refreshed Application Instance report, sans the 0 row

Color helps identify the state better than most visual cues. To add background color to the data rows, return to the Design tab and select the row that displays data values (the bottom row) in the table. Press the F4 key to display Properties and click on the BackgroundColor property. In the BackgroundColor property's value drop-down, select Expression. When the Expression window opens, change the text in the Set Expression for: BackgroundColor textbox from No Color (the default) to the following expression:

```
=Switch(Fields!Status.Value="Success", "LightGreen",  
    , Fields!Status.Value="Failed", "LightCoral",  
    , Fields!Status.Value="Running", "Yellow")
```

When you clean up the report by removing ID columns (which mean little to the user), resetting font sizes, changing text alignment, and adjusting column widths, you can alter the report so it appears as shown in Figure A-25.

Application Instance

Application Name	Start Time	Run Time	Status
SSISApp1	6/29/2014 7:38:36 PM	2	Running
SSISApp1	6/29/2014 7:18:41 PM	5	Success
SSISApp1	6/29/2014 7:14:54 PM	36	Success
SSISApp1	6/29/2014 7:04:14 PM	9	Success
SSISApp1	6/29/2014 6:59:35 PM	7	Failed

Figure A-25. Application Instance—in color!

I call this Operational Intelligence. An enterprise operations person can look at this report and glean lots of information about the current state of enterprise data integration processes.

The Package Instance report is remarkably similar. Let's begin by adding the stored procedure to the database, as shown in Listing A-23.

Listing A-23. Adding the rpt.ReturnPkgInstanceHeader Stored Procedure

```
/* rpt.ReturnPkgInstanceHeader stored procedure */  
If Exists(Select s.name + '.' + p.name  
    From sys.procedures p  
    Join sys.schemas s  
    On s.schema_id = p.schema_id  
    Where s.name = 'rpt'  
    And p.name = 'ReturnPkgInstanceHeader')  
begin  
    print 'Dropping rpt.ReturnPkgInstanceHeader stored procedure'  
    Drop Procedure rpt.ReturnPkgInstanceHeader  
    print 'Rpt.ReturnPkgInstanceHeader stored procedure dropped'  
end  
print 'Creating rpt.ReturnPkgInstanceHeader stored procedure'  
go
```



```

Create Procedure rpt.ReturnPkgInstanceHeader
    @AppInstanceID int
As

    SELECT a.ApplicationName
           ,p.PackageFolder + p.PackageName As PackagePath
           ,cp.StartDateTime
           ,DateDiff(ss,
cp.StartDateTime,
Coalesce(cp.EndDateTime,GetDate())) As RunSeconds
           ,cp.Status
           ,ai.AppInstanceID
           ,cp.PkgInstanceID
           ,p.PackageID
           ,p.PackageName
    FROM log.SSISPackageInstance cp
    Join cfg.AppPackages ap
        on ap.AppPackageID = cp.AppPackageID
    Join cfg.Packages p
        on p.PackageID = ap.PackageID
    Join log.SSISAppInstance ai
        on ai.AppInstanceID = cp.AppInstanceID
    Join cfg.Applications a
        on a.ApplicationID = ap.ApplicationID
    WHERE ai.AppInstanceID = Coalesce(@AppInstanceID,ai.AppInstanceID)
        And a.ApplicationID > 0
    Order By cp.PkgInstanceID desc

go
print 'Rpt.ReturnPkgInstanceHeader stored procedure created.'
print ''

```

In SSDT, add a new report named **Package Instance** just like you added the Application Instance report. Make sure you use the `rpt.ReturnPkgInstanceHeader` stored procedure. To get the Report Wizard to recognize a query that expects parameters, you need to add default parameter values on the Design the Query page. My Query String textbox reads as follows:

```
exec rpt.ReturnPkgInstanceHeader NULL
```

This allows the query builder to locate the columns list returned from the stored procedure (which is what the Report Wizard needs to continue). Once the report is built, remember to first update the dataset, then the report parameter, as you did for the Application Instance report. One cool thing about this particular design is that we can reuse the expression for `BackgroundColor` on the data rows. Once complete, the Package Instance report appears, as shown in Figure A-26.

Package Instance

Application Name	Package Name	Start Time	Run Time	Status
SSISApp1	Child1.dtsx	6/29/2014 8:06:36 PM	3	Running
SSISApp1	ErrorTest.dtsx	6/29/2014 7:45:10 PM	3	Failed
SSISApp1	Child2.dtsx	6/29/2014 7:45:08 PM	2	Success
SSISApp1	Child1.dtsx	6/29/2014 7:44:05 PM	63	Success
SSISApp1	ErrorTest.dtsx	6/29/2014 7:38:47 PM	1	Success

Figure A-26. The Package Instance report

Package instances are “children” of application instances. To reflect that relationship, return to the Application Instance report and add a column to the table to contain Packages links. Enter **Packages** in the column header and as text in the data cell. Right-click the data cell and click Text Box Properties. On the Font page, change the font color to Blue and set the Effects property to Underline. On the Action page, select the Go to Report option for the Enable as an Action property and set the Specify a Report property to Package Instance. In the Use These Parameters to Run the Report grid, click the Add button and map the AppInstanceID parameter to the [AppInstanceID] value. Click the OK button to close the Text Box Properties Editor.

Click the Preview tab to display the Application Instance report. Select one of the Packages” links to navigate to the Package Instance report that will contain only the package instances related to that particular application instance. The Package Instance report should appear similar to the Package Instance report displayed in Figure A-27.

Package Instance

Application Name	Package Name	Start Time	Run Time	Status
SSISApp1	ErrorTest.dtsx	6/29/2014 8:06:46 PM	3	Success
SSISApp1	Child2.dtsx	6/29/2014 8:06:43 PM	3	Success
SSISApp1	Child1.dtsx	6/29/2014 8:06:36 PM	7	Success

Figure A-27. Package instances for a single application instance

Building the reports in this fashion makes sense. The Application Instance report becomes a “gateway” for the Package Instance report—a “dashboard,” if you will. More in a bit ...

Let’s turn our attention to the error log data. To retrieve it, use the T-SQL script shown in Listing A-24.

Listing A-24. Building the rpt.ReturnErrors Stored Procedure

```
/* rpt.ReturnErrors stored procedure */
If Exists(Select s.name + '.' + p.name
  From sys.procedures p
  Join sys.schemas s
    On s.schema_id = p.schema_id
  Where s.name = 'rpt'
    And p.name = 'ReturnErrors')
```

```

begin
    print 'Dropping rpt.ReturnErrors stored procedure'
    Drop Procedure rpt.ReturnErrors
    print 'Rpt.ReturnErrors stored procedure dropped'
end
print 'Creating rpt.ReturnErrors stored procedure'
go

Create Procedure rpt.ReturnErrors
    @AppInstanceID int
    ,@PkgInstanceID int = NULL
As

Select
    a.ApplicationName
    ,p.PackageName
    ,er.SourceName
    ,er.ErrorDateTime
    ,er.ErrorDescription
From log.SSISErrors er
Join log.SSISAppInstance ai
    On ai.AppInstanceID = er.AppInstanceID
Join cfg.Applications a
    On a.ApplicationID = ai.ApplicationID
Join log.SSISPkgInstance cp
    On cp.PkgInstanceID = er.PkgInstanceID
    And cp.AppInstanceID = er.AppInstanceID
Join cfg.AppPackages ap
    On ap.AppPackageID = cp.AppPackageID
Join cfg.Packages p
    On p.PackageID = ap.PackageID
Where er.AppInstanceID = Coalesce(@AppInstanceID, er.AppInstanceID)
    And er.PkgInstanceID = Coalesce(@PkgInstanceID, er.PkgInstanceID)
Order By ErrorDateTime Desc
go
print 'Rpt.ReturnErrors stored procedure created.'
print ''

```

The T-SQL in Listing A-24 constructs the `rpt.ReturnErrors` stored procedure, which will supply data to a new report. Let's build that report now in SSDT.

Add a new report named **Errors** to the `SSISConfig2012Reports` solution. Use the `rpt.ReturnErrors` stored procedure as the source. Remember to update the dataset and both report parameters: `AppInstanceID` and `PkgInstanceID`.

On the table's data row, edit the `BackgroundColor` property, adding the following expression:

```
=Iif(RowNumber(Nothing) Mod 2 = 0,"White","WhiteSmoke")
```

We are not coloring the background of each cell here to reflect status; the report would be filled with `LightCoral` if we did so. But we do need to break up these rows visually, so let's use subtle shading to help keep the eyes moving across the row at 2:15 AM some dark and dreary morning.

Open the Application Instance report. Right-click on the Status data field and click Text Box Properties. Navigate to the Font page and click the f(x) button beside the Color property drop-down. In the Set Expression for: Color textbox, enter the following expression:

```
=Iif(Fields!Status.Value="Failed", "Blue", "Black")
```

If the status is "Failed", then this expression will change the color of the Status text to Blue. Click the f(x) button beside the Effects property drop-down. In the Set Expression for: TextDecoration textbox, add the following expression:

```
=Iif(Fields!Status.Value="Failed", "Underline", "Default")
```

This expression will decorate a "Failed" status with an underline. This and the previous property combine to make "Failed" status appear as a hyperlink. Where does the hyperlink take us? Let's configure that property now. Navigate to the Action page and select the Go to Report option for the Enable as an Action property. Click the f(x) button beside the Specify a Report drop-down and add the following expression to the Set Expression for: ReportName textbox:

```
=Iif(Fields!Status.Value="Failed", "Errors", Nothing)
```

Click the Add button and map the AppInstanceID parameter name to the [AppInstanceID] parameter value. Click the f(x) button in the Omit column of the parameter mapping and add the following expression to the Set Expression for: Omit textbox:

```
=Iif(Fields!Status.Value="Failed", False, True)
```

The two previous property settings configure the Action property of the Status value. If the Status is "Failed", clicking the word *Failed*, which will appear to be a hyperlink, will cause the Errors report to display. When it displays, it will only show those error rows associated with the application instance displayed in that row of data.

Let's test it! When I run the Application Instance report, it now appears as shown in Figure A-28.

Application Instance

Application Name	Start Time	Run Time	Status	Packages
SSISApp1	6/29/2014 8:06:36 PM	13	Success	Packages
SSISApp1	6/29/2014 7:44:05 PM	68	Failed	Packages
SSISApp1	6/29/2014 7:38:36 PM	12	Success	Packages

Figure A-28. The Application Instance report, including Status and Packages decoration

Clicking one of the Failed hyperlinks takes us to the Errors report for that application instance. The report should appear similar to that shown in Figure A-29.

Errors

Application Name	Package Name	Source Name	Error Time	Error Description
SSISApp1	ErrorTest.dtsx	Succeed or Fail?	6/29/2014 7:45:13 PM	The script returned a failure result.

Figure A-29. *Displaying an error*

Quickly isolating the source of an error in an SSIS package is one way to improve overall operational efficiency. These reports, working in tandem, facilitate efficient root cause analysis.

The Events report is very similar to the Errors report. The T-SQL script for creating the `rpt.ReturnEvents` stored procedure is shown in Listing A-25.

Listing A-25. Building the `rpt.ReturnEvents` Stored Procedure

```

/* rpt.ReturnEvents stored procedure */
If Exists(Select s.name + '.' + p.name
          From sys.procedures p
          Join sys.schemas s
            On s.schema_id = p.schema_id
          Where s.name = 'rpt'
            And p.name = 'ReturnEvents')
begin
  print 'Dropping rpt.ReturnEvents stored procedure'
  Drop Procedure rpt.ReturnEvents
  print 'Rpt.ReturnEvents stored procedure dropped'
end
print 'Creating rpt.ReturnEvents stored procedure'
go

Create Procedure rpt.ReturnEvents
  @AppInstanceID int
  ,@PkgInstanceID int = NULL
As

Select
  a.ApplicationName
  ,p.PackageName
  ,ev.SourceName
  ,ev.EventDateTime
  ,ev.EventDescription
From log.SSISEvents ev
Join log.SSISAppInstance ai
  On ai.AppInstanceID = ev.AppInstanceID
Join cfg.Applications a
  On a.ApplicationID = ai.ApplicationID
Join log.SSISPkgInstance cp
  On cp.PkgInstanceID = ev.PkgInstanceID
  And cp.AppInstanceID = ev.AppInstanceID
Join cfg.AppPackages ap
  On ap.AppPackageID = cp.AppPackageID
Join cfg.Packages p
  On p.PackageID = ap.PackageID

```

```

Where ev.AppInstanceID = Coalesce(@AppInstanceID, ev.AppInstanceID)
And ev.PkgInstanceID = Coalesce(@PkgInstanceID, ev.PkgInstanceID)
Order By EventDateTime Desc
go
print 'Rpt.ReturnEvents stored procedure created.'
print ''

```

Add a new report named **Events**, use the `rpt.ReturnEvents` stored procedure, and remember to configure the dataset and report parameters. Add the alternating row shading I demonstrated in the Errors report. The same expression will work in the Events report:

```
=Iif(RowNumber(Nothing) Mod 2 = 0,"White","WhiteSmoke")
```

Return to the Application Instance report and add another column to the data table. Label it **Events** and set the data grid value to Events as well. Open the Text Box Properties window for the Events data field and navigate to the Font page. Change the Color property to Blue and the Effects property to Underline. On the Actions page, change the Enable as an Action property to “Go to Report and the Specify a Report drop-down to Events. Add a parameter mapping and map the AppInstanceID parameter name to the [AppInstanceID] parameter value. Click the OK button to close the Text Box Properties Editor. Let’s test it!

The Application Instance report now appears, as shown in Figure A-30.

Application Instance

Application Name	Start Time	Run Time	Status	Packages	Events
SSISApp1	6/29/2014 8:06:36 PM	13	Success	Packages	Events
SSISApp1	6/29/2014 7:44:05 PM	68	Failed	Packages	Events
SSISApp1	6/29/2014 7:38:36 PM	12	Success	Packages	Events

Figure A-30. The new and improved Application Instance report

Clicking the Events hyperlink takes us to the Events report, which should be similar the report shown in Figure A-31.

Events

Application Name	Package Name	Source Name	Event Date Time	Event Description
SSISApp1	Child2.dtsx	Log Package Variables	6/29/2014 8:06:46 PM	ChildPackagePath: F:\Andy\Projects \PublicFramework_PackageDeployment_2014\SSISConfig2014\ErrorTest.dtsx AppPackageID: 5 PkgInstanceID: 32
SSISApp1	Child1.dtsx	Log Package Variables	6/29/2014 8:06:43 PM	ChildPackagePath: F:\Andy\Projects \PublicFramework_PackageDeployment_2014\SSISConfig2014\Child2.dtsx AppPackageID: 2 PkgInstanceID: 31

Figure A-31. The Events report for an application instance

This latest round of reports and updates to the Application Instance report reinforce its status as the Operational Intelligence Dashboard. Similar changes can be made to the Package Instance report. Let’s add the Failed link functionality and the Events column now.

On the Package Instance report, open the Text Box Properties Editor for the Status data field. As I did for the Status data field in the Application Instance report, navigate to the Font page and click the f(x) button beside the Color property drop-down. In the Set Expression for: Color textbox, enter the following expression:

```
=Iif(Fields!Status.Value="Failed", "Blue", "Black")
```

This expression will change the color of the status text to blue if the status is "Failed". Click the f(x) button beside the Effects property drop-down. In the Set Expression for: TextDecoration textbox, add the following expression:

```
=Iif(Fields!Status.Value="Failed", "Underline", "Default")
```

As with the Application Instance report, this expression will decorate a Failed status with an underline. This and the previous property combine to make the Failed status appear as a hyperlink. Where does the hyperlink take us? Let's configure that property now. Navigate to the Action page and select the Go to Report option for the Enable as an Action property. Click the f(x) button beside the Specify a Report" drop-down and add the following expression to the Set Expression for: ReportName textbox:

```
=Iif(Fields!Status.Value="Failed", "Errors", Nothing)
```

Click the Add button and map the AppInstanceID parameter name to the [AppInstanceID] parameter value. Click the Add button again and map the PkgInstanceID parameter name to the [PkgInstanceID] parameter value. Click the f(x) button in the Omit column of each parameter mapping and add the following expression to each Set Expression for: Omit textbox:

```
=Iif(Fields!Status.Value="Failed", False, True)
```

As with the Application Instance report, the two previous property settings configure the Action property of the Status value. If the status is "Failed", clicking the word Failed, which will appear to be a hyperlink, will cause the Errors report to display. When it displays, it will only show those error rows associated with the application instance displayed in that row of data.

Let's test it! When we run the Package Instance report, it now appears as shown in Figure A-32.

Package Instance

Application Name	Package Name	Start Time	Run Time	Status
SSISApp1	ErrorTest.dtsx	6/29/2014 8:06:46 PM	3	Success
SSISApp1	Child2.dtsx	6/29/2014 8:06:43 PM	3	Success
SSISApp1	Child1.dtsx	6/29/2014 8:06:36 PM	7	Success
SSISApp1	ErrorTest.dtsx	6/29/2014 7:45:10 PM	3	Failed
SSISApp1	Child2.dtsx	6/29/2014 7:45:08 PM	2	Success

Figure A-32. "Failed hyperlinks" for the Package Instance report

Clicking a Failed link takes us to the Errors report for that package instance. Cool. Now let's add the Events column to the Package Instance report. Add a column with the header and data field hard-coded Events. Open the Text Box Properties window for the Events data field and navigate to the Font page. Set the Color property to Blue and the Effects property to Underline. Navigate to the Action page and set the Enable as an Action property to Go to Report. Select the Events report from the Specify a Report drop-down and click the Add button twice to map two parameters. Map the AppInstanceID parameter name to the [AppInstanceID] parameter value and the PkgInstanceID parameter name to the [PkgInstanceID] parameter value. Close the Text Box Properties window and click the Preview tab to test. The Package Instance report should appear as shown in Figure A-33.

Package Instance

Application Name	Package Name	Start Time	Run Time	Status	Events
SSISApp1	ErrorTest.dtsx	6/29/2014 8:06:46 PM	3	Success	Events
SSISApp1	Child2.dtsx	6/29/2014 8:06:43 PM	3	Success	Events
SSISApp1	Child1.dtsx	6/29/2014 8:06:36 PM	7	Success	Events
SSISApp1	ErrorTest.dtsx	6/29/2014 7:45:10 PM	3	Failed	Events
SSISApp1	Child2.dtsx	6/29/2014 7:45:08 PM	2	Success	Events

Figure A-33. The finished Package Instance report

Clicking the Events link will take me to the Events report and display only the events for the package instance on the referenced row.

To wrap it up, you can start at the Application Instance report; it is on the dashboard. Click the Packages link to view all the SSIS child packages that executed as part of the selected SSIS application instance. From there, drill into the Errors report and observe the errors that caused a package to fail, or view all of the events recorded by the OnInformation event handler for the selected package instance on the Events report. You can reach all errors and events for an SSIS application instance from the Application Instance report, as well.

Conclusion

This example isn't an exhaustive example of an SSIS framework, but it demonstrates the utility of patterns-based data integration development using SSIS. The sample framework provides repeatable, metadata-driven SSIS execution without leaving the SSIS and SQL Server database domains. Monitoring is provided by a set of SQL Server Reporting Services reports driven by stored procedures that read metadata automatically captured by the framework's Parent.dtsx SSIS package. Zero lines of code are required in child packages to capture error and event information, and this information is logged centrally in a consistent format, which makes it perfect for reporting.

Index

■ A

AcquireConnection method, [207](#)
Advanced Editor, [259](#)
AndyWeather.com, [275](#)

■ B

Biml2014, [344](#)
Business intelligence (BI), [343](#)
Business Intelligence Development Studio (BIDS), [28](#), [262](#)
Business Intelligence Markup Language (Biml)
 command-line, [375–376](#)
 data integration development, [343](#)
 execution, [374](#)
 file
 BimlScript.biml file, [345](#)
 connect metadata, [346](#)
 error message, [345](#)
 Execute SQL Task, [345](#)
 initial code, [344](#)
 SSIS package, [347](#)
 TestBimlPackage.dtsx, [345](#)
 XML metadata, [344](#)
 history, [343](#)
 incremental load pattern
 add metadata, [349](#)
 databases and tables creation, [347](#)
 data flow task, [350](#)
 debug executing script, [359](#)
 dtsx SSIS package test script, [357](#)
 pre-SSIS-package script, [358](#)
 SSISIncrementalLoad_Source.dbo.tblSource, [359](#)
 transforms (*see* Transforms)
 T-SQL reset rows script, [356](#)
 package metadata
 BimlScript, [369](#)
 C# application, [370](#)
 foreach loop, [372](#)

 FrameworkSQLGen.biml file, [370](#)
 output file, [373](#)
 PackageName and ExecutionOrder, [373](#)
 parameters, [372](#)
 stringbuilder, [371–372](#)
 Transact-SQL script, [370](#)
 SSIS design patterns engine
 add .NET namespaces and initial method, [364](#)
 close out task, [367](#)
 connection variable, [364](#)
 data flow task, [365](#)
 Execute SQL task, [365](#)
 GenerateTableNodes() method, [364](#)
 package node with .NET replacements, [365](#)
 packages node and starting loop, [364](#)
 SSISIncrementalLoad_Stage database, [362](#)
 T-SQL script, [360](#)
 testing
 dynamic SSIS package, [368](#)
 four SSIS packages, [367](#)

■ C

Catalog deployment model, [219](#)
Change data capture (CDC), [237–238](#)
Child-to-Parent variable pattern, [302](#)
Cloud loader
 ADO.Net destination adapter, [280](#)
 AWCleanLoadMaxID variable, [279](#)
 CleanTemperature table, [278](#)
 CleanTempLocalMaxID variable, [279](#)
 dbo.CleanTemperature table, [279](#)
 LoadMetrics table, [277–278](#), [280](#)
 OLE DB source adapter, [279–280](#)
 WeatherData Source Query, [279](#)
Clustered columnstore index (CCI), [175](#)
Command-line execution, [29](#)
Compilation error, [75](#)
Composite domain (CD), [101](#)

Connection managers
 DB2 source patterns
 ASCII, 129
 Data Link properties, 129–130
 EBCDIC, 129
 installation, 127–128
 OLE DB Provider, 128–129
 ADO.NET connection, 81, 89
 connection object, 80
 definition, 88
 ODBC, 89–91
 OLE DB, 81, 91–92
 script component, 81
 ConnectionString property, 325
 Customer Service and Support (CSS), 343

■ D

Data cleansing. *See* Data quality services (DQSs)
 The Data Conversion Transformation Editor, 184
 Data Definition Language (DDL), 40
 Data error
 known errors, 246
 missing data
 adding missing dimension member, 245
 fact tables, 243
 triage table, 246
 unknown member, 243
 simple errors, 242
 unhandled errors, 248
 Data Quality services (DQSs)
 cleansing transform
 advanced tab, 112
 asynchronous component, 117
 CodePlex, 113
 column status value, 114, 116
 Conditional Split transformation, 114–115
 Connection Manager, 108–109
 data flow logic processing, 116
 data flow toolbox, 108
 importing domain values, 123
 KB editor, corrections, 121
 list of columns, 111
 logs information, 113
 map columns, 109
 mapping tab, column selection, 110
 parallel processing, 117
 project name generation, 121
 right hand pane, 122
 rows filtering, Lookup transform, 118–120
 rows tracking, 117–118
 client utility, 101–102
 components, 101
 composite domains, 101

data quality client
 Administration section, 106
 data management, 102
 data quality project, 104
 default knowledge base, 106–107
 knowledge base management, 103
 online reference data, 107
 roles, 102–103
 knowledge base, 101
 Data taps, 35
 Data Transformation Services (DTSs), 71
 Data warehouse patterns
 data error (*see* Data error)
 ETL workflow, 248
 incremental loads (*see* Incremental loads)
 DB2 source patterns
 connection manager
 ASCII, 129
 Data Link properties, 129–130
 EBCDIC, 129
 installation, 127–128
 OLE DB Provider, 128–129
 database version, 126–127
 provider vendor, 127
 query
 Derive Parameters property, 131
 dynamic query, 132–133
 Edit Property Value window, 131
 SQL PL and PL/SQL, 130
 types, 125
 Debug execution, 28
 Deployment
 methods
 Custom Code, 336
 PowerShell, 337
 SQL API, 338
 Wizard command line, 335
 Package Deployment Model, 339
 Project Deployment Model, 331
 SSIS Catalog, 332
 Deployment models. *See* Execution patterns
 Domain management, 103–104
 DWLoader, 176
 Dynamic management functions (DMFs), 3
 Dynamic management views (DMVs), 3

■ E

Error logging
 add 0 ID rows, 417
 AppInstanceID and PkgInstanceID SSIS variables, 413
 build error logging table and stored procedure, 411
 ErrorDescription parameter, 415
 information event, 417

- Log Application Variables, 416
- Log.LogEvents table, 419
- Log Package Variables, 416
- log.SSISErrors table, 414
- Parameter Mapping page, 412
- Parent Package OnError
 - event handler, 413
- SourceName and ErrorDescription, 413
- T-SQL script, 414
- Execute Package Task Editor, 294–295, 301
- Execute Package Utility (DtExecUI), 30
- Execute SQL Task Editor, 299
- Execution patterns
 - command-line execution, 29
 - custom execution framework,
 - SQL Agent jobs, 56
 - debug execution, 28
 - Execute Package task, 57
 - Execute Package Utility, 30
 - file system package schedule, 54–55
 - Integration Server catalogs (*see* Integration Server catalog)
 - Integration Services catalogs, 30
 - managed code (*see* Managed code execution)
 - metadata-driven execution, 58
 - Project Reference Package, 57
 - Public Sub Main(), 28
 - public void Main(), 28
 - SQLAgent jobs, with custom execution
 - framework, 55
 - SSIS catalog, 53–54
- Expression language
 - assignment, 214
 - connection managers
 - backslash (\), 218
 - DATEPART function, 218
 - dynamic file name, 218
 - file-based sources, 218
 - FTP source, 218
 - project-level, 219
 - RIGHT function, 218
 - consistency, 214
 - control flow
 - precedence constraints
 - (*see* Precedence constraints)
 - task-level, 222
 - C-style languages, 215
 - data flow
 - branches, 223
 - custom task/component, 226
 - data cleansing, 222, 225
 - data source component, 225
 - dynamic logic, 226
 - Execute SQL task, 225
 - Script component, 225
 - Script task, 225

- SQL Server Data Quality Services/Master
 - Data Services, 225
 - third-party task/component, 226
- definition, 213
- evaluation, 214
- ForEach Loop, 214
- functional domains, 213
- limitations, 215
- maintenance scope, 214
- packages, 216
- simplicity, 214
- T-SQL, 215
- variables, 217
- Extract, load, and transform (ELT) processes, 177
- Extract, transform, and load (ETL) process, 177

■ F, G

- Flat file source patterns
 - archive file pattern, 163
 - Connection Manager Editor's Columns page, 138
 - data-staging pattern
 - CREATE TABLE Statement, 141
 - data flow path editor, 140
 - ETL developer, 140
 - mapping page, 141
 - Modified CREATE TABLE Statement, 141
 - OLE DB destination auto-mapping, 142
 - editor connection manager configuration, 136
 - footer row
 - Columns page, 148
 - Conditional Split transformation, 149
 - debug variable, 154
 - FileInUse function, 157
 - imports statement, 154
 - Main() Subroutine, 155
 - mapping package parameters, 158
 - MyFileFooterSource.csv, 148
 - parsed row, 150
 - row count and extract date, 149
 - Script task, 153
 - select variables, 154
 - TestParent.dtsx package, 158
 - WriteFileFooter.dtsx package, 153
 - header row
 - CreateNewOutputRows Subroutine, 160
 - Dynamic ConnectionString property, 161
 - Execute Package Task Editor, 162
 - Input0_ProcessInputRow Subroutine, 151
 - iRowNum Integer Variable, 151
 - WriteFileHeader.dtsx, 159
 - splitting record types, 145
 - StagingDB, 136
 - terminate streams, 146
 - variable-length row flat file, 143
- Foreach Loop Editor, 300

■ H

Hardware architecture, [172](#)

■ I, J

Incremental loads

CDC

- brute force detection, [239](#)
- CDC Source, [241](#)
- checksum-based detection, [238](#)
- Control task Editor, [241](#)
- ETL flow, [242](#)
- Hashbytes function, [239](#)
- historical load, [239](#)
- integration services, [237](#)
- supply-side incremental load tool, [237](#)
- table-by-table basis, [237](#)

definition, [227](#)

fact data, [228](#)

MERGE Statement

- auditing, [237](#)
- control flow design, [235](#)
- INSERT, UPDATE, and DELETE operations, [234](#)
- ISNULL() function, [236](#)
- OUTPUT clause, [237](#)
- SSIS load package, [235](#)
- T-SQL arsenal, [234](#)
- T-SQL MERGE functionality, [235](#)

native SSIS components

- load staging, [232](#)
- lookup operations, [231](#)
- moving parts, [229](#)
- SCD Wizard, [232](#)

SDD, [228](#)

systematic identification, [228](#)

Integration Server catalog

- custom execution framework
 - ADO.Net connection, [48](#)
 - Connection property, [48](#)
 - package execution twice, [52](#)
 - Parameter Mapping page, [48](#)
 - parent package control flow, [49–50](#)
 - Result Set page, [48](#)
 - table creation, [41](#)
 - T-SQL statements, [50–52](#)
 - Variable Mappings page, [49](#)

data taps

- DDL statement, [40](#)
- execution, [40](#)
- Flat File Connection manager, [39](#)
- OleDbDest TemperatureStage OLE DB destination adapter, [39](#)
- wrapper-stored procedure script, [35](#)

SSISDB database, [35](#)

T-SQL Script, [31](#)

wrapper stored procedure script, [32](#)

Integration Services, [176](#)

■ K

Knowledge base management, [103](#)

Knowledge discovery, [104](#)

■ L

LoadPackage method, [64](#)

Logging and reporting patterns

catalog

- catalog views, [286](#)
- existing reports, [289](#)
- internal tables, [285](#)
- levels changing, [287](#)
- logging levels, [287](#)
- new reports, [290](#)
- setting up, [283](#)

package

- design pattern, [283](#)
- OnError, [281](#)
- OnPreExecute, [281](#)
- OnVariableValueChanged, [281](#)
- reporting, [282](#)
- set up logging, [281](#)

Log sequence number (LSN), [237](#)

Lookup transforms, [118–120](#)

■ M, N

Managed code execution

demo application, [58](#)

frmMain form

- btnStartFileClick subroutine, [63–64](#)
- frmISTreeHelper, [66](#)
- frmMainHelper module, [61](#)
- ISTree form, [65](#)
- layout, [59](#)
- package execution, [69–70](#)
- package selection, [69](#)
- SSIS Catalog representation, [64](#)
- SSISDB, [64](#)
- View Code, [60](#)

Management Object Model (MOM), [337](#)

Manage Parameter Values, [311–312](#)

Massively parallel processing (MPP), [171](#)

Matching Policy activity, [104](#)

Metadata collection, [1](#)

central repository

- dba_monitor_SQLServerInstances, [6](#)
- server instance monitoring, [5–6](#)

- SSMS, 5
- T-SQL code, 4
- current data and log file sizes retrieval, 15
- data flow task, 14, 19, 23
- data storage and log file size information, 17
- dba_monitor_unusedIndexes table, 23
- dbaToolBox, 16
- description, 1
- DMVs, 16
- Foreach Loop container, 15
- information retrieval, 14
- integration services package, 1
- iterative framework, 7
 - Connection Manager, 9
 - dynamic properties connection, 14
 - editing result set, 10
 - Execute SQL Task Editor, 8–9
 - Foreach Loop container, 11
 - Foreach Loop Editor, 12
 - new integration services project, 7
 - package-scoped variables, 8
 - Property Expressions Editor, 13–14
 - Retrieve SQL Server Instances, 10
 - SQL Task Editor, 10
 - variable mappings, 12–13
 - variables menu, 7
- Mappings page, 24
- OLE DB destination connection manager, 18–19
- OLE DB Source Editor, 16
- package execution, 26
- parameter mapping, 25–26
- SQL server metadata catalog, 3
- SSDT, 1
- stored procedure, 23
- Unused Indexes Retrieval, 20–22
- Update LastMonitored Value, 25
- Metadata-driven execution, 58
- Microsoft Azure SQL Database (MASD), 275
 - AndyWeather database, 277
 - block diagram, 276
 - change detection, 276
- Mist, 344
- My Flat File connection manager, 137

■ O

- OData Connection Manager Editor, 254–255
- OData Source Editor, 256
- OLE DB
 - Command transforms, 269
 - destination, 269
 - Source Editor, 182
- Open Data (OData) protocol
 - Advanced Editor, 259
 - Connection Manager, 254

- default data types, 260
- EDM data type mapping, 253
- metadata document, 252
- Microsoft online services authentication, 254
- query options, 253
- Source Editor, 256
 - Columns page, 258
 - Preview dialog, 257
- Source Fields, 256

■ P, Q

- Package deployment model, 219, 339
- Parallel Data Warehouse (PDW)
 - APS appliance
 - CCI, 175
 - hardware architecture, 172
 - shared-nothing architecture, 175
 - software architecture, 173
 - compression estimation, 185
 - Data Conversion, 183
 - data import pattern
 - Destination Database, PDW code, 186
 - FactSales table, 180
 - loading modes, 188
 - loading server, 178
 - multithreading, 189
 - OLE DB Source Editor, 182
 - package, 181
 - partition function, 179
 - SQL Server database creation, 179
 - SQL Server PDW Connection Manager Editor, 187
 - SQL Server PDW Destination, 186
 - SQL Server PDW Destination Editor, 187
 - staging database, 185
 - Destination Editor, 187
 - distributed table, 185
 - limitations, 190
 - load data
 - Control node, 176
 - Control VM, 176
 - DWLoader *vs.* Integration Services, 176
 - ETL *vs.* ELT, 177
 - MPP, 171
 - replicated table, 185
 - staging database, 184
- Parameter-based configuration model
 - connection managers, 313
 - database table
 - files retrieval, SQL task, 323–326
 - sample values, 323
 - SQL definition, 323
 - default configuration, 314
 - Description field, 306
 - DTEXEC

Parameter-based configuration model (*cont.*)

- command-line switches, 320
- file system, 320
- SSIS Catalog, 321–322
- dynamic package executions, 327–329
- entry-point package, 312
- expressible data flow component properties, 308
- Expression Builder dialog, 308–309
- package execution, 317–319
- package-level parameters, 305–306
- Parameterize UI, 309–310
- parameter values, 305, 317
- Property Expressions Editor dialog, 307
- Required property, validation, 306
- Script task, 326
- Sensitive parameter, 306
- server environment variable, 315–316
- Solution Explorer, 306
- Variables window, SQL Server 307, 2012
- Visual Studio configurations, 310
- Parent-Child patterns
 - Child-Parent variable pattern, 302
 - dynamic child package
 - Create and Insert statements, 297
 - DelayValidation property, 300
 - DesignPatterns, 297
 - Execute Package Task Editor, 301
 - Execute SQL Task Editor, 299
 - Foreach Loop Editor, 300
 - packageListObject, 298
 - Package List Table, 298
 - master package
 - child package, 294
 - parameter binding configuration, 295
- Precedence constraints
 - editor, 220
 - flow-based completion, 219
 - non-standard notation, 221
 - Success constraints, 219
 - TRUNCATE TABLE execution, 219
- PreExecute method, 208
- Project Deployment Model, 331

■ R

- Reference Data Services (RDS), 107
- ReleaseConnection method, 208
- RetainSameConnection property, 269
- Roll-Back Load, 189

■ S

- Scripting patterns, 71
 - design patterns
 - connection managers (*see* Connection managers)
 - naming conventions, 85

- variable data types, 84
- variable syntax code, 83
- variable visibility, 83
- DTS, 71
- maintenance
 - copy/paste, 78
 - custom tasks/components, 79
 - external assemblies, 78
 - source control, 79
- package design, 72
- script component, 77
- Script editor
 - compiler, 74–75
 - features, 72
 - .NET Runtime, 74
 - Project Explorer, 73
- script task, 75
 - configuration, 76
 - vs.* Script component, 71
- Shared-nothing architecture, 175
- Slowly changing dimensions (SCDs), 228
 - advantages and disadvantages, 273
 - data flow components, 267
 - large dimensions, 261
 - merge pattern
 - control flow, 271
 - Type 1 change, 271
 - Type 2 changes, 272
 - optimal performance, 268
 - small number of rows, 261
 - third-party components, 269
- wizard, 232
 - column change types, 264
 - dimension table and keys, 262
 - fixed and changing attribute options, 265
 - historical attribute options, 266
 - inferred dimension members, 267
- Software architecture, 173
- sp_MSforeachdb procedure, 23
- SQL Server Data Tools-Business Intelligence (SSDT-BI), 1
- SQL Server Data Tools (SSDTs), 281
- SQL Server Integration Services (SSIS) packages.
 - See* Execution patterns
- SQL Server Management Studio (SSMS), 5, 179, 314
- SQL Server PDW Connection Manager Editor, 188
- SQL Server Reporting Services (SSRS), 420
- SQL Server source patterns
 - connection manager (*see* Connection managers)
 - setting up, 87
 - Source Assistant, 97, 99
 - source component
 - ADO.NET data access, 95–96
 - Columns menu, 97
 - Data Flow design, 93
 - data translations, 97
 - OLE DB data access, 96

- OLE DB Source Editor, 94
 - SSIS Toolbox, 92–93
 - SSIS Catalog, 332
 - SSISDB database, 285
 - SSIS execution and monitoring framework
 - monitoring execution
 - add variable window, 408
 - AppPackageID variable, 409
 - build instance tables and stored procedures, 399
 - error logging (*see* Error logging)
 - ErrorTest.dtsx, 403, 405
 - Execute SQL Task Editor, 409
 - Log Application Success, 402
 - log.LogPackageFailure, 409
 - log.LogPackageSuccess, 409
 - log.SSISAppInstance table, 399, 405
 - log.SSISPackageInstance table and stored procedures, 406
 - Log Start of Application, 402
 - OnError event handler, 402
 - parameter mapping page, 409
 - Parent.dtsx, 402
 - succeed/fail SSIS package, 403
 - test executions, 409
 - T-SQL Script, 404
 - Unit-testing, 403
 - parent-child pattern
 - cfg.AddSSISPackages Stored Procedure, 385
 - cfg.Packages table, 386
 - cfg.Schema and cfg.packages table, 384
 - Child1.dtsx package, 377–378
 - Child2.dtsx, 383
 - package deployment model, 379–380
 - Parent.dtsx package, 380
 - project deployment model, 378
 - Property Expressions Editor, 381
 - SSISConfig database, 383
 - User::ChildPackagePath, 381–382
 - SSIS applications
 - add application, 389
 - ApplicationName, 393
 - ApplicationName variable, 397
 - cfg.Applications and cfg.AddSSISApplication, 387
 - cfg.GetSSISApplication statement, 394
 - cfg.GetSSISApplication Stored Procedure, 392
 - Child1 and Child2 SSIS packages, 391
 - create cfg.AppPackages and
 - cfg.AddSSISApplicationPackage, 390
 - Foreach Loop Container Editor, 398
 - many-to-many relationship, 390
 - one-to-many relationships, 389
 - T-SQL script, 394
 - User::ChildPackagePath variable, 397
 - SSRS
 - application instance data, 424
 - Application Instance report, 430
 - Application Instance report preview, 425
 - application name, 423
 - color Application Instance, 426
 - Events report, 432
 - Failed hyperlinks, 433
 - Package Instance report, 428
 - refreshed Application Instance report, 425
 - rpt_ReturnAppInstanceHeader, 423
 - rpt.ReturnAppInstanceHeader, 422
 - rpt.ReturnAppInstanceHeader stored procedure, 423–424
 - rpt.ReturnErrors Stored Procedure, 428
 - rpt.ReturnEvents Stored Procedure, 431
 - rpt.ReturnPkgInstanceHeader Stored Procedure, 426
 - rpt.Schema and rpt.ReturnAppInstanceHeader Stored Procedure, 421
 - single application instance, 428
 - SSISConfig, 420
 - Strong data types, 138
- **T, U**
- Transforms
- biml listing file, 355
 - dataflow XML node, 353
 - lookup metadata, 351
 - <ExecuteSQL> tag, 354
 - <InputPath> tag, 351
 - <Lookup> tag, 351
 - <OleDbDestination> tag, 352
 - OleDbDestination node, 353
 - OLE DB source adapter, 350
 - output column nodes, 351
 - XML node, 352
- TryParse() method, 216, 226
- T-SQL data definition language (DDL) script, 392
- T-SQL MERGE statement, 270
- **V**
- VariableDispenser.LockOneForRead() function, 84
 - Variable mappings, 12
 - Visual Basic for Applications (VBAs), 213
 - Visual Studio configurations, 310
 - Vulcan technology, 344
- **W**
- Windows Server 2008 R2, 178

■ X, Y, Z

- XmlReader class, [210–211](#)
- XmlSerializer class, [209–210](#)
- XML source component
 - attributes, [193](#)
 - column metadata, [193](#)
 - column-to-data-type mappings, [205](#)
 - configuration, [203–204](#)
 - connection manager, [207](#)
 - customers database table, [196](#)
 - IsSorted property value, advanced editor, [196–197](#)
 - LINQ, [210](#)
 - Merge Join transform, [199–200](#)
 - .NET Framework, [203](#)
 - output and column names generation, [195](#)
 - schema file, [193](#)
 - ScriptMain class, [207–208](#)
 - Select Script Component Type, [204](#)
 - simple element/subelement structure, [193](#)
 - SortKeyPosition property value,
 - advanced editor, [198](#)
 - source code, [208–209](#)
 - XML document, customer information, [194](#)
 - XmlReader class, [210–212](#)
 - XML Schema Definition Tool, [203](#)
 - XmlSerializer class, [209–210](#)
 - XSLT
 - data flatten/denormalization, [200](#)
 - destination schema, [200](#)
 - XML document, [202–203](#)
 - XML task configuration, [201–202](#)

SQL Server Integration Services Design Patterns

Second Edition



Andy Leonard

Tim Mitchell

Matt Masson

Jessica Moss

Michelle Ufford

Apress®

SQL Server Integration Services Design Patterns

Copyright © 2014 by Andy Leonard, Tim Mitchell, Matt Masson, Jessica Moss, and Michelle Ufford

This work is subject to copyright. All rights are reserved by the Publisher, whether the whole or part of the material is concerned, specifically the rights of translation, reprinting, reuse of illustrations, recitation, broadcasting, reproduction on microfilms or in any other physical way, and transmission or information storage and retrieval, electronic adaptation, computer software, or by similar or dissimilar methodology now known or hereafter developed. Exempted from this legal reservation are brief excerpts in connection with reviews or scholarly analysis or material supplied specifically for the purpose of being entered and executed on a computer system, for exclusive use by the purchaser of the work. Duplication of this publication or parts thereof is permitted only under the provisions of the Copyright Law of the Publisher's location, in its current version, and permission for use must always be obtained from Springer. Permissions for use may be obtained through RightsLink at the Copyright Clearance Center. Violations are liable to prosecution under the respective Copyright Law.

ISBN-13 (pbk): 978-1-4842-0083-4

ISBN-13 (electronic): 978-1-4842-0082-7

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

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

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

Managing Director: Welmoed Spahr

Lead Editor: Jonathan Gennick

Technical Reviewers: Bill Fellows and Louis Davidson

Editorial Board: Steve Anglin, Mark Beckner, Ewan Buckingham, Gary Cornell, Louise Corrigan, Jim DeWolf,

Jonathan Gennick, Jonathan Hassell, Robert Hutchinson, Michelle Lowman, James Markham,

Matthew Moodie, Jeff Olson, Jeffrey Pepper, Douglas Pundick, Ben Renow-Clarke, Dominic Shakeshaft,

Gwenan Spearing, Matt Wade, Steve Weiss

Coordinating Editor: Jill Balzano

Copy Editor: Rebecca Rider

Compositor: SPi Global

Indexer: SPi Global

Artist: SPi Global

Cover Designer: Anna Ishchenko

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

For information on translations, please e-mail rights@apress.com, or visit www.apress.com.

Apress and friends of ED books may be purchased in bulk for academic, corporate, or promotional use. eBook versions and licenses are also available for most titles. For more information, reference our Special Bulk Sales-eBook Licensing web page at www.apress.com/bulk-sales.

Contents

First-Edition Foreword.....	xv
About the Authors.....	xvii
About the Technical Reviewer	xix
■ Chapter 1: Metadata Collection	1
About SQL Server Data Tools.....	1
A Peek at the Final Product.....	1
SQL Server Metadatalog.....	3
sys.dm_os_performance_counters.....	3
sys.dm_db_index_usage_stats.....	3
sys.dm_os_sys_info.....	3
sys.tables	3
sys.indexes.....	3
sys.partitions	4
sys.allocation_units.....	4
Setting Up the Central Repository	4
The Iterative Framework	6
Metadata Collection	14
Summary.....	26
■ Chapter 2: Execution Patterns	27
Building the Demonstration SSIS Package.....	27
Debug Execution.....	28
Command-Line Execution.....	29
Execute Package Utility	30

The SQL Server 2014 Integration Services Service.....	30
Integration Services Catalogs.....	30
Integration Server Catalog Stored Procedures	31
Scheduling SSIS Package Execution.....	53
Scheduling an SSIS Package.....	53
Scheduling a File System Package.....	54
Running SQL Server Agent Jobs with the Custom Execution Framework.....	55
Running the Custom Execution Framework with SQL Server Agent.....	56
Execute Package Task	57
Execution from Managed Code	58
The Demo Application.....	58
The frmMain Form	59
Conclusion.....	70
■ Chapter 3: Scripting Patterns	71
The Toolset	71
Should I Use Script?	72
The Script Editor.....	72
The Script Task	75
The Script Component	77
Script Maintenance Patterns.....	78
Code Reuse.....	78
Source Control.....	79
Scripting Design Patterns.....	79
Connection Managers and Scripting	80
Variables	82
Naming Patterns.....	85
Conclusion.....	85

■ Chapter 4: SQL Server Source Patterns	87
Setting Up a Source.....	87
Selecting a SQL Server Connection Manager and Provider	88
ADO.NET.....	89
ODBC.....	89
OLE DB.....	91
Creating a SQL Server Source Component.....	92
Writing a SQL Server Source Component Query	95
ADO.NET Data Access	95
OLE DB Data Access	96
Waste Not, Want Not.....	97
Data Translations	97
Source Assistant.....	97
Summary	99
■ Chapter 5: Data Correction with Data Quality Services	101
Overview of Data Quality Services	101
Using the Data Quality Client	102
Using DQS with SSIS	108
DQS Cleansing Transform	108
DQS Extensions on CodePlex.....	113
Cleansing Data in the Data Flow	114
Handling the Output of the DQS Cleansing Transform	114
Performance Considerations	117
Approving and Importing Cleansing Rules	121
Conclusion.....	123
■ Chapter 6: DB2 Source Patterns	125
DB2 Database Family	125
Selecting a DB2 Provider.....	126
Find the Database Version.....	126
Pick Provider Vendor.....	127

Connecting to a DB2 Database.....	127
Querying the DB2 Database	130
DB2 Source Component Parameters	131
DB2 Source Component Dynamic Queries.....	132
Summary	133
■ Chapter 7: Flat File Source Patterns.....	135
Flat File Sources.....	135
Moving to SSIS!	136
Strong-Typing the Data.....	138
Introducing a Data-Staging Pattern.....	140
Variable-Length Rows	143
Reading into a Data Flow.....	144
Splitting Record Types	145
Terminating the Streams	146
Header and Footer Rows.....	147
Consuming a Footer Row.....	148
Consuming a Header Row	150
Producing a Footer Row	152
Producing a Header Row	159
The Archive File Pattern	163
Summary	169
■ Chapter 8: Loading a PDW Region in APS.....	171
Massively Parallel Processing.....	171
APS Appliance Overview	172
Hardware Architecture.....	172
Software Architecture	173
Shared-Nothing Architecture	175
Clustered Columnstore Indexes.....	175

Loading Data	176
DWLoader vs. Integration Services.....	176
ETL vs. ELT	177
Data Import Pattern for PDW	178
Prerequisites.....	178
Preparing the Data.....	179
Package Overview	181
The Data Source	181
The Data Transformation	183
The Data Destination	184
Multithreading	189
Limitations.....	190
Summary	191
■ Chapter 9: XML Patterns.....	193
Using the XML Source	193
Dealing with Multiple Outputs	194
Making Things Easier with XSLT	200
Using a Script Component.....	203
Configuring the Script Component	203
Processing XML with XmlSerializer	209
Processing XML with XmlReader and LINQ to XML	210
Conclusion.....	212
■ Chapter 10: Expression Language Patterns.....	213
Getting to Know the Expression Language.....	213
What Is the Expression Language?.....	213
Why Use Expressions?.....	214
Language Essentials.....	215
Limitations.....	215

Putting the Expression Language to Work.....	216
Package Expressions.....	216
Variable Expressions	217
Connection Managers.....	217
Project-Level Connection Managers.....	219
Control Flow.....	219
Data Flow Expressions	222
Conclusion.....	226
■ Chapter 11: Data Warehouse Patterns.....	227
Incremental Loads.....	227
What Is an Incremental Load?	227
Why Incremental Loads?	228
The Slowly Changing Dimension	228
Incremental Loads of Fact Data.....	228
Incremental Loads in SSIS	228
Native SSIS Components.....	229
The Slowly Changing Dimension Wizard	232
The MERGE Statement.....	234
Change Data Capture (CDC).....	237
Data Errors	242
Simple Errors.....	242
Missing Data.....	243
Coding to Allow Errors	246
Data Warehouse ETL Workflow.....	248
Dividing Up the Work	248
One Package = One Unit of Work	249
Conclusion.....	250

■ Chapter 12: OData Source	251
Understanding the OData Protocol	251
Data Type Mappings	252
Query Options	253
Configuring the OData Connection Manager	254
Enabling Microsoft Online Services Authentication.....	254
Configuring the Source Component	256
Overriding Data Types	259
Conclusion.....	260
■ Chapter 13: Slowly Changing Dimensions.....	261
The Slowly Changing Dimension Transform.....	261
Running the Wizard	262
Using the Transformations.....	267
Optimizing Performance.....	268
Third-Party SCD Components.....	269
Merge Pattern	270
Handling Type 1 Changes	271
Handling Type 2 Changes	272
Conclusion.....	272
■ Chapter 14: Loading the Cloud	275
Interacting with the Cloud	275
Incremental Loads to Azure SQL Database	276
Change Detection	276
New Rows (Only)	276
Building the Cloud Loader	277
Conclusion.....	280

■ Chapter 15: Logging and Reporting Patterns	281
Package Logging and Reporting	281
Setting Up Package Logging.....	281
Reporting on Package Logging.....	282
Design Pattern: Package Executions	283
Catalog Logging and Reporting	283
Setting Up Catalog Logging	283
Catalog Tables	285
Changing Logging Levels After the Fact	286
Design Patterns	287
Changing the Logging Level	287
Using the Existing Reports.....	289
Creating New Reports.....	290
Summary	291
■ Chapter 16: Parent-Child Patterns.....	293
Master Package Pattern	293
Assign the Child Package	294
Configure Parameter Binding	295
Dynamic Child Package Pattern	296
Child-to-Parent Variable Pattern	302
Conclusion.....	303
■ Chapter 17: Configuration	305
Parameters.....	305
Configuring Your Package Using Parameters	307
Using the Parametrize Dialog	309
Creating Visual Studio Configurations	310
Specifying Entry-Point Packages.....	312
Connection Managers	313

Parameter Configuration on the Server.....	313
Default Configuration.....	314
Server Environments	315
Default Parameter Values Using T-SQL	317
Package Execution Through the SSIS Catalog.....	317
Parameters with DTEXEC	320
Projects on the File System	320
Projects in the SSIS Catalog	321
Dynamic Configurations	322
Configuring from a Database Table	323
Setting Values Using a Script Task	326
Dynamic Package Executions.....	327
Conclusion.....	329
■ Chapter 18: Deployment	331
Project Deployment Model	331
SSIS Catalog.....	332
Deployment Methods	334
Deployment from the Command Line	335
Deployment Using Custom Code	336
Deployment Using PowerShell.....	337
Deployment Using SQL	338
Package Deployment Model.....	339
Conclusion.....	341
■ Chapter 19: Business Intelligence Markup Language	343
A Brief History of Business Intelligence Markup Language	343
Building Your First Biml File	344
Building a Basic Incremental Load SSIS Package.....	347
Creating Databases and Tables	347
Adding Metadata	349

Specifying a Data Flow Task.....	350
Adding Transforms.....	350
Testing the Biml.....	356
Using Biml as an SSIS Design Patterns Engine.....	360
Time for a Test.....	367
Conclusion.....	368
■ Chapter 20: Biml and SSIS Frameworks.....	369
Using Biml with an SSIS Framework.....	369
Adding SSIS Package Metadata to the Framework.....	369
Executing the Biml File.....	374
Generating the SSIS Command-Line.....	375
Summarizing.....	376
■ Appendix A: Evolution of an SSIS Framework.....	377
Starting in the Middle.....	377
Introducing SSIS Applications.....	387
A Note About Relationships.....	389
Retrieving SSIS Applications in T-SQL.....	392
Retrieving SSIS Applications in SSIS.....	396
Monitoring Execution.....	399
Building Application Instance Logging.....	399
Building Package Instance Logging.....	406
Building Error Logging.....	410
Reporting Execution Metrics.....	420
Conclusion.....	434
Index.....	435

First-Edition Foreword

For me, one of the great pleasures of working at Microsoft was shepherding new products from concept to release. However, it was even more fulfilling to witness the birth and growth of new communities of users, for what is a product without a user? Just bits and bytes on a disk. In my role as Group Product Manager of the SQL Server Integration Services team, it was my privilege to watch the evolution of both the SSIS application and the social network of users.

The Integration Services team, under the exceptional leadership of Kamal Hathi, delivered a product in 2005—SQL Server Integration Services—that was intended to be not only a powerful application in its own right, but a platform for customers and partners to extend and expand as their data integration needs changed and grew over time. Over the years (and through several versions of the product) SQL Server Integration Services has grown to become an industry-leading technology.

When we started developing what users now call SSIS, anyone building a data warehouse had only two choices: expensive, highly specialized tools for extraction, transformation, and loading (ETL), or tedious, difficult-to-maintain, custom coding. With SSIS we wanted to break through those traditional restrictions: to deliver a truly scalable tool, simple enough for the beginner, but with the extensibility and programmability of a platform for the expert.

Little did we anticipate how eagerly the SQL Server user community would embrace this tool! Our user base grew quickly, and, as in any group endeavor, natural leaders emerged. The authors of this splendid book are, quite simply, among the most outstanding contributors to the SSIS social network. They are leaders not only because of their skills, but because of their tireless support and commitment to helping others. This book distills that learning, and that community focus, into a volume to keep by your keyboard for years.

The challenge with a tool such as SSIS is that there are simply so many possibilities facing the user. If I can choose a prebuilt component, which one do I choose? If I can extend the capabilities with script, when should I do that? How do I choose between the many ways to load a slowly-changing-dimension table, or for handling XML?

SQL Server 2012 Integration Services Design Patterns not only provides solutions to such problems; even more usefully, this book channels the authors' extensive experience into *patterns*. In recent years, design patterns have proved their value to software developers as flexible templates for addressing recurring problems that still need specific implementation details. *SSIS Design Patterns* takes this approach, quite uniquely, into the world of data warehousing and ETL.

The result is a collaborative work by experts, suitable for beginners and advanced users alike.

Even though I moved on from the SSIS team, and from Microsoft, some years ago now, it is a pleasure for me to remain in touch with the user community I admire so much. And it is an honor for me to introduce you to this much-anticipated and valuable book.

Happy integrating!

—Donald Farmer
VP Product Management, **QlikTech**

About the Authors



Andy Leonard is an SSIS trainer and consultant, SQL Server database and Integration Services developer, SQL Server data warehouse developer, community mentor, SQL Server Most Valuable Professional (MVP), SQLBlog.com blogger, and engineer. He is co-author of *Professional SQL Server 2005 Integration Services* and *SQL Server MVP Deep Dives*. His background includes web application architecture and development, Visual Basic, ASP, SQL Server Integration Services (SSIS), and data warehouse development using SQL Server 2000, 2005, and 2008.

Tim Mitchell is a business intelligence consultant, database developer, speaker, and trainer. He has been working with SQL Server for more than eight years, primarily in business intelligence, ETL/SSIS, database development, and reporting. He has earned a number of industry certifications, holds a bachelor's degree in computer science from Texas A&M University at Commerce, and is a Microsoft SQL Server Most Valuable Professional (MVP). Tim is a business intelligence consultant for Artis Consulting in the Dallas, Texas area. As an active member of the community, Tim has spoken at venues including numerous SQL Saturday events, Houston Tech Fest, and various user groups and PASS virtual chapters. He is a board member and speaker at the North Texas SQL Server User Group in Dallas, serves as the co-chair of the PASS BI Virtual Chapter, and is an active volunteer for PASS. Tim is an author and forum contributor on SQLServerCentral.com and has published dozens of SQL Server training videos on SQLShare.com. You can visit his website and blog at TimMitchell.net or follow him on Twitter at [@Tim_Mitchell](https://twitter.com/Tim_Mitchell).

Matt Masson is a software development engineer working with the SQL Server Integration Services (SSIS) team. Matt has worked on many aspects of the SSIS product, including upgrade, performance, and overall user experience. He is a frequent presenter at Microsoft conferences and maintains the SSIS Team blog (<http://blogs.msdn.com/b/mattm/>). Prior to joining Microsoft in 2006, Matt was a developer on a number of business intelligence reporting and analytical products. He lives in Montreal, Quebec, and works remotely with his Redmond-based team.



Jessica M. Moss is a well-known author, and speaker on Microsoft SQL Server business intelligence. She has created numerous data warehouse and business intelligence solutions for companies in different industries, and has delivered training courses on Integration Services, Reporting Services, and Analysis Services. While working for a major clothing retailer, Jessica participated in the SQL Server 2005 TAP program, where she developed best implementation practices for Integration Services. Jessica has authored technical content for multiple magazines, websites, and books, and has spoken internationally at conferences such as the PASS Community Summit, SharePoint Connections, and the SQLTeach International Conference. As a strong proponent of developing user-to-user community relations, Jessica actively participates in local user groups and code camps in central Virginia. In addition, Jessica volunteers her time to help educate people through the PASS organization.



Michelle Ufford is a Microsoft Most Valuable Professional (MVP) and principal engineer on the Data Platform team at GoDaddy. She has experience with all stages of the data lifecycle, from OLTP to data warehousing, and from relational data to Big Data. Michelle specializes in development of very large databases (VLDB) and scalable database architecture, and enjoys database automation and data monetization. She is an active member of the SQL Server community and a frequent presenter, most notably at PASS Summit. Michelle has a very popular blog at SQLFool.com and can be found on Twitter @sqlfool.com.

About the Technical Reviewer



Bill Fellows is a Microsoft SQL Server MVP and a business intelligence professional in Kansas City, Missouri. He's been focused on data integration, database design, and development since 1999. He is the organizer of the Kansas City SQL Saturdays and a regular speaker at SQL Server community events.