CHAPTER 11

PowerShell Versus SSIS

Introduction

In this chapter we will discuss using PowerShell as an extract, transform, and load (ETL) tool as an alternative to using SQL Server Integration Services (SSIS). We start by comparing PowerShell to SSIS. Then we define the general pattern of an ETL tool—extract from a source, transform the data, and write the data to a destination. In covering the application of these concepts, we employ a fictitious ETL use case that requires a number of files to be loaded into SQL Server tables.

To streamline PowerShell as an ETL tool, we need a reusable framework that makes code easy to develop and maintain while minimizing data-specific coding. We will use the PowerShell pipeline as a data flow. In prior chapters, we built reusable functions to get data, so the challenge here is to create reusable code to support loading data into a database. We'll review a function that builds a mapping collection that defines the source to destination column names. By leveraging the mapping collection together with the pipeline, we will explain how we can load data into database tables with just function calls. The exception to this is when we need column transformations. While these do require data-specific code, we will demonstrate how it can be isolated to a single function that the pipeline passes through. The transformation function can modify or replace the pipeline. Referencing the use case, we demonstrate how to load a number of different file formats into tables. We show how advanced transformations like code/value look-ups and file joins can easily be accomplished. The latter is provided as a function by the Microsoft PowerShell team. We are using the modules `umd_database`, `umd_etl_functions`, and `umd_northwind_etl` in this chapter, so you need to import these in order to run the examples.

Advantages of PowerShell as an ETL Tool

PowerShell has a number of advantages that make it a good choice for ETL work. It's free and pre-installed, and it has features that are missing from SSIS, such as the ability to support user interaction. PowerShell can easily leverage any Windows program, adding functionality that may be difficult to implement in SSIS. The ability to create script modules that encapsulate functions provides a high degree of reusability. SSIS binds its processes to the underlying metadata, thus making packages sensitive to changes, but PowerShell code can be dynamic and adapt automatically to changes. Finally, SSIS upgrades can be time consuming and costly. PowerShell upgrades occur seamlessly with upgrades to the .NET framework.

Pre-Installed and Free

PowerShell is free and is pre-installed on Windows 7 and later. SSIS is part of SQL Server, which requires a paid license. To develop SSIS packages, SQL Server Data Tools must be installed on the client. For organizations that use SSIS to load SQL Server data warehouses, this may not be an issue. But there are times when SSIS is not available. For example, PowerShell could be used by a vendor to load tables to non-Microsoft databases, such as Oracle or MySQL, where the client does not have SQL Server installed. In this way, PowerShell supports database-platform independence, which is needed for multi-platform vendors. There are scenarios in which an end user needs to load database tables but the organization does not want to give the user the SSIS client tools or allow them access to execute packages in the SSIS catalog. PowerShell resolves these issues.

Supports Interaction

There are times when an ETL script may need to get information from the user at run time. Perhaps the script needs to get a filter parameter or must allow the user to select among some options. SSIS is designed to be executed in batch mode, with no support for user interaction. PowerShell supports interactivity, as we have seen. Windows forms can be easily incorporated for a professional interface, or cmdlets such as `Read-Host` can be used for a simpler implementation.

PowerShell Goes Where SSIS Can't

PowerShell is integrated with the .NET framework, enabling it to programmatically access any Windows resource. In this book, we have seen examples of PowerShell using the Internet Explorer client to navigate the Web, programmatically manipulate office documents, create environment variables, query the event log, and even speak using the speech API (SAPI). But PowerShell can also interface with SharePoint, Exchange Server, Active Directory, and more. The point is that PowerShell was designed to do anything, while SSIS was designed for ETL. There are times when we need those other capabilities. Additionally, Microsoft's cloud environment, Azure, does not have SSIS available. However, PowerShell is available on Azure and is generally promoted as the tool to use to automate work.

Reusability and Extensibility

PowerShell provides the ability to create functions and package them into modules, thus extending PowerShell with new capabilities and a high level of reusability. We can develop a function, deploy it, and never need to write code to do that task again. It can be called anywhere the function is needed. There are many free modules available that add functionality that is not built-in. The great thing is that using modules is a seamless experience. Once imported, the functions are called just like built-in cmdlets. SSIS has little support for reusability. Each package is designed to do a specific task. Other than changing configuration settings, the behavior is static. Even SSIS script tasks and script components cannot be shared. Instead, they need to be copied wherever they are needed, creating code redundancy with related maintenance issues. Note: Technically, functions written in PowerShell script are not considered cmdlets. Only when it is written in a compiled language like C# is it called a cmdlet. However, this distinction is just semantics, and I use the terms *function* and *cmdlet* interchangeably throughout the book.

Dynamic Code

A limitation to SSIS is that packages are tightly coupled to the metadata of the sources and destinations. If a new column is added to a source table, the package may fail because it stores details about the underlying structure. Moreover, there are times when it would be useful to create a dynamic process that could automatically add new tables to the ETL process. An example of this would be a package that loads source tables to a staging area. If a new table is added, it would be nice if the package could just create the table on the destination server and load the data. The database catalog views can provide the metadata needed to do this, but SSIS does not support this level of dynamic processing without getting into some fairly complex scripting. PowerShell is interpreted and therefore is inherently dynamic. Scripts can query the source metadata, create the target tables, and load the data.

Upgrades and Code Dependencies

SSIS is part of SQL Server, and as such gets upgraded with SQL Server releases. Oftentimes, a new release of SSIS is not completely backward compatible and legacy packages must be converted to a new format. This is not a trivial task. For example, to upgrade from SQL Server 2008 R2 to SQL Server 2012, all the packages have to go through the Migration Wizard and be reviewed for any unsupported code. The level of effort depends on how many packages are involved. PowerShell, however, is not affected by SQL Server releases. New versions of PowerShell are released with new versions of the Windows Management Framework, i.e., the .NET library. I have used PowerShell 2.0 through 4.0 and have found no backward compatibility issues. One explanation for this is that PowerShell is so pervasive throughout Windows environments, including Azure, that it must be backward compatible. Microsoft would suffer the most if it were not. Additionally, since PowerShell code is stored in text files, there are no potential file-conversion issues as can be the case with SSIS.

Advantages of SSIS

SSIS has some strengths over PowerShell as an ETL tool. SSIS performs well and can load data very quickly. It provides a sophisticated visual development environment. As part of SQL Server, SSIS integrates extremely well with the SQL Server environment. Over the years, SSIS has developed robust configuration features that simplify ETL maintenance and deployment. SSIS is delivered with a number of useful controls and transformation components designed for ETL work.

Performance

When moving a large volume of data, SSIS can't be beat. It accomplishes this partly by the way it constructs the data-flow buffer. It analyzes the flow from start to finish and allocates enough memory to accommodate the sources, transformations, and derived columns so that the data does not need to be copied more than once. Destination adapters can be adjusted, especially for SQL Server, to optimize batch sizes and use features like fast load. Performance is the forte of SSIS and is often the best reason to choose it for an ETL task.

Visual Development Interface

The Visual Studio development environment for SSIS provides a quick and intuitive interface to develop packages. The ability to drop controls onto the canvas and interactively set properties is appealing to many people. It is ideal for building ETL processes incrementally since the developer can create and test individual components. Unless a script task or script component is needed, no code needs to be written. All work can be done visually. This can be particularly attractive to people without a programming background.

SQL Server Integration and Configuration

With the release of SQL Server 2012, SSIS added tight SQL Server integration. Packages can be grouped into projects that can be deployed directly to the SSIS catalog in SQL Server. SQL Server Management Studio provides a rich visual interface to this catalog, supporting the ability to configure project and package settings. Groups of settings can be saved into a container called an environment, and packages configured to get values from these environments. This makes deployment easy, since the same environment container can exist on each server with different values. SQL Agent jobs running SSIS packages can be configured to use environments as well. A number of built-in reports are available from the SSIS catalog, including reports that provide detailed information about package failures. PowerShell does not inherently support this degree of configurability or integration with SQL Server.

Built-In Components

While PowerShell can do the things that SSIS can do,some coding must be done first. The SSIS built-in data-flow transformations provide fast look up, match, join, and merge functionality. A number of column-transformation functions are available to modify columns in the data flow. For some of these, there is a PowerShell equivalent. For others, it must be written or sought out on the Internet, perhaps in the form of a module.

The PowerShell ETL Framework

ETL (extract, transform, and load), as the name implies, involves three steps. First, extract data from a source. This can be a flat file, an Office document, an XML file, a Web service, or any data source. Usually some level of modification to the data needs to be performed before it is loaded to its destination. Sometimes data needs to be scrubbed of bad values or reformatted to meet reporting requirements. Other times, additional information must be looked up from another source. Often times, the input data needs to be split and sent to multiple destinations. All these tasks are broadly categorized as transforming the data. Once the data transformation is complete, it is loaded to the destination. There are some tasks that don't easily fit into this simple model, such as creating a table in the destination database, unzipping files, or sending email notifications. These additional tasks support the ETL process.

PowerShell is not designed to be an ETL tool. We could just write custom scripts and functions to copy, transform, and load data for each data source. But this would eliminate the benefit of reusability. Instead, let's consider how an ETL framework that supports reusability and maintainability can be created in PowerShell. Let's review a proposed framework in Figure 11-1.

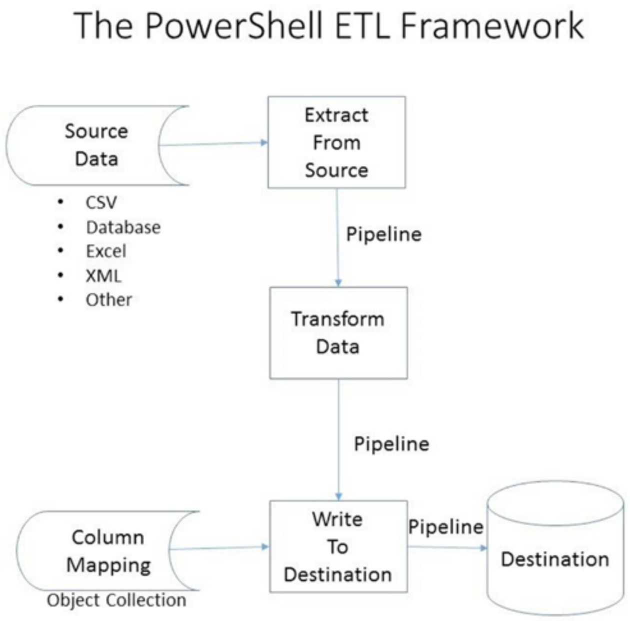

Figure 11-1. *An ETL framework for PowerShell*

SSIS moves data from the source to the destination through something called a data buffer. We don't have this available in PowerShell. However, we do have something analogous, which is the pipeline. In fact, conceptually the SSIS data buffer and the PowerShell pipeline are very similar. So, in our framework we will use the pipeline to move data. This works well, because once data has entered the pipeline, it can be manipulated in a consistent manner. The source becomes irrelevant. In Figure 11-1, we see the first step is for a task to pull data from a source and stream it into the pipeline. From there it flows into a Transform Data step where the pipeline properties are modified, dropped, or added. Once the transformations are complete, the pipeline is passed to the task, which writes it to the destination. The task is not labeled *Load*, because the pipeline data could be used to add, update, or delete the destination data. Note: SSIS destination components only support inserting the data-flow buffer into the destination.

Notice that the "Write to Destination" step has a column-mapping collection as input. To support writing data to the destination, a mapping of source columns to destination columns is needed. Even if no transformations are required, the source and destination column names may be different. We could hard code the mapping in each script, but then our ETL code would be custom developed with no reusability. Instead, we want to create a way to support source-to-destination mapping that can be reused. Note: All the reusable functions we'll look at next that are part of the framework can be found in the module `umd_etl_functions`.

Mapping

To support the framework in Figure 11-1, we'll create a function that develops a mapping collection of source-to-destination columns. This mapping will be used by the output process to determine where the data should be written. Let's look at the function that follows:

```
function Add-UdfMapping {
 [CmdletBinding()]
        param (
              [string]   $incolumn              , # Column from input pipeline.
              [string]   $outcolumn             , # Output column name.
              [string]   $outcolumndelimiter    , # Character to delimit output column.
              [bool]     $iskey                   # True = Key column, False = Not key column
          )

    $mapping = New-Object -TypeName System.Object
    $mapping | Add-Member -MemberType NoteProperty -Name "InColumn" -Value "$incolumn"
    $mapping | Add-Member -MemberType NoteProperty -Name "OutColumn" -Value $outcolumn
    $mapping | Add-Member -MemberType NoteProperty -Name "OutColumnDelimiter" `
                         -Value "$outcolumndelimiter"
    $mapping | Add-Member -MemberType NoteProperty -Name "IsKey" -Value "$iskey"

    RETURN $mapping

}
```

This function accepts the parameters `$incolumn`, which is the source column name, `$outcolumn`, which is the destination column name, `$outcolumndelimiter`, which will be used when generating a SQL statement to delimit values, and `$iskey`, which is `$true` if the column is the key and `$false` if it is not. This function uses a pattern we have seen before—we use `psobject` and attach properties to hold values we need to retain. The function in our example creates an object, attaches the properties, and returns the object to the caller. It's a simple function but it provides a nice abstraction to support our ETL framework.

To get our feet wet using the `Add-UdfMapping` function, we'll start with a simple load. We'll load the state tax rate CSV file to a SQL Server table. Before we create the mapping collection, we can use the SQL script in Listing 11-1 to create the destination table on SQL Server. For the examples, the server instance is `(local)` and the database is `Development`. Change these settings to fit your environment.

Listing 11-1. Create the StateSalesTaxRate table

```
CREATE TABLE [dbo].[StateSalesTaxRate](
        [StateProvinceCD] [varchar](255) NULL,
        [StateProvinceSalesTaxRate] [decimal](5, 2) NULL,
        [CreateDate] [datetime] NULL
)
```

The state sales tax file has only two columns, i.e. `StateCode` and `SalesTaxRate`. The source column names are different than the target SQL Server column names. We'll create a mapping collection to map the source to target columns using the statements here:

```
Import-Module umd_etl_functions
$mappingset = @()  # Create a collection object.
$mappingset += (Add-UdfMapping "StateCode" "StateProvinceCD" "'" $true)
$mappingset += (Add-UdfMapping "SalesTaxRate" "StateProvinceSalesTaxRate" "" $false)
```

The first line creates an empty collection. Then, `Add-UdfMapping` is called, passing in the source column name `StateCode`; the destination column name `StateProvinceCD`; the character to use as a value delimiter, i.e., `'`; and `$true` to the `iskey` Boolean, indicating this is the key column. The last line adds a mapping for the `SalesTaxRate` column to the mapping collection. Notice the `iskey` parameter is `$false`, because this is not the primary key column. Enter the statement that follows to see the contents of `$mappingset`:

```
$mappingset
```

This should display the following output to the console:

```
InColumn     OutColumn                   OutColumnDelimiter  IsKey
--------     ---------                   ------------------  -----
StateCode    StateProvinceCD             '                   True
SalesTaxRate StateProvinceSalesTaxRate                       False
```

The purpose of the mapping collection is to help our code generate the required SQL statements to load the data. We iterate over the collection to build our statements. Building a string that contains a SQL statement can get a little messy. For a given SQL statement, there is a fixed set of words, the input column names, input column values, and the output column names. Consider the `Insert` statement for the state sales tax file:

```
Insert into [dbo].[StateSalesTaxRate]
([StateProvinceCD],
 [StateProvinceSalesTaxRate])
Values
('MA',
.06)
```

We would need a statement like this one for each row in the file. The words "Insert into" are static; i.e., they will always be the same. This is followed by the target table name and column name, so those must be variables, but once assigned they will stay the same for all inserts into the table. Then, the characters "Values (" appear, which are static. The source column values separated by commas must be updated for each insert; i.e., we will need to iterate over the source collection, generate the insert statement, and execute it for each set of values. We want to be able to generate Merge, Update, and Delete statements. To provide for maximum flexibility, let's create two helper functions. What follows is a function that takes a mapping collection as a parameter and returns the list of the input column names as a string separated by commas. See here:

```
function Get-UdfInColumnList
{
 [CmdletBinding()]
        param (
               [psobject]  $mapping
              )

   $inlist = ""                           # Just intializing
   foreach ($map in $mapping)
   {
     $inlist += $map.InColumn + ", "
   }

   $inlist = $inlist.Substring(0,$inlist.Length - 2)
   Return $inlist

}
```

This function takes the mapping collection as a parameter. The code iterates over the collection, extracting the source column name, i.e., the InColumn property, and appending it and a trailing comma to the variable $inlist. When the loop ends, $inlist contains the list of columns separated by commas. However, there is one extra comma at the end that we remove in the statement just before the Return statement. Finally, $inlist is returned to the caller.

Assuming the umd_etl_functions module has been imported, we can call the function Get-UdfInColumnList with the mapping set we created, as shown here:

```
Get-UdfInColumnList $mappingset
```

We should see the following output:

```
StateCode, SalesTaxRate
```

For the Insert statement, we'll need a list of the output column names separated by commas. The function here will do that:

```
function Get-UdfOutColumnList
{
 [CmdletBinding()]
        param (
                 [psobject] $mapping
          )
```

```
   $outlist = ""                             # Just initializing
   foreach ($map in $mapping)
   {
     $outlist += $map.OutColumn + ", "
   }

   $outlist = $outlist.Substring(0,$outlist.Length - 2)
   Return $outlist

}
```

`Get-UdfOutColumnList` takes a mapping collection as a parameter. The code iterates over the collection, extracting the destination column name, i.e., the `OutColumn` property, and appending it and a trailing comma to the variable `$outlist`. When the loop ends, `$outlist` contains the list of columns separated by commas. However, there is one extra comma at the end that we remove in the statement just before the `Return` statement. Finally, `$outlist` is returned to the caller.

Assuming the `umd_etl_functions` module has been imported, we can call the function with the mapping set we created, as shown here:

```
Get-UdfOutColumnList $mappingset
```

We should see the following line written to the console:

```
StateProvinceCD, StateProvinceSalesTaxRate
```

The helper functions are great, but what we really need is a function that will build the SQL statement for us. The function that follows uses the helper functions to generate an `Insert`, `Merge`, or `Delete` statement based on what is requested. See here:

```
function Get-UdfSQLDML {
 [CmdletBinding()]
        param (
                 [psobject] $mapping,
                 [parameter(mandatory=$true,
                            HelpMessage="Enter the DML operation.")]
                            [ValidateSet("Insert","Merge", "Delete")]
                 [string]   $dmloperation,
                 [string]   $destinationtablename
               )
   [string] $sqldml = ""
   Switch ($dmloperation)
   {
   Insert
     {
        $sqldml = "insert into $destinationtablename (" + (Get-UdfOutColumn $mapping) + ") `
                   values (`$valuelist);" ; break
     }
   Merge
     {
        $key = $mapping | Where-Object { $_.IsKey -eq $true }
        $sourcekey = $key.Incolumn
        $targetkey = $key.OutColumn
        $sourcecollist = (Get-UdfInColumnList $mapping);
        $insertstatement = `
        "insert (" + (Get-UdfOutColumn $mapping) + ") values (`$valuelist)";

        $sqldml = "
               Merge into $destinationtablename as Target USING (VALUES ( `$valuelist )) as `
               Source ( $sourcecollist `
         ) ON Target.$targetkey = Source.$sourcekey `
           WHEN MATCHED THEN                           `
                UPDATE SET `$updatestatement
           WHEN NOT MATCHED BY TARGET THEN $insertstatement;"; break;
     }
   Delete
     {
        $key = $mapping | Where-Object { $_.IsKey -eq $true }
        $targetkey = $key.OutColumn
        $sqldml = "Delete from $destinationtablename where $targetkey = `$sourcekeyvalue;"
        break
     }
   }

   Return $sqldml
}
```

The function Get-UdfSQLDML is doing a lot. Its purpose is to return the requested SQL statement to the caller using the mapping-set collection. Let's review the code step by step, starting with the function header, copied here:

```
function Get-UdfSQLDML {
 [CmdletBinding()]
        param (
                 [psobject] $mapping,
                 [parameter(mandatory=$true,
                            HelpMessage="Enter the DML operation.")]
                            [ValidateSet("Insert","Merge", "Delete")]
                 [string]   $dmloperation,
                 [string]   $destinationtablename
               )
```

In the function header, we can see the function accepts $mapping, i.e., the mapping-set collection, as the first parameter, followed by $dmloperation, which is the type of SQL statement desired, and $destinationtablename, which is the name of the target table. Note: The ValidationSet attribute on the $dmloperation parameter limits accepted values to Insert, Merge, or Delete.

The next part of the function evaluates the $dmloperation parameter to determine the code to execute. Let's look at the code now:

```
[string] $sqldml = ""
Switch ($dmloperation)
{
Insert
  {
     $sqldml = "insert into $destinationtablename (" + (Get-UdfOutColumn $mapping) + ") `
                values (`$valuelist);" ; break
  }
```

The Switch command provides a concise way to test a condition and execute code based on the result. Based on the value of $dmloperation, the related code is executed. If $dmloperation is Insert, then the variable $sqldml is assigned a value to generate an insert statement. Notice the call to Get-UdfOutColumn. It is enclosed in parentheses, so the function will be executed before being added to the string; i.e., it appends the list of destination columns extracted from $mapping. Another interesting feature is the use of the ` character before the variable $valuelist. The reason for this is that we want the variable name, not its value, to be in the string for now. The break statement tells PowerShell to exit without performing the other comparisons. Later, when we are iterating through the source values, we will expand the string to fill in the values.

If the caller passed Merge as the $dmloperation parameter, the code here would execute:

```
Merge
  {
     $key = $mapping | Where-Object { $_.IsKey -eq $true }
     $sourcekey = $key.Incolumn
     $targetkey = $key.OutColumn
     $sourcecollist = (Get-UdfInColumnList $mapping);
     $insertstatement =   `
     "insert (" + (Get-UdfOutColumn $mapping) + ") values (`$valuelist)";

     $sqldml = "
     Merge into $destinationtablename as Target USING (VALUES ( `$valuelist )) as `
     Source ( $sourcecollist `
      ) ON Target.$targetkey = Source.$sourcekey `
        WHEN MATCHED THEN                        `
             UPDATE SET `$updatestatement
        WHEN NOT MATCHED BY TARGET THEN $insertstatement;"; break;
  }
```

The Merge statement is more complex than the Insert statement, and more powerful. The Merge statement will match the source data to the target data based on a key column, and then will execute any number of operations depending on whether there is a match or not. In our case, if there is a match, we want to do an update. If there is no match, we want to do an insert. The first statement in the Merge block locates the key column by piping the mapping set through a Where-Object cmdlet, which returns the row where the IsKey attribute is $true. For simplicity, this is limited to one key column. Then, the source key column name is stored in $sourcekey, and the target key column name is stored in $targetkey. The function Get-UdfInColumnList is called, passing $mapping, with the result stored in $sourcecollist. The Insert portion of the Merge statement is assigned to $insertstatement. The rest of the code, up until the break command, is actually one statement broken up over multiple lines using the continuation character. Since the destination table name will be the same for all the generated Merge statements, it is expanded immediately. However, the value list, `$valuelist`, will need to be changed for each source row. Therefore, `$valuelist` is preceded by the escape character ` so it will not be expanded yet. The source column list, target key, and source key will be the same for all generated statements, so they are expanded immediately, i.e., in the variables `$sourcecollist`, `$targetkey`, and `$sourcekey`. The `update` part of the `Merge` statement needs to use each set of row values, so we don't expand it yet—in other words, `$updatestatement` is preceded with the escape character ` so it will not be expanded yet. The last line of the `Merge` statement includes the variable `$insertstatement` we built earlier, and it is immediately expanded, except for the embedded `$valuelist` variable, because that is preceded with the escape character. The `break` statement is needed to break out of the `Switch` statement code block.

The third type of SQL statement is a `Delete` statement. The code that generates this statement type is seen here:

```
Delete
  {
     $key = $mapping | Where-Object { $_.IsKey -eq $true }
     $targetkey = $key.OutColumn
     $sqldml = "Delete from $destinationtablename where $targetkey = `$sourcekeyvalue;"
     break
  }
```

The first line in the `Delete` code block pipes the mapping collection, `$mapping`, into the `Where-Object` cmdlet, which returns just the row flagged as the key. The statement after that stores the target table key column name in `$targetkey`. The last statement builds the `delete` statement using the variables `$destinationtablename` and `$targetkey`, which are expanded immediately, and `$sourcekeyvalue`, which is not expanded, i.e., is prefixed with the escape character.

The function `Get-UdfSQLDML` is a building block that will be used to generate the statements to load data to a database table. Assuming the `umd_etl_functions` module has been imported, here are some examples of calling those statements:

```
Get-UdfSQLDML $mappingset -dmloperation Insert "dbo.Products"
Get-UdfSQLDML $mappingset -dmloperation Merge "dbo.Products"
Get-UdfSQLDML $mappingset -dmloperation Delete "dbo.Products"
```

This produces the following output to the console. Blank lines are added between statements for readability:

```
insert into dbo.Products (StateProvinceCD, StateProvinceSalesTaxRate) values ($valuelist);

Merge into dbo.Products as Target USING (VALUES ( $valuelist )) as Source ( StateCode,
SalesTaxRate
          ) ON Target.StateProvinceCD = Source.StateCode
            WHEN MATCHED THEN
                 UPDATE SET $updatestatement
            WHEN NOT MATCHED BY TARGET THEN insert (StateProvinceCD,
            StateProvinceSalesTaxRate)
            values ($valuelist);

Delete from dbo.Products where StateProvinceCD = $sourcekeyvalue;
```

In these generated statements, notice that there are unexpanded variables. They will be expanded after we have loaded them with values from the source.

We could look to the umd_database module and use the custom connection object created using the function New-UdfConnection, as discussed in chapter 7, to write to the database. However, that object's SQL methods open and close the connection for every execution, which would add a lot of unnecessary overhead. Instead, we will create a simple function that will create and open a database connection and pass it back to us. So we don't have to pass as many parameters, we will still use the connection object created by New-UdfConnection. Let's look at the function:

```
function Get-UdfADOConnection
{
 [CmdletBinding()]
        param (
              [psobject]$connection
          )

if ($connection.UseCredential -eq 'Y')
  {
     $pw =  $sqlpw
     $pw.MakeReadOnly()

     $SqlCredential = `
     new-object System.Data.SqlClient.SqlCredential($connection.UserID, $connection.Password)
     $conn = new-object System.Data.SqlClient.SqlConnection($connection.ConnectionString, `
     $SqlCredential)
  }
  else
  {
     $conn = new-object System.Data.SqlClient.SqlConnection($connection.ConnectionString);
  }

  $conn.Open()

  Return $conn
 }
```

Get-UdfADOConnection creates and opens an ADO connection object and passes it back to the caller. It takes the custom database object created by the function New-UdfConnection from the umd_database module. Although we won't use the returned object to load the data, it provides a convenient way to define the required connection properties. The function checks to see if the UseCredential property is set to 'Y' for yes. If so, it opens the connection using credentials. Otherwise, it just opens the connection using the object's connection string property.

The next function, Invoke-UdfSQLDML, uses these functions to do the work of writing to the database. We stream the data into the function via the pipeline. Let's look at the code:

```
function Invoke-UdfSQLDML
{
 [CmdletBinding()]
        param (
                 [Parameter(ValueFromPipeline=$True)]$mypipe = "default",
                 [Parameter(ValueFromPipeline=$False,Mandatory=$True,Position=1)]
                 [psobject] $mapping,
                 [parameter(ValueFromPipeline=$False,mandatory=$true,Position=2,
                            HelpMessage="Enter the DML operation.")]
                            [ValidateSet("Insert","Update","Merge","Delete")]
```

```
                [string]   $dmloperation,
                [Parameter(ValueFromPipeline=$False,Mandatory=$True,Position=3)]
                [string]   $destinationtablename,
                [psobject] $connection

              )

      begin
      {
       $sqldml = Get-UdfSQLDML $mapping -dmloperation $dmloperation "$destinationtablename"
       $dbconn = Get-UdfADOConnection $connection
       $command = New-Object system.data.sqlclient.Sqlcommand($dbconn)
      }

      process
      {
         $values = ""
         $updatestatement = ""

          foreach($map in $mapping)
          {
             $prop = $map.InColumn.Replace("[", "")
             $prop = $prop.Replace("]", "")
             $delimiter = $map.OutColumnDelimiter
             $values = $values + $delimiter + $_.$prop + $delimiter + ","
             $updatestatement += $map.OutColumn + " = Source." + $map.InColumn + ","
             #  Get the key column value...
             if ($map.IsKey -eq $true)
             {
                  $sourcekeyvalue = $_.$prop
             }
          }

          # Strip off the last comma.
          $updatestatement = $updatestatement.Substring(0,$updatestatement.Length - 1)
          $valuelist = $values.Substring(0,$values.Length - 1)  # Strip off the last comma.
          $sqlstatement = $ExecutionContext.InvokeCommand.ExpandString($sqldml)
          $sqlstatement = $sqlstatement.Replace(",,", ",null,")
          Write-Verbose $sqlstatement

          # Write to database...
          $command.Connection = $dbconn
          $command.CommandText = $sqlstatement
          $command.ExecuteNonQuery()

      }
      end
      {
          $dbconn.Close()
      }

}
```

Let's step through this function to see how it works. We'll start with the function header copied here:

```
function Invoke-UdfSQLDML
{
 [CmdletBinding()]
        param (
                 [Parameter(ValueFromPipeline=$True)]$mypipe = "default",
                 [Parameter(ValueFromPipeline=$False,Mandatory=$True,Position=1)]
                 [psobject] $mapping,
                 [parameter(ValueFromPipeline=$False,mandatory=$true,Position=2,
                            HelpMessage="Enter the DML operation.")]
                            [ValidateSet("Insert","Merge", "Delete")]
                 [string]   $dmloperation,
                 [Parameter(ValueFromPipeline=$False,Mandatory=$True,Position=3)]
                 [string]   $destinationtablename,
                 [psobject] $connection

               )
```

We can see that the first parameter, `$mypipe`, will come in via the pipeline, because the attribute `ValueFromPipeline` is `true`. The second parameter, `$mapping`, is the mapping collection. The third parameter, `$dmloperation`, is the type of SQL statement we want the function to use. The `ValidateSet` attribute limits the allowed values and will be visible to the developer via Intellisense. The fourth parameter, `$destinationtablename`, is the target table to which we want the data written. The last parameter, `$connection`, is a `psobject` created by `New-UdfConnection`. We use the function `New-UdfConnection`, which is part of the `umd_database` module, to create the object to be passed into the last parameter.

`Invoke-UdfSQLDML` is going to process the pipeline, thus it has three parts, which are `begin`, `process`, and `end`. Let's look at the `begin` block:

```
begin
    {
     $sqldml = Get-UdfSQLDML $mapping -dmloperation $dmloperation "$destinationtablename"
     $dbconn = Get-UdfADOConnection $connection
     $command = new-object system.data.sqlclient.Sqlcommand($dbconn)
    }
```

The begin block will execute only once, and it does so before the pipeline is received. It's a good place to do setup work. The first statement calls the function `Get-UdfSQLDML`, passing in the mapping collection, the SQL statement type, and destination table name as parameters. The SQL statement will be returned and stored in the variable `$sqldml`. The SQL statement will then be complete except for the parts requiring data values. The second statement calls `Get-UdfADOConnection`, which will open a database connection and return the reference to `$dbconn`. We don't want to use the SQL methods attached to the connection object passed in as a parameter, because they open and close the connection for each row, which would add a lot of overhead. We want to open the connection once, process all the source rows, and then close the connection. The last line in the begin block creates a command object named `$command`. The command object is the object that submits SQL statements to the database. After the begin block executes, the function is ready to send commands to the database. Let's look at the process block that actually writes to the database:

```
process
    {
       $values = ""
       $updatestatement = ""
```

```
        foreach($map in $mapping)
        {
           $prop = $map.InColumn.Replace("[", "")
           $prop = $prop.Replace("]", "")
           $delimiter = $map.OutColumnDelimiter
           $values = $values + $delimiter + $_.$prop + $delimiter + ","
           $updatestatement += $map.OutColumn + " = Source." + $map.InColumn + ","
           #  Get the key column value...
           if ($map.IsKey -eq $true)
           {
                $sourcekeyvalue = $_.$prop
           }
        }

        # Strip off the last comma.
        $updatestatement = $updatestatement.Substring(0,$updatestatement.Length - 1
        $valuelist = $values.Substring(0,$values.Length - 1)  # Strip off the last comma.
        $sqlstatement = $ExecutionContext.InvokeCommand.ExpandString($sqldml)
        $sqlstatement = $sqlstatement.Replace(",,", ",null,")
        Write-Verbose $sqlstatement

        # Write to database...
        $command.Connection = $dbconn
        $command.CommandText = $sqlstatement
        $command.ExecuteNonQuery()

    }
```

Recall that the begin block prepared the SQL statement, except for supplying the column values. The process block needs to fill in the Values section of the SQL statement and execute the statement. For the State Tax table load, we want to do an Insert. When the process block begins executing, the SQL statement will look like the one that follows. We just need to fill $valuelist with the list of values to be inserted and then tell PowerShell to expand the statement, replacing $valuelist with the variable's contents. Let's look at that statement now:

```
insert into dbo.Products (StateProvinceCD, StateProvinceSalesTaxRate) values ($valuelist);
```

The first two statements in the process block just initialize the variables $values and $updatestatement. Then a ForEach loop iterates over the mapping-set collection so that we can use the mapping to build the values part of the SQL statement. In testing, I discovered that if the source columns have spaces in the names, you need to enclose them in brackets. However, when the data is moved into the pipeline, the brackets are dropped. To generate the SQL statement, we need to remove the bracket characters if present, which is what the first two lines of the ForEach block are doing. Values in a SQL statement require different enclosing delimiters depending on the data type, i.e., single quote for a string and nothing for numbers. The line "$delimiter = $map.OutColumnDelimiter" is getting the value delimiter from the mapping set and storing it in $delimiter. The line after this creates the values part of the SQL statement. The variable name, $values, is critical here, because it must be the same name as the variable embedded in the SQL statement we generated earlier.

After this, the statement $updatestatement += $map.OutColumn + " = Source." + $map.InColumn + "," uses the mapping set to build the input-to-output column assignment needed for an Update statement. In the case of the State Tax table, this will not be used, because we're doing an insert. If we were doing an update, the variable $updatestatement would match the variable name in the generated SQL statement. Next, we see an If block that checks for the IsKey value of true and, if found, loads the value of the column into $sourcekeyvalue.

This variable is needed to create the SQL `Merge` and `Delete` statements, i.e. `where key = somevalue`. By the time the `ForEach` loop is finished, we will have iterated over the entire set of columns, building required parts of the SQL statement. Column names and values in a SQL statement are separated by commas, which the function added to the end of each as the statement was being built. However, this leaves an extra comma at the end of the list. The first two statements after the `ForEach` loop just strip off the trailing comma.

The statement after this leverages the ability to force PowerShell to expand variables in a string on demand. For example, `'$sqlstatement = $ExecutionContext.InvokeCommand.ExpandString($sqldml)'` returns the string `$sqldml`, with all the embedded variables replaced with the variable contents to `$sqlstatement`. The statement is now almost ready to be executed, but we need to do one more thing first. Missing values may cause the statement to fail. They can easily be identified because the string will have two commas together, which SQL Server will reject. The statement `'$sqlstatement = $sqlstatement.Replace(",,", ",null,")'` will replace the double commas with `",null,"` which tells SQL Server this is a null value.

The `Write-Verbose` statement after this is a handy feature that allows us to output the SQL statement to the console. With all this string manipulation, it is easy to get something wrong so it's a good idea to be able to see the statement. The nice thing is that it will only display when the `Verbose` parameter is used. The final three statements in the `process` block attach the connection to the command object, load the SQL statement into the command object, and execute the statement.

The last block in this function is the `End` block, which is copied here:

```
end
    {
        $dbconn.Close()
    }
```

The end block just closes the database connection. The `End` block is only executed after the entire pipeline has been processed, so it's the perfect place to do clean up.

We've created the state tax file columns to destination column mapping collection so we are ready to use the example function to actually load the table. The statements in Listing 11-2 will do this. Note: Listing 11-2 is also coded as a function in the module `umd_northwind_etl` that is named `Invoke-UdfStateSalesTaxLoad`.

Listing 11-2. Script to load the state sales tax table

```
Import-Module umd_database -Force
Import-Module umd_etl_functions -Force

#  State Code List
$global:referencepath = $env:HomeDrive + $env:HOMEPATH + "\Documents\"

$salestax = Import-CSV ($global:referencepath + "StateSalesTaxRates.csv")

<#  Define the mapping... #>
$mappingset = @()  # Create a collection object.
$mappingset += (Add-UdfMapping "StateCode" "StateProvinceCD" "'" $true)
$mappingset += (Add-UdfMapping "SalesTaxRate" "StateProvinceSalesTaxRate" "" $false)

Import-Module umd_database

<#  Define the SQL Server Target  #>
[psobject] $SqlServer1 = New-Object psobject
New-UdfConnection ([ref]$SqlServer1)
```

```
$SqlServer1.ConnectionType = 'ADO'
$SqlServer1.DatabaseType = 'SqlServer'
$SqlServer1.Server = '(local)'
$SqlServer1.DatabaseName = 'Development'
$SqlServer1.UseCredential = 'N'
$SqlServer1.SetAuthenticationType('Integrated')
$SqlServer1.BuildConnectionString()

# Load the table...
$SqlServer1.RunSQL("truncate table [dbo].[StateSalesTaxRate]", $false)
$salestax |
Invoke-UdfSQLDML -Mapping $mappingset -DmlOperation "Insert" -Destinationtablename `
"dbo.StateSalesTaxRate" -Connection $SqlServer1 -Verbose
```

The first two statements import the `umd_etl_functions` and `umd_database` modules. The next line, after the comment, assigns the path to the user's `Documents` folder, where the state tax rate file should be, to variable `$global:referencepath`. Note: Make sure you copy the file there. Then, we load the file into `$salestax`. The code in parentheses forces the path value to be assigned before importing the file. The three lines after that create the mapping set. Since our functions depend on the `umd_database`, it is imported. The lines between '`<#  Define the SQL Server Target  #>`' and '`# Load the table...`' create a database connection object, which is the custom `psobject` returned by `New-UdfConnection`. As mentioned before, we are not going to use this object's built-in SQL methods, because they open and close the connection on each iteration. However, it still provides handy functionality for our needs. The second-to-last line uses the custom database object to truncate the table. The last line pipes the loaded `$salestax` object into the `Invoke-UdfSQLDML` function, which loads it into the SQL Server table. The call passes the parameters `$mappingset`, `"Insert"` as the `DMLOperation`, `dbo.StateSalesTaxRate` as the `Destinationtablename`, and the custom connection object `$SqlServer1` as the `Connection` parameter. Bear in mind, the functions are reusable, so the actual ETL for this table load would be just the script we just looked at. Other table loads would have the same structure.

After running the script, we should get messages to the console like the ones here:

```
VERBOSE: insert into dbo.StateSalesTaxRate (StateProvinceCD, StateProvinceSalesTaxRate)
values ('WA',0.07);
1
VERBOSE: insert into dbo.StateSalesTaxRate (StateProvinceCD, StateProvinceSalesTaxRate)
values ('WV',0.06);
1
VERBOSE: insert into dbo.StateSalesTaxRate (StateProvinceCD, StateProvinceSalesTaxRate)
values ('WI',0.05);
1
VERBOSE: insert into dbo.StateSalesTaxRate (StateProvinceCD, StateProvinceSalesTaxRate)
values ('WY',0.04);
1
VERBOSE: insert into dbo.StateSalesTaxRate (StateProvinceCD, StateProvinceSalesTaxRate)
values ('DC',0.06);
1
```

The output here just shows the last few lines. We can see that the function `Invoke-UdfSQLDML` generated and submitted an `Insert` statement for each row in the input file. A nice feature is that once a data source is loaded into the pipeline, whether it be from another database table, an Excel spreadsheet, XML file, or whatever, we can load it using the function `Invoke-UdfSQLDML`.

Transformations

What happens when we need to change values in the pipeline or add new columns? We can do both of these tasks by processing the pipeline before we send it to the function that will write to the table. I considered trying to abstract this process as I did with the SQL statement generation. For example, we could extend the mapping collection to include a list of the transformations we want performed. However, this would add a great deal of complexity, and it would be difficult to cover all possible transformations. Instead, we'll look at using a source-specific approach.

As a use case, let's assume that we know one of the rows in the sales tax file is wrong. We'll assume that the tax rate for Massachusetts should be .08, and we want to replace the tax rate for just that state. Also, we were told that management wants to consolidate the United States tax rates with the state/province tax rates in other countries. To support this, we want to add a new column named Country and hard code the value 'US' so we will be able to filter out the United Sates tax rates from other countries after it has been loaded into a consolidated table. Let's look at the function to transform the state sales tax data:

```
function Invoke-UdfStateTaxFileTransformation
{
 [CmdletBinding()]
        param (
                 [Parameter(ValueFromPipeline=$True)]$mypipe = "default"
              )
    process
    {

      $mypipe | Add-Member -MemberType NoteProperty -Name "Country" -Value "US"

      if ($mypipe.StateCode -eq "MA" ) { $mypipe.SalesTaxRate = .08 }

      Return $mypipe
    }
}
```

Invoke-UdfStateTaxFileTransformation takes the pipeline as input via parameter $mypipe. Since our goal is just to process the pipeline, we only need the process block. The first line in the process block just adds a NoteProperty to the pipeline called Country with a value of 'US'. Notice that the Add-Member cmdlet call does not include the Passthru parameter because we only want to return the modified pipe. Note also that if you find you are getting duplicate rows in the pipeline when you do a transformation, you should check to see if the Passthru parameter is being passed on any of the Add-Member calls. The second line in the process block tests for a StateCode equal to 'MA' for Massachusetts and, if found, sets the SalesTaxRate property to .08. The last statement returns the modified pipe to the caller.

Let's use Listing 11-3 to test the function `Invoke-UdfStateTaxFileTransformation`.

Listing 11-3. Sales tax transformation

```
Import-Module umd_etl_functions

$salestax = Import-CSV ($env:HomeDrive + $env:HOMEPATH + "\Documents\StateSalesTaxRates.csv")
$salestax | Invoke-UdfStateTaxFileTransformation
```

The first line in Listing 11-3 loads `$salestax` from the sales tax file. The second line pipes this into the transformation function `Invoke-UdfStateTaxFileTransformation`. A portion of the console output is shown to confirm the Massachusetts tax rate was updated:

```
StateCode    SalesTaxRate    Country
---------    ------------    -------
MD           0.06            US
MA           0.08            US
MI           0.06            US
```

We can also see the new `Country` attribute was added. This approach makes transforming data easy. We have all the methods, properties, and cmdlets of PowerShell to use. Although we do not have the transformation GUI that SSIS has, PowerShell provides a much richer set of transformations in the form of cmdlets, including support for regular expressions.

The Northwind Company Use Case

To provide a comprehensive explanation on how to use PowerShell to do ETL work, we will use a fairly extensive use case, which is documented in Figure 11-2.

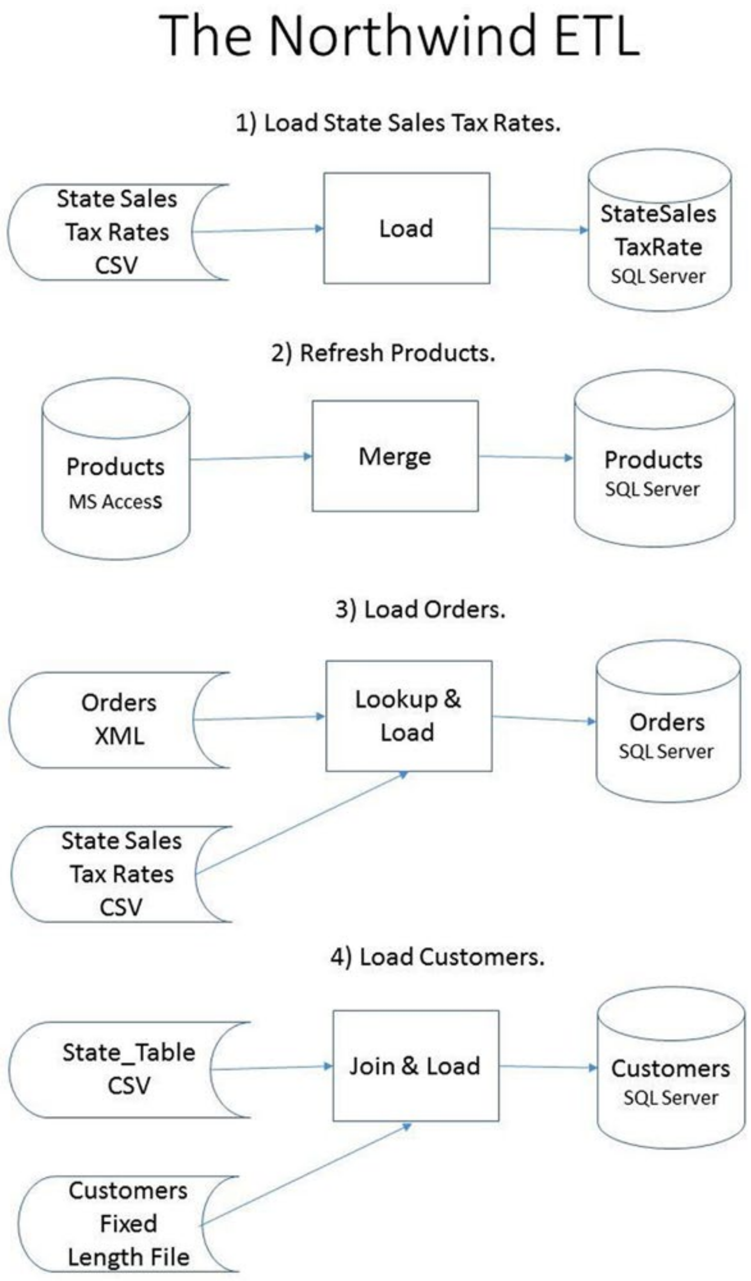

Figure 11-2. *Northwind ETL use case*

Northwind wants to load data into their new data warehouse for reporting. They are starting small and only requested the tables in Figure 11-2 be loaded. We already did Step 1, the state sales tax table load. For Step 2, we need to update the products table in SQL Server. The products data is maintained in a Microsoft

Access database table. We will load the data to the SQL Server table using a Merge statement to update existing rows and insert products not on the existing table. Step 3 loads orders data, which is stored in XML format. Unfortunately, the sales tax rate on the orders file is wrong, so we need to look up the correct tax rate in the state sales tax file. In Step 4, we need to load a customer fixed-length flat file, i.e., a file where the columns are padded with spaces so they are always the same length. We will need to join the customer data to the state table in order to get the state name and region. For convenience, the ETL scripts have been packaged into the module umd_northwind_etl, which you can reference.

The Products Table Load

Now that we've walked through how the framework helps us load data into tables using the state sales tax file, let's move on to the next step in our ETL process, the product table load. As before, we'll use the mapping collection to generate the SQL statements, but there are a couple of differences. First, the source data will be a Microsoft Access table. Second, we will use the SQL Merge statement to process the load. When a key match is found, the columns in the target table will be updated. If no match is found, the row is inserted. The source for this data is the Northwind sample database, which is available in the Access 2013 templates. To run the code in Listing 11-4, you must be in the 32-bit version of PowerShell, i.e., the one that has (x86) in the name. This is because the driver used is a 32-bit driver.

We need to create the target table in SQL Server, which we can do with the script in Listing 11-4.

Listing 11-4. Create the products table

```
CREATE TABLE [dbo].[Products](
        [SupplierIDs] [nvarchar](max) NULL,
        [ID] [int] NOT NULL,
        [ProductCode] [nvarchar](25) NULL,
        [ProductName] [nvarchar](50) NULL,
        [Description] [nvarchar](max) NULL,
        [StandardCost] [money] NULL,
        [ListPrice] [money] NOT NULL,
        [ReorderLevel] [smallint] NULL,
        [TargetLevel] [int] NULL,
        [QuantityPerUnit] [nvarchar](50) NULL,
        [Discontinued] [varchar](5) NOT NULL,
        [MinimumReorderQuantity] [smallint] NULL,
        [Category] [nvarchar](50) NULL,
        [Attachments] [nvarchar](max) NULL
)
```

The only transformation we need is to replace a null Minimum Reorder Quantity with a zero. The function to do this is seen here:

```
function  Invoke-UdfProductsTransformation
{
 [CmdletBinding()]
        param (
                 [Parameter(ValueFromPipeline=$True)]$mypipe = "default"
               )
```

```
    process
    {
        if ($mypipe."Minimum Reorder Quantity" -eq [DBNull]::Value)
        { $mypipe."Minimum Reorder Quantity" = 0 }

       Return $mypipe
    }
}
```

This function is simple. However, I want to call your attention to where the value of the column "Minimum Reorder Quantity" is compared to [DBNull]::Value. That comparison will test for a database null value. Recall that $null is not really a null, but rather a default value based on the data type, so it will not work if we use it.

Now that we have the destination table defined and have reviewed the transformation function, let's look at the function that will load the products table, Invoke-UdfProductLoad, available in module umd_northwind_etl. We'll start by looking at the entire products table load function and then we'll review it in detail:

```
Import-Module umd_etl_functions -Force # uses this module
Import-Module umd_database -Force

function Invoke-UdfProductLoad
{
<#  Define the mapping... #>
$mappingset = @()  # Create a collection object.
$mappingset += (Add-UdfMapping "[Supplier IDs]" "SupplierIDs" "'" $false)
$mappingset += (Add-UdfMapping "[ID]" "ID" "" $true)
$mappingset += (Add-UdfMapping "[Product Code]" "ProductCode" "'" $false)
$mappingset += (Add-UdfMapping "[Product Name]" "ProductName" "'" $false)
$mappingset += (Add-UdfMapping "[Description]" "Description" "'" $false)
$mappingset += (Add-UdfMapping "[Standard Cost]" "StandardCost" "" $false)
$mappingset += (Add-UdfMapping "[Reorder Level]" "ReorderLevel" "" $false)
$mappingset += (Add-UdfMapping "[Target Level]" "TargetLevel" "" $false)
$mappingset += (Add-UdfMapping "[List Price]" "ListPrice" "" $false)
$mappingset += (Add-UdfMapping "[Quantity Per Unit]" "QuantityPerUnit" "'" $false)
$mappingset += (Add-UdfMapping "[Discontinued]" "Discontinued" "'" $false)
$mappingset += (Add-UdfMapping `
              "[Minimum Reorder Quantity]" "MinimumReorderQuantity" "" $false)
$mappingset += (Add-UdfMapping "[Category]" "Category" "'" $false)
$mappingset += (Add-UdfMapping "[Attachments]" "Attachments" "'" $false)

$global:referencepath = $env:HomeDrive + $env:HOMEPATH + "\Documents\"

<# Create the Access database connection object #>
[psobject] $access1 = New-Object psobject
New-UdfConnection ([ref]$access1)
$access1.ConnectionType = 'ODBC'
$access1.DatabaseType = 'Access'
$access1.DatabaseName = ($global:referencepath + 'DesktopNorthwind2007.accdb')
$access1.UseCredential = 'N'
$access1.SetAuthenticationType('DSNLess')
$access1.Driver = "Microsoft Access Driver (*.mdb, *.accdb)"
$access1.BuildConnectionString()
<# Create the SQL Server connection object #>
[psobject] $SqlServer1 = New-Object psobject
New-UdfConnection ([ref]$SqlServer1)

$SqlServer1.ConnectionType = 'ADO'
$SqlServer1.DatabaseType = 'SqlServer'
$SqlServer1.Server = '(local)'
$SqlServer1.DatabaseName = 'Development'
$SqlServer1.UseCredential = 'N'
$SqlServer1.SetAuthenticationType('Integrated')
$SqlServer1.BuildConnectionString()

$access1.RunSQL("select * from Products order by id", $true) |
Invoke-UdfProductsTransformation   `
| Invoke-UdfSQLDML -Mapping $mappingset  -DmlOperation "Merge" -Destinationtablename "[dbo].
Products" -connection $SqlServer1 -Verbose

}
```

The first step is to define the mapping as shown here:

```
function Invoke-UdfProductLoad
{
<#  Define the mapping... #>
$mappingset = @()  # Create a collection object.
$mappingset += (Add-UdfMapping "[Supplier IDs]" "SupplierIDs" "'" $false)
$mappingset += (Add-UdfMapping "[ID]" "ID" "" $true)
$mappingset += (Add-UdfMapping "[Product Code]" "ProductCode" "'" $false)
$mappingset += (Add-UdfMapping "[Product Name]" "ProductName" "'" $false)
$mappingset += (Add-UdfMapping "[Description]" "Description" "'" $false)
$mappingset += (Add-UdfMapping "[Standard Cost]" "StandardCost" "" $false)
$mappingset += (Add-UdfMapping "[Reorder Level]" "ReorderLevel" "" $false)
$mappingset += (Add-UdfMapping "[Target Level]" "TargetLevel" "" $false)
$mappingset += (Add-UdfMapping "[List Price]" "ListPrice" "" $false)
$mappingset += (Add-UdfMapping "[Quantity Per Unit]" "QuantityPerUnit" "'" $false)
$mappingset += (Add-UdfMapping "[Discontinued]" "Discontinued" "'" $false)
$mappingset += (Add-UdfMapping `
               "[Minimum Reorder Quantity]" "MinimumReorderQuantity" "" $false)
$mappingset += (Add-UdfMapping "[Category]" "Category" "'" $false)
$mappingset += (Add-UdfMapping "[Attachments]" "Attachments" "'" $false)
```

Notice that the source columns are enclosed in brackets. This is necessary when querying the columns because the column names have spaces in them. As we saw earlier, when we want to access the columns in the pipeline, we need to take out the brackets. My preference is to never include spaces in object names. It just makes it more difficult to work with them. Note: If you select the lines above in the ISE and execute them, `$mappingset` will be loaded; you can use the following statement to see it:

```
$mappingset
```

We would see the following output:

```
InColumn                   OutColumn               OutColumnDelimiter  IsKey
--------                   ---------               ------------------  -----
[Supplier IDs              SupplierIDs             '                   False
[ID]                       ID                                          True
[Product Code]             ProductCode             '                   False
[Product Name]             ProductName             '                   False
[Description]              Description             '                   False
[Standard Cost]            StandardCost                                False
[Reorder Level]            ReorderLevel                                False
[Target Level]             TargetLevel                                 False
[List Price]               ListPrice                                   False
[Quantity Per Unit]        QuantityPerUni          '                   False
[Discontinued]             Discontinued            '                   False
[Minimum Reorder Quantity] MinimumReorderQuantity                      False
[Category]                 Category                '                   False
[Attachments]              Attachments             '                   False
```

We can see the values for the properties source column name (InColumn), target column name (OutColumn), SQL statement value delimiter (OutColumnDelimiter), and IsKey.

To extract the data from the Access table, we'll need to create a connection object which the statements below, copied from the load function, do. You will need to have Microsoft Access and the Access driver for your version of Access installed to run the code. Adjustments to the Driver property below may also be needed.

```
$global:referencepath = $env:HomeDrive + $env:HOMEPATH + "\Documents\"

[psobject] $access1 = New-Object psobject
New-UdfConnection ([ref]$access1)
$access1.ConnectionType = 'ODBC'
$access1.DatabaseType = 'Access'
$access1.DatabaseName = ($global:referencepath + 'DesktopNorthwind2007.accdb')
$access1.UseCredential = 'N'
$access1.SetAuthenticationType('DSNLess')
$access1.Driver = "Microsoft Access Driver (*.mdb, *.accdb)"
$access1.BuildConnectionString()
```

The first line assigns the path to the Access database file to the global variable $global:referencepath. In a real implementation, configuration variables like this would be better to assign values in the PowerShell profile script, as we covered in Chapter 10. The rest of the lines just assign the connection properties needed for the code to build the connection string.

Next, the function needs to create the destination connection object:

```
<# Create the SQL Server connection object #>
[psobject] $SqlServer1 = New-Object psobject
New-UdfConnection ([ref]$SqlServer1)

$SqlServer1.ConnectionType = 'ADO'
$SqlServer1.DatabaseType = 'SqlServer'
$SqlServer1.Server = '(local)'
$SqlServer1.DatabaseName = 'Development'
```

```
$SqlServer1.UseCredential = 'N'
$SqlServer1.SetAuthenticationType('Integrated')
$SqlServer1.BuildConnectionString()
```

A psobject variable named $SqlServer1 is created and passed into New-UdfConnection, where our custom methods and properties are added. Then, the connection properties are assigned and the method BuildConnectionString is called to generate the connection string.

Now the function is ready to load the data, which it does with the statements here:

```
$access1.RunSQL("select * from Products order by id", $true) | Invoke-
UdfProductsTransformation   `
| Invoke-UdfSQLDML -Mapping $mappingset  -DmlOperation "Merge" -Destinationtablename "[dbo].
Products" -connection $SqlServer1 –Verbose
```

The first statement uses the $access1 custom connection object to get the source data from the Access Products table. Note that the second parameter, $true, indicates that the query does return data. The results are piped into the transformation function Invoke-UdfProductsTransformation, which in turn pipes the data into Invoke-UdfSQLDML; this creates a SQL Merge statement for each row in the source and executes the statement. You should see merge statements generated and displayed to the console that are similar to the ones below, i.e. because we used the Verbose parameter:

```
VERBOSE:
   Merge into [dbo].Products as Target USING (VALUES ( '6',99,'NWTSO-99','Northwind Traders
Chicken Soup','',1
.0000,100,200,1.9500,'','False',0,'Soups','' )) as Source ( [Supplier IDs], [ID], [Product
Code], [Product Name], [Descrip
tion], [Standard Cost], [Reorder Level], [Target Level], [List Price], [Quantity Per Unit],
[Discontinued], [Minimum Reord
er Quantity], [Category], [Attachments]
         ) ON Target.ID = Source.[ID]
           WHEN MATCHED THEN
                UPDATE SET SupplierIDs = Source.[Supplier IDs],ID = Source.[ID],ProductCode
= Source.[Product Code],Produc
tName = Source.[Product Name],Description = Source.[Description],StandardCost = Source.
[Standard Cost],ReorderLevel = Sour
ce.[Reorder Level],TargetLevel = Source.[Target Level],ListPrice = Source.[List
Price],QuantityPerUnit = Source.[Quantity
Per Unit],Discontinued = Source.[Discontinued],MinimumReorderQuantity = Source.[Minimum
Reorder Quantity],Category = Sourc
e.[Category],Attachments = Source.[Attachments]
           WHEN NOT MATCHED BY TARGET THEN insert (SupplierIDs, ID, ProductCode,
ProductName, Description, StandardCost, R
eorderLevel, TargetLevel, ListPrice, QuantityPerUnit, Discontinued, MinimumReorderQuantity,
Category, Attachments) values
('6',99,'NWTSO-99','Northwind Traders Chicken Soup','',1.0000,100,200,1.9500,'','False',0,'
Soups','');
```

The pattern used to load products is the same as for any other load. Define the mappings. Create the source connection. Create the target connection. Pipe the source data through the transformation function and into the function that writes the data to the destination. Now, let's look at the order table load.

The Order Table Load

The next table we want to load is the orders table. The SQL script in Listing 11-5 will create the table on SQL Server so we can load it. Note: The function name of the final load in `umd_northwind_etl` is `Invoke-UdfOrderLoad`.

Listing 11-5. Create the orders table

```
Create table dbo.Orders
(
Order_ID      integer primary key,
Employee_ID   integer,
Order_Date    datetime,
Ship_City     varchar(150),
Ship_State    varchar(100),
Status_ID     integer,
Customer_ID   integer,
Sales_Tax_Rate decimal(8,2)
)
```

The table in Listing 11-5 is a reduced column set from the source that provides an opportunity to discuss some more transformation techniques. For this transformation, instead of modifying the incoming pipeline, we're going to replace it with a new custom pipeline. The Northwind `Orders` table has a tax rate column, but it is not populated, so we're going to look it up in the transformation and load it to the destination table.

To demonstrate how to load an XML file, the orders table was exported as XML. The statement that follows will load the XML file into the *$orders* object:

```
[xml]$orders = Get-Content ("$global:referencepath" + "Orders.XML" )
```

Notice the cast operator `[xml]` before `$orders`. When `Get-Content` loads the file, this tells PowerShell to format it as an XML document. We can easily access the properties by using the `dataroot` property, as shown here:

```
$orders.dataroot.orders
```

This statement should list the orders data to the console, as shown here:

```
Order_x0020_ID                       : 81
Employee_x0020_ID                    : 2
Customer_x0020_ID                    : 3
Order_x0020_Date                     : 2006-04-25T17:26:53
Ship_x0020_Name                      : Thomas Axen
Ship_x0020_Address                   : 123 3rd Street
Ship_x0020_City                      : Los Angelas
Ship_x0020_State_x002F_Province      : CA
Ship_x0020_ZIP_x002F_Postal_x0020_Code : 99999
Ship_x0020_Country_x002F_Region      : USA
Shipping_x0020_Fee                   : 0
Taxes                                : 0
Tax_x0020_Rate                       : 0
Status_x0020_ID                      : 0
```

Notice the characters 'x0020' interspersed into the column names. This is because the source table has spaces in the column names, which get replaced with 'x0020' when the table is exported. We could just do a quick global replace to fix this, but it provides a good opportunity to discuss how to rename pipeline properties. In fact, what we will do is use the order file pipeline as input to build a custom pipeline that will be loaded into SQL Server. Let's look at the transformation function next:

```
function Invoke-UdfOrdersTransformation
{
 [CmdletBinding()]
        param (
                 [Parameter(ValueFromPipeline=$True)]$pipein = "default",
                 [Parameter(ValueFromPipeline=$false)]$filepath
               )
    process
    {

      $statetaxcsv = Import-CSV ("$filepath" + "StateSalesTaxRates.csv" )
      [hashtable] $statetaxht = @{}
      foreach ($item in $statetaxcsv)
      {
          $statetaxht.Add($item.StateCode, $item.SalesTaxRate)
      }

      [psobject] $pipeout = New-Object psobject

      $pipeout | Add-Member -MemberType NoteProperty -Name "Employee_ID"    `
                            -Value $pipein.Employee_x0020_ID
      $pipeout | Add-Member -MemberType NoteProperty -Name "Order_Date"     `
                            -Value $pipein.Order_x0020_Date
      $pipeout | Add-Member -MemberType NoteProperty -Name "Ship_City"      `
                            -Value $pipein.Ship_x0020_City
      $pipeout | Add-Member -MemberType NoteProperty -Name "Ship_State"     `
                            -Value $pipein.Ship_x0020_State_x0020_Province
      $pipeout | Add-Member -MemberType NoteProperty -Name "Status_ID"      `
                            -Value $pipein.Status_x0020_ID
      $pipeout | Add-Member -MemberType NoteProperty -Name "Order_ID"       `
                            -Value $pipein.Order_x0020_ID
      $pipeout | Add-Member -MemberType NoteProperty -Name "Customer_ID"    `
                            -Value $pipein.Customer_x0020_ID

      $pipeout | Add-Member -MemberType NoteProperty -Name "Sales_Tax_Rate" `
                            -Value 0

      $pipeout.Sales_Tax_Rate = $statetaxht[$pipein.Ship_x0020_State_x002F_Province]

      Return $pipeout
    }
}
```

This function takes the pipeline, and a file path as input. The file path is the location of the state sales tax file. Notice that we take the pipeline in as `$pipein`, but the last line returns `$pipeout`. The original pipeline is replaced with a custom one we will create. To support the look-up of the state sales tax rate, we will load the state sales tax file into a hash table, as shown:

```
$statetaxcsv = Import-CSV ("$filepath" + "StateSalesTaxRates.csv")

[hashtable] $statetaxht = @{}

foreach ($item in $statetaxcsv)
     {
         $statetaxht.Add($item.StateCode, $item.SalesTaxRate)
     }
```

The first line loads the file into variable `$statetaxcsv`. The statement after that creates an empty hash table named `$statetaxht`. Then, the `ForEach` loop loads the key/value pairs using the hash table `Add` method.

Next, the line copied here creates a new `psobject` to hold the new pipeline:

```
[psobject] $pipeout = New-Object psobject
```

Once `$pipeout` is created, the function uses the `Add-Member` cmdlet to add note properties to `$pipeout` and assigns them values from the incoming pipeline. The last `NoteProperty`, `Sales_Tax_Rate`, is initialized to zero because we are required to provide a value. However, this value is replaced by the statement that follows:

```
$pipeout.Sales_Tax_Rate = $statetaxht[$pipein.Ship_x0020_State_x002F_Province]
```

Recall that hash tables return the value associated with the key when you request the key, so this line is getting the sales tax rate from the `$pipeout` note property `Sales_Tax_Rate`. The last line simply returns the new pipeline to the caller.

The statements that follow will test this transformation:

```
[xml]$orders = Get-Content ("$global:referencepath" + "Orders.XML" )
$orders.dataroot.orders | Invoke-UdfOrdersTransformation -filepath $global:referencepath
```

The first line loads the sales tax file into `$orders`. The second line pipes this into the transformation function `Invoke-UdfOrdersTransformation`, passing the path to the state tax rate table as a parameter.
A sample of the output to the console is shown here:

```
Employee_ID    : 2
Order_Date     : 2006-04-25T17:26:53
Ship_City      : Los Angelas
Ship_State     :
Status_ID      : 0
Order_ID       : 81
Customer_ID    : 3
Sales_Tax_Rate : 0.08
```

In this output, we can see the custom properties we created, including the new `Sales_Tax_Rate` property. For more ideas about ways pipeline properties can be renamed, consult this interesting blog by Don Jones:

https://technet.microsoft.com/en-us/magazine/ff394367.aspx

Now, let's look at the function with which to load the order table:

```
function Invoke-UdfOrderLoad
{
<#  Define the MS Access Connection - Caution:  32 bit ISE only  #>
$global:referencepath = $env:HomeDrive + $env:HOMEPATH + "\Documents\"

<# Load orders #>
[xml]$orders = Get-Content ("$global:referencepath" + "Orders.XML" )

<#  Define the mapping... #>
$mappingset = @()  # Create a collection object.
$mappingset += (Add-UdfMapping "Order_ID" "Order_ID" "" $true)
$mappingset += (Add-UdfMapping "Employee_ID" "Employee_ID" "" $false)
$mappingset += (Add-UdfMapping "Order_Date" "Order_Date" "'" $false)
$mappingset += (Add-UdfMapping "Ship_City" "Ship_City" "'" $false)
$mappingset += (Add-UdfMapping "Ship_State" "Ship_State" "'" $false)
$mappingset += (Add-UdfMapping "Status_ID" "Status_ID" "" $false)
$mappingset += (Add-UdfMapping "Customer_ID" "Customer_ID" "" $false)
$mappingset += (Add-UdfMapping "Sales_Tax_Rate" "Sales_Tax_Rate" "" $false)

<#  Define the SQL Server Target  #>
[psobject] $SqlServer1 = New-Object psobject
New-UdfConnection ([ref]$SqlServer1)

$SqlServer1.ConnectionType = 'ADO'
$SqlServer1.DatabaseType = 'SqlServer'
$SqlServer1.Server = '(local)'
$SqlServer1.DatabaseName = 'Development'
$SqlServer1.UseCredential = 'N'
$SqlServer1.SetAuthenticationType('Integrated')
$SqlServer1.BuildConnectionString()

$orders.dataroot.orders | Invoke-UdfOrdersTransformation -filepath $global:referencepath   `
| Invoke-UdfSQLDML -Mapping $mappingset  -DmlOperation "Merge" -Destinationtablename
"dbo.Orders" -connection $SqlServer1 -Verbose

}
```

The following line defines the path to the orders XML file:

```
$global:referencepath = $env:HomeDrive + $env:HOMEPATH + "\Documents\"
```

Then, we load the orders file into a variable named `$orders`. Notice that we cast the variable as `[xml]`, which makes PowerShell load the file into a XML object. See here:

```
[xml]$orders = Get-Content ("$global:referencepath" + "Orders.XML" )
```

Since we want to use a custom pipe to load the table, we'll create the mapping and use it as the source, as shown here:

```
$mappingset = @()  # Create a collection object.
$mappingset += (Add-UdfMapping "Order_ID" "Order_ID" "" $true)
$mappingset += (Add-UdfMapping "Employee_ID" "Employee_ID" "'" $false)
$mappingset += (Add-UdfMapping "Order_Date" "Order_Date" "" $false)
$mappingset += (Add-UdfMapping "Ship_City" "Ship_City" "" $false)
$mappingset += (Add-UdfMapping "Ship_State" "Ship_State" "" $false)
$mappingset += (Add-UdfMapping "Status_ID" "Status_ID" "" $false)
$mappingset += (Add-UdfMapping "Customer_ID" "Customer_ID" "" $false)
$mappingset += (Add-UdfMapping "Sales_Tax_Rate" " Sales_Tax_Rate" "" $false)
```

Since we created a custom pipeline, it makes sense that we make the source column names match the destination table column names. Notice we flag the `Order_ID` as the key column. The lines that follow create a database connection object using the `New-UdfConnection` function in `umd_database`:

```
[psobject] $SqlServer1 = New-Object psobject
New-UdfConnection ([ref]$SqlServer1)
```

Now, we just set the target database connection attributes:

```
$SqlServer1.ConnectionType = 'ADO'
$SqlServer1.DatabaseType = 'SqlServer'
$SqlServer1.Server = '(local)'
$SqlServer1.DatabaseName = 'Development'
$SqlServer1.UseCredential = 'N'
$SqlServer1.SetAuthenticationType('Integrated')
$SqlServer1.BuildConnectionString()
```

Finally, the table is loaded with the statements that follow:

```
$orders.dataroot.orders |
Invoke-UdfOrdersTransformation -filepath $global:referencepath |
Invoke-UdfSQLDML -Mapping $mappingset  -DmlOperation "Merge"  `
-Destinationtablename "dbo.Orders" -connection $SqlServer1 -Verbose
```

This code is actually one line formatted for readability using the line continuation character. It starts by piping the XML-formatted data in `$orders.dataroot.orders` to the transformation function `Invoke-UdfOrdersTransformation`, which does the transformations on the pipeline. It passes the path to the state tax rate file, because the transformation needs to load that file to get the sales tax rates. The transformation pipes the data to the function `Invoke-UdfSQLDML`, which loads the data. The call to `Invoke-UdfSQLDML` includes the mapping collection, the type of SQL statement to generate, the destination table name, and the custom connection object.

We see the same pattern to load orders as we do for any other load. The mappings are defined. The source connection, in this case an XML file, is created. The target connection is created. The source data is piped through the transformation function and into the function that writes the data to the destination.

The Customers Table

The last step is to load the customers table. This is sourced as an extract from the DestopNorthwind2007 sample database. Use the script in Listing 11-6 to create the table on SQL Server. Note: The function name of the final load in `umd_northwind_etl` is `Invoke-UdfCustomerLoad`.

Listing 11-6. Create customers table

```
CREATE TABLE [dbo].[Customers](
        [Customer_ID] [int] NOT NULL,
        [Company_Name] [varchar](50) NULL,
        [First_Name] [varchar](50) NULL,
        [Last_Name] [varchar](50) NULL,
        [Title] [varchar](50) NULL,
        [City] [varchar](50) NULL,
        [State] [varchar](50) NULL,
        [Zip] [varchar](15) NULL,
        [Country] [varchar](50) NULL,
        [State_Name] varchar(255) null,
        [State_Region] varchar(255) null
PRIMARY KEY CLUSTERED
(
        [Customer_ID] ASC
) )
```

This load has a couple of twists that will demonstrate the extensibility of PowerShell. First, the main input file, customers, is a fixed-length flat file. Second, we're going to join the input customer file with the `State_Table.csv` file we used earlier in the book to translate the state code into the state name and retrieve the region name. We'll begin by putting the fixed-length flat file though a transformation function that extracts the columns and puts them into pipeline properties. There are several ways to do this, and some are less verbose than the method we use here. I like this method because it is very easy to understand and therefore to maintain. It uses the `substring` method to parse out the columns. Let's look at the code:

```
function Invoke-UdfCustomersTransformation
{
 [CmdletBinding()]
        param (
                 [Parameter(ValueFromPipeline=$True)]$pipein = "default"
               )
    process
    {

      [psobject] $pipeout = New-Object psobject

      $pipeout | Add-Member -MemberType NoteProperty -Name "Customer_ID"    `
                            -Value $pipein.substring(0,10).trim()

      $pipeout | Add-Member -MemberType NoteProperty -Name "Company_Name"    `
                            -Value $pipein.substring(11,50).trim()

      $pipeout | Add-Member -MemberType NoteProperty -Name "Last_Name"    `
                            -Value $pipein.substring(61,50).trim()
        $pipeout | Add-Member -MemberType NoteProperty -Name "First_Name"    `
                              -Value $pipein.substring(111,50).trim()

        $pipeout | Add-Member -MemberType NoteProperty -Name "Title"    `
                              -Value $pipein.substring(211,50).trim()

        $pipeout | Add-Member -MemberType NoteProperty -Name "City"    `
                              -Value $pipein.substring(873,50).trim()

        $pipeout | Add-Member -MemberType NoteProperty -Name "State"    `
                              -Value $pipein.substring(923,50).trim()

        $pipeout | Add-Member -MemberType NoteProperty -Name "Zip"    `
                              -Value $pipein.substring(973,15).trim()

        $pipeout | Add-Member -MemberType NoteProperty -Name "Country"    `
                              -Value $pipein.substring(988,50).trim()
        Return $pipeout

    }
}
```

As we've seen in the other examples, the function Invoke-UdfCustomersTransformation takes the pipeline as a parameter. The first statement in the function creates a new instance of a psobject, which is to be used to create a custom pipeline that will be passed back to the caller. Then, there are a series of Add-Member statements that create a property to hold each column. Each column is extracted using the substring function. Remember, since this is a fixed-length file, the pipeline has just one property, which contains a row from the file. We use the substring method to extract the column based on where it occurs in the line. Admittedly, writing substring method calls can be a bit tedious, but once it has been done, the columns will be defined into separate properties in the pipeline that is returned. Then, it can be used in the same way as any other data source.

Now that we have reviewed the transformation function, let's look at the function that uses it to load the Customers table:

```
function Invoke-UdfCustomerLoad
{
   <#  Define the MS Access Connection - Caution:  32 bit ISE only  #>
   $global:referencepath = $env:HomeDrive + $env:HOMEPATH + "\Documents\"

   #  Load the tables into variables...
   $customer = Get-Content ("$global:referencepath" + "customers.txt" ) |
   Invoke-UdfCustomersTransformation | `
               Sort-Object -Property State

   $states = Import-CSV ("$global:referencepath" + "state_table.csv" ) |
   Sort-Object -Property abbreviation

   <#  Define the mapping and include the columns coming from the joined table... #>
   $mappingset = @()  # Create a collection object.
   $mappingset += (Add-UdfMapping "Customer_ID" "Customer_ID" "" $true)
   $mappingset += (Add-UdfMapping "Company_Name" "Company_Name" "'" $false)
```

```
    $mappingset += (Add-UdfMapping "Last_Name" "Last_Name" "'" $false)
    $mappingset += (Add-UdfMapping "First_Name" "First_Name" "'" $false)
    $mappingset += (Add-UdfMapping "Title" "Title" "'" $false)
    $mappingset += (Add-UdfMapping "City" "City" "'" $false)
    $mappingset += (Add-UdfMapping "State" "State" "'" $false)
    $mappingset += (Add-UdfMapping "Zip" "Zip" "" $false)
    $mappingset += (Add-UdfMapping "Country" "Country" "'" $false)
    $mappingset += (Add-UdfMapping "name" "State_Name" "'" $false)
    $mappingset += `
    (Add-UdfMapping "census_region_name" "State_Region" "'" $false)

    <#  Define the SQL Server Target  #>
    [psobject] $SqlServer1 = New-Object psobject
    New-UdfConnection ([ref]$SqlServer1)

    $SqlServer1.ConnectionType = 'ADO'
    $SqlServer1.DatabaseType = 'SqlServer'
    $SqlServer1.Server = '(local)'
    $SqlServer1.DatabaseName = 'Development'
    $SqlServer1.UseCredential = 'N'
    $SqlServer1.SetAuthenticationType('Integrated')
    $SqlServer1.BuildConnectionString()

    <# Load the data. #>
    $SqlServer1.RunSQL("truncate table [dbo].[Customers]", $false)

    Join-Object -Left $customer -Right $states `
                -Where {$args[0].State -eq $args[1].abbreviation} -LeftProperties "*"    `
                -RightProperties "name","census_region_name" -Type AllInLeft   |
    Invoke-UdfSQLDML -Mapping $mappingset  -DmlOperation `
    "Insert" -Destinationtablename "dbo.Customers" -connection $SqlServer1 -Verbose

}
```

The first thing the function does is set the reference to the file folder:

```
function Invoke-UdfCustomerLoad
{
<#  Define the data path global variable. #>
$global:referencepath = $env:HomeDrive + $env:HOMEPATH + "\Documents\"
```

The file is loaded using `Get-Content` piped into `Invoke-UdfCustomersTransformation` and then sorted by the `State` with `Sort-Object`:

```
#  Load the tables into variables...
$customer = Get-Content ("$global:referencepath" + "customers.txt" ) | `
Invoke-UdfCustomersTransformation | `
            Sort-Object -Property State
```

The final result is loaded into the variable $customer. We need to sort the object in the order of the key we want to join on, i.e., State. Next, we load the state_table.csv file into a variable, as shown here:

```
$states = Import-CSV ("$global:referencepath" + "state_table.csv" ) |
Sort-Object -Property abbreviation
```

Notice we use Import-CSV to load the state_table.csv file into the pipeline. The first row of the file defines the pipeline property names. This is piped into the Sort-Object cmdlet to sort the pipeline by the state code, which is named abbreviation. The final result is stored in $states. Now, let's look at the mapping-set creation statements:

```
<#  Define the mapping and include the columns coming from the joined table... #>
$mappingset = @()  # Create a collection object.
$mappingset += (Add-UdfMapping "Customer_ID" "Customer_ID" "" $true)
$mappingset += (Add-UdfMapping "Company_Name" "Company_Name" "'" $false)
$mappingset += (Add-UdfMapping "Last_Name" "Last_Name" "'" $false)
$mappingset += (Add-UdfMapping "First_Name" "First_Name" "'" $false)
$mappingset += (Add-UdfMapping "Title" "Title" "'" $false)
$mappingset += (Add-UdfMapping "City" "City" "'" $false)
$mappingset += (Add-UdfMapping "State" "State" "'" $false)
$mappingset += (Add-UdfMapping "Zip" "Zip" "" $false)
$mappingset += (Add-UdfMapping "Country" "Country" "'" $false)
$mappingset += (Add-UdfMapping "name" "State_Name" "'" $false)
$mappingset += (Add-UdfMapping "census_region_name" "State_Region" "'" $false)
```

Because we use Add-UdfMapping to create a mapping collection, we have a consistent, repeatable way to define the column mappings regardless of the source or destination of the data. As we've seen previously, these statements create and load the mapping-set collection. Caution: Make sure you carefully review the value delimiter, i.e., the third parameter. This is what is used to generate the SQL statements and, if any are wrong, the SQL statement will fail. We create the target database connection object as shown here:

```
<#  Define the SQL Server Target  #>
[psobject] $SqlServer1 = New-Object psobject
New-UdfConnection ([ref]$SqlServer1)

$SqlServer1.ConnectionType = 'ADO'
$SqlServer1.DatabaseType = 'SqlServer'
$SqlServer1.Server = '(local)'
$SqlServer1.DatabaseName = 'Development'
$SqlServer1.UseCredential = 'N'
$SqlServer1.SetAuthenticationType('Integrated')
$SqlServer1.BuildConnectionString()
```

Since we are loading the data into a staging table, we need to clear it out before loading it:

```
<# Load the data. #>
   $SqlServer1.RunSQL("truncate table [dbo].[Customers]", $false)
```

Now we come to joining the data. I found a great Join-Object function that was developed by the Microsoft PowerShell team that supports inner and outer joins. It provides excellent functionality. Full details about it are available at the link here:

http://blogs.msdn.com/b/powershell/archive/2012/07/13/join-object.aspx

There is mention on the blog entry about this function possibly being added as a standard PowerShell cmdlet. For the time being, a good way to make it easy to use is to save it as a module so we can import it when needed. I named the module `umd_join_object`, keeping with my naming convention, but I will leave the function name as it was defined by Microsoft. However, if the function is ever added to PowerShell as a built-in cmdlet, the import of the custom module `umd_join_object` should be removed wherever it is used and replaced with the new cmdlet. Let's look at the code that joins the data and pipes the output to the function `Invoke-UdfSQLDML` in order to be written to the SQL Server table:

```
Join-Object -Left $customer -Right $states `
            -Where {$args[0].State -eq $args[1].abbreviation} -LeftProperties "*"   `
            -RightProperties "name","census_region_name" -Type AllInLeft |
Invoke-UdfSQLDML -Mapping $mappingset  -DmlOperation "Insert" -Destinationtablename
"dbo.Customers" -connection $SqlServer1 -Verbose
```

We call the `Join-Object` function, passing the `Left` parameter, i.e., the data that will be the left side of the join as `$customer`, and the `Right` parameter, i.e., the right side of the join as `$states`. The `Where` parameter provides the join criteria, which is to join the `State` property on `$customer` with the abbreviation property on `$states`. The `-eq` operand tests for a match. The `LeftProperties` is where we provide a list of what properties from the left-side source we want included in the output. The asterisk signals to include all the left-side properties. The `RightProperties` is where we provide a list of what properties from the right-side source we want included in the output. Here, we just want the state name, i.e., `name`, and `census_region_name`. If we did an inner join, rows where no match was found will be dropped. Since we want all customers returned, we use a left join, which is what the `Type` parameter, `AllInLeft`, is requesting. The result is piped into `Invoke-UdfSQLDML`, which loads the data into the `Customers` table. Notice we included mappings for the retrieved columns from `$state`. The `name` column gets mapped to `State_Name`, and the `census_region_name` gets mapped to `State_Region`. Note: We created the `$SqlServer1` connection earlier in the chapter.

The pattern to load customers is almost the same as that for the other loads, even though we have the added complexity of a join. The only difference is that we had to define two sources and join them before piping the results to the destination. So, the steps in this pattern are: define the mappings, create the source connections, create the target connection, and pipe the joined source data through the transformation function and into the function that writes the data to the destination. An option in any load is to load the source data into a variable and then pipe the variable into the destination, versus piping the data directly from the source to the destination.

Packaging Up the ETL Job

We packaged up the ETL scripts into simple functions and put them into the `umd_northwind_etl` module, which you can copy to your modules folder under a subdirectory of the same name. Once that has been done, you can execute the entire ETL process with the statements in Listing 11-7. Note: This is assuming you have modified all relevant configuration settings to match your environment.

Listing 11-7. The Northwind ETL script

```
Import-Module umd_northwind_etl

Invoke-UdfStateSalesTaxLoad

Invoke-UdfOrderLoad

Invoke-UdfProductLoad

Invoke-UdfCustomerLoad
```

Summary

In this chapter, we discussed using PowerShell as an ETL tool versus using SSIS. We started by comparing the merits of PowerShell and SSIS. Then, we considered the general pattern of an ETL tool—extract from a source, transform the data, and write the data to a destination. To help us cover a range of sources and destinations, we used a fictitious ETL use case that required a number of files to be loaded into SQL Server tables. To get the most value out of PowerShell as an ETL tool, we created a reusable framework that makes it easy to write the code and minimizes data-specific coding. We discussed how the PowerShell pipeline can be used as a data flow. At the center of our PowerShell ETL framework is a mapping collection. In prior chapters we built reusable functions to retrieve data, so the challenge here is getting that data loaded into the destination. The mapping collection will tell the destination function what source columns map to which destination columns. By leveraging the mapping collection together with the pipeline, we explained how a reusable function could load the data into a database table with no custom coding required, i.e. just function calls. The exception to this is when we need column transformations, because these do require data-specific code. However, we showed how transformations can be isolated into a single function and coded in a manner that is easy to read and maintain. Leveraging the use case, we demonstrated how to load a number of different file formats into tables. We even showed how transformations can easily include advanced features like key/value look-ups and file joins based on keys. The latter is a function provided by the Microsoft PowerShell team and is an excellent example of how easily PowerShell can be extended.