CHAPTER 9

Augmenting ETL Processes

This chapter will cover how to use PowerShell to augment ETL development. We often need to load external files into a SQL Server database. However, these files usually need to go through some preparation before they can be loaded. Perhaps they arrive via FTP from an external source. Such files may be compressed. Perhaps they need to be scrubbed of bad values or modified to make them easier to load. After being loaded, the business may want the files archived and retained for a period of time. Before PowerShell, legacy-style batch files were often employed to do these tasks. However, batch files are cryptic, difficult to maintain, and lack support for reusability. In this chapter, we will see how PowerShell scripts can be used to accomplish these tasks. Rather than define a specific business scenario for these tasks, we consider this a common ETL pattern in which we can choose to employ the given tasks that apply. In this pattern, files arrive in a folder and copied to a local server, then are decompressed, loaded, and archived. Typically, when the job starts and ends, email notifications are sent out. Sometimes there are additional requirements. We will discuss functions that help with these tasks. Let's consider the potential ETL steps already mentioned as a template from which we can pick what we need. Figure 9-1 shows a common ETL pattern.

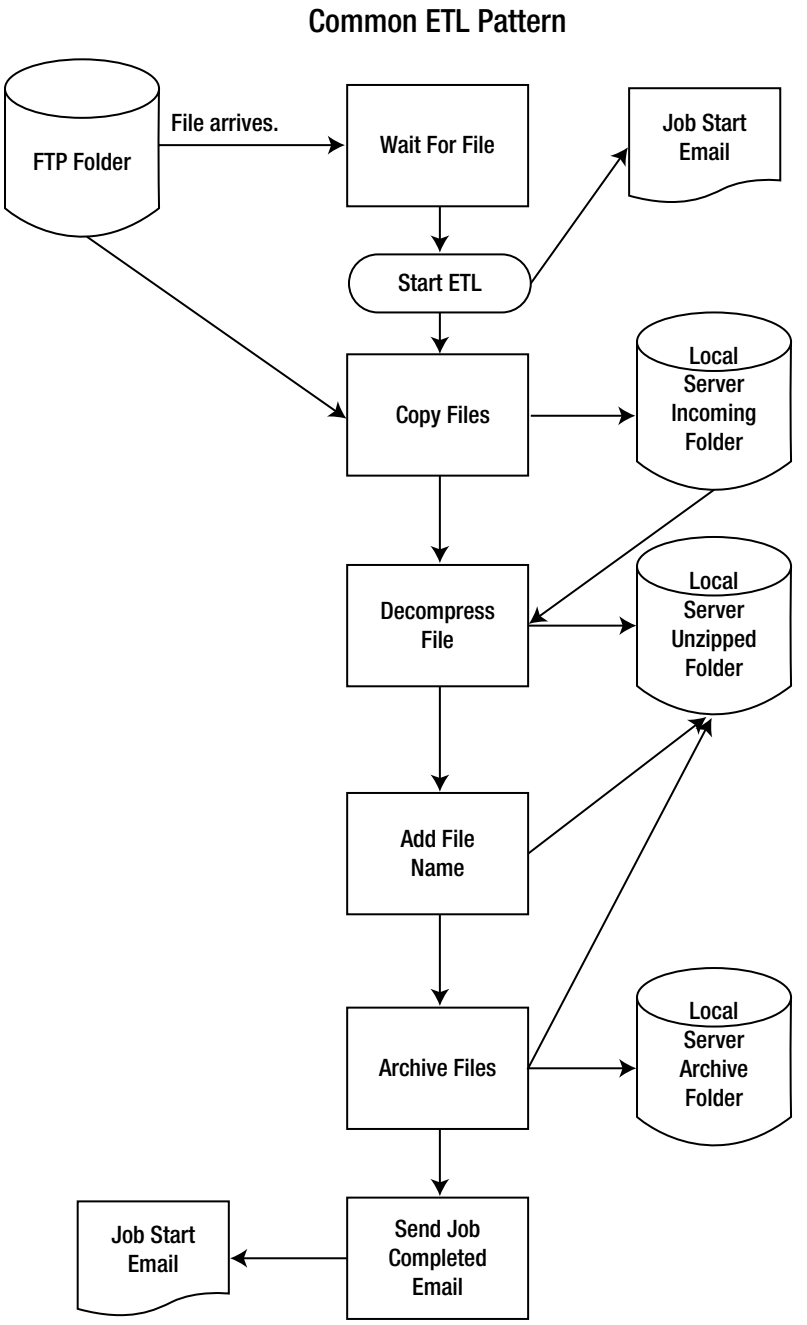

Figure 9-1. *A common ETL pattern*

We will start by reviewing functions that perform the steps in the ETL pattern: wait for a file, copy the file, decompress the file, add the file name as a column so it can be tracked, load the file into SQL Server, and send email notifications. These are presented without a context to show how they can be accomplished. Then we will review a script that orchestrates these functions into a complete ETL load process. In practice,

an SSIS package would probably load the data, but we'll use a PowerShell script to show how that can be done by PowerShell. This will be followed by additional ETL-related functions that validate data, combine files, add column headings to files, and encrypt passwords to send email securely. By the end of this chapter we will know how to use PowerShell to augment our ETL development and have a set of functions we can apply immediately.

The Functions

In this section, we will review reusable functions that prepare the files for load and then archive the files afterward. The first task in an ETL job is often to wait for the file to arrive, which we will do using the `Wait-UdfFileCreation` function. This function will suspend the ETL process until it sees that a file matching the specified name pattern has arrived. Then we need to copy the file to a local folder to be processed, which we do using `Copy-UdfFile`. We unzip the file using `Expand-UdfFile`. To make it easier to track down data problems, we use `Add-UdfColumnNameToFile` to append a new column to the file that has the name of the flat file. We load the file to a SQL Server table. Finally, we archive the file using `Move-UdfFile`. We will use `Send-UdfMail` to send start and end job notifications. We'll start by reviewing each function in detail. Then we'll assemble them into an ETL process. All the functions in this chapter are in the module `umd_etl_functions` except `Save-UdfEncryptedCredential`, which is in `umd_database`.

Wait for a File

Our ETL load process begins when a file arrives. If this occurs on a reliable schedule, then a tool like SQL Agent can be used to run the ETL process based on that schedule. However, often files are not received on a fixed schedule, and the ETL process needs to run as the files come in. In Chapter 3 we saw how to use the `FileSystemObject` to register code to execute when a new file arrives. Here, we will discuss an alternative method of monitoring for a new file. The function that follows periodically checks for a file matching a given file name pattern in the specified folder. If a match is found, the function ends; otherwise, it waits for two seconds and then tries again. This technique works well with SQL Agent. By making the wait for file function the first job step, the job will just stay in that step until a file arrives. The job schedule can be set to run periodically–say, hourly–so once the job has executed, it will automatically schedule itself again within an hour. Since SQL Agent will not queue up a job that is already running, there's no issue of getting duplicate jobs in the queue. Let's review the function:

```
function Wait-UdfFileCreation
{
 [CmdletBinding()]
        param (
              [string] $path,   # Folder to be monitored
              [string] $file    # File name pattern to wait for.
          )

   $theFile = $path + $file

   #  $theFile variable will contain the path with file name of the file we are waiting for.
   While ($true) {
       Write-Verbose $theFile
       If (Test-Path $theFile) {
           #file exists. break loop
           break
       }
```

```
       #sleep for 2 seconds, then check again - change this to fit your needs...
       Start-Sleep -s 2
}
    Write-Verbose 'File found.'
}

# Example call below...
Wait-UdfFileCreation ($env:HOMEDRIVE + $env:HOMEPATH + '\Documents\') 'filecheck.txt'
-Verbose
```

Let's review this function in detail. First, let's look at the function declaration:

```
function Wait-UdfFileCreation
{
 [CmdletBinding()]
        param (
              [string] $path,   # Folder to be monitored
              [string] $file    # File name pattern to wait for.
          )
```

Notice the `CmdletBinding` attribute, which adds a number of features including support for common parameters and parameter validation. It is good practice to use `CmdletBinding` in our functions. The function declaration declares the function `Wait-UdfFileCreation` and tells us :that it supports the string parameters `$path` and `$file`.

The line `'$theFile = $path + $file'` just concatenates the variables `$path` and `$file` and stores the result in `$theFile`.

The following code does the real work:

```
#  $theFile variable will contain the path with file name of the file we are waiting for.
While ($true)
{
       Write-Verbose $theFile
       IF (Test-Path $theFile)
       {
           #file exists. break loop
           break
       }
       #sleep for 2 seconds, then check again - change this to fit your needs...
       Start-Sleep -s 2
}
    Write-Verbose 'File found.'
}
```

The `While` loop will never end, because the comparison is the constant `$true`. We do this because we only want to exit when a file is found. The statement `'Write-Verbose $theFile'` will write the variable `$theFile` to the console if the `Verbose` parameter was passed on the function call. This is great for testing, but when we move the code to production, we need to remove the `Verbose` parameter. SQL Agent will fail a PowerShell script that tries to write to the console. The line `'IF (Test-Path $theFile)'` uses the cmdlet `Test-Path` to check whether a file matching the file name pattern exists in the folder. `Test-Path` returns `$true` if the file does exist and `$false` if it does not. Notice that since only true or false can be returned, we don't need to compare the return value to true or false–i.e., `Test-Path –eq $true`. This is because the `IF` test expression evaluates to a true or false value, so there's no need for extra verbiage. In other words,

"if ($return -eq $true)" is the same as writing "if ($return)". If the Test-Path returns true, the break statement is executed, which exits the While loop, resulting in the 'Write-Verbose 'File found.' message being displayed. Otherwise, the 'Start-Sleep 2' statement executes, which just waits for two seconds and then returns to the start of the loop again.

The wait for file function is useful in many situations and is easy to implement. As a first step in a SQL Agent job, it's the ideal way to dynamically load files as they arrive. The job is suspended until a file arrives, and then it executes the steps to load the file. The SQL Agent scheduler will automatically restart the job based on whatever time frame you need. It may seem that having a job suspended while polling for a file will take up significant resources. However, in practice, I have found the overhead not to be an issue. The duration between each file existence check should be set to something reasonable, like ten minutes. I've used this approach successfully for a number of jobs. If you have many jobs that need to start when a file arrives, you might consider a different approach, such as using the FileSystemWatcher to trap the file-creation event. This approach is discussed in Chapter 13 as part of the discussion on :work flows.

Copy File

Once a file has arrived in the FTP folder, it's a good idea to copy it to a local server folder to be processed. We could just use the Copy-Item cmdlet, but having a function allows for extensibility and for trapping errors. Let's look at the code to copy files::

```
function Copy-UdfFile {
 [CmdletBinding(SupportsShouldProcess=$true)]
        param (
            [string]$sourcepathandfile,
            [string]$targetpathandfile,
            [switch]$overwrite
          )

   $ErrorActionPreference = "Stop"
   $VerbosePreference = "Continue"

   Write-Verbose "Source: $sourcepathandfile"
   Write-Verbose "Target: $targetpathandfile"

   try
   {
    If ($overwrite)   {
     Copy-Item -Path $sourcepathandfile -Destination $targetpathandfile -Force }
    else {
     Copy-Item - Path $$sourcepathandfile -Destination $targetpathandfile }
    }
   catch
   {
      "Error moving file."
   }
}
```

As we've already seen, we use CmdletBinding to define the function parameters. Then we set $ErrorActionPreference = "Stop", telling PowerShell to treat non-terminating errors as if they were terminating. This is so we can trap these errors. The line '$VerbosePreference = "Continue"' tells

PowerShell to display any Verbose messages output by the script providers or cmdlets and continue executing the script. The Write-Verbose message just displays the source and target file path. The real work happens in the Try/Catch block:

```
   try
   {
    If ($overwrite)   {
     Copy-Item -Path $$sourcepathandfile -Destination $targetpathandfile -Force }
    else {
     Copy-Item -Path $$sourcepathandfile -Destination $targetpathandfile }
    }
   catch
   {
      "Error moving file."
   }
}
```

The code ::between the try statement and the catch statement will be executed. We use the switch parameter $overwrite to determine if any existing file should be overwritten by the copy–i.e., if $overwrite is true, the file will be overwritten because the Force parameter is passed to the Copy-Item cmdlet. Otherwise, the file is copied without the Force parameter, which will trigger an error if a file already exists with the destination file name. When an error is raised, the code in the catch block will execute.

The copy file function is simple yet useful and extensible. Depending on our needs, we can extend this function to include logging, notifications, or to meet any other requirements that come up. Best of all, as a common function, we only need to maintain it in one place.

Unzip the File

Usually external files sent via FTP are compressed to save space and reduce network traffic. There are a number of compression formats available, but for demonstration purposes we will use the Zip format. The interesting thing about this function is that it uses the Windows Explorer API to do the work. The function uses a file-name mask to get the list of files to unzip rather than just supporting a single file.

The function that follows decompresses a file that is compressed into zip format:

```
function Expand-UdfFile
{
[CmdletBinding(SupportsShouldProcess=$true)]
        param (
            [string]$sourcefile,
            [string]$destinationpath,
            [switch]$force
          )

    $shell=new-object -com shell.application
    $zipfile = $shell.NameSpace($sourcefile)

    if ($force) {$zipparm = 0x14 }  else { $zipparm = $null }
```

```
   foreach ($item in $zipfile.items())
   {
      Try
      {
         $shell.NameSpace($destinationpath).CopyHere($item, $zipparm)
      }
      Catch {
             Write-Verbose "File exists already and was overwritten."
             }
   }
}
```

In this code, the `CmdletBinding` defines the function parameters, which are the `$sourcefile` (the path to the files to be unzipped), `$destinationpath` (the path and file name of the unzipped file), and the switch `$force`, which, if passed, tells the function to replace the file if it already exists. The next line is interesting:

```
$shell=new-object -com shell.application
```

Using the `New-Object` cmdlet, we are creating an interface to Windows Explorer using the `Shell.Application` object. This exposes the functionality of Windows Explorer to our script. It seems a bit odd that PowerShell is itself an interface to Windows, yet here we create a different interface to Windows. However, Windows Explorer exposes methods and properties not readily available in PowerShell.

The line that follows sets the source path and file name for the zipped file:

```
$zipfile = $shell.NameSpace($sourcefile)
```

The next line sets whether we want to overwrite the destination files if it already exists:

```
if ($force) {$zipparm = 0x14 }  else { $zipparm = $null }
```

In this line, we are testing the `$force` switch parameter and setting the value of `$zipparm` to hexadecimal value `'0x14'` if it is true and null if it is false, i.e., switch was not passed. This value is to be passed with the unzip operation to control how the command behaves. Note: This type of parameter defines a set of flags that are based on a bit value in a given position within a byte. See the link that follows for more information on all the possible values for this parameter:

http://msdn.microsoft.com/en-us/library/windows/desktop/bb787866(v=vs.85).aspx

The next block of code loops through each file in the compressed folder, unzipping each:

```
foreach ($item in $zipfile.items())
{
   Try
   {
      $shell.NameSpace($destinationpath).CopyHere($item, $zipparm)
   }
   Catch
   {
       Write-Verbose "File exists already and was overwritten."
    }
}
```

The object `$zipfile.items()` is a collection, and we iterate through each compressed file it contains using the `foreach` statement. Each file will be placed in the `$item` variable. The `Try` statement will use the `$shell` object to unzip the file by calling the `CopyHere` method and passing the file object to unzip and the optional parameter in $zipparm. The `Catch` statement will display `"File exists already and was overwritten."` if the destination file already exists.

The unzip function showed how we can easily decompress files using the Windows Explorer API. The function supports a file-name mask, so it does not matter how many files need to be unzipped.

If your installation uses Window Server Core, which is a subset of the full Windows installation, the function will not work. Instead, you can use a .NET class available as of .NET 4.5 that will extract files. The blog link here provides an explanation on how to do this:

https://vcsjones.com/2012/11/11/unzipping-files-with-powershell-in-server-core-the-new-way/

For information on what Windows Server Core is, see the link here:

https://msdn.microsoft.com/en-us/library/hh846323%28v=vs.85%29.aspx

Add the File Name to a Flat File

Typically, files are sent on a periodic basis and from different sources. If there is a problem with the data in one of the files, it can be difficult to track down which file contains the bad data. The function that follows adds the file name as a column to the CSV file so it can be loaded along with the other data into the target SQL Server table. This is particularly useful if the file names have identifying values, such as `ABCCompany_Claims_20140101.csv`. Knowing where bad data is coming from allows you to take action, such as requesting the source entity resend the file or correct bugs in their extract process.

The code here adds the file name as a column to the CSV file:

```
function Add-UdfColumnNameToFile
{
[CmdletBinding(SupportsShouldProcess=$true)]
        param (
            [string]$sourcepath,
            [string]$targetpath,
            [string]$filter
          )

      $filelist = Get-ChildItem -Path $sourcepath -include $filter

      foreach ($file in $filelist)
      {
        $csv = get-content $file

        $start = 0

        $file_wo_comma = $file.name.Replace(',','_')
        $file_wo_comma = $file_wo_comma.Replace(' ','')

        $targetpathandfile = $targetpath +  $file_wo_comma
```

```
        foreach ($line in $csv)
           {
           if ($start -eq 0)
               {
                 $line += ",filename"
                 $line > $targetpathandfile
               }
           else
               {
                 $line +=",$file_wo_comma"
                 $line >> $targetpathandfile
               }

           $start = 1

           }
        }
}
```

Let's look at the function header:

```
function Add-UdfColumnNameToFile
{
[CmdletBinding(SupportsShouldProcess=$true)]
        param (
            [string]$sourcepath,
            [string]$targetpath,
            [string]$filter
          )
```

The `CmdletBinding` shows three string parameters: `$sourcepath` is the folder path where the source file is located, `$targetpath` is the destination path where the new file (with file name column) will be written, and `$filter` is the file name filter to match on; i.e., any files matching the pattern will be processed.

We will need to get a list of files to process, which we do with the statement here:

```
$filelist = Get-ChildItem -Path $sourcepath -Include $filter
```

We will need to iterate over the files in the list returned, which we do by using a foreach loop, as shown here:

```
foreach ($file in $filelist)
{
  $csv = Get-Content $file
```

This statement will execute for each file in `$filelist` and place the current file reference into `$file`. Then, we load the current iteration's file into `$csv` using the `Get-Content` cmdlet.

Since this is a CSV file, the first row will be column headings. Therefore, we need to handle the first row differently than we do the other rows. For the first row, we want to add a column heading for the file name. To identify which iteration is the first, we set the variable `$start = 0` before we enter the loop that processes the rows, as shown here:

```
$start = 0
```

Note: After the first row of the file has been processed, we will set `$start = 1` to signal we are done processing the first row.

File names with commas or spaces in the name can cause problems, so we'll replace the comma with an underscore and remove spaces in the target file name with the statements seen here:

```
$file_wo_comma = $file.name.Replace(',','_')
$file_wo_comma = $file_wo_comma.Replace(' ','')
```

Now we can concatenate the target file path with the scrubbed file name, as shown:

```
$targetpathandfile = $targetpath + $file_wo_comma
```

Since we want to add a new column to each row in each file, we need to iterate over the rows, which we can do with the `foreach` statement:

```
foreach ($line in $csv)
   {
```

This statement will cause each row in the file to be placed into `$line`.

The block of code that follows does the actual work of adding the new column:

```
if ($start -eq 0)
    {
      $line += ",filename"
      $line > $targetpathandfile
    }
else
    {
      $line +=",$file_wo_comma"
      $line >> $targetpathandfile
    }

$start = 1
```

If `$start = 0`, we know this is the first iteration and therefore the first row in the file. Therefore, we want to add the column name `filename`, prefixed with a comma, to the output file. The variable `$line` is written to the output file using the `>` operator, which will cause any existing file with the same name to be overwritten. If the `else` code block executes, we know this is not the first line and that we want to append the file name value to the row. (Note: `+=` means append what is to the right of the `+=` to the variable on the left.) Then, the revised row in variable `$line` is appended to the target output file using the `>>` operator. Immediately after the `If` code block, `$row` is set to 1. Since this comes after the code block, the first iteration will have a value of 0, indicating the first line.

Adding the file name to the file and loading it into the target SQL Server table is very useful in tracking down data issues. It's primarily beneficial when files come frequently or from different sources. If the files do not have distinct file names, you may want to rename them before using this function. For example, you could append the current date and, if you can identify the source, perhaps prefix an identifier like `ABCCompany`. The renamed files should be archived to a folder where you can access them later if needed.

There are other ways to accomplish the same thing. In the actual situation where I used this function, I needed to add the file name to each flat file before merging them all into one file. The ETL tool being used did not support iteration over multiple files. A single, fixed file name had to be mapped in the ETL as the input source. Having the originating file name available in the SQL Server table was helpful for troubleshooting data issues.

Move the Files

Moving a file differs from copying one in that we do not leave the original file behind. This would typically be done as a means of archiving the file in case we need it later. It may not seem necessary to write a function to move a file when the cmdlet `Move-Item` can be called directly to do this. However, wrapping the `Move-Item` statement into a function allows the function to be extended to include additional functionality, such as logging and displaying informational messages. Also, I find it is easier to remember how to use my own function since it is customized to my task. Let's look at the following function:

```
function Move-UdfFile
{
[CmdletBinding(SupportsShouldProcess=$true)]
        param (
            [string]$sourcepathandfile,
            [string]$targetpathandfile,
            [switch]$overwrite
          )

      Write-Verbose 'Move file function...'

      try
      {
        If ($overwrite)
        {
         Move-Item -PATH $sourcepathandfile -DESTINATION $targetpathandfile -Force
        }
       else
        {
         Move-Item -PATH $sourcepathandfile -DESTINATION $targetpathandfile
        }
       }
       catch
       {
         "Error moving file."
         break
       }
}
```

We use `CmdletBinding` to define the function parameters: `$sourcepathandfile` to name the object we want to move, `$targetpathandfile` to define where we want the object moved to, and `$overwrite`, which is a switch that, if passed, will cause the function to overwrite the destination object if it already exists. The `try` statement executes the code between the braces that follow. `If ($overwrite)` will execute the code in the braces that follow only if the `overwrite` switch was passed. Otherwise, the code in the braces after the `else` statement is executed. If an error is raised, the code in the `catch` block is executed, i.e., the message `"Error moving file"`. is displayed and the function is exited using the `break` statement.

The move file function copies the file to the destination and deletes it from the source, which is why it is ideal for archiving files.

Send Email

Many times an ETL process needs to notify users of events such as the loading of a file or failure of a process. PowerShell has the built-in cmdlet `Send-MailMessage` to support this. If you are sending email through a local email server behind a firewall, the call does not require authentication, but as a consultant I prefer the ability to develop and test without being logged in to a client's network. Therefore, I developed the function that follows to send email using a service provider like Google or Outlook. To do this, credentials—i.e., user ID and password—must be provided. Note: The mail will be sent using the same type of configuration information a smartphone would use. Let's look at the code:

```
function Send-UdfMail
{
[CmdletBinding(SupportsShouldProcess=$false)]
        param (
            [Parameter(Mandatory = $true, Position = 0)]
            [string]$smtpServer,
            [Parameter(Mandatory = $true, Position = 1)]
            [string]$from,
            [Parameter(Mandatory = $true, Position = 2)]
            [string]$to,
            [Parameter(Mandatory = $true, Position = 3)]
            [string]$subject,
            [Parameter(Mandatory = $true, Position = 4)]
            [string]$body,
            [Parameter(Mandatory = $true, Position = 5)]
            [string]$port,
            [Parameter(Mandatory = $false, Position = 6, ParameterSetName = "usecred")]
            [switch]$usecredential,
            [Parameter(Mandatory = $false, Position = 7, ParameterSetName = "usecred")]
            [string]$emailaccount,
            [Parameter(Mandatory = $false, Position = 8, ParameterSetName = "usecred")]
            [string]$password
            )

   if ($usecredential)
   {
     Write-Verbose "With Credential"
     $secpasswd = ConvertTo-SecureString "$password" -AsPlainText -Force
     $credential = New-Object System.Management.Automation.PSCredential
        ("$emailaccount", $secpasswd)

     Send-MailMessage -smtpServer $smtpServer -Credential $credential -from $from -to $to
     -subject $subject -Body $body -Usessl -Port $port
   }
   Else
   {
     Write-Verbose "Without Credential"
     Send-MailMessage -smtpServer $smtpServer -from $from -to $to -subject $subject -Body
     $body -Usessl -Port $port
   }
}
```

The CmdletBinding has some added parameters we have not seen yet. Let's review:

```
[CmdletBinding(SupportsShouldProcess=$false)]
       param (
           [Parameter(Mandatory = $true, Position = 0)]
           [string]$smtpServer,
           [Parameter(Mandatory = $true, Position = 1)]
           [string]$from,
           [Parameter(Mandatory = $true, Position = 2)]
           [string]$to,
           [Parameter(Mandatory = $true, Position = 3)]
           [string]$subject,
           [Parameter(Mandatory = $true, Position = 4)]
           [string]$body,
           [Parameter(Mandatory = $true, Position = 5)]
           [string]$port,
           [Parameter(Mandatory = $false, Position = 6,
            ParameterSetName = "usecred")]
           [switch]$usecredential,
           [Parameter(Mandatory = $false, Position = 7,
            ParameterSetName = "usecred")]
           [string]$emailaccount,
           [Parameter(Mandatory = $false, Position = 8,
            ParameterSetName = "usecred")]
           [string]$password
           )
```

The Mandatory parameter attribute defines whether the parameter must be provided or not. The Position parameter attribute identifies the sequence of the parameter if it is not passed by name. Normally, we can just let this default, but in this case we are using ParameterSetName, which lets us group parameters. The idea here is that if the switch usecredential is passed, the other parameters in the set usecred should also be provided. So, if we just want to send mail behind the firewall where credentials are not needed, we can just pass parameters 0 through 5. If we want to send email and use credentials, we would add the usecredential switch parameter to the call, along with the user ID and password. Parameter sets are often used to simulate polymorphism, also known as function overloading. The idea is that the function can be passed different types of parameters to cause different behavior. This was covered in detail in Chapter 5.

As shown next, we can see in the line `if ($usecredential) {` that if the switch –usecredential is passed, the code in the braces will be executed:

```
if ($usecredential)
{
  Write-Verbose "With Credential"
  $secpasswd = ConvertTo-SecureString "$password" -AsPlainText -Force
  $credential = New-Object System.Management.Automation.PSCredential
     ("$emailaccount", $secpasswd)

  Send-MailMessage -smtpServer $smtpServer -Credential $credential -from $from -to $to
  -subject $subject -Body $body -Usessl -Port $port
}
```

`Write-Verbose` just displays the message if the `Verbose` parameter is passed. It may be acceptable to have the password passed to the function, since that is behind the firewall, but when the function passes it to the outside it could be read; thus, we need to encrypt it with the `ConvertTo-SecureString` cmdlet. The line `$secpasswd = ConvertTo-SecureString "$password" -AsPlainText –Force` is encrypting the clear-text string `$password` and storing the result in `$secpasswd`. Then, we create a `Credential` object using `New-Object`. We pass the email account and encrypted password and store the object reference in `$credential`.

The format of a call to the `Send-MailMessage` cmdlet is shown here:

```
Send-MailMessage -smtpServer $smtpServer -Credential $credential -from $from -to
$to -subject $subject -Body $body -Usessl -Port $port
```

To use the function, we need to know the SMTP server name and port. I had to search around to find what the correct settings are for Outlook and Google. You can get the settings for Outlook at:

http://email.about.com/od/Outlook.com/f/What-Are-The-Outlook-com-Smtp-Server-Settings.htm

For Google, the settings can be found at:

http://email.about.com/od/accessinggmail/f/Gmail_SMTP_Settings.htm

We can call our send mail function using Outlook as shown here:

```
Send-UdfMail  -verbose "smtp.live.com" "myaccount@msn.com" "useremail@gmail.com" "Hellow"
"Email sent from PowerShell." "587" -usecredential " myaccount@msn.com " "mypassword"
```

And we can call our send mail function using Google as shown here:

```
Send-UdfMail  -verbose "smtp.gmail.com" "myaccount@gmail.com" "useremail@gmail.com" "Hello"
"Email sent from PowerShell." "587" -usecredential " myaccount@gmail.com " "mypassword"
```

The email user name for the credential is just your email address for that service, and the password is the password you would use to log into your mail. We can send the email anywhere we want. The example sends the mail to a Google account. Note: I found that for both Outlook and Google you must specify the port, which is 587 for both services.

If the switch parameter `usecredential` is not passed to the function, the code shown here will execute:

```
Else
{
  Write-Verbose "Without Credential"
  Send-MailMessage -smtpServer $smtpServer -from $from -to $to -subject $subject -Body
  $body -Usessl -Port $port
```

Note: The only difference between when `usecredential` is passed versus not passed is that when it is not passed, no credential is passed to the `Send-MailMessage` cmdlet. If you specify the `usecredential` parameter, you must also pass the email account and password. If you do not pass `usecredential`, do not pass email account and password. Also, when credentials are not used, I found it was necessary to specify the `Usessl` parameter.

Many situations require sending email notifications. The send mail function wraps the `Send-MailMessage` cmdlet up to make it is easier to use and supports two variations. It can be called without supplying credentials, which is often done behind an internal network, or with credentials, which allows us to use public services like Outlook or Google.

Load the Files

The scenario is this example assumes CSV files containing sales data will be arriving in an FTP folder, and that we need to load these files into a SQL Server table. The file-naming format of the external files is sales_COMPANYNAME_YYYYMMDD.csv, where COMPANYNAME is the company sending the data and YYYYMMDD is the date as year, month, and day.

The SQL Server table we will load is on a database named Development. Change this to whatever database you want to run this example on. You can create the table with the statement here:

```
CREATE TABLE [dbo].[sales](
        [SalesPersonID] [int] NOT NULL,
        [FirstName] [varchar](50) NOT NULL,
        [LastName] [varchar](50) NOT NULL,
        [TotalSales] [decimal](32, 2) NULL,
        [AsOfDate] [date] NOT NULL,
        [SentDate] [date] NULL
) ON [PRIMARY]
SET ANSI_PADDING ON
ALTER TABLE [dbo].[sales] ADD [FileName] [varchar](150) NULL
PRIMARY KEY CLUSTERED
(
        [SalesPersonID] ASC
)
```

We could use SSIS to load the files, but I think it is better to keep this all in PowerShell since our goal is to learn about PowerShell. In Chapter 11, we will discuss using PowerShell for ETL, so this will provide an introduction to that topic. The following function loads the files into a SQL Server table:

```
function Invoke-UdfSalesTableLoad
{
 [CmdletBinding()]
        param (
              [string] $path,   # Folder to be monitored
              [string] $filepattern    # File name pattern to wait for.
          )

$conn = new-object System.Data.SqlClient.SqlConnection("Data Source=(local);Integrated
Security=SSPI;Initial Catalog=Development");
$conn.Open()

$command = $conn.CreateCommand()

#  Clear out the table first...
$command.CommandText = "truncate table dbo.sales;"
$command.ExecuteNonQuery()
write-verbose $path
write-verbose $filepattern

$loadfilelist = Get-ChildItem $path $filepattern

write-verbose $loadfilelist
```

```
foreach ($loadfile in $loadfilelist)
{

   $indata = Import-Csv $loadfile.FullName
   $indata

   foreach ($row in $indata)
   {

    $rowdata =  $row.SalesPersonID + ", '" + $row.FirstName + "', '" + $row.LastName + "',
    " + $row.TotalSales + ", '" + $row.AsOfDate + "', '" + $row.SentDate + "', '" + $row.
    filename + "'"

    Write-Verbose $rowdata

    $command.CommandText = "INSERT into dbo.sales VALUES ( " + $rowdata + ");"
    $command.ExecuteNonQuery()
   }

}

$conn.Close()

}
```

The `CmdletBinding` shows two supported parameters: `$path`, which is the path to the source files to be loaded, and `$filepattern`, which is the file-name mask of what files to load—i.e., `sales*.csv`, or all files that start with `sales` and end with `.csv`. Let's take a look:

```
function Invoke-UdfSalesTableLoad
{
 [CmdletBinding()]
        param (
              [string] $path,   # Folder to be monitored
              [string] $filepattern    # File name pattern to wait for.
          )
```

We're going to use the SQL Server .NET objects to load the data, as shown here:

```
$conn = new-object System.Data.SqlClient.SqlConnection("Data Source=(local);Integrated
Security=SSPI;Initial Catalog=Development");
$conn.Open()
```

In this code, we create a SQL connection object using `System.Data.SqlClient.SqlConnection` and storing the reference in `$conn`. Then, we open the connection using the `$conn` Open method, i.e., `$conn.Open()`.

Using the connection object, we can execute SQL commands. To make this rerunnable, we need to truncate the target table. This would be a common practice if we were loading to a staging table. Let's look at the code to do this:

```
$command = $conn.CreateCommand()

#  Clear out the table first...
$command.CommandText = "truncate table dbo.sales;"
$command.ExecuteNonQuery()
```

First, we use the connection object's :CreateCommand method to create a Command subobject. Then, we can assign a SQL statement to execute, i.e., `$command.CommandText = "truncate table dbo.sales;"`. Finally, we use the ExecuteNonQuery method to execute the truncated table statement. Note: ExecuteNonQuery is used because the query does not return data.

We need to get a list of the files to load, which we do with the following statement:

```
$loadfilelist = Get-ChildItem $path $filepattern
```

The heart of the load cannot easily be broken into pieces, so let's look at the whole thing:

```
foreach ($loadfile in $loadfilelist)
{

   $indata = Import-Csv $loadfile.FullName
   Write-Verbose $indata

   foreach ($row in $indata)
   {

    $rowdata =  $row.SalesPersonID + ", '" + $row.FirstName + "', '" + $row.LastName + "',
    " + $row.TotalSales + ", '" + $row.AsOfDate + "', '" + $row.SentDate + "', '" + $row.
    filename + "'"

    Write-Verbose $rowdata

    $command.CommandText = "INSERT into dbo.sales VALUES ( " + $rowdata + ");"
    $command.ExecuteNonQuery()
   }

}
```

The first statement, foreach, creates an outer loop that will cycle through all the files in $loadfilelist. The variable $loadfile will contain each individual file. We use Import-CSV to load $loadfile.FullName into $indata. Note: The property FullName is the fully qualified path and file name.

Now that we have each file, we loop through every row in the file with the inner foreach loop, i.e., foreach ($row in $indata). To load the data, we need to format it into an Insert statement. To prepare for this we need to string all the column values together and include quotes and commas as needed. We do this with the following statement:

```
$rowdata =  $row.SalesPersonID + ", '" + $row.FirstName + "', '" + $row.LastName + "', " +
$row.TotalSales + ", '" + $row.AsOfDate + "', '" + $row.SentDate + "', '" + $row.filename +
"'".
```

Note We need to add single quotes around string and date columns.

Then, we assign the `CommandText` as a prefix of `INSERT into dbo.sales VALUES` followed by the variable `$rowdata` we created earlier. Then, we just execute the command object's `ExecuteNonQuery` method, and the row is inserted.

Finally, we close the connection, as show here:

```
$conn.Close()
```

The load file function demonstrates that PowerShell can be used as an ETL tool, but we could also call an SSIS package to load the data. We could have used the SQLPS module to help load the data, but using the SQLClient library directly demonstrates how easy it is to access SQL Server directly. Now, let's discuss how to use the functions we just discussed in an ETL process.

Let's Process Some Files

Now that we've reviewed the individual functions we plan to use in our ETL process, let's see how it all comes together. We want to perform all the steps, from receiving new files from an external source to loading the files into SQL Server tables. The steps to process these files are as follows:

- File is transferred from external source to your company.
- A polling process identifies that a new file has arrived and starts the load process.
- Send an email notification that the load job has started.
- Copy the files to a local drive on the server.
- Unzip the files to an unzipped folder.
- Add the file name as a new column to the file.
- Load the file into SQL Server.
- Archive the original file to an archive folder.
- Send an email notification that the load job has ended.

To make using the script easier to follow, I packaged up the functions we will need into a module named `umd_etl_functions`. Copy this to a folder where Windows looks for modules, such as `\Windows\PowerShell` in your Documents folder. Create a folder named `umd_etl_functions` and copy the module from the code disk to this folder. Normally, I would probably not include the actual load script in a module since it is not generic, but doing so here makes it easier to run the example.

Now, let's take a look at the ETL load script in Listing 9-1.

Listing 9-1. ETL load script

```
Import-Module umd_etl_functions -Force
Clear-Host  # So we can see the messages easily.

$rootfolder = $env:HOMEDRIVE + $env:HOMEPATH + "\Documents"
$ftppath = $rootfolder + "\FTP\sales_*.zip"
$inbound =  $rootfolder + "\Inbound\"
$zippath = $inbound + "\sales_*.zip"
```

```
$unzippedpath = $rootfolder + "\unzipped\"
$processpath = $rootfolder + "\Process\"
$archivepath = $rootfolder + "\Archive\"

# Wait for the files...
Wait-UdfFileCreation $ftppath  -Verbose

# Notify users the job has started using Outlook...
Send-UdfMail  "smtp.live.com" "emailaddress@msn.com" "emailaddress@msn.com" "ETL Job: Sales
Load has Started" "The ETL Job: Sales Load has started." "587" -usecredential "emailaddress@
msn.com" "password"

# Copy the files...
Copy-UdfFile $ftppath $inbound -overwrite

$filelist = Get-ChildItem $zippath

#  Unzip the files...
Foreach ($file in $filelist)
{
   Expand-UdfFile $file.FullName $unzippedpath -force
}

# Add file name to file...
Add-UdfColumnNameToFile $unzippedpath  $processpath "sales*.csv" -Verbose

# Load the files...
Invoke-UdfSalesTableLoad $processpath "sales*.csv" -Verbose

# Archive files...
Move-UdfFile $ftppath $archivepath  -overwrite

# Notify users the job has finished...
Send-UdfMail  "smtp.live.com" "emailaddress@msn.com" "emailaddress@msn.com" "ETL Job:
Sales Load has ended" "The ETL Job: Sales Load has ended." "587" -usecredential
"emailaddress@msn.com" "password"
```

The nice thing about using functions is that we can create a high-level script like the one here that is easy to understand. If there is a problem, we can focus on the specific function where the error is occurring. Let's take a closer look at the script:

```
Import-Module umd_etl_functions
```

This line just imports our module, making all its functions available to us.

Then, we create some variables to help us with the function parameters we'll need, as shown here:

```
$rootfolder = $env:HOMEDRIVE + $env:HOMEPATH + "\Documents"
$ftppath = $rootfolder + "\FTP\sales_*.zip"
$inbound =  $rootfolder + "\Inbound\"
$zippath = $inbound + "\sales_*.zip"
$unzippedpath = $rootfolder + "\unzipped\"
$processpath = $rootfolder + "\Process\"
$archivepath = $rootfolder + "\Archive\"
```

To make this easier to set up, we use subfolders of your `Documents` folder. I like to keep folders separate when I process external files. The subfolders we will use are:

`FTP` = The initial folder from which the files arrive.
`Inbound` = The local server folder to which files from the FTP folder are copied for processing.
`Unzipped` = The folder to which the compressed files are expanded.
`Process` = The folder to which modified versions of the files are written; i.e., when we add the file name as a new column, we write the file out as a new version to \Process.
`Archive` = Folder to which the original files are moved and then retained.

Now, let's look at the code that polls for new files:

```
# Wait for the files...
Wait-UdfFileCreation $ftppath  -Verbose
```

We're calling the `Wait-UdfFileCreation` function, passing the full folder path and file-name pattern to watch for. The `Verbose` parameter causes the function to display any `Write-Verbose` statements in the function. The messages just display the file name being checked for on each iteration of the loop. Once a file matching the requested pattern is found, the function simply exits, allowing the rest of the load process to take place. To test this, run the script. It will wait for the file. Copy the zipped file `sales_abccompany_20141101` to the `FTP` subfolder. That will trigger the script to run.

When a job starts, it's good practice to send a notification to interested parties, which the statement here does:

```
# Notify users the job has started using Outlook...
Send-UdfMail  "smtp.live.com" "emailaddress@msn.com" "emailaddress@msn.com" "ETL Job:
Sales Load has Started" "The ETL Job: Sales Load has started." "587" -usecredential
"emailaddress@msn.com" "password"
```

You will need to replace the credentials with the desired email address and password for your use. If you do not require authentication, just remove the switch `–usecredential` and the parameters after it.

The next statement just copies the file from the FTP folder to the local folder:

```
# Copy the files...
Copy-UdfFile $ftppath $inbound -overwrite
```

For demonstration purposes, all the folders are local. In practice, you'll want to change this to the FTP folder in your organization. The file is compressed, so we need to unzip it, which we do with these statements:

```
$filelist = Get-ChildItem $zippath

#  Unzip the files...
Foreach ($file in $filelist)
{
   Expand-UdfFile $file.FullName $unzippedpath -force
}
```

The first line just stores the list of zip files we need to unzip in the variable `$filelist`. Then, we loop through these files using the `Foreach` statement. Each file item is stored in `$file`. We pass the `FullName` property to the `Expand-UdfFile`, followed by the path at which to store the unzipped files. We also pass the `Force` switch, which causes any existing files of the same name to be overwritten. As mentioned previously, having the name of the file from which each row came is useful for tracking down problems.

We call the function that follows to add the file name as a column to each file:

```
# Add file name to file...
Add-UdfColumnNameToFile $unzippedpath  $processpath "sales*.csv" -Verbose
```

The input files are in the folder specified by `$unzippedpath`. A new file, with the file-name column included, is written out to the folder specified by `$processpath`.

Then, we load the files with the statement here:

```
# Load the files...
Invoke-UdfSalesTableLoad $processpath "sales*.csv" –Verbose
```

The variable `$processpath` is the folder holding the files to be loaded. `sales*.csv` is the file-name pattern to load—i.e., any file that starts with `sales` and ends with `.csv`. Finally, we can archive the files with the following statement:

```
# Archive files...
Move-UdfFile $ftppath $archivepath  -overwrite
```

We are moving the files in folder `$ftppath` to the folder specified in `$archivepath`, and overwriting any files that are already there.

Now we can send an email notifying users the job has finished, as shown here:

```
# Notify users the job has finished...
Send-UdfMail  "smtp.live.com" "emailaddress@msn.com" "emailaddress@msn.com" "ETL Job:
Sales Load has ended" "The ETL Job: Sales Load has ended." "587" -usecredential
"emailaddress@msn.com" "password"
```

Using functions has enabled us to create a job flow that is easy to read and maintain. If you need to do more work, we can just add in more functions. If files will be coming from multiple sources but follow the same pattern, we can make the job script a function so it can be reused for different sources. The purpose of this script is to demonstrate how effectively PowerShell scripts can orchestrate your ETL processes.

More Useful ETL Functions

Sometimes you will need additional functions to get the job done. For example, we often need to validate the data coming in before loading it. Maybe we need to merge multiple files together. Perhaps the files coming in have no column headings, so we need to add them. We may want to make the send email function more secure by encrypting the password to a file where it can be read by the function. In this section, we will review how to accomplish these tasks.

Data Validation

PowerShell provides many ways to validate our data. Since data validation is a common ETL requirement, let's look at the following function, that provides a set of standard column validations:

```
function Invoke-UdfColumnValidation
{
 [CmdletBinding()]
        param (
              [string] $column,       # value to be checked.
              [ValidateSet("minlength","maxlength","zipcode", "phonenumber", "emailaddress",
                           "ssn", "minintvalue", "maxintvalue")]
```

```
            [string] $validation,
            [parameter(Mandatory=$false)]
            [string] $validationvalue
        )

    [boolean] $result = $false;

    Switch ($validation)
    {

        "minlength"    { $result = ($column.Length -ge [int] $validationvalue); break  }
        "maxlength"    { $result = ($column.Length -le [int] $validationvalue); break  }
        "zipcode"      { $result = $column -match ("^\d{5}(-\d{4})?$")        ; break }
        "phonenumber"  { $result = $column -match ("\d{3}-\d{3}-\d{4}")       ; break }
        "emailaddress" { $result = $column -match
          ("^[_a-z0-9-]+(\.[_a-z0-9-]+)*@[a-z0-9-]+(\.[a-z0-9-]+)*(\.[a-z]{2,4})$"); break }
        "ssn"          { $result = $column -match ("^\d{3}-?\d{2}-?\d{4}$")       ; break }
        "minintvalue"  { $result = ([int]$column -ge [int] $validationvalue); break }
        "maxintvalue"  { $result = ([int]$column -le [int] $validationvalue); break }

    }

    Write-verbose "$column $validation $validationvalue"

    Return $result

 }
```

Let's look closely at the function's `CmdletBinding`:

```
function Invoke-UdfColumnValidation  :
{
 [CmdletBinding()]
        param (
              [string] $column,       # value to be checked.
              [ValidateSet("minlength","maxlength","zipcode", "phonenumber", "emailaddress",
               "ssn", "minintvalue", "maxintvalue")]
              [string] $validation,    # File name pattern to wait for.
              [parameter(Mandatory=$false)]
              [string] $validationvalue
          )
```

We can see that this function supports parameters `$column`, `$validation`, and `$validationvalue`. Parameter `$column` is the value you want to validate. This is always passed as a string, so if the value is not a string, cast it to a string before passing it into the function. Note: You can cast a value by placing the desired data type before it; i.e., `[string] myintegervariable`. Parameter `$validation` is the type of :validation we want performed. Notice we use the `ValidateSet` binding attribute, which will restrict the values allowed into the function to the list of values in the parentheses. This list of values will be displayed by Intellisense when you specify the `Validation` parameter, as shown in Figure 9-2.

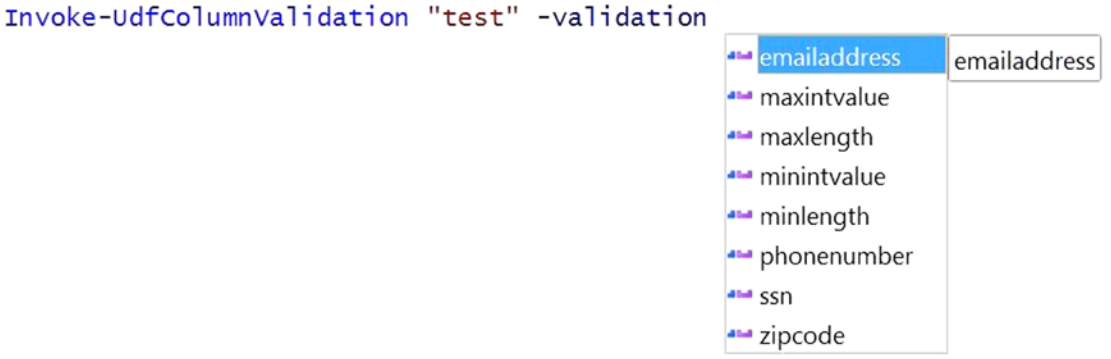

Figure 9-2. *Intellisense showing the list of valid values for a function parameter*

The parameter `$validationvalue`: is optional (the `Mandatory` attribute is `$false`) and should only be passed by a validation that requires a comparison value, such as `minlength`. Format validations like `phonenumber` do not require a comparison value.

The function tells us whether the value passes the test (true) or fails (false)—it will return `true` or `false` when called. The line that follows creates the variable to hold the result and initializes it to `false`:

```
[boolean] $result = $false;
```

The function evaluates what validation is requested by checking the `$validation` parameter. The `Switch` statement provides an elegant way to do this, as shown here:

```
Switch ($validation)
{

    "minlength"    { $result = ($column.Length -ge [int] $validationvalue); break  }
    "maxlength"    { $result = ($column.Length -le [int] $validationvalue); break  }
    "zipcode"      { $result = $column -match ("^\d{5}(-\d{4})?$")        ; break }
    "phonenumber"  { $result = $column -match ("\d{3}-\d{3}-\d{4}")       ; break }
    "emailaddress" { $result = $column -match
     ("^[_a-z0-9-]+(\.[_a-z0-9-]+)*@[a-z0-9-]+(\.[a-z0-9-]+)*(\.[a-z]{2,4})$");
                              break }
    "ssn"          { $result = $column -match ("^\d{3}-?\d{2}-?\d{4}$")       ; break }
    "minintvalue"  { $result = ([int]$column -ge [int] $validationvalue); break }
    "maxintvalue"  { $result = ([int]$column -le [int] $validationvalue); break }

}
```

The `Switch` statement is followed by an expression to be evaluated. Each possible value is listed, with related code in braces after the value. The comparison expression will return `true` or `false`, which gets stored in `$result`. Notice the `break` statement after each code block. We need this because without it the `Switch` statement will continue to be evaluated. Notice how we can use the properties of the `$column` parameter to do some tests, like checking the length. For the pattern matching, we are using regular expressions. Building regular expressions can be difficult, but there is a lot of help on the Internet. I found this site to be particularly rich with examples:

http://www.regexlib.com/

Finally, we return the result to the client, as shown here:

```
Return $result
```

Some example calls for these validations are provided here:

```
# Min/Max tests...
Invoke-UdfColumnValidation "test" -validation "minlength" -validationvalue "2"
Invoke-UdfColumnValidation "test" -validation "maxlength" -validationvalue "2"
Invoke-UdfColumnValidation "1" -validation "minintvalue" -validationvalue "2"
Invoke-UdfColumnValidation "1" -validation "maxintvalue" -validationvalue "2"

# Phone Number format...
Invoke-UdfColumnValidation "123" -validation "phonenumber"
Invoke-UdfColumnValidation "999-999-9999" -validation "phonenumber" -Verbose

# Zip Code format...
Invoke-UdfColumnValidation "02886" -validation "zipcode" -Verbose
Invoke-UdfColumnValidation "02886-1234" -validation "zipcode" -Verbose

# Social Security Number format...
Invoke-UdfColumnValidation "999-99-9999" -validation "ssn" -Verbose
Invoke-UdfColumnValidation "12345" -validation "ssn" -Verbose

# Email Address format—does not validate domain...
Invoke-UdfColumnValidation "022886-1234" -validation "emailaddress" -Verbose
Invoke-UdfColumnValidation "test@msn.com" -validation "emailaddress" -Verbose
Invoke-UdfColumnValidation "test@yahoo.org" -validation "emailaddress" -Verbose
```

Code that uses the validation function might look like this:

```
If (Invoke-UdfColumnValidation "test@yahoo.org" -validation "emailaddress" -Verbose)
{
  "Load the value."
}
Else
{
  "Reject the value."
}
```

This displays the following:

```
VERBOSE: test@yahoo.org emailaddress
Load the value
```

This function provides a starting point and can be extended to include whatever validations we need. We can be creative and add look-ups to find SQL Server tables, calculations, date dimensions, and so on. We can see that PowerShell has a lot of built-in support for validating data, thus allowing us to accomplish a lot with very little code. What we do when data fails validation depends on the requirements. We could filter the row out and write it to an error log, or we could try to fix the value automatically. The real value comes in when we combine this with other functions.

Combine Files

Sometimes it is useful to combine multiple files of the same format into one file. This could be to simplify the load or to consolidate a set of output files. The situation in which I developed this function was that the ETL tool did not support iterating over multiple files in a load. In fact, it required a fixed file name. The function that follows consolidates files into a folder that matches a given file-name pattern:

```
function Add-UdfFile {
[CmdletBinding()]
        param (
              [string] $source,
              [string] $destination,
              [string] $filter
          )

$filelist = Get-ChildItem -Path $source  $filter

write-verbose $source

write-verbose "$filelist"

try
{
 Remove-Item $destination -ErrorAction SilentlyContinue
 write-host "Deleted $destination ."
 }
catch
 {
   "Error:  Problem deleting old file."
 }

# Add-Content $destination "Column1`r`n"

  foreach ($file in $filelist)
  {
   write-verbose $file.FullName

  $fc = Get-Content $file.FullName

  Add-Content $destination $fc
  }
}
```

Let's take a closer look at the function parameters:

```
function Add-UdfFile {
[CmdletBinding()]
        param (
              [string] $source,
              [string] $destination,
              [string] $filter
          )
```

The parameters are `$source`, which is the source folder path; `$destination`, which is the path with the file name of the combined file to be written out; and `$filter`, which is the file-name pattern, such as `*.txt`.

An example of calling this function is seen here:

```
Add-UdfFile 'C:\Users\Documents\' 'C:\Users\ Documents\combined.txt' 'o*.txt' -Verbose
```

The first thing the function needs to do is get a list of the files that need to be combined, which it does as follows:

```
$filelist = Get-ChildItem -Path $source  $filter
```

Then, we need to delete the destination file if it already exists, which we do with the following `Try/Catch` block:

```
try
{
 Remove-Item $destination -ErrorAction SilentlyContinue
 write-host "Deleted $destination ."
 }
catch
 {
   "Error:  Problem deleting old file."
 }
```

Notice that the `Remove-Item` statement includes the `ErrorAction SilientlyContinue` parameter. This is so the code will not fail if the file does not exist. If there is a problem deleting the file, the `catch` block will display an error message.

Now, we need to loop through the file list to merge the files, as shown here:

```
foreach ($file in $filelist)
{
  Write-Verbose $file.FullName

  $fc = Get-Content $file.FullName

  Add-Content $destination $fc
}
```

The `foreach` statement will loop through each file in `$filelist` and store the item in `$file`. `Get-Content` is used to load the file contents using the `$file.FullName` (the path and file name). The `Add-Content` cmdlet is used to append the file contents in `$fc` to the destination file.

Combining files is not a common requirement when your ETL tool is SSIS. When I developed this function, I was using a different ETL tool that required it. This function could also be used to combine multiple output files to be used by an external user.

Add Column Headings

There may be times when the files you are sent do not have column headings, but you know what should be in each column. The function that follows adds column headings to a CSV file. The column headings are a single line at the top of the file, with column names separated by commas. See the code here:

```
function Add-UdfFileColumnHeading {
[CmdletBinding()]
        param (
              [string] $sourcefile,
              [string] $destinationfile,
              [string] $headingline
          )

 if (Test-Path  $destinationfile)
 {
    Remove-Item $destinationfile -Force
    write-verbose "Deleted $destinationfile ."
 }

 Add-Content $destinationfile "$headingline"

 $fc = Get-Content  $sourcefile

 Add-Content $destinationfile $fc
}
```

We can see that the `CmdletBinding` supports the parameters of `$sourcefile`, which is the path and file name of the input file; `$destination`, which is the path and file to the output file; and `$headingline`, which is the exact line you want inserted at the top of the file to serve as column headings—you need to include comma separators. If the output file already exists, we need to delete it, which we do with the code here:

```
if (Test-Path  $destinationfile)
 {
    Remove-Item $destinationfile –Force
    Write-Verbose "Deleted $destinationfile ."
 }
```

The `Test-Path` cmdlet, as we have seen, returns `true` if the path exists and `false` if it does not. So, the previous code will delete the destination file name if it already exists.

Next, we write the heading line to the output file name with this line:

```
Add-Content $destinationfile "$headingline"
```

`Add-Content` adds the second parameter, i.e., `$headingline`, to the first parameter, which is the destination file name. Now, we need to append the source file to the destination file, as follows:

```
$fc = Get-Content  $sourcefile

 Add-Content $destinationfile $fc
```

In this code, first we load the source file into `$fc` using the `Get-Content` cmdlet. Then, we use `Add-Content` to append `$fc` to the destination file.

The function can be called as follows:

```
Add-UdfFileColumnHeading 'C:\Documents\outfile1.txt' 'C:\Documents\filewithheadings.txt' `
"firstname,lastname,street,city,state,zip" -Verbose
```

Sometimes files need to be loaded that do not have column headings. You can load them this way, but it is a lot easier to work with files that have column headings, especially when you want to use PowerShell's `Import-CSV` cmdlet. The function in this example is simple yet effectively adds column headings to a file and, best of all, can be used for any CSV file.

Securing Credentials

Some operations require secure information, like passwords. For example, in the send email function we used in the ETL job, the password was clear text which means anyone can see it. If you send clear text over the Internet, anyone can read it. It's not a good idea to have things like passwords available for anyone to read. A better idea is to encrypt the password and store it in a file so it can be safely used.

Before we look at the encryption function, we need to review a helper function that we will use to prompt the user for the destination file location and name. We're going to leverage the built-in Windows Save File Common Dialog for this purpose.

Let's review the code:

```
function Invoke-UdfCommonDialogSaveFile($initialDirectory)
{
 [System.Reflection.Assembly]::LoadWithPartialName("System.windows.forms") |
 Out-Null

 $OpenFileDialog = New-Object System.Windows.Forms.SaveFileDialog
 $OpenFileDialog.initialDirectory = $initialDirectory
 $OpenFileDialog.filter = "All files (*.*)| *.*"
 $OpenFileDialog.ShowDialog() | Out-Null
 $OpenFileDialog.filename
}
```

This function is just a helper function. It will invoke the Windows Save File common dialog so the user can use a GUI to specify the path and file name of the encrypted password file. The first line defines a reference to the Windows forms object library. Once that is done, we can use the statement copied next to create a new Windows Save File dialog object instance:

```
$OpenFileDialog = New-Object System.Windows.Forms.SaveFileDialog
```

Now that we have the Save File dialog reference, we can set the configuration properties for it, such as the initial directory and default file-name pattern, with the lines seen here:

```
$OpenFileDialog.initialDirectory = $initialDirectory
$OpenFileDialog.filter = "All files (*.*)| *.*"
```

Then, all we have to do to open the dialog box is execute the `ShowDialog` method, as follows:

```
$OpenFileDialog.ShowDialog() | Out-Null
```

In this code, we are piping the output of the ShowDialog method to Out-Null so as to suppress getting back any output from the command. The SaveFileDialog sets the object's filename property to the name of the file selected or entered by the user, so we want to return that to the caller with the line seen here:

```
$OpenFileDialog.filename
```

Now that we have the helper function defined, we can code a function to get and encrypt the password. Review the code that follows, which does this. Note: This function is in the module umd_database. See here:

```
function Save-UdfEncryptedCredential
{
[CmdletBinding()] param ()

   $pw = read-host -Prompt "Enter the password:" -assecurestring

   $pw | convertfrom-securestring |
   out-file (Invoke-UdfCommonDialogSaveFile  ($env:HOMEDRIVE + $env:HOMEPATH + "\Documents\"
) )
}
```

The function Save-UdfEncryptedCredential is very short yet does a lot of work. Notice this function does not take any parameters. When we call it, we are prompted to enter a password by the Read-Host cmdlet, which is masked and encrypted by the AsSecurestring parameter and returned to the variable $pw. The variable $pw is piped into ConvertFrom-SecureString, which converts the data into a string and pipes it into the Out-File cmdlet, which writes it to a file. Let's look more closely at the Out-File statement:

```
out-file (Invoke-UdfCommonDialogSaveFile  (($env:HOMEDRIVE + $env:HOMEPATH + "\Documents\" ) )
```

As in math, the parentheses control the order of execution. The code in the innermost parentheses, which defines the file default path, is executed before the code in the outer parentheses, which prompts us for a file path and name using the Windows Save File common dialog. The result of the Save File dialog is passed to the Out-File cmdlet, i.e., that is where the encrypted file is written.

There are times when we need to use a password to get a task done. It might be the password to connect to SQL Server or to log on to an account. It not a good practice to have passwords in clear text in a script for people to see. Instead, we can use the Save-UdfEncryptedCredential to encrypt the password and store it in a file. From there, it can be read securely and used by any code that needs it.

Send Secure Email

So, we have the password encrypted in a file. How do we use it? The Send Email function we saw earlier had a significant issue. The calling script had to pass the parameter password in clear text. Let's revisit the Send Mail function, which is enhanced to use an encrypted password file:

```
function Send-UdfSecureMail
{
[CmdletBinding(SupportsShouldProcess=$false)]
        param (
            [Parameter(Mandatory = $true)]
            [string]$smtpServer,
            [Parameter(Mandatory = $true)]
            [string]$from,
```

```
            [Parameter(Mandatory = $true)]
            [string]$to,
            [Parameter(Mandatory = $true)]
            [string]$subject,
            [Parameter(Mandatory = $true)]
            [string]$body,
            [Parameter(Mandatory = $true)]
            [string]$port,
            [Parameter(Mandatory = $true)]
            [string]$emailaccount,
            [Parameter(Mandatory = $true)]
            [string]$credentialfilepath
            )

     $credin = Get-Content $credentialfilepath | convertto-securestring

     write-verbose $credentialfilepath

     $credential = New-Object System.Management.Automation.PSCredential ("$emailaccount",  $credin)

     Send-MailMessage -smtpServer $smtpServer -Credential $credential -from $from -to $to
     -subject $subject -Body $body -Usessl -Port $port
}
```

Our new function, `Send-UdfSecureMail`, has a few differences from `Send-UdfMail`. From a parameters standpoint, the big change is that now instead of taking a password parameter, the function accepts a credential path (`$credentialfilepath`) parameter, which is the path and file of the credential file we created earlier. The line here loads the credential file:

```
$credin = Get-Content $credentialfilepath | convertto-securestring
```

`Get-Content` loads the file, which is piped into the `ConvertTo-SecureString` cmdlet to be encrypted. Note: Technically the file was not encrypted. Rather, the encrypted password was converted to an unintelligible string that could be written to a file. `ConvertTo-SecureString` encrypts it again so we can use it securely. The encrypted password is stored in `$credin`.

The line that follows is an example of how to call the secure email function:

```
Send-UdfSecureMail  -verbose "smtp.live.com" "fromemail" "toemail" "Yo!" "Email sent from
PowerShell." "587" "emailaccount" "c:\somedir\mycredential.txt"
```

The secure send email function is a big improvement over the version with a clear-text password parameter. The secure version reads the password from an encrypted file, thus allowing it to send email without exposing the password to anyone who reads the code. The method used here is very basic. You can pass the `SecureKey` parameter to the `ConvertTo-SecureString cmdlet` when you call it in order to provide a key to decode the string. If you require tighter security, you might want to look at third-party encryption tools.

Summary

This chapter focused on how to use PowerShell to augment your ETL development. We began by considering steps common to many ETL processes. We discussed functions to perform these tasks that you can incorporate into your ETL work. These functions include things such as polling a folder for the arrival of new files matching a file-name pattern, copying files, decompressing files, adding the file name as a column to the files, loading the file into SQL Server, and sending email notifications. This culminated in a complete ETL script that used these functions to process a flat file load from start to finish. Then, we discussed additional functions that are useful in ETL development—data validation, file merging, the addition of column headings, the encryption of credentials, and a secure-send email function. All these functions are provided in a module that can be used in your ETL development.