CHAPTER 10

Configurations, Best Practices, and Application Deployment

In this chapter, we are going to discuss several related topics: configuring PowerShell applications, development best practices, and application deployment. First, we will discuss how to configure PowerShell applications to so as maximize flexibility and ease of maintenance. A nice feature of SQL Server Integration Services (SSIS) is that it has a number of configuration methods. In this chapter, we will discuss how to achieve comparable functionality in PowerShell using several approaches. Then, we will discuss the goals of best practices and how to apply them to PowerShell. Finally, we will consider deploying PowerShell applications and ways to simplify this process.

Configuring PowerShell Applications

How we design a configuration depends on the technical environment we are working in. For example, at one company where I worked there were many development, integration, and QA environments for the same application. Each pointed to different database servers, but ultimately each was to deploy to the same production environment. The development and integration environments were split out by geographic region, while the QA environments were for different releases. New environments were created routinely, and others were removed. In this situation, a sophisticated configuration design was required. At a much smaller company, the data warehouse environments consisted of a development server and a production server. The configuration approach used was a simple one that employed environment variables. Let's consider several approaches to configuring PowerShell applications, each designed to support a different level of complexity. Note: You will need to start PowerShell as Administrator to run many of the scripts in this chapter.

Using Environment Variables

Windows environment variables are used by Windows itself and by many applications to store configuration settings. There are three environment variable levels, which are *machine*, *user*, and *process*. A level refers to who or what processes can access the variable and how long the variable persists on the machine. Machine- and user-level variables persist after the PowerShell session ends. In fact, they are stored in the Windows Registry and will be reloaded after a system reboot. Machine-level variables are visible to all users and all processes on the machine. User-level variables are visible only to the user that created the variable. Process-level variables only exist for the duration of the PowerShell session and are only visible to that session.

As we discuss how to create and maintain environment variables, we will also discuss what happens if there is an environment variable with the same name defined at the machine, user, and process levels. The issue is knowing which value we will see in our scripts. Let's test it out and see. Start PowerShell as Administrator and enter the following statement, which creates a machine-level environment variable named `testvar`:

```
[Environment]::SetEnvironmentVariable('testvar', 'machine', 'Machine')
Get-Process explorer | Stop-Process  #  Line to address Windows bug
```

The first line creates a new machine-level environment variable. The line after it addresses a Windows bug. After creating a new machine- or user-level environment variable, Windows Explorer does not function correctly. In fact, in my testing, I found that none of the taskbar programs would work, and Windows said it could not find the program. I found the issue in a blog related to adding environment variables via the control panel, and the solution is to stop and restart Explorer. Hence, we have the second line. Interestingly, in the PowerShell environment, when you stop the Explorer process, it automatically restarts itself. Note: If Explorer does not restart, you can run the statement `'Start-Process Explorer'` to start Windows Explorer. For other issues, rebooting your machine should correct the problem.

Now, exit PowerShell and restart it using the following command so you can see the new environment variable:

```
Get-Item env:testvar
```

You should see the following output:

```
Name                           Value
----                           -----
testvar                        machine
```

Now enter the statements seen here:

```
[Environment]::SetEnvironmentVariable("testvar", "user", "User")
Get-Process explorer | Stop-Process #  Line to address Windows bug
```

Exit PowerShell and restart it so you can see the new environment variable, as shown:

```
Get-Item env:testvar
```

You should see the following output confirming the user environment variable was created:

```
Name                           Value
----                           -----
testvar                        user
```

Now, add a process-level environment variable as follows:

```
[Environment]::SetEnvironmentVariable("testvar", "process", "Process")
```

Do not exit PowerShell, since this is only a process-level environment variable. Display the variable value as shown here:

```
Get-Item env:testvar
```

You should see the following output:

```
Name                           Value
----                           -----
testvar                        process
```

We can see from this exercise that each variable level overrides the next. This can be useful in testing, as we can set user-level variables to override the machine-level values and then remove them when we are done. We could also have different accounts run various scripts and set configuration variables at the user level so that they each get different values. To just override variables for a session, we can create process-level variables. However, be careful with this, because it may not be obvious to other developers or users of your code that there are both user-level and process-level variables, and they may become confused as to where the values are coming from.

If we need to remove a process-level environment variable, we can use the `Remove-Item` cmdlet, as shown next. This cmdlet will not work for user- or machine-level environment variables. See here:

```
# To remove a process-level environment variable.
Remove-Item Env:testvar
```

For user- and machine-level environment variables, the first executable statement that follows is the only way to remove the variable, and it will be visible until you restart PowerShell:

```
[Environment]::SetEnvironmentVariable("testvar",$null,"Machine")
Get-Process explorer | Stop-Process #  Line to address Windows bug
```

Although we can define and use environment variables to directly store all the configuration settings for PowerShell applications, a more flexible approach is to use them in combination with other tools. The previous discussion was simply to get acquainted with defining and using environment variables in PowerShell.

PowerShell Variables Scope

When a PowerShell session is started, it creates a global memory space we can think of as a container. The global container stores the built-in aliases, environment variables, preference variables, and PowerShell drives. When we execute a script, a sub-container is created within the global container to hold the objects created by the script. If a script or function is called from this script, a container within the script's memory is created. As scripts and functions are called, these containers continue to be nested. PowerShell does this so it can keep each memory area separate and protect us from inadvertently changing an object owned by another script or function. Figure 10-1 shows a visual representation of how PowerShell segregates these containers.

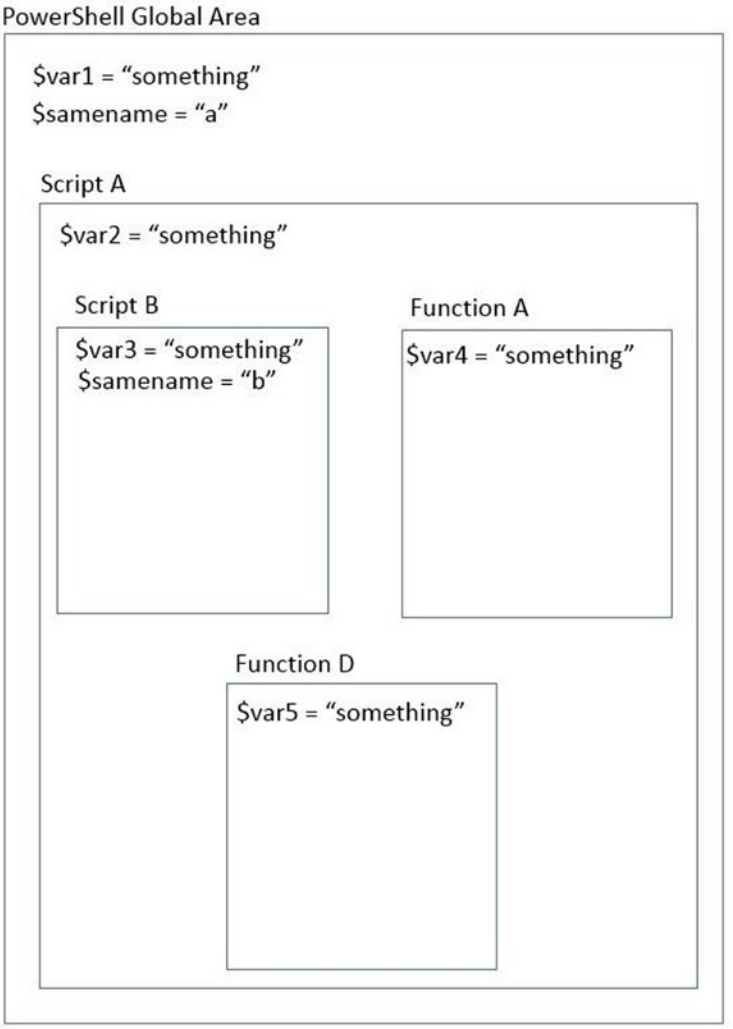

Figure 10-1. *PowerShell memory to support scopes*

In Figure 10-1, we can see that each block of code has its own container, thus creating a hierarchy. Each level of the hierarchy is called a *scope*. The outermost scope is the global scope. Because the Script A container is within another container, i.e., the global container, the global scope is called the parent scope and Script A's scope is called the child scope. The containers are nested to form a hierarchy. Whether code in one container can see objects in another container is determined by whether the other container is in a parent or child scope. By default, code in the child scope can see objects in the parent scope, but code in the parent scope cannot see objects in a child scope. Containers at the same level are called *siblings*. Siblings cannot see each other's objects. In Figure 10-1, we can see that the containers Script B, Function A, and Function D are all children to Script A and are siblings to each other. Therefore, they can see the global variable `$var1` and Script A's variable `$var2`. They cannot see each other's objects because they are siblings. The global scope can only see `$var1`. Script A can only see `$var1` and `$var2`.

To access an object created by the currently executing script, we refer to it with the script scope. *Local scope* is a relative term that refers to the scope of the currently executing code. *Private scope* is explicitly defined by the code when access to the object is restricted. Private scope means that the object can only be seen within the scope in which it was created. There are two ways we can specify a variable's scope. One is to prefix the variable name with a scope, as shown:

```
$global:var1 = 'w'  # This is a global variable.
$script:var2 = 'x' # This is a script-scoped variable.
$private:var3 = 'y' # This is a privately scoped variable.
```

These statements reference variables at each scope. When no scope prefix is used, the default scope used depends on whether the variable is being read or assigned a value. If it is being read, PowerShell will search for a variable, starting in the current scope and going up the hierarchy until it is found. The statement here is an example:

```
If ($myvar –eq 'a') { "Yes" }
```

As a result of this statement, PowerShell will search for the variable `$myvar` in the current scope and move up the scope hierarchy.

When a variable is assigned a value without a scope prefix, the variable will be created in the current scope, if it was not already created there, and will be assigned a value. The statement here is an example of this:

```
$var4 = "z"
```

A new variable is created in the current scope because no scope prefix was defined.

The other way to reference a variable's scope is by using the `Scope` parameter of the variable cmdlets. Let's look at some examples of using the `Scope` parameter with `Set-Variable`:

```
Set-Variable -Name myvar -Value 'something' -Scope global
Set-Variable -Name myvar -Value 'something' -Scope script
Set-Variable -Name myvar -Value 'something'
```

The `Set-Variable` cmdlet assigns a value to a variable. If the variable does not exist, it creates it. The first line assigns a value to a globally scoped variable. The line after that assigns a value to a script-scoped variable. The third line assigns a value to a locally scoped variable. We can also use the `New-Variable` cmdlet to create a variable. Examples of using this cmdlet are seen here:

```
#  Note: we use the visibility parameter to set the scope to private.
New-Variable -name myvar2 -Visibility private -Value 'x'

New-Variable -name myvar3 -Scope global   -Value 'y'
```

The first non-comment statement uses `New-Variable` to create a privately scoped variable named `$myva2` and assigns a value of `'x'` to it. Notice: The statement does not use the `Scope` parameter. Instead, it uses the `Visibility` parameter to specify that the variable is `private`. The line after that creates a globally scoped variable named `$myvar3` and assigns a value of `'y'` to it.

The cmdlet `Get-Variable` can be used to retrieve the value of a variable. The statement that follows gets the value of `$myvar3` in the global scope:

```
Get-Variable myvar3 -Scope global
```

`Get-Variable` can be used with a wildcard filter to get information on multiple variables. The example here demonstrates this:

```
Get-Variable -Name m*
```

This statement will show the name and value of all variables whose name begins with the letter *m*. If we want to see variables of a given scope, we can use the `scope` parameter as shown next. Note: In addition to variables we created, we will see variables called *automatic* and *preference* variables that are used by PowerShell to control preferences and configuration settings. See the `scope` parameter here:

```
Get-Variable -Scope global
```

This statement will get the names and values of all globally scoped variables.

The `Clear-Variable` cmdlet can be used to empty the variable of any value. The `Remove-Variable` cmdlet will delete the variable. Do not try to clear or remove PowerShell's automatic or preference variables.

Now that we've covered the basics of variable scopes, let's look at what happens when a variable of the same name is declared in multiple scopes. The script in Listing 10-1 creates and assigns values to the variable `$myvar` at different scopes. The goal is to demonstrate which scope we see by default when no scope prefix is used and to see if we can access the parent scopes. Listing 10-1 must be executed from the CLI. Running it from within the ISE will not produce the correct results.

Listing 10-1. Examining variable scope

```
# Some script
$global:myvar = 'global'   # All code in the session can access this.
"`$myvar is at the global scope - value is '$myvar'"

'Creating a new variable $myvar'
$myvar = 'script'   # Creates a new variable, which hides the global variable.
"By default `$myvar is at the script scope - value is '$script:myvar'"

"We can still access the global variable using the scope prefix `$global:myvar - value is
'$global:myvar'"

function Invoke-UdfSomething
{
  "Now in the function..."
  "`$myvar in the function sees the script scope - value is '$myvar'"

  # If we assign a value to $myvar we create a new variable in the function scope.
  'We assign a value to $myvar creating a function-scoped variable'
  $myvar = 'function'

  "The default scope for `$myvar is now at the function level - value is '$myvar'"

  "within the function - can use scope prefix to see global variable `$global:myvar - is
   $global:myvar"

  "within the function - can use scope prefix to see script variable `$script:myvar - is
   $script:myvar"
```

```
  "within the function - local scope is function - `$myvar is $myvar"

}

Invoke-UdfSomething # Show variable scope in function

"Out of the function now..."
# Can we still access the global scope?
'We can still see the global scope...'
get-variable myvar -scope global
"`r`nAnd we can still see the script scope..."
get-variable myvar -scope script   # Local scope is script
```

Let's discuss Listing 10-1 and its output. The script creates a variable with the same name at different scopes. First, the variable `$myvar` is created at the global scope and assigned a value of `'global'`, which it then displays. Then, the script attempts to modify `$myvar` by assigning it a value of `'script'`. This creates a new variable of the same name with a script scope. It displays the value of the variable, which we expect to be `'script'`. Then, the globally scoped `$myvar` is displayed to prove we can still access it. Next, a function is called that displays the value of the script variable `$myvar`, i.e., `'script'`. The function attempts to assign a value `'function'` to the `$myvar`, thereby creating a new variable named `$myvar` scoped to the function. The script then displays `$myvar`, which should show a value of `'function'`. The function variable hides the script variable. Note: We can still access it if we use the scope prefix. Then, the function attempts to display the globally scoped `$myvar` using the `global` prefix, which we expect to display `'global'`. It then attempts to display the script-scoped `$myvar` using the `script` prefix, which should display `'script'`. Once again, it displays the locally scoped `$myvar` by omitting the scope prefix, which should be `'function'`. After we exit the function, the script attempts to display the globally scoped `$myvar`, which should display `'global'`, and the script-scoped `$myvar` variable, which should display `'script'`. Let's look at the output of this script:

```
$myvar is at the global scope - value is 'global'
Creating a new variable $myvar
By default $myvar is at the script scope - value is 'script'
We can still access the global variable using the scope prefix $global:myvar - value is 'global'
Now in the function...
$myvar in the function sees the script scope - value is 'script'
We assign a value to $myvar creating a function-scoped variable
The default scope for $myvar is now at the function level - value is 'function'
within the function - can use scope prefix to see global variable $global:myvar - is global
within the function - can use scope prefix to see script variable $script:myvar - is script
within the function - local scope is function - $myvar is function
Out of the function now...
We can still see the global scope...

Name                           Value
----                           -----
myvar                          global

And we can still see the script scope...
myvar                          script
```

To review, the globally scoped variable, `$myvar`, is created and displays the assigned value. When the script assigns a value to `$myvar`, a script-scoped version of `$myvar` is created and assigned the value `'script'`, which is written to the console. Then, the globally scoped variable is displayed. Within the function, we can see that initially `$myvar` is the script-scoped variable, because its value is displayed. When the function assigns the value `'function'` to `$myvar`, a function-scoped version of `$myvar` is created and displayed to the console. Then, the function displays the globally scoped `$myvar` by using the global scope prefix. Then, the script-scoped variable is displayed using the script-scope prefix. The function-scoped variable is displayed again, confirming it is visible as the default local scope. Once we exit the function, the script displays the globally scoped `$myvar`. Then, it displays the script-scoped variable. The purpose of this exercise is to provide a sense of how scoping is applied in PowerShell.

We intentionally held off on covering scope until this chapter. We need to understand scope if we use PowerShell variables to support application configuration settings. However, I believe it is better to avoid depending on PowerShell scoping in our applications. It is critical to make our applications as transparent as possible—i.e., make it easy to read our code and understand what it is doing. By explicitly defining what each script and function will share with another script or function in the form of parameters, we avoid confusion over what values are being passed and what the state of our variables are at a given point in code execution. We also avoid creating dependencies in our code that make reusability and deployment more complex. For example, if a function depends on getting a value from a variable in the calling script, it cannot be easily reused by other scripts unless they are aware of these internal dependencies. It is possible that a function may have a variable of the same name defined in its parent, but since the function's scope will override the parent's scope, this has no effect on the function.

A Simple Configuration Approach

Even in a small enterprise with few servers, a flexible configuration architecture is needed. The best solution minimizes the number of changes required when deploying the application to an environment. It also must be able to easily accommodate changes in the environment, such a drive remapping, database server name changes, and the like. In a small enterprise where only one or two developers maintain the code, a simple configuration approach is to use PowerShell globally scoped variables. We assign these variable values in the profile so that when our scripts run they will always have values. Note: Variables created in the profile are automatically globally scoped.

Assuming we want this configuration to be used by all users and PowerShell hosts, enter the command that follows to edit the PowerShell profile for all users and all hosts. Note: Start PowerShell as Administrator to be able to save to `$profile.AllUsersAllHosts`.

```
Notepad.exe $profile.AllUsersAllHosts
```

Then, enter the lines shown in Listing 10-2.

Listing 10-2. Configuration settings

```
$emailserver='smtp.live.com'
$emailaccount='someacct@msn.com'
$emailfrom='someemail@msn.com'
$emailpw='somepassword'
$edwteamemail='edwteam@msn.com'
$emailport='587'
$ftpinpath='\\ftp\inbound\'
$ftpoutpath='\\ftp\outbound\'
$edwserver='(local)'
```

The profile script in Listing 10-2 creates and assigns values to the variables we will use in our PowerShell application. The assumption is that there are not a lot of different resources for a given environment; i.e., there is one data warehouse server for development. When the PowerShell application is deployed to each environment, the values in the profile would be modified as required for that environment. Assuming the profile script in Listing 10-2 has executed, an example of how we would use it in our code is shown in Listing 10-3. It is a call to the function `Send-UdfMail` from the `umd_etl_functions` module we discussed in Chapter 8.

Listing 10-3. Calling a function using configuration variables

```
Import-Module -Name umd_etl_functions

$Params = @{
    smtpServer = $global:emailserver
    from = $global:emailfrom
    to = $global:edwteamemail
    subject = 'Important Message'
    body = 'ETL Job executed'
    port = $global:emailport
    usecredential = $false
    emailaccount = $global:emailaccount
    password = $global:emailpw
}
Send-UdfMail @Params
```

The code in Listing 10-3 assigns the list of parameters to `$Params` and then passes that to the function. This avoids having to use the line-continuation character. This technique is called *splatting*. The code in Listing 10-3 calls the function `Send-UdfMail` from the `umd_etl_functions` module in order to send an email using the configuration variables created in the profile. This approach can be effective for simple implementations. For large environments or complex applications, it may not meet the needs. Also, because any PowerShell code can create variables with the same names as the configuration variables, there is the risk that the global variable values will be overridden accidentally, causing problems that are difficult to debug. In the next section, we'll discuss a configuration approach that supports namespaces for our configuration variables.

A More Advanced Configuration Approach

A somewhat more robust approach to the one just discussed is to create a *machine-level environment variable* that holds the path to a file that stores the values of configurable parameters appropriate to the environment as name/value pairs. A machine-level environment variable is a variable that all users can see and that persists when we exit PowerShell. The name/value pairs can easily be loaded into a globally scope hash table variable that PowerShell code can reference. We could just create a globally scoped PowerShell variable that points to configuration file, but that could be removed or modified by a PowerShell user or unrelated script or function. Machine-level environment variables can only be changed by users with Administrator privileges.

The first step is to create the machine-level environment variable. To do this, we must start the PowerShell ISE as Administrator. Since the PowerShell cmdlets cannot create environment variables that persist after we exit PowerShell, we need to use the .NET framework, as shown here:

```
#  Create new machine-level environment variable.
#  You need to start PowerShell as Admin to do this.
#  Types are Machine, Process or User level.
[Environment]::SetEnvironmentVariable("psappconfigpath", `
"c:\psappconfig\psappconfig.txt", "Machine")
Get-Process explorer | Stop-Process #  Line to address Windows bug
```

This statement will create a machine-level environment variable named `psappconfigpath` with a value of `c:\psappconfig\psappconfig.txt`. To create a process-level variable, i.e., one that only exists until the PowerShell session ends, we would replace `Machine` with `Process`. To create a user-level variable, we would replace `Machine` with `User`.

The new variable's value will not be accessible when we use the cmdlet `Get-Item` until we exit and restart PowerShell. Instead, we can use the .NET method `GetEnvironmentVariable`, which will display the variable's value without requiring us to restart PowerShell. An example of this is shown here:

```
# Until we exit and restart PowerShell, the new variable may not show up.
[Environment]::GetEnvironmentVariable("psappconfigpath","Machine")
```

We are passing the variable name as the first parameter and the level as the second. We should see the variable's value displayed to the console.

Once we have the environment variable ready, we need to create the file that will hold the configuration values, which we do by entering the following command:

```
Notepad.exe $env:psappconfigpath
```

Then, enter the lines in Listing 10-4 and save the file to `c:\psappconfig\psappconfig.txt`.

Listing 10-4. Configuration file

```
emailserver=smtp.live.com
emailaccount=someacct@msn.com
emailfrom=someemail@msn.com
emailpw=somepassword
edwteamemail=edwteam@msn.com
emailport=587
ftpinpath=\\\\ftp\\inbound\\
ftpoutpath=\\\\ftp\\outbound\\
edwserver=(local)
```

These lines assign name/value pairs that we can load into a hash table. Notice the use of double backslashes. The backslash is interpreted as an escape character, so we need two of them to tell PowerShell we really want the backslash character. Assuming we want this configuration to be used by all users and PowerShell hosts, enter the command that follows to edit the PowerShell profile for all users and all hosts. Note: You must have started PowerShell as Administrator to be able to save to `$profile.AllUsersAllHosts`. Consider carefully which profile to use, and limit the scope to what is needed. If there is a risk that other users will modify or reuse the configuration variables, create them in the user-specific profile.

```
Notepad.exe $profile.AllUsersAllHosts
```

Then, enter the following statements into the profile script, save the script, and exit:

```
$global:psappconfig = Get-Content -Path $env:psappconfigpath | ConvertFrom-StringData
(Get-Variable –Scope global -Name psappconfig).Options = "ReadOnly"
```

The first statement will load the name/value pairs from the file pointed to by the environment variable `psappconfigpath` into the PowerShell globally scoped variable `$global:psappconfig`. Globally scoped variables are visible to all code executing in the current session. The second statement sets the variable to `ReadOnly`, which means developers cannot accidently modify it. Technically, they could use the `Force` parameter, but that requires a conscious knowledge that they are overriding the variable. Since we added the line to the `AllUsersAllHosts` profile script, everyone will see it for any PowerShell host. To prove this, restart PowerShell and enter the statements seen here:

```
$global:psappconfig             # Lists all the name/value pairs to the console.
$global:psappconfig.edwserver   # Uses the name edwserver to lookup the value
```

If we did everything correctly, these statements should display entries from the text file to the console.

One way to make use of configuration values is to use the `$PSDefaultParamterValues`, which was covered in Chapter 8. However, that makes it less obvious what values are being used by our functions. Instead, we pass the `$psappconfig` hash table values as parameters. The code in Listing 10-5 calls the `Send-UdfMail` function we saw in Chapter 9, passing hash table values from the `$psappconfig` variable.

Listing 10-5. Calling Send-UdfMail using global configuration variables

```
Import-Module –Name umd_etl_functions

$Params = @{
    smtpServer = $global:psappconfig.emailserver
    from = $global:psappconfig.emailfrom
    to = $global:psappconfig.edwteamemail
    subject = 'Important Message'
    body = 'ETL Job executed'
    port = $global:psappconfig.emailport
    usecredential = $true
    emailaccount = $global:psappconfig.emailaccount
    password = $global:psappconfig.emailpw
}

Send-UdfMail @Params
```

The first line in Listing 10-5 imports the module `umd_etl_functions`, which includes the function `Send-UdfMail`. Then, we call that function, passing the appropriate values from `$psappconfig`. Notice that we use the hash table's key, which automatically gets translated to the associated value, thus making our code self-documenting. We include the global-scope designation when we get the variable values so as to avoid the possibility that a more locally scoped variable with the same name gets used for the value.

This call is just an example. The idea is to write all function calls using the configuration variable's hash table values for any configurable parameters. The nice thing about using this approach to define parameter values is that to deploy the code to another machine, we just need to copy the code and configuration file to the new machine. Then we edit the configuration values to what is appropriate for that environment. However, this configuration design may still be inadequate. For example, if there were different sources of data that arrived on different FTP servers, it may get unwieldy to have to create distinct names for each folder. The ability to support namespaces for each application or source might be needed. Methods to include namespaces will be covered next.

Taking Configuration a Step Further

For larger and more complex environments, the previous configuration design may be insufficient. Consider an instance where there are many different applications that do not all share the same configuration values, even for the same type of item. For example, in an insurance company, claim files may arrive on a different FTP server than account receivable files do. Email settings may also differ. In such cases, it may be useful to be able to group configuration settings by project. To support this, we create a separate CSV file to hold the configuration data; that data is shown in Listing 10-6. These must be saved in a file named `c:\pasappconfig\psappconfig.csv`.

Listing 10-6. Configuration settings CSV file

```
project,name,value
general,edwserver,(local)
general,emailserver,smtp.live.com
general,ftppath,\\ftpserver\ftp\
edw,teamemail,edwteamdist
edw,stagingdb,staging
finance,ftppath,\\financeftpserver\ftp\
finance,dbserver,financesqlsvr
finance,outfolder,\\somepath\outdata\
```

The first row in Listing 10-6 supplies the column names. The name and value are the configuration properties like `edwservername`. The project column groups the configurations, which allows us to have the same configuration name more than once. This is needed when different parts of the application need different values. In the listing we can see that the team email is under both the EDW project and the finance project, with different values. At the general level, which might be for global settings, there is an `ftppath` with a different value than the `ftppath` seen under the finance project.

To load the configuration values, we've added a function named `Set-UdfConfiguration` to the `umd_etl_functions` module. Let's look at the function's code in Listing 10-7.

Listing 10-7. Set configuations function

```
function Set-UdfConfiguration {
 [CmdletBinding()]
        param (
                [string]      $configpath
              )
    [psobject] $config = New-Object psobject

    $projects   = Import-CSV -Path $configpath | Select-Object -Property Project -Unique
    $configvals = Import-CSV -Path $configpath

    foreach ($project in $projects)
    {
        $configlist = @{}

        foreach ($item in $configvals)
        {
            if ($item.project -eq $project.project )
               { $configlist.Add($item.name, $item.value)  }
        }
```

```
        #  Add the noteproperty with the configuration hash table as the value.
        $config |
        Add-Member -MemberType NoteProperty -Name $project.project -Value $configlist
    }

    Return $config

}
```

The function `Set-UdfConfiguration` takes one parameter, which is the path to a CSV file with the configuration settings. The file has three columns: `project`, `name`, and `value. Project` is used to group names and values. This allows the same configuration name to be used more than once with different values. The function creates a `psobject` and adds a property for each project name. The line in the function that does this is copied here:

```
$projects   = Import-CSV -Path $configpath | Select-Object -Property Project |
Get-unique -AsString
```

The file is imported, piped into the `Select-Object` cmdlet, which extracts the `Project` property and pipes this into the `Get-Unique` cmdlet, which returns a list of distinct values that is stored in `$projects`.

Then, we need to load the list of configuration name/value pairs for each project and create a property on the `psobject` that contains a hash table of the related name/value pairs. Let's look at the code that does this:

```
foreach ($project in $projects)
    {
        $configlist = @{}

        foreach ($item in $configvals)
        {
            if ($item.project -eq $project.project )
               { $configlist.Add($item.name, $item.value)  }
        }

        #  Add the noteproperty with the configuration hash table as the value.
        $config |
        Add-Member -MemberType NoteProperty -Name $project.project -Value $configlist
    }
```

The outer `foreach` loop iterates over the list of projects. For each project, a hash table named `$configlist` is created. Then, an inner `foreach` loop iterates over the name/value pairs. If the inner loop's project has the same value as the outer loop, the name/value pair is added to the `$configlist` hash table. When the inner loop is finished, the hash table will contain all the name/value pairs for the project in the outer loop. It is then added as a property, with the current project's name, to the `$config` object using the `Add-Member` cmdlet. The outer loop will repeat until it has processed all the projects in the file. At that point, the outer loop is exited and the `$config` object is returned to the caller as in the following statement:

```
Return $config
```

To use the function, we need to store the path to the configuration file in a machine-level environment variable. To create this variable, we must start PowerShell as Administrator and enter the following statement. Note: We are using the same environment variable name as before, but the file has a `csv` extension this time rather than `txt`. It is a different file.

```
[Environment]::SetEnvironmentVariable("psappconfigpath",
"c:\psappconfig\psappconfig.csv", "Machine")
```

Then, we need to edit the `AllUsersAllHosts` profile by entering the following statement:

```
Notepad.exe $profile.AllUsersAllHosts
```

We need to add the lines that follow to the profile:

```
Import-Module umd_etl_functions

$global:psappconfig = Set-UdfConfiguration ($env:psappconfigpath)
```

Save the file and exit. Then, exit and restart PowerShell so the new profile will execute. Once back in PowerShell, enter the statement that follows to see the new configuration object:

```
$global:psappconfig | Format-List
```

You should see the following output:

```
general : {emailserver, ftppath, edwserver}
edw     : {teamemail, stagingdb}
finance : {dbserver, outfolder, ftppath}
```

We can access just the values associated with a project by using the appropriate property name, as shown here:

```
$global:psappconfig.edw
```

You should see just the configuration values that you see here:

```
Name                           Value
----                           -----
teamemail                      edwteamdist
stagingdb                      staging
```

To access an individual configuration property, just add the configuration name after the project name, separated by a period, as shown here:

```
$global:psappconfig.edw.teamemail
```

We can access the other projects in the same way, i.e., with statements like these:

```
$global:psappconfig.finance

$global:psappconfig.finance.dbserver

$global:psappconfig.finance.ftppath
```

Now that the configuration object is created, we can use it in code:

```
Import-Module umd_database -Force
$result = Invoke-UdfSQLQuery -sqlserver $global:psappconfig.finance.dbserver   `
                             -sqldatabase $global:psappconfig.finance.dbname   `
                             -sqlquery "select top 10 * from person.person;"
```

This code calls the function `Invoke-UdfSQLQuery` from the `umd_database` module to run a simple query against the finance database. To change the server or database name, we would just need to edit the values in the file pointed to by the environment variable `$env:psappconfigpath`. Admittedly, the calling syntax is a bit verbose. We could simplify it by assigning the environment variables to local variables in our scripts. If this is to be done, it is best to do so at the top of the script so it is visible to developers who read the code.

Storing Configuration Values in a SQL Server Table

A nice option in SQL Server Integration Services (SSIS) configurations is the ability to store configuration settings in a database table. When the package executes, SSIS automatically retrieves the configuration entries and loads the values into the package variables as specified by the developer. The configuration table has to be defined with specific columns and formats. The table includes a `tag` column that groups configuration settings. Let's discuss how we can do the same thing in PowerShell. This is very similar to the prior approach, except we will be loading the configuration values from a SQL Server table. First, we need to create environment variables that store the connection properties to the configuration table. Start PowerShell as Administrator and execute the statements in Listing 10-8.

Listing 10-8. Configuration pointing to SQL Server

```
#  Create new machine-level environment variables to point to
#  the database configuration table.
[Environment]::SetEnvironmentVariable("psappconfigdbserver", "(local)", "Machine")

[Environment]::SetEnvironmentVariable("psappconfigdbname","Development", "Machine")

[Environment]::SetEnvironmentVariable("psappconfigtablename","dbo.PSAppConfig", "Machine")

Get-Process explorer | Stop-Process #  Line to address Windows bug
```

Here, we create environment variables to store the database server, the database name, and the table where the configuration entries are stored. After running these statements, these machine-level variables will be permanently stored on the machine.

In this listing, we are defining the name of the configuration table, `psappconfigtablename`, as `dbo.PSAppConfig`. As with SSIS, we need to define this in a very specific manner. To create the SQL Server table in your environment, you will need a SQL Server client like SQL Server Management Studio (SSMS) so you can execute the required SQL statements. The SQL in Listing 10-9 will create the table for us.

Listing 10-9. The configuration table

```
CREATE TABLE [dbo].PSAppConfig
(
        [Project]     varchar(50),
        [Name]        varchar(100),
        [Value]       varchar(100),
        [CreateDate]  datetime  default(getdate()),
        [UpdateDate]  datetime  default(getdate()),
        Primary Key (Project, Name)
)
```

We really just need the `Project`, `Name`, and `Value` columns. The `CreateDate` will store the date and time the row was inserted, and the `UpdateDate` is meant to hold the last date and time the row was updated. The date columns come in handy when there is a need to know when data was created or last modified. Notice the primary key is the `Project` and `Name` columns concatenated. This is to prevent the insertion of rows with duplicate `Project` and `Name`.

Let's insert some configuration rows with the SQL statements in Listing 10-10.

Listing 10-10. Loading PSAppConfig

```
insert into [dbo].PSAppConfig ([Project], [Name], [Value])
Values ('general','edwserver','(local)');

insert into [dbo].PSAppConfig ([Project], [Name], [Value])
Values ('general','emailserver','smtp.live.com');

insert into [dbo].PSAppConfig ([Project], [Name], [Value])
Values ('general','ftppath','\\ftpserver\ftp\');

insert into [dbo].PSAppConfig ([Project], [Name], [Value])
Values ('edw','teamemail','edwteamdist');

insert into [dbo].PSAppConfig ([Project], [Name], [Value])
Values ('edw','stagingdb','staging');

insert into [dbo].PSAppConfig ([Project], [Name], [Value])
Values ('finance','ftppath','\\financeftpserver\ftp\');

insert into [dbo].PSAppConfig ([Project], [Name], [Value])
Values ('finance','dbserver','financesqlsvr');

insert into [dbo].PSAppConfig ([Project], [Name], [Value])
Values ('finance','outfolder','\\somepath\outdata\');
```

Once we have the SQL Server configuration table populated, we need code that will load it into PowerShell. The function in Listing 10-11 is found in the `umd_etl_functions` module. The function will load the configuration settings for us. Note the statement above the function definition imports the `umd_database` module, because the function calls the `Invoke-UdfSQLQuery` function, which is in that module.

Listing 10-11. Loading configuration settings from a database table

```
Import-Module umd_database

function Set-UdfConfigurationFromDatabase
{
 [CmdletBinding()]
        param (
                [string] $sqlserver,
                [string] $sqldb,
                [string] $sqltable
              )

    [psobject] $config = New-Object psobject

    $projects = Invoke-UdfSQLQuery -sqlserver $sqlserver `
                                   -sqldatabase $sqldb   `
                                   -sqlquery "select distinct project from $sqltable;"

    $configrows = Invoke-UdfSQLQuery -sqlserver $sqlserver -sqldatabase $sqldb `
                  -sqlquery "select * from $sqltable order by project, name;"

    foreach ($project in $projects)
    {
        $configlist = @{}

        foreach ($configrow in $configrows)
        {
            if ($configrow.project -eq $project.project )
                  { $configlist.Add($configrow.name, $configrow.value)  }
        }

        #  Add the noteproperty with the configuration hash table as the value.
        $config |
        Add-Member -MemberType NoteProperty -Name $project.project -Value $configlist
    }

     Return $config
}
```

The function in Listing 10-11 takes three parameters, which are the name of the SQL Server, the database name, and the table in which the configuration data is stored. Aside from the source of the data, this function is similar to the function that loads the configuration data from a CSV file. The first statement in the function, copied here, creates a `psobject` to hold the configuration information:

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

Then, the code gets a list of distinct `project` values using the function `Invoke-UdfSQLQuery` and storing the result in `$projects`:

```
$projects = Invoke-UdfSQLQuery -sqlserver $sqlserver `
                               -sqldatabase $sqldb   `
                               -sqlquery "select distinct project from $sqltable;"
```

The statement here loads the configuration table rows into `$configrows`:

```
$configrows = Invoke-UdfSQLQuery -sqlserver $sqlserver -sqldatabase $sqldb `
                                 -sqlquery "select * from $sqltable order by project, name;"
```

Then, the code that follows loads the `psobject $config` with a property for each project containing the related name/value pairs:

```
foreach ($project in $projects)
{
    $configlist = @{}

    foreach ($configrow in $configrows)
    {
        if ($configrow.project -eq $project.project )
              { $configlist.Add($configrow.name, $configrow.value)  }
    }

    #  Add the noteproperty with the configuration hash table as the value.
    $config | Add-Member -MemberType NoteProperty -Name $project.project -Value $configlist
}
```

The outer `foreach` loop iterates over the list of projects. For each project, a hash table named `$configlist` is created. Then, an inner `foreach` loop iterates over the name/value pairs. If the inner loop's project has the same value as the outer loop, the name/value pair is added to the `$configlist` hash table. When the inner loop is finished, the hash table will contain all the name/value pairs for the project in the outer loop. It is then added as a property with the current project's name to the `$config` object using the `Add-Member` cmdlet. The outer loop will repeat until it has processed all the projects in the file. At that point, the outer loop is exited and the `$config` object returned to the caller via the statement here:

```
Return $config
```

We can execute the function with a statement like the one here:

```
$global:psappconfig = Set-UdfConfigurationFromDatabase '(local)' `
'Development' 'dbo.PSAppConfig'
```

Once we have executed the function, we can verify the configuration object has the configuration data with the statements that follow:

```
$global:psappconfig | Format-List

$global:psappconfig.edw

$global:psappconfig.finance
```

To demonstrate how to use the configuration object, let's use the connection object from `umd_database`, as shown in Listing 10-12.

Listing 10-12. Using configurations from a database table

```
Import-module umd_database

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

$myconnection.ConnectionType = 'ADO'
$myconnection.DatabaseType = 'SqlServer'
$myconnection.Server = $global:psappconfig.finance.dbserver
$myconnection.DatabaseName = $global:psappconfig.finance.dbname
$myconnection.UseCredential = 'N'
$myconnection.SetAuthenticationType('Integrated')
$myconnection.BuildConnectionString()
$myconnection.RunSQL("select top 10 * from [Sales].[SalesTerritory]", $true)
```

We create a `psobject`, which is passed by reference to the `New-UdfConnection` function. The function attaches the methods and properties to support database access. We set the various properties and finally execute a `select` statement on the last line, which should list ten rows from the `Sales.SalesTerritory` table in the `AdventureWorks` database.

Using a Script Module to Support Configuration

There are a couple of criticisms that can be made about the previous configuration solutions. First, although they mitigate the risk of code modifying the configuration variables, they do not eliminate the possibility. Second, application developers may not want to or may not have the access to create environment variables and profile scripts on the server. Another approach that can be modified to incorporate the best features of the previously discussed configuration solutions is to use a script module. The idea is to create a script module with the purpose of returning configuration settings. Let's assume our configuration settings are in a text file, as shown in Listing 10-13.

Listing 10-13. Configuration text file

```
emailserver=smtp.live.com
emailaccount=someacct@msn.com
emailfrom=someemail@msn.com
emailpw=somepassword
edwteamemail=edwteam@msn.com
emailport=587
ftpinpath=\\\\ftp\\inbound\\
ftpoutpath=\\\\ftp\\outbound\\
edwserver=(local)
```

In this listing, notice the file is set up as name/value pairs that are perfect for loading into a hash table. Note: This file is also stored under the name `psappconfig.txt` with the code listings so that the function in Listing 10-14 can find it. Now, let's take a look at the code that will retrieve the configuration values.

Listing 10-14. A script module named umd_appconfig that gets configuration settings

```
$configfile = Join-Path -Path $PSScriptRoot -ChildPath "psappconfig.txt"
$configdata = Get-Content -Path $configfile | ConvertFrom-StringData

function Get-UdfConfiguation ([string] $configkey)
{
   Return $configdata.$configkey
}
```

The first line in Listing 10-14 uses `Join-Path` to concatenate the PowerShell variable `$PSScriptRoot` with the name of the configuration text file. We could just do a string concatenation, but many PowerShell developers prefer to use the cmdlets specifically designed for this purpose, i.e., `Join-Path` to build the path and `Split-Path` to break the path into parts. When the module gets loaded, `$PSScriptRoot` will have the path where the function or module is stored. Therefore, we need to place the configuration text file in the folder where the module script file is located. The nice thing about this is that we don't have to tell PowerShell where to find the file. The module will be stored in a script file named `umd_appconfig.psm1` in folder named `umd_appconfig`, which is located in a folder where PowerShell knows to look for modules, such as `\Documents\WindowsPowerShell\Modules`.

The second line in Listing 10-13 loads the configuration text file into a hash table variable named `$configdata`. Rather than have developers directly access variables in the module, the function `Get-UdfConfiguration` will return the value associated with any key passed into the `$configkey` parameter. An example of using this module is provided in Listing 10-15.

Listing 10-15. Using the configuration script module

```
Import-Module -Name umd_appconfig  #  Must call this to load configurations

Import-Module -Name umd_etl_functions

$Params = @{
    smtpServer = (Get-UdfConfiguation 'emailserver')
    from = (Get-UdfConfiguation 'emailfrom')
    to = (Get-UdfConfiguation 'edwteamemail')
    subject = 'Important Message'
    body = 'ETL Job executed'
    port = (Get-UdfConfiguation 'emailport')
    usecredential = ''
    emailaccount = (Get-UdfConfiguation 'emailaccount')
    password = (Get-UdfConfiguation 'emailpw')
}
Send-UdfMail @Params
```

The first line in Listing 10-15 imports the `umd_appconfig` module, which loads the configuration values and defines the function to retrieve them. Then, the `umd_etl_functions` module is imported, because we need to use the `Send-UdfMail` function. As we saw previously, we load the function parameters using a technique called *splatting*. The call to `Get-UdfConfiguration` must be enclosed in parentheses so that the function will be executed first and the returned value will be assigned to the variable on the left. Although the example shown here uses a text file, it could easily be modified to get the values from a CSV or SQL Server table. This approach is the easiest to deploy and offers the most protection of the configuration values. This makes it well suited to large environments with many servers.

PowerShell Development Best Practices

The best practices for any development language should be similar, because they have the same basic goals. These goals include application stability, ease of maintenance, and simplicity of deployment. This book treats PowerShell as a development platform. It may be unique in that respect, and as such, the best practices here may be different than what is presented elsewhere. Other best practices lists I've seen tend to treat PowerShell as an administrative tool, which sometimes negates the goals of professional application development.

Application stability is greatly affected by the degree of dependencies among the components. For example, if a script shares variables with another script, they have a dependency, which means a change to one of the scripts could break the other. Making applications modular reduces this fragility, because modularity means each component can stand alone.

Ease of maintenance covers a number of different concepts. One is the readability of the code. Undescriptive variable and function names, cramming multiple statements on a single line, and lack of indenture to indicate nesting levels of code all undermine readability. Using white space and descriptive names can help.

Another issue for maintainability is the level of hard-coded values in the application. If file paths, server names, login accounts, and so on are hard coded, then the application needs to be revised when it is deployed to another environment. Even in an existing environment where the application is installed, should one of the values need to be changed, the code will need to be revised and tested. Having support within the application to change such values without changing the code is critical to maintainability and deployment.

Maintainability is also affected by the level of reuse. It is better to have a single, flexible function used by many components to accomplish a task than to have many different identical or slightly different functions to perform the same tasks. Otherwise, a required change in functionality may require modifying and testing many functions instead of just one. For example, if many different versions of a logging function were implemented, a change to add a new piece of data may require making a change to multiple functions.

A reason why developers reinvent the wheel by creating duplicate functions is the lack documentation on existing functions. Developers often feel it is safer to write their own functions. Programmers generally do not like to write documentation. Therefore, documentation needs to be integrated into the development process.

The best practices that follow are meant to address these issues. It is not realistic to expect that all the best practices will be implemented at all times. However, applying them as a standard practice will save enterprises considerable time and money. Not everyone will agree with all the best practices listed. However, I recommend that the reader reflect less on the specifics of the best practice recommendation than on the spirit of it. In that way, you can achieve the benefits while customizing the specifics to fit your preferences.

Function Naming

PowerShell function (also called cmdlet) naming is not arbitrary. Microsoft has established naming standards that are generally adhered to and accepted as a best practice by the PowerShell community. Microsoft recommends naming functions in the format `Verb-Noun`, where the verb conforms to an approved list and the noun describes what the verb acts upon. The list and description of approved verbs can be found at: `https://msdn.microsoft.com/en-us/library/ms714428%28v=vs.85%29.aspx`. It is important that the developer choose the verb that most closely matches what the function does so developers can easily grasp its purpose—and so that when it is listed using PowerShell cmdlets like `Get-Command`, it will be listed with similar functions. The ability to easily find functions by what they do is called *discoverability*. It is important that developers name the function in a way that avoids the risk of duplicating a built-in or third-party function. Otherwise, the wrong function may get called by the application, resulting in a bug that can be difficult to track down. Consider the case where a developer writes a function he calls `Get-OData` that is used by many parts of the application. OData is an open data-retrieval standard promoted by Microsoft. What happens if Microsoft adds a new cmdlet named `Get-OData` to PowerShell? PowerShell will use the custom

function in place of the built-in cmdlet, but that just happens to be the way it is designed, and that could change. Even so, developers that maintain the code might think that the calls to `Get-OData` are calling the Microsoft cmdlet. If there is a bug, developers may be trying to understand why the parameters to `Get-OData` do not match Microsoft's supported parameters. They may even think there is a bug in the Microsoft cmdlet, not realizing there is a custom version. In the support of code maintainability, it is critical to make it clear what code is system delivered and what is custom code. In this book, it is clear what functions are custom versus built-in due to the naming standard used. Unlike the verb, the noun does not have to comply with an approved list. So, we can use a noun that makes it clear that the function is a custom one. In this book, the noun prefix `Udf` is used for this purpose, i.e., `Udf` indicates the function is user defined. So, in our example, we could name the function `Get-UdfOData`. This accomplishes the goal of readability and avoids naming collisions, while adhering to PowerShell naming standards.

File Naming

Unless a function is contained in a script module, a good practice is to store it in a script file of the same name as the function, i.e., `Add-UdfConnection.ps1`. This makes it easier to locate the function. As much as possible, I try to write code as functions because they maximize reusability. However, when I do create script files, I like to prefix them with `scr`, an abbreviation for *script*. I like to name script modules in a way that is very dissimilar to what Microsoft and third parties tend to use. This makes it readily clear that it is a custom module. I prefix the module name with `umd`, which stands for user-defined module, and I separate the words with underscores. I provide examples of each file type name.

For a script containing a function definition:

```
Invoke-UdfSomtTask.ps1
```

For a script that loads sales data:

```
scr_do_sales_load.ps1
```

For a module containing ETL-related functions:

```
umd_etl_functions.psm1
```

Scripts are used to call functions. Although they can call other scripts, I find limiting scripts to fall-through code that just calls reusable functions to be more manageable.

In my development, I have found creating separate folders for each type of code helpful. A typical folder hierarchy is shown here:

```
Directory: C:\Users\BryanCafferky\Documents\PowerShell

OMode                LastWriteTime     Length Name
----                 -------------     ------ ----
d----          7/30/2014  11:33 AM            documentation
d----          3/13/2015   7:47 PM            function
d----         12/28/2014  10:18 PM            module
d----          8/24/2014  11:27 AM            script
```

Notice that there is a documentation folder. This is where documentation on the custom scripts and functions and overall applications can be kept. Also, using the `Write-Verbose` cmdlet within a function can provide additional information to developers about the function.

Make Code Reusable

At the start of development, it is tempting to write quick and dirty scripts that get the job done but are not created in a manner that maximizes reusability. This can lead to many specialized scripts that are all doing a similar task. The problem is that all these scripts have to be maintained, and if a change is needed in one, it is probably needed in the others. I make it a rule to write a function for anything I do that has the potential for reusability. It does not take much longer to add the function header with parameters, and it saves time in the long run. If there are more than a few related functions frequently being called, consider making them into a script module, which makes the functions easier to access and simplifies deployment.

Developers will only use functions that they understand how to call. Like Java, PowerShell supports special documentation tags embedded in the source code. The cmdlets like `Get-Help` can extract these tags to provide help to developers about functions. A good practice is to use a script and function template that has the basic tags in place so developers can just change the function name, fill in the tags, and add the code for the function. No one likes to go back and edit code that is already written, so it is better to start with these tags from the beginning. Another good practice is to use the `CmdletBinding` attribute, because it adds support for the common cmdlet parameters, parameter validation, and parameter sets. The sample template in Listing 10-16 provides a good starting point for creating a new function. The idea is to just load the template script into the ISE, save it under the name of the new function, and edit the function name and tag values to fit your needs.

Listing 10-16. An example of a function template

```
function Verb-UdfNoun
{
            <#
            .SYNOPSIS
            Enter a short description of what the function does.

            .DESCRIPTION
            Enter a longer description about the function.

            .PARAMETER Parm1
            Describe the first parameter.

            .PARAMETER Parm2
            Describe the second parameter.

            .INPUTS
            Can data be piped into this function?

            .OUTPUTS
            Describe what this function returns.

            .EXAMPLE
            C:\PS> Verb-UdfNoun -parm1 "parm1" -parm2 "parm2"

            .LINK
            Enter a url to a help topic if available.

            #>
```

```
 [CmdletBinding()]
 param
 (
    [string]      $parm1,
    [string]      $parm2
 )

#  Insert code below...

}
```

An alternative to using a script as a template is to create a snippet that can easily be inserted into code in the ISE. Details on this can be found at:

https://msdn.microsoft.com/en-us/library/ms714428%28v=vs.85%29.aspx

Use Source Code Control

Database development code, such as ETL, stored procedures, functions, SQL Agent Jobs, and queries, is often overlooked when it comes to source code control. Sometimes code by database developers is not seen as critical code requiring source code control. Other times, database backups are relied upon to serve that function. Source code control is a good practice for all professional application development. Bear in mind that PowerShell code is not stored in the database. Even if PowerShell's usage is limited to preparing data to be loaded and archiving files after they have been loaded, it is still critical to the application. There are many great source code control systems available. I had a client who had no source code control in place, and I did not want to add the infrastructure maintenance of installing one. I selected GitHub because it came with a nice GUI client front end and automatically backed the source up to the cloud for a very inexpensive price. Svn is a good open-source option, and Team Foundation Services (TFS) is also good. The goal is to protect the source code and enable an easy retrieval of any historic version, should it be needed.

Build In Configurations from the Start

One of the most important best practices is to avoid hard coding values that are likely to change. Server names, service accounts, passwords, database names, file paths, and things like this are environment specific. Often they are outside the developer's control. Setting these up as configurable values from the start makes it easier to deploy the application to another environment. If we do this, moving the application from development to production is just a matter of copying files and editing the configuration values. The point is this: it is better to think out what the configurable parameters for your application will be before you start coding. That way, you can code your function calls using the configured parameters, as we saw earlier in this chapter, and you will be ready to deploy once your application is tested. It is tempting to jump in and code with the idea that you can add the configurable values later. That means you will have to go back and edit code a second time and risk making a mistake that could introduce bugs. This is the lesson I learned in developing SSIS projects, and it applies equally to PowerShell. Earlier in this chapter, we discussed a number of approaches that can be applied to achieve maximum configurability.

PowerShell Application Deployment

Three things must be considered when deploying a PowerShell application: script modules, script files, and jobs. The next subsections go over these in the order listed. Let's start with deploying script modules.

Deploying Script Modules

Deploying PowerShell applications is eased by the fact that it is an interpreted language. There are no DLLs or assemblies to install unless your application makes calls to such external programs. Deployments consist of copying the PowerShell scripts and configuration files to the target machine and modifying the configuration settings as appropriate. The use of script modules can simplify deployments, since PowerShell will automatically locate the module when called. However, the current requirement that all the functions of the module be contained in one script file violates the best practice of modularity. To make a change to a single function, the script, potentially containing numerous functions, must be modified and deployed. If an error is made, the other functions could be affected. It also makes locating the function more difficult, because the developer must search for the function name within the module script. In Chapter 6, a method of overcoming this problem was discussed. Here, I want to provide another method by which to retain the benefits of using modules while keeping the code modular. The idea is to store each of the module's functions in a separate script file. When the module is imported, the module will automatically find the functions and dot source them into memory. Let's look at code in Listing 10-17 that does this.

Listing 10-17. A module that dot sources the functions

```
<#

.Author
   Bryan Cafferky
.SYNOPSIS
    A simple module that uses dot sourcing to load the functions.
.DESCRIPTION
    When this module is imported, the functions are loaded using dot sourcing.

#>

#  Get the path to the function files...
$functionpath = $PSScriptRoot + "\function\"

# Get a list of all the function file names...
$functionlist = Get-ChildItem -Path $functionpath -Name

#  Loop over alll the files and dot source them into memory..
foreach ($function in $functionlist)
{
    . ($functionpath + $function)
}
```

In Listing 10-17, the first non-comment line assigns `$functionpath` with the path to where the module script is located concatenated with the subfolder "`\function`". Recall that `$PSScriptRoot` is a built-in PowerShell variable that gets set to the path of the currently executing code. When the module is imported, `$PSScriptRoot` is set to the folder where the module is located. In this case, the module's functions are contained in a subfolder named `function`. The next line in the script calls the `Get-ChildItem` cmdlet, passing `$functionpath` and specifying that only the `Name` property be returned. This list is assigned to the variable `$functionlist`. Then a `foreach` loop iterates over the `$functionlist` collection and issues a dot-source statement for each script file. Recall the use of parentheses, which tell PowerShell to resolve the expression in parentheses before doing the outer command, i.e., dot sourcing the returned file name.

Using the script module in Listing 10-17, we can place all our functions in separate script files in the `function` subfolder. If a new script containing a function is added to the folder, it will automatically get loaded when the module is imported. If a script is deleted from the folder, it will no longer get loaded on an import. If any function needs to be modified, only the specific script file for the function needs to be edited. It also means that if one function needs to be modified, the developer only needs to check out the single function. Another developer can work on a different function at the same time without being concerned about merging the other developer's changes in later. This improves maintainability and decreases the risk of introducing bugs.

Let's see how this works. Create a new folder in the `\Documents\WindowsPowerShell\Modules\` folder named `umd_application`. Then, copy the script in Listing 10-17 to that folder as `umd_application.psm1`. Now, create a folder named `function` within the `umd_application` folder. Copy each of the functions that follow in Listings 10-18 to 10-20 into the script file as specified in those listings.

Listing 10-18. File name: Invoke-UdfAddNumber.ps1

```
function Invoke-UdfAddNumber([int]$p_int1, [int]$p_int2)
{
     Return ($p_int1 + $p_int2)
}
```

Listing 10-19. File name: Invoke-UdfSubtractNumber.ps1

```
function Invoke-UdfSubtractNumber([int]$p_int1, [int]$p_int2)
{
     Return ($p_int1 - $p_int2)
}
```

Listing 10-20. File name: Invoke-UdfMultiplyNumber.ps1

```
function Invoke-UdfMultiplyNumber([int]$p_int1, [int]$p_int2)
{
     Return ($p_int1 * $p_int2)
}
```

Now, start PowerShell and enter the following statement:

```
Import-Module umd_application
```

Assuming you did everything correctly, the functions should get loaded. To test this, enter the statements seen here:

```
Invoke-UdfAddNumber 5 6

Invoke-UdfSubtractNumber 5 2

Invoke-UdfMultiplyNumber 5 2
```

If these were executed as a script, the output should be:

```
11
3
10
```

An extra benefit of this approach is that we can just copy the same script module for any module we want to create and then place all the functions we want it to contain into the `function` subfolder.

Deploying Script Files

Script files can be stored in a folder under an account that will be used to execute the scripts, or in a common location; it depends on how you want to configure the application. I recommend putting as much of the code into modules as possible and limiting script files to calling the module functions to do work. This approach was demonstrated in Chapter 9. However, that example did not use configurations, which we would want to do. Listing 10-21 is the script from Chapter 9 revised to use a configuration approach that loads the configurable values into PowerShell global variables.

Listing 10-21. Using configuration settings from global PowerShell variables

```
Import-Module umd_etl_functions

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

# Notify users the job has started using Outlook...
Send-UdfMail  -Server $global:mailserver -From $global:fromemail -To $global:toemail `
     -Subject "ETL Job: Sales Load has started" –Body "The ETL Job: Sales Load has started." `
     -Port $global:emailport -usecredential $global:emailaccount –Password $global:emailpw

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

$filelist = Get-ChildItem $global:zippath

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

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

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

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

# Notify users the job has finished...
Send-UdfMail  -Server $global:mailserver -From $global:fromemail -To $global:toemail `
               -Subject "ETL Job: Sales Load has ended" -Body "The ETL Job: Sales Load has
                ended." `
               -Port $global:emailport -usecredential $global:emailaccount -Password
$global:emailpw
```

This script is similar to the one in Chapter 9, but here all parameters that might change are stored in globally scoped PowerShell variables; note the `$global` prefix. The assumption here is that these variables were assigned values previously, probably in the profile script. Moving this script to another machine involves copying the script, the module it uses, and the PowerShell script that loads the configuration values, and then modifying the configuration values as appropriate for the new environment. Note: If you want to run the script in Listing 10-21, you need to review all the configuration variable settings to make sure they will work in your environment.

Deploying Jobs

SQL Server Agent provides an excellent place to create PowerShell jobs. SQL Server Management Studio (SSMS) even provides a feature that allows us to generate a script that can be used to copy the job to another server.

To script out a SQL Agent job, start SSMS, connect to the database with the job you want to script, and locate the job. Then right mouse click on the job and select Script Job as, Create to, and New Query Editor Window, as shown in Figure 10-2.

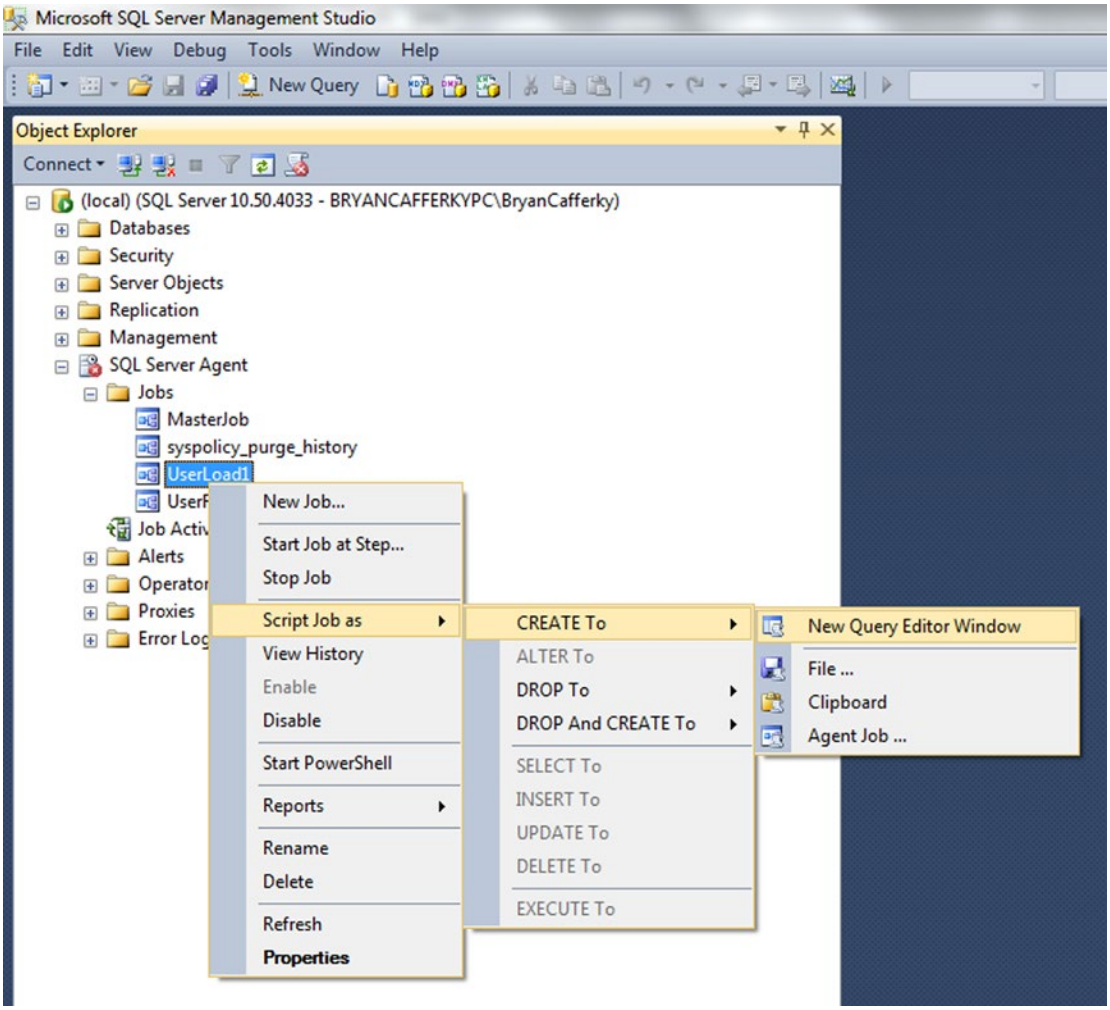

Figure 10-2. *Scripting a SQL Agent Job in SQL Server Management Studio*

The script will be displayed in a new query tab, as shown below in Figure 10-3.

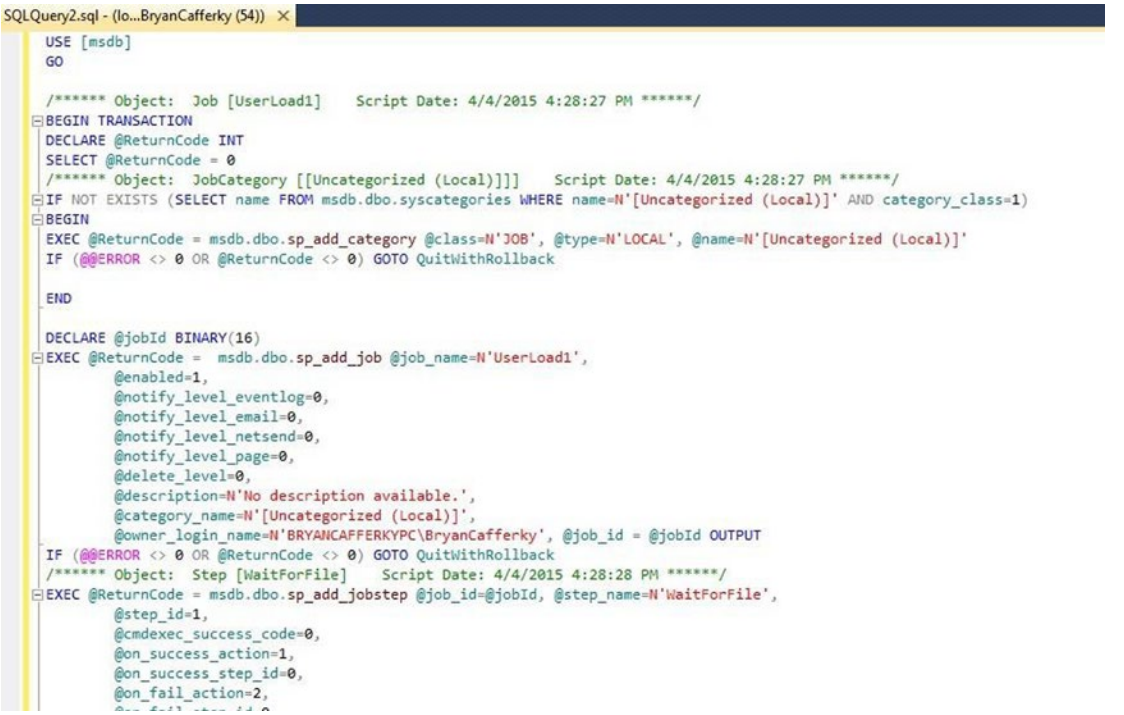

Figure 10-3. *A SQL Agent job-creation script*

From there, just connect to the server you want to copy the job to and execute the script. Of course, you will need to copy any scripts the job calls to the target server as well. Note: If you just want to clone the job on the same server, change the variable `@job_name` in the script to a job name that does not exist on the server.

Using SQL Agent as the job scheduler makes it easy to deploy your application. However, PowerShell has its own built-in scheduler, and the Windows Task Scheduler is also an option. We will discuss these in more detail later.

Summary

In this chapter, we discussed configuring PowerShell applications, best practices, and application deployment. We started with how to configure PowerShell applications to maximize flexibility, ease of maintenance, and simplicity of deployment. A nice feature of SSIS is that it has a number of methods to support configuration. We discussed how to achieve comparable functionality in PowerShell, considering several approaches. Then, we discussed the goals of best practices and how to apply them to PowerShell. We closed by discussing PowerShell application deployment and ways to simplify this process.