

CHAPTER 7



Accessing SQL Server

We have seen some examples of executing queries against SQL Server. There are many ways to access a database, and it can hinder development to custom code each variation, as there are a number of questions to be resolved. What type of access method will be used: ADO.Net, OleDb, ODBC, or the native SQL Server driver? What type of authentication will be used: Windows Integrated Security, Logon ID and password, or a credential object? What database platform: On Premise SQL Server, SQL Azure, or a third-party database like MySQL or PostgreSQL? Why not write some functions to make this task easier? In this chapter, we will be carefully walking through a custom module that provides easy access to databases using ADO.Net.

About the Module

The module, `umd_database`, centers around a function that returns a custom database-connection object that is built on PowerShell's object template, `PSObject`. By setting the object's properties, we tell it how to execute SQL statements to the database. Using a method called `RunSQL`, we will run various queries. To run stored procedures that require output parameters, we can use the `RunStoredProcedure` method. Since an object is returned by the `New-UdfConnection` function, we can create as many database-connection object instances as we want. Each instance can be set for specific database-access properties. For example, one instance might point to a SQL Server database, another might point to an Azure SQL database, and yet another might point to a MySQL database. Once the connection attributes are set, we can run as many queries as we want by supplying the SQL. Beyond providing a useful module, the code will demonstrate many techniques that can be applied elsewhere. Using what we learned in the previous chapter, we copy the module `umd_database` to a folder named `umd_database` in a directory that is in `$env:PSModulePath`, such as `\WindowsPowerShell\Modules\`, under the current user's Documents folder. To run the examples that use SQL Server, we will need the Microsoft training database AdventureWorks installed. You can get it at <http://msftdbprodsamples.codeplex.com/>.

Using umd_database

Before we look at the module's code, let's take a look at how we can use the module. The idea is to review the module's interface to make it easier to understand how the code behind it works. Let's assume we want to run some queries against the SQL Server AdventureWorks database. We'll use the local instance and integrated security, so the connection is pretty simple. Since code using ADO.Net is so similar to code using ODBC, we will only review the ADO.Net-specific code in this chapter. However, you can also see in the module the ODBC-specific coding. Listing 7-1 shows the code to create and define the connection object. Note: You will need to change the property values to ones appropriate for your environment.

Listing 7-1. Using the `umd_database` module

```

Import-Module umd_database

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

$myssint.ConnectionType = 'ADO'
$myssint.DatabaseType = 'SqlServer'
$myssint.Server = '(local)'
$myssint.DatabaseName = 'AdventureWorks'
$myssint.UseCredential = 'N'
$myssint.SetAuthenticationType('Integrated')
$myssint.AuthenticationType

```

In Listing 7-1, the first thing we do is import the `umd_database` module to load its functions. Then we create an instance of a `PSObject` named `$myssint`. Next, a statement calls `New-UdfConnection` to attach a number of properties and methods to the object we created. Note: The parameter `$myssint` is being passed as a `[REF]` type. As we saw previously, a `REF` is a reference to an object, i.e., we are passing our object to the function where it is modified. The lines that follow the function call set the properties of the object. First, we set the `ConnectionType` to `ADO`, meaning `ADO.Net` will be used. The `DatabaseType` is `SQL Server`. The server instance is `'(local)'`. Since we are using integrated security, we will not be using a credential object, which is a secure way to connect when user ID and password are being passed. We just set `UseCredential = 'N'`. Finally, we set the `AuthenticationType` to `'Integrated'`, i.e., integrated security. Notice that the authentication type is being set by a method rather than by our just assigning it a value. This is so the value can be validated. An invalid value will be rejected. With these properties set, we can generate the connection string using the `BuildConnectionString` method, as shown here:

```
$myssint.BuildConnectionString
```

If there is a problem creating the connection string, we will get the message `'Error - Connection string not found!'`. Technically, we don't have to call the `BuildConnectionString` method, because it is called by the method that runs the SQL statement, but calling it confirms a valid connection string was created. We can view the object's `ConnectionString` property to verify it looks correct using the statement here:

```
$myssint.ConnectionString
```

We should see the following output:

```
Data Source=(local);Integrated Security=SSPI;Initial Catalog=AdventureWorks
```

Let's try running a query using our new connection object:

```
$myssint.RunSQL("SELECT top 20 * FROM [HumanResources].[EmployeeDepartmentHistory]", $true)
```

We use the object's `RunSQL` method to execute the query. The first parameter is the query. The second parameter indicates whether the query is a select statement—i.e., returns results. Queries that return results need to have code to retrieve. When we run the previous statement, we should see query results displayed to the console.

We can look at all the object's properties and their current values by entering the object name on a line, as follows:

```
$myssint
```

We should get the following output:

```

ConnectionType      : ADO
DatabaseType        : SqlServer
Server              : (local)
DatabaseName        : AdventureWorks
UseCredential        : N
UserID              :
Password            :
DSNName             : NotAssigned
Driver              : NotAssigned
ConnectionString    : Data Source=(local);Integrated Security=SSPI;Initial
                      Catalog=AdventureWorks
AuthenticationType   : Integrated
AuthenticationTypes  : {Integrated, Logon, Credential, StoredCredential...}
```

Being able to display the object properties is very useful, especially for the connection string and authentication types. If there is a problem running the query, we can check the connection string. If we don't know what the valid values are for authentication types, the `AuthenticationTypes` property shows them. Notice `UserID` and `Password` are blank since they don't apply to integrated security.

Imagine we want to have another connection, but this time using a table in the SQL Azure cloud. Azure does not support integrated security or the credential object, so we'll use a login ID and password. We can code this as follows:

```

[psobject] $azure1 = New-Object psobject
New-UdfConnection ([ref]$azure1)
$azure1 | Get-Member

$azure1.ConnectionType = 'ADO'
$azure1.DatabaseType = 'Azure'
$azure1.Server = 'fxxxx.database.windows.net'
$azure1.DatabaseName = 'Development'
$azure1.UseCredential = 'N'
$azure1.UserID = 'myuserid'
$azure1.Password = "mypassword"

$azure1.SetAuthenticationType('Logon')
$azure1.BuildConnectionString()
```

With the connection properties set, we can run a query as follows:

```
$azure1.RunSQL("SELECT top 20 * FROM [HumanResources].[EmployeeDepartmentHistory]", $true)
```

Assuming you change the properties to fit your Azure database, you should see results displayed in the console.

Now that we have some connection objects, we can execute more queries, as follows:

```
$myssint.RunSQL("SELECT top 20 * FROM [HumanResources].[Department]", $true)
```

```
$azure1.RunSQL("SELECT top 20 * FROM [Person].[Person]", $true)
```

Each statement will return results. The connection is closed after getting the results, but the properties are retained so that we can keep running queries. The result data is not retained, so if we want to capture it, we need to assign it to a variable, as follows:

```
$myresults = $myssint.RunSQL("SELECT top 20 * FROM [HumanResources].[Department]", $true)
```

What if we want to run a query that does not return results? That's where the second parameter to the RunSQL method comes in. By setting it to `$false`, we are telling the method not to return results. For example, the following lines will execute an update statement using the first connection object we created:

```
$esql = "Update [AdventureWorks].[HumanResources].[CompanyUnit] set UnitName = 'NewSales2'
where UnitID = 3;"
$myssint.RunSQL($esql, $false)
```

Here we are assigning the SQL statement to `$esql`, which is passed to the RunSQL method.

How umd_database Works

The `umd_database` module consists of a set of functions and is designed so that it can easily be extended. As we can see from the examples that use the module, the core function, `New-UdfConnection`, is the one that returns the connection object to us. Because this function is so long, I'm not going to list all of it at once. However, you can see the entire function in the code accompanying this book, in the `umd_database` module script. Let's start with the following function header:

```
function New-UdfConnection
{
    [CmdletBinding()]
    param (
        [ref]$p_connection
    )
```

The first thing to notice in the function header is that the parameter is of type `[REF]`, which means reference. The caller passes an object reference to the function. This gives the function the ability to make changes to the object passed, which is how the connection object is customized with properties and methods. Before calling the `New-UdfConnection` function, the calling code needs to create an instance of a `PSObject` and then pass a reference to it, as shown in the example code here:

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

The function `New-UdfConnection` attaches properties and methods to the object passed in, as shown here:

```
# Connection Type...
$p_connection.value | Add-Member -MemberType noteproperty `
                                -Name ConnectionType `
                                -Value $p_connectiontype
```

To change the object passed to the function, we need to use the object's Value property, `$p_connection.value`. The `Add-Member` cmdlet adds properties and methods to the object. `MemberType` specifies what type of property or method we want to attach. In the previous code, we add a `noteproperty`, which is just a place to hold static data. The `Name` parameter assigns an externally visible name to the property. The `Value` parameter assigns an initial value to the property, and the `Passthru` parameter tells PowerShell to pass the property through the pipe. This same type of coding is used to create the following properties: `DatabaseType`, `Server`, `DatabaseName`, `UseCredential`, `UserID`, `Password`, `DSNName`, and `Driver`. Since the coding is identical except for `Name` and `Value`, I'm not going to list them all here.

`AuthenticationType` is handled a bit differently. To start, the property is created just like the other properties are. The code is as follows:

```
# AuthenticationType. Value must be in AuthenticationTypes.
$p_connection.value | Add-Member -MemberType noteproperty `
                                -Name AuthenticationType `
                                -Value 'NotAssigned' `
                                -Passthru
```

However, `AuthenticationType` will not be set by directly assigning a value to it. Instead, a method named `SetAuthenticationType` will be called. This is so we can validate the `AuthenticationType` value before allowing it to be assigned. So, when we call `SetAuthenticationType` on the object as in "`$myssint.SetAuthenticationType('Integrated')`", an invalid value will be rejected. Let's look at the code that stores the valid list of values as an object property:

```
$p_authenticationtypes = ('Integrated','Logon','Credential', 'StoredCredential',
'DSN', 'DSNLess')
```

```
$p_connection.value | Add-Member -MemberType noteproperty `
                                -Name AuthenticationTypes `
                                -Value $p_authenticationtypes
```

In this code, we are creating the array `$p_authenticationtypes` and loading it with values. Then, we use the `Add-Member` cmdlet to attach the array as a property named `AuthenticationTypes` to the object passed in to the function. Attaching the list to the object as a property enables the user to see what the valid values are.

Now, let's look at the code that sets the `AuthenticationType`:

```
$bauth = @'
param([string]$p_authentication)

if ($this.AuthenticationTypes -contains $p_authentication)
{
    $this.AuthenticationType = $p_authentication
}
Else
{
    Throw "Error - Invalid Authentication Type, valid values are " + $this.
    AuthenticationTypes
}

RETURN $this.AuthenticationType

'@
```

The here string is the script to be executed when the `SetAuthenticationType` is called. The only parameter is the authentication type to assign. To validate that the desired authentication type is valid, the code uses the object's `AuthenticationTypes` property. The special keyword `$this` refers to the current object—i.e., the object we are extending. The line “if (`$this.AuthenticationTypes -contains $p_authentication`)” is checking whether the value passed in is contained in the object's `AuthenticationTypes` hash table. If it is, the object `$this.AuthenticationType` is assigned the parameter passed in. If the value is not in the list, an error is thrown with a message telling the caller that the authentication type is not valid.

The code that follows attaches the script to the object as a script method named `SetAuthenticationType`:

```
$sauth = [scriptblock]::create($bauth)

$ps_connection.value | Add-Member -MemberType scriptmethod `
                                -Name SetAuthenticationType `
                                -Value $sauth `
                                -Passthru
```

Then first statement, “`$sauth = [scriptblock]::create($bauth)`”, converts the here string into a script block. Then the `Add-Member` cmdlet is used to attach the scriptblock to the object as a scriptmethod named `SetAuthenticationType`. The method takes one parameter—the value to set the `AuthenticationType`. You may be asking yourself, why is the code for the script method first defined as a here string and then converted to a script block? Why not just define it as a script block to start with? The reason is that a script block has limitations, including that it will not expand variables and it cannot define named parameters. By defining the code as a here string, which does support variable expansion, we can get the script block to support it too; i.e., when we convert it to the script block using the script block constructor `create` method. This technique gives our script methods the power of functions. I got this idea from a blog by Ed Wilson, the Scripting Guy:

<http://blogs.technet.com/b/heyscriptingguy/archive/2013/05/22/variable-substitution-in-a-powershell-script-block.aspx>

It is not always necessary to expand variables, so in some cases we could just directly define the script method as a script block. However, I prefer to code for maximum flexibility since requirements can change, so I make it a practice to always code script methods for my custom objects using this approach.

The next section of the `New-UdfConnection` function creates the script method `BuildConnectionString`, which creates the required connection string so as to connect to the database. Let's look at the code that follows:

```
$bbuildconnstring = @'
    param()

    If ($this.AuthenticationType -eq 'DSN' -and $this.DSNName -eq 'NotAssigned')
    { Throw "Error - DSN Authentication requires DSN Name to be set." }

    If ($this.AuthenticationType -eq 'DSNLess' -and $this.Driver -eq 'NotAssigned')
    { Throw "Error - DSNLess Authentication requires Driver to be set." }

    $Result = Get-UdfConnectionString $this.ConnectionType `
        $this.DatabaseType $this.Server $this.AuthenticationType `
        $this.Databasename $this.UserID $this.Password $this.DSNName $this.Driver

    If (!$Result.StartsWith("Err")) { $this.ConnectionString = $Result } Else {
    $this.ConnectionString = 'NotAssigned' }

    RETURN $Result
'@
```

A here string is assigned the code to build the connection string. The script does not take any parameters. The first two lines do some validation to make sure that, if this is an ODBC DSN connection, the object's `DSNName` property has been set. Alternatively, if this is an ODBC DSNLess connection, it verifies that the `Driver` property is set. If something is wrong, an error is thrown. There could be any number of validations, but the idea here is to demonstrate how they can be implemented. Again, to access the object properties, we use the `$this` reference. If the script passes the validation tests, it calls function `Get-UdfConnectionString`, passing required parameters to build the connection string, which is returned to the variable `$Result`. If there was an error getting the connection string, a string starting with "Err" is returned. `$Result` is checked, and if it does not start with "Err", i.e., "If (`!$Result.StartsWith("Err")`)"; the object's `ConnectionString` property is set to `$Result`. Remember, `!` means NOT. Otherwise, the `ConnectionString` is set to 'NotAssigned'. The function `Get-UdfConnectionString` is not part of the object. It is just a function in the module, which means it is a static function—i.e., there is only one instance of it.

The example shows how we can mix instance-specific methods and properties with static ones. `Get-UdfConnectionString` can also be called outside of the object method, which might come in handy if someone just wants to build a connection string. We'll cover the code involved in `Get-UdfConnectionString` after we've finished covering `New-UdfConnection`. Now, let's take a look at the code that attaches the script to the script method:

```
$sbuildconnstring = [scriptblock]::create($bbuildconnstring)

$p_connection.value | Add-Member -MemberType scriptmethod `
                           -Name BuildConnectionString `
                           -Value $sbuildconnstring `
                           -Passthru
```

The first line converts the here string with the script into a `scriptblock`. Then the `Add-Member cmdlet` is used to attach the `scriptblock` to the `scriptmethod BuildConnectionString`. The `Passthru` parameter tells PowerShell to pass this `scriptmethod` through the pipe.

The next method, `RunSQL`, is the real powerhouse of the object, and yet it is coded in the same way. This method will execute any SQL statement passed to it, using the connection string generated by the `BuildConnectionString` method to connect to a database. Let's look at the script definition here:

```
# Do NOT put double quotes around the object $this properties as it messes up the values.
# **** RunSQL ***
$bsql = @'
    param([string]$p_insql,[boolean]$IsSelect)

    $this.BuildConnectionString()

    If ($this.ConnectionString -eq 'NotAssigned') {Throw "Error - Cannot create connection
    string." }

    $Result = Invoke-UdfSQL $this.ConnectionType $this.DatabaseType $this.Server `
        $this.DatabaseName "$p_insql" `
        $this.AuthenticationType $this.UserID $this.Password `
        $this.ConnectionString $IsSelect `
        $this.DSNName $this.Driver

    RETURN $Result
'@
```

The script is assigned to a variable, `$bsql`, as a here string. The script takes two parameters, `$p_insql`, which is the SQL statement to be executed, and `$IsSelect`, a Boolean, which is `$true` if the SQL statement is a Select statement and `$false` if it is not a Select statement, such as an update statement. Statements that return results need to be handled differently than statements that do not. Just in case the user did not call the `BuildConnectionString` method yet, the script calls it to make sure we have a connection string. Then the script checks that the `ConnectionString` has been assigned a value, and, if not, it throws an error. Finally, `Invoke-UdfSQL` is called with the parameters for connection type, database type, server name, database name, SQL statement, authentication type, User ID, Password, Connection String, `$true` if the statement is a Select statement or `$false` if it is not, DSN Name, and Driver Name. If the SQL were a Select statement, `$Result` will hold the data returned. The code that follows attaches the script to the object:

```
$ssql = [scriptblock]::create($bsql)

$p_connection.value | Add-Member -MemberType scriptmethod `
    -Name RunSQL `
    -Value $ssql `
    -Passthru
```

The here string holding the script is converted to a `scriptblock` variable `$ssql`. Then `$ssql` is attached to the object with the `Add-Member` cmdlet and becomes `scriptmethod RunSQL`. The `Passthru` parameter tells PowerShell to pass the `scriptmethod` through the pipe.

Supporting Functions

There are a few support functions used by the object that `New-UdfConnection` returns. The object user need not be aware of these functions, but they help the connection object perform tasks. If the user is familiar with these functions, they can call them directly rather than through the connection object. This adds flexibility to the module, as a developer can pick and choose what they want to use.

Get-UdfConnectionString

The connection object returned by `New-UdfConnection` calls `Get-UdfConnectionString` when the method `BuildConnectionString` is executed. This function creates a connection string based on the parameters passed to it.

Rather than write code that dynamically formats a connection string, why not read them from a text file? The format of a connection string is pretty static. Given the connection requirements—i.e., connection type, such as ADO.Net or ODBC, the database type, and the authentication method, such as Windows Integrated Security or logon credentials—the connection string format does not change. Only the variables like server name, database name, user ID, and password change. Ideally, we want to use connection strings in a file, like a template in which we can fill in the variable values. Fortunately, PowerShell makes this very easy to do. We'll use the CSV file named `connectionstring.txt` with the columns `Type`, `Platform`, `AuthenticationType`, and `ConnectionString`. `Type`, `Platform`, and `AuthenticationType` will be used to locate the connection string we need. Let's take a look at the contents of the file:

```
"Type","Platform","AuthenticationType","ConnectionString"
"ADO","SqlServer","Integrated","Data Source=$server;Integrated Security=SSPI;Initial
Catalog=$databasename"
"ADO","SqlServer","Logon","Data Source=$server;Persist Security Info=False;
IntegratedSecurity=false;Initial Catalog=$databasename;User ID=$userid;Password=$password"
"ADO","SqlServer","Credential","Data Source=$server;Persist Security Info=False;Initial
Catalog=$databasename"
```



```
"ADO","Azure","Logon","Data Source=$server;User ID=$userid;Password=$password;Initial
Catalog=$databasename;Trusted_Connection=False;TrustServerCertificate=False;Encrypt=True;"
"ODBC","PostgreSQL","DSNLess","Driver={$driver};Server=$server;Port=5432;Database=
$databasename;Uid=$userid;Pwd=$password;SSLmode=disable;ReadOnly=0;"
"ODBC","PostgreSQL","DSN","DSN=$dsn;SSLmode=disable;ReadOnly=0;"
```

Notice that we have what appear to be PowerShell variables in the data, prefixed with \$, as values for things in the connection string that are not static.

Now, let's take a look at the code that uses the text file to build a connection string:

```
function Get-UdfConnectionString
{
    [CmdletBinding()]
    param (
        [string] $type           , # Connection type, ADO, OleDb, ODBC, etc.
        [string] $dbtype         , # Database Type; SQL Server, Azure, MySQL.
        [string] $server         , # Server instance name
        [string] $authentication , # Authentication method
        [string] $databasename   , # Database
        [string] $userid         , # User Name if using credential
        [string] $password       , # password if using credential
        [string] $dsnname        , # dsn name for ODBC connection
        [string] $driver         , # driver for ODBC DSN Less connection
    )

    $connstrings = Import-CSV ($PSScriptRoot + "\connectionstrings.txt")

    foreach ($item in $connstrings)
    {
        if ($item.Type -eq $type -and $item.Platform -eq $dbtype -and $item.AuthenticationType -eq $authentication)
        {
            $connstring = $item.ConnectionString
            $connstring = $ExecutionContext.InvokeCommand.ExpandString($connstring)
            Return $connstring
        }
    }

    # If this line is reached, no matching connection string was found.
    Return "Error - Connection string not found!"
}
```

In this code, the function parameters are documented by comments; let's look at the first statement. The first line loads a list of connection strings from a CSV file into a variable named \$connstrings. It looks for the file in the path pointed to by the automatic variable \$PSScriptRoot, which is the folder the module umd_database is stored in. Make sure you copied the connectionstring.txt file to this folder. Using a foreach loop, we loop through each row in \$connstrings until we find a row that matches the connection type, database type, and authentication type passed to the function. The matching row is loaded into \$connstring. Now, a nice thing about PowerShell strings is that they automatically expand to replace variable names with their values. By having the function parameter names match the variable names used in

the connection strings in the file, we can have PowerShell automatically fill in the values for us by expanding the string using the command `$ExecutionContext.InvokeCommand.ExpandString`. Then, the connection string is returned to the caller. If no match is found, the statement after the `foreach` loop will be executed, which returns a message “Error - Connection string not found!”. Normally, creating connection strings is tedious, but this function can be extended to accommodate many different requirements just by adding new rows to the input file.

Invoke-UdfSQL

`Invoke-UdfSQL` is the function called by the connection object’s `RunSQL` method. This function acts like a broker to determine which specific function to call to run the SQL statement. It currently supports ADO and ODBC, but you can easily add other types like `OleDb`. Let’s take a look at the `Invoke-UdfSQL` function:

```
function Invoke-UdfSQL ([string]$p_inconntype,
                        [string]$p_indbtype,
                        [string]$p_inserver,
                        [string]$p_indb,
                        [string]$p_insql,
                        [string]$p_inauthenticationtype,
                        [string]$p_inuser,
                        $p_inpw, # No type defined; can be securestring or string
                        [string]$p_inconnectionstring,
                        [boolean]$p_inisselect,
                        [string]$p_indsnname,
                        [string]$p_indriver,
                        [boolean]$p_inisprocedure,
                        $p_inparms)
{
    If ($p_inconntype -eq "ADO")
    {
        If ($p_inisprocedure)
        {
            RETURN Invoke-UdfADOStoredProcedure $p_inserver $p_indb $p_insql `
                $p_inauthenticationtype $p_inuser $p_inpw $p_inconnectionstring $p_inparms
        }
        Else
        {
            $datatab = Invoke-UdfADOSQL $p_inserver $p_indb $p_insql `
                $p_inauthenticationtype $p_inuser $p_inpw $p_inconnectionstring $p_inisselect
            Return $datatab
        }
    }
    ElseIf ($p_inconntype -eq "ODBC")
    {
        If ($p_inisprocedure)
        {
            RETURN Invoke-UdfODBCStoredProcedure $p_inserver $p_indb $p_insql `
                $p_inauthenticationtype $p_inuser $p_inpw $p_inconnectionstring $p_inparms
        }
    }
}
```

```

Else
{
    $datatab = Invoke-UdfODBCSQL $p_inserver $p_indb $p_insql $p_inauthenticationtype `
        $p_inuser $p_inpw $p_inconnectionstring $p_inisselect $p_indsnname $driver

    Return $datatab
}
}
Else
{
    Throw "Connection Type Not Supported."
}
}
}

```

This function has an extensive parameter list. Some of them are not needed, but it's good to have the extra information in case we need it. Notice that parameter `$p_inpw` has no type defined. This is so the parameter can accept whatever is passed to it. As I will demonstrate later, the connection object can support either an encrypted or a clear-text password. An encrypted string is of type `securestring`, and clear text is of type `string`. Note: Depending on the connection requirements, some of the parameters may not have values. The function code consists of an If/Else block. If the parameter connection type `$p_inconntype` is 'ADO', then the Boolean is checked to see if this is a stored procedure call. If yes, the function `Invoke-UdfADOSToredProcedure` is called, otherwise `Invoke-UdfADOSQL` is called, with the results loaded into `$datatab`, which is then returned to the caller. If the parameter connection type `$p_inconntype` is 'ODBC', then the Boolean is checked to see if this is a stored procedure call. If yes, the function `Invoke-UdfODBCStoredProcedure` is called, otherwise `Invoke-UdfODBCSQL` is called, with the results loaded into `$datatab`, which is returned to the caller. If the connection type is neither of these, a terminating error is thrown.

Invoke-UdfADOSQL

`Invoke-UdfADOSQL` uses ADO.Net to execute the SQL statement and the connection string passed in. ADO.Net is a very flexible provider supported by many vendors, including Oracle, MySQL, and PostgreSQL. Let's look at the function `Invoke-UdfADOSQL` code:

```

function Invoke-UdfADOSQL
{
    [CmdletBinding()]
    param (
        [string] $sqlserver           , # SQL Server
        [string] $sqldatabase        , # SQL Server Database.
        [string] $sqlquery           , # SQL Query
        [string] $sqlauthenticationtype , # $true = Use Credentials
        [string] $sqluser            , # User Name if using credential
        $sqlpw                      , # password if using credential
        [string] $sqlconnstring      , # Connection string
        [boolean]$sqlisselect        # true = select, false = non select statement
    )
}

```

```

if ($sqlauthenticationtype -eq 'Credential')
{
    $pw = $sqlpw
    $pw.MakeReadOnly()

    $SqlCredential = new-object System.Data.SqlClient.SqlCredential($sqluser, $pw)
    $conn = new-object System.Data.SqlClient.SqlConnection($sqlconnstring, $SqlCredential)
}
else
{
    $conn = new-object System.Data.SqlClient.SqlConnection($sqlconnstring)
}

$conn.Open()

$command = new-object system.data.sqlclient.Sqlcommand($sqlquery,$conn)

if ($sqlisselect)
{
    $adapter = New-Object System.Data.sqlclient.SqlDataAdapter $command
    $dataset = New-Object System.Data.DataSet
    $adapter.Fill($dataset) | Out-Null
    $conn.Close()
    RETURN $dataset.tables[0]
}
Else
{
    $command.ExecuteNonQuery()
    $conn.Close()
}

```

This function takes a lot of parameters, but most of them are not needed; rather, they are included in case there is a need to extend the functionality. Let's take a closer look at the first section of code that handles credentials:

```

if ($sqlauthenticationtype -eq 'Credential')
{
    $pw = $sqlpw
    $pw.MakeReadOnly()

    $SqlCredential = new-object System.Data.SqlClient.SqlCredential($sqluser, $pw)
    $conn = new-object System.Data.SqlClient.SqlConnection($sqlconnstring, $SqlCredential)
}
else
{
    $conn = new-object System.Data.SqlClient.SqlConnection($sqlconnstring);
}

```

We can see that if the authentication type passed is equal to Credential, it indicates that the caller wants to use a credential object to log in. A credential object allows us to separate the connection string from the user ID and password so no one can see them. Note: This concept can also be used in making

web-service requests. If the authentication type is `Credential`, the password parameter, `$sqlpw`, is assigned to `$pw`, and then this variable is set to read only. This is required by the credential object. Then, we create an instance of the `SqlCredential` object, passing the user ID and password as parameters, which returns a reference to the variable `$SqlCredential`. Finally, the connection object is created as an instance of `SqlConnection` with the connection string, `$sqlconnectonstring`, and the credential object, `$SqlCredential`, as parameters. The connection reference is returned to `$conn`. If the authentication type is not equal to `Credential`, the connection is created by passing only the connection string to the `SqlConnection` method. Note: The user ID and password would be found in the connection string if the authentication type were `Logon` or `DSNLess`.

Now the function is ready to open the connection as shown here:

```
$conn.Open()
```

```
$command = new-object system.data.sqlclient.Sqlcommand($sqlquery,$conn)
```

First, the connection must be opened. Then, a command object is created via the `SqlCommand` method, and the SQL statement and connection object are passed as parameters, returning the object to `$command`. The code to execute the SQL statement is as follows:

```
if ($sqlisselect)
{
    $adapter = New-Object System.Data.sqlclient.SqlDataAdapter $command
    $dataset = New-Object System.Data.DataSet
    $adapter.Fill($dataset) | Out-Null
    $conn.Close()
    RETURN $dataset.tables[0]
}
Else
{
    $command.ExecuteNonQuery()
    $conn.Close()
}
```

Above, the Boolean parameter `$sqlisselect` is tested for a value of true—i.e., “if (`$sqlisselect`)”. Since it is a Boolean type, there is no need to compare the value to `$true` or `$false`. If true, this is a select statement and it needs to retrieve data and return it to the caller. We can see a SQL data adapter being created, with the `SqlDataAdapter` method call passing the command object as a parameter. Then a dataset object, `DataSet`, is created to hold the results. The data adapter—`$adapter`—`Fill` method is called to load the dataset with the results. The connection is closed, as we don’t want to accumulate open connections. It is important to take this cleanup step. A dataset is a collection of tables. A command can submit multiple select statements, and each result will go into a separate table element in the dataset. This function only supports one select statement so as to keep things simple. Therefore, only the first result set is returned, as in the statement “`RETURN $dataset.table[0]`”. If `$sqlisselect` is false, the statement just needs to be executed. No results are returned. Therefore, the command method `ExecuteNonQuery` is called. Then, the connection is closed. There is nothing to return to the caller.

Using the Credential Object

The credential object provides good protection of the user ID and password. On-premises SQL Server supports using a credential object at logon, but Azure SQL does not. As we saw, the object returned by `Get-UdfConnection` supports using a credential. However, using it requires a slightly different set of statements. Let's look at the following example:

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

$myconnection.ConnectionType = 'ADO'
$myconnection.DatabaseType = 'SqlServer'
$myconnection.Server = '(local)'
$myconnection.DatabaseName = 'AdventureWorks'
$myconnection.UseCredential = 'Y'
$myconnection.UserID = 'bryan'
$myconnection.Password = Get-Content 'C:\Users\BryanCafferky\Documents\password.txt' |
convertto-securestring

$myconnection.SetAuthenticationType('Credential')
$myconnection.BuildConnectionString() # Should return the connection string.
```

These statements will set all the properties needed to run queries against our local instance of SQL Server. Most of this is the same as what we did before. The line I want to call your attention to is copied here:

```
$myconnection.Password = Get-Content 'C:\Users\BryanCafferky\Documents\password.txt' |
convertto-securestring
```

This line loads the password from a file that is piped into the `ConvertTo-Securestring` cmdlet to encrypt it. If you try to display the password property of `$myconnection`, you will just see `'System.Security.Securestring'`. You cannot view the contents. So the password is never readable to anyone once it has been saved. Other than that difference, the connection object is used the same way as before. We can submit a query such as the one here:

```
$myconnection.RunSQL("SELECT top 20 * FROM [HumanResources].[EmployeeDepartmentHistory]",
$true) | Select-Object -Property BusinessEntityID, DepartmentID, ShiftID, StartDate |
Out-GridView
```

The query results will be returned and piped into the `Select-Object` cmdlet, which selects the columns desired, and then will be piped into the `Out-GridView` for display.

Encrypting and Saving the Password to a File

We have not seen how to save the password to a file. We can do that with this statement:

```
Save-UdfEncryptedCredential
```

This statement calls a module function that will prompt us for a password and, after we enter it, will prompt us for the filename to save it to with the Window's Save File common dialog. Let's take a look at the code for this function:

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

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

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

The first executable line prompts the user for a password, which is not displayed as typed because the `assecurestring` parameter to `Read-Host` suppresses display of the characters. The second line displays the Save File common dialog so the user can choose where to save the password. Because the call to `Invoke-UdfCommonDialogSaveFile` is in parentheses, it will execute first, and the password will be stored to the file specified by the user. The password entered is piped into `ConvertFrom-SecureString`, which obfuscates it by making it a readable series of numbers that is piped into the `Out-File` cmdlet, which saves the file.

For completeness, the function `Invoke-UdfCommonDialogSaveFile`, which displays the Save File common dialog form, is listed here:

```
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 uses the Window's form `SaveFileDialog` to present a user with the familiar Save As dialog. The only parameter is the initial directory the caller wants the dialog to default to. The last line returns the selected folder and filename entered.

Calling Stored Procedures

In simple cases, stored procedures can be called like any other SQL statement. If the procedure returns a query result, it can be executed as a select query. For example, consider the stored procedure in Listing 7-2 that returns a list of employees.

Listing 7-2. The stored procedure `HumanResources.uspListEmployeePersonalInfoPS`

```
Create PROCEDURE [HumanResources].[uspListEmployeePersonalInfoPS]
    @BusinessEntityID [int]
WITH EXECUTE AS CALLER
AS
BEGIN
    SET NOCOUNT ON;

    BEGIN TRY

        select *
        from [HumanResources].[Employee]
        where [BusinessEntityID] = @BusinessEntityID;

    END TRY
    BEGIN CATCH
        EXECUTE [dbo].[uspLogError];
    END CATCH;
END;

GO
```

The procedure in Listing 7-2 will return the list of employees that have the `BusinessEntityID` passed to the function—i.e., one employee, since this is the primary key. We can test it on SQL Server as follows:

```
exec [HumanResources].[uspListEmployeePersonalInfoPS] 1
```

One row is returned. Using the connection object returned by `New-UdfConnection`, we can code the call to this stored procedure, as shown in Listing 7-3. Note: Change properties as needed to suit your environment.

Listing 7-3. Calling a SQL Server stored procedure

```
Import-Module umd_database -Force

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

$myconnection.ConnectionType = 'ADO'
$myconnection.DatabaseType = 'SqlServer'
$myconnection.Server = '(local)'
$myconnection.DatabaseName = 'AdventureWorks'
$myconnection.UseCredential = 'N'

$myconnection.SetAuthenticationType('Integrated')
$myconnection.BuildConnectionString()
$empid = 1

$myconnection.RunSQL("exec [HumanResources].[uspListEmployeePersonalInfoPS] $empid", $true)
```


Since the result set is passed back from a select query within the stored procedure, the call to the procedure is treated like a select statement. Notice that we can even include input parameters by using PowerShell variables.

Calling Stored Procedures Using Output Parameters

When we want to call a stored procedure that uses output parameters to return results, we need to add the parameters object to the database call. To demonstrate, let's consider the stored procedure that follows that takes the input parameters `BusinessEntityID`, `NationalIDNumber`, `BirthDate`, `MaritalStatus`, and `Gender` and returns the output parameters `JobTitle`, `HireDate`, and `VacationHours`. This is a modified version of an AdventureWorks stored procedure named `[HumanResources].[uspUpdateEmployeePersonalInfo]`, with `PS` appended to the name, indicating it is for use by our PowerShell script. The input parameters are used to update the employee record. The output parameters are returned from the call. Let's review the stored procedure in Listing 7-4.

Listing 7-4. A stored procedure with output parameters

```
USE [AdventureWorks]
GO

SET ANSI_NULLS ON
SET QUOTED_IDENTIFIER ON
GO

Create PROCEDURE [HumanResources].[uspUpdateEmployeePersonalInfoPS]
    @BusinessEntityID [int],
    @NationalIDNumber [nvarchar](15),
    @BirthDate [datetime],
    @MaritalStatus [nchar](1),
    @Gender [nchar](1),
    @JobTitle [nvarchar](50) output,
    @HireDate [date] output,
    @VacationHours [smallint] output
WITH EXECUTE AS CALLER
AS
BEGIN
    SET NOCOUNT ON;

    BEGIN TRY
        UPDATE [HumanResources].[Employee]
        SET [NationalIDNumber] = @NationalIDNumber
            ,[BirthDate] = @BirthDate
            ,[MaritalStatus] = @MaritalStatus
            ,[Gender] = @Gender
        WHERE [BusinessEntityID] = @BusinessEntityID;

        select @JobTitle = JobTitle, @HireDate = HireDate, @VacationHours = VacationHours
        from [HumanResources].[Employee]
        where [BusinessEntityID] = @BusinessEntityID;
```

```

END TRY
BEGIN CATCH
    EXECUTE [dbo].[uspLogError];
END CATCH;
END;

GO

```

Listing 7-4 is a simple stored procedure, but it allows us to see how to pass input and output parameters of different data types. We can see that the Update statement will update the employee record with the input parameters. Then a select statement will load the values for JobTitle, HireDate, and VacationHours into the output parameters @JobTitle, @HireDate, and @VacationHours. Let's look more closely at how the parameters are defined:

```

@BusinessEntityID [int],
@NationalIDNumber [nvarchar](15),
@BirthDate [datetime],
@MaritalStatus [nchar](1),
@Gender [nchar](1),
@JobTitle [nvarchar](50) output,
@HireDate [date] output,
@VacationHours [smallint] output

```

Parameters can be of three possible types, which are Input, Output, or InputOutput. Input parameters can be read but not updated. Output parameters by definition should only be updatable but not read. InputOutput parameters can be read and updated. SQL Server does not support an Output parameter that can only be updated. Rather, it treats Output parameters as InputOutput parameters. By default, a parameter is Input, so it does not need to be specified. Notice that for the Output parameters, the word 'output' is specified after the data type.

Before we look at how to call this stored procedure with PowerShell, let's review how we would call it from SQL Server. Actually, it's a good idea to test calls to SQL from within a SQL Server tool like SQL Server Management Studio before trying to develop and test PowerShell code to do the same thing. The short SQL script in Listing 7-5 executes our stored procedure.

Listing 7-5. A SQL script to call a stored procedure

```

Use AdventureWorks
go

Declare @JobTitle [nvarchar](50)
Declare @HireDate [date]
Declare @VacationHours [smallint]

exec [HumanResources].[uspUpdateEmployeePersonalInfoPS] 1, 295847284, '1963-01-01', 'M',
'M', @JobTitle Output, @HireDate Output, @VacationHours Output

print @JobTitle
print @HireDate
print @VacationHours

```

When you run this script, you should see the following output:

```
Chief Executive Officer
2003-02-15
42
```

Don't worry if the actual values are different. It will be whatever is in the database at the time you execute this.

PowerShell Code to Call a Stored Procedure with Output Parameters

Now that we know we can execute the stored procedure with T-SQL, let's do the same thing using PowerShell. Let's look at the PowerShell script in Listing 7-6 that runs the stored procedure. Note: To get the ideas across, the script is pretty hard coded.

Listing 7-6. PowerShell code to call a stored procedure with output parameters

```
Import-Module umd_database

$SqlConnection = New-Object System.Data.SqlClient.SqlConnection
$SqlConnection.ConnectionString = "Data Source=(local);Integrated Security=SSPI;Initial
Catalog=AdventureWorks"
$SqlCommand = New-Object System.Data.SqlClient.SqlCommand
$SqlCommand.CommandText = "[HumanResources].[uspUpdateEmployeePersonalInfoPS]"
$SqlCommand.Connection = $SqlConnection
$SqlCommand.CommandType = [System.Data.CommandType]'StoredProcedure'
$SqlCommand.Parameters.AddWithValue("@BusinessEntityID", 1) >> $null
$SqlCommand.Parameters.AddWithValue("@NationalIDNumber", 295847284) >> $null
$SqlCommand.Parameters.AddWithValue("@BirthDate", '1964-02-02') >> $null
$SqlCommand.Parameters.AddWithValue("@MaritalStatus", 'S') >> $null
$SqlCommand.Parameters.AddWithValue("@Gender", 'M') >> $null

# -- Output Parameters ---
# JobTitle
$outParameter1 = new-object System.Data.SqlClient.SqlParameter
$outParameter1.ParameterName = "@JobTitle"
$outParameter1.Direction = [System.Data.ParameterDirection]::Output
$outParameter1.DbType = [System.Data.DbType]'string'
$outParameter1.Size = 50
$SqlCommand.Parameters.Add($outParameter1) >> $null

# HireDate
$outParameter2 = new-object System.Data.SqlClient.SqlParameter
$outParameter2.ParameterName = "@HireDate"
$outParameter2.Direction = [System.Data.ParameterDirection]::Output
$outParameter2.DbType = [System.Data.DbType]'date'
$SqlCommand.Parameters.Add($outParameter2) >> $null

# VacationHours
$outParameter3 = new-object System.Data.SqlClient.SqlParameter
$outParameter3.ParameterName = "@VacationHours"
$outParameter3.Direction = [System.Data.ParameterDirection]::Output
```

```

$outParameter3.DbType = [System.Data.DbType]'int16'
$SqlCommand.Parameters.Add($outParameter3) >> $null

$SqlConnection.Open()
$result = $SqlCommand.ExecuteNonQuery()
$SqlConnection.Close()

$SqlCommand.Parameters["@jobtitle"].value
$SqlCommand.Parameters["@hiredate"].value
$SqlCommand.Parameters["@VacationHours"].value

```

There's a lot of code there, but we'll walk through it. As before, first we import the `umd_database` module. Then, we define the parameters. First, let's look at the code that creates the SQL connection and command objects:

```

$SqlConnection = New-Object System.Data.SqlClient.SqlConnection
$SqlConnection.ConnectionString = "Data Source=(local);Integrated Security=SSPI;Initial
Catalog=AdventureWorks"
$SqlCommand = New-Object System.Data.SqlClient.SqlCommand
$SqlCommand.CommandText = "[HumanResources].[uspUpdateEmployeePersonalInfoPS]"
$SqlCommand.Connection = $SqlConnection
$SqlCommand.CommandType = [System.Data.CommandType]'StoredProcedure';

```

We've seen the first few lines before—creating the connection, assigning the connection string, and creating a command object. Notice that for the command's `CommandText` property, we're just giving the name of the stored procedure, i.e., `[HumanResources].[uspUpdateEmployeePersonalInfoPS]`. Then, we connect the command to the connection with the line `"$SqlCommand.Connection = $SqlConnection"`. Finally, the last line assigns the `CommandType` property as `'StoredProcedure'`. This is critical in order for the command to be processed correctly. The Input parameters are assigned by the code here:

```

$SqlCommand.Parameters.AddWithValue("@BusinessEntityID", 1) >> $null
$SqlCommand.Parameters.AddWithValue("@NationalIDNumber", 295847284) >> $null
$SqlCommand.Parameters.AddWithValue("@BirthDate", '1964-02-02') >> $null
$SqlCommand.Parameters.AddWithValue("@MaritalStatus", 'S') >> $null
$SqlCommand.Parameters.AddWithValue("@Gender", 'M') >> $null

```

Input parameters can use the abbreviated format for assignment. The parameters collection of the command object holds the parameter details. The `AddWithValue` method adds each parameter with value to the collection. To suppress any output returned from the `AddWithValue` method, we direct it to `$null`. Output parameters need to be coded in a manner that provides more details. Now, let's look at the code that creates the `JobTitle` output parameters:

```

# <---- Output Parameters <---->
# JobTitle
$outParameter1 = new-object System.Data.SqlClient.SqlParameter
$outParameter1.ParameterName = "@JobTitle"
$outParameter1.Direction = [System.Data.ParameterDirection]::Output
$outParameter1.DbType = [System.Data.DbType]'string'
$outParameter1.Size = 50
$SqlCommand.Parameters.Add($outParameter1) >> $null

```

In the first non-comment line above, we create a new SQL parameter object instance to hold the information about the parameter. Once created, we just assign details about the parameter, such as `ParameterName`, `Direction`, `DbType`, and `Size`, to the associated object properties. The `ParameterName` is the name defined in the stored procedure, which is why it has the @ sign prefix, as SQL Server variables and parameters have. The `Direction` property tells whether this is an Input, Output, or InputOutput parameter. Note: Although SQL Server does not support an InputOutput direction, ADO.Net does. We define the direction as Output. The `DbType` property is not of the SQL Server data type. Rather, it is an abstraction that will equate to a SQL Server data type in our case. For another type of database, the underlying database-type columns may be different. For a complete list of `DbType` to SQL Server data-type mappings, see the link: <https://msdn.microsoft.com/en-us/library/cc716729%28v=vs.110%29.aspx>

We use the `DbType` 'string' for the `JobTitle`, which is defined as `nvarchar(50)`. For string types, we need to assign the `Size` property, which is the length of the column. Finally, we use the `Parameters` collection `Add` method to add the parameter to the collection. Table 7-1 shows some of the most common SQL Server data types and their ADO.Net corresponding `DbType`.

Table 7-1. Common Data Types

SQL Server Database Format	ADO.Net DbType
int	Int32
Smallint	Int16
bigint	Int64
varchar	String or Char[]
nvarchar	String or Char[]
char	String or Char[]
bit	Boolean
date	Date
decimal	Decimal
text	String or Char[]
ntext	String or Char[]

Now, let's look at the code that creates the remaining two output parameters:

```
# HireDate
$outParameter2 = new-object System.Data.SqlClient.SqlParameter
$outParameter2.ParameterName = "@HireDate"
$outParameter2.Direction = [System.Data.ParameterDirection]::Output
$outParameter2.DbType = [System.Data.DbType]'date'
$SqlCommand.Parameters.Add($outParameter2) >> $null

# VacationHours
$outParameter3 = new-object System.Data.SqlClient.SqlParameter
$outParameter3.ParameterName = "@VacationHours"
$outParameter3.Direction = [System.Data.ParameterDirection]::Output
$outParameter3.DbType = [System.Data.DbType]'int16'
$SqlCommand.Parameters.Add($outParameter3) >> $null
```

The code to assign the HireDate and VacationHours parameters is not very different from the code we saw to define the JobTitle parameter. Notice that the DbType of date and int16 do not require a value for the size property. Now, let's look at the code to execute the stored procedure:

```
$SqlConnection.Open();
$result = $SqlCmd.ExecuteNonQuery()
$SqlConnection.Close();
```

First, we open the connection. Then, we use the ExecuteNonQuery method of the command object to run the stored procedures, returning any result to \$result. The returned value is usually -1 for success and 0 for failure. However, when there is a trigger on a table being inserted to or updated, the number of rows inserted and/or updated is returned.

We can see the Output parameters by getting their Value property from the Parameters collection, as shown here:

```
$SqlCmd.Parameters["@jobtitle"].value
$SqlCmd.Parameters["@hiredate"].value
$SqlCmd.Parameters["@VacationHours"].value
```

Notice that although the connection is closed, we can still retrieve the Output parameters.

Calling Stored Procedures the Reusable Way

We have seen how we can use PowerShell to call stored procedures. Now, let's take a look at how we can incorporate those ideas as reusable functions in the umd_database module. Overall, executing a stored procedure is like executing any SQL statement, except we need to provide the Input and Output parameters. The challenge here is that there can be any number of parameters, and each has a set of property values. How can we provide for passing a variable-length list of parameters to a function? Since PowerShell supports objects so nicely, why not create the parameter list as a custom object collection? The function that gets the object collection as a parameter can iterate over the list of parameters to create each SQL command parameter object. To help us build this collection, we'll use the helper function Add-UdfParameter, which creates a single parameter as a custom object:

```
function Add-UdfParameter {
    [CmdletBinding()]
    param (
        [string] $name      , # Parameter name from stored procedure, i.e. @myparm
        [string] $direction , # Input or Output or InputOutput
        [string] $value     , # parameter value
        [string] $datatype  , # db data type, i.e. string, int64, etc.
        [int]    $size      # length
    )

    $parm = New-Object System.Object
    $parm | Add-Member -MemberType NoteProperty -Name "Name" -Value "$name"
    $parm | Add-Member -MemberType NoteProperty -Name "Direction" -Value "$direction"
    $parm | Add-Member -MemberType NoteProperty -Name "Value" -Value "$value"
    $parm | Add-Member -MemberType NoteProperty -Name "Datatype" -Value "$datatype"
    $parm | Add-Member -MemberType NoteProperty -Name "Size" -Value "$size"

    RETURN $parm
}
```

This function takes the parameters for each of the properties of the SQL parameter object required to call a stored procedure. The first executable line in the function stores an instance of `System.Object` into `$parm`, which gives us a place to which to attach the properties. We then pipe `$parm` into the `Add-Member` cmdlet to add each property with a value from each of the parameters passed to the function. Finally, we return the `$parm` object back to the caller. Let's see how we would use the `Add-UdfParameter` in Listing 7-7.

Listing 7-7. Using `Add-UdfParameter`

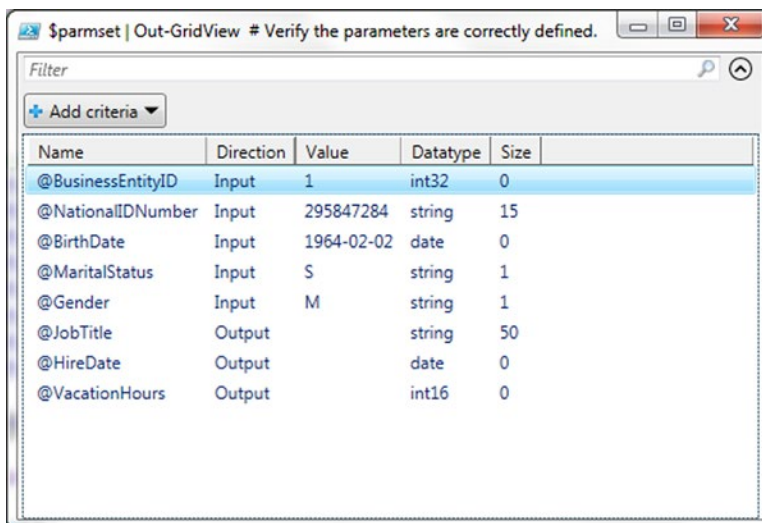
```
Import-Module umd_database

$parmset = @() # Create a collection object.

# Add the parameters we need to use...
$parmset += (Add-UdfParameter "@BusinessEntityID" "Input" "1" "int32" 0)
$parmset += (Add-UdfParameter "@NationalIDNumber" "Input" "295847284" "string" 15)
$parmset += (Add-UdfParameter "@BirthDate" "Input" "1964-02-02" "date" 0)
$parmset += (Add-UdfParameter "@MaritalStatus" "Input" "S" "string" 1)
$parmset += (Add-UdfParameter "@Gender" "Input" "M" "string" 1)
$parmset += (Add-UdfParameter "@JobTitle" "Output" "" "string" 50)
$parmset += (Add-UdfParameter "@HireDate" "Output" "" "date" 0)
$parmset += (Add-UdfParameter "@VacationHours" "Output" "" "int16" 0)

$parmset | Out-GridView # Verify the parameters are correctly defined.
```

Listing 7-7 starts by importing the `umd_database` module. Then, we create an empty collection object named `$parmset` by setting it equal to `@()`. We load this collection with each parameter by calling the function `Add-UdfParameter`. There are a few things to notice here. We enclose the function call in parentheses to make sure that the function is executed first. We use the `+=` assignment operator to append the object returned to the `$parmset` collection. Normally, the `+=` operator is used to increment the variable on the left side of the operator with the value on the right side, i.e., `x += 1` is the same as saying `x = x + 1`. However, in the case of objects, it appends the object instance to the collection. The last statement pipes the parameter collection `$parmset` to `Out-GridView` so we can see the list. We should see a display like the one in Figure 7-1.



Name	Direction	Value	Datatype	Size
@BusinessEntityID	Input	1	int32	0
@NationalIDNumber	Input	295847284	string	15
@BirthDate	Input	1964-02-02	date	0
@MaritalStatus	Input	S	string	1
@Gender	Input	M	string	1
@JobTitle	Output		string	50
@HireDate	Output		date	0
@VacationHours	Output		int16	0

Figure 7-1. *Out-GridView showing the parameter collection*

Now that we have the parameter collection, we want to pass this to a function that will execute the stored procedure. Let's look at the code to do this:

```
function Invoke-UdfADOStoredProcedure
{
    [CmdletBinding()]
    param (
        [string] $sqlserver            , # SQL Server
        [string] $sqldatabase         , # SQL Server Database.
        [string] $sqlspname           , # SQL Query
        [string] $sqlauthenticationtype , # $true = Use Credentials
        [string] $sqluser              , # User Name if using credential
        $sqlpw                        , # password if using credential
        [string] $sqlconnstring        , # Connection string
        $parameterset                 , # Parameter properties
    )

    if ($sqlauthenticationtype -eq 'Credential')
    {
        $pw = $sqlpw
        $pw.MakeReadOnly()

        $SqlCredential = new-object System.Data.SqlClient.SqlCredential($sqluser, $pw)
        $conn = new-object System.Data.SqlClient.SqlConnection($sqlconnstring, $SqlCredential)
    }
    else
    {
        $conn = new-object System.Data.SqlClient.SqlConnection($sqlconnstring);
    }

    $conn.Open()

    $command = new-object system.data.sqlclient.Sqlcommand($sqlspname,$conn)

    $command.CommandType = [System.Data.CommandType]'StoredProcedure';

    foreach ($parm in $parameterset)
    {
        if ($parm.Direction -eq 'Input')
        {
            $command.Parameters.AddWithValue($parm.Name, $parm.Value) >> $null;
        }
        elseif ($parm.Direction -eq "Output" )
        {
            $outparm1 = new-object System.Data.SqlClient.SqlParameter;
            $outparm1.ParameterName = $parm.Name
            $outparm1.Direction = [System.Data.ParameterDirection]::Output;
            $outparm1.DbType = [System.Data.DbType]$parm.Datatype;
            $outparm1.Size = $parm.Size
            $command.Parameters.Add($outparm1) >> $null
        }
    }
}
```



```

$command.ExecuteNonQuery()
$conn.Close()

$outparms = @{}

foreach ($parm in $parameterset)
{
    if ($parm.Direction -eq 'Output')
    {
        $outparms.Add($parm.Name, $command.Parameters[$parm.Name].value)
    }
}

RETURN $outparms
}

```

Let's review this code in detail. The first executable block of lines should look familiar, as it is the same code used in the function earlier to run a SQL statement, i.e., `Invoke-UdfADOSQL`:

```

if ($sqlauthenticationtype -eq 'Credential')
{
    $pw = $sqlpw
    $pw.MakeReadOnly()

    $SqlCredential = new-object System.Data.SqlClient.SqlCredential($sqluser, $pw)
    $conn = new-object System.Data.SqlClient.SqlConnection($sqlconnstring, $SqlCredential)
}
else
{
    $conn = new-object System.Data.SqlClient.SqlConnection($sqlconnstring);
}

$conn.Open()

$command = new-object system.data.sqlclient.Sqlcommand($sqlspname,$conn)

```

First, we check if we need to use a credential object. If yes, then we make the password read only and create the credential object, passing in the user ID and password. Then, we create the connection object using the connection string and credential object. If no credential object is needed, we just create the connection object using the connection string. Then, we open the connection using the `Open` method. By now the code should be looking familiar.

The next statement sets the `CommandType` property to `'StoredProcedure'`:

```

$command.CommandType = [System.Data.CommandType]'StoredProcedure';

```

From here we are ready to create the stored procedure parameters, which we do with the code that follows:

```
foreach ($parm in $parameterset)
{
    if ($parm.Direction -eq 'Input')
    {
        $command.Parameters.AddWithValue($parm.Name, $parm.Value) >> $null;
    }
    elseif ($parm.Direction -eq "Output" )
    {
        $outparm1 = new-object System.Data.SqlClient.SqlParameter;
        $outparm1.ParameterName = $parm.Name
        $outparm1.Direction = [System.Data.ParameterDirection]::Output;
        $outparm1.DbType = [System.Data.DbType]$parm.Datatype;
        $outparm1.Size = $parm.Size
        $command.Parameters.Add($outparm1) >> $null
    }
}
```

The foreach loop will iterate over each object in the collection `$parameterset` that was passed to this function. On each iteration, `$parm` will hold the current object. Remember, we created this parameter by using the helper function `Add-UdfParameter`. On each iteration, if the parameter's direction property is `Input`, we use the `AddWithValue` method to add the parameter to the command's parameters collection. Otherwise, if the direction is `Output`, we create a new `SqlParameter` object instance and assign the required property values from those provided in `$parm`.

Then, we execute the stored procedure with the statements here:

```
$command.ExecuteNonQuery()
$conn.Close()
```

Now, we need to return the `Output` parameters back to the caller. There are a number of ways this could be done. One method I considered was updating the parameter object collection passed into the function. However, that would make the caller do a lot of work to get the values. Instead, the function creates a hash table and loads each parameter name as the key and the return value as the value. Remember, hash tables are just handy lookup tables. Let's look at the code for this:

```
$outparms = @{}

foreach ($parm in $parameterset)
{
    if ($parm.Direction -eq 'Output')
    {
        $outparms.Add($parm.Name, $command.Parameters[$parm.Name].value)
    }
}

RETURN $outparms
}
```

Here, `$outparms` is created as an empty hash table by the statement "`$outparms = @{}`". Then, we iterate over the parameter set originally passed to the function. For each parameter with a direction of `Output`, we add a new entry into the `$outparms` hash table with the key of the parameter name and the value taken from the SQL command parameters collection. Be careful here not to miss what is happening. We are not using the value from the parameter collection passed to the function. We are going into the SQL command object and pulling the return value from there.

Now we have nice, reusable functions to help us call stored procedures. Wouldn't it be nice to integrate this with the `umd_database` module's connection object we covered earlier? That object was returned by `New-UdfConnection`. Let's integrate the functions to call a stored procedure with `New-UdfConnection` so that we have one nice interface for issuing SQL commands. To do that, we'll need to add a method to the connection object returned by `New-UdfConnection`. The code that follows creates that function and attaches it to the object:

```
# For call, $false set for $IsSelect as this is a store procedure.
# $true set for IsProcedure
$bspsql = @'
    param([string]$p_insql, $p_parms)

    $this.BuildConnectionString()

    If ($this.ConnectionString -eq 'NotAssigned')
        {Throw "Error - Cannot create connection string." }

    $Result = Invoke-UdfSQL $this.ConnectionType $this.DatabaseType `
                        $this.Server $this.DatabaseName "$p_insql" `
                        $this.AuthenticationType $this.UserID $this.Password `
                        $this.ConnectionString $false `
                        $this.DSNName $this.Driver $true $p_parms

    RETURN $Result
'@
$sspsql = [scriptblock]::create($bspsql)

$p_connection.value | Add-Member -MemberType scriptmethod `
                    -Name RunStoredProcedure `
                    -Value $sspsql `
                    -Passthru
```

As with other object methods, we first assign the code block to a here string variable, which is called `$bsqlsql` in this case. The code block shows that two parameters are accepted by the function: `$p_insql`, which is the name of the stored procedure, and `$p_parms`, which is the parameter collection we created using `Add-UdfParameter`. The first executable line of the function uses the object's `BuildConnectionString` to create the connection string so as to connect to the database. If there is a problem creating the connection string, as indicated by a value of `'NotAssigned'`, an error is thrown. Finally, the function `Invoke-UdfSQL` is called, passing the object properties and the parameters. Notice two Boolean values are being passed. The first, which is `$false`, is the `$IsSelect` parameter, so we are saying this is not a select query. The second, which is `$true`, is for a parameter that tells the function this is a stored procedure call.

We're almost there, but there is one piece missing. We need to discuss the function being called, `Invoke-UdfSQL`. This function acts as a broker to determine which specific function to call, i.e., for ADO or ODBC. To see how the function determines which call to make, let's look at the code for `Invoke-UdfSQL`:

```
function Invoke-UdfSQL
(
    [string]$p_inconntype,
    [string]$p_indbtype,
    [string]$p_inserver,
    [string]$p_indb,
    [string]$p_insql,
    [string]$p_inauthenticationtype,
    [string]$p_inuser,
    $p_inpw, # No type defined; can be securestring or string
    [string]$p_inconnectionstring,
    [boolean]$p_inisselect,
    [string]$p_indsnname,
    [string]$p_indriver,
    [boolean]$p_inisprocedure,
    $p_inparms
)
{
    If ($p_inconntype -eq "ADO")
    {
        If ($p_inisprocedure)
        {
            RETURN Invoke-UdfADOSToredProcedure $p_inserver $p_indb $p_insql `
                $p_inauthenticationtype $p_inuser $p_inpw $p_inconnectionstring $p_inparms
        }
        Else
        {
            $datatab = Invoke-UdfADOSQL $p_inserver $p_indb $p_insql $p_inauthenticationtype `
                $p_inuser $p_inpw $p_inconnectionstring $p_inisselect

            Return $datatab
        }
    }
    ElseIf ($p_inconntype -eq "ODBC")
    {
        If ($p_inisprocedure)
        {
            write-host 'sp'
            RETURN Invoke-UdfODBCStoredProcedure $p_inserver $p_indb $p_insql `
                $p_inauthenticationtype $p_inuser $p_inpw $p_inconnectionstring $p_inparms
        }
        Else
        {
            $datatab = Invoke-UdfODBCSQL $p_inserver $p_indb $p_insql $p_inauthenticationtype `
                $p_inuser $p_inpw $p_inconnectionstring $p_inisselect $p_indsnname $driver

            Return $datatab
        }
    }
}
```

```

Else
{
    Throw "Connection Type Not Supported."
    Return "Failed - Connection type not supported"
}
}

```

Don't be intimidated by this code. It's really just a set of if conditions. There are a lot of parameters, but some of them are just to support later expansion of the function. This function supports two types of SQL calls: ADO.Net and ODBC. Once it determines that this call is ADO.Net, it checks the Boolean passed in, `$p_inisprocedure`, to determine if this is a stored procedure call. If this is true, then the function `Invoke-UdfADOSToredProcedure` is called. Since `Invoke-UdfADOSToredProcedure` returns a value, i.e., a hash table of the output parameter values, the `RETURN` statement makes the function call. We've seen the other statements before, which have to do with non-stored procedure calls.

Now that we've covered how all this works, let's look at the code in Listing 7-8 that uses the connection object to call the stored procedure.

Listing 7-8. Using the `umd_database` module's connection object to call a stored procedure

```

Import-Module umd_database

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

$myconnection.ConnectionType = 'ADO'
$myconnection.DatabaseType = 'SqlServer'
$myconnection.Server = '(local)'
$myconnection.DatabaseName = 'AdventureWorks'
$myconnection.UseCredential = 'N'
$myconnection.UserID = 'bryan'
$myconnection.Password = 'password'

$myconnection.SetAuthenticationType('Integrated')

$myconnection.BuildConnectionString()

$paramset = @() # Create a collection object.
# Add the parameters we need to use...
$paramset += (Add-UdfParameter "@BusinessEntityID" "Input" "1" "int32" 0)
$paramset += (Add-UdfParameter "@NationalIDNumber" "Input" "295847284" "string" 15)
$paramset += (Add-UdfParameter "@BirthDate" "Input" "1964-02-02" "date" 0)
$paramset += (Add-UdfParameter "@MaritalStatus" "Input" "S" "string" 1)
$paramset += (Add-UdfParameter "@Gender" "Input" "M" "string" 1)
$paramset += (Add-UdfParameter "@JobTitle" "Output" "" "string" 50)
$paramset += (Add-UdfParameter "@HireDate" "Output" "" "date" 0)
$paramset += (Add-UdfParameter "@VacationHours" "Output" "" "int16" 0)

$myconnection.RunStoredProcedure('[HumanResources].[uspUpdateEmployeePersonalInfoPS]',
$paramset)
$paramset # Lists the output parameters

```

Listing 7-8 shows how we use our custom functions to call a stored procedure that returns Output parameters. First, we create a new `PSObject` variable, i.e., `$myconnection`. A reference to this object is passed to the function `New-UdfConnection`, which attaches our custom connection methods and properties. Then, a series of statements sets the properties of our connection object. We call the `BuildConnectionString` method to generate the connection string needed to access the database. Then, the statement `$paramset = @()` creates an empty object collection named `$paramset`. Subsequent statements append custom parameter objects to the collection using `Add-UdfParameter`. Finally, the stored procedure is executed, passing the stored procedure name and the parameter collection. The results should be written to the console. The last line just displays the parameter set, i.e., `$paramset`.

Summary

In this chapter we explored executing queries using ADO.Net, which will handle most database platforms, including SQL Server, MS Access, Oracle, MySQL, and PostgreSQL. This chapter bypassed using the SQLPS module in favor of writing our own custom module that provides a generic and easy-to-use set of functions to run queries. To provide one point of interaction with these functions, the module provides a database connection object that can use ADO.Net or ODBC to execute any SQL statement. The first part of this chapter focused on running `select` statements and queries that yield no results, such as an `update` statement. We covered different authentication modes to provide the most secure connection possible. Then, we discussed calling stored procedures and how to support Output parameters using the SQL Command Parameters collection. We stepped through how the `umd_database` module integrates these features into one consistent interface. Beyond showing how to connect to databases and execute queries, the goal of this chapter was to demonstrate how to create reusable code to extend PowerShell and simplify your development work.