Stairway to U-SQL Level 19: Azure Data Lake Metadata with PowerShell

**By** [**Mike McQuillan**](http://www.sqlservercentral.com/Authors/Articles/Mike_McQuillan/586478/)**, 2018/03/07**

**FROM:** [**http://www.sqlservercentral.com/articles/Azure/168471/?utm\_source=SSC&utm\_medium=pubemail**](http://www.sqlservercentral.com/articles/Azure/168471/?utm_source=SSC&utm_medium=pubemail)

Time to wrap up our PowerShell investigation! In this final article (for now) on PowerShell and Azure Data Lakes, we’ll see how PowerShell can be used to obtain metadata on all of the items in an Azure Data Lake. This includes folders, files, databases, and database objects. There’s still more PowerShell stuff to come as this Stairway progresses, but it will be blended into other articles (including the very next one!).

Considerations

Most of the cmdlets used to interrogate objects in the Data Lake require a path to be specified. The key thing to remember with paths in Azure is they are case-sensitive. If even one letter is in the wrong case, the command will fail with a path not found error. Also, be aware that paths for the Azure Data Lake Store use a forward slash (“/”) as a separator, whilst paths for Azure Data Lake Analytics use a period (“.”).

Setting Up the Script

Just like we’ve done in previous articles looking at PowerShell, we use PowerShell ISE for development. Refer back to [Level 17](http://www.sqlservercentral.com/articles/U-SQL/161910/) for information on this. Every cmdlet we run needs to run in the context of an Azure login, which means we need to re-use the login function we wrote in Level 17. It’s reproduced here:

# Login to the Data Lake using a context file

function Perform-AdlLogin ([string] $contextPath)

{

$contextFileExists = Test-Path $contextPath

Write-Output "Logging in using $contextPath"

Try

{

if ($contextFileExists)

{

# Login to Azure using saved context

Import-AzureRmContext -Path $contextPath

}

else

{

Write-Output "The context file does not exist: $contextPath"

break

}

}

Catch

{

Write-Output "Logging in from context file failed - check your context file."

break

}

With the ability to log-in in place, we can start writing some useful functions. We’ll write a function to check if a database object exists first, then follow it up with a function that returns information on the databases in the Data Lake.

Checking if a Database Object Exists

The cmdlet to use here is *Test-AdlCatalogItem*. This cmdlet accepts an item type and a path for the object name. The cmdlet returns true or false. Typical usage may look like this:

Test-AdlCatalogItem -Account "sqlservercentral" -ItemType Table -Path "UkPostcodes.Postcodes.Postcodes"

The *Test-AdlCatalogItem’s* **ItemType** parameter supports the following types of item:

* Assembly
* Credential
* Database
* ExternalDataSource
* Package
* Procedure
* Schema
* Secret
* Table
* TablePartition
* TableStatistics
* TableValuedFunction
* Types
* View

All of these require the **Path** parameter to be specified, except for **Database**.

Let’s wrap this cmdlet up in a nice little function, which will accept a database name, object type and object name. It’s designed to check if items within a database exist.

function Check-Database-Object-Exists

([string] $accountName, [string] $contextPath, [string] $databaseName, [string] $objectType, [string] $objectName)

{

Perform-AdlLogin $contextPath

$fullPath = "$($databaseName).$($objectName)"

$ItemExists = Test-AdlCatalogItem -Account $accountName -ItemType $objectType -Path $fullPath

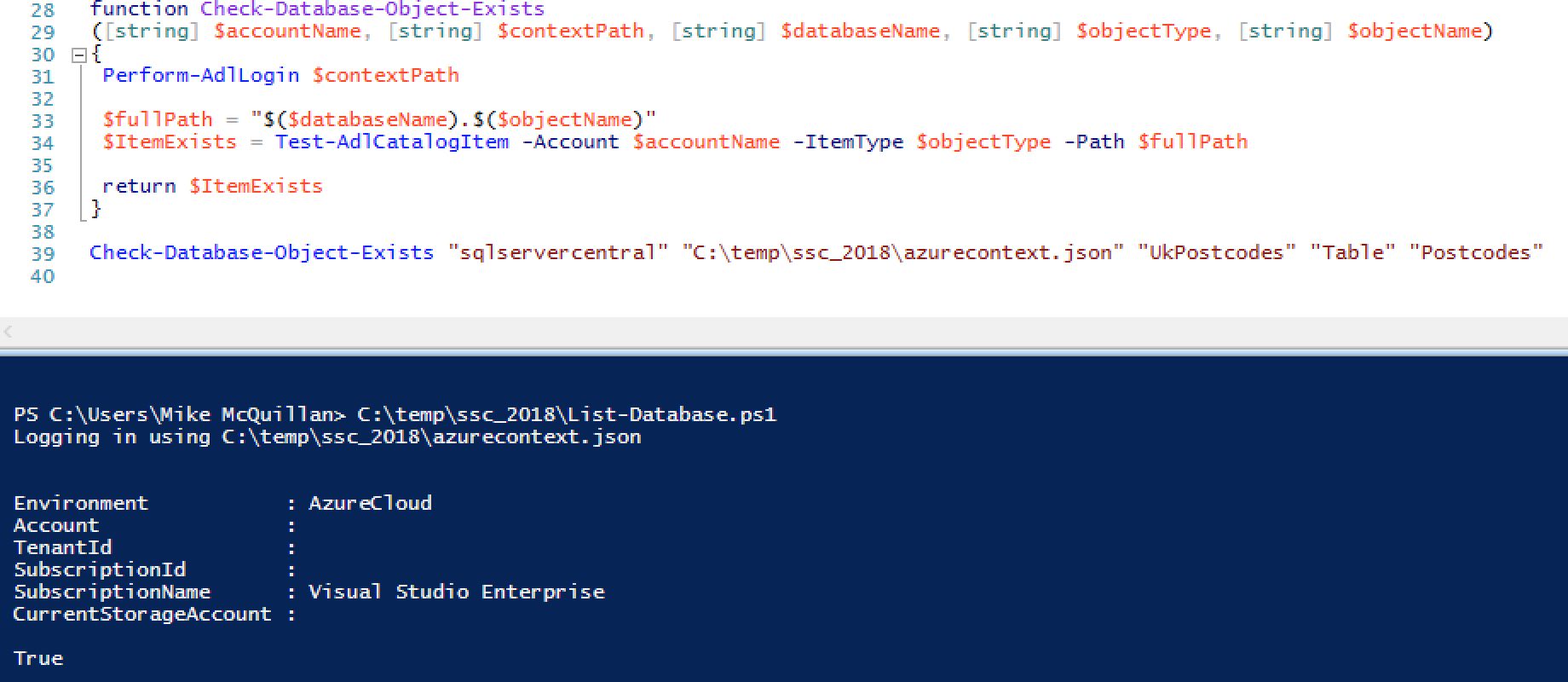
return $ItemExists

}

The **$contextPath** variable is used to perform the login. The path to the object is obtained by merging the database and object names together, after which the check can then be made by calling the *Test-AdlCatalogItem* cmdlet. Say we wanted to check if a table called *Postcodes* exists in the *UkPostcodes*database:

Check-Database-Object-Exists "sqlservercentral" "C:\temp\ssc\_2018\azurecontext.json" "UkPostcodes" "Table" "Postcodes"

A table called Postcodes does indeed exist, so this returns true:



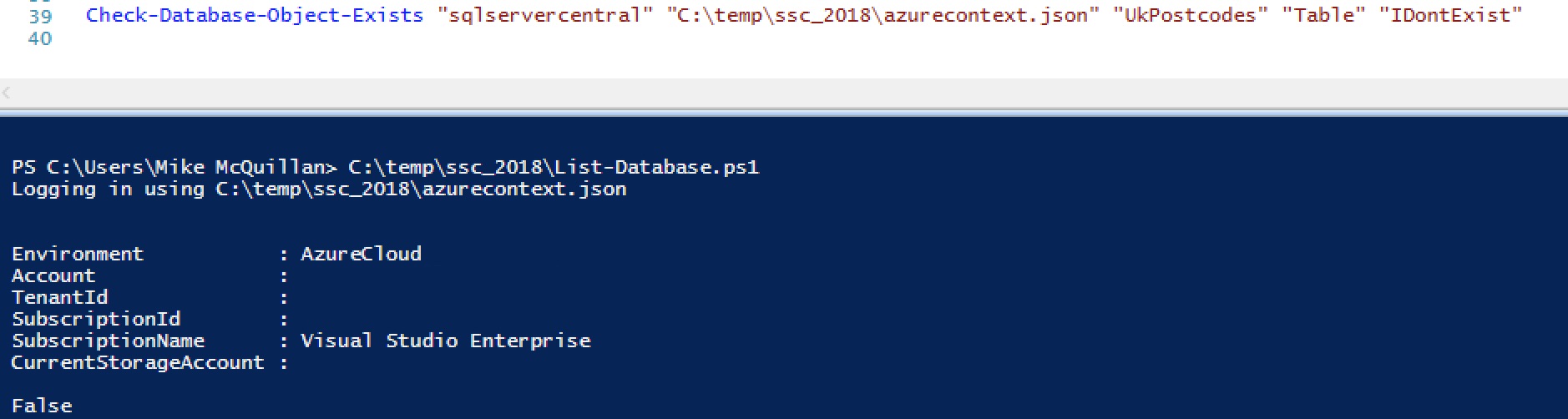
http://www.sqlservercentral.com/Resources/Images/zoom.gif[Zoom in](javascript:;)  |  [Open in new window](javascript:;)

The full path including the schema can be passed, which will also return true:

Check-Database-Object-Exists "sqlservercentral" "C:\temp\ssc\_2018\azurecontext.json" "UkPostcodes" "Table" "Postcodes.Postcodes"

But if an invalid path is given, false is returned.

Check-Database-Object-Exists "sqlservercentral" "C:\temp\ssc\_2018\azurecontext.json" "UkPostcodes" "Table" "IDontExist"



http://www.sqlservercentral.com/Resources/Images/zoom.gif[Zoom in](javascript:;)  |  [Open in new window](javascript:;)

Listing Database Objects

The next function we’ll create makes judicious use of *Get-AdlCatalogItem*. This cmdlet returns information on specific items in the Data Lake – we briefly saw this in action when we were discussing the Data Lake Catalog in [article 16](http://www.sqlservercentral.com/articles/Azure/161158/). It accepts the target account, the item type, and the path (the path isn’t needed if the Item Type is set to **Database**). The supported item types are the same as for *Test-AdlCatalogItem* – take a look at the list above.

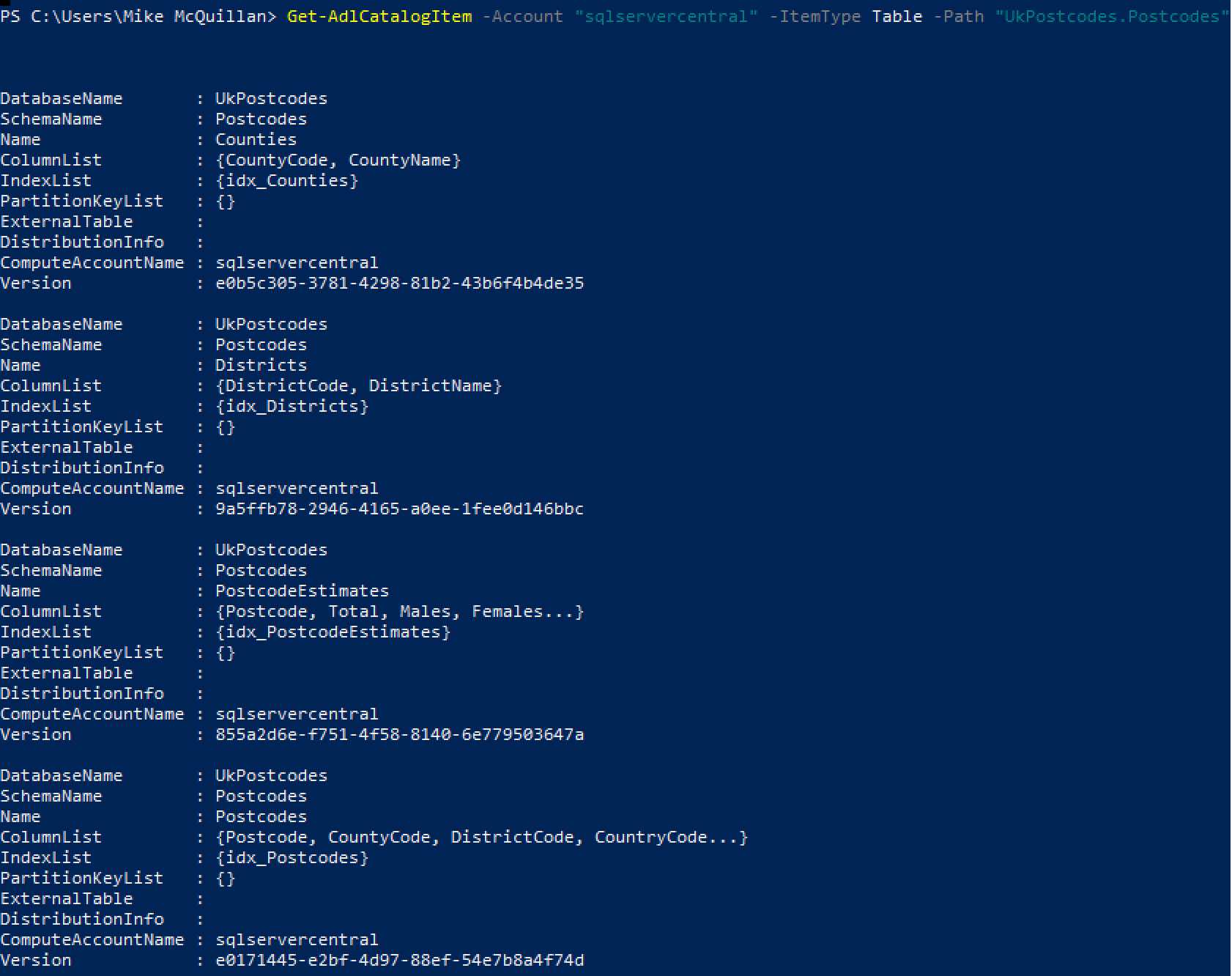
Each of the database object types has different properties. You can find out which properties each object type has by assigning the output of the relevant *Get-AdlCatalogItem* call, then requesting the properties for the object.

Identifying Object Properties

Run this command and you might expect to see details on the Postcodes table.

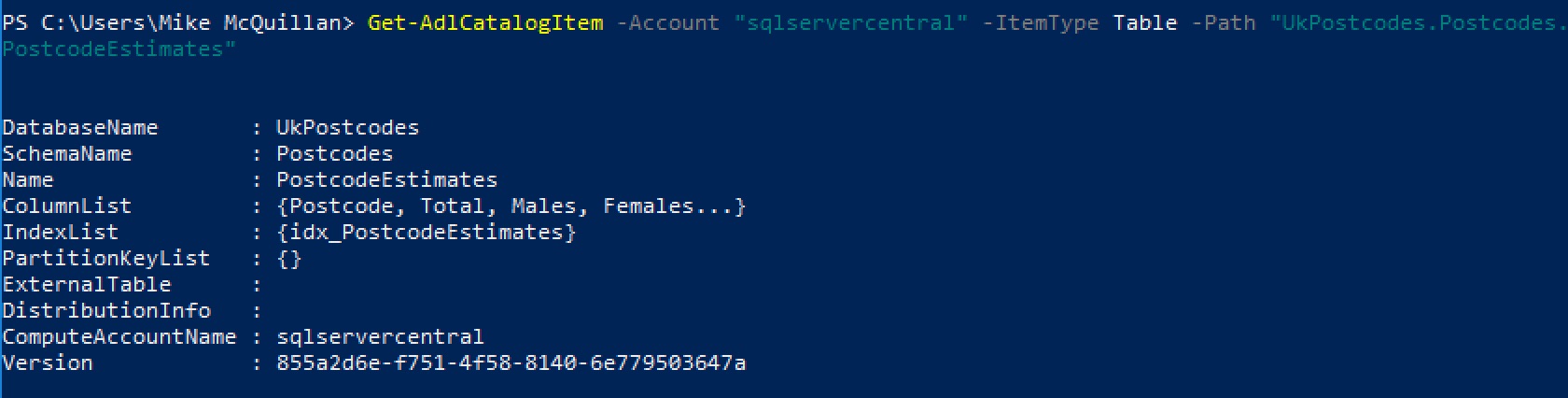
Get-AdlCatalogItem -Account "sqlservercentral" -ItemType Table -Path "UkPostcodes.Postcodes"

But no! What you actually see is details on all tables in the Postcodes schema.



http://www.sqlservercentral.com/Resources/Images/zoom.gif[Zoom in](javascript:;)  |  [Open in new window](javascript:;)

This actually serves to show the properties for each table. If you wanted to see the details for one table, you change the path to include the table name. So the value of the **path** parameter becomes **UkPostcodes.Postcodes.PostcodeEstimates**, for example.



http://www.sqlservercentral.com/Resources/Images/zoom.gif[Zoom in](javascript:;)  |  [Open in new window](javascript:;)

There’s a little trick you can use to view the properties of any object. Assign the output of a command to a variable and pipe the variable to the Select-Object cmdlet. Here’s an example:

$table = Get-AdlCatalogItem -Account "sqlservercentral" -ItemType Table -Path "UkPostcodes.Postcodes.PostcodeEstimates"

$table | Select-Object -Property \*

Function to List Database Objects

A few levels ago, I mentioned it might be fun (I don’t get out much) to write a method which listed all objects in a database. Here’s a function which returns a subset of a database’s objects – the schemas, tables, table-valued functions and stored procedures.

function Get-Database-Info([string] $accountName, [string] $contextPath, [Parameter(Mandatory=$false)][string] $databaseName)

{

Write-Output $contextPath

Perform-AdlLogin $contextPath

if (!$databaseName)

{

$databases = Get-AdlCatalogItem -Account $accountName -ItemType Database

}

else

{

$databases = Get-AdlCatalogItem -Account $accountName -ItemType Database -Path $databaseName

}

foreach ($db in $databases)

{

Write-Output "DATABASE NAME: $($db.Name)"

Write-Output "BELONGS TO ACCOUNT: $($db.ComputeAccountName)"

Write-Output "----------"

Write-Output "SCHEMAS"

Write-Output "----------"

$schemas = Get-AdlCatalogItem -Account $accountName -ItemType Schema -Path $db.Name

foreach ($schema in $schemas)

{

Write-Output $schema.Name

}

Write-Output ""

Write-Output "----------"

Write-Output "TABLES"

Write-Output "----------"

$tables = Get-AdlCatalogItem -Account $accountName -ItemType Table -Path $db.Name

foreach ($table in $tables)

{

Write-Output "TABLE NAME: $($table.Schema).$($table.Name)"

Write-Output "COLUMN COUNT: $($table.ColumnList.Count)"

Write-Output ""

}

Write-Output "----------"

Write-Output "TABLE VALUED FUNCTIONS"

Write-Output "----------"

$functions = Get-AdlCatalogItem -Account $accountName -ItemType TableValuedFunction -Path $db.Name

foreach ($function in $functions)

{

Write-Output "FUNCTION NAME: $($function.Schema).$($function.Name)"

Write-Output ""

}

Write-Output "----------"

Write-Output "PROCEDURES"

Write-Output "----------"

foreach ($schema in $schemas)

{

$path = "$($db.Name).$($schema.Name)"

$procs = Get-AdlCatalogItem -Account $accountName -ItemType Procedure -Path $path

foreach ($proc in $procs)

{

Write-Output "PROC NAME: $($proc.Schema).$($proc.Name)"

}

}

}

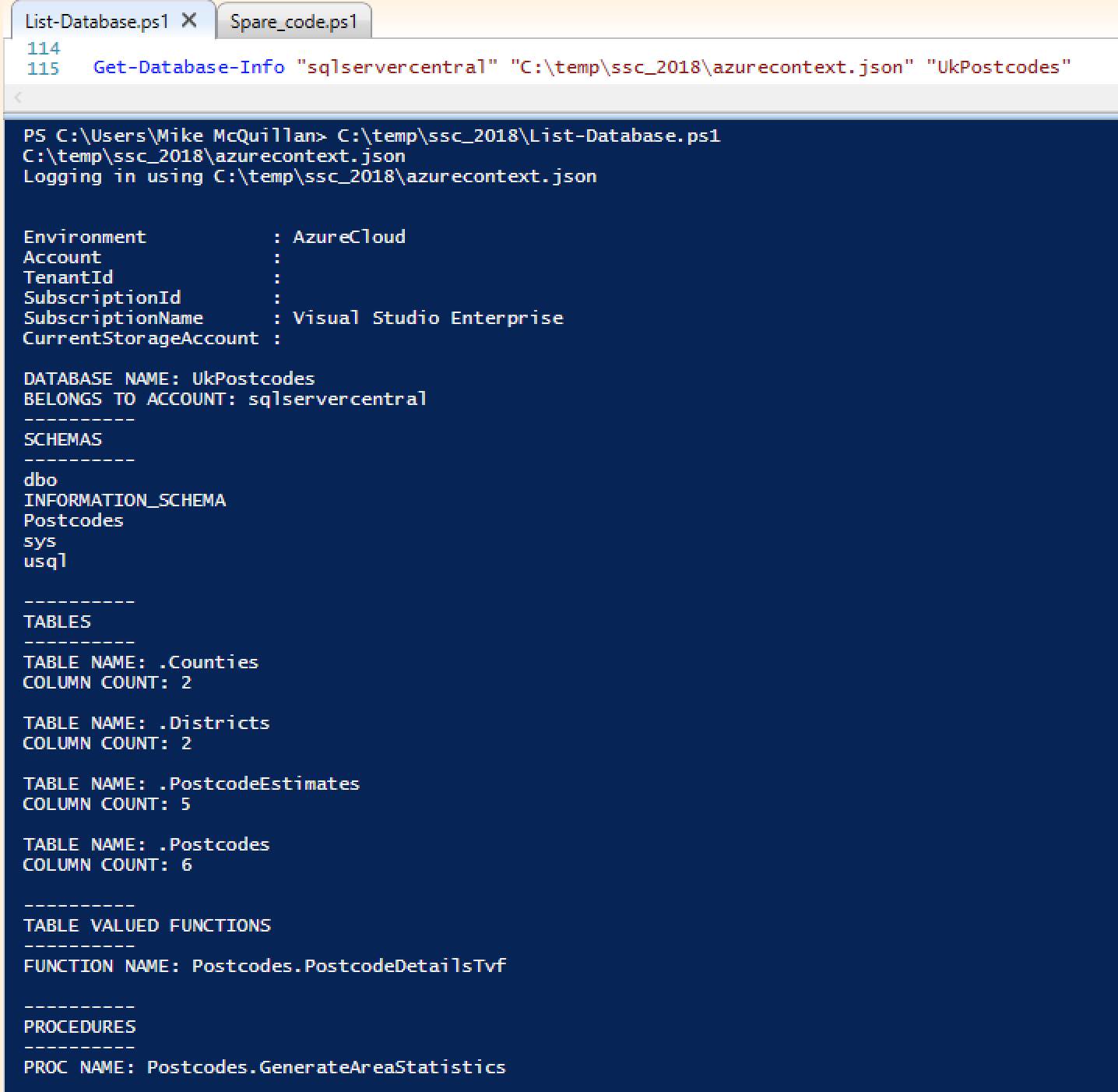
*Get-Database-Info* accepts the ubiquitous **$accountName** and **$contextPath** parameters, which tell the function which Data Lake account to use and how to log in to Azure. The last parameter is optional. If **$databaseName** is passed, just the details for that particular database are returned. If the parameter is not specified, details for all databases in the Data Lake account are returned.

Seeing as my Data Lake account only has a master database and the UkPostcodes database we’ve been building up throughout this series, I don’t have a plethora of databases to select from! Here’s a call which returns the info for UkPostcodes.

*Get-Database-Info* accepts the ubiquitous **$accountName** and **$contextPath** parameters, which tell the function which Data Lake account to use and how to log in to Azure. The last parameter is optional. If **$databaseName** is passed, just the details for that particular database are returned. If the parameter is not specified, details for all databases in the Data Lake account are returned.

Seeing as my Data Lake account only has a master database and the UkPostcodes database we’ve been building up throughout this series, I don’t have a plethora of databases to select from! Here’s a call which returns the info for UkPostcodes.

Get-Database-Info "sqlservercentral" "C:\temp\ssc\_2018\azurecontext.json" "UkPostcodes"



http://www.sqlservercentral.com/Resources/Images/zoom.gif[Zoom in](javascript:;)  |  [Open in new window](javascript:;)

The code for the function is pretty simple. It starts by using *Get-AdlCatalogItem* to obtain the requested database (or all databases). It then moves into a loop over each database, spitting out the database details, then schema details, tables, table-valued functions and finally procedures. Almost of all of these work in a similar manner. The table-valued function code just pops out the name of each function returned by *Get-AdlCatalogItem*.

$functions = Get-AdlCatalogItem -Account $accountName -ItemType TableValuedFunction -Path $db.Name

foreach ($function in $functions)

{

Write-Output "FUNCTION NAME: $($function.SchemaName).$($function.Name)"

Write-Output ""

}

The function name is returned by combining the function object’s **SchemaName** and **Name** properties. The path used is just **$db.Name**, which in this case is “UkPostcodes”. The only object which doesn’t work in this fashion is procedure, which utilises two loops.

foreach ($schema in $schemas)

{

$path = "$($db.Name).$($schema.Name)"

$procs = Get-AdlCatalogItem -Account $accountName -ItemType Procedure -Path $path

foreach ($proc in $procs)

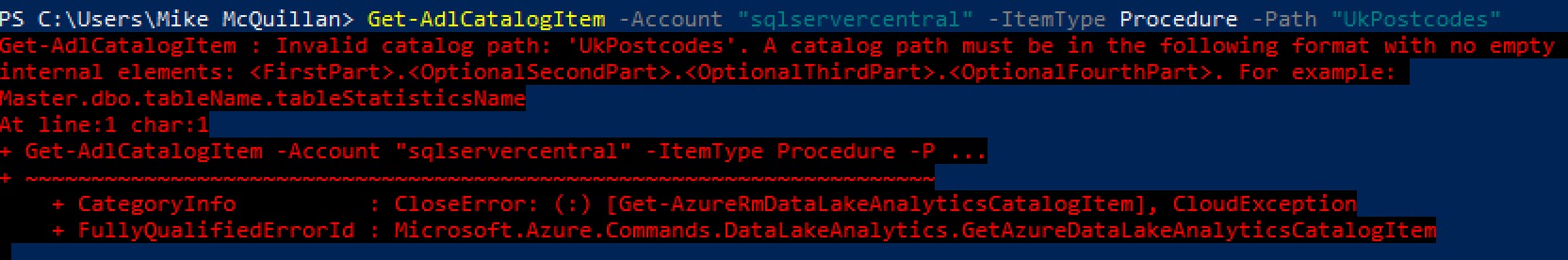
{

Write-Output "PROC NAME: $($proc.SchemaName).$($proc.Name)"

}

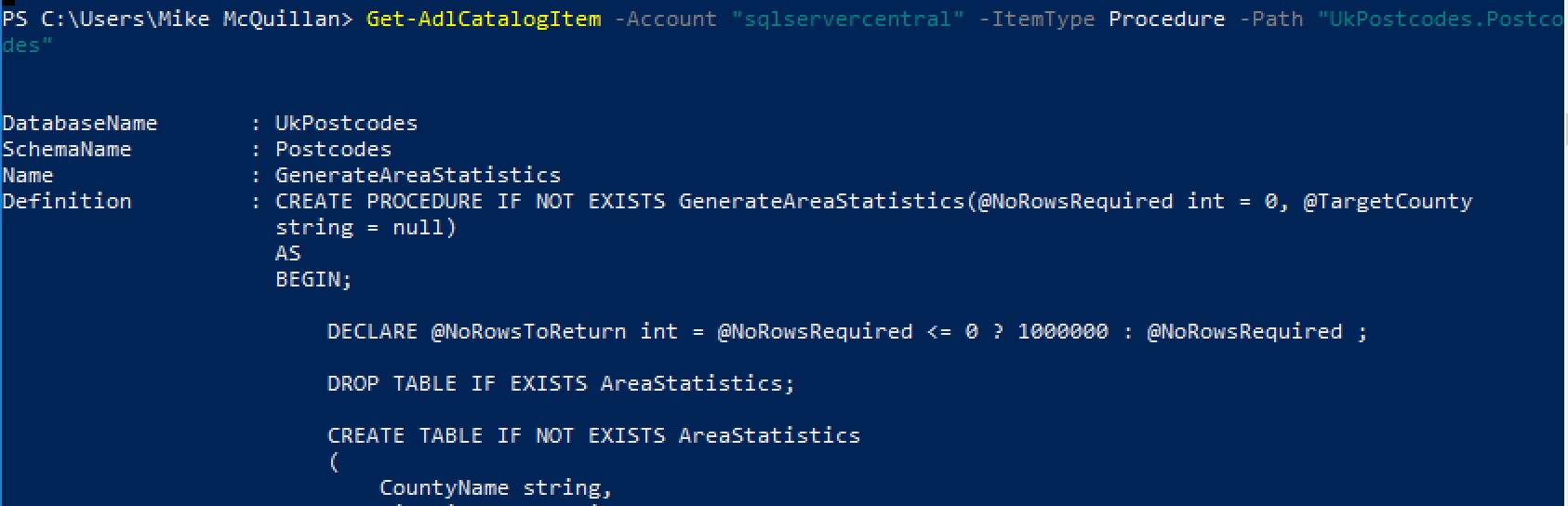
}

What’s so special about procedures then? Why do they need a loop over the schemas first? If we run *Get-AdlCatalogItem* for **Procedure** with just the name of the database, here’s what is returned:



http://www.sqlservercentral.com/Resources/Images/zoom.gif[Zoom in](javascript:;)  |  [Open in new window](javascript:;)

An invalid path error! If it’s run with the path **UkPostcodes.Postcodes** (Postcodes being the schema in which a stored procedure exists), everything is cool.



http://www.sqlservercentral.com/Resources/Images/zoom.gif[Zoom in](javascript:;)  |  [Open in new window](javascript:;)

So the schema name is needed for procedures, hence the extra loop. A bit of an anomaly, but hey.

Object Properties

For reference, here are some of the more interesting properties each of the objects returned by *Get-AdlCatalogItem* has.

*Database*

* ComputeAccountName
* Name
* Version

Schema is the same as above, except it has an additional *DatabaseName* property.

*Table*

* DatabaseName
* SchemaName
* Name
* ColumnList
* IndexList
* PartitionKeyList
* ExternalTable
* DistributionInfo

*Table-Valued Function and Procedure*

* DatabaseName
* SchemaName
* Name
* Definition

A Few More Cmdlets

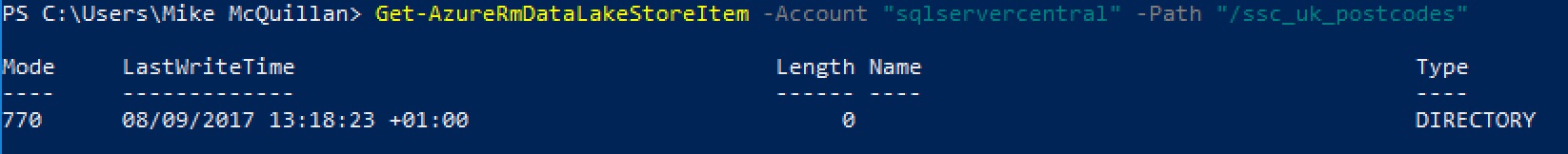
We’ve nearly finished our PowerShell odyssey. Let’s take a last glance at a few other cmdlets you might find useful.

[*Get-AdlAnalyticsAccount*](https://www.sapien.com/powershell/alias/get-adlanalyticsaccount/)

This returns general information about the requested Analytics account – its location, linked store accounts, endpoint and so on. You can also supply an optional parameter limiting the output to a particular resource group.

[*Get-AzureRmDataLakeStoreItem*](https://docs.microsoft.com/en-us/powershell/module/azurerm.datalakestore/get-azurermdatalakestoreitem?view=azurermps-5.2.0)

A cmdlet which reveals the details of a file or folder at a particular path, just like in the example shown.



http://www.sqlservercentral.com/Resources/Images/zoom.gif[Zoom in](javascript:;)  |  [Open in new window](javascript:;)

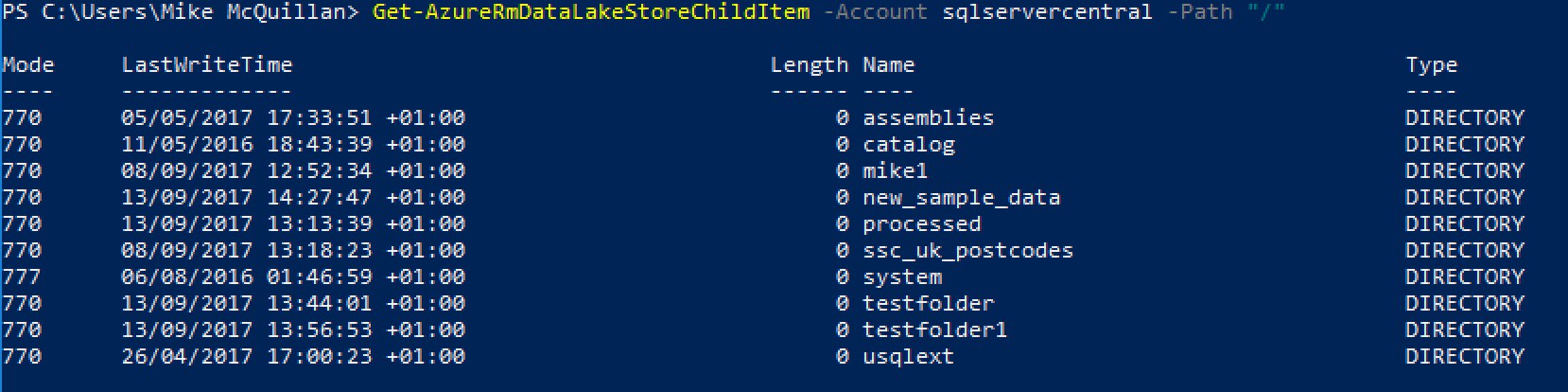
[*Get-AzureRmDataLakeStoreItemContent*](https://docs.microsoft.com/en-us/powershell/module/azurerm.datalakestore/get-azurermdatalakestoreitemcontent?view=azurermps-5.2.0)

This is a very handy cmdlet to know about. As you may have guessed from the name, it returns the contents of the file specified in the path. This command, for instance, would return the contents of the England\_Counties.csv file.

Get-AzureRmDataLakeStoreItemContent -Account sqlservercentral -Path "/ssc\_uk\_postcodes/counties/England\_Counties.csv"

[Get-AzureRmDataLakeStoreChildItem](https://docs.microsoft.com/en-gb/powershell/module/AzureRM.DataLakeStore/Get-AzureRmDataLakeStoreChildItem?view=azurermps-5.2.0)

Do you need to know the children in a particular folder of your Data Lake Store? Then this is just what you are looking for!



http://www.sqlservercentral.com/Resources/Images/zoom.gif[Zoom in](javascript:;)  |  [Open in new window](javascript:;)

Where Next?

Unsurprisingly, Microsoft has documentation on both the [Data Lake Analytics](https://docs.microsoft.com/en-gb/powershell/module/AzureRM.DataLakeAnalytics/?view=azurermps-5.2.0) and [Data Lake Store](https://docs.microsoft.com/en-gb/powershell/module/AzureRM.DataLakeStore/?view=azurermps-5.2.0) cmdlets. Take a look – anything you want to do with your Data Lake can be done in PowerShell. Automate away!

Summary

As mentioned earlier, this article concludes our look at Azure Data Lake PowerShell. We’ve seen how to inspect a Data Lake and pull out metadata about the items in our Data Lake.

With a solid PowerShell grounding behind us, we’re now going to look at accessing external data using U-SQL and the Azure Data Lake. So join me next time, when we look at how data in an Azure SQL database can be used in a Data Lake.