source: http://www.MSSQLTips.com/tip.asp?id=3544 -- printed: 3/11/2015 10:40:22 PM

Using PowerShell With Configuration Tables in SQL Server

Written By: Tim Smith -- 3/10/2015

Problem

I have a need to run the same processes on multiple SQL Servers and I want to use PowerShell, but I also want to pass in values from a configuration table to make this dynamic. What is an example of using PowerShell with configuration tables in SQL Server and where do you typically use this?

Solution

For some development purposes, configuration tables offer an advantage, as we can simply edit their values without changing an entire program, or manually changing values in code. It also offers an alternative to some scripting; for instance, copying procedures from a template server to other servers, though scripting also has advantages. In a few configuration cases, where we see frequent change, I would strongly suggest using a NoSQL database for configurations, such as MongoDB (looping through the keys and obtaining the values). For instance, if we use a configuration table for connection strings, I may have connection strings where I use a username and password, while in other cases, I won't. The same might be true for using no timeouts or timeouts, and this can be true for using configuration tables to connect to APIs (what happens if they decide to use JSON over a CSV file?). A fixed schema is not ideal in these cases, if we expect that we'll face changes and NoSQL offers the flexibility of no fixed schemas. I can perform a loop over all the keys in JSON, getting their names and values stored in them and if those every change, my code will automatically pick up on those changes.

For configurations that seldom (or never) change, we'll use a table structure in SQL Server and have PowerShell access the values in the table. Let's use an example of a configuration table with Ola Hallengren's re-indexing script. Suppose that we have many servers and databases and we have different approaches to re-indexing for each, we can use the below T-SQL structure for our re-indexing and provide the necessary parameters for each server:

```
---- The parameters will be used, but their values may change per server
EXECUTE Maintenance.dbo.stp_IndexOptimize @Databases = ''
        , @FragmentationLow = ''
        , @FragmentationMedium = ''
        , @FragmentationHigh = ''
        , @FragmentationLevel1 =
        , @FragmentationLevel2 =
        , @PageCountLevel =
```

We want to run the re-indexing, but for different servers, our parameters might be different. A configuration table can make this easier, so using T-SQL we'll build a re-indexing configuration table and add a few test values:

```
CREATE TABLE tb_IndexingConfigurations(
    ServerName VARCHAR(500),
    DatabaseNames VARCHAR(1000),
    FragLow VARCHAR(200),
    FragMedium VARCHAR(200),
    FragHigh VARCHAR(200),
    FragLevel1 SMALLINT,
    FragLevel2 SMALLINT,
    PageCountLevel INT
)

----  The below values are contrived
```

```
INSERT INTO tb_IndexingConfigurations
VALUES ('PRODSERVERONE','USER_DATABASES',NULL,'INDEX_REORGANIZE','INDEX_REBUILD_ONLINE,INDEX_REORGANIZE',
 , ('PRODSERVERTWO','USER_DATABASES',NULL,'INDEX_REORGANIZE','INDEX_REBUILD_ONLINE,INDEX_REORGANIZE',20,4
 , ('PRODSERVERTHREE','USER_DATABASES',NULL,'INDEX_REORGANIZE','INDEX_REBUILD_ONLINE,INDEX_REORGANIZE',5,
```

Because in this case, we're performing maintenance, we'll run this out of a maintenance database, so I've edited my Execute-SQL PowerShell script to use the database Maintenance in its connection string, instead of passing in the database. This script will be used to execute Ola's maintenance script:

```
Function Execute-Maintenance {
    Param (
    [string]$server
    , [string]$commandtext
    )
    Process
    {
        ###  T-SQL re-indexing script by Ola Hallengren: https://ola.hallengren.com/sql-server-index-and-
  ###  Assumes a maintenance database with Ola Hallengren's procedures
        $scon = New-Object System.Data.SqlClient.SqlConnection
        $scon.ConnectionString = "Data Source=$server;Initial Catalog=Maintenance;Integrated Security=tru

  $cmd = New-Object System.Data.SqlClient.SqlCommand
        $cmd.Connection = $scon
        $cmd.CommandText = $commandtext
        $cmd.CommandTimeout = 0

  try
        {
            $scon.Open()
            $cmd.ExecuteNonQuery() | Out-Null
        }
        catch
        {
            Write-Warning $_
        }
        finally
        {
            $scon.Dispose()
            $cmd.Dispose()
        }
    }
}
```

Now, we need to read from our configuration table. For configurations, we need an identifier (in this case, **value**) which tells our query what to look for in our table (**table**) and in this case, the identifier is the server name. For this script, *these must be unique*, otherwise, we may end up with the wrong values, unless we want to add to the query more specific identifiers, such as the date the value was added. In cases of using configuration tables for ETL, Azure work, or creating websites, our query might be looking at other unique values (instead of **ServerName**). We also will then select what columns we want (**column**). This second part is important because in some cases with configuration tables, we may skip columns; it depends on how we're using the configuration tables.

```
Function Read-ConfigurationTable {
    Param(
    [string]$server
 , [string]$table
    , [string]$value
    , [string]$column
    )
    Process
    {
        $scon = New-Object System.Data.SqlClient.SqlConnection
  ###  In this script's case, configuration tables are stored in a database called Configuration.
        $scon.ConnectionString = "Data Source=$server;Initial Catalog=Configuration;Integrated Security=t
        $cmd = New-Object System.Data.SqlClient.SqlCommand
        $cmd.Connection = $scon
        $cmd.CommandText = "SELECT * FROM $table WHERE ServerName = '$value'"
        try
        {
            $scon.Open()
            $sqlread = $cmd.ExecuteReader()

            while ($sqlread.Read())
            {
                $returnvalue = $sqlread["$column"]
            }
        }
        catch
        {
            Write-Warning $_
        }
        finally
        {
            $scon.Close()
            $scon.Dispose()
        }
        if ($returnvalue -eq $null)
        {
            $returnvalue = "NULL"
        }
        return $returnvalue
    }
}
```

With the above function, we can now obtain our configuration values from our configuration table. For the re-indexing, we will call a re-indexing function, specifically built for the structure of our re-indexing from Ola Hallengren's procedure. If we had a configuration table for connection strings with three columns, we would obtain the values of those columns using similar code, but would want a function for it:

```
Function Run-OlaReindexing {
    Param(
    [string]$server
 , [string]$table
    , [string]$value
    )
    Process
    {
        [string]$dbs = Read-ConfigurationTable -server $server -table $table -value $value -column "Datab
        [string]$f_low = Read-ConfigurationTable -server $server -table $table -value $value -column "Fra
        [string]$f_med = Read-ConfigurationTable -server $server -table $table -value $value -column "Fra
        [string]$f_high = Read-ConfigurationTable -server $server -table $table -value $value -column "Fr
        [string]$f_l1 = Read-ConfigurationTable -server $server -table $table -value $value -column "Frag
        [string]$f_l2 = Read-ConfigurationTable -server $server -table $table -value $value -column "Frag
        [string]$pagecount = Read-ConfigurationTable -server $server -table $table -value $value -column 

        $runindex = "EXECUTE Maintenance.dbo.stp_IndexOptimize @Databases = '$dbs'
                , @FragmentationLow = $f_low
                , @FragmentationMedium = '$f_med'
                , @FragmentationHigh = '$f_high'
                , @FragmentationLevel1 = $f_l1
                , @FragmentationLevel2 = $f_l2
                , @PageCountLevel = $pagecount
        "
        Execute-Maintenance -server $server -commandtext $runindex
    }
}

### We could loop through an array as well
Run-OlaReindexing -server "MaintenanceServer" -table "tb_IndexingConfigurations" -value "PRODSERVERONE"
Run-OlaReindexing -server "MaintenanceServer" -table "tb_IndexingConfigurations"  -value "PRODSERVERTWO"
Run-OlaReindexing -server "MaintenanceServer" -table "tb_IndexingConfigurations"  -value "PRODSERVERTHREE
```

For some development purposes, configuration tables offer benefits and large amounts of code and approach re-use and, in some cases, can be an alternative to scripting. With an added front-end allowing for ease of updating values, PowerShell can do most of the work involving executing what needs to be completed.

Next Steps

- Consider what would benefit from a configuration table approach, such as connection strings, re-indexing, backing up databases, etc.
- Test.
- Read more PowerShell tips