### [Performing an INSERT from a PowerShell script](http://www.sqlservercentral.com/blogs/nycnet/2013/05/24/performing-an-insert-from-a-powershell-script/)

* Posted on 24 May 2013
* [Comments](http://www.sqlservercentral.com/blogs/nycnet/2013/05/24/performing-an-insert-from-a-powershell-script/#comments)
* [](http://www.sqlservercentral.com/blogs/nycnet/2013/05/24/performing-an-insert-from-a-powershell-script/#comments) Briefcase
*  [Print](http://www.sqlservercentral.com/blogs/nycnet/2013/05/24/performing-an-insert-from-a-powershell-script/print/)

This code demonstrates how to do an INSERT into SQL Server from a PowerShell script using an ADO.NET command object with strongly-typed parameters. This script issues a DIR \*.\* command and inserts the results to the database. While this example is not particularly useful, it is hopefully very simple and understandable to explain the concepts.

In general, parameterized SQL queries (like the below) perform better and are less vulnerable to SQL injection attacks than SQL queries created using string concatenation\*.

The script code displayed in the blog post performs several INSERT statements using a row-by-row method. As a bonus, I have uploaded a version of this script to SkyDrive that uses two flavors of Table-Valued parameters which should give you better performance for big datasets.

**Download Scripts:**

[SkyDrive link to Row by Row only Script](http://sdrv.ms/12SP6gF) (has less commentary than this blog post, but code is identical)

[SkyDrive link to Row by Row and Table Valued Parameter Script](http://sdrv.ms/12SPcVC)

**Note:** You have to edit the above files in a text editor to set the SQL Server name and DB name. Once you edit the file it is no longer considered "remote" so you can run the file as long as the PowerShell ExecutionPolicy is set to RemoteSigned.

To try this script, first create a table in a junk database on a test SQL server:

CREATE TABLE FilesInFolder (

ID INT IDENTITY(1,1) NOT NULL,

TheFileName NVARCHAR(260) NOT NULL,

FileLength BIGINT NOT NULL,

LastModified DATETIME2 NOT NULL,

PRIMARY KEY (ID)

);

Then paste the following code into a .PS1 file (you have to edit the $DBServer and $DBName values first) and [run it using Powershell](http://ss64.com/ps/syntax-run.html):

function Do-FilesInsertRowByRow ([Data.SqlClient.SqlConnection] $OpenSQLConnection) {

$sqlCommand = New-Object System.Data.SqlClient.SqlCommand

$sqlCommand.Connection = $sqlConnection

# This SQL query will insert 1 row based on the parameters, and then will return the ID

# field of the row that was inserted.

$sqlCommand.CommandText = "SET NOCOUNT ON; " +

"INSERT INTO dbo.FilesInFolder (TheFileName,FileLength,LastModified) " +

"VALUES (@TheFileName,@FileLength,@LastModified); " +

"SELECT SCOPE\_IDENTITY() as [InsertedID]; "

# I am adding the parameters without values outside the loop. This means that inside the

# loop all I have to do is assign the values and say "run" - much less work than setting

# up everything from scratch in each iteration.

# Also notice the doubled-up (()) - this is how you create a new object and then

# immediately pass it as a function parameter in PowerShell.

# Next, the class names in square brackets are .NET types. Powershell assumes System.,

# so you can omit it. The double-colon is how you reference enumeration members.

# Finally, I am piping the output of these calls to Out-Null since otherwise PowerShell

# would output the properties of the command object to the console on each call since

# there is no assignment made to a variable or another pipeline destination.

$sqlCommand.Parameters.Add((New-Object Data.SqlClient.SqlParameter("@TheFileName",[Data.SQLDBType]::NVarChar, 260))) | Out-Null

$sqlCommand.Parameters.Add((New-Object Data.SqlClient.SqlParameter("@FileLength",[Data.SQLDBType]::BigInt))) | Out-Null

$sqlCommand.Parameters.Add((New-Object Data.SqlClient.SqlParameter("@LastModified",[Data.SQLDBType]::DateTime2))) | Out-Null

# I love how I can foreach over a call like "dir \*.\*" in PowerShell!!

foreach ($file in dir \*.\*) {

# Here we set the values of the pre-existing parameters based on the $file iterator

$sqlCommand.Parameters[0].Value = $file.FullName

$sqlCommand.Parameters[1].Value = $file.Length

$sqlCommand.Parameters[2].Value = $file.LastWriteTime

# Run the query and get the scope ID back into $InsertedID

$InsertedID = $sqlCommand.ExecuteScalar()

# Write to the console.

"Inserted row ID $InsertedID for file " + $file.Name

}

}

# Open SQL connection (you have to change these variables)

$DBServer = "MySQLServerName\MyInstanceName"

$DBName = "MyJunkDBName"

$sqlConnection = New-Object System.Data.SqlClient.SqlConnection

$sqlConnection.ConnectionString = "Server=$DBServer;Database=$DBName;Integrated Security=True;"

$sqlConnection.Open()

# Quit if the SQL connection didn't open properly.

if ($sqlConnection.State -ne [Data.ConnectionState]::Open) {

"Connection to DB is not open."

Exit

}

# Call the function that does the inserts.

Do-FilesInsertRowByRow ($sqlConnection)

# Close the connection.

if ($sqlConnection.State -eq [Data.ConnectionState]::Open) {

$sqlConnection.Close()

}

\*By "in general", I mean that this is true most of the time, and in the few times when it is not true, the use of parameters is unlikely to leave you worse off than not using parameters. Plainly, if you are using string concatenation to write SQL statements in any language, you are doing it *very wrong* and potentially leaving your database open for attack and exploitation.

I'm a member of the SkyDrive Insiders program; you can learn more about this program [here](http://sdrv.ms/V0buD7). I published a quick-start guide to using SkyDrive at work [here](http://bit.ly/WQ5kFh). If you have any questions about SkyDrive, please ask me