[Phil Factor](http://www.red-gate.com/blog/author/phil-factor)

[Automating database deployments to and from source control using SQL Compare and PowerShell](http://www.red-gate.com/blog/automating-database-deployments-to-and-from-source-control-using-sql-compare-and-powershell" \o "Automating database deployments to and from source control using SQL Compare and PowerShell)

Posted on 27 March 2017

Guest Blogger:   
Phil Factor

This is a guest post from Phil Factor.    
Phil Factor (real name withheld to protect the guilty), aka Database Mole, has 30 years of experience with database-intensive applications.  
  
Despite having once been shouted at by a furious Bill Gates at an exhibition in the early 1980s, he has remained resolutely anonymous throughout his career.  
  
He is a regular contributor to [**Simple Talk**](http://www.red-gate.com/blog/“http:/www.simple-talk.com/”) and [**SQLServerCentral**](http://www.red-gate.com/blog/“http:/www.sqlservercentral.com/”).

Our goal is to show how to use [**SQL Compare**](http://www.red-gate.com/products/sql-development/sql-compare/) Pro command line with PowerShell to automate the following processes:

* **Getting a new database into version control**, or updating the object scripts in the VCS to reflect the latest database changes
* **Creating a new build script** for a database version, from the object scripts in the VCS
* **Creating a database migration script** to upgrade a target database to the version represented by the source object scripts

Getting a new database into version control

If we have an existing database that is not yet in source control, we can use SQL Compare to compare the source database to a directory of object scripts, which is linked to a Version Control System such as GitHub, SVN or TFS.

SQL Compare will update the target directory, producing a set of object creation scripts that represent the current state of the database metadata in the source database. If the directory is empty, SQL Compare will write out the script files for every object in the database and create the necessary subdirectories to hold them. Then, it is easy to use a version control tool such as TortoiseSVN, TortoiseGit or GitHub to create a repository.

Feodor Georgiev’s article, [How to create a directory of object-level scripts using SQL Compare](http://www.red-gate.com/blog/object-level-scripts), shows how to do this using the SQL Compare UI, mainly, but also includes a PowerShell script that automates the database-to-directory deployment using the SQL Compare Pro command line. Please refer to that article for the details on this script; here I use an adapted version of that code to generate the initial directory of scripts for my example database, [Customers](https://www.red-gate.com/wp-content/uploads/2016/12/Customers-Database-v2.sql) (the link is to the build script):

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17  18  19  20  21  22  23  24  25  26  27 | $SQLCompare="${env:ProgramFiles(x86)}\Red Gate\SQL Compare 12\sqlcompare.exe"# full path  $MyServerInstance='MyServer\MyInstance' #The SQL Server instance  $MyDatabase='Customers' #The name of the database  $MyDatabasePath =  "$($env:HOMEDRIVE)$($env:HOMEPATH)\Documents\GitHub\$MyDatabase\Source"  $MyReportDirectory = "$($env:HOMEDRIVE)$($env:HOMEPATH)\Documents\GitHub\$MyDatabase\Reports"    if (-not (Test-Path -PathType Container $MyReportDirectory))      {          # we create the report directory          New-Item -ItemType Directory -Force -Path $MyReportDirectory;      }    if (-not (Test-Path -PathType Container $MyDatabasePath))      {          # we create the script directory (normally you get this from the VCS)          New-Item -ItemType Directory -Force -Path $MyDatabasePath;      }    $AllArgs = @("/server1:$MyServerInstance", "/database1:$MyDatabase ",  "/scripts2:$MyDatabasePath", '/q', '/synch',  "/report:$MyReportDirectory\$MyDatabase.html",  "/reportType:Simple","/rad","/force" )  &$SQLCompare $AllArgs  if ($?) {'updated successfully'}  else {if ($LASTEXITCODE=63) {'Database and scripts were identical'}  else {"we had an error! (code $LASTEXITCODE)"}} |

*Listing 1: Deploying from database-to-script folder*

Of course, Listing 1 represents only the basics of what’s possible. You can set a range of other [switches](https://documentation.red-gate.com/display/SC12/Switches+used+in+the+command+line) and project [options](https://documentation.red-gate.com/display/SC12/Options+used+in+the+command+line) during command line deployments.

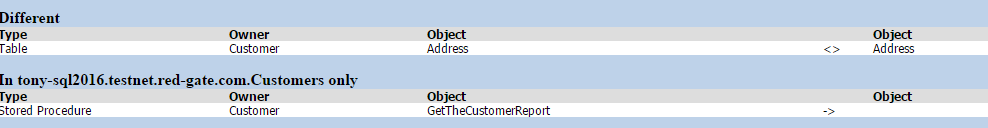
Using a SQL Compare project in a command line switch

An alternative approach is to set up a database-to-directory deployment through the SQL Compare GUI, and then save the resulting project and [execute it from a batch file](http://www.red-gate.com/blog/sql-compare-command-line) whenever you need to push changes into the VCS. This is useful if you want to use the UI to establish complex object filtering rules, and makes use of other advanced project comparison, and then automate. Command-line use is ideal for regular routine tasks that are usually scheduled.

Developers can also use the script in Listing 1 to ‘back fill’ the ‘state’ of each object in the VCS, so that its definition reflects their changes. For each tested set of changes to their development database, a developer can compare their source database to the script folder in the VCS and generate a database-to-script folder deployment script.

Each deployment will result in the VCS source file of any changed object being updated so that its definition matches that in the source database. Likewise, a file will be added to the VCS for any new object, and the file for any object that is deleted in the database will usually be deleted from the directory (occasionally, I’ve seen an empty, 0KB, file left in the directory).

Figure 1 shows a sample of the report generated after adding a stored procedure to my development database, and increasing the size of the **Customer.Address.City** column from **NVARCHAR(30)** to **NVARCHAR(40)**, and then running Listing 1.



*Figure 1*

Take care when back-filling the database ‘state’, in this manner. If you’re working on a shared database, each developer on the team will need to add a [**/filter** argument](http://documentation.red-gate.com/display/SC11/Using+filters?__hstc=81186632.a2a55ce4965e1421d4f01aabe6fe066a.1415891801020.1490010456558.1490096003939.1635&__hssc=81186632.2.1490096003939&__hsfp=941192876), to be sure to capture only his or her changes. Having updated their local working directory with any changes, each developer can push those changes to the master branch in the remote shared repository, so other developers can access them. In bigger teams, it’s potentially easier to coordinate all this through a tool such as [**SQL Source Control**](http://www.red-gate.com/products/sql-development/sql-source-control/).

Each developer can pull from the project’s GitHub repository the latest object-level scripts, with the committed changes of other developers, use SQL Compare to compare their newly-updated working folder with their development database, and generate a deployment script to update the latter with these changes. They can also use the latest set of scripts to build a new copy of the current database version.

Deploying a new database from version control

The database build mechanism must create a build script from the correct version of the object creation scripts for all necessary components, including the DDL scripts to create all the schema-scoped objects, such as tables, stored procedures, functions, views, aggregates, and synonyms.

SQL Compare can read a directory of scripts, parse them to create a ‘model’ of the database from the scripts, compare that with a target database, and generate a deployment script that will synchronize the two. When comparing to an empty target database, we can instruct SQL Compare to compare to the [**model**](https://msdn.microsoft.com/en-us/library/ms186388.aspx) database, the template from which SQL Server builds all user databases. The net result is that SQL Compare will generate a complete build script for the database, with all the objects automatically built in the correct dependency order.

Listing 2 demonstrates this approach. Having created a blank **Customers** database on the target SQL Server instance, running it will generate a database build script in the VCS project directory. If a build script already exists, we simply overwrite it in this simple example:

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17  18  19 | $SQLCompare="${env:ProgramFiles(x86)}\Red Gate\SQL Compare 12\sqlcompare.exe"# full path  $MyServerInstance='MyServer\MyInstance' #The SQL Server instance  $MyDatabase='Customers' #The name of the database  $MyDatabasePath =  "$($env:HOMEDRIVE)$($env:HOMEPATH)\Documents\GitHub\$MyDatabase\Source"  $MyBuildFile = "$($env:HOMEDRIVE)$($env:HOMEPATH)\Documents\GitHub\$MyDatabase\Build"    if (-not (Test-Path -PathType Container $MyBuildFile))      {          # we create the build directory          New-Item -ItemType Directory -Force -Path $MyBuildFile;      }    $AllArgs = @("/scripts1:$MyDatabasePath", "/server2:$MyServerInstance",        '/quiet', "/database2:model","/scriptfile:$MyBuildFile\$MyDatabase.sql")  if (Test-Path "$MyBuildFile\$MyDatabase.sql" ) {Remove-item "$MyBuildFile\$MyDatabase.sql" }  &$SQLCompare $AllArgs  if ($?) {'Script generated successfully'}     else {"we had an error! (code $LASTEXITCODE)"} |

*Listing 2: Creating a database build script*

An alternative to comparing to the model database on the target instance is simply to use the [/empty2](https://documentation.red-gate.com/display/SC12/Switches+used+in+the+command+line#Switchesusedinthecommandline-/empty2) command line switch instead, if that’s more convenient. The risk here is that the resulting database may have database option settings different to those defined in the model database for that instance.

Having verified and tested the build script, simply set the target database to **Customers** and run the script, using SSMS or SQLCMD, to deploy a new build of that database version. The team can use this same script to reproduce that exact same build on any target server.

Updating an existing database from version control

The script for updating an existing database with the latest changes in the VCS, shown in Listing 3, is suspiciously similar, except this time we are not comparing with **model** but with the target database.

Again, it will generate a migration script in the project directory for that database. You’ll need to ensure that the migration script is named in a way that indicates the source and target version of the database on which it operates. You now have a script that takes a database from one version to another, reliably, on any environment:

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17  18  19 | $SQLCompare="${env:ProgramFiles(x86)}\Red Gate\SQL Compare 12\sqlcompare.exe"# full path  $MyServerInstance='MyServer\MyInstance ' #The SQL Server instance  $MyDatabase='Customers' #The name of the database  $MyDatabasePath =  "$($env:HOMEDRIVE)$($env:HOMEPATH)\Documents\GitHub\$MyDatabase\Source"  $MyMigrationFile = "$($env:HOMEDRIVE)$($env:HOMEPATH)\Documents\GitHub\$MyDatabase\Migrations"    if (-not (Test-Path -PathType Container $MyMigrationFile))      {          # we create the migrations directory          New-Item -ItemType Directory -Force -Path $MyMigrationFile;      }    $AllArgs = @("/scripts1:$MyDatabasePath", "/server2:$MyServerInstance",        '/quiet', "/database2:$MyDatabase","/scriptfile:$MyMigrationFile\$MyDatabase.sql")  if (Test-Path "$MyMigrationFile\$MyDatabase.sql" ) {Remove-item "$MyMigrationFile\$MyDatabase.sql" }  &$SQLCompare $AllArgs  if ($?) {'Script generated successfully'}     else {"we had an error! (code $LASTEXITCODE)"} |

*Listing 3: Creating a database migration script*

Ultimately, you’ll be deploying this sort of script to a target database that contains data, so it will need careful testing to ensure that the script preserves all data correctly in the target database. Occasionally, there will be messy problems that SQL Compare can’t tackle automatically, such as when you split a table or rename a column. In such cases, you’ll need to hand craft those migration scripts, or use a migrations tool such as [**ReadyRoll**](http://www.red-gate.com/products/sql-development/readyroll/) to help.

In this simple example, we again just delete any existing migration script and replace it with the current one. In practice, you’ll have a unique naming convention for each script, based for example on the source and target database version, or on the date.

Summary

Here we’ve simply sketched out, using working examples, the ways that [**SQL Compare**](http://www.red-gate.com/products/sql-development/sql-compare/) can help with aspects of deployment. The [**SQL Toolbelt**](http://www.red-gate.com/products/sql-development/sql-toolbelt/) allows you to do much more. I’ve already shown you in this series of articles how to [automatically fill the build database with dat](https://www.red-gate.com/blog/multiple-development-databases-powershell-sql-data-generator)a for testing using [**SQL Data Generator**](http://www.red-gate.com/products/sql-development/sql-data-generator/), and how to [document a new build automatically](https://www.red-gate.com/blog/sql-doc-powershell-epub-database-documentation) via [**SQL Doc**](http://www.red-gate.com/products/sql-development/sql-doc/). It’s also very easy to provision all of your test cells with a database stocked with data via [**SQL Data Compare**](http://www.red-gate.com/products/sql-development/sql-data-compare/). Once you understand the principles you can then go on to add plenty more, such as code analysis, to the toolchain.