Phil Factor 20 May 2016

Phil Factor
20 May 2016





# How to Build and Deploy a Database from Object-Level Source in a VCS

> It is easy for someone who is developing a database to shrug and say 'if only my budget would extend to buying fancy tools, then I could start using version control when developing or maintaining databases'. Phil Factor sets out to show that there are ways of maintaining object-level source control for SQL Server databases just using what Microsoft provides, It may not be perfect, but there are ways of doing it whatever your budget.

In this article, I will show how to keep the metadata of a database in version control, using state-based object-level source code, in much the same way as you would with application code. I will show how to use it with a third-party tool, and with tools that are provided by Microsoft with the SQL Server product. I will show how to start a repository, keep it up to date, build a database and also to create a build script for changing in existing database from one version to another.

## Is a database just like an application?

It has often been written that databases are just like applications and therefore the source should be stored in version control in the same way. Well, that's true up to a point, but actually, a SQL database is unlike applications in several ways. Although you can generate much of the code for an application via a tool, the end result generally

consists of files with code and maybe configuration information.

Database objects are more disconcerting because any radical development work on a database will affect a considerable number of inter-related and dependent objects at once. As well as this, they can be altered rather than just deleted and recreated, so as to preserve data and permissions. It is perfectly easy and legitimate to alter views and tables without even seeing any code. The underlying DDL code to perform the magic is there, but the code that does the alteration can only be executed in context. When we are obliged to alter a whole number of database objects to make a change, I use entity-relationship diagramming tools, changing at a sweep a whole host of tables and the routines or 'modules' that reference them. I'm typical of database developers who will take a broad view of database structures rather than obsessing at the object-level. However, under the covers, it is still just code: This code is accessible to you via the IDE, a simple DDL trigger, the default trace or a Service-broker notification. You can collect up all these alterations and save them as change scripts. As long as everything went as planned, you can replay all the DDL scripts in the same order to create a migration script that would take a database from the version it was at before you started, to the new version.

Imagine that you have just drunk a lot of coffee and restructured a database via a variety of tools using your individual developer's login, tested it obsessively, and found it good. You want to make the change permanent for the development, so you need to check all this into source control. With a decent SQL Source Control system that is designed for your RDBMS, you give a chuckle as you see, in SSMS, all the code you've changed light up like fairy lamps, allowing you to check in your changes, object by object, with an explanation. The CREATE script of each changed or new object is saved to version control, and deletion of objects is confirmed.

What if, for some reason, you haven't such a tool? You probably haven't got the CREATE DDL script for the changed objects, though you may have the ALTER DDL code, either saved from the widgets you used to make the

changes or from an automatic collection device such as I've described. This means you've only the migration information rather than the state information. What now?

This article aims to answer this question.

# The Aim

When we make a change that transcends the simple model of working with single database objects, we would like to 'back-fill' the CREATE statements for each changed object in a source control directory so that we can get all the advantages of conventional version control, such as annotations and history. We would also like to be easily able to reconstruct any version of a database from the scripts and objects in the Version Control System (VCS), and to generate change scripts that will migrate a database from one version to another.

# Saving Database Changes to Version Control

In this simple method, you need to be careful to commit to source control only those changes to the database that you are responsible for, otherwise it is of very limited use in a team database project. If each database developer is working in their own schema, it is less likely that you will accidentally capture other changes, but as long as you don't commit them you're OK anyway.

To do this, we'll choose to use Github, though the process is very similar for any other current Version Control System. We will pretend that AdventureWorks is our development database.

First we install AdventureWorks on our development server.

Having installed GitHub Desktop we then create a repository called AdventureWorks

I'm going to use two PowerShell-based approaches: The first one uses a database comparison tool that can compare source code in files to a live database (I'm using SQL Compare) and the second uses SMO and DacFx.

Because I'm doing it two different ways, I'll save it to two different directories.

## Saving Changes via a SQL Comparison Tool

Because we are actually comparing the live database to a directory and asking SQL Compare to synchronise them, SQL Compare Pro will, when faced with an empty directory, write out the script files and create the necessary subdirectories to hold them. (SQL Compare will also do this but you would have use the GUI instead)

To put the source control directory up to date with the state of the database…

```
$SQLCompare="${env:ProgramFiles(x86)}\Red
 $MyServerInstance='MyServer\MyInstance' #
 $MyDatabase='AdventureWorks' #The name of
 $MyDatabasePath = "$($env:HOMEDRIVE)$($en

 $AllArgs = @("/server1:$MyServerInstance"
    "/scripts2:$MyDatabasePath", '/q', '/s
    "/report:$($MyDatabasePath).html", "/r

 if (-not (Test-Path -PathType Container $
 { New-Item -ItemType Directory -Force -Pa
 &$SQLCompare $AllArgs
 if ($?) { 'updated successfully' }
 else
 {
    if ($LASTEXITCODE -eq 63) { 'Database
    else { "we had an error! (code $LASTEX
 }
```

**Listing 1: PowerShell routine to save changes to a database in the local github folder ready for checking in**

If your servers are outside the domain, you would probably need credentials with SQL Server security so your $**AllArgs** settings will be different. Updates use exactly the same code (You will need to add a **/filter argument** if you are team-working so as to include only your schema). You will need to run this PowerShell routine before you start your work session on the database. When we repeat this after making changes, SQL Compare will update the directory to represent the metadata, updating only those files that represent updated objects, and deleting files that represent dropped objects. Basically, it looks after all the messy details.

After you execute this, you then go into GitHub, select the repository, click on 'Changes' and add a summary and description to any changes you've made. If you have hand-crafted object scripts with plenty of documentation and guard clauses, replace the auto-generated object-level script with your hand-crafted script. Then hit the 'commit to master' button. This has saved everything to the VCS as a set of changes at object-level that you have made. If you are working as part of a team, you will need to pull the object-level scripts with their committed changes from the project GitHub repository and update your development database with these changes. If there are a lot of changes, then SQL Compare can prepare a change-script for you to apply these changes. I do this with the GUI so I can inspect everyone else's changes.

## What if you don't have the right SQL Comparison tool?

The SMO version is more complicated, but some of this is due to the routine having to generate a sequenced list of files that you'd need to execute to manage a database build in the correct order. We'll cover this build process later in the article when we have to build a database from an object source. SMO can only record the correct order of scripting a new build from the live database. This would mean updating a simple manifest whenever you save your work to source control. The trick, then, is to create a sort of manifest file from a live database. This file is actually a SQLCMD file that can be executed from within SSMS or by the SQLCMD utility. This recreates the database in the right order.

Whenever you need to run integration tests, you can then quickly run an automated task that builds the database from scratch and then runs the appropriate integration tests on it in background.

To work with GitHub, you need to change the format of text script file that SMO generates. UTF8 is the safest. The same applies to the manifest file.

Here is the PowerShell script.

```
<# this script uses powershell v4. It will
```

```
$Server = 'PhilFactorTestServer' #the name
$Instance = 'sql2008'            #The serv
$Database = 'Adventureworks'     #the datal
#this is the directory where you wish to p
$DatabaseScriptPath = "$($env:HOMEDRIVE)$(
#in case you wish debug information
$WarningPreference='Stop' #because if it c
$VerbosePreference = "SilentlyContinue"

$DoWeLookForParentObjects = $true #don't c
$SMO = 'Microsoft.SqlServer.Management.SMO
#read in the SQLPS module if it isn't in a
import-module 'sqlps' -DisableNameChecking

trap {
      Write-Error ('Failed to access "{0}"
      $_.Exception.Message, $_.InvocationI
      exit
    }
<# SMO likes to do build scripts in Object
Table Types and Procedures come next. Then
Here we define their order, but this order
#>
$ObjectTypeOrder = @{
    'users' = 1;
    'Roles' = 2;
    'Schemas' = 3;
    'Assemblies' = 4;
    'AsymmetricKeys' = 5;
    'Certificates' = 6;
    'XmlSchemaCollections' = 7;
    'FileGroups' = 8;
    'FullTextCatalogs' = 9;
    'FullTextStopLists' = 10;
    'LogFiles' = 11;
    'PartitionFunctions' = 12;
    'PartitionSchemes' = 13;
    'PlanGuides' = 14;
    'UserDefinedTypes' = 15;
    'UserDefinedDataTypes' = 16;
    'UserDefinedTableTypes' = 17;
    'UserDefinedAggregates' = 18;
    'ApplicationRoles' = 19;
    'Rules' = 20;
    'Defaults' = 21;
    'Tables' = 22;
    'StoredProcedures' = 23;
    'UserDefinedFunctions' = 24;
    'Views' = 25;
    'DatabaseAuditSpecifications' = 26;
    'SearchPropertyLists' = 27;
    'Sequences' = 28;
    'ServiceBroker' = 29;
    'SymmetricKeys' = 30;
    'Triggers' = 31;
    'Synonyms' = 32;
    'ExtendedStoredProcedures' = 33;
    'ExtendedProperties' = 34;
}
# first we access the server using the PSP
$Srv = get-item "SQLSERVER:\SQL\$Server\$I
#now we create all our lists of objects, u
$urnCollection = new-object "$SMO.UrnColle
$OrderedURNCollection = new-object "$SMO.U
```

```
$PostTableurnCollection = new-object "$SMO
#we now determine the types we don't want
$TypesWeDontWant = '(?im)Roles|Federations
$pathsWritten=[array] @()
if ([int]$srv.version.Major -gt 10) { $Typ
# first we access the database in order to
Set-location "SQLSERVER:\SQL\$Server\$Inst
Get-ChildItem | where-object{ "$($_.PSChil
<#and now we sort them in object-dependenc
Select-Object `
@{ E = { $ObjectTypeOrder."$_" }; N = "Bui
@{ E = { $_ }; N = "ObjectType" } | sort-ob
# then we get all the objects of each type
foreach { if (test-Path "$($_.ObjectType.P
#get subtypes if necessary (stuff like ser
foreach  {
    if ($_.GetType().BaseType -notlike '*s
    else { $_ }
} |
#and we check if the object is scriptable,
where { ($_.PSobject.Members.name -match "
Foreach{
    $urn = $_.urn
    if ($CurrentSQLObject.BuildOrder -lt 2
    elseif ($CurrentSQLObject.BuildOrder -
    else { $PostTableurnCollection.add($ur
}
 <# discovering dependencies only works fo

#now  we set up the scripter object just t
$scr = New-Object "$SMO.Scripter"
#now choose options for the scripter that
$options = New-Object "$SMO.ScriptingOptio
$options.DriAll = $False
$options.AllowSystemObjects = $false
$options.WithDependencies = $False
$scr.Options = $options
$scr.Server = $srv

#we create a dependency walker object
$DependencyWalker = new-object ("$SMO.Depe
#
#now we set up an event listnenr go get pr
$ProgressReportEventHandler = [Microsoft.S
$scr.add_DiscoveryProgress($ProgressReport
#create the dependency tree
$dependencyTree = $scr.DiscoverDependencie
#and walk the dependencies to get the depe
$depCollection = $scr.WalkDependencies($de
#we just extract the root nodes and add th
$Depcollection | where { $_.IsRootNode -ne
#now we add the tail of the ordered list t
$PostTableURNCollection | foreach{ $Ordere


<#now set up the scripting options that yo
$ScrOptions = new-object ("$SMO.ScriptingO
$ScrOptions.ExtendedProperties = $true # y
$ScrOptions.IncludeIfNotExists = $false #
$ScrOptions.IncludeHeaders = $false # e.g.
$ScrOptions.ToFileOnly = $true #do not ech
$ScrOptions.DRIAll = $true # and all the c
$ScrOptions.AllowSystemObjects = $false #
$ScrOptions.Indexes = $true # Yup, these w
```

```
$ScrOptions.Triggers = $true # This should
$ScrOptions.ScriptBatchTerminator = $true
$ScrOptions.Encoding = [System.Text.Encodi

# we now create the script directory if it
if (!(Test-Path -path $DatabaseScriptPath)
{
    Try { New-Item $DatabaseScriptPath -ty
    Catch [system.exception]{
        Write-Error "error while creating
        return
    }
}
<# we now create the start of our SQLCMD s
$SQLCMDBuildScript=@"
/**
summary:   >
  This is a SQLCMD file that can be execut
  can use the SQLCMD command-line utility
Author: Phil Factor
Revision: 1.5
date: $(get-date)
example: sqlcmd -S $server\$instance -d $(

**/

SET NOCOUNT ON
GO
PRINT 'Creating the database objects in or


"@
#for each object in our ordered list of ob
$OrderedURNCollection | Foreach{
    # work out the full file path where we
    $fullPath = "$DatabaseScriptPath\$($_.
    $filename = "$(if ($_.GetAttribute('Sc
    { $($_.GetAttribute('Schema') -replace
    else { '' })$($_.GetAttribute('Name')
    # create the file path if it doesn't e
    if (!(Test-Path -path "$fullPath"))
    {
        Try { New-Item "$fullPath" -type d
        Catch [system.exception]{
            Write-Error "error while creat
            return
        }
    }
    $ScrOptions.Filename = "$($fullpath)\$
    $srv.GetSmoObject($_).script($ScrOptio
$SQLCMDBuildScript+=@"

    PRINT 'creating the $($_.Type) $($_.Ge
    go
    :r $($fullpath)\$($filename)
    :On Error exit
"@ #and write the lines into the contents
   $PathsWritten+="$($fullpath)\$($filenam

}
$SQLCMDBuildScript+=@"

PRINT 'Database creation is now complete'
GO
```

```
"@ #now finish off the file. By using the

[System.IO.File]::WriteAllLines("$Database
<# now we need to delete the files that do
dropped objects are dropped in source cont
$PathsWritten+="$DatabaseScriptPath\Databa
<# now delete anything else #>
$DeletedCount=0
Get-ChildItem -path $DatabaseScriptPath -i
  where {$Pathswritten -notcontains $_.ful
if ($DeletedCount -gt 0) {Write-Verbose "d
"Did I do well?"
```

**Listing 2: An SMO routine for writing out object-level scripts into a directory, and adding an executable manifest file**

You will see that it is more complicated. You will have ended up with all the object-level scripts in their own file, in subdirectories according to the type of object. Any object script it finds that isn't represented by an object in the database is deleted. The manifest is updated. You can do most of this from SSMS, of course, as long as you don't need the manifest to do the build (The build is essential for CI). You would need to remember to opt for 'Single file per object', not to 'include descriptive headers', and make sure you 'save as Unicode text'.

The SMO approach has another problem in that it doesn't really know what has changed from your source control directory, though there is actually a '**DateLastModified**' property on the individual database objects that enables you to script only objects that are modified later than a specified date. I don't do that in this script because it is simpler to let Git determine what has changed by doing a textual comparison. This works fine unless you let SMO put headers on the object scripts, which contain the date that these were scripted. Oh no. False positive. A text comparison is never as good as a semantic comparison.

A further complication of merely writing out all the files afresh is that the script needs to delete the file containing the object script if it isn't in the database, because it means that you've deleted the object. To do this, it has to compare the list of files it knows about with the files in the Git directory and delete any files defining objects that it doesn't know about.

In this script, I have shown how to do a single database. I've also assumed Windows Authentication. It is easy to

alter this to do all your servers and databases, but it would be a distraction for this article to show all this, and include all the PowerShell error-handling that you'll need..

### Problems with the SMO approach

For AdventureWorks, there were very few problems. Bindings to rules (with the sp_bindrule) on user-defined types are missing. The documentation (extended properties) on all the clustered indexes of primary keys is missing. If you have rules, or documentation on the indexes that support your primary keys, you're out of luck.

The use of a manifest gets around build problems as long as you don't make a change to a database object that changes the dependency chain. This would change the order of executing the files, so you'd need to change the manifest too.

# Building a database

## Doing a build with the SQL Comparison tool.

SQL Compare can read a directory of scripts or can pull scripts out of source control. It has its own parser and so can first create a model of the database from the scripts, and then compare that with whatever you wish: in our case a blank database on the server.

When you need to create a build script you first pull from master to include everyone's work, and then compare these scripts with the MODEL database on the target server. You can use the /empty2 switch instead if that's more convenient and you're not concerned about clashing with settings in MODEL.

```
$AllArgs = @("/scripts1:$MyDatabasePath",
      '/quiet', "/database2:model","/scrip
if (Test-Path "$MyDatabasePath\$MyDatabase
&$SQLCompare $AllArgs
if ($?) {'Script generated successfully'}
   else {"we had an error! (code $LASTEXIT
```

This will create a build script in the same directory that can be inspected, modified as necessary and executed.

When you execute this in an empty database using SSMS or SQLCMD, you will get a new database with no data.

## SMO

## Doing the build the SMO way.

With the SMO approach, the work has been already done for a clean new build. You just need to do this:

```
C:\Users\Phil.Factor>sqlcmd -S PhilFactorT
```

You'll see something like this

```
Creating the database objects in order
creating the User Brigitte_Bardot
creating the User Sophia_Loren
creating the Schema HumanResources
creating the Schema Person
creating the Schema Production
creating the Schema Purchasing
creating the Schema Sales
creating the Schema webteam
creating the XmlSchemaCollection HRResumeS
creating the XmlSchemaCollection Additiona
creating the XmlSchemaCollection ManuInstr
creating the XmlSchemaCollection ProductDe
creating the XmlSchemaCollection Individua
creating the XmlSchemaCollection StoreSurv
creating the UserDefinedDataType AccountNu
creating the UserDefinedDataType Flag
creating the UserDefinedDataType Name
creating the UserDefinedDataType NameStyle
creating the UserDefinedDataType OrderNumb
creating the UserDefinedDataType Phone
creating the Rule rule_PhNo
```

…and so on.

If you don't need to automate the process, you could always just load the manifest into SSMS, put it into SQLCMD mode, and execute it.

# Upgrading an existing version of the database

You can often change one version of a database for another whilst preserving all the existing data within its tables. This gets progressively more difficult the more different they are, and at some point it becomes impossible. How, for example could you logically migrate

AdventureWorks to NorthWind? If the two databases are close, and obviously the same database, then it can be done automatically with the right synchronization tool, but as the databases diverge in structure, it requires more and more hand-crafting of the output script of the tool you use.

There are two basic methods for upgrading an existing database whilst preserving the data in it. One way is to save all the ALTER scripts you use to change objects, the CREATE scripts you use to create objects, and the DROP scripts you use to delete them. Then you run them all in the same sequence. The other way is to use a synchronization tool and let it do the heavy lifting, altering it by hand where tables have become too heavily refactored to make sense.

## Upgrading an existing version of the database the SQL Compare way.

Having done your tests, you may decide to update the live database. Now, you are not comparing with MODEL but with your target database. However, the details are spookily similar.

```
$MyTargetInstance='MyTargetServer\MyInstan
$TargetDatabase='AdventureWorks2'#The name

$AllArgs = @("/scripts1:$MyDatabasePath",
      '/quiet', "/database2:$targetDatabas
if (Test-Path "$MyDatabasePath\$MyDatabase
&$SQLCompare $AllArgs
if ($?) {' Script generated successfully'}
   else {"we had an error! (code $LASTEXIT
```

You'll see that the end-product is a script. You need to check through this and add any extra code you might need if you've made radical changes to tables that can't be sussed out automatically. Occasionally, there will be messy problems that can't be tackled by SQL Compare, such as when you split a table or rename a column. You then test it out in Staging and, when all works fine, run it on the server. Save this script in source control, being careful to record the source and target version of the database that the change script operates on. After all, you now have a script that takes a database from one version to another. Run these in the right sequence and you can repeat the process at some point.

## Upgrading an existing version of the database the SMO way.

First you do the build of the empty database as I've already described. Then you create a DacPac of it and finally compare this DacPac with the target database. Strictly speaking, neither DacFX nor its command-line sister SQLPackage are part of SMO, but I'm not too bothered about speaking strictly. I won't repeat the description of the build process. We pick up the story at the point where we have the built version on MyInstance\MyDatabase and we wish to create a migration script to upgrade MyTargetInstance\MyTargetDatabase to the same version. We save the script in a file called MyPathAndFile.SQL.

```
#extract a DacPac
$MyInstance = 'MyInstance' #the instance w
$MyDatabase = 'MyDatabase'
$MyTargetInstance = 'MyTargetInstance' #th
$MyTargetDatabase = 'MyTargetDatabase'
$WhereTheDacPacGoes = 'MyPath'
$WhereTheScriptGoes = 'MyPath'
$DacpacFile = "$WhereTheDacPacGoes\$MyDatal
$scriptFile = "$WhereTheScriptGoes\$($MyDa
#Check that the directories exist, if not
($WhereTheDacPacGoes, $WhereTheScriptGoes)
    if (!(Test-Path -path $_))
    {
        Try { New-Item $_ -type directory
        Catch [system.exception]{
             Write-Error "error while crea
            return
        }
    }
}


& "$env:programfiles (x86)\Microsoft SQL S
'/Action:Extract', #extract it
"/SourceServerName:$MyInstance", #The SQL
"/SourceDatabaseName:$MyDatabase", #The da
"/TargetFile:$DacpacFile",
'/p:ExtractAllTableData=false') # and the


& "$env:programfiles (x86)\Microsoft SQL S
'/Action:Script', #extract it
"/Sourcefile:$DacpacFile", #The SQL Server
"/TargetServerName:$MyTargetInstance", #Th
"/TargetDatabaseName:$MyTargetDatabase", #
"/OutPutPath:$scriptFile") # The migration
```

Now, although SQLPackage looks like a free lunch, you may find that migration scripts aren't transactional:

meaning that it will actually bomb out if it hits an error, leaving your database in an indeterminate state. This is, in a production deployment, the database equivalent of running out of the operating theatre and hiding when you make a mistake by severing something vital. A roll-back would be better. However, SQLPackage is definitely better than nothing. There is a parameter, /p:IncludeTransactionalScripts, which specifies whether SQlPackage should use transactional statements wherever possible when publishing to a database, but this doesn't seem to do the same thing (*Ed:* see comment by paschott below. )

# Conclusion

Database Version control is available for anyone, whatever their budget.

I'm often told that SQL Server development is particularly difficult because the tooling doesn't exist to easily build a database, or to update an existing database from object-level source. Not so: In fact, there is enough provided for everyone to use version control with SQL Server, and to work towards team-based continuous delivery of database functionality. Naturally, it is much less work, and less hassle, with a set of third-party tools but Database Version Control is there for everyone to use and always has been, whatever your resources.