Continuous Data Migration using Visual Studio & TFS

This article will explain how to set up a continuous SQL data migration using Visual Studio & TFS on an ongoing development project which has already production data model.

While developing on top of an application which is running on a production environment, we often carry a need of data migration. However, we care very little about data migration during the development phase and tend to push this activity to the later part of the development. While it may allow an undivided focus on migration it also comes with few risks.

* You may overlook the possibility and the complexity of migration during a design change or extension.
* You may miss little detail like data fix which would be required for the new system.
* The lesser time you get to test your code, the more the risk you carry.

Continuous data migration is not a new term / process I am coining in; it has been there for a while but it never got the attention it deserves from developers / architects. The idea dictates to develop migration scripts as and when the design changes and version them appropriately. This will allow you to test the migration along with the development you do.

In this article, I will propose a solution which will take the maximum out of Visual Studio and Team Foundation Server to achieve our goal. Do keep in mind that there are some third-party software like Red Gate “Ready Roll” which are capable of doing this.

# The Issue

Visual Studio by design, can do an incremental publish of the database with the help of “SqlPackage.exe” but it also lacks in quite lot of places. The evident items like adding a column in between the columns of a table, Changing the seed data needed for the solution, Normalizing / De-normalizing tables etc. doesn’t go well with SqlPackage.

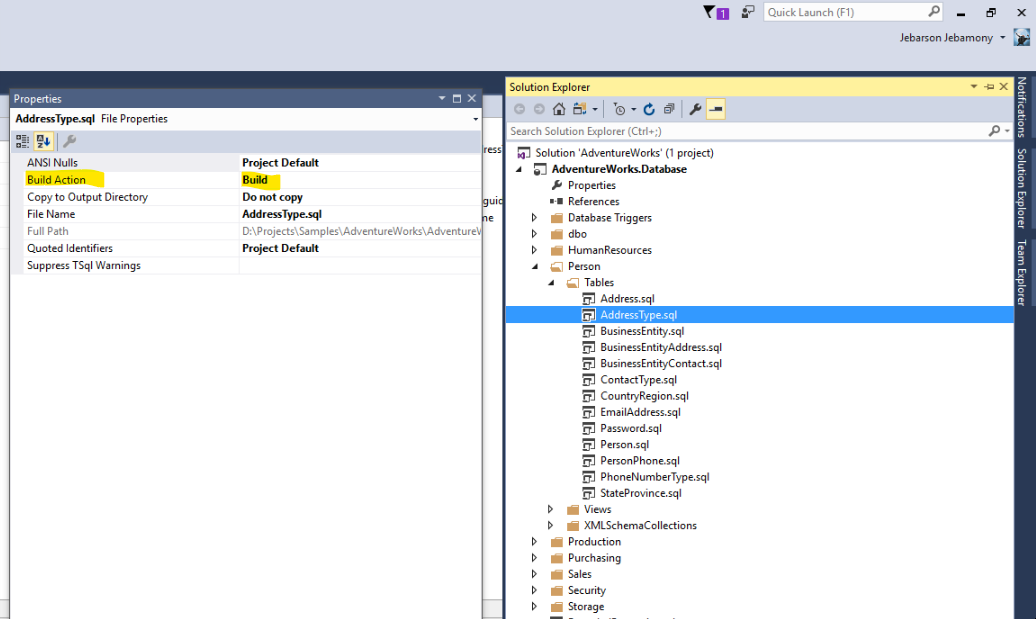
On the other side of the issue, there is a need of versioning your changes. It is very important when you need to do targeted deployment and fixes. E.g.) you may need to increment the value of a column by 10 when you migrate from v1.2 to v1.3 but not in any other flow; this can’t be achieved without versioning.

## The Proposal

I would want to approach the issue by designing a solution which will take full advantage of what Visual Studio / SqlPackage can offer and build a solution where it lacks, then make both work in a harmony.

A typical database project has two types of items; compiled scripts and non-compiled scripts. All the objects like schema, tables, views, stored procedures, etc. will go as a compiled script. The seed script or any runtime queries will get into the post deployment script.

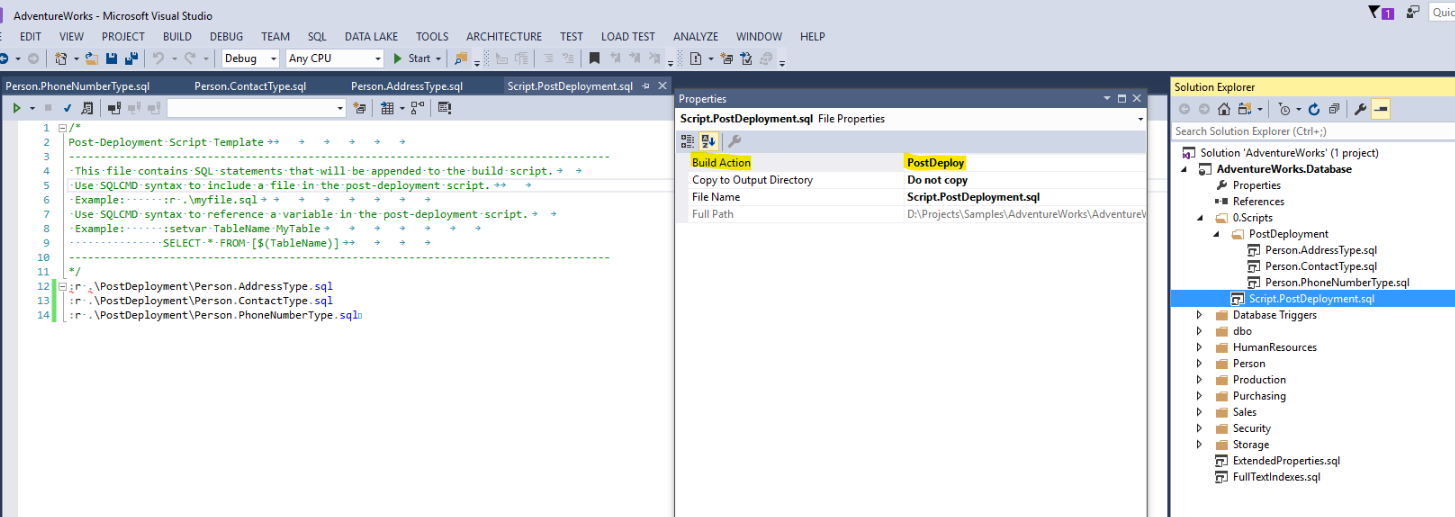
Let’s get down with an example. Below is the screenshot of “AdventureWorks.Database”; a database project (imported from the backup available [here](https://msftdbprodsamples.codeplex.com/downloads/get/880661)). As you can see all the objects are put in as compiled script.



The seed data script (data which is needed for the application to function) is put in as non-compiled script and referred in post deployment script. The usage of post deployment script can be found [here](https://msdn.microsoft.com/en-us/library/jj889461(v=vs.103).aspx).



To ensure that post deployment script is capable of handling incremental deployment, I have added a “NOT EXISTS” clause in front of all the “INSERT” statements.



For the sake of simplicity and easy maintenance we will keep all the seed script on their respective file and refer them back in the post deployment script.

Now, we have a project which will deploy the latest schema and seed data at any point of time. Also, it is capable of doing an incremental deployment on an existing database if the project is not introducing any breaking changes. However, the restrictions mentioned above in the “issue” section will come to play. Also, as of today, there is a bug which breaks the incremental deployment when a UDT is changed and unfortunately VS team has marked as won’t fix. Therefore, we will need to workaround that as well. The detail of the bug is found [here](https://developercommunity.visualstudio.com/content/problem/22062/changing-user-defined-type-without-stored-procedur.html).

Let us move on to versioning and migration. Since SQL doesn’t offer any infrastructure on versioning, we will go with our own implementation.

### Versioning

We will create a table “[internal].[DatabaseVersion]”; I would like to keep it inside the “internal” schema where I keep all the objects which are not of a business to the application.

CREATE TABLE [internal].[DatabaseVersion]

(

[DatabaseVersionId] INT IDENTITY(1,1) NOT NULL,

[Major] INT NOT NULL,

[Minor] INT NOT NULL,

[Build] INT NOT NULL,

[Revision] INT NOT NULL,

[CreatedBy] NVARCHAR (256)

CONSTRAINT [DFDatabaseVersionCreatedBy] DEFAULT ('') NOT NULL,

[CreatedOn] DATETIME

CONSTRAINT [DFDatabaseVersionCreatedOn] DEFAULT (GETUTCDATE()) NOT NULL,

[ModifiedBy] NVARCHAR (256)

CONSTRAINT [DFDatabaseVersionModifiedBy] DEFAULT ('') NOT NULL,

[ModifiedOn] DATETIME

CONSTRAINT [DFDatabaseVersionModifiedOn] DEFAULT (GETUTCDATE()) NOT NULL,

CONSTRAINT [PKDatabaseVersion] PRIMARY KEY CLUSTERED ([DatabaseVersionId] ASC)

);GO

### Migration

As a first step into migration, we will need to create a new database project “AdventureWorks.Database.Migration”.

#### Design

* The [internal].[DatabaseVersion] will have all the versions which has been released.
* The migration project will have both the script for schema / data changes isolated for every version. There is no need to include the addition of new tables / db objects / seed scripts inside the migration project as VS can generate those scripts during incremental deployment.

#### Migration Scenario

* Version 1.1
  + Inserted a new column “IsEmployee” to “[HumanResources].[Employee]” after “JobTitle” column.
  + Changed the “[Person].[AddressType]” name from “Main Office” to “Office”.
  + Changed SPs (no need to include in migration project)
  + New SPs (no need to include in migration project)
* Version 1.2
  + Changed the “IsEmployee” in [HumanResources].[Employee]” to “EmployeeType” referring to “[HumanResources].[EmployeeType]”
  + Changed SPs (no need to include in migration project)
  + New table (no need to include in migration project)
  + SP Change (no need to include in migration project)

Below is the flow diagram of how the migration will work for upgrading AdventureWorks from 1.0 / 1.1 to 1.2



#### Migration Project

In the migration project “AdventureWorks.Database.Migration”, we will add a post deployment script. This script will have the logic to do a continuous migration. The script is as below

DECLARE @currentDBVersion BIGINT = NULL;

-- Get the current version of the database.

SELECT TOP 1 @currentDBVersion = Id FROM [internal].[DatabaseVersion] ORDER BY Id DESC

-- Jump to the incremental migration scripts based on the current version.

IF @currentDBVersion = 1 GOTO Version11

ELSE IF @currentDBVersion = 2 GOTO Version12

ELSE

RETURN

-- Script to migrate to v1.1

Version11:

:r .\Scripts\Migration\v11\SchemaChangeScript.sql

:r .\Scripts\Migration\v11\DataChangeScript.sql

EXEC [internal].[CreateDatabaseVersion] @id = 2, @major = 1, @minor = 1, @build = 0128, @revision = 212

-- Script to migrate to v1.2

Version12:

:r .\Scripts\Migration\v12\SchemaChangeScript.sql

:r .\Scripts\Migration\v12\DataChangeScript.sql

EXEC [internal].[CreateDatabaseVersion] @id = 3, @major = 1, @minor = 2, @build = 0414, @revision = 096

Now if you correlate the above code with the design and the flowchart, you will be able to connect the dots on how to we are planning to do a migration.

#### Example

You can download the attached solution to understand how the whole design works. The sample solution has 3 projects.

* AdventureWorks.Database – This project is where you will make all the changes with respect to all the database objects. This project is now at v1.2 which also means that if you deploy this project, it will be the latest.
* AdventureWorks.Database.V10 – This project is also “AdventureWorks.Database” but the initial one we imported from the backup. This is included in the solution just as a reference. In your project, it will be in TFS history
* AdventureWorks.Database.Migration – This project is the migration project which will migrate any older version of “AdventureWorks” to be compatible with the current version.

To test the functioning of the sample,

1. Publish “AdventureWorks.Database.V10” to “AdventureWorks” in your database instance. (This is the initial version of AdventureWorks)
2. Publish “AdventureWorks.Database.Migration” on top of “AdventureWorks”. Be sure to uncheck “Always recreate database”. (This will execute all the scripts through 1.1 to 1.2 which has all the scripts which VS can’t generate itself)
3. Publish “AdventureWorks.Database” on top of “AdventureWorks”. Be sure to uncheck “Always recreate database”. (This will bring your database to the current version)

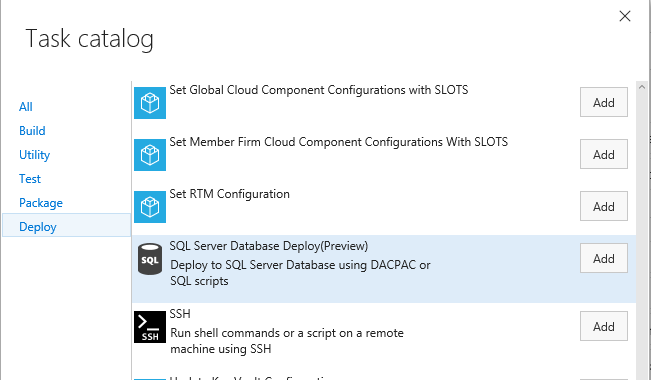
By following the above design and process you will be able to continuously migrate the database with data to the current database version. Whenever you change the schema, you will need to triage as to whether you need to include it in the migration project and then version it. This will ensure that you are continuously migrating your database and testing your migration scripts.

#### Cheat list

|  |  |
| --- | --- |
| Change | Script in Migration |
| New Table / View / Sp / any object | NA |
| Change in View / SP / Function | NA |
| Change in User defined type | Drop all the related SP of the UDT. This is a workaround for the bug mentioned above |
| Addition of a new column to table | Script a migration from the existing table to a new table with the correct column order. (Refer the attached sample) |
| Change in a data | Script out the data change. |

### Configuring TFS build to deploy continuous migration

When you deploy the database as a part of the build, you will be configuring “Azure SQL Database Deployment” or “SQL Server Database Deploy” task based on your environment



1. As a first step, you will need to publish “AdventureWorks.Database.Migration” to “AdventureWorks”.
2. Add another step where you will publish “AdventureWorks.Database” to “AdventureWorks”

Ensure that you put “/p: CreateNewDatabase=False” in the SqlPackage.exe argument

And there you have an automated “Continuous Migration” along with your “Continuous Integration” build.