Copyright © 2011

Revision	Description
6/15/2011	Original
9/5/2011	Minor updates

Pre-and-Post Deployment Scripts

Introduction

When deploying changes to databases, we can't just focus on schema changes: changes that relate to data are equally important. In cases, schema changes and supporting scripts go hand-in hand. For example, let's say we add a new non-Nullable column on a table; we might use a DEFAULT to support the non-Nullable aspect, but then need to set the value on existing records based on a more complex calculation, after which we can drop the DEFAULT. (From a modeling standpoint, let's assume we're not willing to make new fields Nullable, just for deployment concerns.)

What are some possible considerations in the above scenario? First, we need to perform this update only once on live systems – immediately after the schema change happens. Second, we don't want to lose this update script by having it fall outside of an existing build process. Third, we may or may not need to worry about including this change in a larger "rebuild" – if we needed to roll back to an older version of the database and roll forward again, we'd probably like to have it applied. Fourth, we'd like to remove as much conscious effort to include this script in a particular build as possible: having information like this floating around in e-mails, for example, has proven to be notoriously brittle.

A common approach to managing these kinds of scripts is to build up a consolidated script that assumes it can be applied as part of rebuilding a database from its starting point. This doesn't always match reality, however: imagine a staging region full of data for user test cases, where we *must* rely on incremental deployment. Some tools such as the database compare feature built into Visual Studio 2010 acknowledge this fact. Treating arbitrary support scripts like regular schema objects – by storing them in stored procedures, for example – is one way to start addressing the previously mentioned considerations. A missing element, however, is in not just reconciling these support stored procedures, but *executing* them as part of a deployment process in an automated way.

SQL-Hero Support

SQL-Hero (version 0.9.8.130+) recently added a new way to think of these pre and post deployment scripts. Instead of housing them on the "outside" of a database, it encourages developers to include them on the *inside*. What this means practically is we house these scripts inside stored procedures which themselves are housed in specially named schemas. The SQL-Hero scripting engine is aware of these schemas and if it sees objects in these schemas being moved from one database to another, it includes the execution of these objects – either at the start of the change script (pre) or end of the script (post).

Copyright © 2011

The advantages of this approach include:

- Greater visibility to the scripts involved: we can quickly find these scripts, since they all live in one of two schemas (one for pre, one for post).

- Proper linking to the version they apply to: when SQL-Hero detects that the script is in a source database and not a target database, it both copies the script (i.e. the stored procedure that houses it) *and executes it*, effectively applying the script at the correct time. Subsequent compares and scripting where the script object is no longer new means no EXECUTE is automatically added.

- Integration with existing build management features in SQL-Hero: since scripts become schema objects, schema object change tracking is already present and usable in ways like that described in the whitepaper entitled "SQL-Hero Web Portal." This ensures greater confidence that scripts will not be "forgotten" as part of the overall build process.

- Transactional scripting: if transactional scripts are used, the data changes will be properly committed or rolled back as part of the overall deployment unit.

- Integration with rollback scripting (with non-transactional scripting): since SQL-Hero already has the ability to generate a rollback script (effectively a compensating transaction to undo changes from a generated change script), script generated for objects in the pre and post schemas allows their execution to include the @IsRollback parameter which will be 1 or 0 depending on the script type. (This is optional but does at least offer the ability to safely perform compensating transaction work from scripts.)

An Example

Let's say we have a simple database with Customer, History.Customer and CustomerType tables. Suppose that the History.Customer table is a replica schema-wise of the Customer table, but holds all prior record versions. Further suppose that we have a LastUpdatedDate on Customer and History.Customer, such that the following query would give us the created date for a given customer:

```
SELECT MIN(LastUpdatedDate)
FROM History.Customer
WHERE CustomerID = @CustomerID
```

Now assume that we make a decision that having the actual created date persisted on the base Customer table would be an advantage from a physical modeling standpoint. We would like to add this new CreatedDate field as non-Nullable since we will always have a created date in reality. We might use this T-SQL to add the field with a DEFAULT:

```
ALTER TABLE Customer ADD CreatedDate datetime NOT NULL DEFAULT
(GETUTCDATE())
```

This works fine for new Customer records, but what about existing customers? We would typically write a data update script such as this and execute it after the ALTER TABLE is applied:

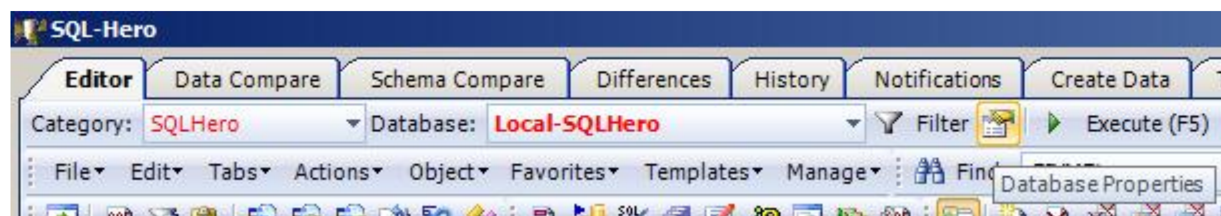
```
UPDATE c
SET CreatedDate =
      (SELECT MIN(hc.LastUpdatedDate)
      FROM History.Customer hc
      WHERE hc.CustomerID = c.CustomerID)
FROM Customer c
```

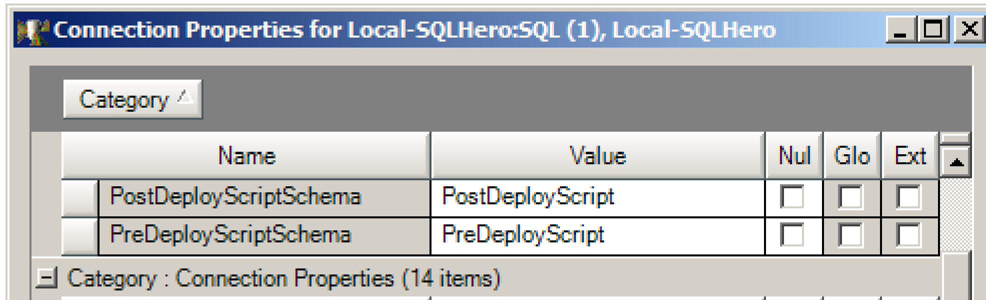
If we made this schema change in a development database, when it comes time to move such a change to a QA region, we can use a SQL-Hero schema compare to determine that the Customer table needs to be altered in QA, and we can get the script to do so. (This is true of other schema compare tools also – they're aware of just the schema difference, not any complex rules that lie to the outside, like our UPDATE statement.)

The first step to automate this is to simply wrap our UPDATE as follows:

```
CREATE PROCEDURE PostDeployScript.up_UpdateCustomerCreatedDate
AS
BEGIN
      UPDATE c
      SET CreatedDate =
            (SELECT MIN(hc.LastUpdatedDate)
            FROM History.Customer hc
            WHERE hc.CustomerID = c.CustomerID)
      FROM Customer c
END
```

As you can see, we're using a specific schema, PostDeployScript, to house scripts such as this. (As such, you may need to create this schema and grant necessary permissions beforehand.) Is the schema name hardcoded? Technically no: this is a database-level user-defined property setting within SQL-Hero, where the defaults are "PreDeployScript" and "PostDeployScript". You can find it using the Properties button for the "source" database (i.e. where copying objects from):





When we issue a schema compare now between our development and QA databases, we will see both the Customer table and PostDeployScript.up_UpdateCustomerCreatedDate objects as being in development, not in QA. When we create script for moving these objects, we'll get both the ALTER TABLE and CREATE PROCEDURE shown above, as would be expected. In addition, we get:

```
EXECUTE PostDeployScript.up_UpdateCustomerCreatedDate
GO
```

This takes care of ensuring the CreatedDate is properly set on all existing Customers, following the schema change. Subsequent compares will not show up_UpdateCustomerCreatedDate as "new for QA" so no EXECUTE will be added.

Hopefully this illustrates how a simple concept like pre-post deployment schemas can provide big benefits to database build processes, without adding a lot of extra effort or complexity.