JJ's Reference Architecture

Part 7

-

Author: Jan-Joost van Zon
Date: December 2014 – June 2017
Under Construction

Database Conventions

Developing a Database

Developing a database generally involves the following steps:

- Create tables
- Add primary keys
    - Create ID column
    - Set primary key
    - Set Identity Yes
- Add columns
    - Give them the right data types (see further down)
- Most columns not nullable
    - Limit the nullable columns as much as possible
    - But keep them nullable where it functionally makes sense.
- Make foreign keys on columns that link to other tables

- Add indexes on foreign keys columns
- Add unique indexes
    - However, sometimes ORM's will trip over unique keys at which we promptly remove the unique constraint.
    - Note that you do not need an additional index when there is already a unique constriant whose first column is the column you would like to index.
- Add indexes to search columns and alternative keys
- Add indexes for problem queries

Keep in mind to limit the use of 'exotic' data types.

For instance: if a number would fit in a tinyint, use an 'int' anyway, because this saves the system a lot of casting, and a 32-bit number is better for memory / disk alignment, which is better for performance. Chances are, a system will reserve 32 bit for a 8-bit number anyway to accomplish this memory alignment. But this is just an example.

Here are the recommended data types:

- bit
- int
- decimal
- float
- real
- datetime
- nvarchar
- uniqueidentifier
- varbinary

Only if you need a bigger range:

- bigint
- datetime2

Naming Conventions

Object Type	Example Name

Database name	ShopDB
Tables	MyTable
Columns	MyColumn
ID column	ThingID
Indexes	IX_MyTable_MyColumn
Primary keys	PK_MyTable
Foreign keys	FK_MyTable_OtherTable
	Or when there are multiple relations between the same tables:
	FK_MyTable_<*OtherTable_ColumnName_MinusID*>
	FK_MyTable_ThingA
	FK_MyTable_ThingB
Unique keys	When not many columns:
	IX_MyTable_MyColumn_Unique
	When many columns and only one constraint in the table:
	IX_MyTable_Unique
Stored procedures	SP_DoSomething / spDoSomething
Functions	FN_DoSomething / fnDoSomething
Triggers	TR_MyTable_OnInsert / trgMyTable_Insert / …

- Avoid using keywords as column names. Think of a different name instead.
- 'Index' is an SQL Server keyword! Avoid that name. Think of another one. IndexNumber or SortOrder.

Rules

Do not use the following object types, because these things are managed in .NET:

- Views
- Stored Procedures
- Functions
- Triggers
- Defaults

- Check constraints
- Computed columns
- Cascade rules
- Synonyms
- Assemblies
- Types
- Rules

For new databases, prefer int's as primary keys over guids, because guids create performance penalties throughout the software stack.
Only use additional guid columns as an alternative key for entities that need to be unique across multiple systems or databases. Do not forget to put an index on the guid column. Prefer surrogate keys rather than complicated composite keys. Prefer auto-incremented ID's, except for enum-like tables.
<TODO: Mention: Security? Guids can be safe for security. For instance, for smaller underlying entities you could not guess the ID and sneekily change someone elses data, when only the user-ownership of higher objects are checked.>

For development databases use the "DEV_" prefix, e.g. DEV_ShopDB.
For test use the prefix "TEST_" and for acceptance use the prefix "ACC_". For production use no prefix at all.
On development databases add the user dev with password dev. For test add the user test with password test. For acceptance you might use specific user names depending on security demands, otherwise add user name acc with password acc. In production databases use the administrator user's password with the administrator password for databases or create a separate user name for production with a strong password.

Upgrade Scripts

Database upgrade scripts are managed as follows.

Excel Sheet

Each database structure gets an Excel in which all the upgrade SQL scripts are registered.
The Excel sheet and SQL scripts are put in a Visual Studio project to manage them easily.
Always edit the Excel in the dev branch, because Excels cannot be merged.

The name of an SQL file has a specific format:

2014-08-28 040 ShopDB Supplier.Name not null.sql

So it has the format:

{Date} {Number} {DatabaseStructureName} {DatabaseObject}{SubDatabaseObject} {Change}.sql

	Description	Examples
Date	Use the format yyyy-mm-dd	2014-08-28
Number	Use 3 digits and count in 10's so you might insert one in betweeen	040
DatabaseStructureName		ShopDB
DatabaseObject	A table name or index name or other database object name	Supplier IX_Supplier_Name FK_Supplier_Branch
SubDatabaseObject	Optional. Usually a column name	.Name
Change	Optional. Usually left out. You can sometimes mention a specific change, but be brief.	not null

In the Excel, add a column for each database instance for that database structure. There can be different databases with the same structure for different staging areas (dev, test, acc, prod) or a database for different customers or databases running on different servers. Put 'TRUE' (or 'WAAR' in Dutch) where the upgrade script has been executed. For instance:

OrderDB Structure Changes	SCRIPTS	DEV		TEST		ACC		PROD		
		10.40.10.12	10.40.10.121	88.22.55.12	88.22.55.12	88.22.55.12	88.22.55.12	213.51.23.142	213.51.23.146	
Script		OrderDB	ShopDB	OrderDB	ShopDB	OrderDB	ShopDB	OrderDB	ShopDB	Release Date
2015-01-23 010 OrderDB Order.DeliveryDateTimeUtc.sql	TRUE	TRUE	TRUE	TRUE	TRUE	TRUE	TRUE	TRUE	TRUE	2015-02-16
2015-01-23 010 OrderDB Order.DeliveryDateTimeUtc not null.sql	TRUE	TRUE	TRUE							

Also include a column saying whether you have scripted it at all (for if you are in a hurry and have no time to script it). A release date column is also handy, to get some sense of when things went live.

Exceptional Cases

For upgrades that should only be executed on a specific database, put 'N/A' (or 'N.V.T.' in Dutch) in the appropriate spread sheet cell.
You can also add something to the SQL file name to indicate this:

2015-01-23 010 OrderDB SHOPDB ONLY Order.DeliveryDateTimeUtc.sql

Some things should be done manually and not with SQL. Those actions should also be mentioned in the Excel:

2015-01-23 020 OrderDB OrderID Identity Yes DO MANUALLY

If a script requires that you be extra careful, you can mention this as follows:

2015-01-23 010 OrderDB Order.DeliveryDateTimeUtc.sql CHECK MANUALLY
2015-01-23 010 OrderDB Order.DeliveryDateTimeUtc.sql EXECUTE SEPARATELY

But be sparse with that, because the person running the script might not actually know what it is he is supposed to check and will feel uneasy executing this script since it is obviously so dangerous, while he has no idea why.

Summary

This section covered:

- Visual Studio project
- Excel sheet
- SQL script name format
- Upgrades for specific databases and manual upgrades

Scripts

The individual upgrade SQL scripts should not contain GO statements. GO is not an SQL keyword, it is a Management Studio command telling it to execute the script up until that point. What must be separated by GO statements in Management Studio must be split up into multiple SQL files in the database upgrade scripts.

Also get rid of any automatically generated SET ANSI_NULLS ON and SET QUOTED_IDENTIFIER ON statements. Those are the default behavior anyway, and it just add unnecessary fluff to your scripts. Also: SET ANSI_NULLS OFF will generate an error in future versions of SQL Server anyway.

The upgrade scripts should be incremental: DO make assumptions about the previous state of the database structure and script a specific change. Do not write scripts like 'if not exists' then add, or 'drop and create table' scripts, because you may be throwing away data, or execute things on the wrong database. It is better to make a specific change and *not* be tolerant to differences.

DO NOT script changes from Identity Yes to Identity No or the other way around. Changes in the Identity property of a column require recreating the whole database table. If you script it now, executing it onto a database does not only add the Identity Yes property, it will also restore the whole table structure to the state it had at the time you scripted the Identity.

Summary

This section covered:

- No GO statements
- Split up into separate files
- Incremental scripts (no 'if exists' checks or drop and recreate).
- DO NOT script Identity Yes and Identity No

Deployment

To deploy multiple database structure changes you can use the Excel.
Always edit the Excel in the dev branch, because Excels cannot be merged.

OrderDB Structure Changes	SCRIPTS	DEV		TEST		ACC		PROD		
		10.40.10.12	10.40.10.121	88.22.55.12	88.22.55.12	88.22.55.12	88.22.55.12	213.51.23.142	213.51.23.146	
Script		OrderDB	ShopDB	OrderDB	ShopDB	OrderDB	ShopDB	OrderDB	ShopDB	Release Date
2015-01-23 010 OrderDB Order.DeliveryDateTimeUtc.sql	TRUE	TRUE	TRUE	TRUE	TRUE	TRUE	TRUE	TRUE	TRUE	2015-02-16
2015-01-23 010 OrderDB Order.DeliveryDateTimeUtc not null.sql	TRUE	TRUE	TRUE							

You can easily see which scripts are still to be executed onto the database.
After you have executed them, put TRUE in the appropriate spread sheet cells.

You could execute the scripts one by one, but there is a handier, safer way to do it.

With some creative copying and pasting the SQL file names, you can create a composite upgrade script like this:

```
begin try
    print 'Begin transaction.';
    begin transaction;

    declare @verbose bit = 0;
    declare @folder varchar(255) = 'C:\JJ\Install\SqlScripts';
```

```
    exec spExecuteSqlFile @folder, '2015-01-23 010 OrderDB Order.DeliveryDateTimeUtc.sql', @verbose;
    exec spExecuteSqlFile @folder, '2015-01-23 010 OrderDB Order.DeliveryDateTimeUtc not null.sql', @verbose;

    --print 'Rolling back transaction.';
    --rollback transaction;

    print 'Committing transaction.';
    commit transaction;
end try
begin catch
    print Error_Message();
    print 'Rolling back transaction.';
    rollback transaction;
end catch
```

This safely executes all changes in a single transaction and shows error information if something goes wrong.
This does require you to add the following stored procedure to the database:

```
create procedure spExecuteSqlFile(@folderPath varchar(255), @fileName varchar(255), @verbose bit = 0) as
begin
   set nocount on;

   print 'Executing ''' + @fileName + '''';

   declare @filePath varchar(255) = @folderPath + '\' + @fileName;

   declare @readFileSql varchar(1024) = 'select BulkColumn from openrowset(bulk ''' + @filePath + ''', single_blob) x;'

   declare @temp table (contents varchar(max));
   insert into @temp exec (@readFileSql);

   declare @sql varchar(max);
   set @sql = (select top 1 contents FROM @temp);

   -- Remove BOM from UTF-8.
   if (LEFT(@sql, 3) = 'ï»¿') set @sql = RIGHT(@sql, LEN(@sql) - 3);

   exec (@sql);

   if (@verbose = 1) print @sql;
end
```

Summary

This section covered:

- Composite upgrade scripts
- spExecuteSqlScript

SqlScripts table

Consider maintaining a list of executed database upgrade SQL scripts in a table, because it happens too often, that someone has put a database somewhere, without administrating the Excel file, no matter how many times you say it.

C#-Based Migrations

Some data migrations are easier to program using C# than SQL scripts.

Sometimes the the contrast between how easy it is to do in C# or SQL is so large, that the benefits of programming it in C# outweigh the downsides. It could be a factor 20 difference in development time in some cases.

A benefit of SQL scripts is that it always operates on the right intermediate version of the entity model, while C# code always operates on the latest version of the entity model. This means that earlier C#-based migrations might not compile anymore for a newer version of the entity model, and can only work with an older version of the model.

This problem with C#-based migrations can be mitigated in several ways. Here are a few ideas:

Always rerunnable tool	Replace the one-off C# migration by a tool that does something more general, that can operate on any version of the model. For instance, in a certain project, resaving most data to the database using newer business logic would set a lot of things right in the data and this procedure was rerunnable at any time, regardless of the version of the model. 'Run the resaver' would be the description in the list of data migrations to execute.

Get specific version, build, get specific version, build	You can let the C#-based migration operate on a specific version of the model by getting the older version of the software from source control, then building it. Each time you have to do a C#-based migration, you can make a separate executable, that operates on a specific version of the code. As soon as a migration does not compile anymore, you can simply outcomment or remove it.
Snapshots of entity model	Storing a snapshot of an entity model in a separate project specifically intended for that migration might be a solution. (Not tried out in practice. Might turn out to be very impractical.)
Any other ideas	are welcome.