

Database unit testing with Red Gate SQL Tools

Contents

[Introduction](#)

[Test data generation](#)

[Example: writing a database unit test](#)

[Conclusions](#)

Introduction

This article examines some of the challenges of integrating databases with the development cycle, and describes how a combination of Red Gate tools and common unit testing frameworks can be used to improve development quality.

This introduction addresses the following questions:

- What is unit testing?
- How are databases different?
- How can Red Gate tools help?

What is unit testing?

Unit testing aims to reduce errors in final application code by separating and evaluating the smallest meaningful units of testable software, for example an individual function. Each such unit is tested individually, giving you confidence in your code at a granular level, and making it easier to diagnose errors that arise in more complex integration testing.

Additionally, unit testing facilitates continuous integration, agile development, and test driven development (where the final functionality of a piece of software is outlined as a set of test conditions, and code is then written to satisfy those tests).

Unit tests are typically run locally on a developer's machine once code has been written, to ensure that it satisfies requirements before being committed to source control, and shared with the team, or passed on for QA. They may then be run again as part of a continuous integration or automated build process.

For more information on continuous integration, see [Continuous integration for databases using Red Gate SQL tools \(.pdf\)](#)

How are databases different?

Because SQL is declarative, and much database code describes relational structures, the idea of a testable “unit” is not always relevant. DDL statements modify the current state of the database, receiving instant validation. So an improperly defined object will be rejected by SQL Server.

Modern databases typically contain more than pure data. For performance or organizational reasons, a varying amount of business logic may be held in the database, for example as functions, stored procedures, or CLR objects. Many objects have and require metadata, and relatively simple changes can impact referential integrity. Individual units of procedural code must be tested, to ensure they function as required.

Testing as part of a continuous integration process is especially hindered as databases are rarely source controlled, and so cannot be deployed alongside application code as part of the build process.

More generally, database deployment is already a significant development bottleneck. It typically requires the manual creation of migration scripts. Once deployed, a database must then be populated with any data required for testing. This is also frequently a manual scripting operation, or may rely on performing a time-consuming restore from a backup. So even when a single developer is locally unit testing their own changes, the process may not be simple.

How can Red Gate tools help?

Red Gate tools work together with common unit testing frameworks, so you can automatically deploy a database, populate it with realistic data, and run unit tests.

Red Gate offers the following tools for automating database development:



Source controls database schema within SQL Server Management Studio.



Compares and synchronizes database schema.



Compares and synchronizes database data.



A bundle of APIs for our comparison tools, accessible via C# or Visual basic.



Generates realistic test data based on your database schema.

The professional editions of SQL Compare and SQL Data Compare enable you to create, compare, and synchronize folders of SQL scripts representing a database's schema and data. The process is rapid, accurate, and - crucially - can be automated using the command line interface. For more flexibility, you can also automate deployments with the SQL Comparison SDK.

If desired, a representation of the database can therefore be created and automatically maintained in a source control repository without manual intervention.

SQL Source Control builds on this technology. It is an add-in for SQL Server Management Studio that source controls your database schema in either Subversion or Microsoft's Team Foundation Server, within the IDE. This makes database source control simple to set up and use.

The development process with Red Gate tools, including unit testing, was demonstrated at the 6th [SQL Bits conference](#) in April 2010. You can [see a repeat of that demonstration here](#).

Test data generation

Testing often requires a database to be populated with known test data. For instance, a test may verify that a stored procedure returns the expected results after a schema change.

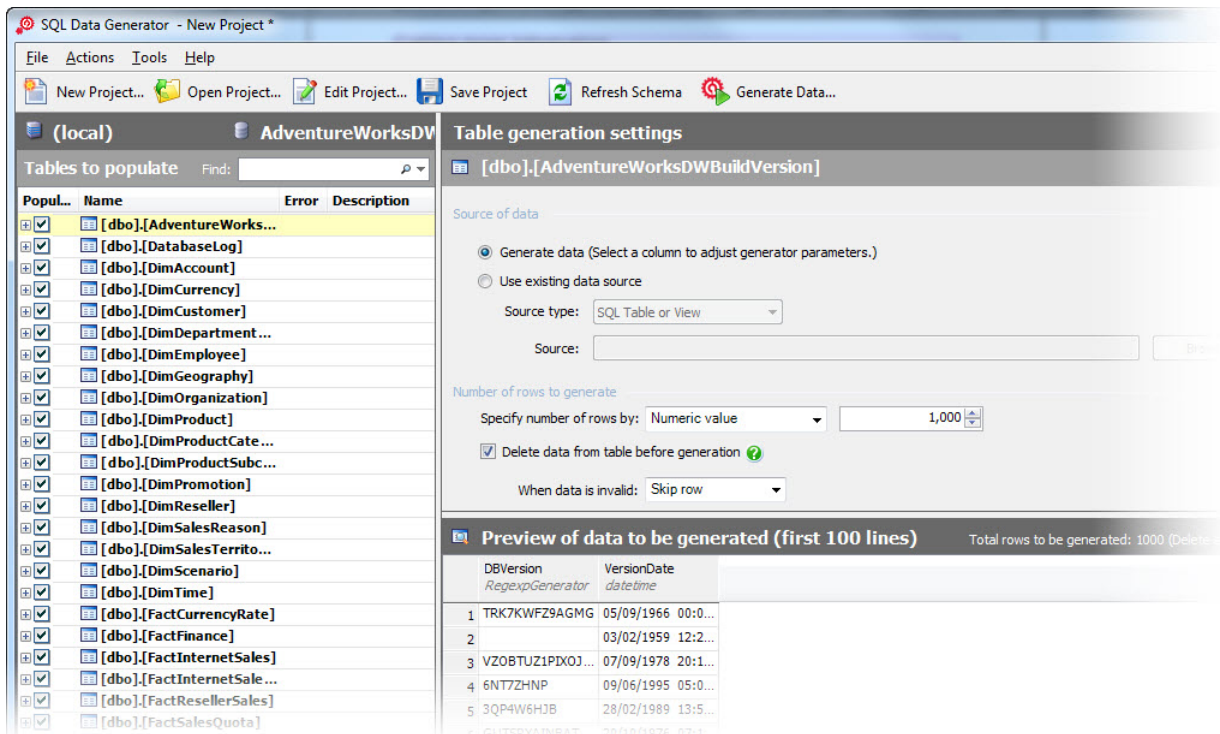
It is not always appropriate to use production data in a development environment, and manually creating test data is a monotonous, time-consuming task.

Red Gate offers two solutions to the problem of test data population: migrating existing data with SQL Data Compare, and creating realistic test data with SQL Data Generator.

This article gives some more detail on configuring SQL Data Generator for use with unit testing.

Using SQL Data Generator

SQL Data Generator has a simple graphical user interface that allows you to choose the type of data you want to generate for each table and column:



The screenshot shows the SQL Data Generator application window titled "SQL Data Generator - New Project *". The interface is divided into several sections:

- Table generation settings:** The selected table is "[dbo].[AdventureWorksDWB...]". The "Source of data" section has "Generate data (Select a column to adjust generator parameters.)" selected. The "Source type" is set to "SQL Table or View". The "Number of rows to generate" is set to 1,000. The "Delete data from table before generation" checkbox is checked. The "When data is invalid" dropdown is set to "Skip row".
- Preview of data to be generated (first 100 lines):** A table showing the first 5 rows of generated data. The total rows to be generated is 1000.

DBVersion	VersionDate	RegexpGenerator	datetime
1	TRK7KWFZ9AGMG	05/09/1966	00:0...
2	03/02/1959	12:2...	
3	VZ0BTUZ1PIX01...	07/09/1978	20:1...
4	6NT7ZHNP	09/06/1995	05:0...
5	3QP4W6HJB	28/02/1989	13:5...

SQL Data Generator automatically assigns a generator to each column based on its table name, column name, data type, and length. If the column has constraints, SQL Data Generator uses these to set the generator parameters for the column. You can change the generator used by a particular column later if required. Alternatively, you can create your own regular expression, or import data from an existing table.

Automating data generation

You can save your data generation settings as a SQL Data Generator project file (*.sqlgen*).

The project file can be used with the SQL Data Generator command line interface:

```
cd "c:\program files (x86)\red gate\sql data generator 1"  
sqldatagenerator /project:"c:\<location>\Project.sqlgen"
```

You can then create a method in your unit test that uses the project via the command line interface to generate the data you require.

For example, the following C# code defines a `DataGenerator.PopulateData()` method which can then be used in a test by supplying the name of a `.sqlgen` project file:

```
{
internal class DataGenerator
{
    /// <summary>
    /// Use SQL Data Generator to populate a table from a project file.
    /// </summary>
    /// <param name="projectFilename"></param>
    static internal void PopulateData(string projectFilename)
    {
        //Generate the data into Person.Contact table
        ProcessStartInfo startInfo = new ProcessStartInfo();
        //Point at the SQL Data Generator application
        startInfo.FileName = @"C:\Program Files\Red Gate\SQL Data Generator
1\SQLDataGenerator.exe";
        //Specify the SQL Data Generator project file to use
        startInfo.Arguments = String.Format(@" /project:""C:\DB Unit Testing
Demo\DataGenerator\{0}""", projectFilename);
        startInfo.CreateNoWindow = true;
        startInfo.UseShellExecute = false;

        //Start the process with the info we specified and wait for SQL Data
        //Generator to finish
        using (Process dataGeneratorProcess = Process.Start(startInfo))
        {
            dataGeneratorProcess.WaitForExit();
        }
    }
}
}
```

Example: writing a database unit test

This is a brief example of constructing a database unit test in C# with the [NUnit framework](#). It describes connecting to a database, querying a system view, and asserting the expected results.

For more information, see the [NUnit quick start guide](#).

The complete code example used here is available as a [downloadable zip file](#).

Connecting to the database

We encapsulate all the required connection information into one object. This makes it simple to modify the connection details. For example, here we are connecting to a local SQL Server instance using Windows authentication, but you may want to use a different server, or include a user name and password.

We create an instance of [SqlConnectionStringBuilder](#):

```
testconn = new SqlConnectionStringBuilder
{
    IntegratedSecurity = true,
    DataSource = @".\USER",
    InitialCatalog = "TestDB"
};
```

The connection can now be used to run tests. We do this with a using statement, to ensure the connection is disposed of properly after the test is complete.

For example:

```
using (SqlConnection connection = new
SqlConnection(testconn.ToString()))
{
    //Test code goes here
}
```

Querying the database

Normal T-SQL can be run within the `using` statement. We will query a system view, and ensure that a table in the database called *Names* has the expected number of columns.

We use the following SQL query:

```
SELECT count(column_name) from Information_schema.columns where
table_name ='Names'
```

There are several ways to run this query. Here, we use `SqlCommand.ExecuteScalar()` which returns the value from the first row of the first column in the results set. This is all we need for a simple test. You could also return and manipulate a whole results set. We use the following test code:

```
SqlCommand command = new SqlCommand(query, connection);
connection.Open();
object reader = command.ExecuteScalar();
```

This returns the result to the object reader.

Verifying results

To check the test results is as expected, we write an assertion using NUnit.

In this example, we expect the table to have three columns, and so the expected result is 3. Because the result is returned as an object, it must then be parsed into an integer.

We use the following code:

```
Assert.That(Int32.Parse(reader.ToString()), Is.EqualTo(3));
```

The complete code example used here is available as a [downloadable zip file](#).

Further tests

The test in this example could be used to check changes that other developers have made. However, it is very simple and would fail often in an actual development environment.

More realistic unit tests might cover:

- **Permissions and security**

For example, can users in a given role access the stored procedures, tables, views, and so on that they require. These tests could attempt to query relevant objects, and check for errors using NUnit's `Throws Constraint`.

- **Triggers and constraints**

For example, does a trigger or constraint function correctly when specific data is entered. This can be automated by running SQL Data Generator as part of the test.

- **Consistency**

For example, does a stored procedure run and return data as expected. This can be tested with a small set of known data or by running SQL Data Generator as part of the test.

Conclusions

Red Gate SQL tools substantially simplify the path to adopting database unit testing, and so to raising code quality. The tools described in this paper work together with NUnit, XUnit, MbUnit, or any other unit testing framework.

This offers flexibility, and does not require that you change the tools you currently use for unit testing.

The Red gate tools discussed in this article can be accessed via their command line interfaces, and for more configurability you can access the SQL comparison APIs that comprise our SDK.

NUnit, XUnit, and MbUnit are all available as free, open-source downloads.

All Red Gate tools are available as a free, fully-functional 14 day trial.

Getting more information

To find out more about improving database development and unit testing, see:

- [Using SQL Data Generator with your Unit Tests](#)
A Simple-Talk.com article by Ben Hall
- [Testing Times Ahead: Extending NUnit](#)
Ben's follow-up article on expanding SQL unit tests
- [SQL Source Control demonstration video](#)
A repeat of the demonstration given at SQL Bits VI, showing automated unit tests in action
- [Improving database development with Red Gate SQL tools \(.pdf\)](#)
Our overview paper on database development
- [Continuous integration for databases using Red Gate SQL tools \(.pdf\)](#)
Our guide to implementing continuous integration and source control