Av rating: ★★★★☆
Total votes: 5
Total comments: 3
send to a friend
printer friendly version

Getting Started Testing Databases with tSQLt

08 April 2013
by Robert Sheldon

There are several frameworks for assisting with the testing of SQL Server databases, but tSQLt is popular because it is written in TSQL and is simple for a database developer to set up and use. It doesn't get in the way. Rob Sheldon shows you how to get started.

Unit testing has become in integral part of application development. Within a unit-testing framework, developers can create test cases that verify blocks of code in isolation from other code. For example, a developer might create a test case to check that a class method returns the results expected based on the prescribed input. The developer can run the test case as often as necessary to verify each modification to the code. If later the code needs to be updated, the developer can rerun the test case to check that nothing was broken during the update process, assuming the test case still applies.

By incorporating unit testing within their development processes, developers can help to ensure the viability of the many individual parts that make up an application. In the same way, unit testing can also be useful to SQL Server developers, particularly for catching issues in components such as stored procedures and user-defined functions. Yet the world of SQL Server development has been slow to embrace unit testing, mainly because most of the available tools made test case maintenance cumbersome and ineffective. In many cases, the data and schemas needed to perform the tests could not be relied on to sustain the long-term use of those tests.

But tSQLt has changed all that by providing a unit-testing framework that works seamlessly with all editions of SQL Server from 2005 (service pack 2) through 2012. The tSQLt framework lets you create isolated test cases that are defined with the data you need so you can perform your tests without being affected by outside changes unrelated to the test's purpose. In addition, each test case runs in its own transaction, so you can keep the test independent of other processes and reduce the type of cleanup you often have to perform after manual testing, such as undoing data modifications. Plus, you can quickly set up tSQLt and start creating test cases as easily as you would a stored procedure, while organizing your test cases within special schemas—*test classes*—that provide the structure needed to reference a specific test case or run multiple test cases.

In this article, I introduce you to tSQLt and provide the information you need to get started. I also demonstrate how

SQL Categories

SQL Home
Learn SQL Server
SQL Training
Database Administration
T-SQL Programming
Performance
Backup and Recovery
SQL Tools
Editor's Corner
SSIS
Reporting Services

Custom RSS feeds

✓ SQL
✓ .NET
✓ SysAdmin
✓ Opinion
✓ Books
✓ Blogs
Get my feed

If you update your feed, please remember to tell your RSS reader the new URL

Click here for advanced RSS options

Phil Factor

to create several test cases that use many of the tSQLt features. If you want to try out the examples on your own system, you'll need to set up a test environment, which in this case is the **BikeShop** database. The following T-SQL script creates the **BikeShop** database, adds the **Bike** table to the database, and then inserts sample data into the table:

```
USE master;
GO

IF DB_ID('BikeShop') IS NOT NULL
DROP DATABASE BikeShop;
GO

CREATE DATABASE BikeShop;
GO

USE BikeShop;
GO

CREATE TABLE dbo.Bike
(
  BikeID INT PRIMARY KEY,
  ListPrice MONEY NOT NULL,
  ReorderPoint SMALLINT NOT NULL
);
GO

ALTER TABLE Bike ADD CONSTRAINT ck_ReorderPoint_min
    CHECK (ReorderPoint > 10);
GO

INSERT INTO Bike VALUES (101, 695.95, 35);
INSERT INTO Bike VALUES (102, 735.95, 25);
GO
```

Notice that the script also adds a check constraint to the table. The constraint ensures that all **ReorderPoint** values are greater than 10. I've included the constraint to help test one of the features in tSQLt. Later in the article, we'll also add a function and a couple stored procedures to the database to try out other tSQLt functionality as well.

Setting Up the tSQLt Environment

Not surprisingly, your first step in getting started with tSQLt is to install it on your system. The T-SQL "platform" comprises a set up objects that you add to the database you want to test. As such, tSQLt is specific to the database on which it's installed and remains isolated from other databases or the server as a whole. If you want to perform unit testing on a different database, you must set up tSQLt on that one as well.

The following steps describe how to set up tSQLt on a database (in this case, the **BikeShop** database we created above):

1. Download tSQLt from SourceForge, and unzip the file.

2. Ensure that the common language runtime (CLR) is enabled on the target database. You can use the following T-SQL to enable CLR on your database:

```
EXEC sp_configure 'clr enabled', 1;
RECONFIGURE;
```

3. Ensure that the **TRUSTWORTHY** database property is set to **ON**. You can use the following T-SQL to set the property on your database:

```
ALTER DATABASE BikeShop SET TRUSTWORTHY ON;
```

4. Run the **tSQLt.class.sql** script against the database. The script is included in the zip file that you downloaded from SourceForge.

That's all there is to setting up tSQLt on your database. If you then open **Object Explorer** in SQL Server Management Studio (SSMS), you'll find that the **BikeShop** database now contains the **tSQLt** schema and, assigned to that schema, numerous tables, views, stored procedures, and user-defined functions, both table-valued and scalar. These are the components that do all the heavy lifting when you're creating and running test cases against your database.

When you create a test case in tSQLt, you add it to a test class. That means, before you create a test case, you should first create the test class where the test case will be located. A test class is basically a schema configured with an extended property that tells tSQLt it is a test class. To create the test class, you use the **NewTestClass** stored procedure that's part of the **tSQLt** schema. When you run the stored procedure, you must pass in a test class name as an argument, as shown in the following **EXEC** statement:

```
EXEC tSQLt.NewTestClass 'TestBikeShop';
GO
```

The **EXEC** statement creates a test class named **TestBikeShop**. Once we've completed this step, we can add one or more test cases to that test class, so let's get started doing just that.

Testing the AddSalesTax Function

A test case is essentially a stored procedure that's part of a test class and that uses tSQLt elements to perform the testing. Although database unit testing is often thought of in terms of running tests on stored procedures and functions, it can also be used to test such components as **FOREIGN KEY** constraints or **WHERE** clause filters. Even so, for this article we'll focus on functions and stored procedures because they provide a good foundation for getting started with tSQLt.

The first example we'll review creates a test case for a user-defined function. But first, we must add that function to the **BikeShop** database. The following T-SQL code creates the **AddSalesTax** function, which simply adds a 9.5% sales tax to a specified amount:

```
IF OBJECT_ID('AddSalesTax', 'FN') IS NOT NULL
```

```
DROP FUNCTION AddSalesTax;
GO

CREATE FUNCTION AddSalesTax(@amt MONEY)
RETURNS MONEY
AS BEGIN
RETURN (@amt * .095) + @amt
END;
GO
```

As you can see, this is a very simple, straightforward function. We might decide to use it in a computed column or view or in some other way. As with any user-defined function, we can create a tSQLt test case that ensures the accuracy of the function before we actually implement it.

To create a test case, we use the **CREATE PROCEDURE** statement. The procedure name must start with the word "test" and be created in an existing test class; otherwise, creating the test case is much like creating any other procedure. The following T-SQL script creates a test case named **TestAddSalesTax** in the **TestBikeShop** test class:

```
IF OBJECT_ID('TestBikeShop.TestAddSalesTax', 'P') IS NOT NULL
DROP PROCEDURE TestBikeShop.TestAddSalesTax;
GO

CREATE PROCEDURE TestBikeShop.TestAddSalesTax
AS
BEGIN
  DECLARE @total MONEY
  SELECT @total = dbo.AddSalesTax(10);

  EXEC tSQLt.AssertEquals 10.95, @total;
END;
GO

EXEC tSQLt.Run 'TestBikeShop.TestAddSalesTax';
```

The part we're most concerned about in this test case is in the main body of the procedure definition (between the **BEGIN** and the **END** keywords). First, we declare the **@total** variable with the **MONEY** data type to hold the value returned by the **AddSalesTax** function.

Next, we define a **SELECT** statement that calls the function and assigns its output to the **@total** variable. Notice that we pass in a value of **10** as the function argument.

What we've done so far is set up our test scenario. The next step is to run the test itself, and for that we use the tSQLt stored procedure **AssertEquals**. The procedure compares two values. If the values are equal, the test passes. Otherwise, the test fails.

For this test, we want to compare the expected results, **10.95**, with the actual results returned by the function, as saved to the **@total** variable. We use an **EXEC** statement to run the **AssertEquals** procedure, passing in the two arguments.

That's all there is to defining our test case. To run the test case, we use the **EXEC** statement to call the tSQLt **Run** stored procedure, and pass in as an argument the names of the test class and test case, separated by a period (**TestBikeShop.TestAddSalesTax**). Because the test passes in this case the **Run** stored procedure returns the following results:

```
+----------------------+
|Test Execution Summary|
+----------------------+

|No|Test Case Name                  |Result |
+--+--------------------------------+-------+
|1 |[TestBikeShop].[TestAddSalesTax]|Success|
-----------------------------------------------------------------------------
Test Case Summary: 1 test case(s) executed, 1 succeeded, 0 failed, 0 errored.
-----------------------------------------------------------------------------
```

Now suppose that we had expected different results from what we specified in the example above. For example, the sales tax might actually be 8.5%, so we would expect the function to return 10.85 rather than 10.95. As a result, we would specify **10.85** as the first argument to the **AssertEquals** stored procedure, as shown in the following example:

```
IF OBJECT_ID('TestBikeShop.TestAddSalesTax', 'P') IS NOT NULL
DROP PROCEDURE TestBikeShop.TestAddSalesTax;
GO

CREATE PROCEDURE TestBikeShop.TestAddSalesTax
AS
BEGIN
  DECLARE @total MONEY
  SELECT @total = dbo.AddSalesTax(10);

  EXEC tSQLt.AssertEquals 10.85, @total;
END;
GO

EXEC tSQLt.Run 'TestBikeShop.TestAddSalesTax';
```

The only different between this example and the preceding one is the expected amount passed into the **AssertEquals** stored procedure. However, the original **AddSalesTax** function remains unchanged. Consequently, when we call the **Run** procedure, the test now fails, as shown in the following results:

```
[TestBikeShop].[TestAddSalesTax] failed: Expected: <10.85> but was: <10.95>

+----------------------+
|Test Execution Summary|
+----------------------+

|No|Test Case Name                    |Result |
+--+----------------------------------+-------+
|1 |[TestBikeShop].[TestAddSalesTax]|Failure|
-----------------------------------------------------------------------------
Msg 50000, Level 16, State 10, Line 1
Test Case Summary: 1 test case(s) executed, 0 succeeded, 1 failed, 0 errored.
-----------------------------------------------------------------------------
```

As you can see, the tSQLt framework provides a quick and easy way to check a function's viability, without impacting any other components or data in the database. And we can rerun the test as often as necessary. You probably noticed, however, that the function is independent of any database data. Although you can run the procedure within the context of a specific table value, you don't have to, so test data in this case, is not much of a factor, other than having to provide a value to pass into the **AddSalesTax** function when we call it. But often test data is an important consideration, which you'll see in our next example.

Testing the UpdateListPrice Stored Procedure

At times, you might want to test an operation—such as an update performed by a stored procedure—that relies on a specific set of data to return consistent results each time you perform your test. In such cases, you often need to maintain test data that can be "reset" to the original after each test. However, that is often easier said than done, especially in a shared test environment.

With tSQLt, reliable and consistent test data is not a problem because you can incorporate that data into your test case. To demonstrate how this works, we'll add a stored procedure to the **BikeShop** database that updates the list price for a specific bike ID. The following **CREATE PROCEDURE** statement adds the **UpdateListPrice** stored procedure to the database:

```
IF OBJECT_ID('UpdateListPrice', 'P') IS NOT NULL
DROP PROCEDURE UpdateListPrice;
GO

CREATE PROCEDURE UpdateListPrice
  @BikeID INT,
  @ListPrice MONEY = 0
AS
BEGIN
  UPDATE Bike
  SET ListPrice = @ListPrice
  WHERE BikeID = @BikeID;
END;
GO
```

As with the **AddSalesTax** function, we've created a very simple procedure. However, unlike the function, this procedure actually updates data in the **Bike** table. Of course, we wouldn't want to test the procedure by running it against a production database, and even running it in a test environment can turn into a complex process if it means always ensuring that the test data is exactly what we need it to be when we run our tests and schema changes do not affect our test. For example, if you're running this procedure against a test database used by multiple developers, another developer might add a constraint that affects your test, and a different developer might remove the data to test an insert operation. Even if you're developing against your own test database, you must always ensure the reliability of the data and schema after each test. Any changes can impact your tests.

Fortunately, tSQLt provides a mechanism for building the schema structure (up to a certain point) and test data right into your test case, thus preventing unrelated outside changes from affecting that test case. By using the tSQLt **`FakeTable`** stored procedure, you can create a special temporary table with the same name as the table referenced within the stored procedure you're testing. You can then populate the test table with data. Any subsequent references you then make in your test case to that table will always point to the test table and not the actual table in your database. Let's look at an example to help understand how this works.

The following **`CREATE PROCEDURE`** statement creates a test case (**`TestUpdateListPrice`**) that uses the **`FakeTable`** stored procedure to create a temporary version of the **`Bike`** table:

```
IF OBJECT_ID('TestBikeShop.TestUpdateListPrice', 'P') IS NOT NULL
DROP PROCEDURE TestBikeShop.TestUpdateListPrice;
GO

CREATE PROCEDURE TestBikeShop.TestUpdateListPrice
AS
BEGIN
  EXEC tSQLt.FakeTable 'dbo.Bike';

  INSERT INTO dbo.Bike (BikeID, ListPrice) VALUES (101, 495.95);

  EXEC dbo.UpdateListPrice 101, 595.95

  DECLARE @NewPrice MONEY
  SELECT @NewPrice = ListPrice FROM dbo.Bike WHERE BikeID = 101;

  EXEC tSQLt.AssertEquals 595.95, @NewPrice;
END;
GO

EXEC tSQLt.Run 'TestBikeShop.TestUpdateListPrice';
```

Once again, let's zoom in on the main body of the **`CREATE PROCEDURE`** statement. The first step we take is to use an **`EXEC`** statement to run the **`FakeTable`** stored procedure, passing in **`dbo.Bike`** as an argument. This creates our temporary table based on the structure of the **`Bike`** table. Any subsequent references to the **`Bike`** table within the test case automatically point to the temp table, not the real one.

Next, we use an **`INSERT`** statement to add test data into our temp table, as we would a regular table. Notice, however, that tSQLt lets us add only the data we need. In other words, our **`INSERT`** statement can specifically target certain columns. We do not have to populate every column, even if those columns in the actual table are configured as **`NOT NULL`**. In this case, we're concerned only with the **`BikeID`** and **`ListPrice`** columns because those are the columns targeted by the **`UpdateListPrice`** stored procedure. This has the added benefit of keeping the test case immune to unrelated changes to the source table, such as the addition of a column.

After we insert data into our temp table, we execute the **UpdateListPrice** stored procedure, passing in a **BikeID** value (**101**) and a new **ListPrice** value (**595.95**) as the two arguments. Although the stored procedure targets the actual **Bike** table, tSQLt is smart enough to know to use the temp table, so any data modifications made by the stored procedure are against the data that we inserted into the temp table.

Our next step is to then retrieve the updated data from the temp table, after the stored procedure has run. First, we declare the **@NewPrice** variable to hold our updated **ListPrice** value, and then we use a **SELECT** statement to assign the new value to the variable.

Finally, as you saw in the previous examples, we again use the **AssertEquals** stored procedure to compare the expected price (**595.95**) to the updated value, as it is saved to the **@NewPrice** variable. That's all we need to do to create our test case for the **UpdateListPrice** stored procedure. Again, we can run the test case by using the tSQLt **Run** stored procedure. Not surprisingly, the test case passes, as shown in the following results:

```
+----------------------+
|Test Execution Summary|
+----------------------+

|No|Test Case Name                      |Result |
+--+------------------------------------+-------+
|1 |[TestBikeShop].[TestUpdateListPrice]|Success|
-----------------------------------------------------------------------------
Test Case Summary: 1 test case(s) executed, 1 succeeded, 0 failed, 0 errored.
-----------------------------------------------------------------------------
```

If the test case had failed, we would have seen results very similar to what we saw when our function test case failed. However, regardless of whether the test passes or fails, we're able to run this test without impacting the actual table and without needing to maintain a special set of test data.

Testing the SetReorderPoint Stored Procedure

Now that you've had a sample of how you can work with tables and data in a test case, let's look at another feature built into tSQLt—the ability to compare tables. First, we need to add a second stored procedure to the **BikeShop** database. The following T-SQL script creates the **SetReorderPoint** stored procedure, which updates all **ReorderPoint** values in the **Bike** table based on a specified percentage:

```
IF OBJECT_ID('SetReorderPoint', 'P') IS NOT NULL
DROP PROCEDURE SetReorderPoint;
GO

CREATE PROCEDURE SetReorderPoint
  @percent SMALLINT
AS
BEGIN
  UPDATE Bike
  SET ReorderPoint = ReorderPoint * (@percent * .01);
END;
GO
```

The procedure takes one argument, an integer, which is converted into a percentage and then multiplied against the existing **ReorderPoint** value. Now let's look at how to create our test case. This time, we want to compare multiple rows of data against our expected results. In the following **CREATE PROCEDURE** statement, we again use the tSQLt **FakeTable** stored procedure to create a temporary table for testing:

```
IF OBJECT_ID('TestBikeShop.TestSetReorderPoint', 'P') IS NOT NULL
DROP PROCEDURE TestBikeShop.TestSetReorderPoint;
GO

CREATE PROCEDURE TestBikeShop.TestSetReorderPoint
AS
BEGIN
  EXEC tSQLt.FakeTable 'dbo.Bike';

  INSERT INTO dbo.Bike (ReorderPoint) VALUES (20);
  INSERT INTO dbo.Bike (ReorderPoint) VALUES (30);
  INSERT INTO dbo.Bike (ReorderPoint) VALUES (40);
  INSERT INTO dbo.Bike (ReorderPoint) VALUES (50);
  INSERT INTO dbo.Bike (ReorderPoint) VALUES (60);

  EXEC dbo.SetReorderPoint 200;

  SELECT ReorderPoint INTO #ActualValues FROM dbo.Bike;

  CREATE TABLE #ExpectedValues (ReorderPoint SMALLINT);

  INSERT INTO #ExpectedValues (ReorderPoint) VALUES (40);
  INSERT INTO #ExpectedValues (ReorderPoint) VALUES (60);
  INSERT INTO #ExpectedValues (ReorderPoint) VALUES (80);
  INSERT INTO #ExpectedValues (ReorderPoint) VALUES (100);
  INSERT INTO #ExpectedValues (ReorderPoint) VALUES (120);

  EXEC tSQLt.AssertEqualsTable #ActualValues, #ExpectedValues;
END;
GO

EXEC tSQLt.Run 'TestBikeShop.TestSetReorderPoint';
```

After we create our temporary table, we insert several rows of data. However, because the stored procedure we're testing targets only the **ReorderPoint** column, that's the only column we need to populate. Next, we run the **SetReorderPoint** stored procedure, passing in the value **200** as an argument. That means the **ReorderPoint** values should all be multiplied by 200%, or doubled.

Our next step is to use a **SELECT**…**INTO** statement to create the **#ActualValues** temporary table and insert the values from our **Bike** temporary table into the **#ActualValues** temporary table. We have to create the **#ActualValues** temporary table so we can do an exact comparison with our expected results, which would be difficult to do directly with the **Book** temporary table because it contains other columns, even if they're not populated.

Next, we create a second temporary table to hold our expected results. The temporary table, **`#ExpectedValues`**, includes only the **`ReorderPoint`** column. We then populate that column with the values that the **`SetReorderPoint`** stored procedure should generate.

Finally, we use the tSQLt **`AssertEqualsTable`** stored procedure to compare the data in the **`#ActualValues`** table to the data in the **`#ExpectedValues`** table. When we run the test case, it should evaluate to true and return the following results:

```
+----------------------+
|Test Execution Summary|
+----------------------+

|No|Test Case Name                      |Result |
+--+------------------------------------+-------+
|1 |[TestBikeShop].[TestSetReorderPoint]|Success|
-----------------------------------------------------------------------------
Test Case Summary: 1 test case(s) executed, 1 succeeded, 0 failed, 0 errored.
-----------------------------------------------------------------------------
```

In this case, our actual values matched the projected values, so our test succeeded. However, suppose we populate the **`#ExpectedValues`** table with different values because we expect the **`SetReorderPoint`** stored procedure to add the 200, rather than multiplying the values by 200%. The following test case definition would look as follows:

```
IF OBJECT_ID('TestBikeShop.TestSetReorderPoint', 'P') IS NOT NULL
DROP PROCEDURE TestBikeShop.TestSetReorderPoint;
GO

CREATE PROCEDURE TestBikeShop.TestSetReorderPoint
AS
BEGIN
  EXEC tSQLt.FakeTable 'dbo.Bike';

  INSERT INTO dbo.Bike (ReorderPoint) VALUES (20);
  INSERT INTO dbo.Bike (ReorderPoint) VALUES (30);
  INSERT INTO dbo.Bike (ReorderPoint) VALUES (40);
  INSERT INTO dbo.Bike (ReorderPoint) VALUES (50);
  INSERT INTO dbo.Bike (ReorderPoint) VALUES (60);

  EXEC dbo.SetReorderPoint 200;

  SELECT ReorderPoint INTO #ActualValues FROM dbo.Bike;

  CREATE TABLE #ExpectedValues (ReorderPoint SMALLINT);

  INSERT INTO #ExpectedValues (ReorderPoint) VALUES (220);
  INSERT INTO #ExpectedValues (ReorderPoint) VALUES (230);
  INSERT INTO #ExpectedValues (ReorderPoint) VALUES (240);
  INSERT INTO #ExpectedValues (ReorderPoint) VALUES (250);
  INSERT INTO #ExpectedValues (ReorderPoint) VALUES (260);

  EXEC tSQLt.AssertEqualsTable #ActualValues, #ExpectedValues;
END;
GO

EXEC tSQLt.Run 'TestBikeShop.TestSetReorderPoint';
```

The only difference in this example, compared to the preceding one, is that we updated the **ReorderPoint** values inserted into the **#ExpectedValues** temporary table. But this change is enough for our test to fail because the two tables are no longer equal. The test case now returns the following results:

```
[TestBikeShop].[TestSetReorderPoint] failed: unexpected/missing resultset rows!
|_m_|ReorderPoint|
+---+------------+
|<  |40          |
|<  |60          |
|<  |80          |
|<  |100         |
|<  |120         |
|>  |220         |
|>  |230         |
|>  |240         |
|>  |250         |
|>  |260         |

+----------------------+
|Test Execution Summary|
+----------------------+

|No|Test Case Name                      |Result |
+--+------------------------------------+-------+
|1 |[TestBikeShop].[TestSetReorderPoint]|Failure|
-----------------------------------------------------------------------------
Msg 50000, Level 16, State 10, Line 1
Test Case Summary: 1 test case(s) executed, 0 succeeded, 1 failed, 0 errored.
-----------------------------------------------------------------------------
```

Now let's look at one other tSQLt feature. As you'll recall, when we created the **BikeShop** database, we added a constraint to the table that restricted the values that can be inserted into the **ReorderPoint** column. However, the temporary tables you create with the **FakeTable** stored procedure, by default, don't include the original constraints. One reason this can be useful is because changes to constraints on the table won't impact your test cases.

However, we can override this behavior by using the **ApplyConstraint** stored procedure, which lets us apply individual constraints to our temporary table. The following example uses the **ApplyConstraint** procedure to enforce the **ck_ReorderPoint_min** check constraint defined on the **Bike** table:

```
IF OBJECT_ID('TestBikeShop.TestSetReorderPoint', 'P') IS NOT NULL
```

```
DROP PROCEDURE TestBikeShop.TestSetReorderPoint;
GO

CREATE PROCEDURE TestBikeShop.TestSetReorderPoint
AS
BEGIN
  EXEC tSQLt.FakeTable 'dbo.Bike';
  EXEC tSQLt.ApplyConstraint 'dbo.Bike', 'ck_ReorderPoint_min'

  INSERT INTO dbo.Bike (ReorderPoint) VALUES (20);
  INSERT INTO dbo.Bike (ReorderPoint) VALUES (30);
  INSERT INTO dbo.Bike (ReorderPoint) VALUES (40);
  INSERT INTO dbo.Bike (ReorderPoint) VALUES (50);
  INSERT INTO dbo.Bike (ReorderPoint) VALUES (60);

  EXEC dbo.SetReorderPoint 200;

  SELECT ReorderPoint INTO #ActualValues FROM dbo.Bike;

  CREATE TABLE #ExpectedValues (ReorderPoint SMALLINT);

  INSERT INTO #ExpectedValues (ReorderPoint) VALUES (40);
  INSERT INTO #ExpectedValues (ReorderPoint) VALUES (60);
  INSERT INTO #ExpectedValues (ReorderPoint) VALUES (80);
  INSERT INTO #ExpectedValues (ReorderPoint) VALUES (100);
  INSERT INTO #ExpectedValues (ReorderPoint) VALUES (120);

  EXEC tSQLt.AssertEqualsTable #ActualValues, #ExpectedValues;
END;
GO

EXEC tSQLt.Run 'TestBikeShop.TestSetReorderPoint';
```

Notice that when we specify the **`ApplyConstraint`** stored procedure, we pass in two arguments: the table and the constraint names. Yet even though we've included this procedure, our test will still succeed because we're multiplying our **`ReorderPoint`** values by 200%, far above the check constraint's minimum.

Now suppose we instead pass in a value of **`10`** when we call the **`SetReorderPoint`** stored procedure, as shown in the following example:

```
IF OBJECT_ID('TestBikeShop.TestSetReorderPoint', 'P') IS NOT NULL
DROP PROCEDURE TestBikeShop.TestSetReorderPoint;
GO

CREATE PROCEDURE TestBikeShop.TestSetReorderPoint
AS
BEGIN
  EXEC tSQLt.FakeTable 'dbo.Bike';
  EXEC tSQLt.ApplyConstraint 'dbo.Bike', 'ck_ReorderPoint_min'
  INSERT INTO dbo.Bike (ReorderPoint) VALUES (20);
  INSERT INTO dbo.Bike (ReorderPoint) VALUES (30);
  INSERT INTO dbo.Bike (ReorderPoint) VALUES (40);
  INSERT INTO dbo.Bike (ReorderPoint) VALUES (50);
  INSERT INTO dbo.Bike (ReorderPoint) VALUES (60);

  EXEC dbo.SetReorderPoint 10;

  SELECT ReorderPoint INTO #ActualValues FROM dbo.Bike;

  CREATE TABLE #ExpectedValues (ReorderPoint SMALLINT);

  INSERT INTO #ExpectedValues (ReorderPoint) VALUES (2);
  INSERT INTO #ExpectedValues (ReorderPoint) VALUES (3);
  INSERT INTO #ExpectedValues (ReorderPoint) VALUES (4);
  INSERT INTO #ExpectedValues (ReorderPoint) VALUES (5);
  INSERT INTO #ExpectedValues (ReorderPoint) VALUES (6);

  EXEC tSQLt.AssertEqualsTable #ActualValues, #ExpectedValues;
END;
GO

EXEC tSQLt.Run 'TestBikeShop.TestSetReorderPoint';
```

Not only have we changed the **SetReorderPoint** argument, but also the expected values we insert into the **#ExpectedValues** temporary table. Now when we run the test case, we receive neither a success or failure message; rather, we receive an error message, as shown in the following results:

```
[TestBikeShop].[TestSetReorderPoint] failed: The UPDATE statement conflicted with the
CHECK constraint "ck_ReorderPoint_min". The conflict occurred in database "BikeShop",
table "dbo.Bike", column 'ReorderPoint'.{SetReorderPoint,6}

+----------------------+
|Test Execution Summary|
+----------------------+

|No|Test Case Name                      |Result|
+--+------------------------------------+------+
|1 |[TestBikeShop].[TestSetReorderPoint]|Error |
-----------------------------------------------------------------------------
Msg 50000, Level 16, State 10, Line 1
Test Case Summary: 1 test case(s) executed, 0 succeeded, 0 failed, 1 errored.
-----------------------------------------------------------------------------
```

As you can see, our test case has violated the check constraint. At this point, we can now modify our test case, or we can change the check constraint itself.

The Wonderful World of tSQLt

Unit testing is not for everyone, nor is it appropriate for every situation. You would not, for example, want to create unit tests for each column in every table to verify that the column is in the correct format. But in those circumstances in which you need to test a process in isolation from other components, without being impacted by constraints, unrelated schema modifications, or changing data, tSQLt could prove an invaluable tool, especially since it eliminates the need to manage special data sets for this sort of testing.

And what we've covered in this article should give you the start you need to dig deeper into the various features that tSQLt supports. And there are plenty. But you now have the basics, and those will allow you to go a long way with unit testing. You can, of course, use Red Gate's SQL Test add-in to SSMS to make the process of unit testing during development as simple as possible. For integration testing, you can also integrate tSQLt with continuous integration tools such as TeamCity and CruiseControl. And because the tSQLt infrastructure and its test cases are added right to the database, you don't have to implement a special strategy to manage script files or other components. Everything is right there where you need it, when you need it.

Author profile: Robert Sheldon

After being dropped 35 feet from a helicopter and spending the next year recovering, Robert Sheldon left the Colorado Rockies and emergency rescue work to pursue safer and less painful interests—thus his entry into the world of technology. He is now a technical consultant and the author of numerous books, articles, and training material related to Microsoft Windows, various relational database management systems, and business intelligence design and implementation. He has also written news stories, feature articles, restaurant reviews, legal summaries, and the novel 'Dancing the River Lightly'. You can find more information at http://www.rhsheldon.com.

Search for other articles by Robert Sheldon

Subject: **Nice write-up but what about SRP?**
Posted by: *gregmlucas* (view profile)
Posted on: *Monday, April 15, 2013 at 5:06 AM*
Message: Robert,
This is a nice succinct write up that would be easy for someone new to tSQLt to follow. However, I take issue with your last example.
I question whether we should be testing the check constraint as part of the stored procedure test. In applying the Single Responsibility Principle, the job of the procedure SetReorderPoint is to update the ReportPoint values (not validate them). Any restriction on what those values can be is dealt with separately by the check constraint ck_ReorderPoint_min. This separation is good design but the tests for update logic and validation logic should also be separate.
Applying the constraint to the faked Bike table re-introduces an unnecessary dependency into this test. Future changes to the check constraint could cause unrelated failures in TestSetReorderPoint - just such a situation that tSQLt's Faketable was designed to help avoid. This can result in brittle tests, leading to high levels of ongoing test maintenance - one of the more common reasons for teams giving up on database unit testing.
In my experience, many SQL Developers struggle with the concept of mock objects and the need to write unit tests that test only the module under test. Good design dictates that business rules should be implemented in the fewest possible places within the database code – so that when it inevitably changes, the code only needs to be changed and tested in one place. So a business rule that says a value must be within a given range should either be written as a check constraint or built into the stored procedure – but not both.
Therefore, it stands to reason that you would want this same separation within your tests.

Subject: **Set-up and Tear-down**
Posted by: *Bill Nicolich* (not signed in)
Posted on: *Monday, April 15, 2013 at 1:20 PM*
Message: Robert - Let's say I have four tests in a suite - and I want to do set-up that will apply to all of them. Is this available in tSQLt?

For instance, let's say that I need a fake table in each of the tests. Can I do that in a "set-up" step so I can set it up once and then use it in all the tests in the suite?

Thank you,
Bill N.

Subject: **Special Setup procedure**
Posted by: *gregmlucas* (view profile)
Posted on: *Tuesday, April 16, 2013 at 2:15 AM*

Message:	Bill There is support for common setup steps in tSQLt. You can create a procedure called SetUp in a test class (schema). When you run any or all tests in that schema, tSQLt will look for and, if found, execute the procedure "SetUp" before running each test. This occurrs within the explicit transaction started by tSQLt for each individual test so any changes that are made within SetUp are rolled back in the same way as any changes made within the tests themselves.