Testing our SQL Server Backends

Eric Selje
Salty Dog Solutions, LLC
Madison WI
Voice: 708-4NaClK9
Twitter: EricSelje
Email:Eric@SaltyDogLLC.com

Nobody wants to find out the "one little change" they made to their database resulted in a data breach or massive billing error, so how can we proactively minimize that risk? Applying the concept of Unit Tests to your database allows you to quickly ascertain whether the changes you make to functions, stored procedures, and other objects in your database have a negative unforeseen impact. In this session, we'll walk through the basics of what unit testing is, how to set up a simple Unit Testing project in SSMS, how we can tell

whether something has negatively changed in our functions or procedures. We'll also enumerate any 3rd party tools available to help us with Unit Testing our databases.

Cover Your Assets

I will assert, with tongue firmly in cheek, that Database Administrators (DBAs) have it easy. As long as they don't introduce anything into the database that gets noticed they can continue to stay in their ivory towers (or poorly lit closet offices) with their heads down and avoid the day-to-day tumult that we developers are constantly dealing with.

You might think that they would do everything they can to prevent introducing any errors, but for some reason most DBAs don't take advantage of a basic skill that developers have been using for years – *unit tests*. These small bits of code can be the difference between a weekend at the cottage or a weekend back at the office trying to undo the damage caused by that "one little change" to the database. This paper will go over what a unit test is and is not, and how to write them for your database so you can cover your assets.

What is a Unit Test?

In the simplest terms, a unit test is code that tests other code. This "metacode" is structured in such a way that it returns a simple pass/fail to let the developer know whether the most recent changes caused the code to act differently than we expected. If the developer "breaks" their code and has a well-written suite of unit tests, the unit tests will "fail" and alert the developer to the problem. How exactly that alert happens depends on where in the software development lifecycle the tests are run – it could be immediately, it could be when the code is checked into source control, or it could be in the nightly build.

Note: In Visual FoxPro the premiere test harness is the open-source VFPX project named "FoxUnit". FoxUnit is *ideal* for testing Visual FoxPro code, decent at testing Visual FoxPro databases, and *could* be used for testing SQL Server databases but in my experience it doesn't make sense to use FoxUnit to test SQL Server databases when there is a better way. We'll look at that better way later in the whitepaper and set FoxUnit aside altogether.

A simple example:

Suppose you had a function that returned whether a username and password is a valid combination. Its pseudocode might look like this

```
Function isValidUser(strUserName, strPassword)
[…do some stuff…]
Return isValidUser
```

A very simple function, but *crucial* to your company's success because if this function were to break anyone and everyone could get into your application.

The first Unit Test you'd want to write for that function would be to ensure that a valid username / password combination returns *True*. That unit test might look something like this:

```
Function testIsValidUserWhenUserNamePasswordAreCorrect
*% Whew! That's a long function name! %*

ExpectedValue = True // We expect this will work
ActualValue = isValidUser("ValidUserName", "ValidPassword")
Return ExpectedValue=ActualValue
```

A couple of things to note here:

1. The test name starts with 'test'. We'll see why that's good in a little bit.
2. After the 'test' part of the test name is the function name we're testing. That's not necessary but it helps organize our tests and leads to better function naming.
3. The last part of the name is what values we're testing. In this instance we're testing whether a valid user name and password does indeed work.
4. We could have named the test "Test1" or even 'Ted" but those are so nondescriptive as to be actually counterproductive.

We would expect this test to pass if it's written correctly, but that success does not mean the function is completely written correctly. What if it returns *True* no matter who tries to log in? We definitely want to write a test to ensure this function doesn't allow someone in with invalid credentials? Thus

```
Function testNotIsValidUserWhenUserNamePasswordAreIncorrect
ExpectedValue = False // We expect this will FAIL
ActualValue = isValidUser("InvalidUserName", "InvalidPassword")
Return ExpectedValue=ActualValue
```

This test is actually testing for failure – we want that function to return *False* if invalid credentials are sent in. The test name is modified with a 'Not' to reflect that change.

Along those lines we should also write tests for when a valid username is sent in, but an invalid password, or vice versa.

But wait, there's more!

For example in Visual FoxPro we know that without the proper *SET EXACT* setting, a substring compared to a string may return True. Wouldn't it be awful if our function allowed users to log in using a substring of a valid username or password? Let's be sure to write a test for that!

```
Function testNotIsValidUserWhenUserNameIsSubstring
ExpectedValue = False // We expect this will FAIL
ActualValue = isValidUser("ValidUs", "ValidPassword") // Note the username
Return ExpectedValue=ActualValue
```

Along those lines this one method should also have test for any other "edge conditions". Eg.

- testNotIsValidUserWhenUserNameIsEmpty

- testNotIsValidUserWhenPasswordIsEmpty
- testNotIsValidUserWhenUserNameIsNull
- testNotIsValidUserWhenPasswordIsNull

All of these tests for one function are necessary and desirable, and show that there is usually a one-to-many relationship between functions and tests. For code that might get called in a less-controlled environment, you may also want to include tests that validate parameter types and values as well.

How Many Tests?

You might imagine the number of tests grows quickly when you have a lot of code and the idea of writing and managing all of these tests might be daunting. Indeed if you wrote a unit test for every possible function that would be the case, but please do not let that intimidate you into not writing any unit tests at all. Some is better than none, and you can slowly integrate unit tests into your code as you touch code.

"Test-Driven Development"

In the above example we wrote a unit test after we saw what our function does, but there is a faction of coders that believes it is better to write the tests for your code first, and then write the code itself. This idea of turning testing on its head is called "Test-Driven Development" (TDD) and it's one of the tenets of the Agile Method.

A lot has been written about this which I won't repeat here except to distill the idea down to three simple ideas.

1. In TDD, the first thing you write is a test. That test will fail because you have no code behind it.

2. The only code you write is to get a test to pass. No other code can be written.

3. You are not allowed to write any more code than is necessary to make the test pass. That is, unless there's a test already written for a code path that's failing, you don't write code for it.

Coding in this way has a number of benefits:

1. It forces you to think about your code before you write it. You need to plan exactly what you want it to do, something that too many developers don't do, and you actually implement your own code in the tests.

2. It constrains your code to specific bits of functionality, preventing "code bloat" and feature creep.

3. It catches software defects very early in the software development lifecycle. It is exponentially cheaper to fix a bug on the developer's desktop than it is to fix a bug that's been shipped to the client, or worse, has caused a database leak!

TDD has its downsides. Writing tests does take time, which may be perceived as "wasted" by non-supportive management or clients, and the tests themselves need to be maintained along with the rest of the codebase. That means understood and shared with the other developers as well as backed up properly. Another downside is you may never be able to show that the extra effort actually saved your bacon because it's impossible to prove a negative.

That is all I'm going to write about Test-Driven Development. There's a lot more that can be said about it, but it's not the focus of this particular whitepaper and for many it's realistically rare to even have the opportunity to work in such "greenfield" environments, but I hope I've introduced you to where Unit Testing fits into the bigger picture.

Unit Testing vs. ????

It's important to note that Unit Testing isn't a panacea for deploying error-free, usable code. The full software development lifecycle includes many other types of checks and balances to reduce the gap between what you deliver and what the customer expects. These tests include but aren't limited to:

- Acceptance Testing
- Load Testing
- Compatibility Testing
- Accessibility Testing
- Requirements Compliance

Unit Testing a *Database?*

What does all of this unit testing have to do with your database? Simply, there *is* code in your database. Lots of it. You'll find it in stored procedures and functions, naturally, but you'll also find it behind in your view definitions and in your field and row constraints and perhaps in a few other places depending on which features of SQL Server you use.

So what does it mean to unit test a database? It means pretty much the same thing as unit testing your classes in regular code – writing code that ensures the code tucked in your database passes.

Where would keep such code?

Our First Unit Test

In my first passes at writing unit tests for SQL Server (before I knew better), I wrote the unit tests as TSQL Scripts and kept them in a solution (.sln) file so I could easily retrieve them from the Solution Explorer (**Figure 1**).

Our first unit test (testStatus1, **in Figure 2**) tests that we get back what we expect when we send in a valid value. This test is testing an easy and simply scalar function from the AdventureWorks database (dbo.*ufnGetSalesOrderStatusText,* **in Figure 3***).*

And sure enough,when we run it we get

```
Success
```

In our Output Window. Great!

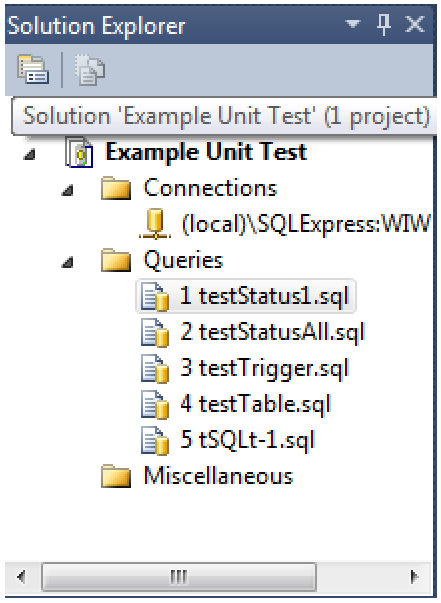


Figure 1: Tests stored in SSMS's Solution Explorer

```
-- testStaus1.sql
-- Make sure the Sales Order status returns what I think it will
DECLARE @actual varchar(30)
DECLARE @expected varchar(30) = 'In process'
DECLARE @result varchar(10) = 'Failed' – Default to ‘Failed’
-- Make the actual call to the function
SELECT @actual = AdventureWorksLT2012.dbo.ufnGetSalesOrderStatusText(1)
-- Output result
IF @actual = @expected
BEGIN
SET @result = 'Success'
END

PRINT @result
-- PRINT @actual
GO
```

Figure 2: Our first unit test in TSQL

```
ALTER FUNCTION [dbo].[ufnGetSalesOrderStatusText](@Status [tinyint])
RETURNS [nvarchar](15)
AS
-- Returns the sales order status text representation for the status value.
BEGIN
    DECLARE @ret [nvarchar](15);

    SET @ret =
        CASE @Status
            WHEN 1 THEN 'In process'
            WHEN 2 THEN 'Approved'
            WHEN 3 THEN 'Backordered'
            WHEN 4 THEN 'Rejected'
            WHEN 5 THEN 'Shipped'
            WHEN 6 THEN 'Cancelled'

            ELSE '** Invalid **'
        END;

    RETURN @ret
END;
```

Figure 3: AdventureWorksLLT2012.dbo.ufnGetSalesOrderStatusText, our function to test

Note: If you think it's ironic that this sample function that comes from Microsoft for SQL Server, one of the top databases in the world, isn't data-driven – well, this author agrees. It does make a great code sample for unit testing for which may have been their intent and for which I am grateful to now be able to use.

Our Second Unit Test, or, Is This Going to Scale?

As we said earlier, it's not enough to be satisfied that our function works just because it passes one unit test and all the parameters are what we expected. We should also write tests for edge conditions, null values, invalid parameter types, and whatever else we can contrive. But where would we put such tests in our current arrangement?

We could simple create a new script similar to the one we had in **Figure 2**, and that would work great. Setting up the scripts this way would make it easy to run individual tests, but that would mean if we wanted to run all of our unit tests we'd have to open each script and hit Execute to run it. That seems like a real pain in the butt.

We could work around that pain by creating another script that calls each of the unit tests scripts, but we'd have to be certain that we keep that "metascript" maintained because there's no facility to "Run All Scripts" in SSMS.

You may start getting the sense that perhaps writing our unit tests as TSQL scripts may not be an ideal, er, solution and you'd be right. But before we explore something better let's take a look at two more unit tests to discover more shortcomings of this solution.

Testing a Trigger

There is code everywhere in SQL Server databases, but one place you may not think of at first is "Triggers," code that automatically fires when a change is made to a table or field.

In the AdventureWorks table, there is a trigger on the SalesOrderDetail table that will automatically recalculate and update the SalesOrderHeader.SubTotal field. This is a pretty common type of trigger to have, and one that I'd be very nervous about messing up if I were in there modifying the code.

To ameliorate that nervousness, I'd write a unit test for this trigger. My first attempt at such a unit test might take an existing purchase order detail line and, say, double the quantity of an item ordered, and then check the SalesOrderHeader.SubTotal to see if matches what I think it will.

However, if you think about this, you'll never be able to run this test twice because the SubTotal will constantly be updating to a new expected value. D'oh!

Another solution might be to just insert a new SalesOrderHeader and add some known detail to that SalesOrder and check the SubTotal. This isn't bad but soon your database will get cluttered up with fake sales orders so it's not ideal.

A simple solution to both of these issues is to *wrap your tests in a transaction*. That way you can do what you please in your tests, and then rollback when you're done to the state you were in when your test began. Add records, update records, delete records – no problem. The transaction makes everything all right.

If you're writing your unit tests as TSQL scripts, then it is up to you to remember to wrap all of your tests in a transaction. This can easily escape the mind of the busy coder, however, and it would be much better if mortals didn't have to remember this crucial step. More on that later…

Testing Changes to a View Definition

Another non-intuitive location of code in database is in the definition of views, and views are something that actually get changed more often than other code. In fact the entire reason I got started interested in the topic of unit testing databases was that I was facing the daunting task of refactoring a database schema and I wanted to be certain that the views returned the exact same result even though the source data was now going to be coming from additional tables.

Like all unit tests, the first thing you need is an *Expected Value*. Views are different than functions though in that the expected value isn't a single data type, but rather a special *Table* data type. This is somewhat analogous to an in-memory cursor in Visual FoxPro.

To get our expected table, we need to pull some known records from a section of the source databases that won't be changing. Records from long ago are best as long as they give a decent enough sample size. But how do you get the records into a table value if the view is changing? That seems like a moving target.

What I used to do before I upgraded to SQL Server 2012 was use the free SSMS Toolpack to grab the output of a SQL Statement, right click on it, and select "Output to INSERT statements" [*Note: there is a Thor tool that does exactly this for VFP.]* But SSMS Toolpack is no longer free so I had to improvise another method which you can see commented out in **Figure 4** involving using the SELECT statement to a long string of INSERT statements.

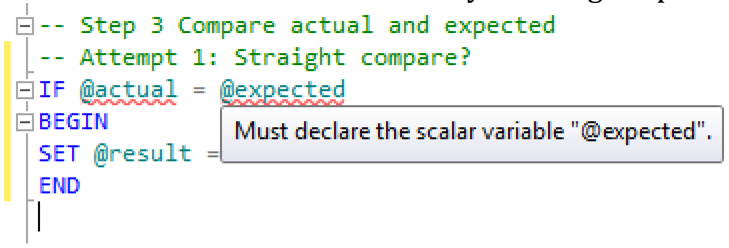
Once you have your expected rows in a table variable that never changes, you can get your

```
-- Test where the results are a table
SET NOCOUNT ON

DECLARE @result varchar(10)='Fail'

-- Set up temporary table variables rather than scalar values
DECLARE @expected TABLE(CustomerId int, TaxAmt money, OrderYear int)
DECLARE @actual TABLE(CustomerId int, TaxAmt money, OrderYear int)

-- Step 1 Set the expected values as they were at one time
-- (Values achieved by issuing this statement and sending results to text)
--select 'insert into @expected  (CustomerId, TaxAmt, OrderYear) values ('
--      + CAST(CustomerId AS varchar) +
--      ', ' + CAST(TaxAmt AS varchar) +
--      ', ' + CAST(OrderYear AS varchar) + ')'
--   FROM [AdventureWorksLT2012].[SalesLT].[vGetCustomerTax]

-- SSMSToolPack has a feature to export results to this, but it's no longer free :(
insert into @expected  (CustomerId, TaxAmt, OrderYear) values (29847, 70.43, 2004)
insert into @expected  (CustomerId, TaxAmt, OrderYear) values (30072, 6.30, 2004)
insert into @expected  (CustomerId, TaxAmt, OrderYear) values (30113, 3073.50, 2004)
insert into @expected  (CustomerId, TaxAmt, OrderYear) values (29736, 8684.95, 2004)
insert into @expected  (CustomerId, TaxAmt, OrderYear) values (29660, 4610.77, 2004)
insert into @expected  (CustomerId, TaxAmt, OrderYear) values (30025, 3.27, 2004)
insert into @expected  (CustomerId, TaxAmt, OrderYear) values (29982, 238.46, 2004)
insert into @expected  (CustomerId, TaxAmt, OrderYear) values (29781, 8.52, 2004)
insert into @expected  (CustomerId, TaxAmt, OrderYear) values (29531, 530.74, 2004)
insert into @expected  (CustomerId, TaxAmt, OrderYear) values (30050, 7862.30, 2004)
insert into @expected  (CustomerId, TaxAmt, OrderYear) values (29546, 7105.03, 2004)
insert into @expected  (CustomerId, TaxAmt, OrderYear) values (29741, 3.12, 2004)

-- Step 2 Get the actual values currently
INSERT INTO @actual
       SELECT * FROM [AdventureWorksLT2012].[SalesLT].[vGetCustomerTax]

-- Step 3 Compare actual and expected

[Here there be dragons]

PRINT @result
GO
```

Figure 4: Testing a View

actual rows by just selecting them from the view you're modifying you can just compare the two variables.

But uh oh, here's where things get sticky in TSQL. You cannot just do a straight compare of two table variables. SSMS will tell you straight up:

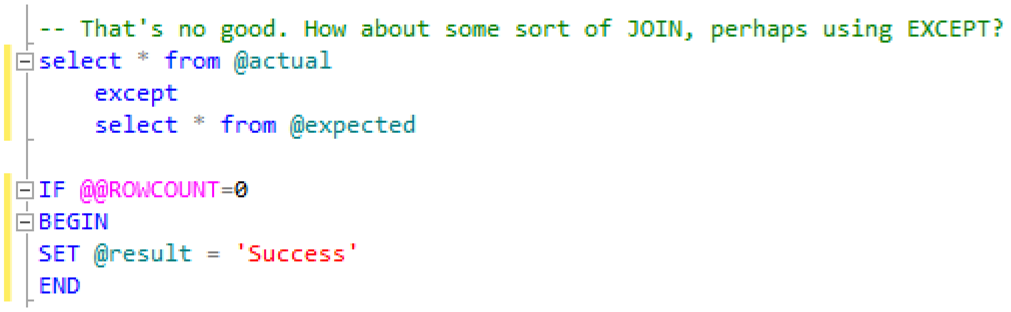


```
-- Step 3 Compare actual and expected
-- Attempt 1: Straight compare?
IF @actual = @expected
BEGIN
SET @result =
END
```

Must declare the scalar variable "@expected".

Ok, we'll have to be a bit cleverer than that. How about if we just see how many rows are in each resultset? That's no good because the row count alone doesn't indicate whether the data is the same.

How about doing a JOIN of the two tables using the EXCEPT clause, which should give us all the rows in one table that aren't in the other table. If we get zero rows, we'll know they're the same, right?



```
-- That's no good. How about some sort of JOIN, perhaps using EXCEPT?
select * from @actual
    except
    select * from @expected

IF @@ROWCOUNT=0
BEGIN
SET @result = 'Success'
END
```

Well not necessarily. We'll also get zero rows if there are *more* rows in the Expected table than the Actual table. We could fix that by doing a UNION of the opposite join I suppose, but this is getting a little crazy. And most importantly you'll find that if you actually run this it doesn't work anyway because some precision of the decimals was lost in our initial creation of the Expected table. Ugh.

Let's set this aside for a second and rethink everything...

Pro and Cons of Using TSQL scripts for Unit Tests

Pros	***Cons***
✓ Tests are written in SQL Server Management Studio, a free tool we are familiar with. ✓ Tests are written using TSQL rather than a separate language. ✓ Test are file-based, so they're easy to check into source control using our existing version control system.	✗ We must remember to wrap our tests in transactions lest we mess up our databases. ✗ We have to manually push Execute on each test to get it to run. Alternatively we have to maintain a separate "metascript" to run our tests. ✗ The output of the tests is in the Output window in SSMS, which isn't useful for integrating into a Continuous Integration Solution. ✗ Tests are file-based, so we have to check them into source control in order to share them with other members of our team. ✗ Tests are solution-based, so they're separate from the database and can get separated from them. This slows down new devs coming into the team. ✗ There aren't natively good ways to create mock data that's accurate. ✗ There also aren't good ways to natively compare table values.

Your list may differ, but you can see the pros of using native TSQL are completely outweighed by the cons. Wouldn't it be great if there was a solution integrated into SSMS, so you wouldn't have to use a separate product, even Visual Studio which many DBAs don't have access to or know how to use? And wouldn't it be great if that product addressed a lot of the cons on this list? (You know what's coming next, right?)

Introducing tSQLt

tSQLt is a unit testing framework that works within SSMS Management Studio. It's free and open source, and has a lot of nice features to overcome the shortcomings that we've noticed when trying to write our own unit tests. A short list of those features:

- tSQLt stores the tests as stored procedures in the database itself, rather than as a separate SSMS Solution file on disk. This means all developers of a database have immediate access to all tests (assuming the development database is shared among them). If you *want* your unit tests under source control you could do that using a 3rd party tool like RedGate's SQL Source Control.

- tSQLt groups the tests by taking advantage of the concept of a *schema* in SQL Server. Many people always stick to the default 'dbo' schema, but by properly using schemas you can set up more discrete levels of hierarchy and security in your database and tSQLt does that. With these groups you can run individual tests, all the tests in one schema, or every test in your database with one command.

- tSQLt automatically wraps every tests in a transaction, so you don't have to remember to do that. This keeps your database safe and consistent.

- I will buy the first person at Southwest Fox 2013 who tells me they read this sentence a drink. Conference organizers excepted as I suspect they read every whitepaper. They're really fantastic people.

- tSQLt follows many of the conventions of other unit testing frameworks, including having Setup() and Teardown() functionality as well as a variety of Assert statements, such as the expected AssertEquals and the nice to have AssertLike, but most impressively an AssertEqualsTable which completely takes care of the struggles we had in the last section figuring out how to compare table values.

Rather than rehash the entire tSQLt manual, I'll refer you to http://tSQLt.org where you can reference the very fine and up-to-date documentation yourself. A couple of points I'll emphasize:

1. Don't ever do unit tests on production databases.

2. Don't ever do unit tests on production databases.

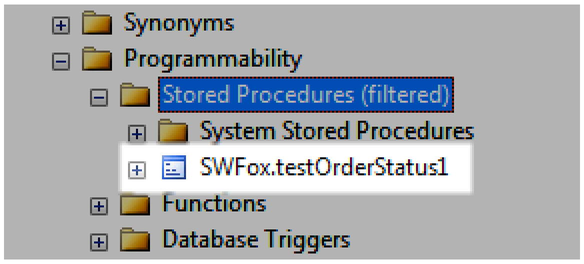
3. Don't ever do unit tests on production databases.

The tSQLt equivalent code for our very first unit test is shown here in **Figure 5**.

```
use AdventureWorksLT2012;

-- Create the "Test Class" (This actually creates a schema in our database)
EXEC tSQLt.NewTestClass ‘SWFox';

-- Create a test in that test class
CREATE PROCEDURE SWFox.testOrderStatus1 AS
BEGIN
DECLARE @actual varchar(30)
DECLARE @expected varchar(30) = 'In Process'
SELECT @actual = AdventureWorksLT2012.dbo.ufnGetSalesOrderStatusText(1)
exec tsqlt.AssertEquals
       @Expected=@expected,
       @Actual=@actual,
       @Message='Sales Order Status 1 did not return In Process'
END
GO

-- Run that test
exec tsqlt.Run ‘SWFox.testOrderStatus1’
GO

-- Run all tests in the class
exec tsqlt.RunTestClass ‘SWFox’;
```

Figure 5: Our first unit test as a tSQLt test

After we create the test, you can see in the database we know have a new stored procedure for that test, in the SWFox schema.



To run that test, we don’t just execute the stored procedure. We need to call the tSQLt.Run stored procedure with the test name as a parameter:

```
exec tsqlt.Run ‘SWFox.testOrderStatus1’
```

and we get better looking results than our simple ‘Success/Fail’ text:

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

|No|Test Case Name            |Result |
+--+--------------------------+-------+
|1 |[SWFox].[testOrderStatus1]|Success|
-----------------------------------------------------------------------------
Test Case Summary: 1 test case(s) executed, 1 succeeded, 0 failed, 0 errored.
-----------------------------------------------------------------------------
```

We can run all the tests in our test class, which perhaps are all aimed at one particular function we're testing, using a command like this:

```
exec tsqlt.RunTestClass 'SWFox';
```

...and our results would enumerate each test.

Finally we could run every test in our database with this command:

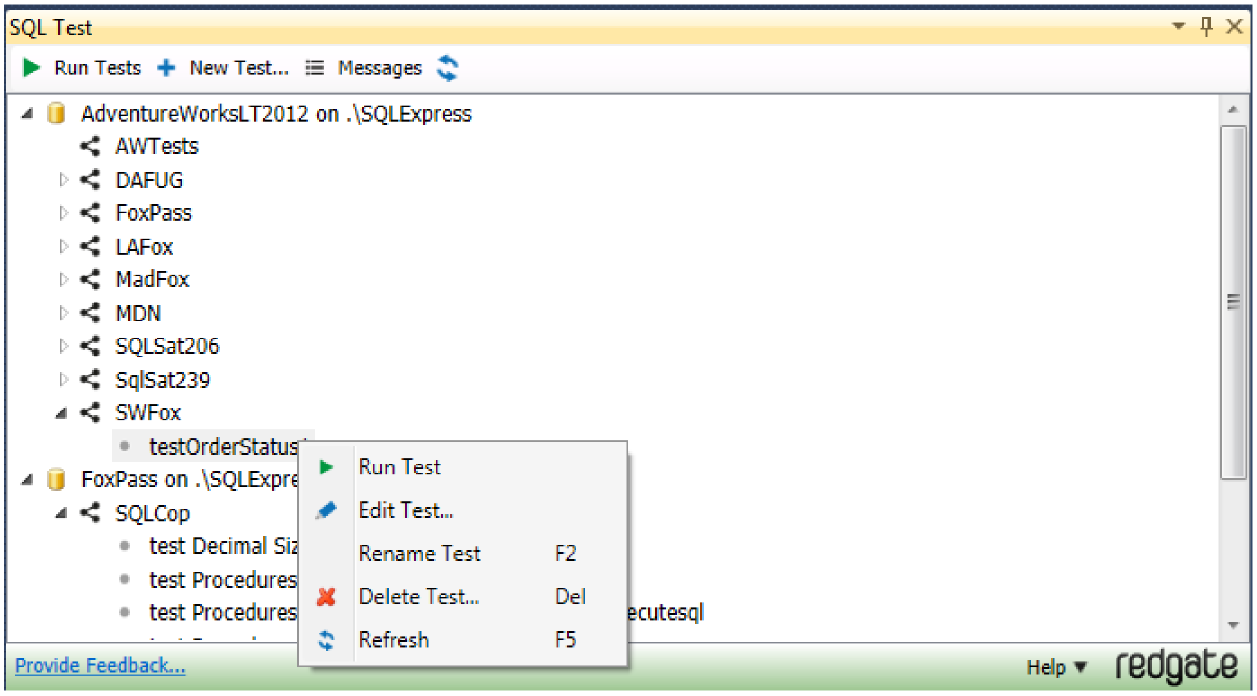
```
exec tsqlt.RunAll;
```

and tSQLt would execute every stored procedure. But wait, how does tSQLt know which stored procedures are unit tests and which are not? Remember way back when I said it is a good idea to preface the name of each test with 'test'? This now comes into play because tSQLt actually only executes every stored procedure that starts with those four letters.

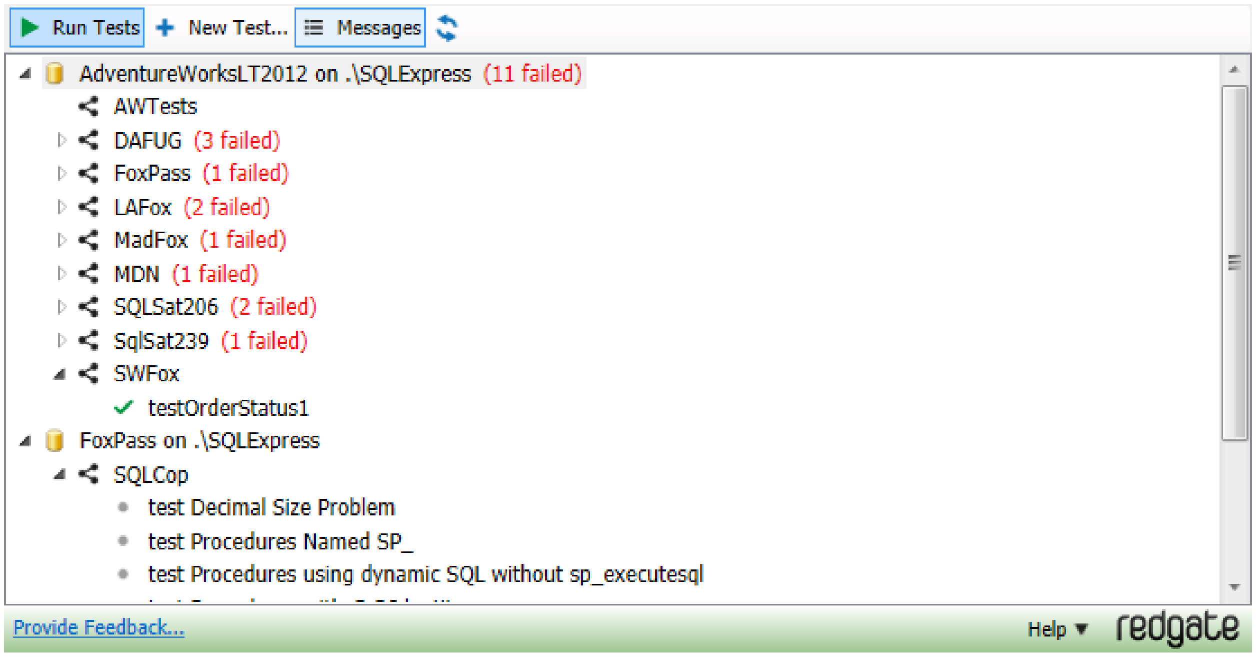
Red Gate Takes tSQLt Up a Notch

If the idea of typing at all seems daunting to you and you'd prefer a push button approach to unit testing, consider spending $295 on SQL Test from Red Gate (or more if you want to buy the whole developer's toolbelt, which includes SQL Test).

SQL Test wraps a very nice Graphical User Interface around tSQLt, but it's still tSQLt under the cover. You can see by the screenshot how easy it is to see your test classes, and running a test is as easy as right+clicking.



For those of us who prefer the Pavlovian response of the green checkmarks that come with passing tests, SQL Test also gives color coded results that make it easy to see what passed and what failed.



```
AdventureWorksLT2012 on .\SQLExpress (11 failed)
    AWTests
    DAFUG (3 failed)
    FoxPass (1 failed)
    LAFox (2 failed)
    MadFox (1 failed)
    MDN (1 failed)
    SQLSat206 (2 failed)
    SqlSat239 (1 failed)
    SWFox
        ✓ testOrderStatus1
FoxPass on .\SQLExpress
    SQLCop
        ● test Decimal Size Problem
        ● test Procedures Named SP_
        ● test Procedures using dynamic SQL without sp_executesql
```

Anybody who tells me at Southwest Fox they've read this far in my whitepaper will be entered into a drawing for a free copy of SQL Test, courtesy of Red Gate. Even if you don't win, you might decide $295 is money well spent to make your testing easier and more colorful.

Conclusion

Unit testing is for databases. There's quite a bit of code in databases and if you want to modify it without worrying about taking down your whole company then you should write unit tests for that code the same as you would for your other code. It will take some time to get used to writing tests for everything, time which you may not believe is productive initially. You may also be overwhelmed by the sheer number of tests you believe you'll have to write. I encourage you to practice this discipline and reap the peace of mind that comes with knowing you can make changes to the database code and know that you didn't break anything.

Much more could be said about unit testing's place in the cycle of Continuous Integration, and I hope you'll read my other Southwest Fox 2013 whitepaper for more on that!

Thanks for reading.

Resources

- Unit Testing Database with tSQLtTen Things I Wish I'd Known

- 48 SQL Cop Tests (Zip File)
- Using SQL Test in Continuous Integration, by Dave Green
- How to write good unit tests
- Are Unit Tests overused?
- Test Driven Development