TDD Your Database Demos

Getting Started

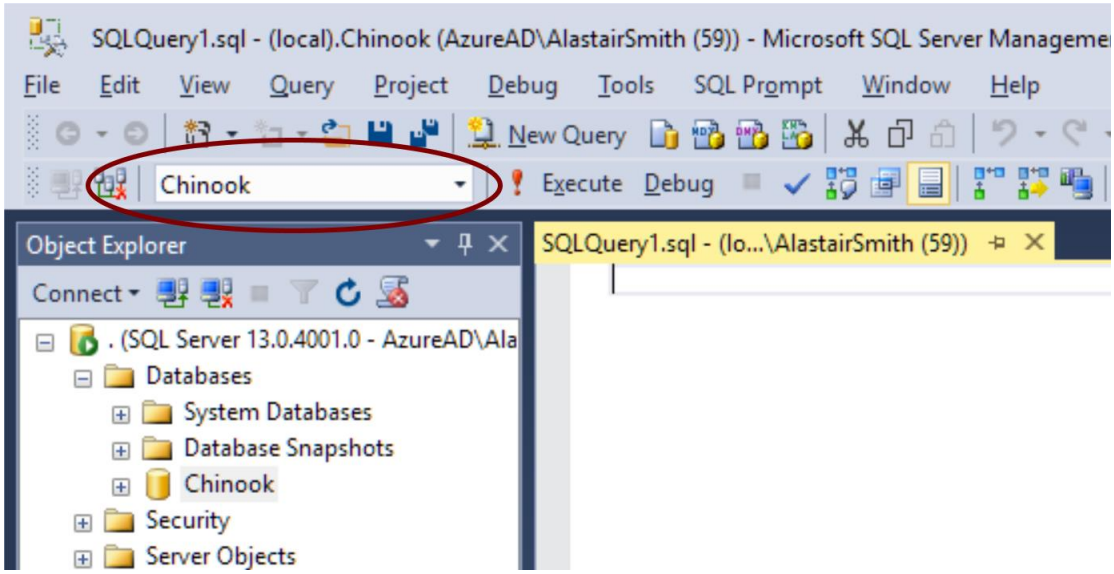
The Chinook database is an open-source sample database representing an online music store. It has tables for albums, artists, customers, employees… It is available for a bunch of RDBMS, such as SQL Server, Postgres, MySQL and more. Download it from https://chinookdatabase.codeplex.com/.

These demos were prepared against Microsoft SQL Server 2014, and SQL Server Management Studio (SSMS). You can download a free version of both of these from http://downloadsqlserverexpress.com/.

Download the tSQLt unit testing framework from http://tSQLt.org/. The completed code samples are available on GitHub at https://github.com/alastairs/tsqlt-demos

In SSMS:

1. Open the Chinook database creation script `Chinook_SqlServer_AutoIncrementPKs.sql`, and press F5 to run.
2. Open `tSQLt\SetClrEnabled.sql`, and press F5 to run.
    - tSQLt requires SQL CLR to be enabled and the database to be TRUSTWORTHY, which this script will handle for you
3. Open `tSQLt\tSQLt.class.sql`, ensure the Chinook database is selected and press F5 to run.
    - tSQLt must be installed to each and every database you wish to test



tSQLt is now installed to the database. Let's check it works!

Your first test

First we need to create a test class. This is like a Test Suite in NUnit/Junit, etc.

```
EXEC tsqlt.NewTestClass @ClassName = N'DoesItWork'
GO
```

This creates a schema within the database in which our test will be confined. This aids test isolation: a test from one test class will not impact tests from another test class.

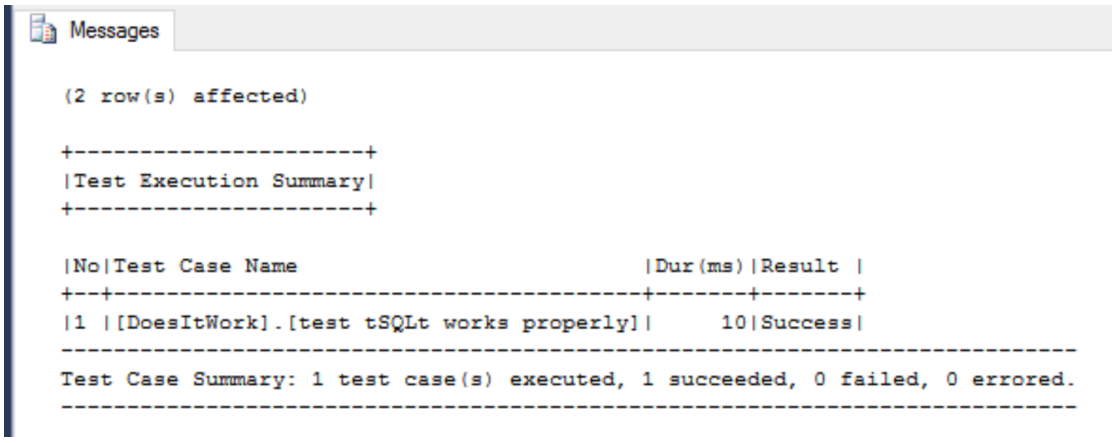
tSQLt tests are just stored procedures, but like the original Junit the names **must** start with the word "test" to be discovered.

```
CREATE PROCEDURE DoesItWork.[test tSQLt works properly]
AS
BEGIN
	CREATE TABLE [Album.Expected] (
		[AlbumId] [INT] IDENTITY(1,1) NOT NULL,
		[Title] [NVARCHAR](160) NOT NULL,
		[ArtistId] [INT] NOT NULL
	);

	EXEC tSQLt.AssertResultSetsHaveSameMetaData
	    @expectedCommand = N'SELECT * FROM dbo.[Album.Expected]',
	    @actualCommand = N'SELECT * FROM dbo.Album'
END;
GO
```

The [ ] are T-SQL quotes – we can use real strings in our test names!

This simple test checks tSQLt is hooked up properly, by checking the shape of the `Album` table is as expected: first we create a table called `Album.Expected` with three columns called `AlbumId`, `Title`, and `ArtistId`. `AssertResultSetsHaveSameMetaData` compares the results of two commands to see if the structures are the same. In this case, we `SELECT` everything from both the real `Album` table and the `Album.Expected` table to compare their structures.

To run the tests, we run `EXEC tSQLt.RunAll`



You can press Ctrl+R to hide the Results window again, should you wish.

Stored Procedures (SPROCs)

Just as we wouldn't compare object structure in application tests, we shouldn't compare data structure in database tests. These are very brittle tests to maintain: any change in data structure will result in the test failing.

Let's test a real business requirement: we want to add a page to our application featuring all albums released by a given artist. We can compile this into our database as a SPROC to achieve the associated query optimisation and performance benefits.

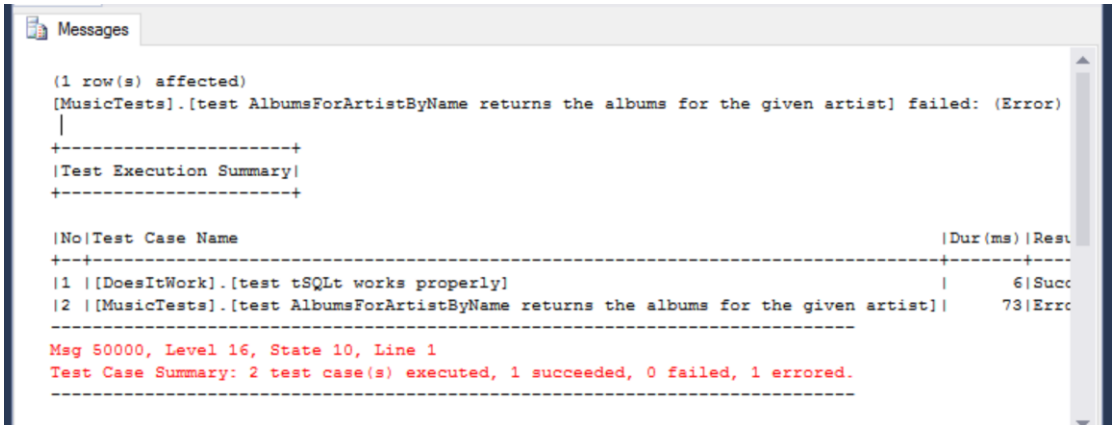
We know there are three albums in the database by Guns 'n' Roses: Appetite for Destruction, Use Your Illusion I, and Use Your Illusion II.

We can use the test-first approach here:

```
USE [Chinook] -- Ensure the Chinook database is selected
GO

EXEC tSQLt.NewTestClass 'MusicTests' -- Create a new test class for these tests
GO

CREATE PROCEDURE [MusicTests].[test AlbumsForArtistByName returns the albums
for the given artist]
AS
BEGIN
	-- Arrange
	CREATE TABLE #actual (
		AlbumTitle NVARCHAR(160)
	);

	-- Act
	INSERT INTO #actual
	EXEC dbo.AlbumsForArtistByName @artistName = N'Guns N'' Roses';

	-- Assert
	CREATE TABLE #expected (
		AlbumTitle NVARCHAR(160)
	);

	INSERT INTO #expected (AlbumTitle) VALUES (N'Appetite for Destruction');
	INSERT INTO #expected (AlbumTitle) VALUES (N'Use Your Illusion I');
	INSERT INTO #expected (AlbumTitle) VALUES (N'Use Your Illusion II');

	EXEC tSQLt.AssertEqualsTable
		@Expected = '#expected',
		@Actual = '#actual'
END;
GO
```

Now let's run that test to see it fails as expected:

```
EXEC tSQLt.RunAll
```



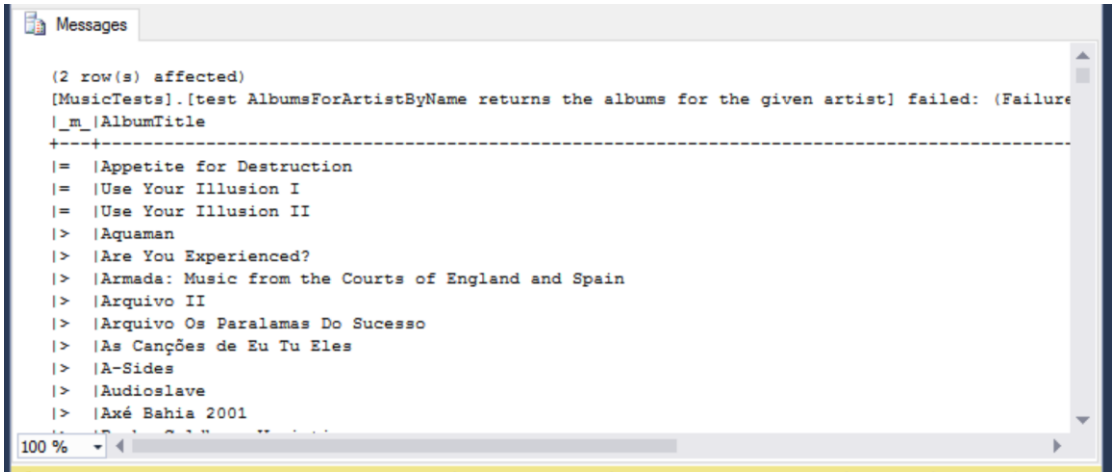
So let's create the blank SPROC to get our test compiling.

```
CREATE PROCEDURE AlbumsForArtistByName
	@artistName nvarchar(120)
AS
BEGIN
	-- SET NOCOUNT ON added to prevent extra result sets from
	-- interfering with SELECT statements.
	SET NOCOUNT ON;

END
GO
```

Now, add the first implementation to the body of the SPROC, after `SET NOCOUNT ON`:

```
SELECT dbo.Album.Title
FROM dbo.Album;
```

And re-run the tests (`EXEC tSQLt.RunAll`):

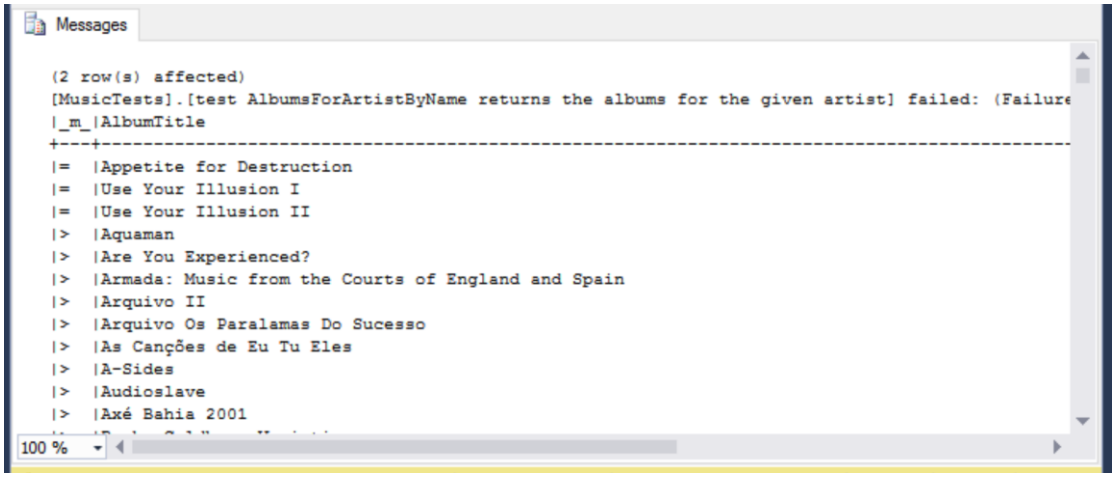


Notice how tSQLt shows us the difference between the two result sets: what appears in the expected vs. the actual results. Items that are the same are marked with an = sign; those appearing in the expected results but not the actual results are marked with a < sign; and those appearing in the actual results but not the expected results are marked with a > sign.

Let's now ensure that the result set is properly joined: the `Album` table has a foreign key on `Artist.ArtistId` that we can use in our join:

```
INNER JOIN dbo.Artist
ON dbo.Album.ArtistId = dbo.Artist.ArtistId;
```

And re-run the tests (`EXEC tSQLt.RunAll`). There's no change in the output here and the test is still failing, so we've still a little work to do.



Finally, we can add the restriction on the specified artist name, to ensure that only the albums for the requested artist are returned:

```
WHERE dbo.Artist.Name = @artistName;
```

Run the tests one last time(`EXEC tSQLt.RunAll`):

It passes, hooray!

Constraints

We can also test that the constraints we place upon our data are correct. This requires some mental gymnastics, as constraints only have observable effects when the constraints are violated. Therefore, we have to write a test that violates the constraint, detect the failure, and assert that the failure was detected.

In the Chinook database, we have a foreign key constraint from Customer to Employee, for `SupportRepId`: a Customer has a problem, and they speak to a member of the Support team (an Employee of the company) to help them resolve the problem.

Following the above logic, we can write a test that the `SupportRepId` must be valid: an error should be thrown when inserting an invalid value into the table.

Arrange

This test is a bit more involved, as you might have guessed. Let's start by creating a new test procedure as before:

```
EXEC tsqlt.NewTestClass @ClassName = N'CustomerSupport' -- nvarchar(max)
GO

CREATE PROCEDURE CustomerSupport.[test Support Rep Id should be valid]
AS
BEGIN

END;
GO
```

First, we must declare our flag that is set when the constraint is violated:

```
DECLARE @errorThrown BIT; SET @errorThrown = 0;
```

Next, we fake the Customer table. tSQLt's built-in support for faking tables allows us to test this constraint in isolation without being subject to the other constraints on the table, such as `NOT NULL` or `PRIMARY KEY` constraints.

```
EXEC tSQLt.FakeTable @TableName = N'Customer',
           @SchemaName = N'dbo';
```

And finally, we apply the foreign key constraint to the fake table. This literally copies the constraint definition `FK_CustomerSupportRepId` from the real table to the fake table.

```
EXEC tSQLt.ApplyConstraint @TableName = N'Customer',
           @ConstraintName = N'FK_CustomerSupportRepId',
           @SchemaName = N'dbo';
```

Act

Now comes the fun. We start with a try..catch block, so that we can trap the constraint violation:

```
	BEGIN TRY

	END TRY
	BEGIN CATCH
		SET @errorThrown = 1
	END CATCH;
```

Within that try block, we attempt an `INSERT` into the Customer table:

```
INSERT INTO dbo.Customer (
	FirstName, LastName, Company, Address,
	City, State, Country, PostalCode,
	Phone, Fax, Email, SupportRepId
) VALUES (
	N'Alastair', -- FirstName - nvarchar(40)
	N'Smith', -- LastName - nvarchar(20)
	N'Cloud Hub 360', -- Company - nvarchar(80)
	N'123 Main Street', -- Address - nvarchar(70)
	N'Cambridge', -- City - nvarchar(40)
	NULL, -- State - nvarchar(40)
	N'United Kingdom', -- Country - nvarchar(40)
	N'CB1 2AB', -- PostalCode - nvarchar(10)
	NULL, -- Phone - nvarchar(24)
	NULL, -- Fax - nvarchar(24)
	N'chinook@alastairsmith.me.uk', -- Email - nvarchar(60)
	0  -- SupportRepId - int
)
```

Assert

Finally, we must assert that the error was thrown, and fail the test if it was not:

```
EXEC tSQLt.AssertEquals @Expected = 1, @Actual = @errorThrown,
@Message = N'INSERT succeeded: Should not be able to write a row with an
invalid SupportRepId'
```

Test Data Builders

So, that was a lot to type, right? Particularly the `INSERT` statement, where the only thing we care about in that test is the SupportRepId column. Let's make our lives a bit easier by introducing a Test Data Builder. Test Data Builders make our tests more readable by using design techniques to abstract away aspects of our code base that the tests are not interested in. Instead of `INSERT`ing a whole new `Customer` row as we did in the previous test, we can write a new SPROC that will do that for us while also defining default values for each column in `Customer`.

Let's start by creating a schema for our test data builders, to keep them isolated from our other code:

```
IF NOT EXISTS (
	SELECT  schema_name
	FROM    information_schema.schemata
	WHERE   schema_name = 'TestDataBuilder'
)
BEGIN
	EXEC sys.sp_executesql N'CREATE SCHEMA [TestDataBuilder]'
END
GO
```

Now we'll define our Builder SPROC. We use the default parameter values feature of T-SQL to supply the default values.

```
CREATE PROCEDURE [TestDataBuilder].[CustomerBuilder]
(
	@FirstName NVARCHAR(40) = N'', @LastName NVARCHAR(20) = N'',
	@Company NVARCHAR(80) = N'', @Address NVARCHAR(70) = N'',
	@City NVARCHAR(40) = N'', @State NVARCHAR(40) = N'',
	@Country NVARCHAR(40) = N'', @PostalCode NVARCHAR(10) = N'',
	@Phone NVARCHAR(24) = N'', @Fax NVARCHAR(24) = N'',
	@Email NVARCHAR(60) = N'', @SupportRepId INT = 0,
	@CustomerId INT = 0 OUT
)
AS
BEGIN
	DECLARE @returnValue INT = 0;
	BEGIN TRY
		INSERT INTO [dbo].[Customer] (
			[FirstName], [LastName], [Company], [Address], [City],
			[State], [Country], [PostalCode], [Phone], [Fax], [Email]
		) VALUES (
			@FirstName, @LastName, @Company, @Address, @City,
			@State, @Country, @PostalCode, @Phone, @Fax, @Email
		)

		SET @CustomerId = SCOPE_IDENTITY(); -- ID from the INSERTed row
	END TRY
	BEGIN CATCH
		-- Report any errors via a tSQLt failure.
		DECLARE @ErrorMessage nvarchar(2000) =
			'[TestDataBuilder].[CustomerBuilder] - ERROR: '
			+ ERROR_MESSAGE();
		SET @ReturnValue = ERROR_NUMBER()

		EXEC tSQLt.Fail @ErrorMessage;
	END CATCH

	RETURN @returnValue;
END;
```

(Note: the [fuller example in the GitHub repository] also handles `INSERT`ing into faked tables.)

With that function defined, we can now re-write tests against Customer data in a much more compact way. The INSERT statement we previously saw is now abstracted away, and we can call our Builder function instead:

```
EXEC TestDataBuilders.CustomerBuilder @FirstName = N'Alastair';
```

This function has the side-effect of creating the given customer in the table, and we can now assume the presence of that data as needed.