CSharp Sql Tests

TL/DR You know you should be testing stored procedures/complex queries etc if you’re underwhelmed by the T-Sql based frameworks available you might like to use a nice fluent C# framework using markdown table syntax to define data! If so, read on…

## Background

We are currently preferring Dapper to Entity Framework, this means we are writing more stored procedures and naturally want to cover those stored procedures with tests. At first, we tried using xUnit tests, which worked perfectly well but the setup and teardown of test data proved a little cumbersome. We then tried switching approach to the tSQLt framework, which again worked perfectly well and had the added advantage of running against a temporary database instance with a DacPac project deployed and running each test in its own Sql transaction. However, as developers spoilt by the lovely syntax of modern languages like C#, we found the tSQLt tests quite unpleasant to both read and write!

I thought that there must be a better method that combines the best bits of both worlds and this Sql test framework is what I came up with. Hopefully, it will be useful to other people too.

## In a nutshell

The framework allows tests to be written in C# using familiar test frameworks (xUnit is used in the examples). The tests are run against a temporary instance of localDb, where a DacPac has optionally been deployed. Each test is given a SqlConnection and its own SqlTransaction which isolates it from any other tests. The framework also includes some helper classes and an easy way to define tabular data for test data setup and/or assertions.

A test looks like this:

[Fact]  
public void spFetchOrderById\_returns\_an\_order\_matching\_the\_supplied\_order\_Id()  
{  
 // the numbers in comments relate to the explanation below:   
 new LocalDbTestContext("TestDatabaseName") // 1  
 .DeployDacpac() // 2  
 .RunTest((connection, transaction) => // 3  
 {  
 // 4  
 var order = @"  
 | Id | Customers\_Id | DateCreated | Product | Quantity | Price | Notes |  
 | -- | ------------ | ----------- | ------- | -------- | ----- | ----------- |  
 | 23 | 1 | 2021/07/21 | Apples | 21 | 5.29 | emptyString |";  
  
 Given.UsingThe(\_context)  
 .TheFollowingSqlStatementIsExecuted("ALTER TABLE Orders DROP CONSTRAINT FK\_Orders\_Customers;") // 5  
 .And().TheFollowingDataExistsInTheTable("Orders", order); // 6  
  
 When.UsingThe(\_context)  
 .TheStoredProcedureIsExecutedWithReader("spFetchOrderById", ("OrderId", 23)); // 7  
  
 Then.UsingThe(\_context)  
 .TheReaderQueryResultsShouldContain(@"| Id |  
 | -- |  
 | 23 |"); // 8  
  
 });  
}

Hopefully, the test is fairly self-explanatory :) but this is what’s going on:

1. the LocalDbTestContext constructor creates a temporary instance of localDb and creates a connection to it.
2. the DeployDacpac method deploys DacPac containing the database schema we want to test.
3. \_context.RunTest(...) this is where we define an Action delegate which is the actual test, making use of the supplied connection and transaction.
4. var order = @" here some setup data is defined using the markdown table syntax, this will be parsed into a TabularData object, see the [TabularData](#tabulardata) section below.
5. Given...TheFollowingSqlStatementIsExecuted() here an arbitrary Sql command is executed, in this case removing a foreign key constraint so we only have to set up the data we specifically need for the test.
6. TheFollowingDataExistsInTheTable() this method takes the order TabularData we just defined and inserts it into the temporary database instance (inside the supplied transaction).
7. When...TheStoredProcedureIsExecutedWithReader() This method executes the named stored procedure that we are trying to test.
8. Then...TheReaderQueryResultsShouldContain() This method asserts that the result returned from the line above contains some data defined in a second tabular data string.

## Leveraging existing work

The LocalDbTestContext class’s constructor is responsible for setting everything up ready for a set of tests to be executed. For managing the temporary LocalDb instance, I am using the excellent MartinCostello.SqlLocalDb package which makes this task relatively trivial and can be found [here on GitHub](https://github.com/martincostello/sqllocaldb) and [here on Nuget](https://www.nuget.org/packages/MartinCostello.SqlLocalDb/)

For the DacPac deployment, I took inspiration from [this StackOverflow thread](https://stackoverflow.com/questions/43365451/improve-the-performance-of-dacpac-deployment-using-c-sharp)

## LocalDbTestContext RunTest method

Once the localDb instance is running and we have optionally deployed the DacPac project, we can use the RunTest() method, passing in an Action<IDbConnection, IDbTransaction> which is the test to be executed.

The action is executed in the context of a new SqlTransaction and wrapped in a try-finally block, which is used to tidy up any open DataReaders and roll back the test’s individual SqlTransaction this ensures that the tests cannot affect each other.

public LocalDbTestContext RunTest(Action<IDbConnection, IDbTransaction> useConnection)  
{  
 try  
 {  
 SqlTransaction = SqlConnection.BeginTransaction();  
 useConnection(SqlConnection, SqlTransaction);  
 }   
 finally   
 {  
 // close any open datareaders as they   
 // are against the connection and will   
 // stuff up other tests  
 if(LastQueryResult is IDataReader lastQueryResultAsReader)   
 lastQueryResultAsReader.Close();   
  
 // leave the context untouched for the next test  
 SqlTransaction?.Rollback();   
 }  
  
 return this;  
}

## Some nice extra features

### TabularData

The TabularData class is used for human-readable data definition.

We are used to defining tabular data in Markdown tables and also using Specflow’s example tables, data expressed in this format is far easier for a human to ‘parse’ than Sql statements. So, I created a class called TabularData which has methods for converting to and from markdown table strings and also converting to Sql statements, and from SqlDataReader, it also has methods for evaluating whether two TabularData are equal and whether one contains another. The code can be found [here](https://github.com/andrewjpoole/CSharpSqlTests/blob/main/CSharpSqlTests/TabularData.cs). Here is an example of its use:

var testString = @" | id | state | created | ref |  
 | -- | --------- | ---------- | ------------ |  
 | 1 | created | 2021/11/02 | 23hgf4hj3gf4 |  
 | 2 | pending | 2021/11/01 | 623kj4hv6hv4 |  
 | 3 | completed | 2021/10/31 | e0v9736eu476 |";

#### String value interpretation

The following table shows some special cases for how string values are interpreted in TabularData:

| string value | interpreted value |
| --- | --- |
| emptyString | "" |
| null | null |
| true | True |
| false | False |
| 2021/11/02T09:24:17.000 etc | DateTime.TryParse() is used, so any valid date string should work |

The string value will be parsed/matched in the following order: DateTime, Int, Decimal, Float, Double, ‘emptyString’, ‘null’, ‘true/false’, null or finally just taken as a string.

TabularData also has a static builder method in case you want to build one programmatically rather than use the markdown string etc:

var tabularData = TabularData  
 .CreateWithColumns("column1", "column2")  
 .AddRowWithValues("valueA", "valueB")  
 .AddRowWithValues("valueC", "valueD");

### Given, When and Then helper classes

Recently I have started separating out the arrange, act and assert parts of a test into Given, When and Then classes, this gives a nice fluent interface and makes the tests nice and readable. You can write whatever C# code you like in the test and you could use the connection directly with System.Data like I’m doing or via Dapper or EF etc, this framework is completely un-opinionated.

The Given class contains methods for executing Sql statements (e.g. to remove a foreign key constraint) and methods for inserting test data into a table using markdown table strings or an instance of a TabularData.

The When class contains methods for executing Stored Procedures and various types of query, the results are stored on the shared instance of the LocalDbTestContext so that the Then class can neatly access them for assertions, but there are also overloads which return the result an Out argument.

The Then class has methods for asserting that query results are equal to or contain data specified using markdown table strings or an instance of a TabularData passed in as an argument. It also has methods for executing either scaler or reader queries in case you need to assert against data in the database changed by a stored procedure under test.

All three share the instance of the LocalDbTestContext, which allows them to access its State dictionary and its LastQueryResult object. This object will contain the result of queries made against the connection, for ExecuteReader() queries it will contain an IDataReader, for ExecuteNonQuery() queries it will contain the number of rows affected. When using the When.TheStoredProcedureIsExecutedWithReturnParameter() it will contain the actual return parameter from the stored procedure.

These classes are partial so you can extend them to add more methods easily.

Here is some of the code for the Given class showing the use of System.Data types to utilise the connection:

public partial class Given  
{  
 private readonly ILocalDbTestContext \_context;  
 private readonly Action<string>? \_logAction;  
  
 public Given(ILocalDbTestContext context, Action<string>? logAction = null)  
 {  
 \_context = context;  
 \_logAction = logAction;  
 }  
  
 public static Given UsingThe(LocalDbTestContext context, Action<string>? logAction = null) => new(context, logAction);  
  
 public Given And() => this;  
  
 private void LogMessage(string message)  
 {  
 \_logAction?.Invoke(message);  
 }  
  
 public Given TheDacpacIsDeployed(string dacpacProjectName = "")  
 {  
 \_context.DeployDacpac(dacpacProjectName);  
  
 return this;  
 }  
  
 public Given TheFollowingDataExistsInTheTable(string tableName, string markdownTableString)  
 {  
 var tabularData = TabularData.FromMarkdownTableString(markdownTableString);  
 return TheFollowingDataExistsInTheTable(tableName, tabularData);  
 }  
  
 public Given TheFollowingDataExistsInTheTable(string tableName, TabularData tabularData)  
 {  
 try  
 {  
 var cmd = \_context.SqlConnection.CreateCommand();  
 cmd.CommandText = tabularData.ToSqlString(tableName);  
 cmd.CommandType = CommandType.Text;  
 cmd.Transaction = \_context.SqlTransaction;  
  
 \_context.LastQueryResult = cmd.ExecuteNonQuery();  
  
 LogMessage("TheFollowingDataExistsInTheTable executed successfully");  
  
 return this;  
 }  
 catch (Exception ex)  
 {  
 LogMessage($"Exception thrown while executing TheFollowingDataExistsInTheTable, {ex}");  
 throw;  
 }  
 }  
 // some methods removed for brevity   
}

To extend Given to add more methods you would do the following:

public partial Given  
{  
 public Given SomeOtherSetupOperationIsPerformed()  
 {  
 // do something here  
 // e.g. access the shared \_context   
 // across the Given, When and Then  
 \_context.State.Add("newStateObject", 87654)   
  
 // returning this enables the fluent method chaining.  
 return this;   
 }  
}

## Performance and efficiency

Starting a temporary localDb and deploying a DacPac are both quite expensive tasks, so it is best to do these jobs once for a set of tests, various test frameworks achieve this in different ways, in xUnit it is the IClassFixture<T>. A test class that implements this interface will have an instance of T injected into its constructor and the T will disposed after any tests have been run.

Here is the class that I am injecting using IClassFixture which creates the localDb instance and deploys the DacPac.

public class LocalDbContextFixture : IDisposable  
{  
 public LocalDbTestContext Context;  
  
 public LocalDbContextFixture(IMessageSink sink)  
 {  
 Context = new LocalDbTestContext("SampleDb", log => sink.OnMessage(new DiagnosticMessage(log)));  
 Context.DeployDacpac();  
 }   
  
 public void Dispose()  
 {  
 Context.TearDown();  
 }  
}

In my experience, the localDb takes around 10 seconds to be created and the DacPac takes another 10 seconds, but this is roughly comparable to the setup time that an equivalent tSQLt test would take.

So that’s it, a lightweight framework that sets up your db instance and then hopefully gets out of the way, letting you test your db objects however you like but providing a nice human-readable way to specify tabular data for setups and assertions. Hope you find it useful and thanks for reading!