CSharp SQL Tests

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

## Background

At ClearBank, 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 just used 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 there must be a better way to combine 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 can be run against localDb or some other SQL server. A DacPac can optionally be deployed. Each test is isolated from the other tests. Given, When and Then helper classes are included to and also a way to define tabular data for test data setup and/or assertions. More on those things later.

A test looks like this:

[Fact]  
public void spFetchOrderById\_returns\_an\_order\_matching\_the\_supplied\_order\_Id()  
{  
 // the numbers in the 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 | Product |  
 | -- | ------- |  
 | 23 | Apples |"); // 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, but you can populate data however you like.
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.
9. After the test is complete its transaction is rolled back leaving the database context unpolluted for the next test.

## 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 LocalDb instances, 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). The framework contains a class named DacPacInfo which is passed a string either containing a path to a dacpac file or a dacpac file name. If passed a path it will just use that path, otherwise it will traverse up the solution directory structure to a configurable number of levels and then use the file name to search for matching dacpac files and use the first onenit finds, this second method obviously takes longer.

## LocalDbTestContext RunTest method

The RunTest() method unsuprisingly runs a test defined in the Action<IDbConnection, IDbTransaction> passed into the method.

A new connection is opened and a new SqlTransaction created which are passed to the Action. The Action is called within a try finally block, which is then used to tidy up any open DataReader objects on the connection and roll back the SqlTransaction after the test Action has been invoked, this ensures that each test starts with a clean slate unaffected by other tests.

## Some nice extra features

### TabularData

The TabularData class can be 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 how string values are interpreted in TabularData:

| String value | Interpreted Value |
| --- | --- |
| 2021-11-03 | a DateTime, use any parsable date and time string |
| 234 | an int |
| null | null |
| emptyString | an empty string |
| true | a boolean true |
| false | a boolean false |
| “2” | a string |

The markdown table string methods work best for tables with a small number of columns, but for larger tables 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");

One thing to note, the TabularData class doesn’t know the schema of the database table it describes. So if you will be using one to populate a table, the column names and rows data types that you populate it with need to match the table schema otherwise a SqlException will be thrown when attempting to insert the data.

### 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 short which improves readability. The framework is un-opinionated about how you interact with the database, but these helper classes just use the System.Data namespace.

The Given class is responsible for the ‘arrange’ part of the test, it contains methods for inserting test data into a table using markdown table strings or an instance of a TabularData. It also contains methods for executing arbitrary SQL statements (e.g. to remove a foreign key constraint to reduce the amount of data seeding required) and

The When class is responsible for the ‘act’ part of the test and 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 is responsible for the ‘assert’ part of the test and contains 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 IDbTestContext, which allows them to access its State dictionary, its CurrentDataReader and its LastNonReaderQueryResult objects which contain the results of DataReader and non-DataReader queries against the database.

The shared IDbTestContext in these classes is public so you can extend them using extension methods, there is an example of this below.

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

public class Given  
{  
 public ILocalDbTestContext Context;  
  
 public Given(ILocalDbTestContext context)  
 {  
 Context = context;  
 }  
  
 public static Given UsingThe(LocalDbTestContext context) => new(context, logAction);  
  
 public Given And => this; // pointless syntactic sugar to make the tests read nicely  
   
 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 and the triple slash documentation is removed for brevity   
}

To extend Given to add more methods use extension methods:

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

## 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 be 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(); // If the DacPac name != database name, pass the DacPac name in here, or even better an absolute path to the file.  
 }   
  
 public void Dispose()  
 {  
 Context.TearDown();  
 }  
}

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

### Modes

The framework has the following modes:

| Name | Explanation | Use-case |
| --- | --- | --- |
| TemporaryLocalDbInstance | The framework spins up a temporary localDb instance and tears it down again afterwards | Great for local development |
| ExistingLocalDbInstanceViaInstanceName | The framework will locate and use a named pre-existing localDb instance | Can be useful in some CI scenarios, especially where the context is monitored e.g. by SqlCover |
| ExistingDatabaseViaConnectionString | The framework will connect to a SQL Server instance using the supplied connection string | Can be useful in some CI scenarios, especially where the context is monitored e.g. by SqlCover or where SQL is hosted in a container |

The mode is set in the constructor of the DbTestContext but it can also be overridden using environment variables:

| Environment Variable Name | Purpose |
| --- | --- |
| “CSharpSqlTests\_Mode” | Set the mode |
| “CSharpSqlTests\_ConnectionString” | Specify a connection string to a SQL Server instance |
| “CSharpSqlTests\_ExistingLocalDbInstanceName” | Specify the name of an existing LocalDB instance to use |

This means your code can use TemporaryLocalDbInstance by default for local dev but your CI build can switch to a different mode using environment variables.

### SQLCover

If you would also like to measure code coverage of your SQL code, there is a great tool named [SqlCover](https://github.com/GoEddie/SQLCover) which you can read about in Ed Courage’s post [Automate SQL Testing using Azure DevOps and SQLCover](https://www.clear.bank/newsroom/automate-sql-testing-using-azure-devops-and-sqlcover). If you follow the method in Ed’s post, you would deploy the DacPac during the setup of the SQL container and then use the ExistingDatabaseViaConnectionString and supply the DbTestContext with the connection string to the SQL server in the container. At ClearBank we have a yaml template which takes care of this part for us.

## In closing

All of the source code can be found on GitHub [here](https://github.com/andrewjpoole/CSharpSqlTests) Contributions are very welcome.

Packages are available on Nuget, the main [CSharpSqlTests package here](https://www.nuget.org/packages/CSharpSqlTests/), additional [Xunit extensions package here](https://www.nuget.org/packages/CSharpSqlTests.Xunit/) and additional [NUnit extensions package here](https://www.nuget.org/packages/CSharpSqlTests.NUnit/)

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 also providing a few helpful classes for common tasks. Hope you find it useful and thanks for reading!