

T.S.T.

T-SQL Test Tool

<http://tst.codeplex.com>

Version 1.8

Contents

1.	Introduction	5
1.1.	Feature Highlights.....	5
1.2.	License.....	6
2.	Key Concepts.....	6
3.	How to Setup TST	6
4.	Quick Start	7
4.1.	Quick Start: Installing the Quick Start sample database	7
4.2.	Quick Start: How to write a test	7
4.3.	Quick Start: How to run the tests	8
4.4.	Quick Start Example: Using simple Assert APIs	11
4.5.	Quick Start Example: Testing a function that returns a value.....	12
4.6.	Quick Start Example: Testing a function that returns a table	13
4.7.	Quick Start Example: Testing a stored procedure that returns a table	14
4.8.	Quick Start Example: Testing a view	16
4.9.	Quick Start Example: Testing if an expected error is raised.....	17
5.	How to run TST tests	18
5.1.	Executing TST tests by using TST.BAT	18
5.1.1.	Running all the tests in a database using TST.BAT	18
5.1.2.	Running a specific test suite using TST.BAT	18
5.1.3.	Running a specific test using TST.BAT	19
5.2.	Executing TST tests by invoking the test runner stored procedures	19
5.2.1.	Running all the tests in a database using a test runner	19
5.2.2.	Running a specific test suite using a test runner	21
5.2.3.	Running a specific test suite using a test runner	22
5.3.	Custom processing of the test results.....	23
6.	Where to store the test stored procedures.....	26
7.	How to write a test stored procedure	26
8.	How to group tests into suites	27
9.	How to specify SETUP or TEARDOWN procedures.....	28

10.	The TST Rollback.....	30
10.1.	Testing stored procedures that use transactions	31
10.2.	How to disable the TST Rollback.....	31
11.	Failures vs. Errors	33
12.	The TST API.....	34
12.1.	Assert.LogInfo	34
12.2.	Assert.Ignore.....	35
12.3.	Assert.Pass.....	35
12.4.	Assert.Fail	36
12.5.	Assert.Equals	37
12.6.	Assert.NotEquals.....	39
12.7.	Assert.NumericEquals	40
12.8.	Assert.NumericNotEquals	41
12.9.	Assert.FloatEquals.....	43
12.10.	Assert.FloatNotEquals	44
12.11.	Assert.IsLike	46
12.12.	Assert.IsNotLike	47
12.13.	Assert.IsNull.....	48
12.14.	Assert.IsNotNull	49
12.15.	Assert.TableEquals.....	50
12.15.1.	Forcing Assert.TableEquals to ignore certain columns	52
12.15.2.	How to change the case sensitivity or collation for comparing string columns	53
12.15.3.	Troubleshooting Assert.TableEquals.....	53
12.16.	Assert.IsTableNotEmpty	54
12.17.	Assert.IsTableEmpty.....	55
12.18.	Assert.RegisterExpectedError.....	56
13.	Generating XML Results.....	57
14.	TSTCheck and Self Validation of TST	58
15.	Using a custom Prefix instead of "SQLTest_"	58
16.	Code Samples	59

1. Introduction

TST is a tool that simplifies the task of writing and running test automation for code written in T-SQL. At the heart of the TST tool is the TST database. This database contains a series of stored procedures that are exposed as a test API. Part of this API is similar with the API contained in Unit Testing libraries familiar to programmers in C# or Java. Programmers can use TST to apply some of the unit testing concepts in the T-SQL world. However, certain unit testing practices that were developed in languages like C# or Java are harder to translate in the SQL world due to certain limitations of the T-SQL language. For example:

- Mocking can be impractical in T-SQL.
- Performance considerations can sometimes make difficult to separate the code in units at the granularity required by unit testing.
- SQL uses a different paradigm than languages like C# or Java. The complexity of the SET operations supported by SQL can make defining the correct tests a difficult task.

TST only works on SQL Server 2005 or higher. At this point it was tested on SQL Server 2005.

1.1.Feature Highlights

Some of the most important characteristics of TST are:

- Support for table comparison.
- A reliable implementation of Assert.Equals / Assert.NotEquals procedures. They can detect when a comparison should not be made due to incompatibility of the data types. Alternate procedures like Assert.NumericEquals, Assert.NumericNotEquals, Assert.FloatEquals, Assert.FloatNotEquals are provided.
- Can run concurrent test sessions against the same or different databases.
- Can produce XML results.
- TST automatically rolls back the database changes done during the setup and test simplifying the clean-up issues.
- When the tested code has its own transactions, TST uses a reliable way of detecting the cases where its own rollback mechanism becomes ineffective. The TST rollback can be disabled at the test, suite or global level.
- No need to register the test stored procedures; they are picked up automatically by the TST test runners.
- Can be run from the command prompt or directly by calling one of the test runner stored procedures.
- The TST infrastructure is isolated in a separate database.
- Easy to install and integrate with other test frameworks.
- Does not have any dependencies besides SQL Server.
- Very small learning curve for someone already familiar with T-SQL.

1.2. License

This project is released under the “Eclipse Public License - v 1.0”. See <http://www.eclipse.org/legal/epl-v10.html>

2. Key Concepts

Although these may seem obvious to anyone who is familiar to unit testing we will start by clarifying some key concepts.

Unit under test

This is the subject that has to be tested. It may be a stored procedure, a view or a function. In this documentation, depending on the type of the object we may refer to it as the ***tested stored procedure***, ***tested function*** or ***tested view***.

Test stored procedure

This is a stored procedure specially written to test the unit under test. Depending on your preference and requirements you can choose to store the test stored procedures in the same database where the tested objects are or in a separate database.

Test runner

These are stored procedures provided in the TST database. They can detect the test stored procedures and run them. There are several such test runners. Whenever you want to run tests you will directly or indirectly invoke one of these tests runners. See [How to run TST tests](#).

3. How to Setup TST

Prerequisites

1. SQL Server has to be installed on the machine where you run the TST setup
2. The user logged-in must have sufficient permissions on the SQL Server to create a database.

Steps to install TST

1. Download the TST tool on a local folder.

2. Run the TST.BAT file from the folder where you downloaded the TST tool. This installs the TST database on the local SQL Server.

Note: TST.BAT has several other functions. You can use it to do the following:

- Setup the TST database.
- Run tests
- Install the Quick Start database.

For more details on using TST.BAT open a command prompt, go to the download location and run: `TST.BAT /?`

4. Quick Start

If you want to get a feel for how TST can be used before reading more details here are a few examples.

All the code that we are going to show in the Quick Start section can be installed in a database named TSTQuickStart. The code source can be found in the download location in the file `DOC\SetTSTQuickStart.sql`.

The Quick Start database contains some very simple views, functions and stored procedures that stand for the code that needs to be tested. In the same database you also have the test stored procedures. In a real system you may decide to isolate the test stored procedures in a separate database. To simplify the structure of the Quick Start database we decided to store both the tested objects and the test stored procedures in the same database.

4.1.Quick Start: Installing the Quick Start sample database

To install the Quick Start database open a command prompt, go to the download location and run:

```
TST.BAT /QuickStart
```

4.2.Quick Start: How to write a test

This is as simple as creating a stored procedure with no parameters and prefix its name with 'SQLTest_'. Note that the prefix can be customized. See [Using a custom Prefix instead of "SQLTest "](#).

```
-- =====  
-- PROCEDURE SQLTest_SimplestTest  
-- This sproc is a very simple example of a test stored procedure.  
-- =====  
CREATE PROCEDURE SQLTest_SimplestTest  
AS  
BEGIN  
    DECLARE @Sum int  
  
    SET @Sum = 1 + 1  
    EXEC TST.Assert.Equals '1+1 should be 2', 2, @Sum  
END  
GO
```

More details are given in the section [How to write a test stored procedure](#).

Tests can be grouped in suites. See [How to group tests into suites](#). The 'SQLTest_' prefix can be customized. See [Using a custom Prefix instead of "SQLTest "](#).

4.3.Quick Start: How to run the tests

More details are given in the section [How to run TST tests](#). Very briefly, the tests can be run in two ways:

In the command prompt

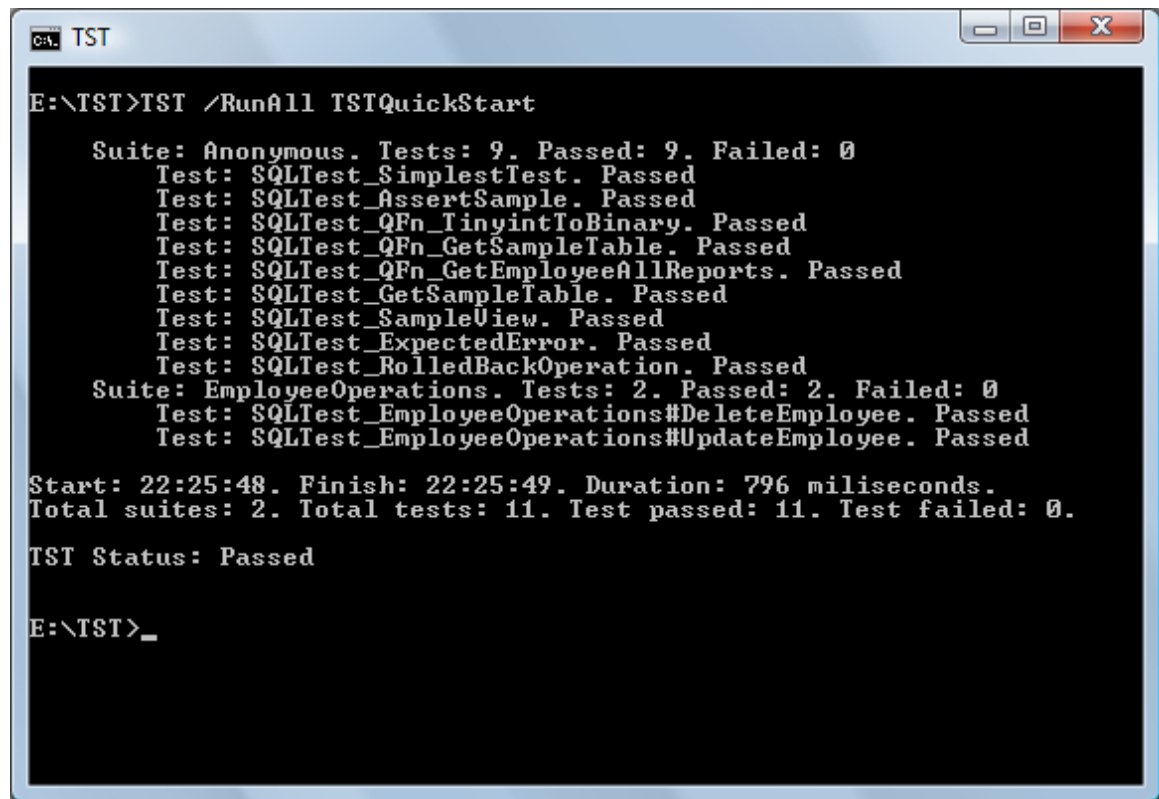
Go to the download location and run one of the following commands:

```
TST /RunAll DatabaseName or  
TST /RunSuite DatabaseName SuiteName  
TST /RunTest DatabaseName TestName
```

Here 'DatabaseName' is the name of the database where the test stored procedures are implemented.

For example:

```
TST.BAT /RunAll TSTQuickStart
```

```
E:\TST>TST /RunAll TSTQuickStart

Suite: Anonymous. Tests: 9. Passed: 9. Failed: 0
Test: SQLTest_SimplestTest. Passed
Test: SQLTest_AssertSample. Passed
Test: SQLTest_QFn_TinyintToBinary. Passed
Test: SQLTest_QFn_GetSampleTable. Passed
Test: SQLTest_QFn_GetEmployeeAllReports. Passed
Test: SQLTest_GetSampleTable. Passed
Test: SQLTest_SampleView. Passed
Test: SQLTest_ExpectedError. Passed
Test: SQLTest_RolledBackOperation. Passed
Suite: EmployeeOperations. Tests: 2. Passed: 2. Failed: 0
Test: SQLTest_EmployeeOperations#DeleteEmployee. Passed
Test: SQLTest_EmployeeOperations#UpdateEmployee. Passed

Start: 22:25:48. Finish: 22:25:49. Duration: 796 milliseconds.
Total suites: 2. Total tests: 11. Test passed: 11. Test failed: 0.

TST Status: Passed

E:\TST>_
```

Note: you can produce the report in XML format. See: [Producing XML Results](#)

In SQL Management Console

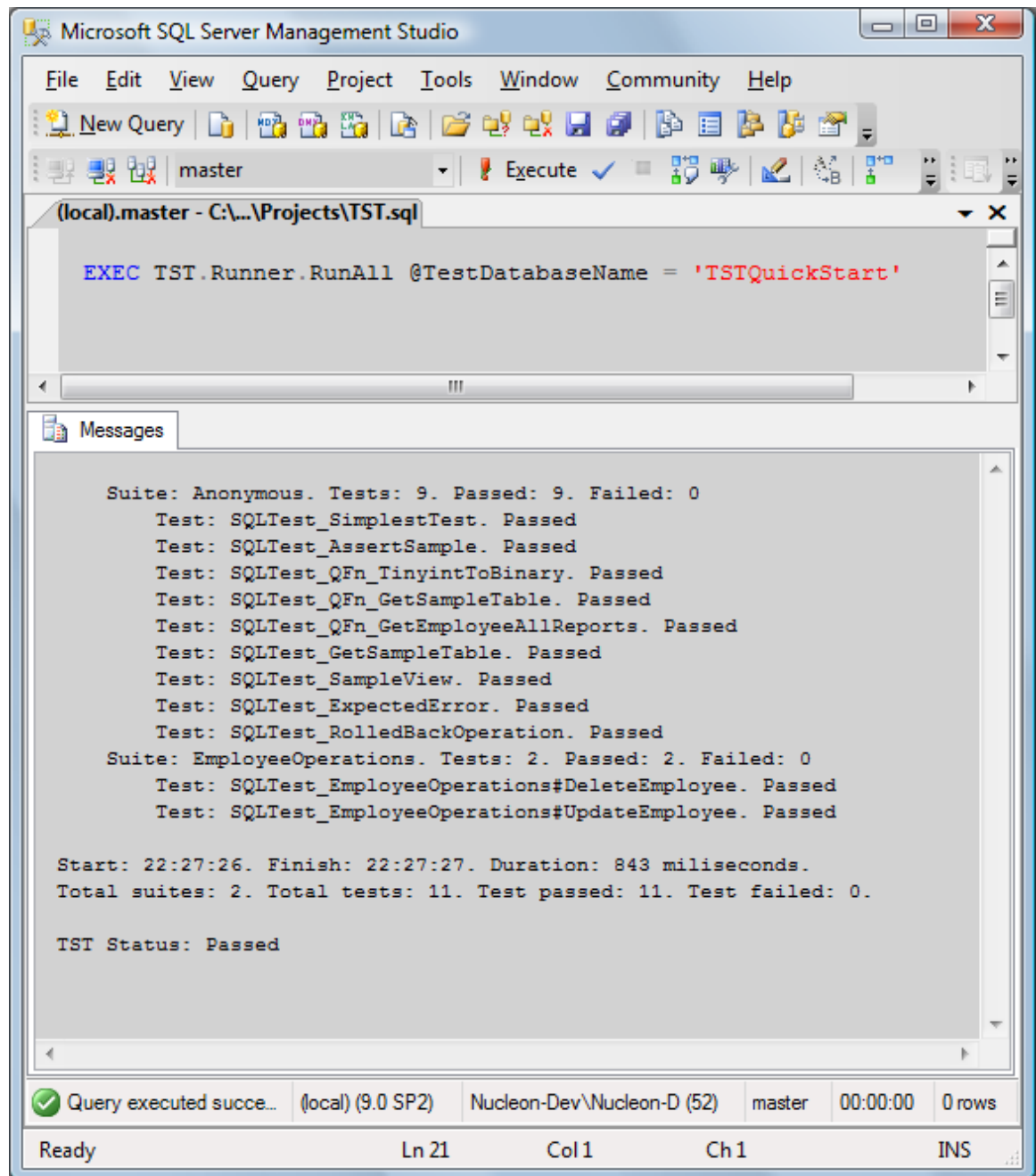
Run one of the following queries:

```
EXEC TST.Runner.RunAll 'DatabaseName'
EXEC TST.Runner.RunSuite 'DatabaseName', 'SuiteName'
EXEC TST.Runner.RunTest 'DatabaseName', 'TestName'
```

Here 'DatabaseName' is the name of the database where the test stored procedures are implemented.

For example:

```
EXEC TST.Runner.RunAll 'TSTQuickStart'
```



4.4.Quick Start Example: Using simple Assert APIs

Here is a simple example of a test stored procedure that illustrates the usage of some of the Assert APIs.

```
-- =====
-- PROCEDURE SQLTest_AssertSample
-- This sproc demonstrates simple usage of various Assert APIs.
-- =====
CREATE PROCEDURE SQLTest_AssertSample
AS
BEGIN

    DECLARE @Sum                int
    DECLARE @DecimalValue       decimal(18,10)
    DECLARE @MoneyValue         money
    DECLARE @FloatValue         float
    DECLARE @SomeString         varchar(20)

    SET @Sum = 2 + 2
    EXEC TST.Assert.Equals      '2+2 should be 4'          , 4, @Sum
    EXEC TST.Assert.NotEquals   '2+2 should not be 5'      , 5, @Sum
    EXEC TST.Assert.IsNotNull   '@Sum should not be NULL', @Sum

    SET @Sum = 2 + NULL
    EXEC TST.Assert.IsNull      '@Sum should be NULL', @Sum

    SET @DecimalValue = 10.0 / 3.0
    EXEC TST.Assert.NumericEquals
        '10.0/3.0 should be approximately 3.33',
        3.33, @DecimalValue, 0.01

    SET @MoneyValue = 1.0 / 3.0
    EXEC TST.Assert.NumericEquals
        '1.0/3.0 should be approximately 0.33',
        0.33, @MoneyValue, 0.01

    SET @FloatValue = 1.002830959602E+26
    EXEC TST.Assert.FloatEquals
        @ContextMessage = 'Atoms in one liter of water',
        @ExpectedValue  = 1.00283e+026,
        @ActualValue    = @FloatValue,
        @Tolerance       = 0.00001e+026

    SET @SomeString = 'klm abc klm'
    EXEC TST.Assert.IsLike
        '@SomeString should contain ''abc'' ',
        '%abc%',
```

```

        @SomeString

EXEC TST.Assert.IsNotLike
    '@SomeString should not contain 'xyz' ',
    '%xyz%',
    @SomeString

END
GO

```

4.5.Quick Start Example: Testing a function that returns a value

Let's take as example a function that converts a tinyint value into a string in binary format. For example it converts 129 into '10000001'. This function is called QFn_TinyintToBinary and is declared as:

```

CREATE FUNCTION dbo.QFn_TinyintToBinary (@Value tinyint)
RETURNS varchar(8)
AS ...

```

In this case the test stored procedure may look like this:

```

-- =====
-- PROCEDURE SQLTest_QFn_TinyintToBinary
-- Validates the behavior of QFn_TinyintToBinary.
-- =====
CREATE PROCEDURE SQLTest_QFn_TinyintToBinary
AS
BEGIN

    DECLARE @BinaryString varchar(8)

    SET @BinaryString = dbo.QFn_TinyintToBinary(NULL)
    EXEC TST.Assert.IsNull 'Case: NULL', @BinaryString

    SET @BinaryString = dbo.QFn_TinyintToBinary(0)
    EXEC TST.Assert.Equals 'Case: 0', '0', @BinaryString

    SET @BinaryString = dbo.QFn_TinyintToBinary(1)
    EXEC TST.Assert.Equals 'Case: 1', '1', @BinaryString

    SET @BinaryString = dbo.QFn_TinyintToBinary(2)
    EXEC TST.Assert.Equals 'Case: 2', '10', @BinaryString

```

```

SET @BinaryString = dbo.QFn_TinyintToBinary(129)
EXEC TST.Assert.Equals 'Case: 129', '10000001', @BinaryString

SET @BinaryString = dbo.QFn_TinyintToBinary(254)
EXEC TST.Assert.Equals 'Case: 254', '11111110', @BinaryString

SET @BinaryString = dbo.QFn_TinyintToBinary(255)
EXEC TST.Assert.Equals 'Case: 255', '11111111', @BinaryString

END
GO

```

4.6.Quick Start Example: Testing a function that returns a table

Let's take as example a function that returns a table with the following schema and content:

ID [int PK]	Col1 [int]	Col2[varchar(10)]
1	NULL	NULL
2	NULL	'abc'
3	0	NULL
4	0	'abc'
5	123	'xyz'

This function is called QFn_GetSampleTable and is declared as:

```

CREATE FUNCTION QFn_GetSampleTable()
RETURNS @SampleTable TABLE (
    Id    int PRIMARY KEY NOT NULL,
    Col1  int,
    Col2  varchar(10) )
AS ...

```

In this case the test stored procedure may look like this:

```

-- =====
-- PROCEDURE SQLTest_QFn_GetSampleTable
-- This sproc validates the behavior of QFn_GetSampleTable
-- =====
CREATE PROCEDURE SQLTest_QFn_GetSampleTable
AS
BEGIN

```

```

-- Create the test tables #ActualResult and #ExpectedResult
CREATE TABLE #ExpectedResult (
    Id    int PRIMARY KEY NOT NULL,
    Col1  int,
    Col2  varchar(10)
)

CREATE TABLE #ActualResult (
    Id    int PRIMARY KEY NOT NULL,
    Col1  int,
    Col2  varchar(10)
)

-- Store the expected data in #ExpectedResult
INSERT INTO #ExpectedResult (Id, Col1, Col2) VALUES (1, NULL, NULL)
INSERT INTO #ExpectedResult (Id, Col1, Col2) VALUES (2, NULL, 'abc')
INSERT INTO #ExpectedResult (Id, Col1, Col2) VALUES (3, 0, NULL)
INSERT INTO #ExpectedResult (Id, Col1, Col2) VALUES (4, 0, 'abc')
INSERT INTO #ExpectedResult (Id, Col1, Col2) VALUES (5, 123, 'xyz')

-- Store the actual data in #ActualResult
INSERT INTO #ActualResult SELECT * FROM dbo.QFn_GetSampleTable()

-- Assert.TableEquals compares the schema and content of tables
-- #ExpectedResult and #ActualResult.
EXEC TST.Assert.TableEquals 'Some contextual message here'

END
GO

```

For more details see [Assert.TableEquals](#).

4.7.Quick Start Example: Testing a stored procedure that returns a table

Testing a stored procedure that returns a table is similar with testing a function that returns a table. There is a small difference in the syntax that is used to transfer into a table the dataset returned by a stored procedure.

Let's take as example a stored procedure that returns a table with the following schema and content:

Id [int PK]	Col1 [int]	Col2[varchar(10)]
1	NULL	NULL
2	NULL	'abc'

3	0	NULL
4	0	'abc'
5	123	'xyz'

This function is called GetSampleTable and is declared as:

```
CREATE PROCEDURE GetSampleTable
AS ...
```

In this case the test stored procedure may look like this:

```
-- =====
-- PROCEDURE SQLTest_GetSampleTable
-- This sproc validates the behavior of GetSampleTable
-- =====
CREATE PROCEDURE SQLTest_GetSampleTable
AS
BEGIN

    -- Create the test tables #ActualResult and #ExpectedResult
    CREATE TABLE #ExpectedResult (
        Id      int PRIMARY KEY NOT NULL,
        Col1    int,
        Col2    varchar(10)
    )

    CREATE TABLE #ActualResult (
        Id      int PRIMARY KEY NOT NULL,
        Col1    int,
        Col2    varchar(10)
    )

    -- Store the expected data in #ExpectedResult
    INSERT INTO #ExpectedResult (Id, Col1, Col2) VALUES(1, NULL, NULL)
    INSERT INTO #ExpectedResult (Id, Col1, Col2) VALUES(2, NULL, 'abc')
    INSERT INTO #ExpectedResult (Id, Col1, Col2) VALUES(3, 0, NULL)
    INSERT INTO #ExpectedResult (Id, Col1, Col2) VALUES(4, 0, 'abc')
    INSERT INTO #ExpectedResult (Id, Col1, Col2) VALUES(5, 123, 'xyz')

    -- Store the actual data in #ExpectedResult
    INSERT INTO #ActualResult EXEC GetSampleTable

    -- Assert.TableEquals compares the schema and content of tables
    -- #ExpectedResult and #ActualResult.
    EXEC TST.Assert.TableEquals 'Some contextual message here'

END
GO
```

For more details see [Assert.TableEquals](#).

4.8.Quick Start Example: Testing a view

Testing a view is similar with testing a function or a stored procedure that returns a table.

Let's take as example a view that returns a dataset with the following schema and content:

Id [int]	Col1 [int]	Col2 [varchar(10)]
1	NULL	NULL
2	NULL	'abc'
3	0	NULL
4	0	'abc'
5	123	'xyz'

This view is called SampleView. In this case the test stored procedure may look like this:

```
-- =====
-- PROCEDURE SQLTest_SampleView
-- This sproc validates the view SampleView
-- =====
CREATE PROCEDURE SQLTest_SampleView
AS
BEGIN

    -- Create the test tables #ActualResult and #ExpectedResult
    CREATE TABLE #ExpectedResult (
        Id    int PRIMARY KEY NOT NULL,
        Col1  int,
        Col2  varchar(10)
    )

    CREATE TABLE #ActualResult (
        Id    int PRIMARY KEY NOT NULL,
        Col1  int,
        Col2  varchar(10)
    )

    -- Store the expected data in #ExpectedResult
    INSERT INTO #ExpectedResult (Id, Col1, Col2) VALUES (1, NULL, NULL)
    INSERT INTO #ExpectedResult (Id, Col1, Col2) VALUES (2, NULL, 'abc')
```



```

INSERT INTO #ExpectedResult (Id, Col1, Col2) VALUES (3, 0, NULL)
INSERT INTO #ExpectedResult (Id, Col1, Col2) VALUES (4, 0, 'abc')
INSERT INTO #ExpectedResult (Id, Col1, Col2) VALUES (5, 123, 'xyz')

-- Store the actual data in #ExpectedResult
INSERT INTO #ActualResult SELECT * FROM SampleView

-- Assert.TableEquals compares the schema and content of tables
-- #ExpectedResult and #ActualResult.
EXEC TST.Assert.TableEquals 'Some contextual message here'

END
GO

```

For more details see [Assert.TableEquals](#).

4.9.Quick Start Example: Testing if an expected error is raised

Let's take as example a scenario where we expect that the tested stored procedure will raise an error. A test stored procedure written to verify this scenario may look like this:

```

-- =====
-- PROCEDURE SQLTest_ExpectedError
-- This sproc demonstrates the concept of "expected error".
-- =====
CREATE PROCEDURE SQLTest_ExpectedError
AS
BEGIN

    EXEC TST.Assert.RegisterExpectedError
        @ContextMessage      = 'Testing RaiseAnError',
        @ExpectedErrorMessage = 'Test error'

    -- RaiseAnError will execute: RAISERROR('Test error', 16, 1)
    EXEC dbo.RaiseAnError @Raise = 1

END
GO

```

For more details see [Assert.RegisterExpectedError](#)

5. How to run TST tests

Before we go over how to run TST tests note that:

- In order for TST to recognize a stored procedure as a test, that stored procedure has to be named according to a specific convention. Basically its name has to be prefixed with 'SQLTest_'. See [How to write a test stored procedure](#).
- Tests can be written as standalone tests or grouped in suites. See [How to group tests into suites](#).

The TST database provides a few stored procedures that know how to identify tests and run them. These stored procedures are generically called test runners. You can invoke these stored procedures in SQL Server Management Console or you can call them programmatically in the same way you would call any other SQL stored procedure. You can also invoke them via the TST.BAT batch file. In both cases you can control aspects like the format of the output or the verbosity level.

5.1.Executing TST tests by using TST.BAT

5.1.1. Running all the tests in a database using TST.BAT

Open a command prompt. Go to the download location and run:

```
TST.BAT /RunAll DatabaseName  
[/Verbose] [/XmlResults PathToXmlFile]
```

For more information on the XML format see [Producing XML Results](#):

For more information on TST.BAT run TST.BAT /?

5.1.2. Running a specific test suite using TST.BAT

Open a command prompt. Go to the download location and run:

```
TST /RunSuite DatabaseName SuiteName  
[/Verbose] [/XmlResults PathToXmlFile]
```

For more information on the XML format see [Producing XML Results](#):

For more information on TST.BAT run TST.BAT /?

5.1.3. Running a specific test using TST.BAT

Open a command prompt. Go to the download location and run:

```
TST /RunTest DatabaseName TestName  
[/Verbose] [/XmlResults PathToXmlFile]
```

For more information on the XML format see [Producing XML Results](#):

For more information on TST.BAT run TST.BAT /?

5.2.Executing TST tests by invoking the test runner stored procedures

5.2.1. Running all the tests in a database using a test runner

RunAll identifies all the tests in a specified database and runs them.

Majority of the parameters for RunAll are needed only in advanced scenarios. Except the first one they have default values so you don't have to specify them. In most of the cases you only need to indicate the database that contains the test stored procedures.

Example

```
EXEC TST.Runner.RunAll @TestDatabaseName = 'TSTQuickStart'
```

Declaration

```
CREATE PROCEDURE Runner.RunAll  
    @TestDatabaseName sysname,  
    @Verbose bit = 0,  
    @ResultsFormat varchar(10) = 'Text',  
    @NoTimestamp bit = 0,  
    @CleanTemporaryData bit = 1,  
    @TestSessionId int = NULL OUT,  
    @TestSessionPassed bit = NULL OUT
```

Parameters

- @TestDatabaseName. Specifies the database that contains the test stored procedures.
- @Verbose.

- 0 - Only the failures and errors are included in the test report.
- 1 – All entries including informational log entries and the 'Pass' log entries are included in the test report.

The default value is 0.

- @ResultsFormat. For advanced use.
Specifies the format of the test report. The valid values are:
 - 'Text'. The test report is generated in plain text format. It contains a line showing the passed/failed status in the format:
TST Status: Passed or TST Status: Failed
 - 'XML'. The test report is generated in XML format. See [Producing XML Results](#)
 - 'Batch'. The same as 'Text' and additionally it prints the test session Id in the format:
TST TestSessionId: X
where X is the test session id. For more information on the test session id see the remarks section.
Useful in custom automation of the test runner.
 - 'None'. The test report is not generated.

The default value is 'Text'

- @NoTimestamp. For advanced use.
If 1 then timestamps and durations are not included in the test report.
The default value is 0.
- @CleanTemporaryData. For advanced use.
 - 1 - The test log entries are deleted before RunAll completes, just after the test report is generated.
 - 0 - The test log entries are not deleted.

This parameter is useful in custom automation of the test runner. See the section [Custom processing of the test results](#) for more information.

The default value is 1.

- @TestSessionId. OUTPUT parameter. For advanced use.
At return will contain the test session id. The test session id is explained in the remarks section.
- @TestSessionPassed. OUTPUT parameter. At return will indicate the outcome of the test session:
 - 1 – All tests passed.
 - 0 – There was at least one error or failure.

Remarks

RunAll examines all the stored procedures in the database specified by @TestDatabaseName. It identifies the tests and the test suites based on a naming convention. Basically tests are prefixed with 'SQLTest_'. For more details see [How to write a test stored procedure](#) and [How to group tests into suites](#). RunAll then executes all tests grouped by suite. Tests that don't belong to a suite are grouped in an "Anonymous" suite. If a test fails it stops at its first failure but the other tests in the same suite will still be executed.

Test Session Id

Any given execution of a TST test runner is called a test session. Each test session has an ID called the "test session id". Rows in the tables that contain the log entries are tagged with the "test session id". This allows multiple concurrent calls of the TST test runners. Using the test session id, the TST test runners are able to manage the results of each test session independently.

Making the test session id visible to the caller also makes possible custom processing of the test results. See [Custom processing of the test results](#) for more information.

5.2.2. Running a specific test suite using a test runner

RunSuite identifies all the tests in a specified database and suite and runs them.

Majority of the parameters for RunSuite are needed only in advanced scenarios. Except the first two they have default values so you don't have to specify them. In most of the cases you only need to indicate the database that contains the test stored procedures and the suite you want to run.

Example

```
EXEC TST.Runner.RunSuite
    @TestDatabaseName = 'TSTQuickStart',
    @SuiteName         = 'EmployeeOperations'
```

Declaration

```
CREATE PROCEDURE Runner.RunSuite
    @TestDatabaseName sysname,
    @SuiteName         sysname,
    @Verbose           bit = 0,
    @ResultsFormat     varchar(10) = 'Text',
    @NoTimestamp       bit = 0,
    @CleanTemporaryData bit = 1,
    @TestSessionId     int = NULL OUT,
```

```
@TestSessionPassed      bit = NULL OUT
```

Parameters

- @TestDatabaseName. Specifies the database that contains the test stored procedures.
- @SuiteName. Specifies the name of the suite you want to run.
- @Verbose. See @Verbose in [RunAll](#).
- @ResultsFormat. For advanced use. See @ResultsFormat in [RunAll](#).
- @NoTimestamp. For advanced use. See @NoTimestamp in [RunAll](#).
- @CleanTemporaryData. For advanced use. See @CleanTemporaryData in [RunAll](#).
- @TestSessionId. OUTPUT parameter. For advanced use. See @TestSessionId in [RunAll](#).
- @TestSessionPassed. OUTPUT parameter. At return will indicate the outcome of the test session:
 - 1 – All tests passed.
 - 0 – There was at least one error or failure.

Remarks

RunSuite examines all the stored procedures in the database specified by @TestDatabaseName. It identifies the specified suite and all the tests belonging to that suite based on a naming convention. See [How to group tests into suites](#). RunSuite then executes all those tests. If a test fails it stops at its first failure but the other test in the same suite will still be executed.

For more information see the remarks section for [RunAll](#).

5.2.3. Running a specific test suite using a test runner

RunTest identifies the specified test in the specified database and run it.

Majority of the parameters for RunTest are needed only in advanced scenarios. Except the first two they have default values so you don't have to specify them. In most of the cases you only need to indicate the test that you want executed and the database where that test stored procedure is implemented.

Example

```
EXEC TST.Runner.RunTest  
      @TestDatabaseName = 'TSTQuickStart',
```

```
@TestName          = 'SQLTest_AssertSample'
```

Declaration

```
CREATE PROCEDURE Runner.RunTest
    @TestDatabaseName sysname,
    @TestName          sysname,
    @Verbose           bit = 0,
    @ResultsFormat     varchar(10) = 'Text',
    @NoTimestamp       bit = 0,
    @CleanTemporaryData bit = 1,
    @TestSessionId     int = NULL OUT,
    @TestSessionPassed bit = NULL OUT
```

Parameters

- @TestDatabaseName. Specifies the database that contains the test stored procedures.
- @TestName. Specifies the name of the test.
- @Verbose. See @Verbose in [RunAll](#).
- @ResultsFormat. For advanced use. See @ResultsFormat in [RunAll](#).
- @NoTimestamp. For advanced use. See @NoTimestamp in [RunAll](#).
- @CleanTemporaryData. For advanced use. See @CleanTemporaryData in [RunAll](#).
- @TestSessionId. OUTPUT parameter. For advanced use. See @TestSessionId in [RunAll](#).
- @TestSessionPassed. OUTPUT parameter. At return will indicate the outcome of the test session:
 - 1 – All tests passed.
 - 0 – There was at least one error or failure.

Remarks

RunTest identifies the specified test stored procedure from the specified database and execute it. If the test fails it stops at its first failure.

For more information see the remarks section for [RunAll](#).

5.3. Custom processing of the test results

While a test session is executed TST will store test related information in several tables:

- In table TST.Data.Suite information about suites.
- In table TST.Data.Test information about test stored procedures.
- In table TST.Data.TestLog will store the test log entries.
- In table TST.Data.SystemErrorLog will store system error information.

The data in TST.Data.TestLog is normalized and you may find it easier to use one of the two available views: Data.TSTResults or Data.TSTResultsEx. Both these views join data from several tables to present the results in a friendlier format, Data.TSTResultsEx offering slightly more information.

Normally the caller of RunAll/RunSuite/RunTest does not need to access these tables. A typical caller will care only about the text report that is generated or about the value of the output parameter @TestSessionPassed.

If you need to write a custom processing of the test results you must do the following:

- Optionally disable the output by setting @ResultsFormat to 'None'.
- Set @CleanTemporaryData to 0.
- Store the value returned in @TestSessionId in a local variable.
- Store the value returned in @TestSessionPassed in a local variable.

In this case, after RunAll/RunSuite/RunTest completes you will be able to access the log entries from the TST tables. Since you set @CleanTemporaryData to 0 the log entries are still in those tables. You know the test session id so you can make sure that you access only those entries that correspond to your test session.

After you are done processing the log tables you should make sure the log entries generated for your test session will not remain in TestLog and SystemErrorLog. You will have to call:

```
EXEC TST.Internal.CleanSessionData @TestSessionId
```

Example

```
DECLARE @CountAllSuites          int
DECLARE @CountAllTestProcedures  int
DECLARE @CountAllRegularTestProcedures int
DECLARE @CountAllSetupProcedures int
DECLARE @CountAllTeardownProcedures int

DECLARE @CountAllTestLogEntries  int
DECLARE @CountAllInfoLogEntries  int
DECLARE @CountAllPassingLogEntries int
DECLARE @CountAllFailureLogEntries int
DECLARE @CountAllErrorLogEntries int

DECLARE @TestSessionId          int
DECLARE @TestSessionPassed      bit
```



```

EXEC TST.Runner.RunSuite
    @TestDatabaseName      = 'Northwind_Test'           ,
    @SuiteName              = 'Suite1'                  ,
    @ResultsFormat          = 'None'                    ,
    @CleanTemporaryData     = 0                          ,
    @TestSessionId          = @TestSessionId OUT        ,
    @TestSessionPassed      = @TestSessionPassed OUT

/*
SELECT * FROM TST.Data.Suite
WHERE TestSessionId = @TestSessionId

SELECT * FROM TST.Data.Test
WHERE TestSessionId = @TestSessionId

SELECT * FROM TST.Data.TestLog
WHERE TestSessionId = @TestSessionId

SELECT * FROM TST.Data.TSTResults
WHERE TestSessionId = @TestSessionId

SELECT * FROM TST.Data.TSTResultsEx
WHERE TestSessionId = @TestSessionId
*/

SELECT @CountAllSuites = COUNT(*) FROM TST.Data.Suite
WHERE TestSessionId = @TestSessionId

SELECT @CountAllTestProcedures = COUNT(*) FROM TST.Data.Test
WHERE TestSessionId = @TestSessionId

SELECT @CountAllRegularTestProcedures = COUNT(*) FROM
TST.Data.Test
WHERE TestSessionId = @TestSessionId AND SProcType = 'Test'

SELECT @CountAllSetupProcedures = COUNT(*) FROM TST.Data.Test
WHERE TestSessionId = @TestSessionId AND SProcType = 'Setup'

SELECT @CountAllTeardownProcedures = COUNT(*) FROM TST.Data.Test
WHERE TestSessionId = @TestSessionId AND SProcType = 'Teardown'

SELECT @CountAllTestLogEntries = COUNT(*) FROM TST.Data.TestLog
WHERE TestSessionId = @TestSessionId

SELECT @CountAllInfoLogEntries = COUNT(*) FROM TST.Data.TestLog
WHERE TestSessionId = @TestSessionId AND EntryType = 'L'

SELECT @CountAllPassingLogEntries = COUNT(*) FROM
TST.Data.TestLog
WHERE TestSessionId = @TestSessionId AND EntryType = 'P'

SELECT @CountAllFailureLogEntries = COUNT(*) FROM
TST.Data.TestLog
WHERE TestSessionId = @TestSessionId AND EntryType = 'F'

SELECT @CountAllErrorLogEntries = COUNT(*) FROM TST.Data.TestLog
WHERE TestSessionId = @TestSessionId AND EntryType = 'E'

```

```
-- After the test results were examined we must clean
-- the data associated with this test session.
EXEC TST.Internal.CleanSessionData @TestSessionId
```

6. Where to store the test stored procedures

One of the first things you may want to decide is where to store the test stored procedures. You have two choices:

- a. Store the test stored procedures in the same database where you store the objects that you want to test.
- b. Store the test stored procedures in a separate database.

Which way you choose depends on your preference, the build process you use and answers to questions like: “is deploying test stored procedures along with the production code acceptable/allowed?”

TST does not care one way or the other. When executing tests you have to point the TST test runners to the database where the test stored procedures are located. TST distinguishes between test stored procedures and regular stored procedures based on a naming convention.

Important: You should definitely not store the test suites in the TST database. The TST database is meant to serve tests written for multiple databases and it would be a bad idea to dump tests written for different databases in the same place. Also the TST database is dropped and recreated when you install a new version of it.

7. How to write a test stored procedure

The basic rule is that you have to prefix a test stored procedure with ‘SQLTest_’ (although the prefix is customizable; see [Using a custom Prefix instead of “SQLTest_”](#)). For example if you need to test a function named Fn_CalculateSomething you may want to create a stored procedure with the name SQLTest_Fn_CalculateSomething. It must take no parameters (or at least no parameters without default values). Its structure is shown here:

```

CREATE PROCEDURE SQLTest_Fn_CalculateSomething
AS
BEGIN

    -- Setup the test data if needed
    -- Here you can call other stored procedures or functions.
    .....

    -- Invoke the function that you want to test.
    .....

    -- Validate the actual values against the expected ones
    -- Call one or more of the Assert APIs
    EXEC TST.Assert.XXX ...

END
GO

```

When you want to run this test you have to invoke a test runner (see [How to run TST tests](#)). Then you can choose to run this test along with other tests or run it alone. You will point the test runner to the database where your test stored procedure is. Unless you specifically target one test, the test runner picks up the tests based on the fact that they are prefixed with 'SQLTest_' and runs them.

You can group several tests in one test suite. See [How to group tests into suites](#).

Note that by default, all the changes done in a database (during the execution of a test) are rolled back by the TST framework. This is implemented by a feature called TST Rollback. For more details about it including scenarios where it needs to be disabled see [TST Rollback](#).

Important: You can have transactions in the *tested stored procedure* (see [Testing stored procedures that use transactions](#)). However DO NOT use transactions in the *test stored procedure*. TST will by default rollback the database changes made during the execution of the test (that includes the Setup, Test and Teardown). If you add your own transaction in the test stored procedure and execute a rollback that will:

- Interfere with the TST Rollback.
- Cause the log entries saved during the test to be lost.

8. How to group tests into suites

There are two reasons you may want to group tests into suites:

- a. You want to be able to run all the tests in a suite with one command

- b. They have a common setup procedure that needs to run before any test in the suite.

Let's take as example the case where you have two test stored procedures and you need to group them in the same suite called SuiteX. In this case you will create two stored procedures called for example SQLTest_SuiteX#TestA and SQLTest_SuiteX#TestB. Having them prefixed with the string "SQLTest_<SuiteName>#" marks them as belonging to the same suite.

9. How to specify SETUP or TEARDOWN procedures

You can specify setup and teardown pairs at two levels:

- At the test session level.
- At the test level.

Setup and Teardown at the Test Session Level

You can provide a stored procedure to be run at the beginning of each test session or another stored procedure to be run at the end of each test session. A test session is one invocation of a TST test runner. It does not matter how many suites or tests are run during the test session, the test session setup and the test session teardown will be run only once each. If you need a test session setup you simply have to create a stored procedure called SQLTest_SESSION_SETUP. This stored procedure must declare one int parameter. When TST will call this stored procedure it will set the parameter with the value of the test session ID. See [Custom processing of the test results](#) to find how you can use this parameter. For the test session teardown you will have to declare a stored procedure called SQLTEST_SESSION_TEARDOWN. It will also take an int parameter. When TST will call this stored procedure it will set the parameter with the value of the test session ID.

Example

```
CREATE PROCEDURE SQLTest_SESSION_SETUP
    @TestSessionId int
AS
BEGIN
    EXEC TST.Assert.LogInfo 'SQLTest_SESSION_SETUP'
    ...
END
GO

CREATE PROCEDURE SQLTEST_SESSION_TEARDOWN
    @TestSessionId int
AS
BEGIN
    EXEC TST.Assert.LogInfo 'This is SQLTEST_SESSION_TEARDOWN'
    ...
END
```

GO

If the procedure `SQLTest_SESSION_SETUP` fails for any reason, no test will be performed however the procedure `SQLTest_SESSION_TEARDOWN` will still be invoked. You can place `SQLTest_SESSION_SETUP` and `SQLTest_SESSION_TEARDOWN` in any schema but they have to be in the same schema and you cannot create more than one of each. For example `Schema1.SQLTest_SESSION_SETUP` and `Schema2.SQLTest_SESSION_SETUP` will be considered illegal.

Setup and Teardown at the Test Level

If you want to provide a common setup procedure that runs before each test of a suite you have to create a stored procedure named `SQLTest_SETUP_<SuiteName>`. For example if your suite is called `SuiteX` you will have the setup implemented in a stored procedure called `SQLTest_SETUP_SuiteX`.

If you want to provide a common teardown procedure that runs after each test of a suite you have to create a stored procedure named `SQLTest_TEARDOWN_<SuiteName>`. In the case of a suite called `SuiteX` that will be called `SQLTest_TEARDOWN_SuiteX`.

If defined, the setup stored procedure is called first. If it fails for any reason the execution skips the test procedure and continues with the teardown.

If the setup procedure is not defined or completes successfully, the execution continues with the test procedure. Regardless of the outcome of the test procedure, the execution will continue with the teardown if a teardown is defined.

If defined, the teardown stored procedure will be called even after the setup or the test procedure failed. Knowing this, you have to make sure that teardown procedure works correctly regardless of how much of the setup and test procedures actually completed.

Important: Please avoid writing test teardowns unless you really need them. By default the TST framework automatically rolls back all the changes made in the Setup/Test/Teardown at the end of each test. This makes the test teardown unnecessary in most scenarios. See [TST Rollback](#) for more information. That is not the same with the test session teardown that is executed outside of the TST transaction.

Setup and teardown stored procedures can invoke test API like `Assert`, `Log`, `Assert.Equals`, `Assert.Fail` and so on. Hence they can fail as any other test procedure. Note that invoking `Assert.Pass` has no effect and is never needed in a setup or teardown. For a regular test to be certified as passing it has to invoke at least one `Assert` API. An empty test procedure or a test procedure that does not invoke any `Assert` API will fail. `Assert.Pass` will allow you to explicitly declare a test as passed. However the rule that a test has to call an `Assert` API in order to pass

does not apply to setup or teardowns. Hence there is never a need to call `Assert.Pass` in that context.

Let's assume you have the following six stored procedures:

```
SQLTest_SESSION_SETUP  
SQLTest_SESSION_TEARDOWN  
SQLTest_SETUP_SuiteX  
SQLTest_TEARDOWN_SuiteX  
SQLTest_SuiteX#TestA  
SQLTest_SuiteX#TestB.
```

When the TST test runner is asked to run the suite `SuiteX` it will execute the following steps:

1. Invoke `SQLTest_SESSION_SETUP`.
If `SQLTest_SESSION_SETUP` fails the execution will jump at step 6.
2. BEGIN TRANSACTION (unless the TST rollback is disabled).
 - a. Invoke `SQLTest_SETUP_SuiteX`.
If this fails, the execution will jump at step 2.c.
 - b. Invoke `SQLTest_SuiteX#TestA`.
 - c. Invoke `SQLTest_TEARDOWN_SuiteX`.
3. ROLLBACK TRANSACTION (unless the TST rollback is disabled).
4. BEGIN TRANSACTION (unless the TST rollback is disabled).
 - a. Invoke `SQLTest_SETUP_SuiteX`.
If this fails, the execution will jump at step 4.c.
 - b. Invoke `SQLTest_SuiteX#TestB`.
 - c. Invoke `SQLTest_TEARDOWN_SuiteX`.
5. ROLLBACK TRANSACTION (unless the TST rollback is disabled).
6. Invoke `SQLTest_SESSION_TEARDOWN`.

If the test session will include more than one suite, the procedure `SQLTest_SESSION_SETUP` will be executed only once at the beginning of the test session and `SQLTest_SESSION_TEARDOWN` will be executed only once at the end of the test session.

10. The TST Rollback

The TST test runners wrap the execution of each of your tests in a transaction that is rolled back at the end of the test. If your test is included in a suite that has a setup and a teardown, then the setup and teardown are executed inside the same transaction in the following order:

1. BEGIN TRANSACTION
 - a. Invoke the SETUP procedure (if defined)
 - b. Invoke the test stored procedure
 - c. Invoke the TEARDOWN procedure (if defined)
2. ROLLBACK TRANSACTION

Considering this, in most of the cases you should not have to write teardown stored procedures.

10.1. Testing stored procedures that use transactions

If the code you are testing uses SQL transactions, those transactions that may interfere with the [TST Rollback](#).

If the code you are testing does a BEGIN TRANSACTION / COMMIT TRANSACTION, then you are fine and the TST Rollback works as expected. In that case all the changes made by the code you are testing are rolled back at the end of the test by the TST infrastructure.

If the code you are testing does a BEGIN TRANSACTION / ROLLBACK TRANSACTION, that transaction will interfere with the transaction opened by the TST framework. In SQL Server, a ROLLBACK TRANSACTION executed in a nested transaction causes the rollback to propagate to the outermost level. This will in effect terminate the transaction opened by TST and have all the subsequent changes executed outside of a transaction. That will render the TST Rollback useless. If a scenario like this occurs, the TST framework detects it and fails the test with an error indicating that the TST Rollback could not function as expected. In this case you have two choices:

- Change the tested code by replacing the BEGIN TRANSACTION / ROLLBACK TRANSACTION with SAVE TRANSACTION / ROLLBACK TRANSACTION
- Disable the TST Rollback for that test (see [How to disable the TST Rollback](#)). In that case you may also want to write a teardown procedure.

10.2. How to disable the TST Rollback

See [Testing stored procedures that use transactions](#) for scenarios where you may need to disable the TST Rollback.

To disable the TST Rollback you have to:

1. Create a stored procedure called TSTConfig. This stored procedure has to be in the same database where your test stored procedure is. TST invokes TSTConfig (if it exists) at the beginning of each test session.
2. Inside TSTConfig you have to call a TST API called SetConfiguration.

By calling SetConfiguration you can enable or disable the TST Rollback for all the tests, for specific suites or for specific tests. You can call SetConfiguration more than one time, on each call specifying a different scope.

SetConfiguration

Example

To disable the TST Rollback only for a test called SQLTest_RolledBackOperation you have to write the following code:

```
-- =====
-- TSTConfig. TST calls this at the start of each test session
-- to allow the test client to configure TST parameters.
-- =====
CREATE PROCEDURE dbo.TSTConfig
AS
BEGIN
    EXEC TST.dbo.SetConfiguration
        @ParameterName='UseTSTRollback',
        @ParameterValue='0',
        @Scope='Test',
        @ScopeValue='SQLTest_RolledBackOperation'
END
GO
```

Declaration

```
CREATE PROCEDURE dbo.SetConfiguration
    @ParameterName    varchar(32),
    @ParameterValue    varchar(100),
    @Scope             sysname,
    @ScopeValue        sysname = NUL
```

Parameters

- @ParameterName. Specifies the configuration parameter that has to be set. The only valid value is 'UseTSTRollback'.
- @ParameterValue. Specifies parameter value. Valid values are:
 - 1 – The TST Rollback is enabled for the given scope.

- 0 – The TST Rollback is disabled for the given scope.
- @Scope. Specifies the scope for which the configuration parameter is set. Valid values are:
 - 'All' – The TST Rollback setting is in effect for all suites and tests.
 - 'Suite' – The TST Rollback setting is in effect for the suite specified in @ScopeValue.
 - 'Test' – The TST Rollback setting is in effect for the test specified in @ScopeValue.
 -
- @ScopeValue. Together with @Scope specifies the scope for which the configuration parameter is set.

Remarks

SetConfiguration called for one specific test takes precedence over SetConfiguration called for its suite or for the global scope (@Scope='All'). SetConfiguration called for one specific suite takes precedence over SetConfiguration called for the global scope (@Scope='All').

11. Failures vs. Errors

The TST framework makes the distinction between test failures, test errors and system errors.

Test Failures

Test failures occur when one of the ASSERT APIs that you call fails. For example having the following T-SQL statement inside your test will result in a test failure:

```
EXEC TST.Assert.Equals
      @ContextMessage = '...',
      @ExpectedValue  = 1,
      @ActualValue    = 2
```

A test failure causes the execution of that test to stop. The current test session will continue with the next test.

The goal of the tests you are writing is to detect failures. However while executing the tests, errors can also occur.

Test Errors

Test errors occur when during the execution of a test an error is raised. This can happen if:

- The error is explicitly raised by a RAISERROR
- SQL Server detects a run time error like a primary key violation or a conversion error.
- If an Assert API is called with an invalid parameter.

A test error causes the execution of that test to stop. The current test session will continue with the next test. When it comes to the final outcome, a test failure and a test error have the same effect: they will cause the test session to fail.

Note that TST supports the concept of “Expected Errors”. See [Assert.RegisterExpectedError](#) or [Quick Start Example: Testing that an expected error is raised](#)

System Errors

A system error is an error that occurs outside of the context of a test; during the execution of code that is part of the TST infrastructure. An example of such system error is when you invoke a test runner indicating a database that is not found.

12. The TST API

12.1. Assert.LogInfo

Records an informational log entry.

Example

```
EXEC TST.Assert.LogInfo 'Some message'
```

Declaration

```
CREATE PROCEDURE Assert.LogInfo
    @Message nvarchar(max)
```

Parameters

- @Message. The string that is recorded in the log table.

Remarks

The log entries produced by calling LogInfo are shown in the output only in verbose mode (see [How to run TST tests](#)).

12.2. Assert.Ignore

When used in a suite setup procedure it will skip the suite.

When used in a test stored procedure it will skip the code of the test procedure that follows after Assert.Ignore.

Example

```
EXEC TST.Assert.Ignore 'Some message'
```

Declaration

```
CREATE PROCEDURE Assert.Ignore  
  @Message nvarchar(max) = ''
```

Parameters

- @Message. The string that is recorded in the log table.
Default value: an empty string.

Remarks

Useful when you want to exclude a test stored procedure or an entire suite from a test session. Using Assert.Ignore is a better alternative to simply commenting code sections since the commented code may degrade in time in the sense that it may get out of sync with the database schema after schema changes are implemented.

Note that any TST API executed before the Assert.Ignore will still have effect. As a result you can place the Assert.Ignore in the middle of a test procedure. When Assert.Ignore is executed the stored procedure containing Assert.Ignore is interrupted and the test session continues with the rest of the test procedures.

Using Assert.Ignore in the test session setup, the test session teardown or in a suite teardown is illegal.

12.3. Assert.Pass

Records a success.

Example

```
EXEC TST.Assert.Pass 'Some message'
```

Declaration

```
ALTER PROCEDURE Assert.Pass  
@Message nvarchar(max) = ''
```

Parameters

- @Message. The string that is recorded in the log table.
Default value: an empty string.

Remarks

In most scenarios the success or failure of a test results from the success or failure of the Assert APIs that the test is calling. If all the Assert APIs that are called are successful the test passes. If one fails then the test fails. However, in some cases the test code may not call any Assert API. It may be that the test code detects the success or failure on different branches of an IF statement. If the test completes without calling at least one Assert or Fail, the outcome of the test will be considered suspect and the test declared as failed. In solve this one has to explicitly call Pass on the code path that corresponds to success.

If a test makes a call to Assert.Fail or a call to any kind of assert that fails then the test fails regardless of any call to Assert.Pass. For a given test, any failure or error supersedes any Assert.Pass.

The log entries produced by calling Assert.Pass are shown in the output only in verbose mode (see [How to run TST tests](#)).

12.4. Assert.Fail

Forces a test failure.

Example

```
EXEC TST.Assert.Fail 'Some message'
```

Declaration

```
ALTER PROCEDURE Assert.Fail
    @ErrorMessage nvarchar(max)
```

Parameters

- @ErrorMessage. The string that is recorded in the log table.

Remarks

A call to Fail causes the test to stop with a failure. An entry of type “Fail” is saved in the TST test log.

12.5. Assert.Equals

Validates that two SQL values of compatible types are equal.

Example

```
DECLARE @Sum      int
DECLARE @String   varchar(10)

SET @Sum      = 1 + 1
SET @String = 'a' + 'b' + 'c'

EXEC TST.Assert.Equals '1+1 should be 2'      , 2, @Sum
EXEC TST.Assert.Equals 'a+b+c should be abc', 'abc', @String
```

Declaration

```
CREATE PROCEDURE Assert.Equals
    @ContextMessage nvarchar(1000),
    @ExpectedValue  sql_variant,
    @ActualValue    sql_variant
```

Parameters

- **@ContextMessage.** A string that is appended to the log entry generated by `Assert.Equals`.
- **@ExpectedValue.** The expected value.
NULL is an invalid value for `@ExpectedValue`. If `@ExpectedValue` is NULL then `Assert.Equals` stops with an error. If you need to validate that `@ActualValue` is NULL use `Assert.IsNull` instead.
- **@ActualValue.** The value being validated.
If `@ActualValue` IS NULL then `Assert.Equals` fails.

Remarks

`Assert.Equals` only compares values of compatible type. If the values in `@ExpectedValue` and `@ActualValue` have incompatible types then `Assert.Equals` results in an error. Types are compatible if they belong to the same data type family:

Data Type Family	Data Type
Date and Time	Datetime
	Smalldatetime
Approximate numeric	Float
	Real
Exact numeric	Numeric
	Decimal
	Money
	Smallmoney
	Bigint
	Int
	Smallint
	Tinyint
	Bit
Unicode	Nvarchar
	Nchar
	Varchar
	Char
Binary	Varbinary
	Binary
Uniqueidentifier	Uniqueidentifier

`Assert.Equals` does not accept values of type float or real. Float and real are approximate data types and a direct comparison of such values may cause unreliable results. To compare values of type float or real use [Assert.FloatEquals](#). If any of `@ExpectedValue` and `@ActualValue` has the type float or real then `Assert.Equals` results in an error.

If the values contained in @ExpectedValue and @ActualValue are equal then Assert.Equals logs an entry of type “Pass” and returns.

If the values contained in @ExpectedValue and @ActualValue are not equal then Assert.Equals causes the test to stop with a failure. An entry of type “Fail” is also saved in the test log.

If an error occurs (for example @ExpectedValue IS NULL) then Assert.Equals causes the test to stop with an error. An entry of type “Error” is also saved in the TST test log.

12.6. Assert.NotEquals

Validates that two SQL values of compatible types are not equal.

Example

```
DECLARE @Count      int
DECLARE @String     varchar(10)

SET @Count = 1
SET @String = 'abc'

EXEC TST.Assert.NotEquals 'Count != 0', 0, @Count
EXEC TST.Assert.NotEquals 'abc != xyz', 'xyz', @String
```

Declaration

```
CREATE PROCEDURE Assert.NotEquals
    @ContextMessage      nvarchar(1000),
    @ExpectedNotValue    sql_variant,
    @ActualValue         sql_variant
```

Parameters

- @ContextMessage. A string that is appended to the log entry generated by Assert.NotEquals.
- @ExpectedNotValue. The value against which @ActualValue is tested. NULL is an invalid value for @ExpectedNotValue. If @ExpectedNotValue is NULL then Assert.NotEquals stops with an error. If you need to validate that @ActualValue is not NULL use Assert.IsNotNull instead.

- @ActualValue. The value being validated.
If @ActualValue IS NULL then Assert.NotEquals fails.

Remarks

Assert.NotEquals only accepts values of compatible type and does not accept values of type float or real. This behavior is consistent with that of Assert.Equals. For more information see the [Remarks](#) for Assert.Equals.

If the values contained in @ExpectedNotValue and @ActualValue are not equal then Assert.NotEquals logs an entry of type “Pass” and returns.

If the values contained in @ExpectedValue and @ActualValue are equal then Assert.NotEquals causes the test to stop with a failure. An entry of type “Fail” is also saved in the test log.

If an error occurs (for example @ExpectedNotValue IS NULL) then Assert.NotEquals causes the test to stop with an error. An entry of type “Error” is also saved in the TST test log.

12.7. Assert.NumericEquals

Validates that two numeric values are equal within a given comparison tolerance.

Example

```
DECLARE @Ammount    money
DECLARE @Ratio      decimal(10,5)

SET @Ammount  = 100.57
SET @Ratio    = 10.0/3.0

EXEC TST.Assert.NumericEquals '...', 100.57, @Ammount, 0
EXEC TST.Assert.NumericEquals '...', 3.33, @Ratio , 0.01
```

Declaration

```
CREATE PROCEDURE Assert.NumericEquals
    @ContextMessage    nvarchar(1000),
    @ExpectedValue      decimal(38, 15),
    @ActualValue        decimal(38, 15),
    @Tolerance          decimal(38, 15)
```


Parameters

- @ContextMessage. A string that is appended to the log entry generated by Assert.NumericEquals.
- @ExpectedValue. The expected value.
NULL is an invalid value for @ExpectedValue. If @ExpectedValue is NULL then Assert.NumericEquals stops with an error. If you need to validate that @ActualValue is NULL use Assert.IsNull instead.
- @ActualValue. The value being validated.
If @ActualValue IS NULL then Assert.NumericEquals fails.
- @Tolerance. Indicates the comparison tolerance.
NULL is an invalid value for @Tolerance. If @Tolerance is NULL then Assert.NumericEquals stops with an error.
@Tolerance must be greater or equal than 0. If @Tolerance is less than 0 then Assert.NumericEquals stops with an error.

Remarks

If the absolute value of the difference between @ExpectedValue and @ActualValue is less than or equal to @Tolerance then Assert.NumericEquals logs an entry of type “Pass” and returns.

When the absolute value of the difference between @ExpectedValue and @ActualValue is greater than @Tolerance then Assert.NumericEquals causes the test to stop with a failure. An entry of type “Fail” is also saved in the test log.

If an error occurs (for example @ExpectedValue IS NULL) then Assert.NumericEquals causes the test to stop with an error. An entry of type “Error” is also saved in the TST test log.

If the value you need to validate is of type float or real then you should use [Assert.FloatEquals](#) instead.

12.8. Assert.NumericNotEquals

Validates that two numeric values are not within a given comparison tolerance.

Example

```
DECLARE @Ammount money
DECLARE @Ratio decimal(10,5)

SET @Ammount = 100.57
SET @Ratio = 10.0/3.0
```

```
EXEC TST.Assert.NumericNotEquals '...', 101, @Ammount, 0
EXEC TST.Assert.NumericNotEquals '...', 3, @Ratio, 0.01
```

Declaration

```
CREATE PROCEDURE Assert.NumericNotEquals
    @ContextMessage      nvarchar(1000),
    @ExpectedNotValue    decimal(38, 15),
    @ActualValue         decimal(38, 15),
    @Tolerance           decimal(38, 15)
```

Parameters

- **@ContextMessage.** A string that is appended to the log entry generated by Assert.NumericNotEquals.
- **@ExpectedNotValue.** The value against which @ActualValue is tested.
NULL is an invalid value for @ExpectedNotValue. If @ExpectedNotValue is NULL then Assert.NumericNotEquals stops with an error. If you need to validate that @ActualValue is not NULL use Assert.IsNotNull instead.
- **@ActualValue.** The value being validated.
If @ActualValue IS NULL then Assert.NumericNotEquals fails.
- **@Tolerance.** Indicates the comparison tolerance.
NULL is an invalid value for @Tolerance. If @Tolerance is NULL then Assert.NumericNotEquals stops with an error.
@Tolerance must be greater or equal than 0. If @Tolerance is less than 0 then Assert.NumericNotEquals stops with an error.

Remarks

If the absolute value of the difference between @ExpectedValue and @ActualValue is greater than or equal to @Tolerance then Assert.NumericNotEquals logs an entry of type “Pass” and returns.

If the absolute value of the difference between @ExpectedValue and @ActualValue is less than or equal to @Tolerance then Assert.NumericNotEquals causes the test to stop with a failure. An entry of type “Fail” is also saved in the test log.

If an error occurs (for example @ExpectedValue IS NULL) then Assert.NumericNotEquals causes the test to stop with an error. An entry of type “Error” is also saved in the TST test log.

If the value you need to validate is of type float or real then you should use [Assert.FloatNotEquals](#) instead.

12.9. Assert.FloatEquals

Validates that two float or real values are equal within a given comparison tolerance.

Example

```
DECLARE @AtomsInOneLiterOfWater float
SET @AtomsInOneLiterOfWater =
    3          *      -- Atoms in a H2O molecule
    6.022e+23  *      -- Avogadro's number
    (1000      /      -- grams in a liter
    18.015)    -- The molecular weight of water

EXEC TST.Assert.FloatEquals
    @ContextMessage = 'Atoms in one liter of water',
    @ExpectedValue  = 1.00283e+026,
    @ActualValue    = @AtomsInOneLiterOfWater,
    @Tolerance      = 0.00001e+026
```

Declaration

```
CREATE PROCEDURE Assert.FloatEquals
    @ContextMessage    nvarchar(1000),
    @ExpectedValue     float(53),
    @ActualValue       float(53),
    @Tolerance         float(53)
```

Parameters

- @ContextMessage. A string that is appended to the log entry generated by Assert.FloatEquals.
- @ExpectedValue. The expected value.
NULL is an invalid value for @ExpectedValue. If @ExpectedValue is NULL then Assert.FloatEquals stops with an error. If you need to validate that @ActualValue is NULL use Assert.IsNull instead.
- @ActualValue. The value being validated.
If @ActualValue IS NULL then Assert.FloatEquals fails.

- **@Tolerance.** Indicates the comparison tolerance.
 NULL is an invalid value for **@Tolerance**. If **@Tolerance** is NULL then **Assert.FloatEquals** stops with an error.
@Tolerance must be greater or equal than 0. If **@Tolerance** is less than 0 then **Assert.FloatEquals** stops with an error.

Remarks

If the absolute value of the difference between **@ExpectedValue** and **@ActualValue** is less than or equal to **@Tolerance** then **Assert.FloatEquals** logs an entry of type “Pass” and returns.

If the absolute value of the difference between **@ExpectedValue** and **@ActualValue** is greater than **@Tolerance** then **Assert.FloatEquals** causes the test to stop with a failure. An entry of type “Fail” is also saved in the test log.

If an error occurs (for example **@ExpectedValue** IS NULL) then **Assert.FloatEquals** causes the test to stop with an error. An entry of type “Error” is also saved in the TST test log.

Important! If the value you need to validate is of other numeric type (numeric, decimal, money and so on) use [Assert.NumericEquals](#) instead. Calling **Assert.FloatEquals** for numbers with more than 15 digits causes unreliable results. The following assert passes (when in fact you would expect to fail):

```
EXEC TST.Assert.FloatEquals
  @ContextMessage      = '...',
  @ExpectedValue       = 12345678901234567890,
  @ActualValue         = 12345678901234567000,
  @Tolerance           = 0
```

This is because when converted to float both values are expressed as 1.234567890123457e+019.

12.10. Assert.FloatNotEquals

Validates that two float or real values are not within a given comparison tolerance.

Example

```
DECLARE @BigNumber float
```

```
SET @BigNumber = 15.4e+45

EXEC TST.Assert.FloatNotEquals '...', 10e+45, @BigNumber, 1e+45
```

Declaration

```
CREATE PROCEDURE Assert.FloatNotEquals
    @ContextMessage      nvarchar(1000),
    @ExpectedNotValue    float(53),
    @ActualValue         float(53),
    @Tolerance           float(53)
```

Parameters

- **@ContextMessage.** A string that is appended to the log entry generated by Assert.FloatNotEquals.
- **@ExpectedNotValue.** The value against which @ActualValue is tested. NULL is an invalid value for @ExpectedNotValue. If @ExpectedNotValue is NULL then Assert.FloatNotEquals stops with an error. If you need to validate that @ActualValue is not NULL use Assert.IsNotNull instead.
- **@ActualValue.** The value being validated. If @ActualValue IS NULL then Assert.FloatNotEquals fails.
- **@Tolerance.** Indicates the comparison tolerance. NULL is an invalid value for @Tolerance. If @Tolerance is NULL then Assert.FloatNotEquals stops with an error. @Tolerance must be greater or equal than 0. If @Tolerance is less than 0 then Assert.NumericNotEquals stops with an error.

Remarks

If the absolute value of the difference between @ExpectedValue and @ActualValue is greater than or equal to @Tolerance then Assert.FloatNotEquals logs an entry of type “Pass” and returns.

If the absolute value of the difference between @ExpectedValue and @ActualValue is less than or equal to @Tolerance then Assert.FloatNotEquals causes the test to stop with a failure. An entry of type “Fail” is also saved in the test log.

If an error occurs (for example @ExpectedValue IS NULL) then Assert.FloatNotEquals causes the test to stop with an error. An entry of type “Error” is also saved in the TST test log.

Important! If the value you need to validate is of other numeric type (numeric, decimal, money and so on) use [Assert.NumericNotEquals](#) instead. Calling `Assert.FloatNotEquals` for numbers with more than 15 digits causes unreliable results. The following assert fails (when in fact you would expect to pass):

```
EXEC TST.Assert.FloatNotEquals
    @ContextMessage      = '...',
    @ExpectedNotValue    = 12345678901234567890,
    @ActualValue         = 12345678901234567000,
    @Tolerance           = 0
```

This is because when converted to float both values are expressed as 1.234567890123457e+019.

12.11. Assert.IsLike

Validates that a string value matches an expected pattern.

Example

```
DECLARE @String varchar(20)
SET @String = 'klm abc klm'

EXEC TST.Assert.IsLike '...', '%abc%', @String
```

Declaration

```
CREATE PROCEDURE Assert.IsLike
    @ContextMessage      nvarchar(1000),
    @ExpectedLikeValue   nvarchar(max),
    @ActualValue         nvarchar(max),
    @EscapeCharacter     char = NULL
```

Parameters

- `@ContextMessage`. A string that is appended to the log entry generated by `Assert.IsLike`.
- `@ExpectedLikeValue`. The string pattern against which `@ActualValue` is tested. NULL is an invalid value for `@ExpectedLikeValue`. If `@ExpectedLikeValue` is NULL then `Assert.IsLike` stops with an error. If you need to validate that `@ActualValue` is NULL use

Assert.IsNull instead.

- @ActualValue. The value being validated.
If @ActualValue IS NULL then Assert.IsLike fails.
- @EscapeCharacter. Optional parameter.
Can be used if one needs to escape wildcard characters like %_[]^ from the pattern. See the SQL documentation for the operator LIKE.
By default is NULL case in which it has no effect.

Remarks

The validation is based on the T-SQL expression:

```
@ActualValue LIKE @ExpectedLikeValue ESCAPE @EscapeCharacter
```

If the value contained in @ActualValue matches the pattern in @ExpectedLikeValue then Assert.IsLike logs an entry of type “Pass” and returns.

If the value contained in @ActualValue does not match the pattern in @ExpectedLikeValue then Assert.IsLike causes the test to stop with a failure. An entry of type “Fail” is also saved in the test log.

If an error occurs (for example @ExpectedLikeValue IS NULL) then Assert.IsLike causes the test to stop with an error. An entry of type “Error” is also saved in the TST test log.

12.12. Assert.IsNotLike

Validates that a string value does not match an expected pattern.

Example

```
DECLARE @String varchar(20)
SET @String = 'klm abc klm'

EXEC TST.Assert.IsNotLike '...', '%xyz%', @String
```

Declaration

```
CREATE PROCEDURE Assert.IsNotLike
    @ContextMessage          nvarchar(1000),
    @ExpectedNotLikeValue    nvarchar(max),
    @ActualValue              nvarchar(max),
```

```
@EscapeCharacter      char = NULL
```

Parameters

- @ContextMessage. A string that is appended to the log entry generated by Assert.IsNotLike.
- @ExpectedNotLikeValue. The string pattern against which @ActualValue is tested. NULL is an invalid value for @ExpectedNotLikeValue. If @ExpectedNotLikeValue is NULL then Assert.IsNotLike stops with an error. If you need to validate that @ActualValue is not NULL then use Assert.IsNotNull instead.
- @ActualValue. The value being validated. If @ActualValue IS NULL then Assert.IsNotLike fails.
- @EscapeCharacter. Optional parameter. Can be used if one needs to escape wildcard characters like %_[]^ from the pattern. See the SQL documentation for the operator LIKE. By default is NULL case in which it has no effect.

Remarks

The validation is based on the T-SQL expression:

```
@ActualValue NOT LIKE @ExpectedNotLikeValue ESCAPE @EscapeCharacter
```

If the value contained in @ActualValue does not match the pattern in @ExpectedNotLikeValue then Assert.IsNotLike logs an entry of type “Pass” and returns.

If the value contained in @ActualValue matches the pattern in @ExpectedNotLikeValue then Assert.IsNotLike causes the test to stop with a failure. An entry of type “Fail” is also saved in the test log.

If an error occurs (for example @ExpectedNotLikeValue IS NULL) then Assert.IsNotLike causes the test to stop with an error. An entry of type “Error” is also saved in the TST test log.

12.13. Assert.IsNull

Validates that a SQL value is NULL.

Example

```
DECLARE @Sum int
```



```
SET @Sum = 1 + NULL

EXEC TST.Assert.IsNull '...', @Sum
```

Declaration

```
CREATE PROCEDURE Assert.IsNull
    @ContextMessage      nvarchar(1000),
    @ActualValue         sql_variant
```

Parameters

- @ContextMessage. A string that is appended to the log entry generated by Assert.IsNull.
- @ActualValue. The value being validated.

Remarks

If the value contained in @ActualValue is NULL then Assert.IsNull logs an entry of type “Pass” and returns.

If the value contained in @ActualValue is not NULL then Assert.IsNull causes the test to stop with a failure. An entry of type “Fail” is also saved in the test log.

12.14. Assert.IsNotNull

Validates that a SQL value is not NULL.

Example

```
DECLARE @Sum int
SET @Sum = 1 + 1

EXEC TST.Assert.IsNotNull '...', @Sum
```

Declaration

```
CREATE PROCEDURE Assert.IsNotNull
    @ContextMessage      nvarchar(1000),
    @ActualValue         sql_variant
```

Parameters

- @ContextMessage. A string that is appended to the log entry generated by Assert.IsNotNull.
- @ActualValue. The value being validated.

Remarks

If the value contained in @ActualValue is not NULL then Assert.IsNotNull logs an entry of type “Pass” and returns.

If the value contained in @ActualValue is NULL then Assert.IsNotNull causes the test to stop with a failure. An entry of type “Fail” is also saved in the test log.

12.15. Assert.TableEquals

Validates that the temporary tables #ExpectedResult and #ActualResult contain the same data.

Example

```
-- Create the temporary tables #ActualResult and #ExpectedResult
CREATE TABLE #ExpectedResult (
    Id  int PRIMARY KEY NOT NULL,
    C1  int,
    C2  varchar(10)
)

CREATE TABLE #ActualResult (
    Id  int PRIMARY KEY NOT NULL,
    C1  int,
    C2  varchar(10)
)

-- Store the expected data in #ExpectedResult
INSERT INTO #ExpectedResult (Id, C1, C2) VALUES (1, 100, NULL)
INSERT INTO #ExpectedResult (Id, C1, C2) VALUES (2, 200, 'abc')
INSERT INTO #ExpectedResult (Id, C1, C2) VALUES (3, 300, 'xyz')

-- Store the actual data in #ExpectedResult
-- calling the stored procedure GetSampleTable.
INSERT INTO #ActualResult EXEC GetSampleTable

-- This call compares the schema and content of tables
-- #ExpectedResult and #ActualResult.
EXEC TST.Assert.TableEquals 'Some contextual message here'
```

Declaration

```
CREATE PROCEDURE Assert.TableEquals
    @ContextMessage      nvarchar(1000),
    @IgnoredColumns      ntext = NULL
```

Parameters

- @ContextMessage. A string that is appended to the log entry generated by Assert.TableEquals.
- @IgnoredColumns. A semicolon delimited list that specifies the columns that will be ignored during the comparison. Spaces if any including leading or trailing spaces will be considered as part of the column names.
By default is NULL which means that no columns are excluded from comparison.
For more details see the section [Forcing Assert.TableEquals to ignore certain columns](#).

Remarks

The caller is responsible for creating two temporary tables with the same schema. These tables must have the names #ExpectedResult and #ActualResult. When called, Assert.TableEquals validates the fact that the two tables have the same schema and data. To make this validation possible the two temporary tables: #ExpectedResult and #ActualResult must each have a primary key defined. The primary keys in the two tables must have identical definitions.

The schema validation includes:

- The name of the columns.
- The type of the columns.
- The length of the columns. For example 200 in varchar(200).
- The collation of the columns.
- The definition of the primary keys including the order in which columns are specified in the primary key definition.

There is one exception to the schema validation. Any column specified in @IgnoredColumns does not have to be in both tables. If it is present in one table but absent in the other the schema validation will skip it.

If the two temporary tables #ExpectedResult and #ActualResult have the same schema and data then Assert.TableEquals logs an entry of type "Pass" and returns.

If any of the two temporary tables #ExpectedResult and #ActualResult is not created then Assert.TableEquals causes the test to stop with an error. An entry of type "Error" is also saved in the test log.

The schema must be the same as a prerequisite for the data comparison. If the caller of `Assert.TableEquals` fails to ensure that tables `#ExpectedResult` and `#ActualResult` have the same schema then `Assert.TableEquals` causes the test to stop with an error. An entry of type “Error” is also saved in the test log.

If the two temporary tables: `#ExpectedResult` and `#ActualResult` do not contain the same data then `Assert.TableEquals` causes the test to stop with a failure. An entry of type “Fail” is also saved in the test log.

If you need to delete the content of the two temporary tables you can call `TST.Util.DeleteTestTables`.

For sample code see [Quick Start Example: Testing a function that returns a table](#)

12.15.1. Forcing `Assert.TableEquals` to ignore certain columns

Consider the case where you need to validate the table returned by a stored procedure and that table contains data that is nondeterministic. That is the case of timestamp columns or in some scenarios datetime columns. You may not be able to predict the “correct” values for those columns and you will need to exclude them from the comparison. When you populate `#ActualResult` based on the return of a stored procedure you will most likely use a query like this:

```
INSERT INTO #ActualResult EXEC SomeStoredProcedure
```

In this case, the schema of `#ActualResult` has to coincide with the schema of the table returned by the stored procedure. If that table contains columns that you want to exclude from comparison then you can use the `@IgnoredColumns` parameter. For example, if you want to exclude the columns `[Create Date]` and `[Modified Date]` then write:

```
...
EXEC TST.Assert.TableEquals
    @ContextMessage = '...',
    @IgnoredColumns = 'Create Date;Modified Date'
...
```

Spaces (including leading or trailing spaces) in `@ IgnoredColumns` will be considered as part of the column names.

Columns that are specified in `@IgnoredColumns` don’t have to be in both `#ActualResult` and `#ExpectedResult`. In our example we are forced to have `[Create Date]` and `[Modified Date]` in `#ActualResult` because of the way we are populating it. However they don’t have to be in

#ExpectedResult. We can simply skip those two columns when creating the table #ExpectedResult.

When validating views and functions you do not need to use @IgnoredColumns. In that case you can populate the #ActualResult with a regular SELECT query that will allow you to explicitly specify the columns transferred from the function or view in the table.

12.15.2. How to change the case sensitivity or collation for comparing string columns

If you need to force the string comparison for a column to be case sensitive or case insensitive or if you need to control other collation aspects then you have to set the column collation when declaring the temporary tables.

Example: Force a case sensitive comparison of two nvarchar columns:

```
CREATE PROCEDURE SQLTest_CaseSensitiveTableComparison
AS
BEGIN
    CREATE TABLE #ExpectedResult (
        Id int PRIMARY KEY NOT NULL,
        Coll nvarchar(10) COLLATE SQL_Latin1_General_CP1_CS_AS)

    CREATE TABLE #ActualResult (
        Id int PRIMARY KEY NOT NULL,
        Coll nvarchar(10) COLLATE SQL_Latin1_General_CP1_CS_AS)

    INSERT INTO #ExpectedResult VALUES (1, 'abc')
    INSERT INTO #ExpectedResult VALUES (2, 'xyz')

    INSERT INTO #ActualResult VALUES (1, 'abc')
    INSERT INTO #ActualResult VALUES (2, 'Xyz')

    EXEC TST.Assert.TableEquals 'xyz != Xyz'

END
GO
```

12.15.3. Troubleshooting Assert.TableEquals

If you encounter errors in the test code where you create the temporary tables consider the followings:

1. Is the definition of the temp tables correct in the CREATE TABLE statement? Try to copy paste and run the create table statement in the SQL Server Management Console.
2. Is the definition of the primary key correct? Did you accidentally declared a column that participates in the primary key as nullable? In this case the error captured by TST is:
“Error: 1750, Could not create constraint. See previous errors.”
3. Are you trying to call Assert.TableEquals without creating the temporary tables? Note that temporary tables are destroyed when go out of scope. If you have nested sprocs and create the temporary table in an inner sproc, the outer sproc won't be able to access it.

12.16. Assert.IsTableNotEmpty

Validates that the temporary table #ActualResult is not empty (has one or more rows).

Example

```
CREATE TABLE #ActualResult (  
    Id int PRIMARY KEY NOT NULL,  
    C1 int,  
    C2 varchar(10)  
)  
INSERT INTO #ActualResult SELECT * FROM dbo.QFn_MyFunction()  
  
EXEC TST.Assert.IsTableNOTEmpty '...'
```

Declaration

```
CREATE PROCEDURE Assert.IsTableEmpty  
    @ContextMessage nvarchar(1000)
```

Parameters

- @ContextMessage. A string that is appended to the log entry generated by Assert.IsTableEmpty.

Remarks

The caller is responsible of creating a temporary table called #ActualResult. When called, Assert.IsTableEmpty validates the fact that #ActualResult has one or more rows.

If the temporary table #ActualResult has one or more rows then Assert.IsTableNotEmpty logs an entry of type “Pass” and returns.

If the temporary table #ActualResult is not created then Assert.IsTableNotEmpty causes the test to stop with an error. An entry of type “Error” is also saved in the test log.

If the temporary table #ActualResult has no rows then Assert.IsTableNotEmpty causes the test to stop with a failure. An entry of type “Fail” is also saved in the test log.

12.17. Assert.IsTableEmpty

Validates that the temporary table #ActualResult is empty (has no rows).

Example

```
CREATE TABLE #ActualResult (  
    Id int PRIMARY KEY NOT NULL,  
    C1 int,  
    C2 varchar(10)  
)  
INSERT INTO #ActualResult SELECT * FROM dbo.QFn_MyFunction()  
  
EXEC TST.Assert.IsTableEmpty '...'
```

Declaration

```
CREATE PROCEDURE Assert.IsTableEmpty  
    @ContextMessage nvarchar(1000)
```

Parameters

- @ContextMessage. A string that is appended to the log entry generated by Assert.IsTableEmpty.

Remarks

The caller is responsible of creating a temporary table called #ActualResult. When called, Assert.IsTableEmpty validates the fact that #ActualResult has no rows.

If the temporary table #ActualResult has no rows then Assert.IsTableEmpty logs an entry of type “Pass” and returns.

If the temporary table #ActualResult is not created then Assert.IsTableEmpty causes the test to stop with an error. An entry of type “Error” is also saved in the test log.

If the temporary table #ActualResult has one or more rows then Assert.IsTableEmpty causes the test to stop with a failure. An entry of type “Fail” is also saved in the test log.

12.18. Assert.RegisterExpectedError

Asserts the fact that the test store procedure is expected to raise a specific error.

Example

```
-- =====  
-- PROCEDURE SQLTest_ExpectedError  
-- This sproc demonstrates the concept of "expected error".  
-- =====  
CREATE PROCEDURE SQLTest_ExpectedError  
AS  
BEGIN  
  
    EXEC TST.Assert.RegisterExpectedError  
        @ContextMessage          = 'Testing RaiseAnError',  
        @ExpectedErrorMessage    = 'Test error'  
  
    -- RaiseAnError will execute: RAISERROR('Test error', 16, 1)  
    EXEC dbo.RaiseAnError @Raise = 1  
  
END  
GO
```

Declaration

```
CREATE PROCEDURE Assert.RegisterExpectedError  
    @ContextMessage          nvarchar(1000),  
    @ExpectedErrorMessage    nvarchar(2048) = NULL,  
    @ExpectedErrorProcedure  nvarchar(126) = NULL,  
    @ExpectedErrorNumber     int = NULL
```

Parameters

- @ContextMessage. A string that is appended to the log entry that will eventually be generated as a result of calling Assert.RegisterExpectedError.

- @ExpectedErrorMessage. This is the message of the error that is expected to be raised. If it is NULL then the error message is not used in matching.
- @ExpectedErrorProcedure. This is the procedure that is expected to raise the error. If it is NULL then the procedure is not used in matching
- @ExpectedErrorNumber. This is the error number of the error that is expected to be raised. If it is NULL then the error number is not used in matching.

Remarks

If the test procedure that calls RegisterExpectedError ends with an error, that error is matched against the “expected error”.

If the error that was actually raised and the expected error are a match, then a “Pass” entry is saved in the TST test log. If no other failures are recorded during the test, the test passes.

If the error that was actually raised and the expected error are not a match, the test that called RegisterExpectedError fails. An entry of type “Fail” is also saved in the TST test log.

If the test that called RegisterExpectedError completes without an error being raised then the test fails. An entry of type “Fail” is also saved in the TST test log.

Note: A call to RegisterExpectedError can be made only during the test procedure. If a call to RegisterExpectedError is made during a setup or teardown procedure then the test stops with an error. An entry of type “Error” is also saved in the TST test log.

13. Generating XML Results

TST can produce the test results in XML format.

If you run the tests from the command prompt then use the /XmlResults switch as in:

```
TST.BAT /RunAll TSTQuickStart /XmlResults PathToXmlFile
```

If you run the tests by calling a test runner stored procedure then use the @ResultsFormat parameter as in:

```
EXEC TST.Runner.RunAll
    @TestDatabaseName='TSTQuickStart',
    @ResultsFormat='XML'
```

For more details about various parameters you can use when running tests see [How to run TST tests](#)

The XML schema

```
<?xml version="1.0" encoding="UTF-8" ?>
<TST status="TSTStatus" testSessionId="TestSessionId" start="HH:MM:SS.mmm"
finish=" HH:MM:SS.mmm " duration="mmm" >
  <SystemErrors>
    <SystemError>_Error_Message_</SystemError>
    .....
  </SystemErrors>
  <Suites>
    <Suite suiteName="Name" testsCount="X" passedCount="X" failedCount="X" >
      <Tests>
        <Test name="TestSprocName" status="TestStatus"></Test>
        .....
      </Tests>
    </Suite>
    .....
  </Suites>
</TST>
```

Where

- TSTStatus is "Passed" or "Failed".
- TestStatus is "Passed" or "Failed".
- TestSessionId is the test session Id. See [Test Session Id](#)

14. TSTCheck and Self Validation of TST

In the event that you want to change the code that implements the TST features be aware that TST has a self-validation process. To run the TST self-validation, open a command prompt, go to the download location and run:

```
TSTCheck.bat
```

Note: At present TSTCheck will work correctly only on an English SQL Server. On a localized server TSTCheck may present a few false negatives.

15. Using a custom Prefix instead of "SQLTest_"

The prefix "SQLTest_" used by TST to identify the test procedures may break naming conventions that your organization may have. You can set the prefix to a custom value by running the command:

```
TST.bat /set NULL SqlTestPrefix otherPrefix_
```

This assuming that your new prefix is "otherPrefix_" and it is global per all test databases. You can also set the custom prefix for a given test database by running:

```
TST.bat /set TestDbName SqlTestPrefix otherPrefix_
```

You can also set the custom prefix by running a SQL command:

```
EXEC Utils.SetTSTVariable NULL, 'SqlTestPrefix', 'otherPrefix_'
```

or

```
EXEC Utils.SetTSTVariable TestDbName, 'SqlTestPrefix', 'otherPrefix_'
```

Note that the test database referred here is the database containing the test procedures and not TST or the database containing the tested procedures. If you change the prefix "SQLTest_", this change will impact the name of the test procedures, the name of the suite setup and suite teardown procedures and the name of the test session setup and test session teardown procedures.

16. Code Samples

Code samples can be found in the TSTQuickStart database. The code source can be found in the download location in the file DOC\SetTSTQuickStart.sql.

To install the Quick Start database open a command prompt, go to the download location and run: TST.BAT /QuickStart