Walkthrough: Creating and Running a SQL Server Unit Test

Article • 03/04/2023

In this walkthrough, you create a SQL Server unit test that verifies the behavior of several stored procedures. You create SQL Server unit tests to help identify code defects that might cause incorrect application behavior. You can run SQL Server unit tests and application tests as part of an automated suite of tests.

In this walkthrough, you perform the following tasks:

- Create a script that contains a database schema
- Create a database project and import that schema
- Deploy the database project to an isolated development environment
- Create SQL Server unit tests
- Define test logic
- Run SQL Server unit tests
- Add a negative unit test

After one of the unit tests detects an error in a stored procedure, you correct that error and re-run your test.

Prerequisites

To complete this walkthrough, you must be able to connect to a database server (or LocalDB database) on which you have permissions to create and deploy a database. For more information, see Required Permissions for Database Features of Visual Studio.

Create a Script that Contains a Database Schema

To create a script from which you can import a schema

1. On the **File** menu, point to **New**, and then click **File**.

The **New File** dialog box appears.

2. In the **Categories** list, click **General** if it is not already highlighted.
3. In the **Templates** list, click **Sql File**, and then click **Open**.

   The Transact-SQL editor opens.

4. Copy the following Transact-SQL code, and paste it into the Transact-SQL editor.

```
PRINT N'Creating Sales...';
GO
CREATE SCHEMA [Sales]
    AUTHORIZATION [dbo];
GO
PRINT N'Creating Sales.Customer...';
GO
CREATE TABLE [Sales].[Customer] (
    [CustomerID]   INT           IDENTITY (1, 1) NOT NULL,
    [CustomerName] NVARCHAR (40) NOT NULL,
    [YTDOrders]    INT           NOT NULL,
    [YTDSales]     INT           NOT NULL
);
GO
PRINT N'Creating Sales.Orders...';
GO
CREATE TABLE [Sales].[Orders] (
    [CustomerID] INT      NOT NULL,
    [OrderID]    INT      IDENTITY (1, 1) NOT NULL,
    [OrderDate]  DATETIME NOT NULL,
    [FilledDate] DATETIME NULL,
    [Status]     CHAR (1) NOT NULL,
    [Amount]     INT      NOT NULL
);
GO
PRINT N'Creating Sales.Def_Customer_YTDOrders...';
GO
ALTER TABLE [Sales].[Customer]
    ADD CONSTRAINT [Def_Customer_YTDOrders] DEFAULT 0 FOR [YT-
DOrders];
GO
PRINT N'Creating Sales.Def_Customer_YTDSales...';
GO
ALTER TABLE [Sales].[Customer]
    ADD CONSTRAINT [Def_Customer_YTDSales] DEFAULT 0 FOR [YTD-
Sales];
GO
PRINT N'Creating Sales.Def_Orders_OrderDate...';
GO
ALTER TABLE [Sales].[Orders]
    ADD CONSTRAINT [Def_Orders_OrderDate] DEFAULT GetDate() FOR
```

```
[OrderDate];
GO
PRINT N'Creating Sales.Def_Orders_Status...';
GO
ALTER TABLE [Sales].[Orders]
    ADD CONSTRAINT [Def_Orders_Status] DEFAULT 'O' FOR [Status];
GO
PRINT N'Creating Sales.PK_Customer_CustID...';
GO
ALTER TABLE [Sales].[Customer]
    ADD CONSTRAINT [PK_Customer_CustID] PRIMARY KEY CLUSTERED
([CustomerID] ASC) WITH (ALLOW_PAGE_LOCKS = ON, ALLOW_ROW_LOCKS =
ON, PAD_INDEX = OFF, IGNORE_DUP_KEY = OFF, STATISTICS_NORECOMPUTE
= OFF);
GO
PRINT N'Creating Sales.PK_Orders_OrderID...';
GO
ALTER TABLE [Sales].[Orders]
    ADD CONSTRAINT [PK_Orders_OrderID] PRIMARY KEY CLUSTERED
([OrderID] ASC) WITH (ALLOW_PAGE_LOCKS = ON, ALLOW_ROW_LOCKS =
ON, PAD_INDEX = OFF, IGNORE_DUP_KEY = OFF, STATISTICS_NORECOMPUTE
= OFF);
GO
PRINT N'Creating Sales.FK_Orders_Customer_CustID...';
GO
ALTER TABLE [Sales].[Orders]
    ADD CONSTRAINT [FK_Orders_Customer_CustID] FOREIGN KEY ([Cus-
tomerID]) REFERENCES [Sales].[Customer] ([CustomerID]) ON DELETE
NO ACTION ON UPDATE NO ACTION;
GO
PRINT N'Creating Sales.CK_Orders_FilledDate...';
GO
ALTER TABLE [Sales].[Orders]
    ADD CONSTRAINT [CK_Orders_FilledDate] CHECK ((FilledDate >=
OrderDate) AND (FilledDate < '01/01/2030'));
GO
PRINT N'Creating Sales.CK_Orders_OrderDate...';
GO
ALTER TABLE [Sales].[Orders]
    ADD CONSTRAINT [CK_Orders_OrderDate] CHECK ((OrderDate >
'01/01/2005') and (OrderDate < '01/01/2030'));
GO
PRINT N'Creating Sales.uspCancelOrder...';
GO
CREATE PROCEDURE [Sales].[uspCancelOrder]
@OrderID INT
AS
BEGIN
DECLARE @Delta INT, @CustomerID INT
BEGIN TRANSACTION
    SELECT @Delta = [Amount], @CustomerID = [CustomerID]
     FROM [Sales].[Orders] WHERE [OrderID] = @OrderID;

UPDATE [Sales].[Orders]
   SET [Status] = 'X'
```

```
WHERE [OrderID] = @OrderID;

UPDATE [Sales].[Customer]
   SET
   YTDOrders = YTDOrders - @Delta
    WHERE [CustomerID] = @CustomerID
COMMIT TRANSACTION
END
GO
PRINT N'Creating Sales.uspFillOrder...';
GO
CREATE PROCEDURE [Sales].[uspFillOrder]
@OrderID INT, @FilledDate DATETIME
AS
BEGIN
DECLARE @Delta INT, @CustomerID INT
BEGIN TRANSACTION
    SELECT @Delta = [Amount], @CustomerID = [CustomerID]
     FROM [Sales].[Orders] WHERE [OrderID] = @OrderID;

UPDATE [Sales].[Orders]
   SET [Status] = 'F',
       [FilledDate] = @FilledDate
WHERE [OrderID] = @OrderID;

UPDATE [Sales].[Customer]
   SET
   YTDSales = YTDSales - @Delta
    WHERE [CustomerID] = @CustomerID
COMMIT TRANSACTION
END
GO
PRINT N'Creating Sales.uspNewCustomer...';
GO
CREATE PROCEDURE [Sales].[uspNewCustomer]
@CustomerName NVARCHAR (40)
AS
BEGIN
INSERT INTO [Sales].[Customer] (CustomerName) VALUES (@Customer-
Name);
RETURN SCOPE_IDENTITY()
END
GO
PRINT N'Creating Sales.uspPlaceNewOrder...';
GO
CREATE PROCEDURE [Sales].[uspPlaceNewOrder]
@CustomerID INT, @Amount INT, @OrderDate DATETIME, @Status CHAR
(1)='O'
AS
BEGIN
DECLARE @RC INT
BEGIN TRANSACTION
INSERT INTO [Sales].[Orders] (CustomerID, OrderDate, FilledDate,
Status, Amount)
     VALUES (@CustomerID, @OrderDate, NULL, @Status, @Amount)
```

```
SELECT @RC = SCOPE_IDENTITY();
UPDATE [Sales].[Customer]
   SET
   YTDOrders = YTDOrders + @Amount
    WHERE [CustomerID] = @CustomerID
COMMIT TRANSACTION
RETURN @RC
END
GO
CREATE PROCEDURE [Sales].[uspShowOrderDetails]
@CustomerID INT=0
AS
BEGIN
SELECT [C].[CustomerName], CONVERT(date, [O].[OrderDate]),
CONVERT(date, [O].[FilledDate]), [O].[Status], [O].[Amount]
  FROM [Sales].[Customer] AS C
  INNER JOIN [Sales].[Orders] AS O
     ON [O].[CustomerID] = [C].[CustomerID]
  WHERE [C].[CustomerID] = @CustomerID
END
GO
```

5. Save the file. Make note of the location because you must use this script in the next procedure.

6. On the **File** menu, click **Close Solution**.

   Next you create a database project and import the schema from the script that you have created.

Create a Database Project and Import a Schema

To create a database project

1. On the **File** menu, point to **New**, and click **Project**.

   The **New Project** dialog box appears.

2. Under **Installed Templates**, select the **SQL Server** node, and then select **SQL Server Database Project**.

3. In **Name**, type **SimpleUnitTestDB**.

4. Select the **Create directory for solution** check box if it is not already selected.

5. Clear the **Add to Source Control** check box if it is not already cleared, and click **OK**.

The database project is created and appears in **Solution Explorer**. Next you import the database schema from a script.

To import a database schema from a script

1. On the **Project** menu, click **Import** and then **Script (*.sql)**.
2. Click **Next** after you read the Welcome page.
3. Click **Browse**, and go to the directory where you saved the .sql file.
4. Double-click the .sql file, and click **Finish**.

   The script is imported, and the objects that are defined in that script are added to your database project.

5. Review the summary, and then click **Finish** to complete the operation.

   > ⓘ **Note**
   >
   > The Sales.uspFillOrder procedure contains an intentional coding error that you will discover and correct later in this procedure.

To examine the resulting project

1. In **Solution Explorer**, examine the script files that were imported into the project.
2. In **SQL Server Object Explorer**, look at the database in the Projects node.

Deploying to LocalDB

By default, when you press F5, you deploy (or publish) the database to a LocalDB database. You can change the database location by going to the Debug tab of the project's property page and changing the connection string.

Create SQL Server Unit Tests

To create a SQL Server unit test for the stored procedures

1. In **SQL Server Object Explorer**, expand the projects node for **SimpleUnitTestDB** and then expand **Progammability** and then the **Stored Procedures** node.

2. Right-click one of the stored procedures, and click **Create Unit Tests** to display the **Create Unit Tests** dialog box.
3. Select the check boxes for all five stored procedures: **Sales.uspCancelOrder**, **Sales.uspFillOrder**, **Sales.uspNewCustomer**, **Sales.uspPlaceNewOrder**, and **Sales.uspShowOrderDetails**.
4. In **Project** drop-down list, select **Create a new Visual C# test project**.
5. Accept the default names for the project name and class name, and click **OK**.
6. In the test configuration dialog box, in the **Execute unit tests using the following data connection**, specify a connection to the database that you deployed earlier in this walkthrough. For example, if you used the default deployment location, which is LocalDB, you would click **New Connection** specify **(LocalDB)\Projects**. Then, choose the name of the database. Then, click OK to close the **Connection Properties** dialog box.

   > ⓘ **Note**
   >
   > If you must test views or stored procedures that have restricted permissions, you would typically specify that connection in this step. You would then specify the secondary connection, with broader permissions, to validate the test. If you have a secondary connection, you should add that user to the database project, and create a login for that user in the pre-deployment script.

7. In the test configuration dialog box, in the **Deployment** section, select the **Automatically deploy the database project before unit tests are run** check box.
8. In **Database project**, click **SimpleUnitTestDB.sqlproj**.
9. In **Deployment configuration**, click **Debug**.

   You might also generate test data as part of your SQL Server unit tests. For this walkthrough, you will skip that step because the tests will create their own data.

10. Click **OK**.

    The test project builds and the SQL Server Unit Test Designer appears. Next, you will update test logic in the Transact-SQL script of the unit tests.

Define Test Logic

This very simple database has two tables, Customer and Order. You update the database by using the following stored procedures:

- uspNewCustomer - This stored procedure adds a record to the Customer table, which sets the customer's YTDOrders and YTDSales columns to zero.
- uspPlaceNewOrder - This stored procedure adds a record to the Orders table for the specified customer and updates the YTDOrders value on the corresponding record in the Customer table.
- uspFillOrder - This stored procedure updates a record in the Orders table by changing the status from 'O' to 'F' and increments the YTDSales amount on the corresponding record in the Customer table.
- uspCancelOrder - This stored procedure updates a record in the Orders table by changing the status from 'O' to 'X' and decrements the YTDOrders amount on the corresponding record in the Customer table.
- uspShowOrderDetails - This stored procedure joins the Orders table with the Custom table and shows the records for a specific customer.

Note

This example illustrates how to create a simple SQL Server unit test. In a real-world database, you could sum the total amounts of all orders with a status of 'O' or 'F' for a particular customer. The procedures in this walkthrough also contain no error handling. For example, they do not prevent you from calling uspFillOrder for an order that has already been filled.

The tests assume that the database starts in a clean state. You will create tests that verify the following conditions:

- uspNewCustomer - Verify that the Customer table contains one row after you run the stored procedure.
- uspPlaceNewOrder - For the customer who has a CustomerID of 1, place an order for $100. Verify that the YTDOrders amount for that customer is 100 and that the YTDSales amount is zero.
- uspFillOrder - For the customer who has a CustomerID of 1, place an order for $50. Fill that order. Verify that YTDOrders and YTDSales amounts are both 50.
- uspShowOrderDetails - For the customer who has a CustomerID of 1, place orders for $100, $50, and $5. Verify that uspShowOrderDetails returns the right number of

columns and that the result set has the expected checksum.

ⓘ **Note**

For a complete set of SQL Server unit tests, you would typically verify that the other columns were set correctly. To keep this walkthrough at a manageable size, it does not describe how to verify the behavior of uspCancelOrder.

To write the SQL Server unit test for uspNewCustomer

1. In the navigation bar of the SQL Server Unit Test Designer, click **Sales_uspNewCustomerTest**, and make sure that **Test** is highlighted in the adjacent list.

   After you perform the previous step, you can create the test script for the test action in the unit test.

2. Update the Transact-SQL statements in the Transact-SQL editor to match the following statements:

```
-- ssNoVersion unit test for Sales.uspNewCustomer
DECLARE @RC AS INT, @CustomerName AS NVARCHAR (40);

SELECT @RC = 0,
       @CustomerName = 'Fictitious Customer';

EXECUTE @RC = [Sales].[uspNewCustomer] @CustomerName;

SELECT * FROM [Sales].[Customer];
```

3. In the **Test Conditions** pane, click the Inconclusive test condition, and then click the **Delete Test Condition** icon (the red X).

4. In the **Test Conditions** pane, click **Row Count** in the list, and then click the **Add Test Condition** icon (the green +).

5. Open the **Properties** window (select the test condition and press F4), and set the **Row Count** property to 1.

6. On the **File** menu, click **Save All**.

   Next you define the unit test logic for uspPlaceNewOrder.

To write the SQL Server unit test for uspPlaceNewOrder

1. In the navigation bar of the SQL Server Unit Test Designer, click **Sales_uspPlaceNewOrderTest**, and make sure that **Test** is highlighted in the adjacent list.

   After you perform this step, you can create the test script for the test action in the unit test.

2. Update the Transact-SQL statements in the Transact-SQL editor to match the following statements:

   ```
   -- ssNoVersion unit test for Sales.uspPlaceNewOrder
   DECLARE @RC AS INT, @CustomerID AS INT, @Amount AS INT, @Order-
   Date AS DATETIME, @Status AS CHAR (1);
   DECLARE @CustomerName AS NVARCHAR(40);

   SELECT @RC = 0,
          @CustomerID = 0,
          @CustomerName = N'Fictitious Customer',
          @Amount = 100,
          @OrderDate = getdate(),
          @Status = 'O';

   -- NOTE: Assumes that you inserted a Customer record with
   CustomerName='Fictitious Customer' in the pre-test script.
   SELECT @CustomerID = [CustomerID] FROM [Sales].[Customer] WHERE
   [CustomerName] = @CustomerName;

   -- place an order for that customer
   EXECUTE @RC = [Sales].[uspPlaceNewOrder] @CustomerID, @Amount,
   @OrderDate, @Status;

   -- verify that the YTDOrders value is correct.
   SELECT @RC = [YTDOrders] FROM [Sales].[Customer] WHERE [Cus-
   tomerID] = @CustomerID

   SELECT @RC AS RC
   ```

3. In the **Test Conditions** pane, click the Inconclusive test condition, and click **Delete Test Condition**.

4. In the **Test Conditions** pane, click **Scalar Value** in the list, and then click **Add Test Condition**.

5. In the **Properties** window, set the **Expected Value** property to 100.

6. In the navigation bar of the SQL Server Unit Test Designer, click **Sales_uspPlaceNewOrderTest**, and make sure that **Pre-Test** is highlighted in the adjacent list.

   After you perform this step, you can specify statements that put your data into the state that is required to execute your test. For this example, you must create the Customer record before you can place an order.

7. Click **Click here to create** to create a pre-test script.

8. Update the Transact-SQL statements in the Transact-SQL editor to match the following statements:

```
/*
Add Transact-SQL statements here that you want to run before
the test script is run.
*/
-- Add a customer for this test with the name 'Fictitious
Customer'
DECLARE @NewCustomerID AS INT, @CustomerID AS INT, @RC AS INT,
@CustomerName AS NVARCHAR (40);

SELECT @RC = 0,
       @NewCustomerID = 0,
   @CustomerID = 0,
       @CustomerName = N'Fictitious Customer';

IF NOT EXISTS(SELECT * FROM [Sales].[Customer] WHERE CustomerName
= @CustomerName)
BEGIN
EXECUTE @NewCustomerID = [Sales].[uspNewCustomer] @CustomerName;
END

-- NOTE: Assumes that you inserted a Customer record with
CustomerName='Fictitious Customer' in the pre-test script.
SELECT @CustomerID = [CustomerID] FROM [Sales].[Customer] WHERE
[CustomerName] = @CustomerName;

-- delete any old records in the Orders table and clear out the
YTD Sales/Orders fields
DELETE from [Sales].[Orders] WHERE [CustomerID] = @CustomerID;
UPDATE [Sales].[Customer] SET YTDOrders = 0, YTDSales = 0 WHERE
[CustomerID] = @CustomerID;
```

9. On the **File** menu, click **Save All**.

   Next you create the unit test for uspFillOrder.

To write the SQL Server unit test for uspFillOrder

1. In the navigation bar of the SQL Server Unit Test Designer, click **Sales_uspFillOrderTest**, and make sure that **Test** is highlighted in the adjacent list.

   After you perform this step, you can create the test script for the test action in the unit test.

2. Update the Transact-SQL statements in the Transact-SQL editor to match the following statements:

```
-- ssNoVersion unit test for Sales.uspFillOrder
DECLARE @RC AS INT, @CustomerID AS INT, @Amount AS INT, @Filled-
Date AS DATETIME, @Status AS CHAR (1);
DECLARE @CustomerName AS NVARCHAR(40), @OrderID AS INT;

SELECT @RC = 0,
       @CustomerID = 0,
       @OrderID = 0,
       @CustomerName = N'Fictitious Customer',
       @Amount = 100,
       @FilledDate = getdate(),
       @Status = 'O';

-- NOTE: Assumes that you inserted a Customer record with
CustomerName='Fictitious Customer' in the pre-test script.
SELECT @CustomerID = [CustomerID] FROM [Sales].[Customer] WHERE
[CustomerName] = @CustomerName;
-- Get the most recently added order.
SELECT @OrderID = MAX([OrderID]) FROM [Sales].[Orders] WHERE
[CustomerID] = @CustomerID;

-- fill an order for that customer
EXECUTE @RC = [Sales].[uspFillOrder] @OrderID, @FilledDate;

-- verify that the YTDOrders value is correct.
SELECT @RC = [YTDSales] FROM [Sales].[Customer] WHERE [Cus-
tomerID] = @CustomerID

SELECT @RC AS RC;
```

3. In the **Test Conditions** pane, click the Inconclusive test condition, and click **Delete Test Condition**.

4. In the **Test Conditions** pane, click **Scalar Value** in the list, and then click **Add Test Condition**.

5. In the **Properties** window, set the **Expected Value** property to 100.

6. In the navigation bar of the SQL Server Unit Test Designer, click **Sales_uspFillOrderTest**, and make sure that **Pre-Test** is highlighted in the adjacent list. After you perform this step, you can specify statements that put your data into the state that is required to execute your test. For this example, you must create the Customer record before you can place an order.

7. Click **Click here to create** to create a pre-test script.

8. Update the Transact-SQL statements in the Transact-SQL editor to match the following statements:

```
/*
Add Transact-SQL statements here that you want to run before
the test script is run.
*/
BEGIN TRANSACTION

-- Add a customer for this test with the name 'CustomerB'
DECLARE @NewCustomerID AS INT, @RC AS INT, @CustomerName AS
NVARCHAR (40);

SELECT @RC = 0,
       @NewCustomerID = 0,
       @CustomerName = N'Fictitious Customer';

IF NOT EXISTS(SELECT * FROM [Sales].[Customer] WHERE CustomerName
= @CustomerName)
BEGIN
EXECUTE @NewCustomerID = [Sales].[uspNewCustomer] @CustomerName;
END

DECLARE @CustomerID AS INT, @Amount AS INT, @OrderDate AS
DATETIME, @Status AS CHAR (1);

SELECT @RC = 0,
       @CustomerID = 0,
       @CustomerName = N'Fictitious Customer',
       @Amount = 100,
       @OrderDate = getdate(),
       @Status = 'O';

-- NOTE: Assumes that you inserted a Customer record with
CustomerName='Fictitious Customer' in the pre-test script.
SELECT @CustomerID = [CustomerID] FROM [Sales].[Customer] WHERE
[CustomerName] = @CustomerName;

-- delete any old records in the Orders table and clear out the
YTD Sales/Orders fields
DELETE from [Sales].[Orders] WHERE [CustomerID] = @CustomerID;
UPDATE [Sales].[Customer] SET YTDOrders = 0, YTDSales = 0 WHERE
```

```
[CustomerID] = @CustomerID;

-- place an order for that customer
EXECUTE @RC = [Sales].[uspPlaceNewOrder] @CustomerID, @Amount,
@OrderDate, @Status;

COMMIT TRANSACTION
```

9. On the **File** menu, click **Save All**.

To write the SQL Server unit test for uspShowOrderDetails

1. In the navigation bar of the SQL Server Unit Test Designer, click **Sales_uspShowOrderDetailsTest**, and make sure that **Test** is highlighted in the adjacent list.

   After you perform this step, you can create the test script for the test action in the unit test.

2. Update the Transact-SQL statements in the Transact-SQL editor to match the following statements:

```
-- ssNoVersion unit test for Sales.uspFillOrder
DECLARE @RC AS INT, @CustomerID AS INT, @Amount AS INT, @Filled-
Date AS DATETIME, @Status AS CHAR (1);
DECLARE @CustomerName AS NVARCHAR(40), @OrderID AS INT;

SELECT @RC = 0,
       @CustomerID = 0,
       @OrderID = 0,
       @CustomerName = N'Fictitious Customer',
       @Amount = 100,
       @FilledDate = getdate(),
       @Status = 'O';

-- NOTE: Assumes that you inserted a Customer record with
CustomerName='Fictitious Customer' in the pre-test script.
SELECT @CustomerID = [CustomerID] FROM [Sales].[Customer] WHERE
[CustomerName] = @CustomerName;

-- fill an order for that customer
EXECUTE @RC = [Sales].[uspShowOrderDetails] @CustomerID;

SELECT @RC AS RC;
```

3. In the **Test Conditions** pane, click the Inconclusive test condition, and click **Delete Test Condition**.

4. In the **Test Conditions** pane, click **Expected Schema** in the list, and then click **Add Test Condition**.

5. In the **Properties** window, in the **Configuration** property, click the browse button ('**...**').

6. In the **Configuration for expectedSchemaCondition1** dialog box, specify a connection to your database. For example, if you used the default deployment location, which is LocalDB, you would click **New Connection** specify **(LocalDB)\Projects**. Then, choose the name of the database.

7. Click **Retrieve**. (If necessary, click **Retrieve** until you see data.)

   The Transact-SQL body of your unit test is executed, and the resulting schema appears in the dialog box. Because the pre-test code was not executed, no data is returned. As you are only verifying the schema and not the data, this is fine.

8. Click **OK**.

   The expected schema is stored with the test condition.

9. In the navigation bar of the SQL Server Unit Test Designer, click **Sales_uspShowOrderDetailsTest**, and make sure that **Pre-Test** is highlighted in the adjacent list. After you perform this step, you can specify statements that put your data into the state that is required to execute your test. For this example, you must create the Customer record before you can place an order.

10. Click **Click here to create** to create a pre-test script.

11. Update the Transact-SQL statements in the Transact-SQL editor to match the following statements:

```
/*
Add Transact-SQL statements here to run before the test script is
run.
*/
BEGIN TRANSACTION

-- Add a customer for this test with the name 'FictitiousCus-
tomer'
DECLARE @NewCustomerID AS INT, @RC AS INT, @CustomerName AS
NVARCHAR (40);
```

```
SELECT @RC = 0,
       @NewCustomerID = 0,
       @CustomerName = N'Fictitious Customer';

IF NOT EXISTS(SELECT * FROM [Sales].[Customer] WHERE CustomerName
= @CustomerName)
BEGIN
EXECUTE @NewCustomerID = [Sales].[uspNewCustomer] @CustomerName;
END

DECLARE @CustomerID AS INT, @Amount AS INT, @OrderDate AS
DATETIME, @Status AS CHAR (1);

SELECT @RC = 0,
       @CustomerID = 0,
       @CustomerName = N'Fictitious Customer',
       @OrderDate = getdate(),
       @Status = 'O';

-- NOTE: Assumes that you inserted a Customer record with
CustomerName='Fictitious Customer' in the pre-test script.
SELECT @CustomerID = [CustomerID] FROM [Sales].[Customer] WHERE
[CustomerName] = @CustomerName;

-- delete any old records in the Orders table and clear out the
YTD Sales/Orders fields
DELETE from [Sales].[Orders] WHERE [CustomerID] = @CustomerID;
UPDATE [Sales].[Customer] SET YTDOrders = 0, YTDSales = 0 WHERE
[CustomerID] = @CustomerID;

-- place 3 orders for that customer
EXECUTE @RC = [Sales].[uspPlaceNewOrder] @CustomerID, 100, @Or-
derDate, @Status;
EXECUTE @RC = [Sales].[uspPlaceNewOrder] @CustomerID, 50, @Order-
Date, @Status;
EXECUTE @RC = [Sales].[uspPlaceNewOrder] @CustomerID, 5, @Order-
Date, @Status;

COMMIT TRANSACTION
```

12. In the navigation bar of the SQL Server Unit Test Designer, click **Sales_uspShowOrderDetailsTest**, and click **Test** in the adjacent list.

    You must do this because you want to apply the checksum condition to the test, not to the pre-test.

13. In the **Test Conditions** pane, click **Data Checksum** in the list, and then click **Add Test Condition**.

14. In the **Properties** window, in the **Configuration** property, click the browse button ('**...**').

15. In the **Configuration for checksumCondition1** dialog box, specify a connection to your database.

16. Replace the Transact-SQL in the dialog box (under the **Edit Connection** button) with the following code:

```
BEGIN TRANSACTION

-- Add a customer for this test with the name 'CustomerB'
DECLARE @NewCustomerID AS INT, @RC AS INT, @CustomerName AS
NVARCHAR (40);

SELECT @RC = 0,
       @NewCustomerID = 0,
       @CustomerName = N'Fictitious Customer';

IF NOT EXISTS(SELECT * FROM [Sales].[Customer] WHERE CustomerName
= @CustomerName)
BEGIN
EXECUTE @NewCustomerID = [Sales].[uspNewCustomer] @CustomerName;
END

DECLARE @CustomerID AS INT, @Amount AS INT, @OrderDate AS
DATETIME, @Status AS CHAR (1);

SELECT @RC = 0,
       @CustomerID = 0,
       @CustomerName = N'Fictitious Customer',
       @OrderDate = getdate(),
       @Status = 'O';

-- NOTE: Assumes that you inserted a Customer record with
CustomerName='Fictitious Customer' in the pre-test script.
SELECT @CustomerID = [CustomerID] FROM [Sales].[Customer] WHERE
[CustomerName] = @CustomerName;

-- delete any old records in the Orders table and clear out the
YTD Sales/Orders fields
DELETE from [Sales].[Orders] WHERE [CustomerID] = @CustomerID;
UPDATE [Sales].[Customer] SET YTDOrders = 0, YTDSales = 0 WHERE
[CustomerID] = @CustomerID;

-- place 3 orders for that customer
EXECUTE @RC = [Sales].[uspPlaceNewOrder] @CustomerID, 100, @Or-
derDate, @Status;
EXECUTE @RC = [Sales].[uspPlaceNewOrder] @CustomerID, 50, @Order-
Date, @Status;
EXECUTE @RC = [Sales].[uspPlaceNewOrder] @CustomerID, 5, @Order-
Date, @Status;

COMMIT TRANSACTION
```

```
-- ssNoVersion unit test for Sales.uspFillOrder
DECLARE @FilledDate AS DATETIME;
DECLARE @OrderID AS INT;

SELECT @RC = 0,
       @CustomerID = 0,
       @OrderID = 0,
       @CustomerName = N'Fictitious Customer',
       @Amount = 100,
       @FilledDate = getdate(),
       @Status = 'O';

-- NOTE: Assumes that you inserted a Customer record with
CustomerName='Fictitious Customer' in the pre-test script.
SELECT @CustomerID = [CustomerID] FROM [Sales].[Customer] WHERE
[CustomerName] = @CustomerName;

-- fill an order for that customer
EXECUTE @RC = [Sales].[uspShowOrderDetails] @CustomerID;

SELECT @RC AS RC;
```

This code combines the Transact-SQL code from the pre-test with the Transact-SQL from the test itself. You need both to return the same results that the test will return when you run it.

17. Click **Retrieve**. (If necessary, click **Retrieve** until you see data.)

    The Transact-SQL that you specified is executed, and a checksum is calculated for the returned data.

18. Click **OK**.

    The calculated checksum is stored with the test condition. The expected checksum appears in the Value column of the Data Checksum test condition.

19. On the **File** menu, click **Save All**.

    At this point, you are ready to run your tests.

Run SQL Server Unit Tests

To run the SQL Server unit tests

1. On the **Test** menu, point to **Windows**, and then click **Test View** in Visual Studio 2010 or **Test Explorer** in Visual Studio 2012.

2. In the **Test View** window (Visual Studio 2010), click **Refresh** on the toolbar to update the list of tests. To see the list of tests in **Test Explorer** (Visual Studio 2012), build the solution.

   The **Test View** or **Test Explorer** window lists the tests that you created earlier in this walkthrough and to which you added Transact-SQL statements and test conditions. The test that is named TestMethod1 is empty and is not used in this walkthrough.

3. Right-click **Sales_uspNewCustomerTest**, and click **Run Selection**.

   Visual Studio uses the privileged context that you specified to connect to the database and apply the data generation plan. Visual Studio then switches to the execution context before it runs the Transact-SQL script in the test. Finally, Visual Studio evaluates the results of the Transact-SQL script against those that you specified in the test condition, and a result of either pass or fail appears in the **Test Results** window.

4. View the result in the **Test Results** window.

   The test passes, which means that **SELECT** statement returns one row when it is run.

5. Repeat step 3 for the Sales_uspPlaceNewOrderTest, Sales_uspFillOrderTest, and Sales_uspShowOrderDetailsTest tests. Results should be as follows:

   Expand table

Test	Expected Result
Sales_uspPlaceNewOrderTest	Pass
Sales_uspShowOrderDetailsTest	Pass
Sales_uspFillOrderTest	Fails with the following error: "ScalarValueCondition Condition (scalarValueCondition2) Failed: ResultSet 1 Row 1 Column 1: values do not match, actual '-100' expected '100'." This error occurs because the definition of the stored procedure contains a minor error.

   Next, you will correct the error and re-run your test.

To correct the error in Sales.uspFillOrder

1. In the **SQL Server Object Explorer** Projects node for your database, double-click the **uspFillOrder** stored procedure to open its definition in the Transact-SQL editor.

2. In the definition, find the following Transact-SQL statement:

```
UPDATE [Sales].[Customer]
   SET
   YTDSales = YTDSales - @Delta
    WHERE [CustomerID] = @CustomerID
```

3. Change the SET clause in the statement to match the following statement:

```
UPDATE [Sales].[Customer]
   SET
   YTDSales = YTDSales + @Delta
    WHERE [CustomerID] = @CustomerID
```

4. On the **File** menu, click **Save uspFillOrder.sql**.

5. In the **Test View**, right-click **Sales_uspFillOrderTest**, and click **Run Selection**.

   The test passes.

Add a Negative Unit Test

You might create a negative test to verify that a test fails when it should fail. For example, if you try to cancel an order that was already filled, that test should fail. In this part of the walkthrough, you create a negative unit test for the Sales.uspCancelOrder stored procedure.

To create and verify a negative test, you must perform the following tasks:

- Update the stored procedure to test for failure conditions
- Define a new unit test
- Modify the code for the unit test to indicate that is expected to fail
- Run the unit test

To update the stored procedure

1. In the **SQL Server Object Explorer** Projects node for the SimpleUnitTestDB database, expand the Programmability node, expand the Stored Procedures node,

and double-click uspCancelOrder.

2. In the Transact-SQL editor, update the procedure definition to match the following code:

```
CREATE PROCEDURE [Sales].[uspCancelOrder]
@OrderID INT
AS
BEGIN
    DECLARE @Delta INT, @CustomerID INT, @PriorStatus CHAR(1)
    BEGIN TRANSACTION
        BEGIN TRY
            IF (NOT EXISTS(SELECT [CustomerID] from [Sales].[Or-
ders] WHERE [OrderID] = @OrderID))
            BEGIN
                -- Specify WITH LOG option so that the error is
                -- written to the application log.
                RAISERROR( 'That order does not exist.', --
Message text
                           16, -- severity
                            1 -- state
                        ) WITH LOG;
            END

            SELECT @Delta = [Amount], @CustomerID = [CustomerID],
@PriorStatus = [Status]
             FROM [Sales].[Orders] WHERE [OrderID] = @OrderID

            IF @PriorStatus <> 'O'
            BEGIN
                -- Specify WITH LOG option so that the error is
                -- written to the application log.
                RAISERROR ( 'You can only cancel open orders.', -
- Message text
                            16, -- Severity
                            1 -- State
                            ) WITH LOG;
            END
            ELSE
            BEGIN
                -- If we make it to here, then we can cancel the
order. Update the status to 'X' first...
                UPDATE [Sales].[Orders]
                   SET [Status] = 'X'
                WHERE [OrderID] = @OrderID
                -- and then remove the amount from the YTDOrders
for the customer
                UPDATE [Sales].[Customer]
                       SET
                           YTDOrders = YTDOrders - @Delta
                WHERE [CustomerID] = @CustomerID
                COMMIT TRANSACTION
                RETURN 1; -- indicate success
            END
        END TRY
        BEGIN CATCH
            DECLARE @ErrorMessage NVARCHAR(4000);
            DECLARE @ErrorSeverity INT;
            DECLARE @ErrorState INT;

            SELECT @ErrorMessage = ERROR_MESSAGE(),
                   @ErrorSeverity = ERROR_SEVERITY(),
                   @ErrorState = ERROR_STATE();

            ROLLBACK TRANSACTION
            -- Use RAISERROR inside the CATCH block to return
            -- error information about the original error that
            -- caused execution to jump to the CATCH block.
            RAISERROR (@ErrorMessage, -- Mesasge text
                       @ErrorSeverity, -- Severity
                       @ErrorState -- State
                      );
            RETURN 0; -- indicate failure
        END CATCH;
END
```

3. On the **File** menu, click **Save uspCancelOrder.sql**.

4. Press F5 to deploy **SimpleUnitTestDB**.

   You deploy the updates to the uspCancelOrder stored procedure. You changed no other objects, so only that stored procedure is updated.

   Next you define the associated unit test for this procedure.

To write the SQL Server unit test for uspCancelOrder

1. In the navigation bar of the SQL Server Unit Test Designer, click **Sales_uspCancelOrderTest**, and make sure that **Test** is highlighted in the adjacent list.

   After you perform this step, you can create the test script for the test action in the unit test.

2. Update the Transact-SQL statements in the Transact-SQL editor to match the following statements:

```
-- ssNoVersion unit test for Sales.uspFillOrder
DECLARE @RC AS INT, @CustomerID AS INT, @Amount AS INT, @Filled-
Date AS DATETIME, @Status AS CHAR (1);
DECLARE @CustomerName AS NVARCHAR(40), @OrderID AS INT;

SELECT @RC = 0,
       @CustomerID = 0,
       @OrderID = 0,
       @CustomerName = N'Fictitious Customer',
       @Amount = 100,
       @FilledDate = getdate(),
       @Status = 'O';

-- NOTE: Assumes that you inserted a Customer record with
CustomerName='Fictitious Customer' in the pre-test script.
SELECT @CustomerID = [CustomerID] FROM [Sales].[Customer] WHERE
[CustomerName] = @CustomerName;
-- Get the most recently added order.
SELECT @OrderID = MAX([OrderID]) FROM [Sales].[Orders] WHERE
[CustomerID] = @CustomerID;

-- try to cancel an order for that customer that has already been
filled
EXECUTE @RC = [Sales].[uspCancelOrder] @OrderID;

SELECT @RC AS RC;
```

3. In the **Test Conditions** pane, click the Inconclusive test condition, and click the **Delete Test Condition** icon.

4. In the **Test Conditions** pane, click **Scalar Value** in the list, and then click the **Add Test Condition** icon.

5. In the **Properties** window, set the **Expected Value** property to 0.

6. In the navigation bar of the SQL Server Unit Test Designer, click **Sales_uspCancelOrderTest**, and make sure that **Pre-Test** is highlighted in the adjacent list. After you perform this step, you can specify statements that put your data into the state that is required to execute your test. For this example, you must create the Customer record before you can place an order.

7. Click **Click here to create** to create a pre-test script.

8. Update the Transact-SQL statements in the Transact-SQL editor to match the following statements:

```sql
/*
Add Transact-SQL statements here to run before the test script is
run.
*/
BEGIN TRANSACTION

-- Add a customer for this test with the name 'CustomerB'
DECLARE @NewCustomerID AS INT, @RC AS INT, @CustomerName AS
NVARCHAR (40);

SELECT @RC = 0,
       @NewCustomerID = 0,
       @CustomerName = N'Fictitious Customer';

IF NOT EXISTS(SELECT * FROM [Sales].[Customer] WHERE CustomerName
= @CustomerName)
BEGIN
EXECUTE @NewCustomerID = [Sales].[uspNewCustomer] @CustomerName;
END

DECLARE @CustomerID AS INT, @Amount AS INT, @OrderDate AS
DATETIME, @FilledDate AS DATETIME, @Status AS CHAR (1), @OrderID
AS INT;

SELECT @RC = 0,
       @CustomerID = 0,
   @OrderID = 0,
       @CustomerName = N'Fictitious Customer',
       @Amount = 100,
       @OrderDate = getdate(),
   @FilledDate = getdate(),
       @Status = 'O';

-- NOTE: Assumes that you inserted a Customer record with
CustomerName='Fictitious Customer' in the pre-test script.
SELECT @CustomerID = [CustomerID] FROM [Sales].[Customer] WHERE
[CustomerName] = @CustomerName;

-- delete any old records in the Orders table and clear out the
YTD Sales/Orders fields
DELETE from [Sales].[Orders] WHERE [CustomerID] = @CustomerID;
UPDATE [Sales].[Customer] SET YTDOrders = 0, YTDSales = 0 WHERE
[CustomerID] = @CustomerID;

-- place an order for that customer
EXECUTE @OrderID = [Sales].[uspPlaceNewOrder] @CustomerID,
@Amount, @OrderDate, @Status;

-- fill the order for that customer
EXECUTE @RC = [Sales].[uspFillOrder] @OrderID, @FilledDate;

COMMIT TRANSACTION
```

9. On the **File** menu, click **Save All**.

   At this point, you are ready to run your tests.

To run the SQL Server unit tests

1. In **Test View**, right-click **Sales_uspCancelOrderTest**, and click **Run Selection**.

2. View the result in the **Test Results** window.

   The test fails and the following error appears:

   Test method TestProject1.SqlServerUnitTests1.Sales_uspCancelOrderTest threw exception: System.Data.SqlClient.SqlException: You can only cancel open orders.

   Next, you modify the code to indicate that the exception is expected.

To modify the code for the unit test

1. In **Solution Explorer**, expand **TestProject1**, right-click **SqlServerUnitTests1.cs**, and click **View Code**.

2. In the code editor, navigate to the Sales_uspCancelOrderTest method. Modify the attributes of the method to match the following code:

   ```
   [TestMethod(), ExpectedSqlException(Severity=16,
   MatchFirstError=false, State=1)]
   public void Sales_uspCancelOrderTest()
   ```

   You specify that you expect to see a specific exception. You can optionally specify a specific error number. If you do not add this attribute, the unit test will fail, and a message appears in the Test Results window

   > ⓘ **Important**
   >
   > Currently, Visual Studio 2012 does not support the ExpectedSqlException attribute. For information to work around this, see Unable to Run "Expected Failure" Database Unit Test.

3. On the File menu, click Save SqlServerUnitTests1.cs.

   Next, you re-run your unit test to verify that it fails as expected.

To re-run the SQL Server unit tests

1. In **Test View**, right-click **Sales_uspCancelOrderTest**, and click **Run Selection**.

2. View the result in the **Test Results** window.

   The test passes, which means that the procedure failed when it was supposed to fail.

Next Steps

In a typical project, you would define additional unit tests to verify that all critical database objects work correctly. When the set of tests is complete, you would check those tests into version control to share them with the team.

After you establish a baseline, you can create and modify database objects and then create associated tests to verify whether a change will break expected behavior.

See Also

Creating and Defining SQL Server Unit Tests
Verifying Database Code by Using SQL Server Unit Tests
How to: Create an Empty SQL Server Unit Test
How to: Configure SQL Server Unit Test Execution