Designing a Test in SentryOne Test

Last Modified on 13 November 2019

Designing a Test

As you begin to work with SentryOne Test, a natural first question is "How do I design a test?" This article reviews designing tests with SentryOne Test, and although it uses an SSIS package as an example, the design strategy is applicable to all tests.

Note: The steps to design a test with SentryOne Test are similar regardless of what you're testing.

1. Understand the object being tested.
2. Identify the assertions that legitimize the object being tested.
3. Define the actions that need to be taken to do the tests.
4. List the assets needed for the actions to work.

Test Component	Description
Object being tested	With SentryOne Test, the object being tested can be a multitude of things. SSIS packages, data, database objects such as stored procedures and functions, SSRS reports, SSAS cubes and dimensions, and more.
Assertions	The assertions can be thought of as the way in which we prove the object works as designed. In human readable terms, a test might assert “If the row count from a query matches an entered number, the test passes”.
Actions	Actions are the things a test will need to do in order to perform an assertion. For example, running queries, loading packages, running reports, etc.
Assets	Assets are the things actions need in order to run. Connection strings, queries, and package references are all

Test Component	Description
	examples of assets.

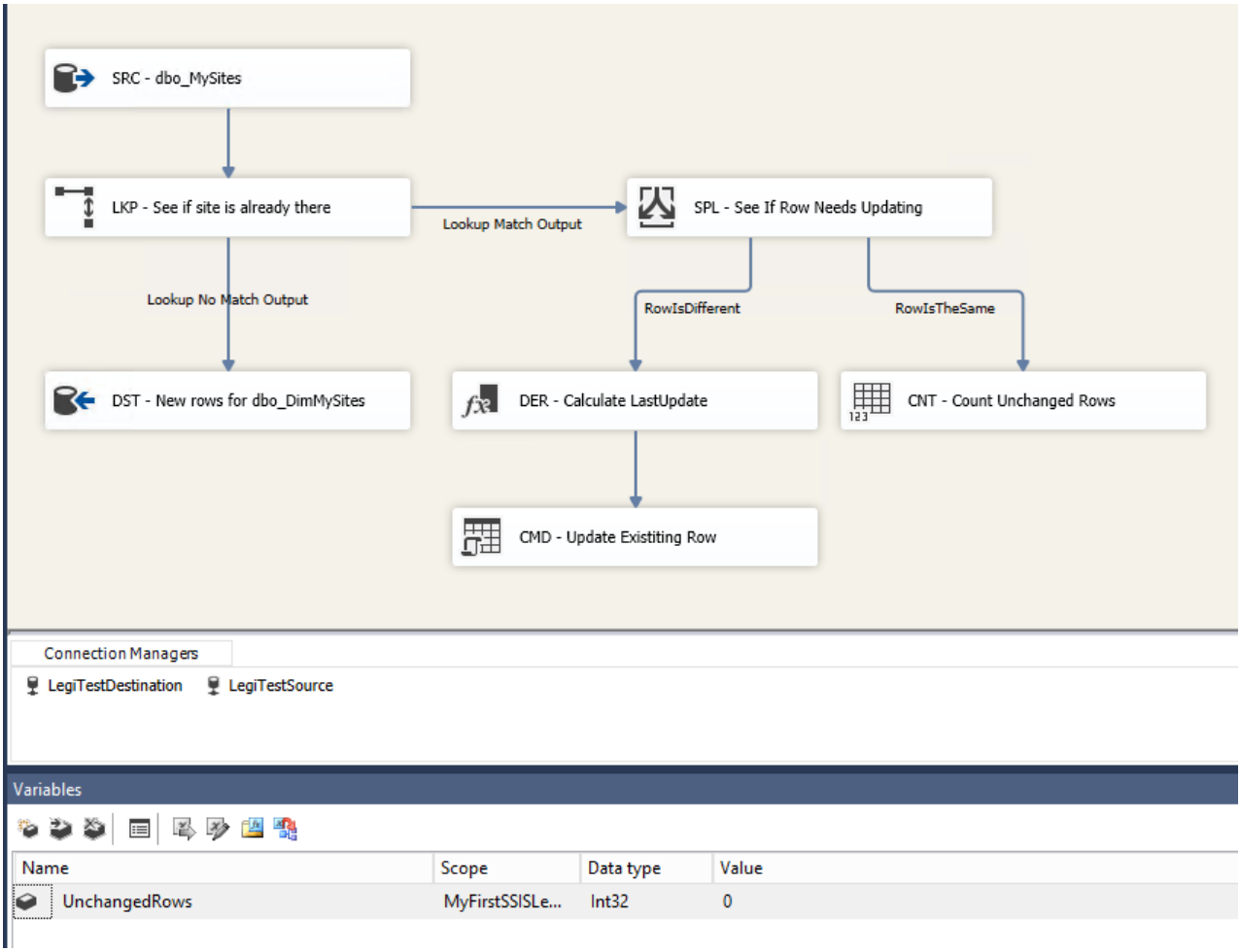
For a more in depth discussion on these key concepts, please see the Visual Studio Extension Overview.

The Package Being Tested

The first step is to understand the object being tested. The package being used in this example is simple, but performs a common job in the SSIS world.

- This package takes data from a source table
- Checks if that data is already in the target database based on the Primary Key column.
    - If the data is not present, it inserts it into the target table.
    - If that data is present, but is different from what is already in the target, it updates the target.
    - Otherwise, if the source and target are the same, it increment the unchanged rows variable, but otherwise take no action.

Here is a screen shot of the data flow task inside the package we are working with:



As mentioned, this package uses two databases, one a source and one a target, each with one table. In the first database, **LegiTestSource**, we'll create one table.

```
CREATE TABLE [dbo].[MySites](
       [PrimaryKey] [int] NOT NULL,
       [Name] [nvarchar](250) NOT NULL,
       [URL] [nvarchar](250) NULL
) ON [PRIMARY]
```

Here is an example of the kind of data that might go in it.

```
INSERT INTO [dbo].[MySites]
 ([PrimaryKey], [Name], [URL])
VALUES
  (1, 'SentryOne', 'http://sentryone.com'),
  (2, 'LegiTest', 'http://legitest.com'),
  (3, 'SQL skills', 'http://sqlskills.com');
```

For our target database, **LegiTestTarget**, we create a fake **dimension** table.

```
CREATE TABLE [dbo].[DimMySites](
       [PrimaryKey] [int] NOT NULL,
       [Name] [nvarchar](250) NOT NULL,
       [URL] [nvarchar](250) NULL,
       [LastUpdate] [datetime] NOT NULL DEFAULT (getdate())
) ON [PRIMARY]
```

Of course, this isn't a real dimension table in any sense of the word, in this example we use the Dim name for illustration purposes only.

The package needs two connection managers, one for each database. For this sample they were made local to the package.

> ℹ **Note:** The Connection managers were renamed to remove the **local** denotation that SSIS automatically places on the front. The package also has one variable, which tracks the number of unchanged rows.

The package we create does a lookup; if the row doesn't exist, it inserts the row into **DimMySites**. If it does exist, a flow goes to the conditional split. In the conditional split the name and URL are compared to see if anything has changed.

```
(Name != DimName) || (URL != DimURL)
```

The purpose of the derived column transform is to calculate the **LastUpdate** date for the target table. The variable is **LastUpdate**, and the expression is simply **(DT_DATE)GETDATE()**.

The contents of the OLEDB CMD destination should be relatively obvious. The command is a simple update statement.

```
UPDATE [dbo].[DimMySites]   SET [Name] = ?, [URL] = ?, [LastUpdate] = ? WHER
E [PrimaryKey] = ?
```

> ℹ **Note:** This is not the optimal way to design a good data warehouse package. Normally the OLEDB Command would not be used, due to its slowness. However this works well for this foray into SentryOne Test.

Now that we understand the object being tested, in this example the package, how would we go about testing it? What are the types of assertions we could run against this package to ensure it works as designed?

Assertions

A first assertion is to ensure that the number of rows in both tables is the same after the package executes . Another assertion would ensure the data in both tables is the same after the package executes (excluding the LastUpdate column which only exists in the target).

What else should we test?

Looking at the package, there is logic around existing rows. We want to ensure data is updated properly, but that is covered in the previous test on matching the source to the target. Another part of the logic routes unchanged rows to a counter. A test around this could actually consist of two assertions :

- The first should validate that the row did not change between the time the package started and the time it ended.
- The second should check the row counter.

To make this testable, we need some special data loaded into the target table.

```
 INSERT INTO [dbo].[DimMySites]
 ([PrimaryKey], [Name], [URL])
VALUES
  (1, 'SentryOne', 'http://sentryone.com'),
  (3, 'sqlskills', 'http://sqlskills.com');
```

Referring back to the data loaded into the source, the row with **Primary Key 1** is the same data and should not change during the update process. The row with **Primary Key 3** will be updated, but that's tested and validated when the target table is compared to the source table. As a final test, we want our package to execute in under a set amount of time. We want to verify the execute time to ensure it meets the defined parameters.

☑ Assertions List Summary

1. Assert that the source and target tables have the same number of rows after the package executes. (For the test database we have set up, this is three rows.)
2. Assert that the data, excluding the **LastUpdate** column (that doesn't exist in the source), matches between the source and target.
3. Assert that the row that should not be changed (P**rimary Key 1**) is not altered, based on the **LastUpdate** column.
4. Assert that the **UnchangedRows** variable containing the unchanged rows row count has a value of one after the package executes.
5. Assert that the package executed in under a set amount of time.

At this point, we have now identified the assertions that are needed within each test. To do these tests, what actions do we need to take before we can run assertions within our tests?

Actions

Prior to running individual tests, there are actions that must be set. This step would likely be done at the **Group Initialization** area of SentryOne Test.

First, we need to set the source and target databases to a known state. Often the source database is already in a known state, which is the case here, so we won't have to do anything within the test. We want to ensure the target database is reset correctly prior to the tests being done.

After the target database is set up, we must determine how to check whether the row that's used in our unchanged row has been changed or not. We need to execute a query to get the **LastUpdate** date-time for that row before the test executes.

Packages can exist in many different places. As a next step we must get a reference to the package itself. Once we have that package loaded in memory, we will run it.

☑ Actions List Summary

1. Reset the target table to a known state.
2. Execute a query to get the **LastUpdate** data for the row that shouldn't change.
3. Load the package into memory.
4. Execute the package.

Now that the group actions are identified, we know that each test itself needs to run its own series of actions before the assertion in that test can take place. Individual tests within SentryOne Test follow a typical pattern:

1. Execute one or more actions to determine the state of data after the object executes.
2. Run one or more assertions that use the output of the actions to determine if the assertion is true or false.

Specific to this test, the actions we take for the first four tests execute a query action against the database. The final test performs a **get properties action** to get the execution time.

Assets

With the bulk of the work completed, the final task is to identify the assets needed for the actions to work. The SentryOne Test being designed here works with two databases, so a connection string for each database is manifested as connection assets.

In the actions area it was mentioned that there are actions that determine the state of the data. These are queries, and thus a series of query assets must be generated.

One of the assertions we listed compares data between the source and target. To do this, SentryOne Test must know how to map the columns between the two systems. To do that we use an asset called a comparison manifest.

Finally, to load a package we need to tell SentryOne Test where it is. That is done

through a package reference asset.

☑ Asset List Summary

1. Connections to the source and target database.
2. Queries to get row counts for both the source and target tables.
3. Queries to get data for both the source and target tables.
4. Query to get the **LastUpdate** column.
5. A comparison manifest that's used to compare the row and source data.
6. A reference to the **DimMySites** package.

Summary

As you can see, this is a logical progression you can use to design any SentryOne Test test. First, before the design of any test begins you must understand what is being tested. Second, identify the ways in which you can validate that the object being tested works as designed. After this step it should start to become clear which actions are needed to perform the validation. As the fourth and final step, identify the assets that are needed to perform the actions and asserts.

☑ Test Design Summary

1. Understand the object being tested.
2. Identify the assertions that legitimize the object being tested.
3. Define the actions that need to be taken to do the tests.
4. List the assets needed for the actions to work.

Sidebar: The Real World

In an optimal world, it would be nice to have everything identified before we did any coding, and laid out so neatly, as we were able to do in this article. We all know that

the real world isn't so neat, however, the steps listed can be performed on an iterative basis. Instead of identifying everything up front, this could be done on a test-by-test basis.

☑ In-Practice Summary

1. Understand the object being tested.
2. Identify the assertion(s) for an individual test.
3. Define the actions needed for that test to perform its assertion.
4. Create the assets for that particular test, or identify and reuse any assets that may already exist.
5. Return to step one for creation of the next test.

What's Next?

In this tutorial we walked through the design of a SentryOne Test test, using an SSIS package as illustration. In the Building a Test article we implement this design by building a complete SentryOne Test test around it.