Unit test SQL PL routines by using db2unit framework in DB2 LUW

An introduction to xUnit framework for the SQL PL programming language

Andres Gomez Casanova (angoca@gmail.com)
DB2 DBA and Software Engineer
Consultant

09 January 2015

SQL PL routines stored in DB2® databases contain business logic that can be used by all applications that access the database. However, these routines are not always completely tested. Until now, no standard procedure existed to perform automatic tests against them. With the incursion of test-driven development from the eXtreme Programming paradigm, it is important to write test cases that check all possible conditions. Doing so even before starting development ensures the quality of the software. db2unit is a framework that helps you follow these guidelines by writing better code and creating more reliable applications. Learn how to use the innovative framework that, automating unit tests for the SQL PL routines.

Overview

Software development is not only about writing code but it is part of a lifecycle that includes different activities with the common purpose of building mature applications. This lifecycle has a testing phase that creates metrics after executing tests at different levels. These metrics provide information about the quality of the code, which helps ensure that the code does what it is supposed to do. Unit tests are part of this testing phase, and their objective is to perform checks at the component level. Almost every modern programming language has a unit-testing tool, and this tutorial presents a unit-testing framework for the SQL PL language.

A unit-testing framework executes a battery of tests and retrieves a report of their execution to indicate which tests have passed, failed, or hit any errors. Modern programming methodologies such as eXtreme Programming (XP) have pushed unit tests to the limit by creating a practice called test-driven development (TDD), which heavily relies on unit-testing frameworks. This programming methodology emphasizes testing every component of an application and writing tests even before starting to code the application itself. This approach guides you to write better code and makes sure it works as designed.
Normally, tests for routines written in SQL PL are manually executed. The developer calls the stored procedure or user-defined function (UDF) with different parameters, and verifies the results. The result can be a returned value or modification of data in the database. If the behavior is not as expected, the developer changes the code, redeployes it, and verifies the results again until the routine seems to be correct. But this approach brings many problems because performed tests are not repeatable. They are time-consuming, and the probability to perform a regression is high.

There is a standard architecture for unit-testing frameworks called xUnit, which emerged from sUnit (for Smalltalk) that was the base of the well-known jUnit framework (for Java™ technology). These, like other unit-testing frameworks, are useful for testing software developed in the same programming language. However, they are not well suited to test external code, such as SQL routines because they do not exploit the capabilities of the database. For this reason, db2unit is the best suitable framework to test SQL PL code in DB2.

db2unit is a framework completely written in SQL PL for DB2 LUW. This framework follows the basic component structure of xUnit and takes its power from recent DB2 features such as autonomous transactions and modules. The project is hosted as a public project on GitHub. It is licensed under the terms of the GPL V3 license.

**Downloading and installing db2unit**

At the time this tutorial was written, the first stable version of the framework was released: db2unit v1. It is available for download in the section for Releases on GitHub.
Both types of available archive files (.tar.gz and .zip) contain the necessary files to install the framework on any supported platform (Windows or Linux/UNIX/Mac OS X). The following items are necessary to install db2unit:

- DB2 9.7 LUW or later (see Resources).
- A database for testing purposes, since it is better to have a dedicated database with mock data. DB2 Express-C edition is ideal for this kind of non-production tests.
- dbadm authority on the database where the framework will be installed or a database administrator kind enough to create the schema and grant the necessary rights on the respective objects.
- log4db2, an event logging utility for DB2, must be installed in the database (see Resources).

After the database has been prepared, the installation can be performed. The framework includes different types of installers for each of the DB2-supported platforms:

- Shell scripts for Linux, UNIX, and Mac OS X. It is important to have the DB2 environment loaded (for example: . ${DB2INSTANCE}/sqllib/db2profile) and be connected to a database before the execution of the installer. To execute the install script, as in Figure 2, it must be called with the source command represented by a dot: . ./install.
Two types of installers for Windows: one for `cmd.exe (db2cmd.exe)` and another for PowerShell. In PowerShell, it is necessary to load the DB2 environment with `set-item -path env:DB2CLP -value "**$$**"`.

A platform-independent way to install this framework by using CLP*Plus. However, this installer does not perform all environment checks as other installers do.

Finally, db2unit can also installed manually by executing each script sequentially. For example, by opening each file in the IBM Data Studio client and executing them one by one.

Figure 2. Installing db2unit on Linux

![Figure 2. Installing db2unit on Linux](image)

Figure 3. Installation completed on Windows (db2cmd)

![Figure 3. Installation completed on Windows (db2cmd)](image)

If the installation is successful, a message (shown in Figure 3) will be displayed, indicating the name of the database where the framework was installed, its version, and the schema used. If any errors occurred during installation, a descriptive message is shown, which includes a link to the wiki on GitHub. The wiki also provides information on how to fix installation errors (see Resources).

db2unit uses a different schema for every single version released. The schema reflects the version of the utility that enables you to run multiple versions of the same framework in a database (see Serge Rielau's tutorial in Resources). For the first version, the schema name used is `db2unit_1`. 
All functions and stored procedures of this framework are part of a public module called db2unit, which is stored in the previously cited schema.

**Test definition**

The xUnit architecture defines different components for a unit-testing framework, but this architecture is focused for object-oriented languages. Because SQL PL is a procedural language, this architecture was adapted to DB2 by using a set of tables, schemas, and stored procedures inside a module.

**Figure 4. Relationships between different components of the xUnit architecture**

You have to implement the following components of the xUnit architecture:

- **The test case** is the most basic component in the xUnit architecture. It holds the interactions with the tested database object and checks if the given predicates are true. In db2unit, a test case is represented by a stored procedure with the following rules:
  - Its name starts with the prefix `TEST_`.
  - The procedure does not have any parameters.
- **Test fixtures** change the execution environment before and after the execution of each or all test cases. The xUnit architecture defines four types of test fixtures; each can be defined by a stored procedure with a specific name and without parameters:
  - Before the execution of all test cases (Before all): `ONE_TIME_SETUP`
  - Before the execution of each test case (Before): `SETUP`
  - After the execution of each test case (After): `TEAR_DOWN`
  - After the execution of all test cases (After All): `ONE_TIME_TEAR_DOWN`
- **The test suite** aggregates several test cases and test fixtures, and it is defined as a schema in DB2. It can have any name if DB2 supports it as a name for a schema, and each schema is considered as a different test suite.
- **Assertions** are predicates that indicate if the predicate is true or false. They are represented by a set of stored procedures included in the framework. You can extend this set of assertions to check your own types of predicates. The available assertions are based on the basic DB2 data types: boolean, strings (char, varchar), numbers (smallint, integer, bigint), decimal (float, double), date/time (date, time, timestamp); they are described in the API section.
of the wiki (see Resources). In addition to these assertions, the framework has two more procedures: REGISTER_MESSAGE to associate a raised SQL exception with a message and FAIL that automatically considers a test case as failed.

In addition to the elements you have to define, an xUnit framework has other components that db2unit includes:

- The **test runner** is represented by a stored procedure called RUN_SUITE.
- When the test runner is executed, it generates a **test result** stored in a table called REPORT_TESTS, which is stored in the same schema as the test suite. The result of the execution of a test case has three types:
  - Passed: The execution did not hit any error and all predicates were true.
  - Failed: The execution was successful, but at least one predicate was false.
  - Error: There was a problem during the execution of the test case, and it raised a signal.
- All test suite executions are stored in the EXECUTION_REPORTS table, and each execution is identified by a EXECUTION_ID.

**Figure 5. Components of a test suite in db2unit**

In Figure 5, you can see the implementation of db2unit in DB2 in contrast to the xUnit architecture described in Figure 4.

A key element of this framework is the independence of transactions between the transaction that execute the test case and the one that writes the results. This feature is called autonomous transaction in DB2. The transaction executing the test case might test predicates and after that perform a rollback. If the autonomous transaction feature is not enabled, the performed process before the rollback is lost. With the autonomous transaction feature, the result of these assertions is saved in the database, and the test case can perform a complete rollback.

**Test suite development**

This section shows an example of a test suite about currency conversion. This example could run on a database with the following objects:
• A table with the exchange rate between different currencies and the U.S. dollar.
• A function that performs currency conversion by passing following parameters: source value, source currency, and target currency. It returns a target value in the target currency.

**Listing 1. Sample code to create objects to be tested**

```sql
CREATE TABLE US_DOLLAR_EXCHANGE_RATES (  
    CURRENCY_NAME CHAR(3),  
    RATE DECIMAL (10,2)  
) @

INSERT INTO US_DOLLAR_EXCHANGE_RATES VALUES  
('EUR', 0.79),  
('USD', 1),  
('COP', 2052.50) @

CREATE OR REPLACE FUNCTION CONVERT_CURRENCY (  
    SOURCE_VALUE DECIMAL,  
    SOURCE_CURRENCY CHAR(3),  
    TARGET_CURRENCY CHAR(3)  
) RETURNS DECIMAL (10,2)  
BEGIN  
    DECLARE RET_VALUE DECIMAL (10,2);  
    DECLARE US_DOLLARS DECIMAL (10,2);  

    SET US_DOLLARS = (SELECT SOURCE_VALUE / RATE  
        FROM US_DOLLAR_EXCHANGE_RATES  
        WHERE CURRENCY_NAME = SOURCE_CURRENCY);  
    SET RET_VALUE = (SELECT US_DOLLARS * RATE  
        FROM US_DOLLAR_EXCHANGE_RATES  
        WHERE CURRENCY_NAME = TARGET_CURRENCY);  

    RETURN RET_VALUE;  
END @
```

Based on the previous objects, you can create the following test cases:

• How many Colombian pesos can be bought with 10 U.S. dollars? COP 20 525.
• How many Euros can be bought with two U.S. dollars? EUR 1.58€.
• How many Euros can be bought with 10,000 Colombian pesos? EUR 3.84€.

A normal test case has the following three internal parts:

• Preparation of the environment — This part includes the declaration and initialization of variables, including cursors, handlers, and other type of variables. It can also include the definition of the expected value.
• Execution of operations — When the environment is ready, the object to test is harnessed and the returned value is saved in a variable (actual value).
• Calls to assertion procedures — Expected and actual values are passed to the corresponding assertion procedure that checks whether values are the same. Remember that the comparison is not the only type of assertion procedure that exists in the framework.

The following stored procedures contain test cases for the previous functions. They are created with the OR REPLACE option to be capable to recreate these objects.
Listing 2. Test 1: Checks the conversion of 10 USD to COP

```sql
CREATE OR REPLACE METHOD TEST_1_USD_TO_COP
BEGIN
  -- Prepares the environment.
  DECLARE EXPECTED_VALUE DECIMAL;
  DECLARE ACTUAL_VALUE DECIMAL;

  SET EXPECTED_VALUE = 20525;

  -- Executes operations.
  SET ACTUAL_VALUE = CONVERT_CURRENCY(10, 'USD', 'COP');

  -- Calls assertions.
  CALL DB2UNIT.ASSERT_DEC_EQUALS(EXPECTED_VALUE, ACTUAL_VALUE);
END @
```

Listing 3. Test 2: Checks the conversion of 2 USD to EUR

```sql
CREATE OR REPLACE METHOD TEST_2_USD_TO_EUR
BEGIN
  -- Prepares the environment.
  DECLARE EXPECTED_VALUE DECIMAL;
  DECLARE ACTUAL_VALUE DECIMAL;

  SET EXPECTED_VALUE = 1.58;

  -- Executes operations.
  -- Note: The currency name should be a 3-letters word.
  SET ACTUAL_VALUE = CONVERT_CURRENCY(2, 'USD', 'EURO');

  -- Calls assertions.
  CALL DB2UNIT.ASSERT_DEC_EQUALS(EXPECTED_VALUE, ACTUAL_VALUE);
END @
```

Listing 4. Test 3: Checks the conversion of 10,000 COP to EUR

```sql
CREATE OR REPLACE METHOD TEST_3_COP_TO_EUR
BEGIN
  -- Prepares the environment.
  DECLARE EXPECTED_VALUE DECIMAL;
  DECLARE ACTUAL_VALUE DECIMAL;

  -- Note: This is not the real expected value.
  SET EXPECTED_VALUE = 4;

  -- Executes operations.
  SET ACTUAL_VALUE = CONVERT_CURRENCY(10000, 'COP', 'EUR');

  -- Calls assertions.
  CALL DB2UNIT.ASSERT_DEC_EQUALS('Colombian Pesos to Euros', EXPECTED_VALUE, ACTUAL_VALUE);
END @
```

It is important to know the data type of the tested value to call the assertions before executing the procedure in the database. The procedure called to check the predicate must match the data type; otherwise, DB2 raises an error.

All assertion procedures provided by the framework can receive a descriptive message as first parameter. This message is printed if the assertion fails. This is useful to identify the origin of the problem if many assertions are called inside the same test case.
Compilation and execution of the test suite

When the code of the test cases is finished, it can be executed to create the objects (stored procedures and its dependencies for tests) in the database. This process is performed in the same way any other stored procedure is created in DB2. This can be done from the CLP, in Data Studio, or any other tool that executes DDLs.

Now tests are created in the database, and it is necessary to execute them. The procedure that executes the test suite is `DB2UNIT.RUN_SUITE`. The framework provides two ways to execute a test suite: sequential, in which case the order of the test cases is alphabetical; and random, with a different order for the test cases in each execution. To define the execution order, the following procedure must be called: `CALL DB2UNIT.RANDOM_SORT(FALSE)`.

By default, the execution of a test suite includes the execution of all test cases defined in the schema. However, you can execute one specific test case by specifying its name when calling the test runner.

The order of a random execution of a test suite is registered in the database. This same random order can be reproduced for the same test suite by providing the `EXECUTION_ID` of the previous execution when calling the test runner:

- Execute a given test suite:
  - `CALL DB2UNIT.RUN_SUITE(TEST_SUITE_NAME)`
- Execute a specific test case of a test suite:
  - `CALL DB2UNIT.CALL_DB2UNIT.RUN_SUITE(TEST_SUITE_NAME, TEST_NAME => TEST_CASE_NAME)`
- Execute a test suite in the same order of a previous execution:
  - `CALL DB2UNIT.RUN_SUITE(TEST_SUITE_NAME, EXECUTION_ID)`

The `RUN_SUITE` procedure is the heart of db2unit. It performs many checks and calls another procedure to execute a test suite. You can find a sequence diagram in the Architecture section of the wiki that explains every step of this stored procedure (see Resources).

As cited in this section, the identifier for each execution is called `EXECUTION_ID`, which is a random number generated by the framework. With this unique number, you can identify a specific execution of a test suite and use it to retrieve the report of an execution or take the order of the test cases to re-execute the test suite.

At the end of the execution, a report is presented to you with the output of the assertions for each test case and a summary of the execution with the account of test cases that passed, failed, or hit errors.
Figure 6. Execution of tests in the current schema

You can see the report of the execution of the test suite. The test suite was developed to not be successful to show the types of results (one passed, one failed, and one hit an error). In the upper part, the first result set shows the output of tests. The first column indicates which stored procedure was executed, a test fixture (before all, after all) or a test case. The second column shows the final state of the execution, if the test passed, failed, or hit an error. The third one has the time it took to execute that test case. And the fourth one is the message; it might be the name of the stored procedure, a message from the assertion, or the message of a non-caught SQL exception. The last lines of that result set show the summary of the execution. This information is stored in the `REPORT_TEST` table in the same schema of the test suite.

The second result set is more generic, it shows which test suite was executed with its generated `EXECUTION_ID`, and the total time it took to execute everything. This information is stored in the `EXECUTION_REPORT` table of the db2unit schema.

**Extra features in db2unit**

Until now, this tutorial covered the main features of the framework, but it is important to know other extra capabilities db2unit provides:

- By default, the framework runs in autonomous transaction mode, which allows having different boundaries between the tested object and the framework. However, this option can be deactivated, and it is possible to integrate db2unit operations inside the same transaction boundaries of the object being tested:
Unit test SQL PL routines by using db2unit framework in DB2

• CALL DB2UNIT.SET_AUTONOMOUS(FALSE)

• The framework relies on a simple schema of tables.

Figure 7. ER diagram of db2unit

• If the execution of a test suite is cancelled, it is necessary to clean the environment before you re-run any test suite in the same DB2 session (db2bp). It is necessary because the framework uses many session variables.
  • CALL DB2UNIT.CLEAN()

• Each time a test suite is executed, a corresponding lock is created to prevent parallel executions of the same test suite. The lock is established when the execution starts and released when the report has been generated. If the execution of a test suite is canceled in between, the lock has to be released manually; otherwise, the framework processes this situation like if there is a current execution of the test suite. If multiple test suites were canceled simultaneously, all locks in the database can be released at the same time.
  • CALL DB2UNIT.RELEASE_LOCK('MY_SUITE')
  • CALL DB2UNIT.RELEASE_LOCKS()

• The framework includes a set of assertions for the main data types; however, you can create your own assertions and integrate them with the framework.

• Finally, you can reset the entire framework configuration and bring db2unit to the state when it was installed (Vanilla configuration). It deletes all executions, test suite registrations, locks, and other objects.
  • CALL DB2UNIT.RESET_TABLES()

Advice for writing good test cases

Items to take into consideration when writing rapidly effective test cases:

• All test cases belong to a test suite. If test cases are created without a schema, they belong to the current schema, which usually corresponds to the current user name.
• Test suites can have just one test case without any test fixture.
• The test case name must not be surrounded by quotes and it must be always be capitalized. DB2 is case-sensitive with the name of objects; to prevent errors, you must always allow DB2 to store names in capitals.
• Test suites must have a high cohesion, and not all tests must be part of a single test suite. It is better to group test cases according to its nature; this reduces long-time executions and it is easier to detect errors.

• A good way to develop a new test suite is by following the "divide and conquer" strategy. Start by developing test fixtures and execute them in the database. The execution must be successful and without any errors before its continuation. When test fixtures are correct, add a procedure to the test suite and execute all again to validate the code. Repeat this process for each test case.

• When developing new test suites, they must be executed sequentially. When they are stable, the execution mode can be changed to random execution mode. This practice is important to reduce errors generated by lack of test-case isolation.

• The name of test cases must not be short nor long but concise and descriptive.

• All test cases must include at least one call to the following procedures:
  • Register message — For test cases whose process is successful without validating any predicate. This is typical when test cases check that no signals are raised.
  • Fail — For test cases that should not felt into a specific predicate conditions (IF, ELSEIF, ELSE; never arrive to the else block) and for this reason, the test case must fail.
  • Assertions — This set of procedures validates predicates.

• The framework has a procedure that registers a descriptive message of the test case's purpose. If a signal is raised, db2unit catches it, and the test report shows the message after the execution is finished. This procedure must be the first call in each test case.
  
  CALL DB2UNITREGISTER_MESSAGE (MESSAGE);

• Each time an assertion is called, a descriptive message might be passed. This eases to identify the problem in the test result.
  
  CALL DB2UNITASSERT_INT_EQUALS('Test A, comparing a with b', EXPECTED, ACTUAL);

• When calling an assertion, values passed as arguments must have been set before to local variables. It is not a good practice to put functions, arithmetic operations or concatenations in assertions as arguments. There might be errors in the given values and the debugging process becomes more difficult to identify the origin of the problem. Here are some examples of bad calls:
  
  CALL DB2UNITASSERT_INT_EQUALS(EXPECTED, MOD(5, 2))
  CALL DB2UNITASSERT_INT_EQUALS(EXPECTED, 'Text' || null)

• To have a trace of the execution, it is a good practice to write messages in a log table, in the db2diag.log, in a file or the standard output. log4db2 is an utility whose objective is to register log messages in different places. This utility is an installation dependency of db2unit.
  
  CALL LOGGERGET_LOGGER('myStoredProcedure', ID);
  CALL LOGGERINFO(ID, 'Informative message);

• log4db2 uses the concept of logger to write messages. You can configure loggers with a high verbosity level to start writing tests. After test cases are stable, you can reduce the logging level.
• It is a good idea to leave debugging messages in the code, instead of removing them when the code is finished and working. These messages could be used when the code is refactored or modified, and errors appear again.

**Conclusion**

It is important to have well-written routines stored in the database, such as stored procedures and user-defined functions, because they are common for all applications that access the database. db2unit is a unit-testing framework that helps you create more robust code by providing a way to automate the tests execution. This framework is based on the xUnit standard, and this fact reduces the adoption time by having a smooth learning curve for people who know jUnit or similar testing frameworks. The framework can be easily integrated into a continuous integration software and be part of any team that uses eXtreme Programming or DevOps. log4db2, along with db2unit, provides several features to create a more reliable code written in SQL PL.
Resources

- "Online Rolling Application Upgrades" (SQL Tips for DB2 LUW, February 20, 2012): explains how to have multiple versions of SQL PL routines in a DB2 database.
- "Test-Driven Development" (Wikipedia): Learn more about this programming strategy.
- "xUnit frameworks" (Wikipedia): Description of all components that are part of the xUnit architecture.
- "FAQs of db2unit": Search for known problems when installing the framework.
- "API of db2unit": List of available assertions to check predicates.
- "Sequence diagram": Learn about the interactions of db2unit.
- "db2unit" - Releases section at GitHub to download a stable version of this framework.
- "log4db2" - Releases section at GitHub to download a stable version of the logging utility.
- "DB2 Express-C" - Free to develop, free to deploy, free to distribute.
About the author

Andres Gomez Casanova

Andres Gomez Casanova is a DB2 DBA consultant with more than nine years working with DB2 LUW in different industries. He is a systems engineer from the Colombian School of Engineering (Escuela Colombiana de Ingeniería), and he has a master's degree in computer engineering (Génie Informatique) from Joseph Fourier, Grenoble I University. He has a deep knowledge in software engineering and has developed several open source projects, mainly for Java and SQL PL. He loves to travel around the world and visit the Unesco sites. You can follow him on Twitter at @angoca.

© Copyright IBM Corporation 2015
Trademarks
(www.ibm.com/developerworks/ibm/trademarks/)