

GeneXus[™]
by **Globant**

Unit Testing Capability

GeneXus[™]

Unit testing is a method by which individual units of source code are tested to determine if they are fit for use. The goal of unit testing is to separate each part of the program and test that the individual parts are working correctly.

Benefits of Unit Testing

GeneXus[™]
by Globant



GeneXus developers need to run and debug their procedures in some way after writing their code and rules. Traditionally they test input/output using self-made Web Panels or UI to run it using different input and evaluating outputs. All this effort can be reduced (and automated) by using the unit test object by which the developers's effort to validate their business logic is reusable.

Unit tests make possible to provide immediate feedback to developers, even before they commit a buggy change because each time a procedure changes and it is built, GeneXus can run this tests automatically in development environment and provide feedback to the user if that test is broken.

It gives an opportunity to detect bugs in GeneXus code without even creating an environment and deploying the solution. Unit Testing helps to detect issues at an early stage of the development process preventing lots of costs and bugs in the future without affecting any other parts of the software.

Moreover, detecting and fixing bugs is easier when these are found on a unit of code rather than on the whole system. Unit tests run very fast and can be shared through GeneXus Server, but they must never be deployed to production.

It also provides the opportunity to make the code more robust, reliable, and stable.

Unit testing in GeneXus

GeneXus™
by Globant



Procedures



Data Providers

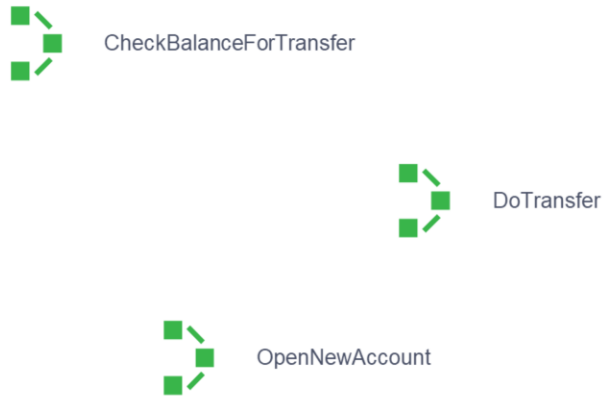


Business components

In GeneXus, the procedures are the way to encapsulate business logic code and reutilize it on different panels. For that reason, to test the business logic you must encapsulate it in procedures. If you don't have this development practice, you must refactor your application and change the way you program.

Besides, part of the application logic is included in data providers and business components. We can create tests to verify the behavior of these objects.

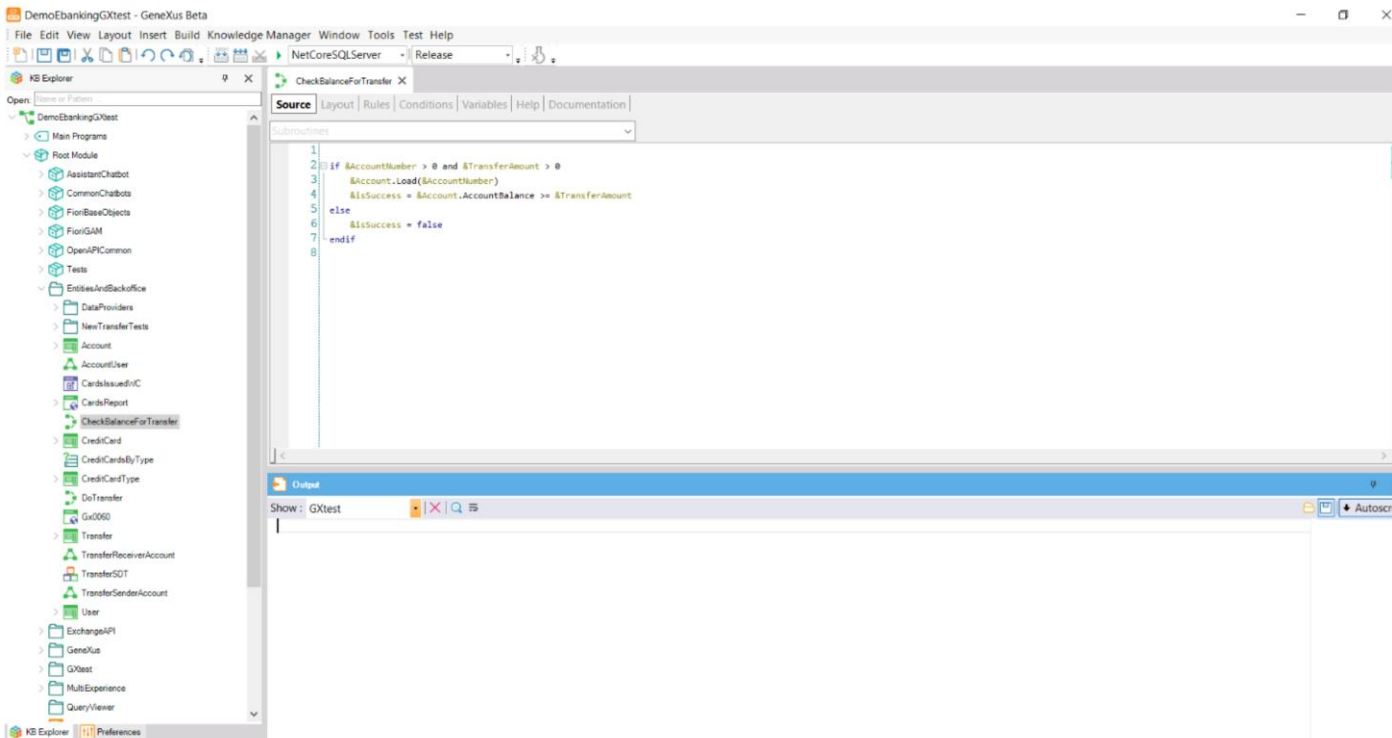
From GeneXus IDE it is possible to generate unit tests easily over these non-interface objects. And also, it is possible to create standalone unit tests to verify the integration between different processes.



The most weight of unit tests in the KB will be over the procedures, as they allow us to check the core business logic behavior.

For every procedure of the KB logic, we will create a unit test that atomically verifies how does it work. If it is necessary, we will create a unit test that calls for several procedures to test the integration of procedures/functions.

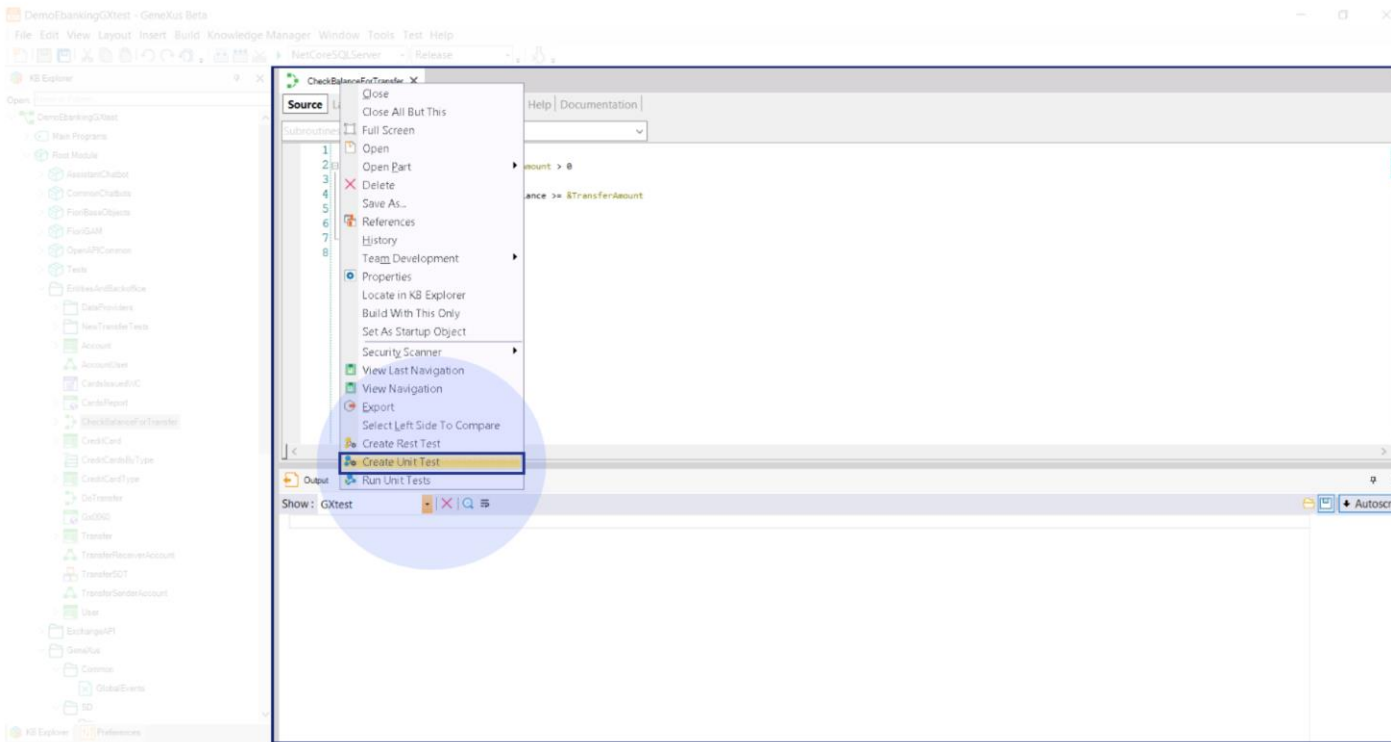
Creating the first unit test



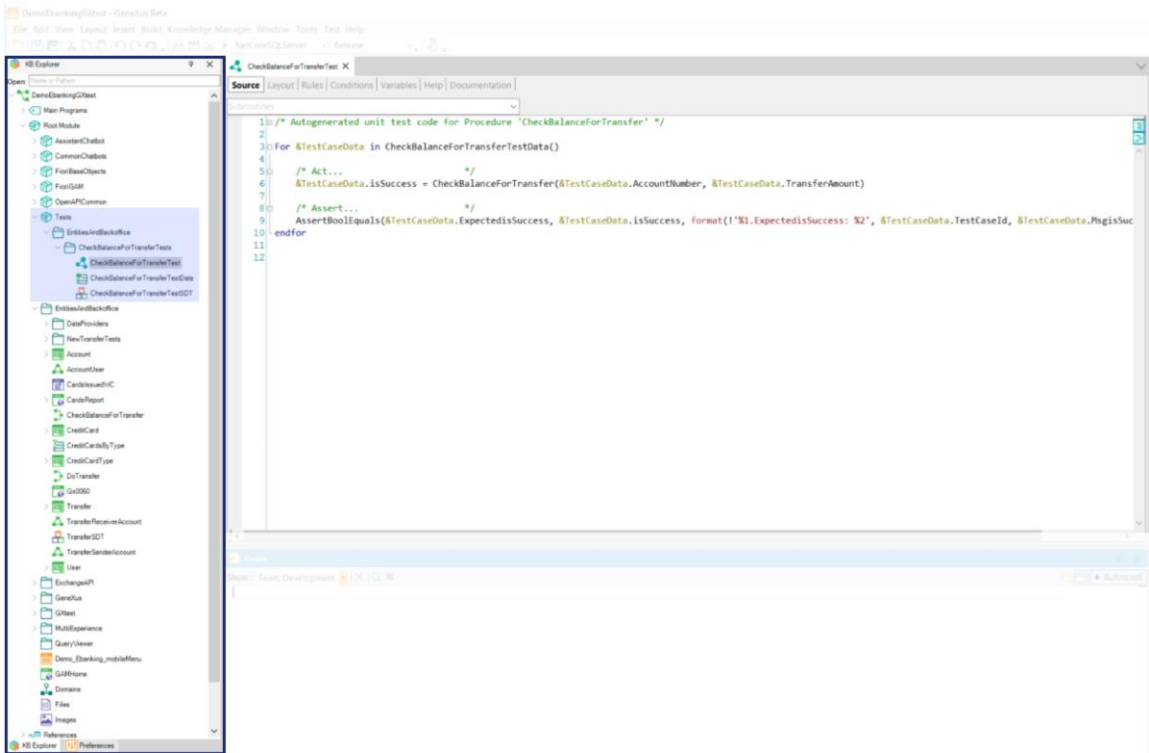
Suppose you have a procedure called `CheckBalanceForTransfer` - it has two input parameters (account number and transfer amount) and an output parameter with a boolean variable of transfer success.

The procedure approves or rejects the transfer, depending on the validity and balance of the account.

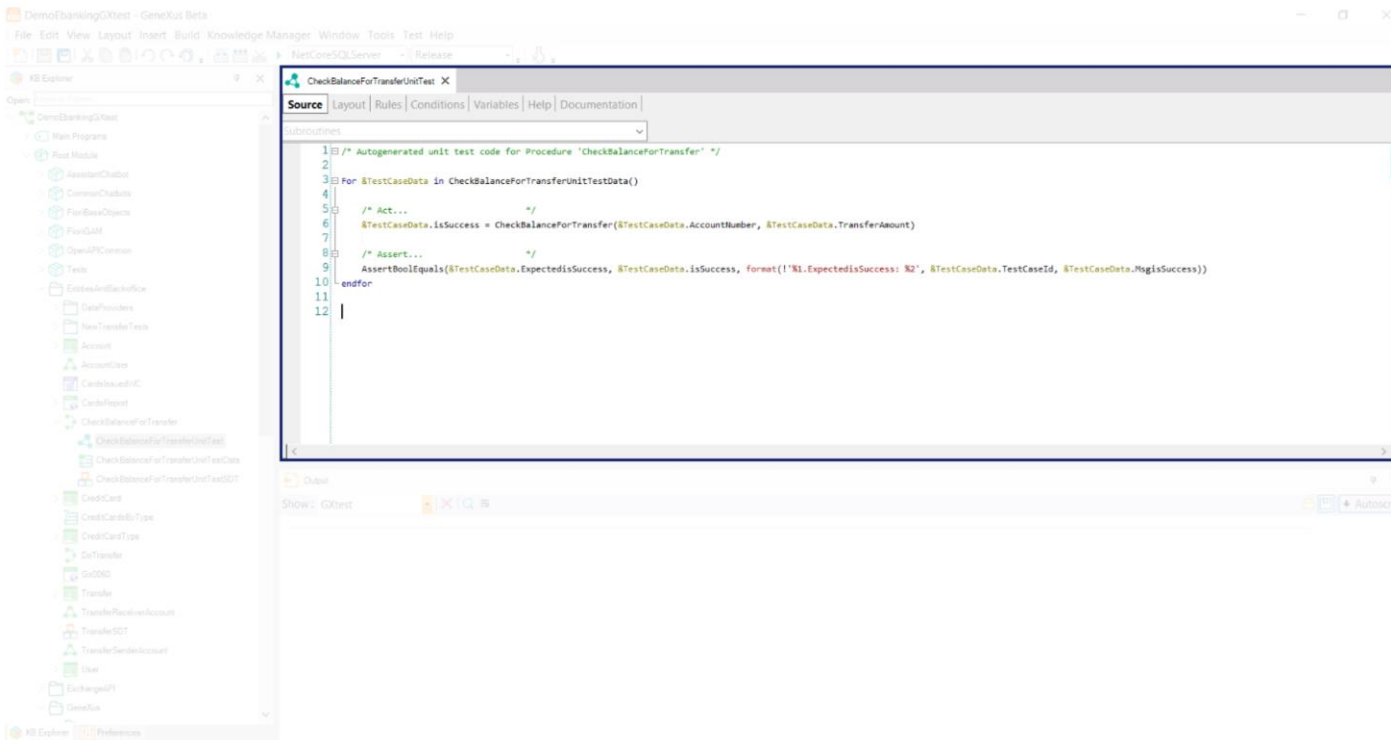
So, as a developer, instead of doing manual testing with an auxiliary panel, button, and input fields to call the procedure, the developer should create the unit test with the test cases. Let's see how to easily do it in GeneXus!



To create the unit test object we just click right over the procedure tab, or in the KB explorer, and select the option “Create Unit Test”.



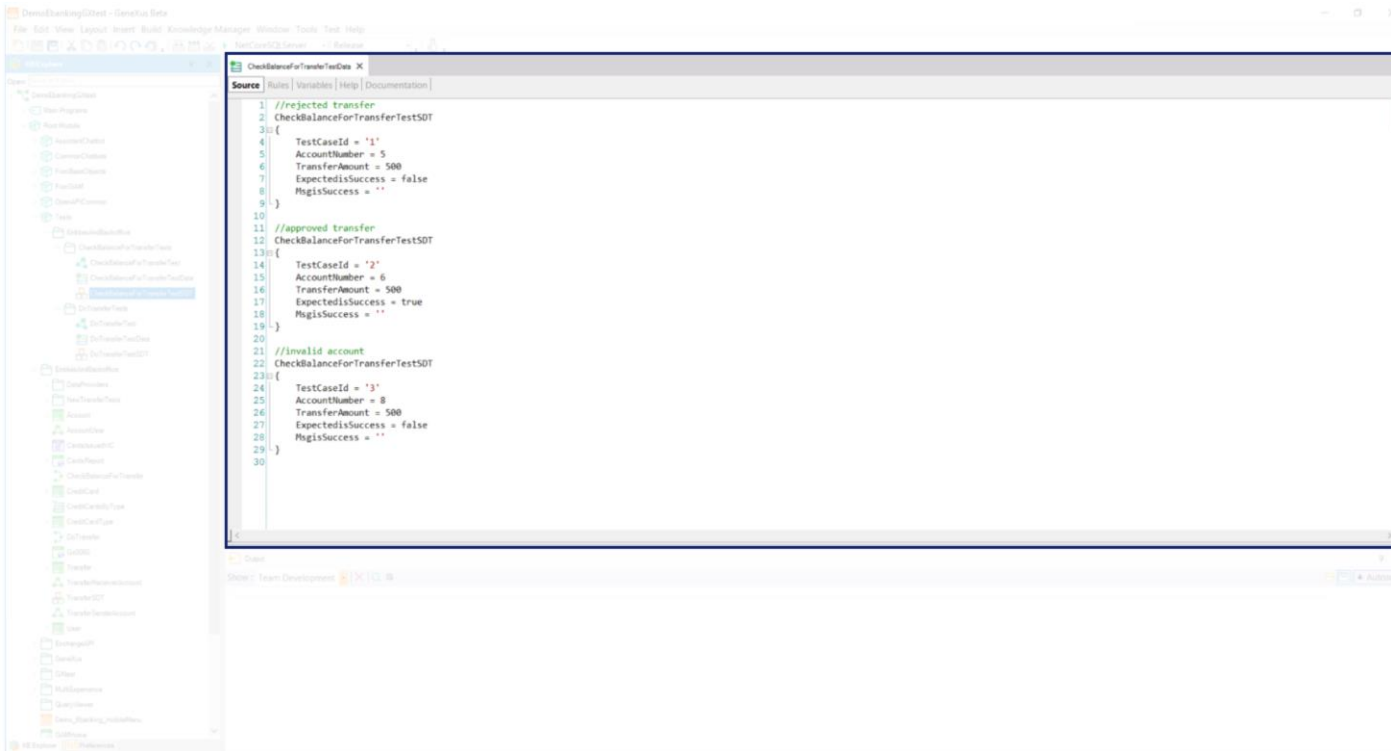
This option will automatically generate the test objects within a module: the unit test object, the data provider in which we will define the test cases and the SDT with the test cases structure.



In the example, you can see that the generated unit test template iterates over the test cases collection `CheckBalanceForTransferUnitTestData`, in which we just add the test cases.

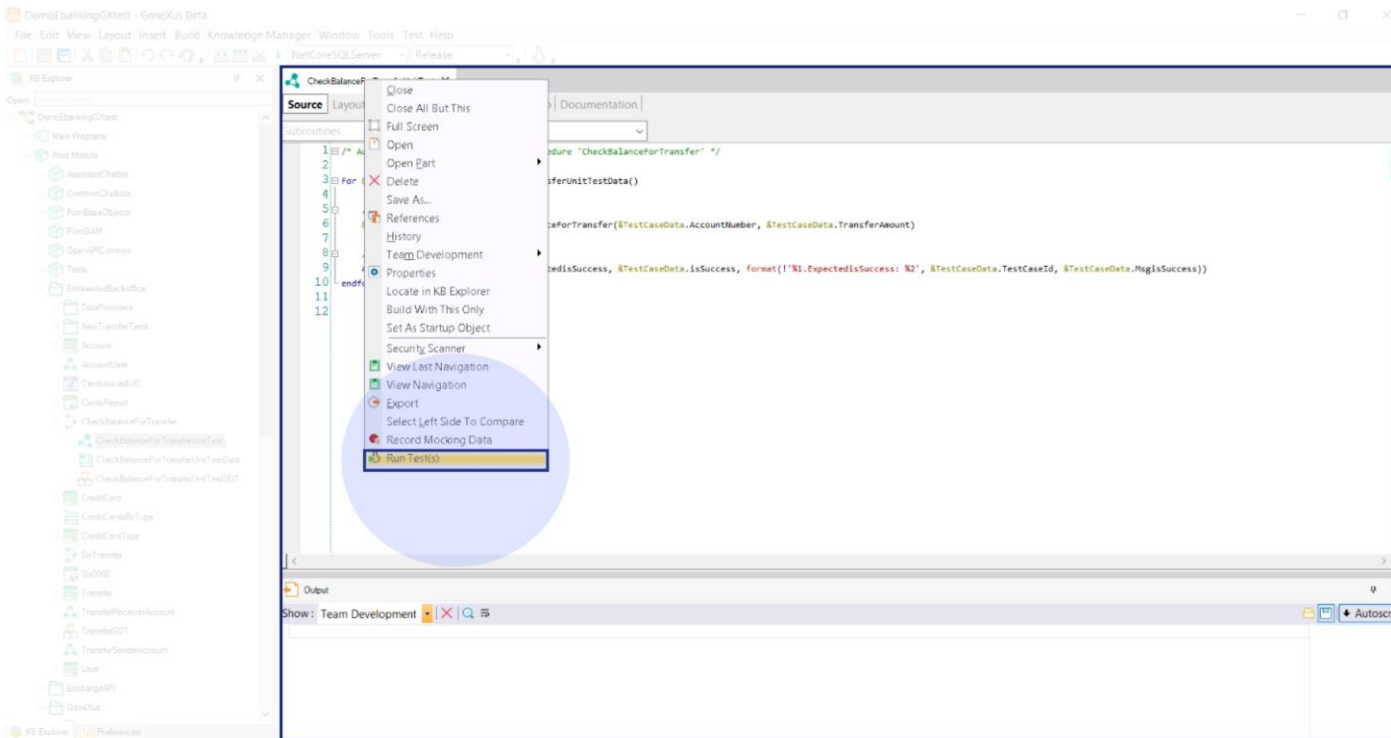
In each iteration, the unit test calls the procedure that we want to test and verifies the expected values with assertion functions. Assertions are mechanisms that defines whether tests pass or fail. There are different assertion functions depending on the type of the parameters they assert on (boolean, numeric, string).

Note that in this example the assertion was automatically generated and it validates the output parameter of the procedure.

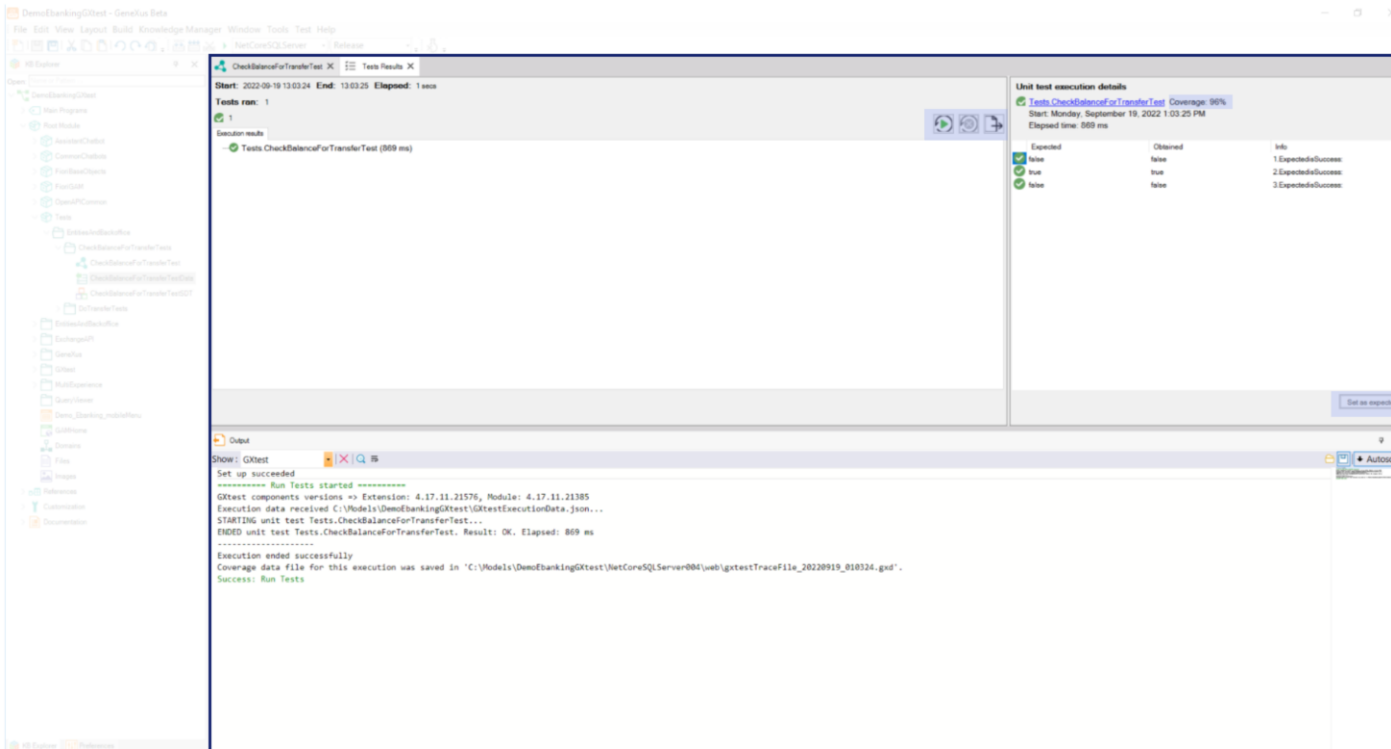


What remains is to add the test cases in the Data provider.

In this example: the test case 1, with account number 5, is from an account without balance, so the transfer must be rejected. The test case 2, is from a valid account with balance, so, the transfer must be approved. Finally, the test case 3, with account number 8 is an invalid account number, so, it must be rejected too.



To execute the test, right click over it and select “Run Test(s)” option. A build process is performed and, if the build is successful, the test is executed.



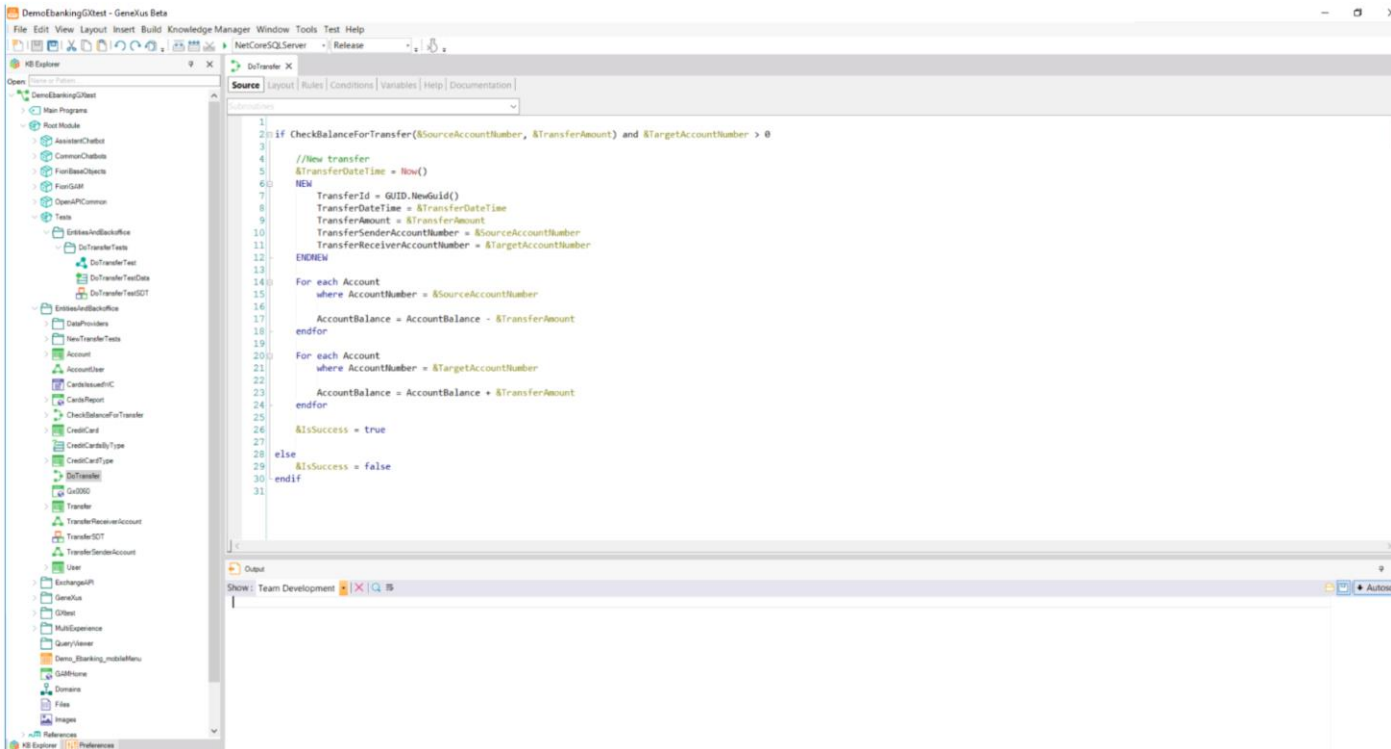
After the execution, you can see information about the result and elapsed time in the Test Results panel. Also, for each test, you can visualize the expected and obtained values for each assertion.

From the Tests Results panel, you can perform some actions like “Run again”, “Run failed again” and “Export execution results as HTML”.

In the Unit test execution details section, you can also visualize the “Test coverage” percentage and “Set as expected” the obtained values, if applies.

We will see the “Coverage” and “Set as expected” features later on.

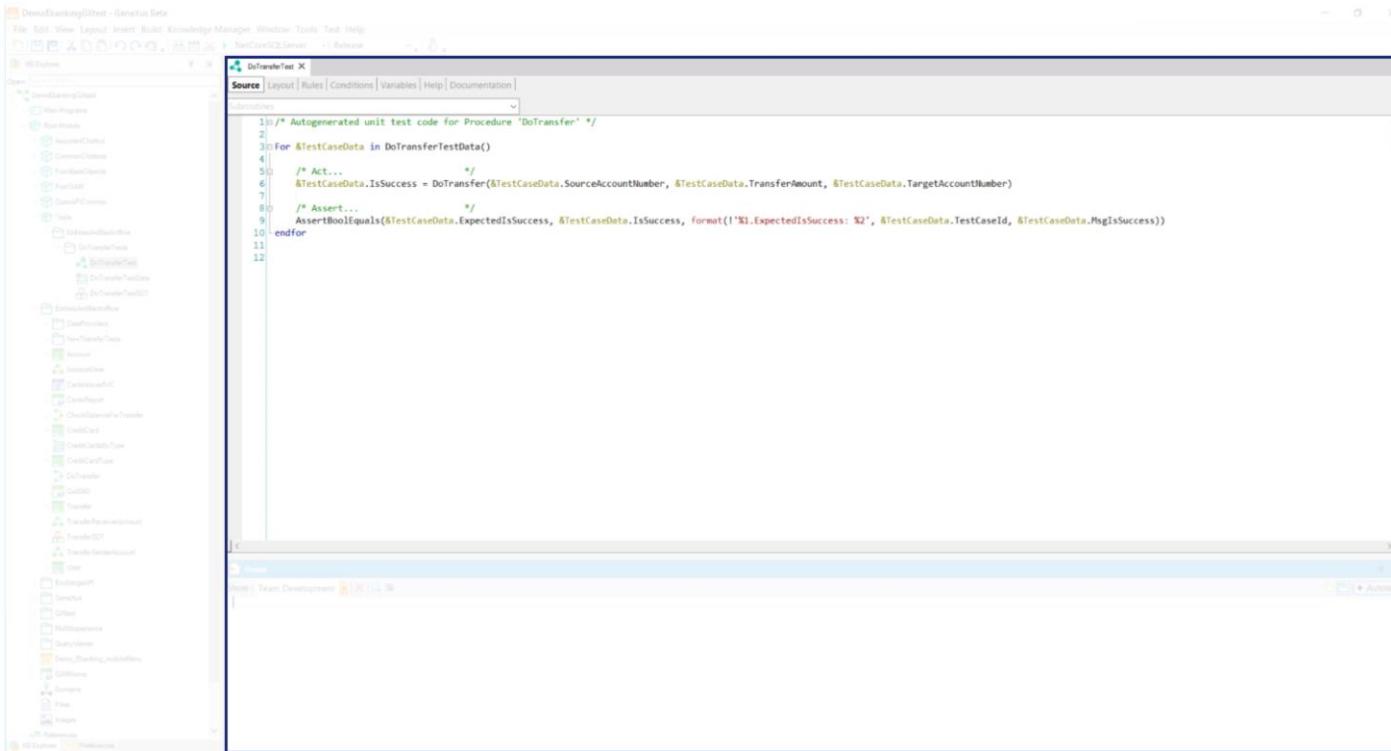
Database validations



Now that we know about the unit test concept, let's see how to add database validations.

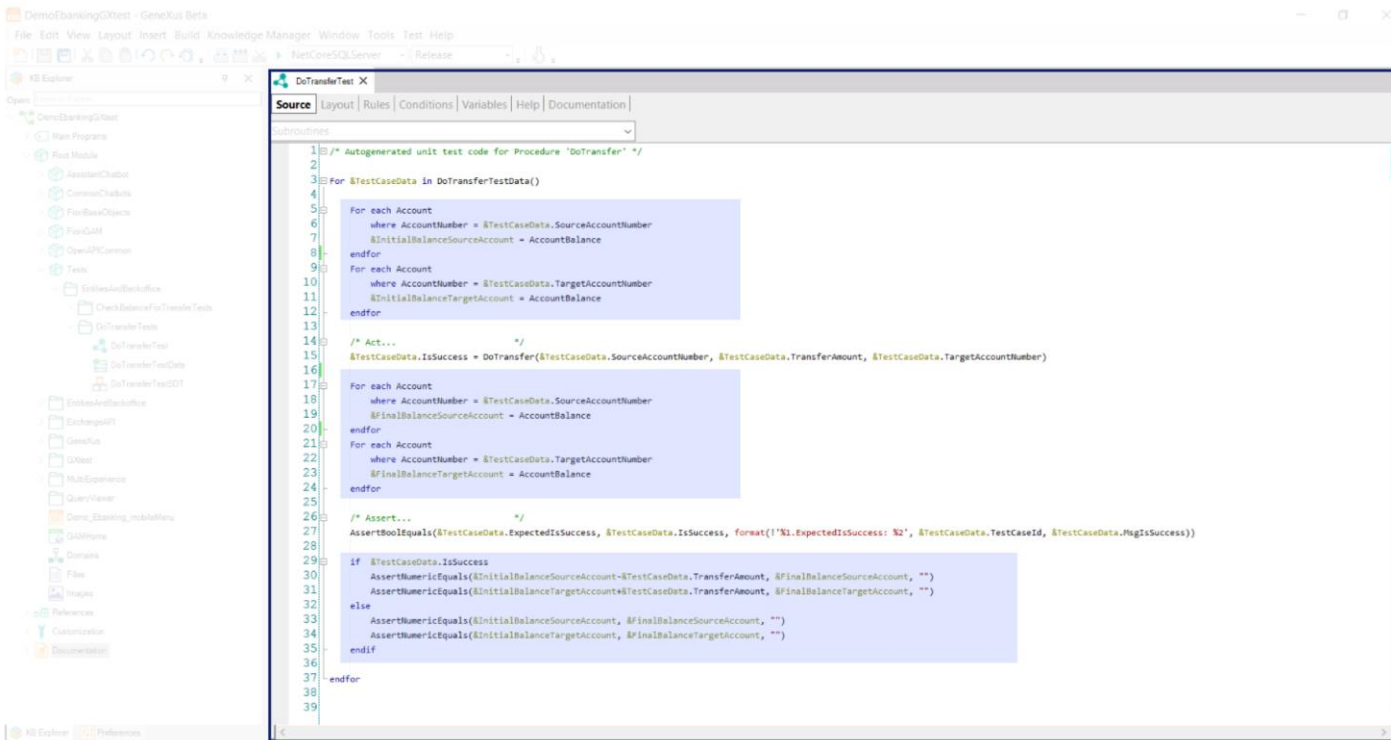
Suppose that you have a procedure called DoTransfer. It has as input parameters the SourceAccountNumber, TransferAmount and TargetAccountNumber variables, and IsSuccess as output variable.

The procedure checks the account balance, if it is enough, it creates the Transfer and updates the balances. Finally, it sets the variable IsSuccess in True.



When GeneXus generates the unit test, it automatically generates a template with the output parameter assertion. The developer should add in genexus code the database validations in the unit test.

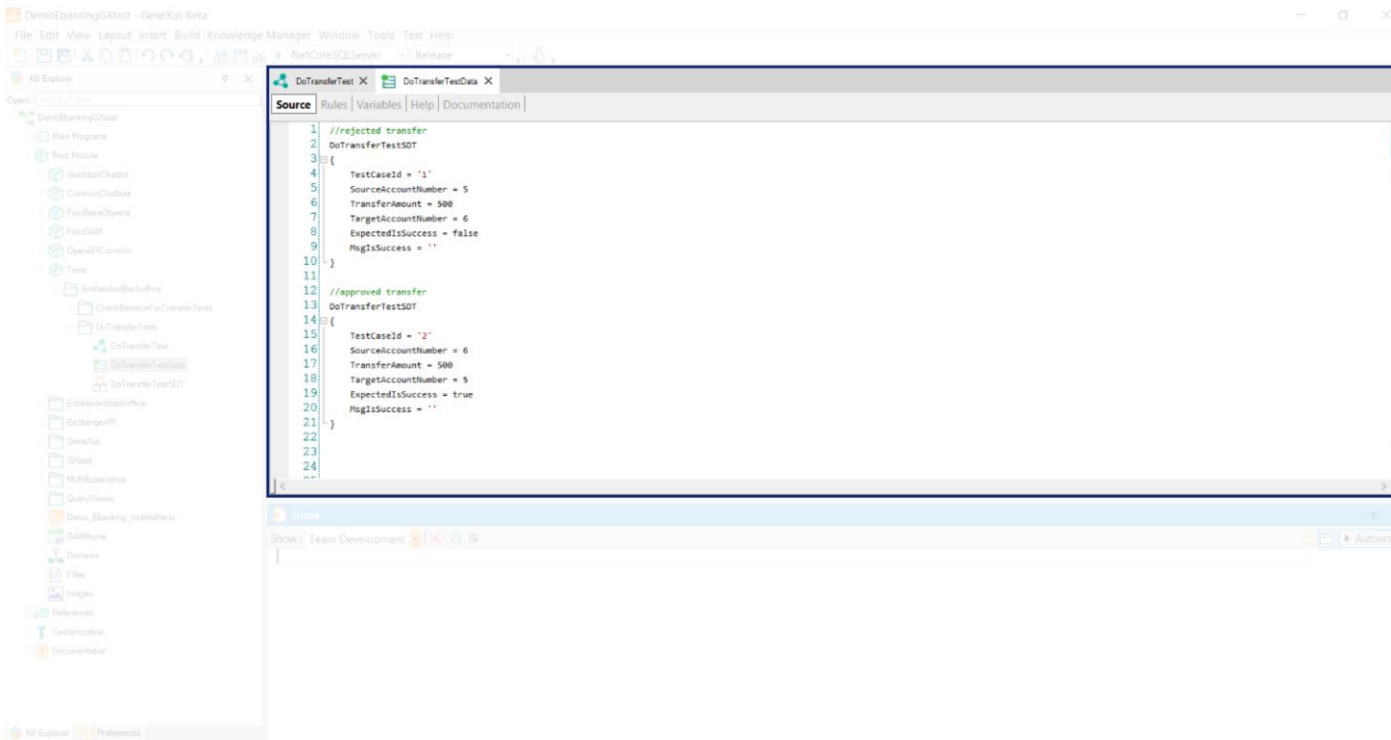
Remember that the unit test is a special GeneXus procedure, so, you can add any code that you need to do your test more robust.



```
1 /* Autogenerated unit test code for Procedure 'DoTransfer' */
2
3 For &TestCaseData In DoTransferTestData()
4
5     For each Account
6         where AccountNumber = &TestCaseData.SourceAccountNumber
7             &InitialBalanceSourceAccount = AccountBalance
8         endfor
9     For each Account
10        where AccountNumber = &TestCaseData.TargetAccountNumber
11            &InitialBalanceTargetAccount = AccountBalance
12        endfor
13
14 /* Act... */
15 &TestCaseData.IsSuccess = DoTransfer(&TestCaseData.SourceAccountNumber, &TestCaseData.TransferAmount, &TestCaseData.TargetAccountNumber)
16
17 For each Account
18     where AccountNumber = &TestCaseData.SourceAccountNumber
19         &FinalBalanceSourceAccount = AccountBalance
20     endfor
21 For each Account
22     where AccountNumber = &TestCaseData.TargetAccountNumber
23         &FinalBalanceTargetAccount = AccountBalance
24     endfor
25
26 /* Assert... */
27 AssertBoolean(&TestCaseData.ExpectedIsSuccess, &TestCaseData.IsSuccess, format('!%i.ExpectedIsSuccess: %2', &TestCaseData.TestCaseId, &TestCaseData.MsgIsSuccess))
28
29 If &TestCaseData.IsSuccess
30     AssertNumericEquals(&InitialBalanceSourceAccount-&TestCaseData.TransferAmount, &FinalBalanceSourceAccount, "")
31     AssertNumericEquals(&InitialBalanceTargetAccount+&TestCaseData.TransferAmount, &FinalBalanceTargetAccount, "")
32 else
33     AssertNumericEquals(&InitialBalanceSourceAccount, &FinalBalanceSourceAccount, "")
34     AssertNumericEquals(&InitialBalanceTargetAccount, &FinalBalanceTargetAccount, "")
35 endif
36
37 endfor
38
39
```

In this example, we add Balance validations over the Account table after exercising the procedure DoTransfer.

We could add more validations, for example if a new transfer was created in the Transfer table.



Finally, we set the test cases in the `DoTransferDataProvider` with a rejected and a successful transfer.

The screenshot displays the GeneXus IDE interface during a unit test execution. The main window shows the test results for 'DoTransferTest', which passed successfully. The test execution details panel on the right provides a summary of the test, including coverage and execution time. The output window at the bottom shows the detailed execution log, including component versions and the successful completion of the test.

Unit test execution details

Expected	Obtained	Info
false	false	1 ExpectedSuccess
0	0	
100000	100000	
99500	99500	2 ExpectedSuccess
500	500	

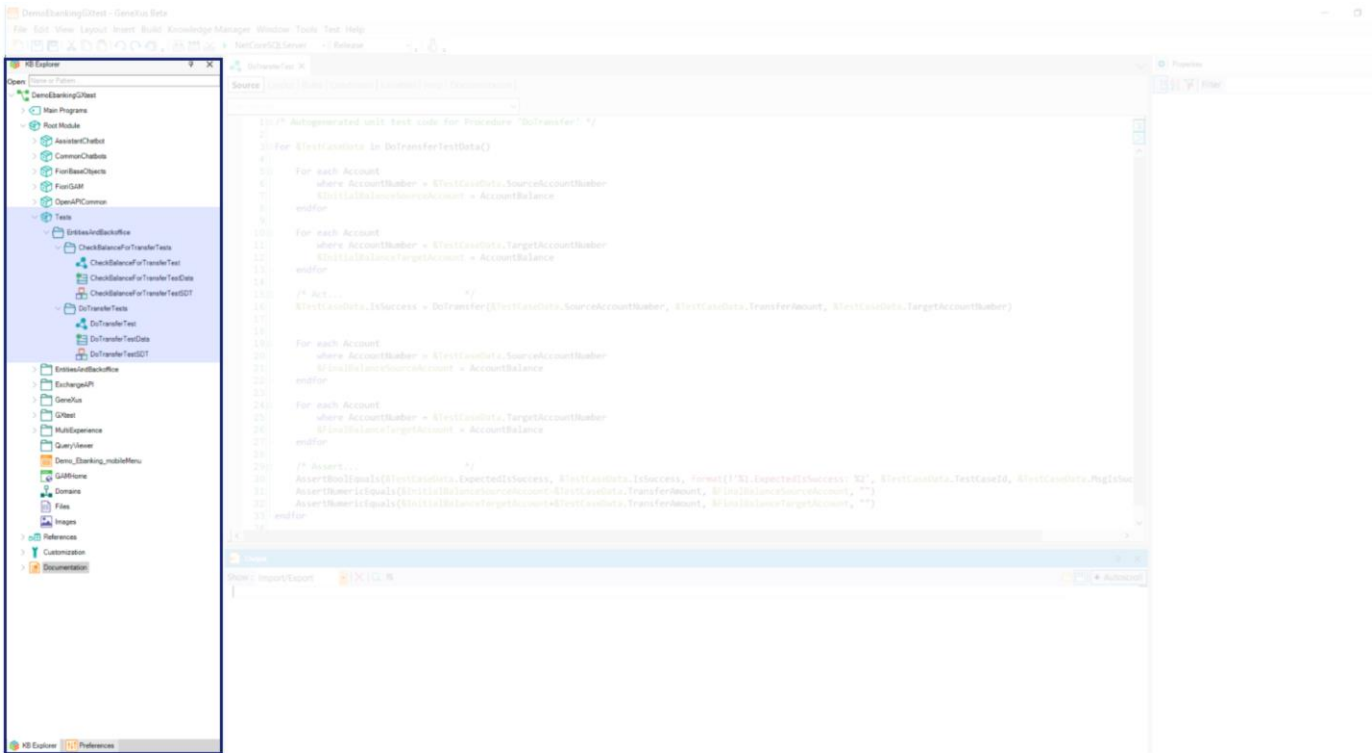
```

===== Run Tests started =====
GXtest components versions -> Extension: 4.17.11.21576, Module: 4.17.11.21385
Execution data received C:\Models\DemoEbankingGXtest\GXtest\ExecutionData.json...
STARTING unit test Tests.DoTransferTest...
ENDED unit test Tests.DoTransferTest. Result: OK. Elapsed: 1807 ms
=====
Execution ended successfully
Coverage data file for this execution was saved in 'C:\Models\DemoEbankingGXtest\NetCoreSQLServer004\web\gxtest\TraceFile_20220920_100542.gxd'.
Success: Run Tests

```

Once is all set, we run the unit test.

In the Tests Results panel, you can see all executed assertions. For each test case, there is the output parameter assertion and then the Balance validations over the Account table. Note that in the Info column is shown the third parameter of the assertions, usually used to identify the assertion and as a context for future modifications.



Also, note that tests are placed in the “Tests” module, and each proc test is placed in a folder with the name of procedures.

As we commented previously, with the unit tests is also possible to test data providers (for example if the data provider gets data from an external service or procedure) and business components. And it is possible/necessary to create standalone unit tests to verify the integration between different functions.

GeneXus[™]
by **Globant**

training.genexus.com
wiki.genexus.com