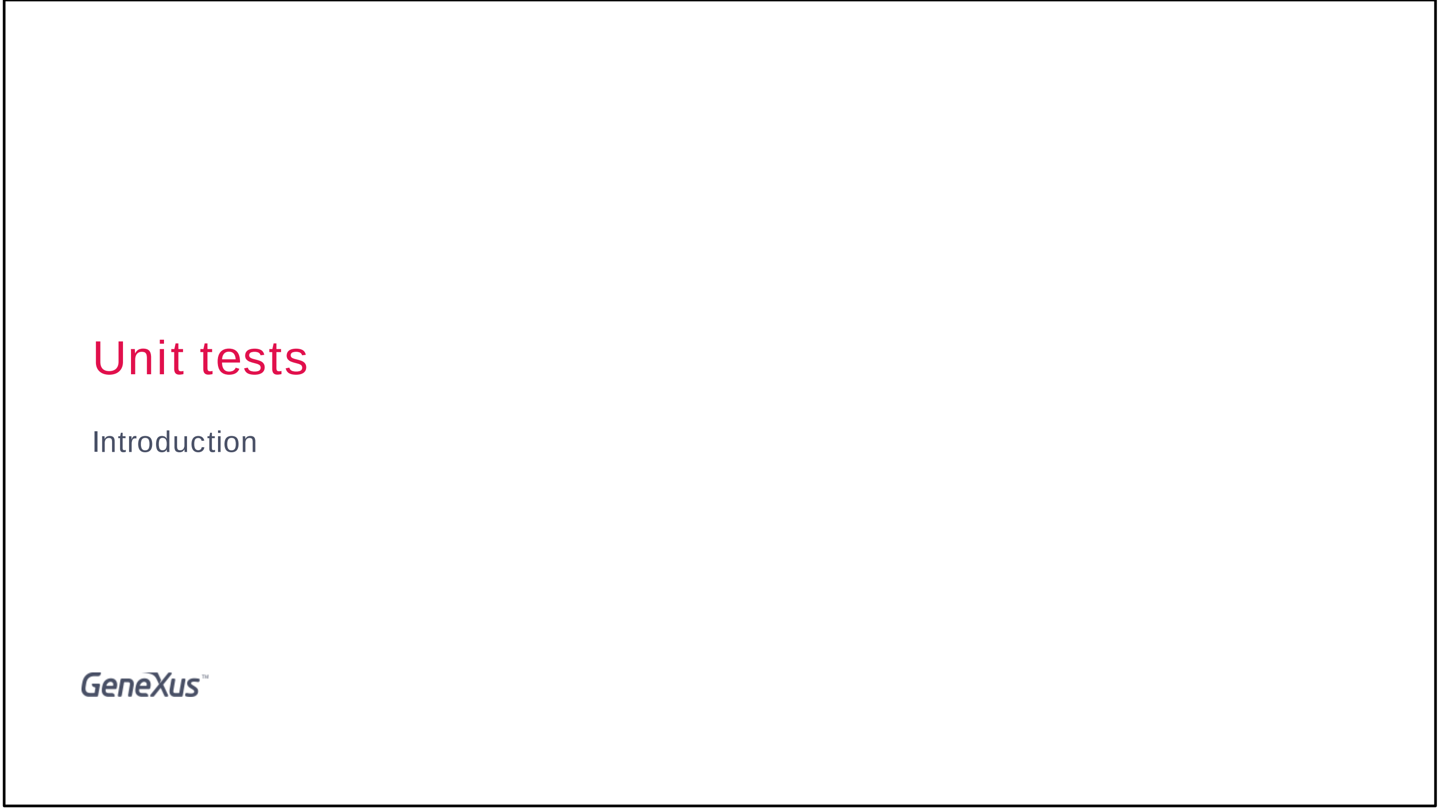
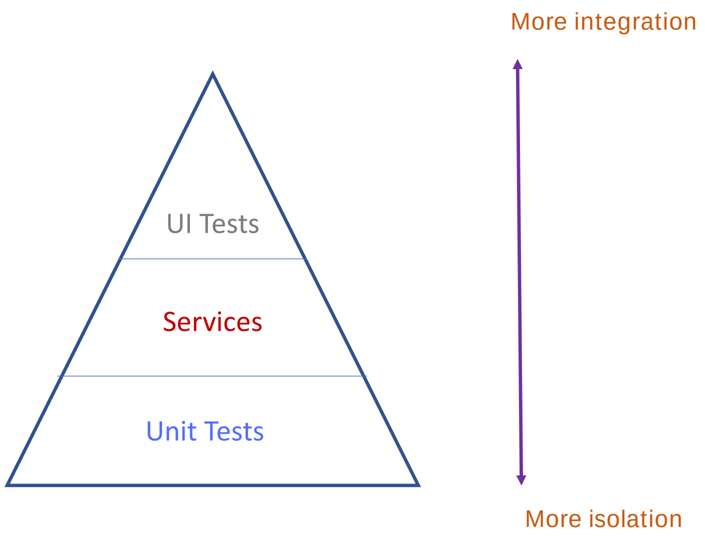


Unit tests

Introduction

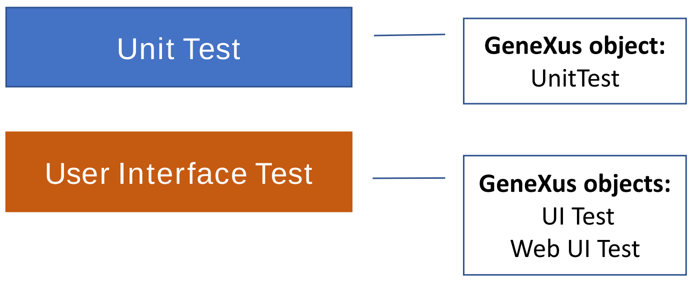
Throughout the development of our application for the Travel Agency, we have mentioned the importance of separately testing the new functionalities we are developing, and then testing the entire application to make sure it behaves as expected.

Automatic tests

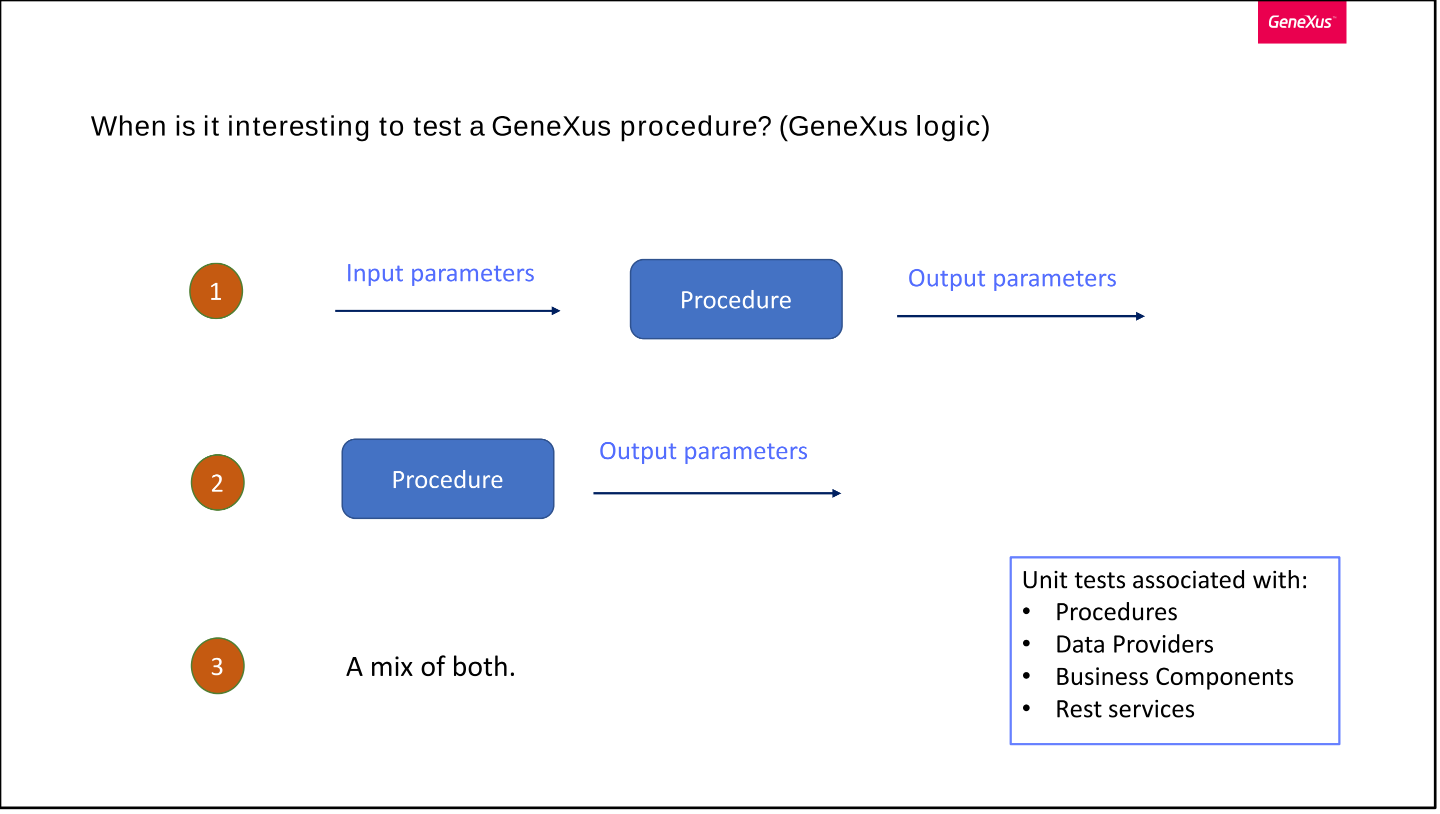


As the application grows, this type of task can become increasingly burdensome, so GeneXus helps us by providing features to create and run automatic tests, in order to reduce some of the manual verification work.

Automatic tests



- **Unit tests** allow us to test a part of the application separately. In GeneXus, unit tests without an interface are applied to tests on procedures, Data Providers, and Business Components. In short, on those components where the business logic of our application should reside, and that's why the Unit Test object exists.
- The **Interface test** allows us to create tests simulating a user’s actions on the browser, in order to test entire application flows. For this purpose, there are UITest objects for mobile interfaces, and Web UITest objects for web interfaces.



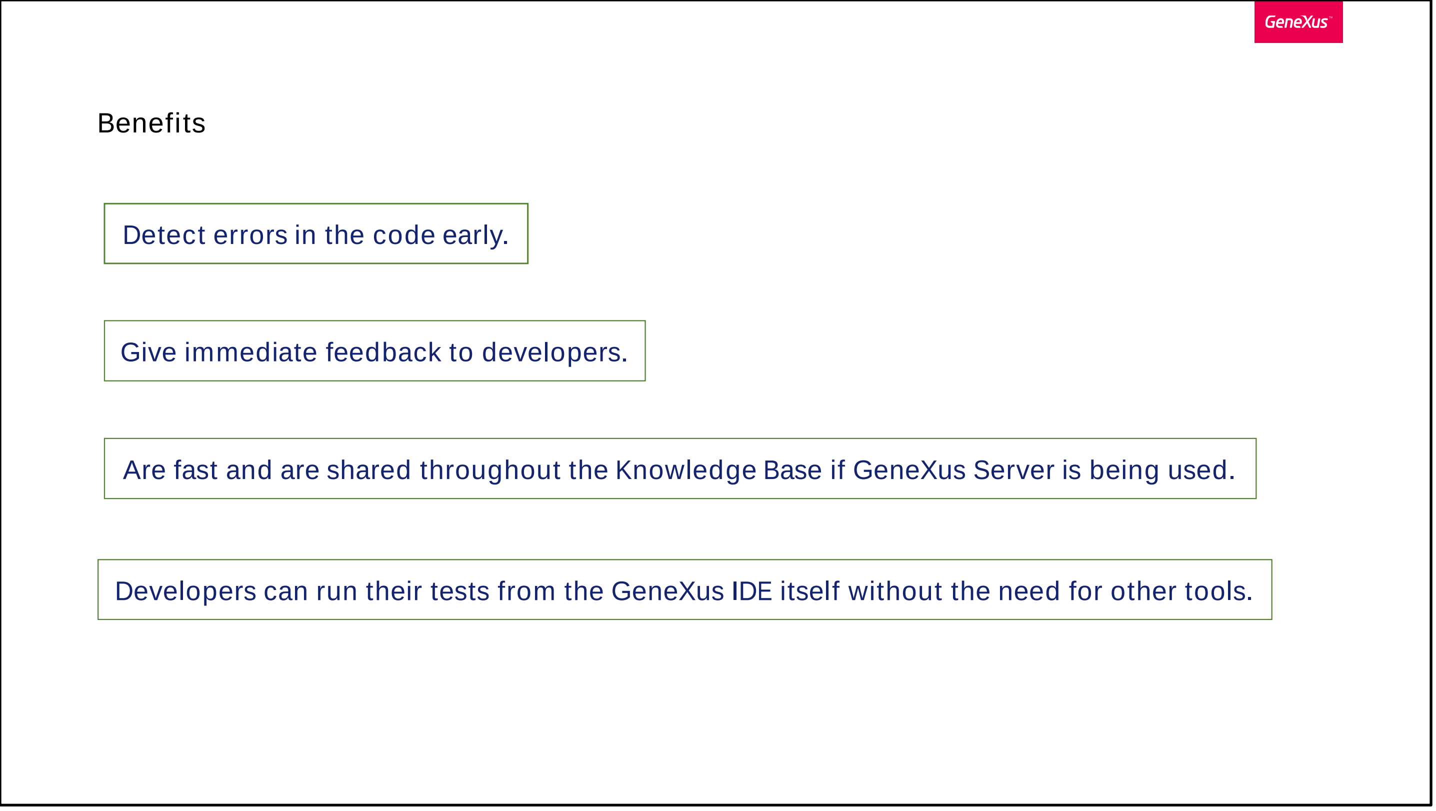
When is it interesting to test a GeneXus procedure? (GeneXus logic)

Unit tests associated with:
- Procedures
- Data Providers
- Business Components
- Rest services

So, when should a GeneXus procedure be tested?

- When given some input parameters, the output of the Procedure is to be tested by comparing the expected results (using Assertions) with the results of the process (either output variables, files, or database records). To this end, assertions—that is, affirmations or statements included in the procedure—are used.
- When there are no input parameters, but from parameters or database settings the Procedure is expected to return some expected results (for example, variables or database records).
- And when there is a mix of both scenarios.

This means that any GeneXus Procedure can be tested to verify that it works according to the defined specifications. In addition to Procedures, unit tests associated with Data Providers and Business Components can also be defined.



Benefits

Detect errors in the code early.

Give immediate feedback to developers.

Are fast and are shared throughout the Knowledge Base if GeneXus Server is being used.

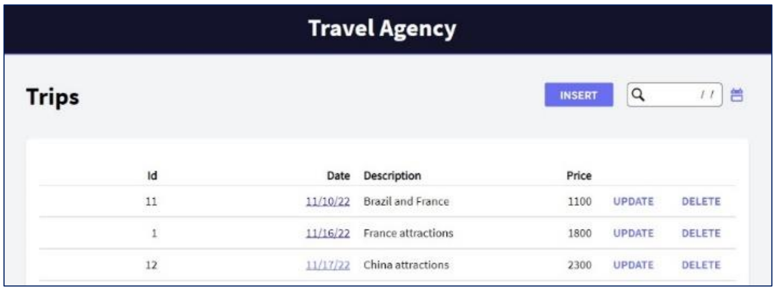
Developers can run their tests from the GeneXus IDE itself without the need for other tools.

The main benefits of performing unit tests are as follows:

- Detect errors in the code early.
- Give immediate feedback to developers.
- They are fast and shared throughout the Knowledge Base if GeneXus Server is being used.
- Developers can run their tests from the GeneXus IDE itself with no need for other tools.

Example: UpdateTripPrice

Procedure that updates the price of a trip according to a percentage received by parameter.



Travel Agency

Trips

INSERT

Id	Date	Description	Price		
11	11/10/22	Brazil and France	1100	UPDATE	DELETE
1	11/16/22	France attractions	1800	UPDATE	DELETE
12	11/17/22	China attractions	2300	UPDATE	DELETE

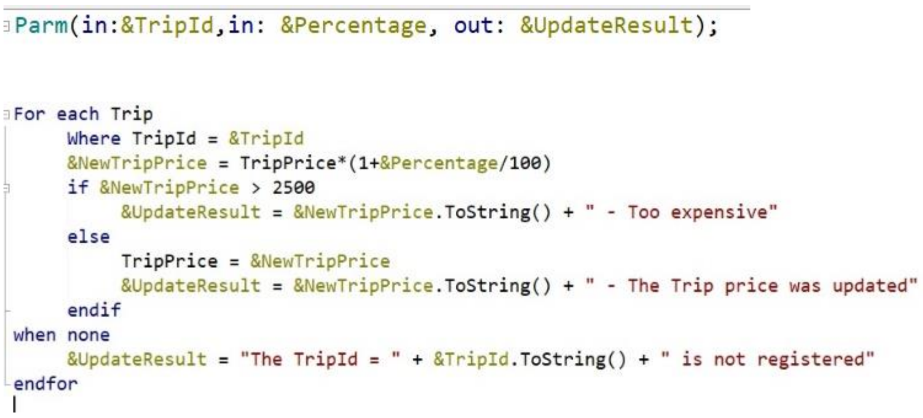
OK, let's see an example.

From the launchpad, we run Work with Trip, and see that we have three trips registered with their current costs.

1100, 1800, and 2300

GeneXus

Example: UpdateTripPrice



```
Parm(in:&TripId,in: &Percentage, out: &UpdateResult);


For each Trip
     Where TripId = &TripId
     &NewTripPrice = TripPrice*(1+&Percentage/100)
     if &NewTripPrice > 2500
          &UpdateResult = &NewTripPrice.ToString() + " - Too expensive"
     else
          TripPrice = &NewTripPrice
          &UpdateResult = &NewTripPrice.ToString() + " - The Trip price was updated"
     endif
when none
     &UpdateResult = "The TripId = " + &TripId.ToString() + " is not registered"
endfor
```

We have created a procedure, named UpdateTripPrice, which calculates the price increase of a given trip, according to a percentage received by parameter. For the Travel Agency, the final price of a trip cannot exceed 2500.

We see then that the procedure receives two input parameters:

The &TripId variable and the increase percentage.

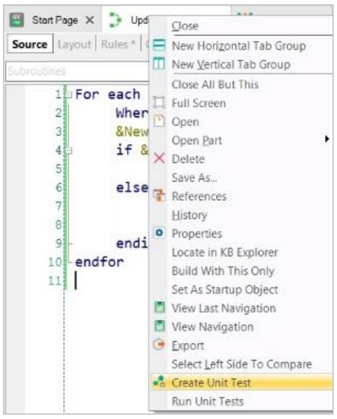
Also, it has an output parameter that returns the value obtained by the increase along with a comment that indicates whether the update was performed or the maximum amount indicated was exceeded.

Then, for the TripId received, the procedure calculates the value according to the percentage that was also received; if that value is greater than 2500, it does not update and returns the corresponding message.

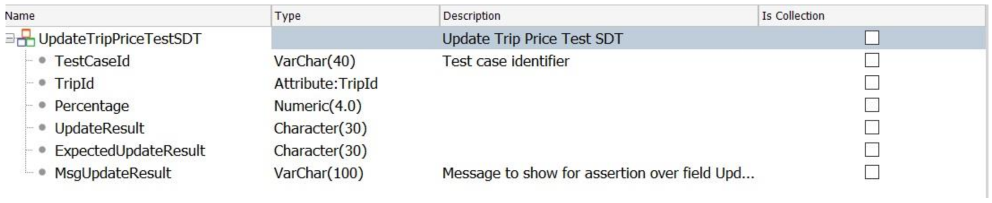
If the value is lower than or equal to 2500, it does update the value of the TripPrice attribute and also returns the corresponding message.

In case there is no trip with the TripId value received by parameter, a message will be returned indicating that the trip is not registered.

Creation of the associated Unit test



Object: UpdateTripPriceTestSDT



Name	Type	Description	Is Collection
UpdateTripPriceTestSDT		Update Trip Price Test SDT	☐
TestCaseId	VarChar(40)	Test case identifier	☐
TripId	Attribute:TripId		☐
Percentage	Numeric(4.0)		☐
UpdateResult	Character(30)		☐
ExpectedUpdateResult	Character(30)		☐
MsgUpdateResult	VarChar(100)	Message to show for assertion over field Upd...	☐

To test this procedure, we are going to create the corresponding Unit test.

To do this, we right-click on the procedure tab and select Create Unit Test.

Then GeneXus creates three objects, which can be seen below the new Tests node in the KBExplorer window:

The UpdateTripPriceTestSDT object,

which defines the structure of a given test case for the object we are testing.

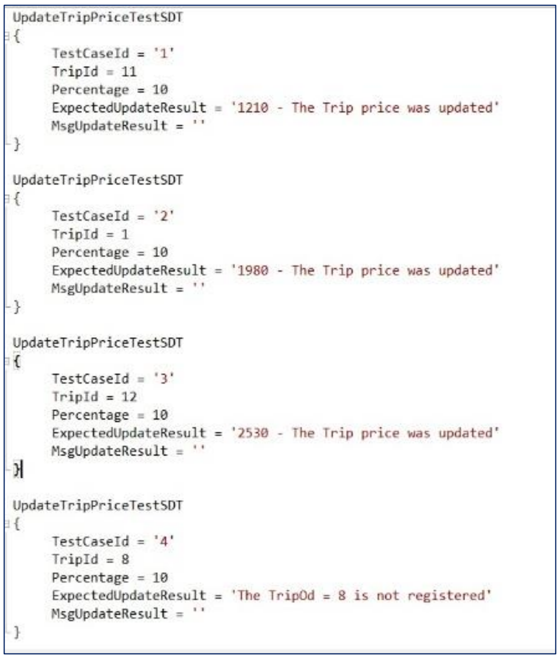
Note that—as we would do—it defines both input variables with the same name as the parameters of the procedure, and defines an ExpectedUpdateResult variable where we will be able to define the value of the expected result.

In short, we will be able to say, for example, that for the trip TripId=11, with a current cost of 1100, if we indicate an increase percentage of 10% we expect a result of 1210—with a message indicating that the price was updated correctly.

We can also assign a message that we want to be displayed in case the result is different from the one indicated.

Creation of the associated Unit test

Object: **UpdateTripPriceTestData**



```
UpdateTripPriceTestSDT
{
	TestCaseId = '1'
	TripId = 11
	Percentage = 10
	ExpectedUpdateResult = '1210 - The Trip price was updated'
	MsgUpdateResult = ''
}

UpdateTripPriceTestSDT
{
	TestCaseId = '2'
	TripId = 1
	Percentage = 10
	ExpectedUpdateResult = '1980 - The Trip price was updated'
	MsgUpdateResult = ''
}

UpdateTripPriceTestSDT
{
	TestCaseId = '3'
	TripId = 12
	Percentage = 10
	ExpectedUpdateResult = '2530 - The Trip price was updated'
	MsgUpdateResult = ''
}

UpdateTripPriceTestSDT
{
	TestCaseId = '4'
	TripId = 8
	Percentage = 10
	ExpectedUpdateResult = 'The TripOd = 8 is not registered'
	MsgUpdateResult = ''
}
```

Now let's move on to the **UpdateTripPriceTestData object**, also created automatically by GeneXus.

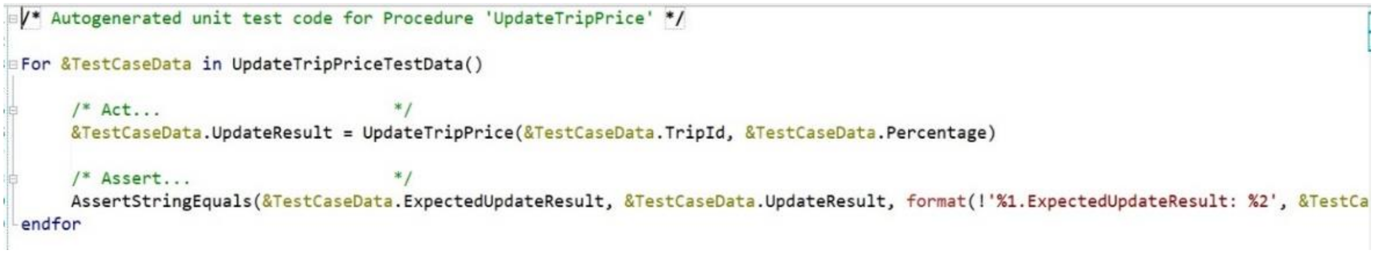
This Data Provider is based on the SDT we saw earlier, and allows us to define a data set. By default, 5 test cases are created, but we can modify it.

We have three trips registered, so we are going to define three tests with an increase percentage of 10%. In each case, we indicate the expected value returned by the procedure.

But we also define a fourth test set for TripId= 8 which does not currently exist in our database.

Creation of the associated Unit test

Object: UpdateTripPriceTest



```
/* Autogenerated unit test code for Procedure 'UpdateTripPrice' */

For &TestCaseData in UpdateTripPriceTestData()

    /* Act...                   */
    &TestCaseData.UpdateResult = UpdateTripPrice(&TestCaseData.TripId, &TestCaseData.Percentage)

    /* Assert...                */
    AssertStringEquals(&TestCaseData.ExpectedUpdateResult, &TestCaseData.UpdateResult, format(!'%1.ExpectedUpdateResult: %2', &TestCa
endfor
```

Assert command to compare an expected result with an obtained result

- AssertStringEquals – to compare texts
- AssertBoolEquals – to compare boolean values
- AssertNumericEquals – to compare numeric values

Lastly, the **UpdateTripPriceTest object** is the one that will run through the collection of test cases, and for each one of them it will invoke our procedure and validate whether the result obtained matches the expected result.

This object is a GeneXus procedure and is programmed as such, so we are going to see a very familiar notation.

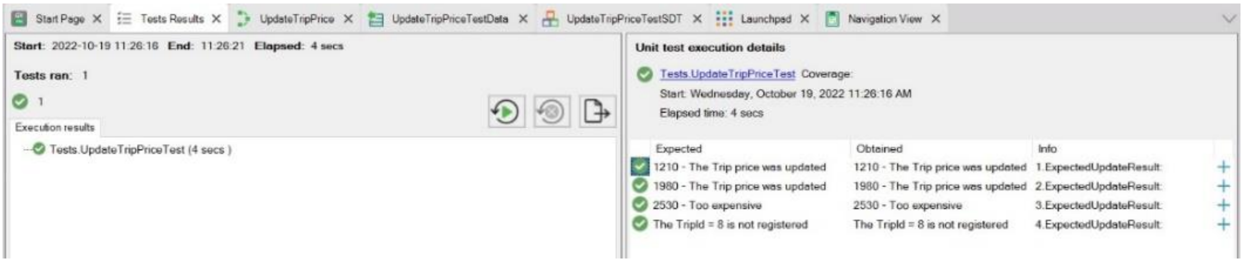
That is, FOR EVERY test case in the collection of Tests that we define in the data Provider, a call is made to the procedure we are testing with the input parameters defined in the test case and a variable as output value.

What is new in the unit test is the **ASSERT** command. It basically compares an expected result—defined as part of the test case—against the result obtained.

If the expected result and the result obtained are the same, the test is successful and is said to PASS, and if there is any difference, the test FAILS and an error is reported showing an associated message.

Here we are using the AssertStringEquals function to validate the result since the result is a text, but it is also possible to use AssertBoolEquals to compare Booleans or AssertNumericEquals for comparing numeric values.

Unit test execution



Start: 2022-10-19 11:26:16 **End:** 11:26:21 **Elapsed:** 4 secs

Tests ran: 1

✓ 1

Execution results

✓ Tests.UpdateTripPriceTest (4 secs)

Unit test execution details

✓ Tests.UpdateTripPriceTest Coverage:

Start: Wednesday, October 19, 2022 11:26:16 AM

Elapsed time: 4 secs

	Expected	Obtained	Info
✓	1210 - The Trip price was updated	1210 - The Trip price was updated	1.ExpectedUpdateResult:
✓	1980 - The Trip price was updated	1980 - The Trip price was updated	2.ExpectedUpdateResult:
✓	2530 - Too expensive	2530 - Too expensive	3.ExpectedUpdateResult:
✓	The TripId = 8 is not registered	The TripId = 8 is not registered	4.ExpectedUpdateResult:

OK. To run it, we right-click and select Run Test.

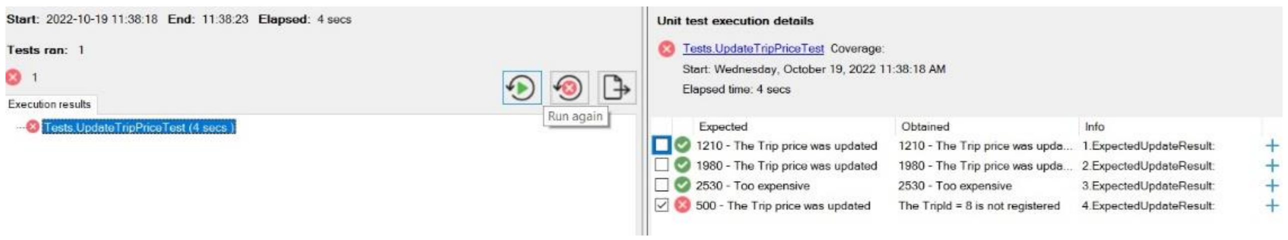
Once the test execution is complete, we will see the new window—named TEST-RESULTS—where we confirm that the test was executed (UpdateTripPriceTest) and that the result was successful because it is marked in green.

It also gives us information about the test execution time.
Here we see a line for each Assert we have defined in our test. For each one we can see the expected result, the result obtained, and the green or red mark depending on whether the Assert failed or passed.

Unit test execution

The Data Provider is modified to generate a fault.

```
UpdateTripPriceTestSDT
{
    TestCaseId = '4'
    TripId = 8
    Percentage = 10
    ExpectedUpdateResult = '500 - The Trip price was updated'
    MsgUpdateResult = ''
}
```



Start: 2022-10-19 11:38:18 **End:** 11:38:23 **Elapsed:** 4 secs

Tests ran: 1

🟥 1

Execution results

Tests.UpdateTripPriceTest (4 secs)

Unit test execution details

Tests.UpdateTripPriceTest Coverage:

Start: Wednesday, October 19, 2022 11:38:18 AM

Elapsed time: 4 secs

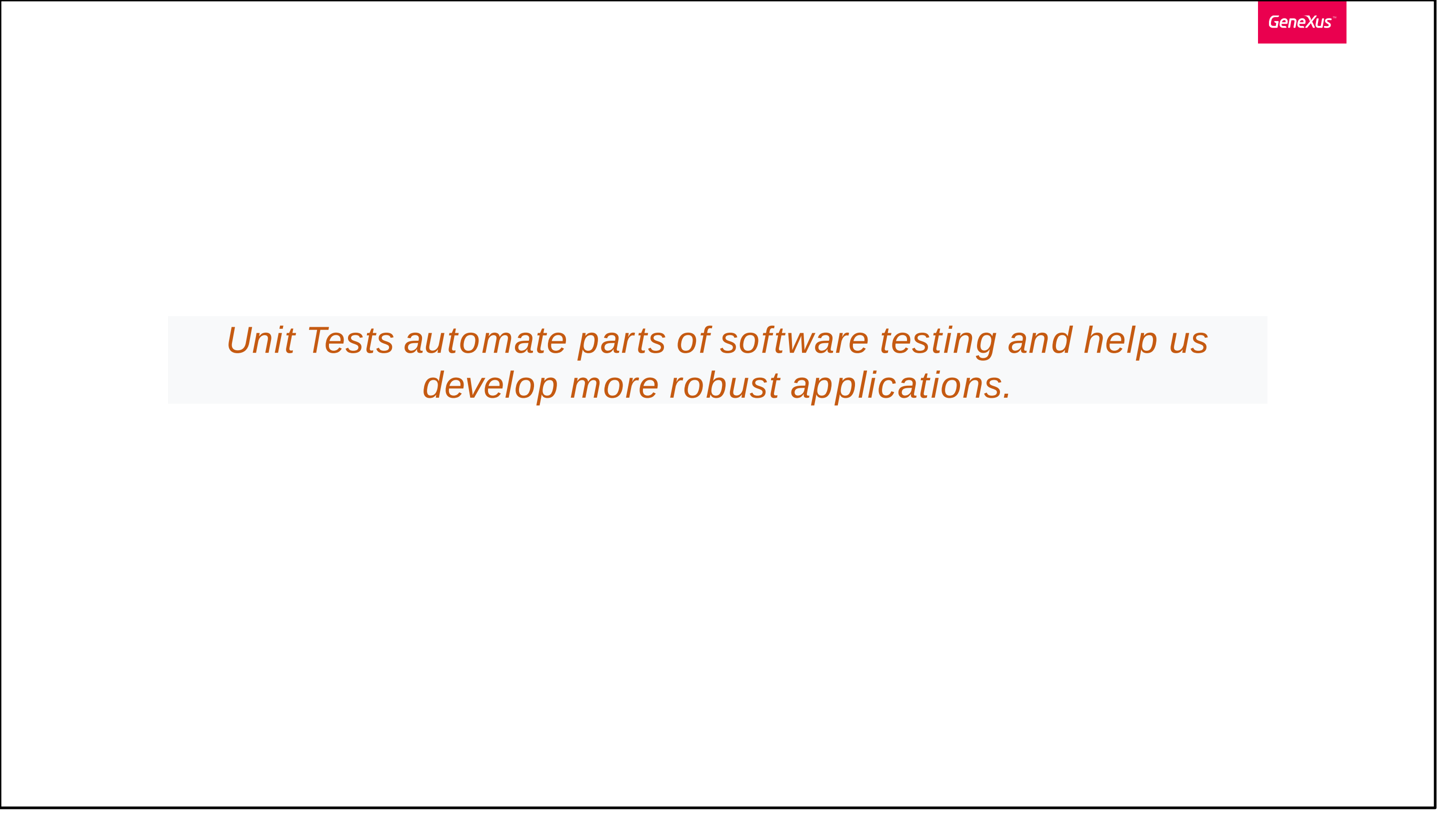
		Expected	Obtained	Info
☐	✓	1210 - The Trip price was updated	1210 - The Trip price was upda...	1.ExpectedUpdateResult:
☐	✓	1980 - The Trip price was updated	1980 - The Trip price was upda...	2.ExpectedUpdateResult:
☐	✓	2530 - Too expensive	2530 - Too expensive	3.ExpectedUpdateResult:
☑	✗	500 - The Trip price was updated	The TripId = 8 is not registered	4.ExpectedUpdateResult:

We are going to run the test again, but first we are going to return the costs to the initial state, and modify the Data Provider assuming that the trip with identifier TripId = 8 exists in the database and we assign it a certain cost.

The idea is to generate a failure in this test set because our procedure will indicate that the trip is not registered. We run the test again by pressing this button, and see that a test set failed since the expected value and the obtained value are different:

From here, we can see the comparison between both results:

And from here we can run again the test sets that failed. We can also export the test result to HTML.



Unit Tests automate parts of software testing and help us develop more robust applications.

Although we have seen a simple example, as we have already said, we can test all kinds of procedures, Data Providers and Business Components. We can perform much more complex tests, using real data from our application (either in a real or simulated database) and cover the validation of a very important part of our application.

The set of unit tests that we build for each functionality automates part of the regression testing, and helps us develop a more robust application.

training.genexus.com
wiki.genexus.com