Unit Test

GeneXus™

Now that we have seen a basic example of how a unit test is carried out in GeneXus, we will be considering a couple of more advanced examples that include interaction with the database.

GeneXus

Procedure that only reads the database

- We will start by the case of a procedure that queries the database without updating it.
- Later we will see the other case, in another demo.

We will start by a procedure that queries the database but does not update it.

PROCEDURE THAT ONLY READS FROM THE DB

We have a procedure called “doLoadCountries”. This is a service we will use in feeding the various country reports in the app. The service receives an SDT called CountryREportOptions as entry parameter. This SDT includes a member where we define the filters by which we will select the countries. For the time being, we will only do it based on the minimum number of attractions that we want the country to have.
There is another element, called SortCriteria where we may define an Order criterion. This is an enumerated element allowing us to return the country list ordered by name or by number of attractions.

This service then receives a filtering criterion and an order criterion and returns a variable called SDTCountryCollection where the collection of countries that fulfill the filter specified is located, and it’s ordered by the filter selected.

The interesting part of implementing this service is that we may perform automatic unit tests that validate the fact that data from reports is correct in all the scenarios we deem relevant for testing. Then we program the reports using this service as our data source instead of accessing the DB directly. At the time of testing, this provides us with confidence as to the correctness of the data reported thanks to the unit test we did previously, and all we need to do is verify that the report format is correct.

This example shows a greater complexity than what we find in the Discount procedure, because the input and output parameters are SDTS instead of simple data. Therefore, the load of test cases will imply a little more work, and also because in this case we will be accessing the database, so, at the time of deciding which values to test, we must know the database that we are testing.

Let's now create the unit test and then we will continue considering these concepts as we move on.
Something important to point out is that objects created for the unit test are nested under the object to be tested, and they are not generated when we do a build-all of the KB. They are generated only when necessary to execute automatic tests.

The fact that they are nested under the procedure simply means that, if we delete the procedure, the associated tests will also be deleted.
Objects are not synchronized in the sense that, if we modify the input or the output parameters of the procedure doloadcountries, we will have to update tests manually as well. And, in fact, we will have to think of the impact of the changes in the input and output parameters in our test cases.

And in relation to test cases, in order to execute our unit test, the first thing to do is define the test cases, or at least one test case with which we will try.
We will see the dataprovider generated automatically and we note that it is somewhat different than the one in the Discount case, where all input and output parameters were simple. In this case, with SDTs , the dataprovider automatically generates a structure that we must complete.

To do that, we may take and drag the SDTs so as to bring the SDT's structure to the dataprovider as we always do, or, in this case, because it's simple, we have the option of writing them.
In our first test case, we will do a test useful for the report that shows the countries with more than 2 attractions, that returns them showing the countries with a greater number of attractions on top. In order to define the values expected we have two options: one is to go to the DB and analyze the values to then enter them here in the results of the test case. Alternatively, what we will do now is to execute the test with an Empty expected result –which, as we already know, will fail. We will see the result obtained and we will validate it manually, and if we are satisfied with those values we will copy them as the expected result.

The interesting thing here is that we may take a photo of the procedure's result, with certain input values, and if the procedure is modified in the future, we will have an easy way to know if nothing was broken, that is: if the same output or the same report data is generated for the same input data.

Beware! Because it's very important to know the DB on which we are testing, and also to have a stable DB to execute the tests so as to avoid having tests that fail due to changes in the DB.
As expected, the test fails. Now we do not see the value expected and the value obtained in the Test-Results screen because the result of our procedure is an SDT –handled here as a json in order to compare it as string. To view it in detail we have the option of expanding the result and seeing it in a text comparer.

In our case, we see that the result obtained is that France and China are the countries with two or more attractions. The value expected is Blank because we didn't define it. If we verify that these are the correct values, we may now set this information as the expected result in our data Provider.
Here we may copy and paste the info to the dataprovider, though the result format is a json, so we will have to change its structure to the correct one in our dataprovider
We execute again, and now we know that our test will be successful.

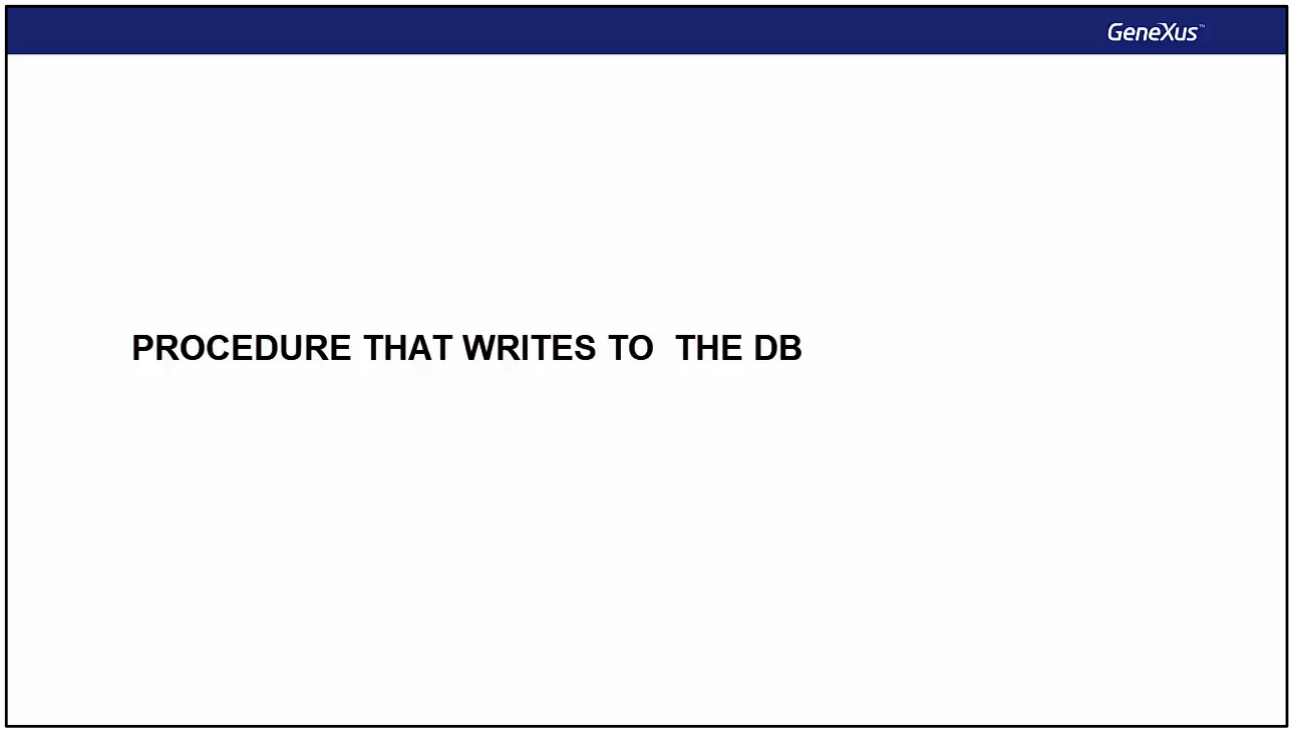
We can go on adding test cases; in this case, the procedure is very simple, so we could add a case to test another order criterion and perhaps an empty filter. In a more complex case with more filtering criteria we would probably need a richer set of tests.
We will not see it in this course, but there is a way to skip the DB so that, provided that nothing changes at the tested procedure level, the execution takes place against a saved state of the DB, though not against the actual DB. This is known as "data mocking". You will find further information on this in our wiki.

GeneXus

Procedure that updates the database

- And now we will consider the case of a procedure that updates the database.

We will now see a second example, where we will be testing a procedure that makes changes to the DB.



To do this, we will see AttractionInsert –a procedure that inserts attractions in our DB.
This procedure receives, as parameters, a data structure with the information about the attraction, then it does the insert and then it returns, as result, a data structure with a result code and a message, in addition to the ID of the attraction created.
Let's take a look at this procedure's implementation.
Something important to consider is that the procedure uses a BC to do the insert, so, when we test this procedure, we're not only testing the procedure itself, but the rules we defined in the BC as well.
To create the test, we do the same as usual, right click and select "Create Unit Test".
Then we will define the test cases as we always do. The input parameter in our procedure was an AttractionSdt, so, since we do not remember its structure, we will look for it in the KB and we will click and drag it to bring the structure here. Then we complete the values according to the case that we want to test. Our first test case will be the happy case, that means that we will use valid values so that a new attraction is successfully included.

And we will do the same for the expected result, which in this case is an InsertResponse. So, we will bring the InsertResponse structure to the

dataprovider and here we defined what we expect. The InsertRESPONSE structure has a result code which in our case we expect that it will be successful. A message that, if the insert is successful, will be empty, and the ID of the attraction that will be inserted.

Note that, here, we leave it empty because the ID to be assigned is an autonumber and we cannot tell in advance the value that will be assigned there. If we executed this test, and assigned a value to it here in the DataProvider, then the test could function on its own the first time –if we knew the next autonumber to be assigned– but the second time we would try to execute it the autonumber would be a different one and the test would fail. So, to avoid that problem, we will have to modify the unit test. In that sense, what we will do is, after invoking the AttractionInsert procedure and obtaining a reply, we will modify our expected value with the value actually obtained. This will ensure that the values will be always the same, that is, the value expected and the value obtained of the ID will always originate in the same place. Therefore, the Assert of only that element will be a pass. But we are interested in the rest, that is: the result code and the message.
Now we execute our test, and while that takes place, we must take note that the only thing we are validating in this test –the only thing on which we are doing Assert- is the result code and the message returned by the procedure, but we are not actually validating that the information is duly recorded in the database.
To do that, we must modify our unit test in order to add an Assert that is verifying the data expected. Now we can execute our test again and we know that we will be validating the data in the base.
Note that our test failed; the fact is that a single key exists for the name and the country of the attraction, so we must include some sort of variation in the attraction's name in order to execute this test as many times as we want. To do that we will just assign a time-stamp to it.
Now we can execute our test again and this time it will be successful.
We can see that we have several Asserts, or several results, one for each Assert done in the unit test.
Now that we have our first test in operation we can continue adding new test cases. For example, to practice any of the validation rules in the transaction, we can see that, in the Attraction transaction, there is an error when the name of the attraction is empty. So, we could create a test case to practice that rule. Here, we already added in our DataProvider a new group of the test cases to test this new case.

So, we leave the attraction name empty, and in the result code we

indicate that we expect a failure. In the message we put the message of the error rule that is not this one returning the transaction –in this case the BC- and in the error message we indicate that we expect a failure when the name of the attraction is empty.

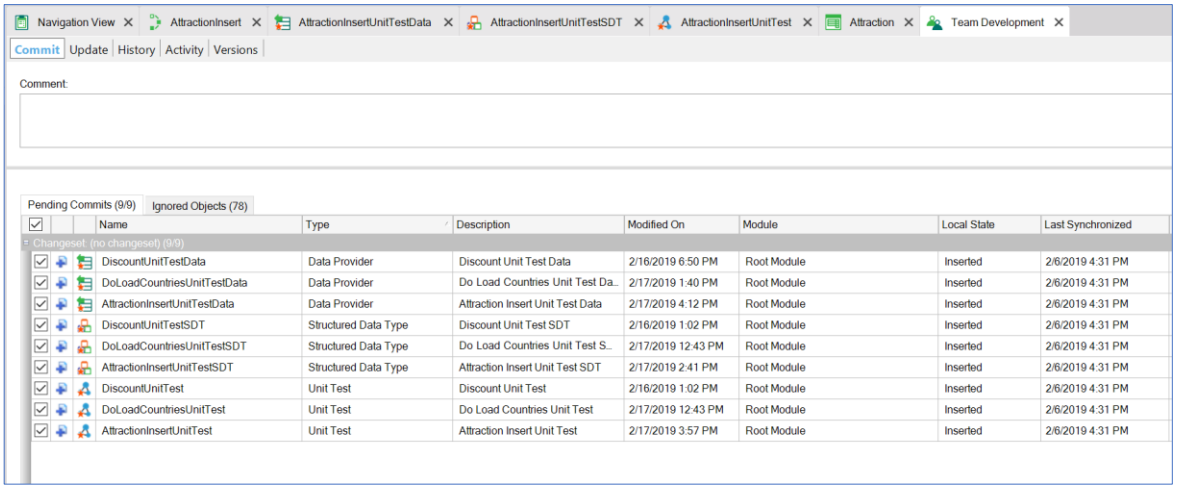
Let's then execute our tests.

Something that we can see as our test is executed is that, among our Asserts, we have a For Each that always awaits for a new attraction to be entered, and this will not only be the case. In sum, in this second test case, we actually expect the BC to pick up an error and that no insert take place. So, we can already know that, since our test is not correct, it will generate a false positive, indicating an error where there isn't any.

What we have to do is fix our unit test for it to do the correct Asserts. In sum, what we will do is validate the data on the table only for the case when the insert has been successful, and we will force a fail in the case of a record in the database when the code of the result of the insert event has not been ok.
On the other hand, we will force the failure in the when none, only when the result code of the insert is OK, because in that case we expect the record to exist.

Here we are doing the IF against the result code obtained upon invoking AttracionInsert. However, it would have been the same to do it with the expected result code since we previously did an Assert validating that they were the same. If they were not, our test would have failed there.

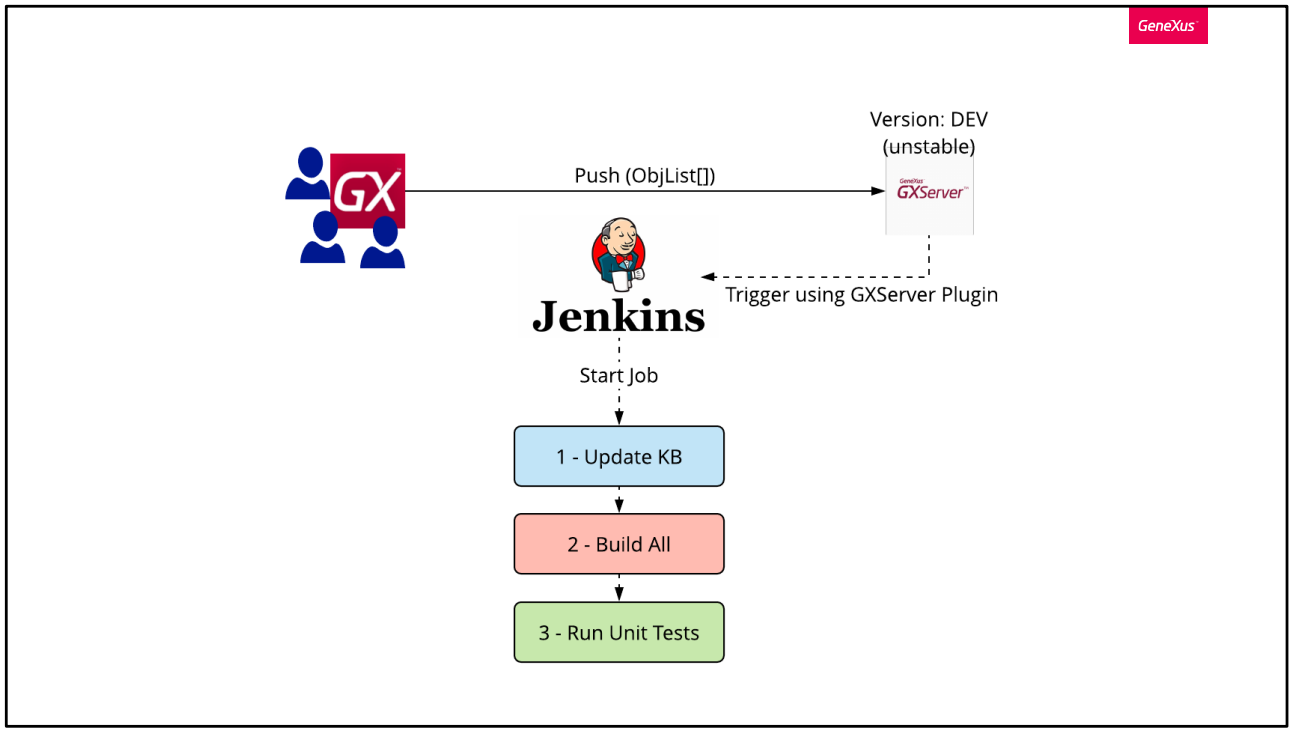
We upload everything to GeneXus Server



So far, we have seen a couple of examples on tests for procedures that interact with the database and we also saw the process that would aid us in creating and perfecting our tests. .

This is useful to us as we develop, for validating the functionality we implement. But, as opposed to when we worked with a screen or messages on the console for testing, the time used in developing the unit tests will now capitalize our KB, because unit tests will continue to be executed repeatedly, thus becoming part of the regression testing.

Tests may be uploaded to the GeneXus Server, and they may be executed in different environments –to the extent that we have a known database, or if we use mocking.
Note that, after tests are shared on the server, the GxTest license will be required.
For further information on licensing check our Wiki.



What's interesting in sharing tests and keeping them on the server is that we may automate their execution as part of the Continued Integration process in Jenkins, so as to avoid publishing a version when errors are encountered and for quickly alerting the team in that respect so that the problem may be promptly solved with lower costs as well.

In our wiki you will find further information on Continued Integration and on how to integrate unit tests in Jenkins.

GeneXus™

training.genexus.com
wiki.genexus.com