

# Testing de aplicaciones en GeneXus

Test Unitario- Avanzado

*GeneXus* 16

Ahora que hemos visto un ejemplo básico de cómo hacer un test unitario en GeneXus, vamos a ver un par de ejemplos más avanzados interactuando con la base de datos.

**Procedimiento que solo lee de la base de datos**

- Comenzaremos viendo el caso de un procedimiento que consulta la base de datos pero no la actualiza.
- Luego veremos el otro caso, en otra demo.



Vamos a comenzar viendo un procedimiento que consulta la base de datos pero no la actualiza.

## PROCEDURE THAT ONLY READS FROM THE DB

[ DEMO: <https://youtu.be/nnOMEWs9OkQ> ]

Tenemos un procedimiento “doLoadCountries” que es un servicio que vamos a utilizar para alimentar los diferentes reportes de países en la aplicación. El servicio recibe como parámetro de entrada un SDT que llamamos CountryREportOptions, este SDT tiene un miembro donde definimos los filtros por los cuales vamos a seleccionar los países. Por ahora solo lo hacemos en base al mínimo numero de atracciones que queremos que el país tenga.

Y luego tiene otro elemento SortCriteria donde podemos definir un criterio de Orden

Esto es un enumerado que nos permite devolver la lista de países ordenado por nombre, o por cantidad de atracciones.

Este servicio entonces recibe un criterio de filtro y uno de orden, y nos devuelve una variable llamada SDTCountryCollection donde esta la colección de países que cumple con el filtro especificado y esta ordenado por el filtro especificado.

La gracia de implementar este servicio es que podemos hacer tests unitarios automáticos, que validan que los datos de los reportes sean los correctos en todos los escenarios que consideremos importante probar, luego programamos los reportes usando este servicio como fuente de datos en vez de acceder directamente a la BD, de esta manera cuando vayamos a testear los reportes ya tenemos confianza de que los datos reportados son los correctos gracias al test unitario que hicimos previamente y solo nos resta ver que el formato del reporte sea el correcto.

Este ejemplo es mas complejo que el procedimiento de Discount porque los parámetros de entrada y salida son SDTS en vez de datos simples y por tanto la carga de los casos de prueba nos va a dar un poco mas de trabajo y también porque en este caso vamos a estar accediendo a la base de datos, por lo cual a la hora de elegir los valores a probar, necesitamos conocer la base de datos que estamos testeando

Vamos a crear el test unitario y seguiremos hablando de estos conceptos a medida que avanzamos

Algo importante a destacar es que los objetos que se crean para el test unitario quedan anidados bajo el objeto a testear y no se generan cuando hacemos un build-all de la KB. Se generan solo cuando es necesario para ejecutar las pruebas automáticas.

El hecho de que queden anidados bajo el procedimiento simplemente significa que si borramos el procedimiento, se van a borrar los tests asociados, los objetos no están sincronizados en el sentido en que si modificamos los parámetros de entrada o los parámetros de salida del procedimiento `doloadcountries` vamos a tener que actualizar a mano también los tests, y de hecho vamos a tener que pensar el impacto de los cambios en los parámetros de entrada y de salida en nuestros casos de prueba..

Y hablando de casos de prueba, para poder ejecutar nuestro test unitario, lo primero es definir cuales son los casos de prueba – o al menos 1 caso de prueba con lo cual queremos probar. Vamos a ver el `dataproducer` que se genere de forma automática y vemos que es un poco diferente que el del caso de `Discount` donde todos los parámetros de entrada y salida eran simples, en este caso como son SDTs el `dataproducer` nos genera automáticamente una estructura que nosotros debemos completar

Para ello podemos pinchar y arrastrar los SDTs de forma de traer la estructura del SDT al `dataproducer` como hacemos siempre, o alternatively en este caso como es simple los vamos a escribir.

En nuestro primer caso de prueba, vamos a testear un caso que sirve para el reporte que muestre aquellos países con mas de 2 atracciones, y los devuelve mostrando los países con mayor cantidad de atracciones arriba. Para definir los valores esperados tenemos 2 opciones.. Una es ir a la DB, analizar los valores y venir e ingresarlos aquí en los resultados del caso de prueba. Alternativamente lo que vamos a hacer ahora es ejecutar la prueba con un resultado esperado Vacío – que sabemos que va a fallar – vamos a ver cual es el resultado obtenido, lo vamos a validar a mano y en caso que estemos satisfechos con esos valores vamos a copiar esos valores como el resultado esperado.

La gracia de esto es que podemos sacar una foto al resultado del procedimiento – dados ciertos valores de entrada – y si en el futuro el procedimiento se modifica, vamos a tener una forma muy fácil de saber si no se rompió nada, es decir , si frente a los mismos datos de entrada, se genera la misma salida o los mismos datos del reporte.

Ojo es super importante conocer la bd sobre la que estamos testeando, incluso tener una BD estable para ejecutar las pruebas, de manera de no tener tests que fallen por cambios en la BD.

Como era de esperar el test falla, ahora no podemos ver en la pantalla `Test-Results` el valor esperado y el valor obtenido porque el resultado de nuestro procedimiento es un SDT – que aquí se esta manejando como un json para poder compararlo como string, para poder verlo en detalle tenemos la opción de expandir el resultado y verlo en un comparador de texto

En nuestro caso vemos que el resultado obtenido es que Francia y China son los países que tienen 2 o mas atracciones, el valor esperado esta en Blanco porque no lo definimos. Si verificamos que estos son los valores correctos podemos ir ahora a poner esta información como resultado esperado en nuestro `data Provider`.

Aquí podemos copiar pegar la info al `dataproducer`, aunque el formato del resultado es un json

asique vamos a tener que darle la estructura correcta en nuestro data-Provider

Volvemos a ejecutar y ahora sabemos que vamos a tener una prueba exitosa

Podemos luego seguir agregando casos de prueba, en este caso el procedimiento es muy sencillo, por lo cual podríamos agregar un caso que probara otro criterio de orden y quizás un filtro vacío.. En un caso mas complejo, que tuviera mas criterio de filtro seguramente tendríamos que tener un juego de tests mas rico.

No vamos a ver en este curso pero existe una forma de saltarse la BD de forma de que mientras nada cambie a nivel de procedimiento testeado se corra contra un estado guardado de la BD pero no contra la BD real. Esto se llama mocking de datos y pueden obtener mas información en nuestro wiki

## Procedimiento que actualiza la base de datos

- Ahora sí, veremos el caso de un procedimiento que actualiza la base de datos.



Vamos a ver un segundo ejemplo, ahora testeando un procedimiento que realiza cambios en la BD.

## PROCEDURE THAT WRITES TO THE DB

[ DEMO: <https://youtu.be/YVK6-jMiGsc> ]

Para esto vamos a ver AttractionInsert q es un procedimiento que inserta atracciones en nuestra BD. Este procedimiento recibe como parámetros una estructura de datos con la información de la atracción, realiza el insert y luego devuelve como resultado una estructura de datos que tiene un código de resultado y un mensaje, además del ID de la atracción creada

Vamos a mirar la implementación de este procedimiento

Algo importante a notar es que el procedimiento utiliza para hacer el insert un BC, por lo tanto cuando testeemos este procedimiento, no solo estamos testando el procedimiento en si, sino también las reglas que hayamos definido en el BC.

Para crea la prueba hacemos lo mismo que siempre, damos botón derecho y seleccionamos "Create Unit Test"

Luego vamos a definir como siempre los casos de prueba. El parámetro de entrada de nuestro procedimiento era un AttractionSdt, entonces como no recordaos la estructura del mismo vamos a buscarlo en la KB y lo pinchamos y arrastramos para traer la estructura aquí. Luego completamos los valores según el caso que queremos probar. Nuestro primer caso de prueba va a ser el caso feliz, es decir vamos a poner valores validos para que se de alta exitosamente una nueva atracción

Vamos a hacer lo mismo para el resultado esperado, que en este caso es un InsertResponse, vamos a buscar entonces la estructura InsertResponse, vamos a traerla para el dataprovider y definir aquí lo que esperamos. La estructura de InsertRESPONSE tiene un código de resultado q en nuestro caso esperamos que sea exitoso, un mensaje que en el caso en que el insert sea exitoso es vacío, y el ID de la atracción que se va a insertar.

Nótese que aquí la dejamos vacía porque el ID a asignarse es un autonumérico y no podemos saber a priori cual va a ser el valor asignado allí. Si ejecutáramos esta prueba y le asignáramos un valor aquí en el DataProvider, la prueba podría funcionar solo la primera vez – si supiéramos cual es el próximo autonumérico a asignar – pero ya la segunda vez que la quisiéramos ejecutar el auto numerado sería diferente y la prueba fallaría, entonces para evitar este problema lo que vamos a tener que hacer es modificar el test unitario

Para ello lo que vamos a hacer es una vez que se haya invocado al procedimiento AttractionInsert y ya hayamos obtenido una respuesta, vamos a modificar nuestro valor esperado con el valor realmente obtenido, de esta manera estos valores van a ser siempre iguales. Es decir, el valor esperado y el valor obtenido del ID van a salir siempre del mismo lugar y por lo tanto el Assert solo de ese elemento va a ser un pass, los que nos interesan son los demás, es decir el código de resultado y el mensaje

Vamos a ejecutar nuestra prueba, y mientras ejecuta una cosa importante a notar es que lo único que estamos validando en este test, es decir, lo único sobre lo cual estamos haciendo Assert es el código de resultado y el mensaje que devuelve el procedimiento pero no estamos validando realmente que este quedando la información bien grabada en la base de datos.

Para hacerlo tenemos que modificar nuestro test unitario, para agregar algún Assert que este verificando los datos que esperamos. Ahora podemos ejecutar nuestro test nuevamente y ahora si sabemos que vamos a estar validando los datos en la base.

Nótese que nuestro test fallo, y es que existe una clave única para el nombre y el país de la atracción, por lo tanto necesitamos introducir algún tipo de variación en el nombre de la atracción para poder ejecutar esta prueba tantas veces como queramos, para ello vamos simplemente a asignarle un time-stamp.

Ahora podemos volver a ejecutar nuestra prueba y esta vez la misma será exitosa. Vemos que tenemos varios Assert, o varios resultados, uno por cada Assert que se esta haciendo en el test unitario

Ahora que tenemos nuestro primer test funcionando podemos seguir agregando nuevos casos de prueba, por ejemplo para ejercitar alguna de las reglas de validación que tenga la transacción, podemos ver por ejemplo que en la transacción Attraction se da un error cuando el nombre de la atracción es vacío, por tanto podríamos crear un caso de prueba que ejerce esta regla.

Aquí ya agregamos un nuevo grupo en nuestro DataProvider de los casos de prueba para probar este nuevo caso.

Dejamos entonces el nombre de la atracción vacía, en el código de resultado indicamos que esperamos que falle, en el mensaje ponemos el mensaje de la regla de error que no es esta devolviendo la transacción – en este caso el BC – y en el mensaje de error indicamos que esperamos que falle cuando el nombre de la atracción esta en blanco

Vamos a ejecutar entonces nuestra prueba

Algo que ya podemos ir viendo mientras nuestro test ejecuta, es que entre nuestros Assert tenemos un For Each que siempre espera q se haya dado de alta una atracción y esto no siempre va a ser el caso. En definitiva en este segundo caso de prueba en realidad esperamos que el BC levante un error y no se haga un insert, por lo tanto ya podemos saber que como nuestro test no es correcto, va a generar un falso positivo, es decir, va a indicar un error donde realmente no lo hay



Lo que tenemos que hacer es arreglar nuestro test unitario para que haga los Assert correctos, en definitiva vamos a validar los datos en la tabla solo para el caso en que el insert haya sido exitoso y vamos a forzar un fallo en caso que haya un registro en la base de datos cuando el código del resultado del evento insert no haya sido ok, Por otro lado vamos a forzar el fallo en el when none solo cuando el código de resultado del insert fue OK ya que en ese caso esperamos que el registro exista.

Acá estamos haciendo el IF contra el código de resultado obtenido al invocar AttracionInsert pero hubiera sido igual hacerlo con el código de resultado esperado, ya que previamente hicimos un Assert validando que fueran iguales y en el caso de no serlo nuestro test ya hubiera fallado por ahí.

## Subimos todo a GeneXus Server

Navigation View X AttractionInsert X AttractionInsertUnitTestData X AttractionInsertUnitTestSDT X AttractionInsertUnitTest X Attraction X Team Development X

Commit Update History Activity Versions

Comment:

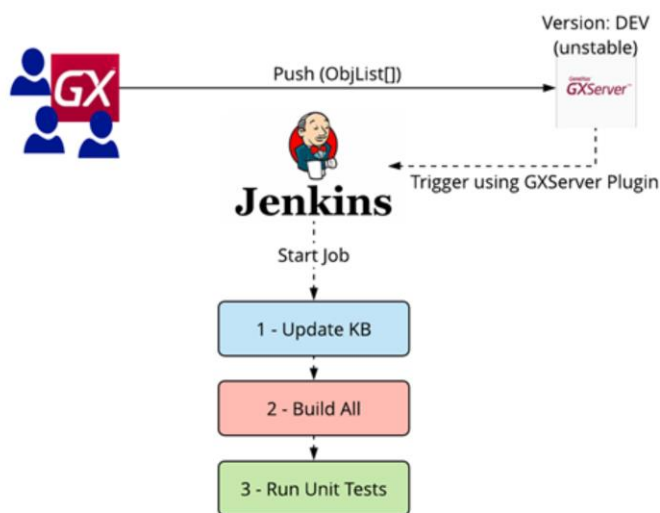
Pending Commits (9/9) Ignored Objects (78)

<input checked="" type="checkbox"/>	Name	Type	Description	Modified On	Module	Local State	Last Synchronized
<input checked="" type="checkbox"/>	DiscountUnitTestData	Data Provider	Discount Unit Test Data	2/16/2019 6:50 PM	Root Module	Inserted	2/6/2019 4:31 PM
<input checked="" type="checkbox"/>	DoLoadCountriesUnitTestData	Data Provider	Do Load Countries Unit Test Da..	2/17/2019 1:40 PM	Root Module	Inserted	2/6/2019 4:31 PM
<input checked="" type="checkbox"/>	AttractionInsertUnitTestData	Data Provider	Attraction Insert Unit Test Data	2/17/2019 4:12 PM	Root Module	Inserted	2/6/2019 4:31 PM
<input checked="" type="checkbox"/>	DiscountUnitTestSDT	Structured Data Type	Discount Unit Test SDT	2/16/2019 1:02 PM	Root Module	Inserted	2/6/2019 4:31 PM
<input checked="" type="checkbox"/>	DoLoadCountriesUnitTestSDT	Structured Data Type	Do Load Countries Unit Test S..	2/17/2019 12:43 PM	Root Module	Inserted	2/6/2019 4:31 PM
<input checked="" type="checkbox"/>	AttractionInsertUnitTestSDT	Structured Data Type	Attraction Insert Unit Test SDT	2/17/2019 2:41 PM	Root Module	Inserted	2/6/2019 4:31 PM
<input checked="" type="checkbox"/>	DiscountUnitTest	Unit Test	Discount Unit Test	2/16/2019 1:02 PM	Root Module	Inserted	2/6/2019 4:31 PM
<input checked="" type="checkbox"/>	DoLoadCountriesUnitTest	Unit Test	Do Load Countries Unit Test	2/17/2019 12:43 PM	Root Module	Inserted	2/6/2019 4:31 PM
<input checked="" type="checkbox"/>	AttractionInsertUnitTest	Unit Test	Attraction Insert Unit Test	2/17/2019 3:57 PM	Root Module	Inserted	2/6/2019 4:31 PM

Hasta aquí hemos visto un par de ejemplos de tests sobre procedimientos que interactúan con la base de datos y hemos visto el proceso por el cual iríamos creando y puliendo nuestros tests.

De esta forma, nos sirve mientras desarrollamos, para validar la funcionalidad que estamos implementando. Pero a diferencia de cuando hacíamos alguna pantalla o mensajes en consola para testear, el tiempo invertido en desarrollar los test unitarios capitaliza nuestra KB ya que los tests unitarios van a poder seguirse ejecutando de manera repetible – constituyendo parte de las pruebas de regresión.

Los tests pueden subirse al GeneXus Server y se pueden ejecutar en diferentes ambientes – mientras que la base de datos sea conocida, o estemos utilizando mocking -. Nótese que una vez que los test se comparten en el server se necesita licencia de GxTest . Para mas información sobre licenciamiento consulte el Wiki.



<https://wiki.genexus.com/comm/wiki/servlet/wiki?38332,CI+%2F+CD+integration+for+Unit+Tests>

La gracia de compartir los tests y tenerlos en el server, es que podemos automatizar la ejecución de los mismos como parte del proceso de Integración Continua en Jenkins, de manera de evitar publicar una versión si se descubren errores y alertar de los mismos rápidamente al equipo para que pueda resolverse el problema de forma temprana y por tanto con menor costo.

Pueden ver en nuestro wiki mas información sobre Integración Continua y cómo integrar las pruebas unitarias en Jenkins.



Videos

[training.genexus.com](http://training.genexus.com)

Documentation

[wiki.genexus.com](http://wiki.genexus.com)

Certifications

[training.genexus.com/certifications](http://training.genexus.com/certifications)