

## DATA PROVIDERS

### Aportes sobre su lenguaje y conclusiones

*GeneXus® 16*

Introduciremos nuevos conocimientos sobre el uso de los Data Providers.

### Características de un Data Provider

**Output:**

- SDT simple
- SDT colección
- BC simple
- BC colección

**Parámetros:**

- Variable
- Atributo

**Origen de los datos:**

- Datos fijos
- Datos provenientes de la BD:
  - Para cargar un SDT: de una o varias tablas
  - Para cargar un BC:
    - traídos de la misma tabla
    - de otra tabla

Recordemos que el objetivo de un Data Provider es devolver cargada una estructura de datos en memoria (que puede ser colección o no). Para lograrlo nos provee un lenguaje declarativo que pone el foco en la estructura de la salida, de modo que básicamente sólo debemos indicar cómo obtener cada uno de esos elementos de información.

Puede usarse para cargar un SDT simple o colección, o una estructura de Business Component, tanto simple como colección.

Como cualquier otro objeto, un Data Provider puede recibir parámetros (tanto variables como atributos). Pero a diferencia de los demás objetos, precisamente debido al objetivo de permitir al desarrollador concentrarse en la salida, ésta no se declara en la regla parm sino explícitamente como propiedad Output del objeto.

Vimos que los datos que se usan para cargar las estructuras pueden ser datos fijos, o pueden ser variables, extraídos de una o varias tablas de la base de datos. En particular, cuando estamos cargando una estructura del tipo business component, los datos pueden ser de la tabla asociada a la transacción sobre la que se definió el BC, o de otra tabla de la base de datos.

## Inicializar una tabla con datos fijos

- Crear registros en tabla CATEGORY

```
CategoryCollection
{
  Category
  {
    CategoryName = "Museum"
  }
  Category
  {
    CategoryName = "Monument"
  }
  Category
  {
    CategoryName = "Tourist site"
  }
}
```

Este grupo se agrega por claridad, no es necesario si usamos la propiedad Collection=True

El data provider **no tiene tabla base** porque son datos fijos

Estos no son nombres de atributos sino de los elementos del business component basado en la transacción Category

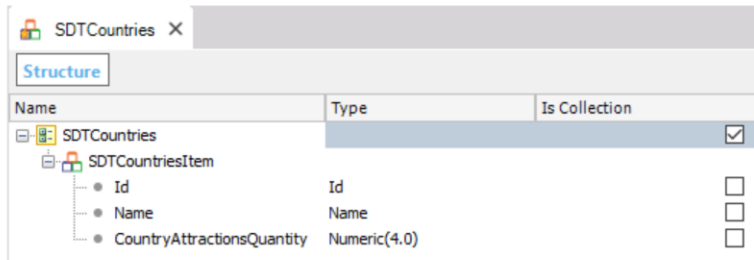
Aquí estamos inicializando una estructura de business components de Category, utilizando un data provider para ello.

Observemos que repetimos los grupos, uno por cada categoría que se va a crear. El grupo CategoryCollection podríamos no definirlo, ya que como nuestra intención es devolver una colección de elementos Category, ya setearmos la propiedad Collection del data provider, en el valor True.

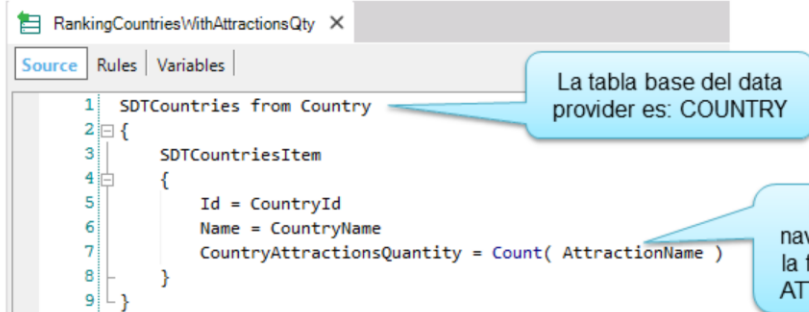
Otra cosa a observar es que como el data provider no tendrá que recorrer ninguna tabla para obtener los datos, debido a que los datos se lo estamos dando nosotros como valores fijos. Por lo tanto, este data provider no tendrá tabla base y no será necesario utilizar una cláusula "from".

Por último, es importante notar que los elementos CategoryName que están del lado izquierdo de las asignaciones, no son los atributos de la transacción Category, sino los elementos del business component basado en la transacción Category, que estamos cargando mediante el data provider.

### Datos provenientes de la base de datos Carga de un SDT de una o varias tablas: Ranking de países



Name	Type	Is Collection
SDTCountries		<input checked="" type="checkbox"/>
SDTCountriesItem		
Id	Id	<input type="checkbox"/>
Name	Name	<input type="checkbox"/>
CountryAttractionsQuantity	Numeric(4,0)	<input type="checkbox"/>



```
1 SDTCountries from Country
2 {
3   SDTCountriesItem
4   {
5     Id = CountryId
6     Name = CountryName
7     CountryAttractionsQuantity = Count( AttractionName )
8   }
9 }
```

En este ejemplo vemos cómo podemos cargar un SDT con datos de varias tablas. Nuestro objetivo es construir un ranking de países por la cantidad de atracciones de cada país.

Para eso, definimos un SDT colección, para almacenar el identificador, el nombre y la cantidad de atracciones de cada país; para cargar este SDT utilizaremos un data provider.

Para obtener los datos de los países el data provider recorre la tabla COUNTRY y para cada país, la fórmula count navega por la tabla ATTRACTION para contar las atracciones de ese país.

Una vez obtenida la colección, podemos ordenar la misma en forma descendente por la cantidad de atracciones.

### Datos provenientes de la base de datos

- Carga de un business component de una única tabla

```
Country from Country
{
    CountryId = CountryId
    CountryName = CountryName
    CountryFlag = CountryFlag
}
```

La tabla base del data provider es: COUNTRY

```
Country from Country
{
    CountryId
    CountryName
    CountryFlag
}
```

Como los elementos del business components se llaman igual que los atributos, es posible usar notación abreviada

En este ejemplo estamos cargando en memoria los datos de los países.

## Datos provenientes de la base de datos

- Carga de un business component con datos de una tabla diferente

```
ServiceCard from Customer
{
    ServiceCardCardType = Type.Full if count(TripId)>3; Type.Partial otherwise
    CustomerId = CustomerId
}
```

La tabla base del  
data provider es:  
**CUSTOMER**

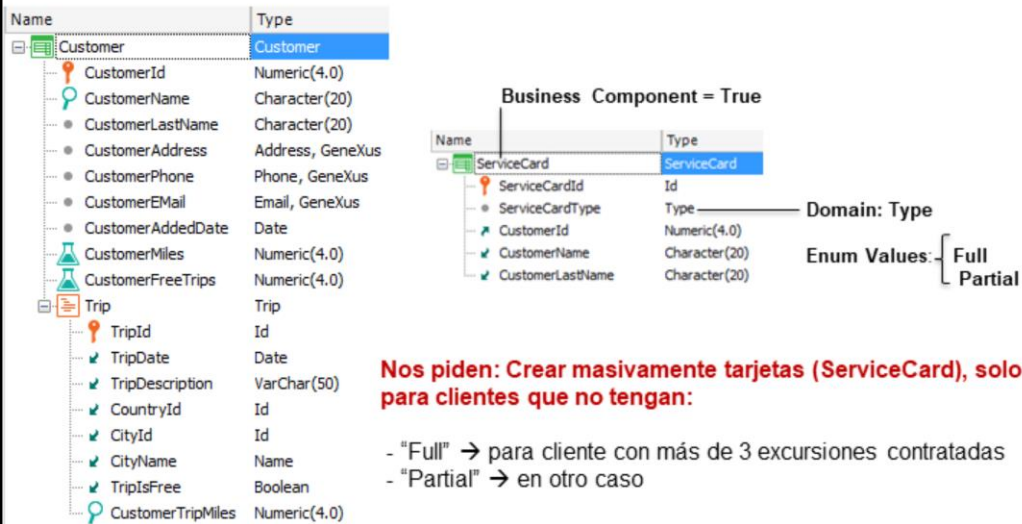
Los elementos son de un  
business component de la  
transacción **SERVICECARD**

En este ejemplo se desea otorgar una tarjeta especial (del tipo total o parcial) a aquellos clientes que han contratado más de 3 excursiones, para lo cual recorreremos los clientes para contar la cantidad de atracciones y asignarle la tarjeta correspondiente.

Para eso mediante un data provider cargamos una estructura de business component de la transacción ServiceCard, para luego recorrer dicha colección y guardar en la base de datos esta información.

Este ejemplo lo desarrollaremos a continuación.

## Ejemplo

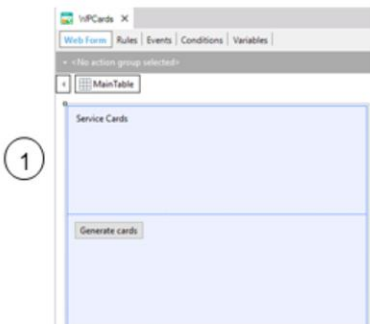


Supongamos que la agencia de viajes decide que todos los clientes que han contratado más de 3 excursiones, recibirán una tarjeta especial de tipo "Full Services", que les permitirá disfrutar de todos los servicios en forma gratuita. Y en caso de haber contratado menos de 3 excursiones, recibirán la tarjeta de tipo "Partial Services".

Observemos las transacciones con las cuales contamos. La transacción "Customer" y la transacción "ServiceCard" que define a las tarjetas.

Cada tarjeta tiene un identificador que se autonumera, un cliente y hemos definido el atributo ServiceCardType basado en el dominio Type, enumerado (que admite solamente los valores "Full" o "Partial").

Aportes sobre su lenguaje y conclusiones
GeneXus®



Solución...

2 Proponemos un **Data Provider** que cargue y devuelva el conjunto de tarjetas a ser generadas:

Business Component = True

Name	Type
ServiceCard	ServiceCard
ServiceCardId	Id
ServiceCardType	Type
CustomerId	Numeric(4,0)
CustomerName	Character(20)
CustomerLastName	Character(20)

Arrastramos la transacción al source del Data Provider

----->

```
ServiceCard from Customer
{
  ServiceCardId =
  ServiceCardCardType =
  CustomerId =
}
```

No son atributos sino elementos de la estructura que se cargará en memoria.

Hemos definido el web panel WPCards, que simplemente ofrece un botón que disparará el proceso automático de generación y visualización de nuevas tarjetas.

¿Qué deberá suceder al presionar el botón?

Para todos aquellos clientes que aún no tengan tarjeta emitida, se les deberá crear una tarjeta del tipo que corresponda, teniendo en cuenta la cantidad de excursiones que han contratado a la agencia.

Propondremos una solución **usando un Data Provider** que cargue y devuelva la colección de tarjetas a ser generadas. **Y luego recorreremos la colección y grabaremos las tarjetas en la base de datos.**

¿Cómo hacemos esto?

En primer lugar, configuramos la transacción ServiceCard como Business Component, para grabar las tarjetas haciendo uso del concepto de Business Component.

Después creamos un objeto Data Provider de nombre DPCards y arrastramos la transacción ServiceCard hacia el source del Data Provider.

Como ya hemos visto en videos anteriores, esto definirá una estructura en memoria con ítems de igual nombre y tipo que los atributos de la transacción definida como Business Components.

Ahora necesitamos recorrer la tabla CUSTOMER, filtrar los clientes que aún no tienen una tarjeta emitida **y para ellos** agregar una tarjeta en la colección.

La tabla que navegará el Data Provider se determina por la transacción base definida en la cláusula from, en este caso CUSTOMER.



Observemos que la tabla que recorreremos para obtener los datos, no es la misma tabla asociada a la transacción sobre la cual fue definida el Business Components, es decir SERVICECARD.

Solución...

3

```
ServiceCard from Customer
{
  ServiceCardId =
  ServiceCardCardType =
  CustomerId =
}
```

→ **ServiceCardId** basado en dominio  
autonumerado

... la estructura del Data Provider quedará...

```
ServiceCard from Customer
{
  ServiceCardCardType = Type.Full if count(TripId)>3; Type.Partial otherwise
  CustomerId = CustomerId
}
```

También puede utilizarse un procedimiento que devuelva un valor

Tabla base del DP:  
**CUSTOMER**

Al ítem CustomerId es claro que le vamos a asignar el atributo CustomerId.

Al ítem ServiceCardId no le necesitamos asignar un valor específico, porque recordemos que esta estructura a ser cargada fue arrastrada de una transacción declarada como Business Components y por lo tanto este ítem está basado en el atributo ServiceCardId, que pertenece al dominio Id y su propiedad Autonumber está configurada en Yes.

Veamos que a ServiceCardType le podemos asignar el valor que retorne una fórmula condicional, es decir que la fórmula devolverá el tipo "Full" o "Partial" de acuerdo a la cantidad de excursiones que ha contratado el cliente. También podríamos haber utilizado un procedimiento que calcule y devuelva este valor.

Para determinar la tabla base que navegará el data provider, GeneXus se fija en la transacción base que agregamos con la cláusula from, por lo tanto la tabla que recorrerá será la asociada a dicha transacción, en este caso la tabla CUSTOMER.

GeneXus también verificará que los atributos que agreguemos del lado derecho de los signos de asignación pertenezcan a la tabla extendida de CUSTOMER, de lo contrario se producirá un error que veremos reportado en el listado de navegación del data provider.

Notemos que los atributos que están dentro de fórmulas, no se tienen en cuenta para esta verificación, ya que los mismos solamente son tomados en cuenta para determinar la tabla a ser navegada por la fórmula.

Observemos que al elemento correspondiente al atributo autonumerado ServiceCardId, no es necesario asignarle valor debido a que lo toma automáticamente por la autonumeración del atributo. De igual modo, cualquier elemento del business component al que no deseamos cargarle valor podemos dejarlo sin asignar y cuando cree el registro en la tabla, el atributo quedará con un

valor vacío.

En este caso se aplican las restricciones a las claves foráneas, a la cuales deberemos asignar un valor a no ser que hayamos definido el atributo como nullable.

Solución..

... pero solamente queremos recorrer los clientes que aún no tienen tarjeta...

```
ServiceCard from Customer
Where count(ServiceCardType) = 0
{
    ServiceCardCardType = Type.Full if count(TripId)>3; Type.Partial otherwise
    CustomerId = CustomerId
}
```

... simplifiquemos...

```
ServiceCard from Customer
Where count(ServiceCardType) = 0
{
    ServiceCardCardType = Type.Full if count(TripId)>3; Type.Partial otherwise
    CustomerId
}
```

... analicemos la salida...

Output

Infer Structure	No
Output	ServiceCard
Collection	True
Collection Name	Cards

Se configuró automáticamente al arrastrar la trn

Recordemos que no queremos navegar todos los clientes y para cada uno cargar una tarjeta, sino que queremos navegar solamente aquellos clientes que aún no tienen tarjeta.

Los data providers permiten en su sintaxis, que les incluyamos todas las cláusulas permitidas en el For each, así que agregamos la cláusula Where mostrada en la diapositiva.

Dado que el único atributo referenciado en el Where, está dentro de una fórmula, como ya hemos dicho, no será incluido en la verificación de si pertenece o no a la tabla extendida de Customer.

Observemos que en lugar de poner CustomerId = Customer Id, pusimos simplemente CustomerId. Recordemos que el CustomerId a la izquierda de la asignación es el elemento de la estructura que se cargará en memoria y el de la derecha el atributo que le dará el valor. Como se llaman igual, podemos utilizar la **notación abreviada** y escribir solamente CustomerId.

Ahora analicemos la salida, es decir qué devuelve el data provider. Al haberse arrastrado la transacción ServiceCard sobre el source, la propiedad Output quedó asociada automáticamente al business component ServiceCard asociado a la transacción.

Y la propiedad Collection? La estructura que estamos cargando no representa una colección. Solamente representa una instancia en memoria, con la estructura de la transacción ServiceCard. Sin embargo necesitamos obtener una **colección de tarjetas generadas**, entonces configuramos la propiedad Collection con valor True... e indicamos un nombre para la colección que devolverá cargada este data provider.

GeneXus

Aportes sobre su lenguaje y conclusiones

Solución..

Invocando al Data Provider...

Event 'Generate cards'

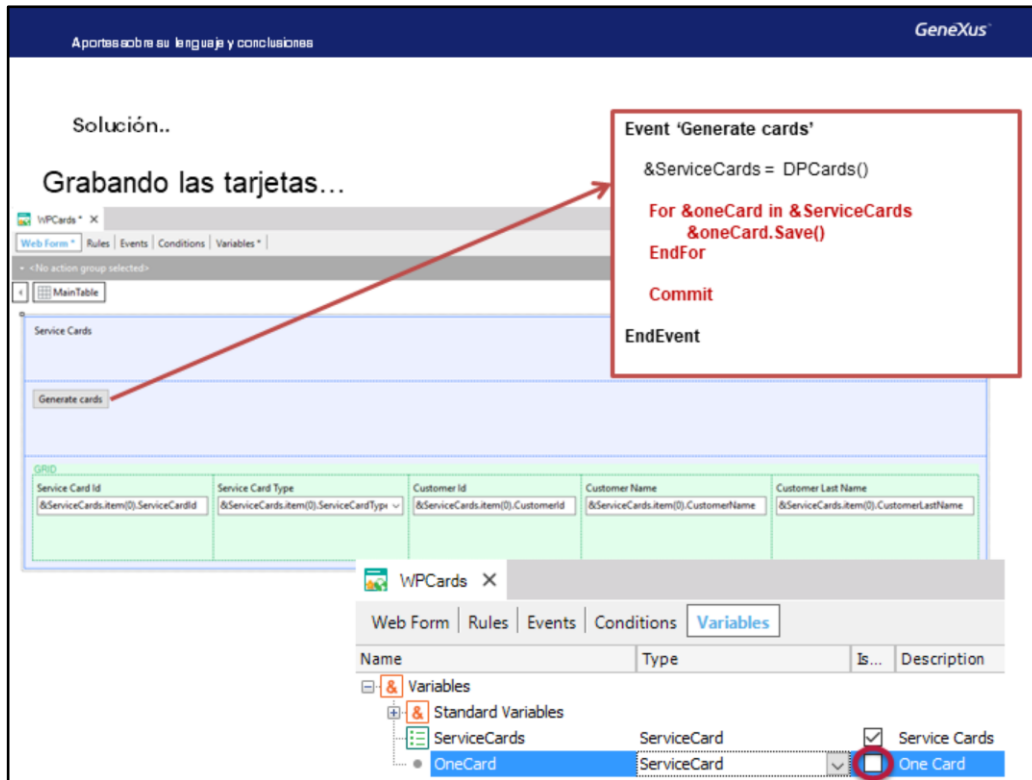
```
&ServiceCards = DPCards()
EndEvent
```

The screenshot shows the GeneXus IDE interface. At the top, there's a dark blue header with the text 'Aportes sobre su lenguaje y conclusiones' on the left and 'GeneXus' on the right. Below the header, the main workspace is divided into two panes. The top pane shows the 'Event 'Generate cards'' with the code: `&ServiceCards = DPCards()` followed by `EndEvent`. A red arrow points from the 'Generate cards' button in the bottom pane to this code. The bottom pane shows a web form titled 'WPCards \* X' with tabs for 'Web Form \*', 'Rules', 'Events', 'Conditions', and 'Variables \*'. The 'Variables \*' tab is active, showing a table with columns 'Name', 'Type', 'Is...', and 'Description'. It lists 'ServiceCards' as a 'Standard Variable' of type 'ServiceCard'. Below the variables, there's a 'GRID' section titled 'Service Cards' containing a table with columns: 'Service Card Id', 'Service Card Type', 'Customer Id', 'Customer Name', and 'Customer Last Name'. Each column has a corresponding data binding expression: `&ServiceCards.item(0).ServiceCardId`, `&ServiceCards.item(0).ServiceCardType`, `&ServiceCards.item(0).CustomerId`, `&ServiceCards.item(0).CustomerName`, and `&ServiceCards.item(0).CustomerLastName`.

Volvamos ahora al web panel para invocar al data provider.

En el evento 'Generate cards' asociado al botón, le asignamos a una variable (&ServiceCards) definida como una colección de tarjetas (ServiceCard), lo que devolverá el data provider.

En el form del web panel, insertamos la variable &ServiceCards. Por tratarse de una colección, automáticamente GeneXus entiende que debe mostrar el contenido en un grid.



Ahora, ¿esto alcanza para que las tarjetas devueltas por el data provider efectivamente se graben en la tabla SERVICECARD asociada a la transacción ServiceCard?

No. Por ahora las tarjetas están cargadas en memoria y hemos mostrado el contenido de la colección.

Cuando estudiamos la utilización de los business components, vimos que para grabar debemos usar el método Save y luego ejecutar Commit. Así que nos está faltando recorrer la colección devuelta por el data provider y proceder a grabar cada elemento de la colección como registro en la tabla física. Y posteriormente a la grabación de todas las tarjetas, declaramos el comando Commit.

Para recorrer la colección de tarjetas devuelta por el data provider, empleamos el comando **For elemento in Colección**. Esta variable &oneCard debe ser definida como el tipo business component ServiceCard y representa a cada elemento que se va iterando de la colección.

Aportes sobre su lenguaje y conclusiones
GeneXus®

Solución mejor

## Grabando las tarjetas...

The screenshot shows the GeneXus IDE interface. At the top, there's a tab labeled 'MainTable'. Below it, a 'Service Cards' section contains a 'Generate cards' button. Below that is a data grid with the following columns and data sources:

Service Card Id	Service Card Type	Customer Id	Customer Name	Customer Last Name
&ServiceCards.item(0).ServiceCardId	&ServiceCards.item(0).ServiceCardType	&ServiceCards.item(0).CustomerId	&ServiceCards.item(0).CustomerName	&ServiceCards.item(0).CustomerLastName

**Event 'Generate cards'**

```

&ServiceCards = DPCards()

If &ServiceCards.Insert()
    Commit
endif

EndEvent
          
```

Sin embargo, recordemos que tenemos el método Insert de una variable colección de Business Components, que ya hace automáticamente lo que hicimos antes manualmente.

Y no solo eso, sino que además devuelve True si todas las inserciones de la colección fueron exitosas, y False en caso contrario. De este modo podemos hacer el commit si todo fue exitoso. Si no, tendríamos que proceder a consultar los mensajes de error y tomar las acciones que consideremos pertinentes.

Solución..

## En ejecución...

Application Name

Recents Customer — WPCards

### Service Cards

Generate cards

Service Card Id	Service Card Type	Customer Id	Customer Name	Customer Last Name
1	Full	1	John	Smith
2	Partial	2	Susan	Brown
3	Partial	3	Ayra	Smart
4	Partial	4	Robert	Hill

1) Se presiona el botón.

**Resultado:** Se muestran las tarjetas generadas.

2) Se presiona nuevamente el botón.

**Resultado:** No se generan tarjetas.

Application Name

Recents Customer — WPCards

### Service Cards

Generate cards

Service Card Id	Service Card Type	Customer Id	Customer Name	Customer Last Name
-----------------	-------------------	-------------	---------------	--------------------

Ahora sí el desarrollo de lo que nos solicitaron está completo. Ejecutamos el web panel y presionamos el botón. Vemos en la grilla la lista de tarjetas que se generaron.

Puede surgirnos la pregunta: ¿Qué sucederá si volvemos a presionar el botón "Generate Cards"?

¿Se volverán a crear las tarjetas para los mismos clientes?

No, porque en el data provider filtramos que solamente queríamos navegar los clientes sin tarjetas.

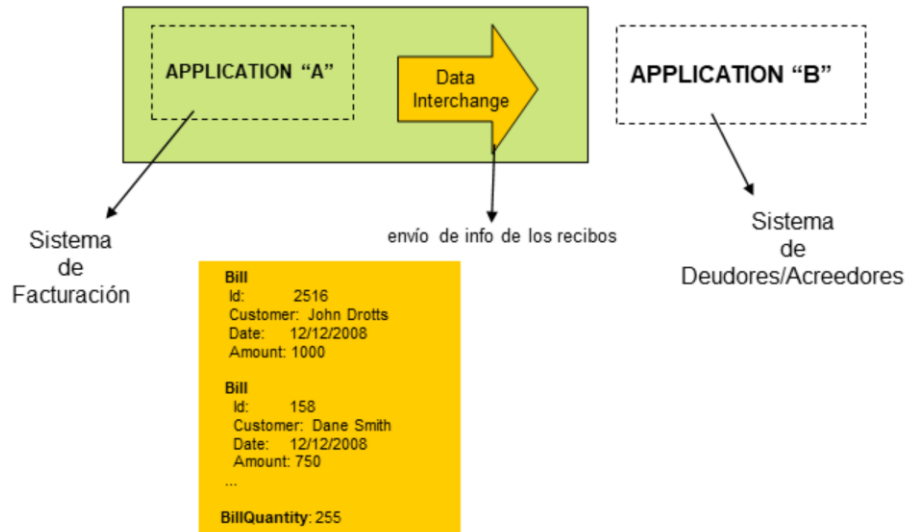
Vale mencionar que hay otras soluciones para resolver el mismo requisito en GeneXus. **Con esta implementación hemos usado el concepto de Business Component para actualizar la base de datos y hemos combinado su uso con el hecho de cargar previamente una estructura colección en memoria con los datos a grabar.**

**El uso de un Data Provider para esto, es muy sencillo y nos ahorra escribir código explícito.**



## Sobre el lenguaje

- Escenario: intercambio de información jerárquica entre módulos de una misma aplicación.

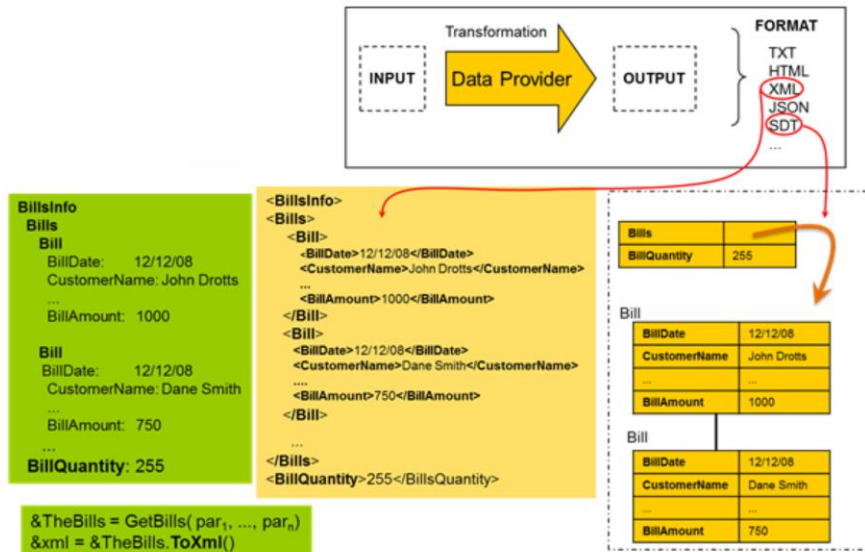


Supongamos que el sistema de la agencia de viajes está dividido en módulos para manejar el sistema de facturación y el sistema de deudores/acreedores. Le queremos enviar desde el sistema de facturación al sistema de deudores y acreedores, un listado de los recibos correspondientes a cierto período de facturación (es decir, para un período determinado se desea sumarizar para cada cliente el importe total que se le ha facturado y generarle un recibo).

Se trata de información jerárquica (enviaremos info de recibos, cada uno de los cuáles tiene determinados datos).

Los formatos más usuales de intercambio de información jerárquica suelen ser Xml, y json, pero hay más.

## Sobre el lenguaje



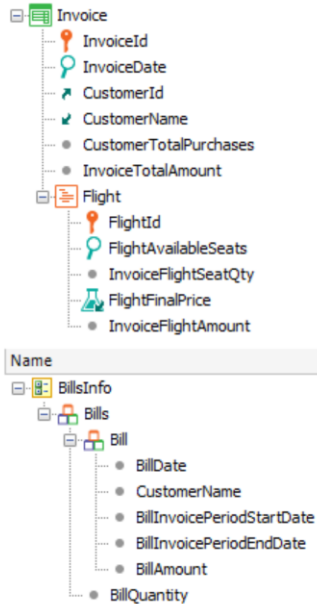
Recordemos que con un Data Provider el foco está ubicado en el lenguaje de salida: se indica en una estructura jerárquica de qué se compone ese Output.

Luego, para cada elemento de la estructura jerárquica habrá que indicar en el Source del Data Provider cómo se calcula.

Una misma información estructurada podrá representarse utilizando los diferentes formatos existentes.

Esa es la idea del Data Provider. Si en el futuro aparece un nuevo formato de representación de información estructurada, el Data Provider continuará invariable... GeneXus implementará el método de transformación a ese formato, y solo habrá que utilizarlo.

## Sobre el lenguaje



GetBills \* X

Source \* Rules Variables \*

```
1 BillsInfo
2 {
3     Bills from Customer
4     {
5         &quantity = 0
6         Bill
7         {
8             BillDate = &Today
9             CustomerName
10            BillInvoicePeriodStartDate = &start
11            BillInvoicePeriodEndDate = &end
12            BillAmount = sum(InvoiceTotalAmount,
13                           InvoiceDate>0&start
14                           and InvoiceDate<=&end)
15            &quantity = &quantity + 1
16        }
17        BillQuantity = &quantity
18    }
19 }
```

Output: BillsInfo  
Collection: False

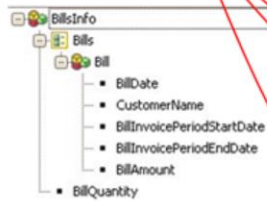
&TheBills = GetBills(&start, &end)

Vemos que en este caso estamos utilizando una estructura más compleja (un SDT que tiene un elemento Quantity y una colección de Bills).

## Sobre el lenguaje

## • Componentes básicos:

- Grupos
- Elementos
- Variables



```
BillsInfo  
{  
  Bills  
  {  
    Bill  
    {  
      BillDate = &today  
      CustomerName = CustomerName  
      BillInvoicePeriodStartDate = &start  
      BillInvoicePeriodEndDate = &end  
      BillAmount = sum( InvoiceTotal, ... )  
      &quantity = &quantity + 1  
    }  
  }  
  BillQuantity = &quantity  
}
```

Aquí podemos identificar los componentes básicos del lenguaje de Data Providers.

## Sobre el lenguaje

- Un grupo repetitivo es análogo a un for each:
  - Determina tabla base (de igual forma que en un for each)
  - Tiene disponibles las mismas cláusulas que para un for each:

```

BillsInfo
{
  Bills
  {
    Bill
    {
      BillDate = &today
      CustomerName = CustomerName
      BillInvoicePeriodStartDate = &start
      BillInvoicePeriodEndDate = &end
      BillAmount = sum( InvoiceTotal, ...)
      &quantity = &quantity + 1
    }
  }
  BillQuantity = &quantity
  &quantity = 0
}

```

```

from BaseTransaction
[skip expr1] [count expr2]
[[[order] order_attributesi [when condi]]... | [order none] [when condx]]
[using DataSelectorName([[parm1 [.parm2 [. ...]]]])]
unique att1, att2,...,attn
[[where {conditioni when condi} |
[attribute IN DataSelectorName([[parm1 [.parm2 [. ...]]]])]...]]

```

En el ejemplo el grupo de nombre Bill será repetitivo. ¿Por qué? Para contestar la pregunta, hagamos otra: ¿y si fuera un for each donde los elementos de la izquierda de las asignaciones corresponden a los distintos elementos de una variable SDT? En este caso la presencia de *CustomerName* (a la derecha de la segunda asignación) permite afirmar que hay tabla base. Es que queremos iterar sobre la tabla CUSTOMER. Por lo que escribiremos la cláusula “from Customer” análogamente a lo que haríamos en un for each.

Obsérvese que el grupo de nombre BillsInfo, en cambio, no será repetitivo, no tiene cláusulas asociadas, y los elementos que contiene están definidos en base a variables y no a atributos:

```

BillQuantity = &quantity
&quantity = 0

```

¿Y qué pasa con el grupo Bills? Obsérvese que en este caso, es un grupo que sólo contiene otro grupo. El grupo contenido será repetitivo, por lo que Bills será una colección de Bill. Por este motivo, el subgrupo Bill podría omitirse (solamente dejar Bills) y que quede implícito. De este modo, las cláusulas del grupo que permiten definir order, filtros, se pueden asociar a este grupo.

### Sobre el lenguaje

- Los grupos pueden repetirse en el Source:

```
Clients
{
  Client
  {
    Name = 'Lou Reed'
    Country = 'United States'
    City = 'New York'
  }
  Client where CountryName = 'Mexico'
  {
    Name = CustomerName
    Country = CountryName
    City = CityName
  }
}
```

El resultado retornado será una colección de N+1 ítems: siendo N el número de clientes de México.

Si la condición se hubiese colocado en el grupo Clients, aplicaría a los dos subgrupos Client. Por eso es que se permite que las cláusulas operen a nivel de los grupos que se repiten (ítems), y no solo a nivel del grupo que es colección de ítems.

## Otros ejemplos del lenguaje

- Uso de parámetros y cláusulas de paginado

```
Customers                                     parm(&PageNumber, &PageSize)
{
  Customer [Count = &PageSize] [Skip = (&PageNumber - 1) * &PageSize]
  {
    Code = CustomerId
    Name = CustomerName
  }
}
```

- Uso de variables, invocación a otro Data Provider, cláusula Input

```
CustomersFromAnotherDataProvider
{
  &CustomersSDT = GetCustomers() // a DataProvider that Outputs Customers collection
  Customer Input &Customer in &CustomersSDT
  {
    Id   = &Customer.Code
    Name = &Customer.Name
  }
}
```

Con lo visto en este curso no agotamos el tema. Por ejemplo, las variables que pueden utilizarse pueden cargarse de otro Data Provider, pueden utilizarse cláusulas específicas, como la Input, etcétera.

Aquí encontrará toda la información sobre el lenguaje de los Data Providers:  
<http://wiki.genexus.com/commwiki/servlet/wiki?5309,Toc%3AData+Provider+language>

Y aquí la documentación completa de este objeto:  
<http://wiki.genexus.com/commwiki/servlet/wiki?5270,Category%3AData+Provider+object>,

# GeneXus™

**The power of doing.**

Videos

Documentation

Certifications

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

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

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