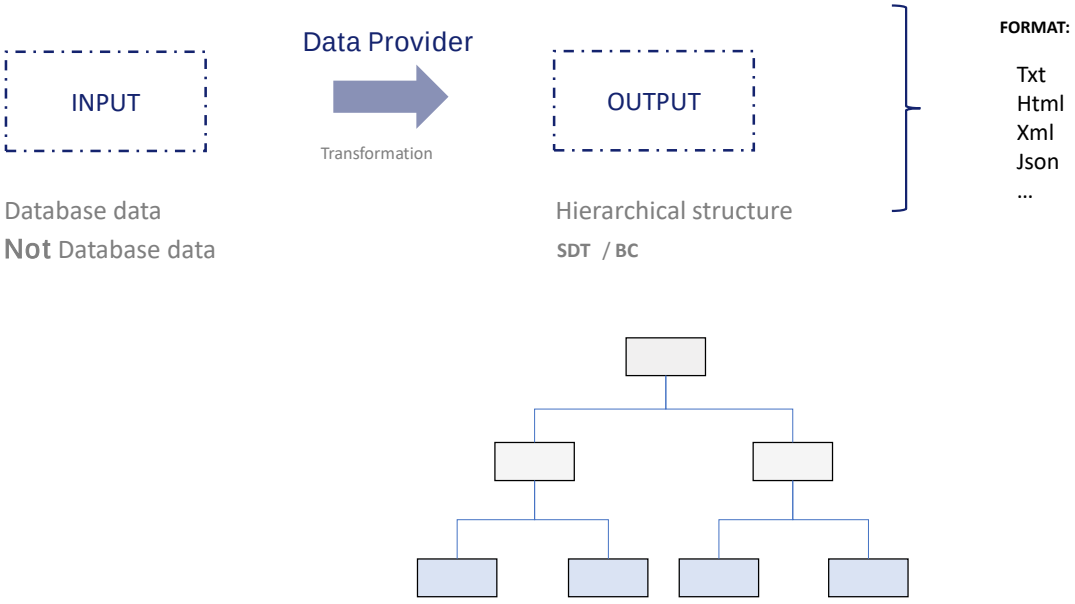
# Data Providers

Lenguaje y algunos ejemplos

GeneXus™

Data Provder






El propósito de los Data Providers es obtener información jerárquica para que quien la necesita pueda luego hacer algo con ella.

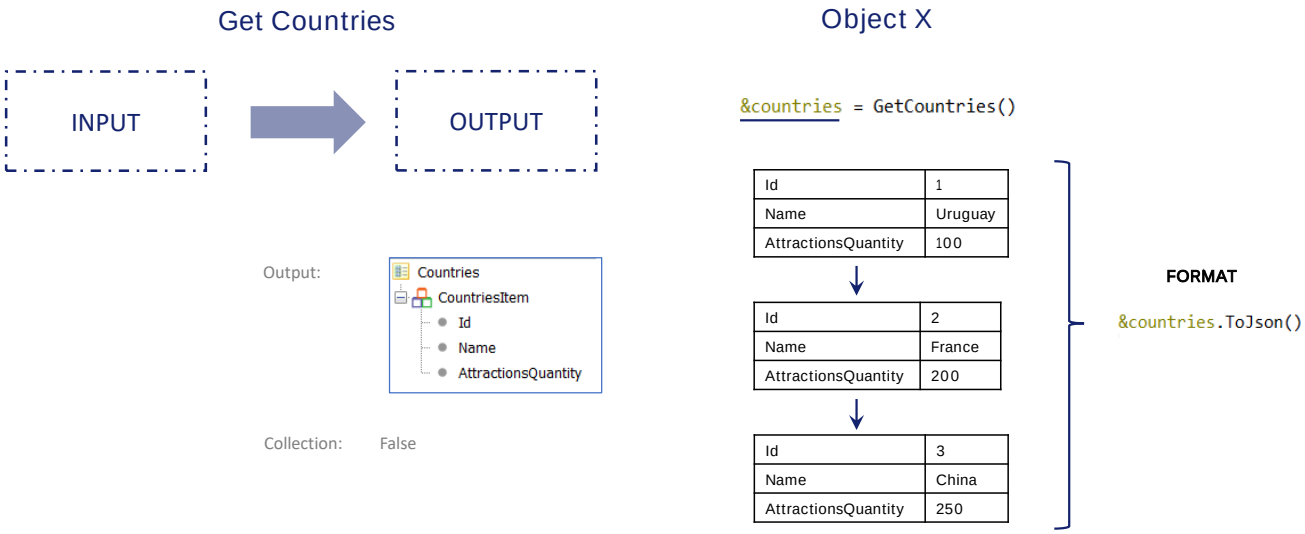
Recordemos que en los Data Providers el foco está ubicado en el lenguaje de salida: se indica en una estructura jerárquica cómo se compone ese output. Por eso se habla de un proceso de transformación de los datos de entrada en esa salida estructurada. Datos que pueden ser de la base de datos o no.

La manera de representar estructuras jerárquicas en GeneXus es con el objeto SDT, junto con la posibilidad de definir colecciones. Por supuesto, un Business Component puede pensarse estructuralmente como un SDT. Es por ello que en la propiedad Output del Data Provider podremos especificar tanto un SDT como un Business Component. Y además tenemos la propiedad Collection para poder indicar que la salida será una colección de ese tipo de datos indicado, o no, o será un único ítem.

De modo que un Data Provider siempre devolverá a quien lo llame una jerarquía, ya sea un SDT, una colección de SDTs, un Business Component, o una colección de Business Components.

Quien lo invoque, por tanto, es el responsable de hacer lo que necesite con esa información jerárquica. Por ejemplo, convertirla en otro formato de representación de información jerárquica, como XML o JSON que son formatos útiles para interactuar con terceros.

Data Provder



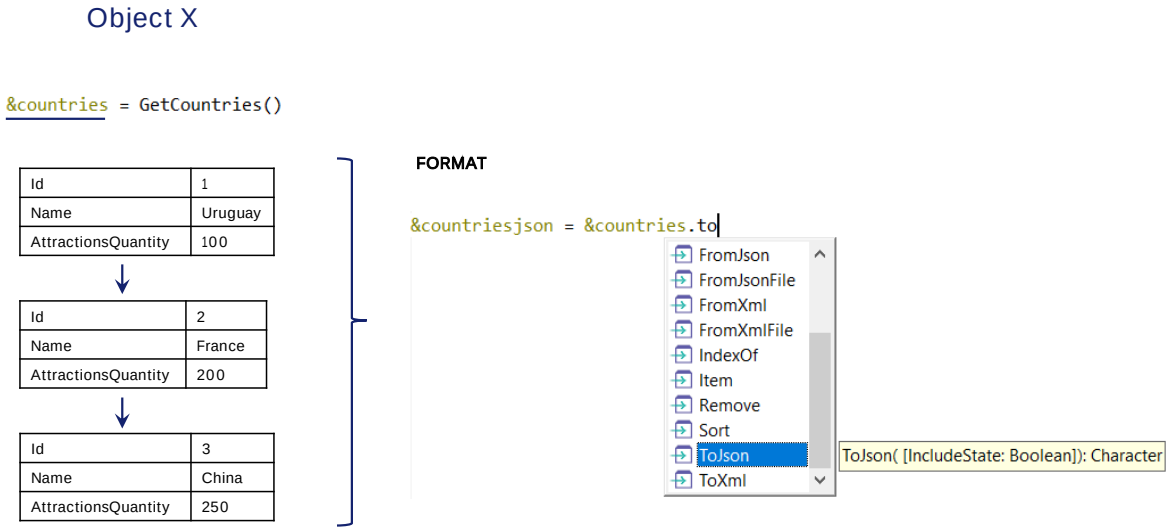

En este ejemplo tenemos un Data Provider de nombre GetCountries, que devolverá el tipo de datos al que hemos llamado Countries. Como se ve, será una colección de SDTs simples, cada uno de los cuales asumirá el nombre CountriesItem.

En las propiedades del Data Provider tendremos:

La propiedad Output, con el objeto SDT de nombre Countries.
Y la propiedad Collection en False, ya que no queremos una colección de Countries, si estuviera en True sería una colección de colecciones.

En este ejemplo, quien invoca al Data Provider está asignando su resultado a la variable countries del mismo tipo de datos que la salida. Este resultado será una colección específica en memoria, con valores específicos, los calculados dentro de ese Data Provider.

Luego, el programa que esté trabajando con esa variable puede hacer lo que desee con ella, por ejemplo, convertir su contenido a formato Json.

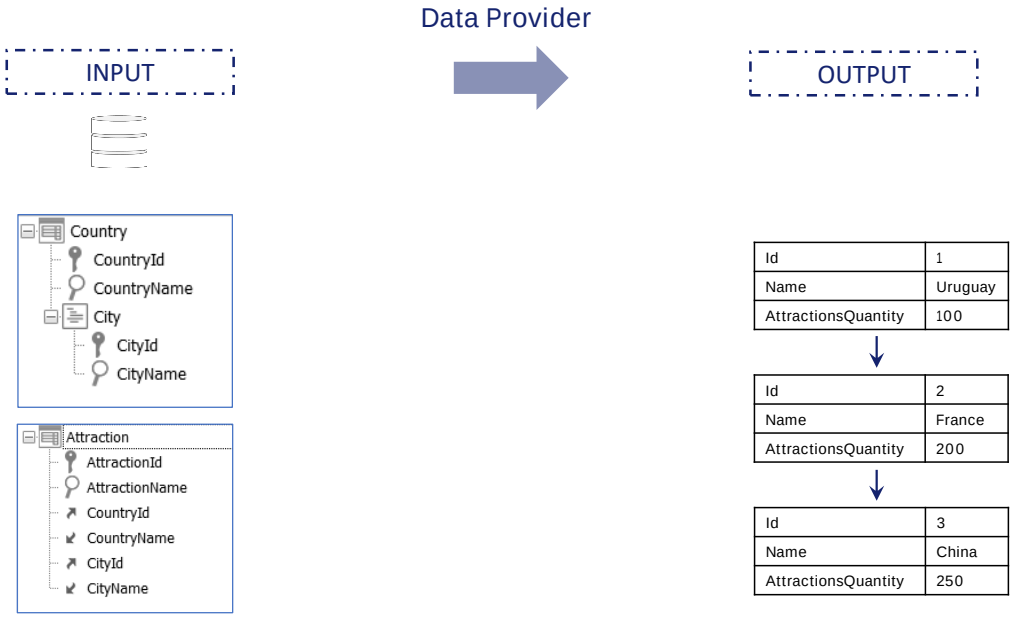



Aquí vemos cómo GeneXus ofrece diferentes métodos de conversión entre SDTs y algunos de esos otros formatos.

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.

Podemos tanto convertir de SDT a otro formato, como a la inversa: de ese otro formato a SDT.

Esto ya no tiene que ver con el Data Provider en si, sino con los tipos de datos estructurados.
El obtener la colección de países podría haberse logrado con un procedimiento en lugar de un Data Provider, y la parte de la conversión sería idéntica.

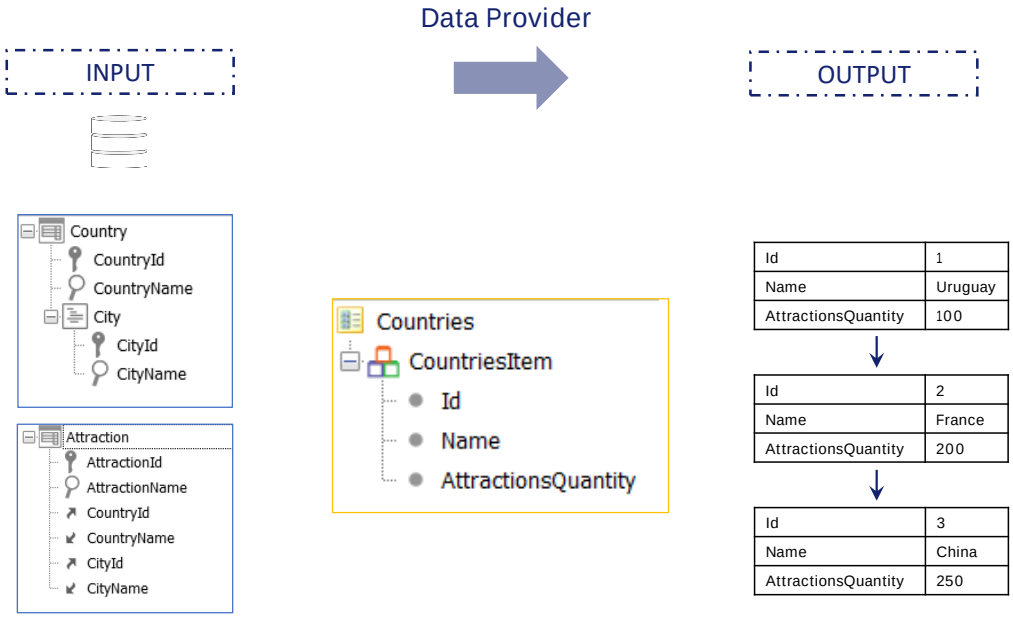



Veamos este ejemplo. Supongamos que, en el contexto de una aplicación para una agencia de viajes, necesitamos mostrar en pantalla un ranking de países, ordenados de mayor a menor por cantidad de atracciones turísticas para visitar que cada uno tiene.

En nuestra realidad tenemos las transacciones Country y Attraction con los siguientes atributos.
Una forma sencilla de conseguirlo es declarar un Data Provider que devuelva una colección de países donde para cada uno se agregue, además de su nombre e identificador, la cantidad de atracciones que posee. Y luego procesar esa colección en orden inverso por esa cantidad.

Como dijimos, el lenguaje del Data Provider coloca el foco en la salida, se calculan los elementos desde el punto de vista de la jerarquía que resultará.

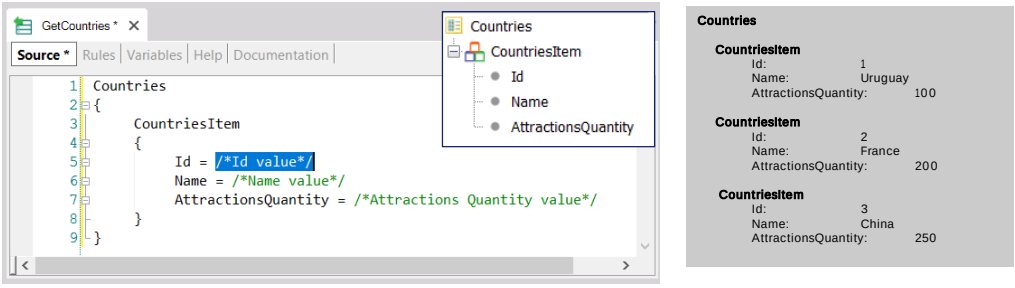
# Data Provder



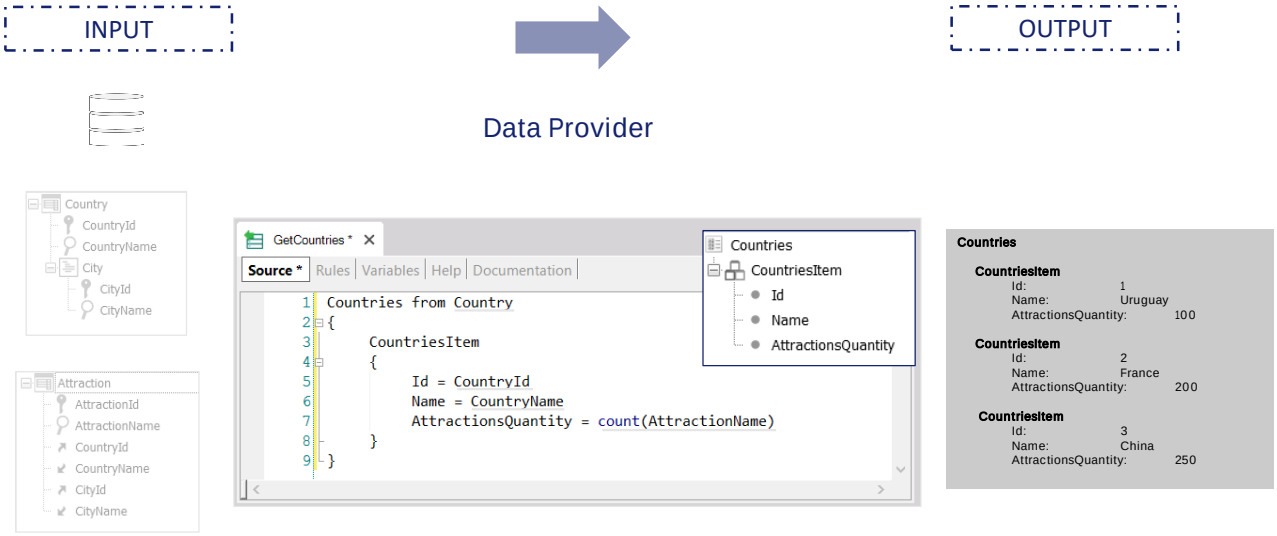

Para representar este ejemplo, creamos la siguiente estructura de datos que será la que posteriormente devuelva el Data Provider. Y luego debemos cargar este objeto SDT dentro del Source del Data Provider.

INPUT → OUTPUT

Data Provider




Al arrastrar dentro de él el SDT que será el output del Data Provider, ya nos presenta la estructura que tenemos que cargar. Vemos claramente cómo su lenguaje está orientado hacia la declaración de salida.




Aquí vemos lo que sería el Input de nuestro Data Provider, es decir, de donde se están tomando los datos. Tenemos una transacción base especificada, atributos y una fórmula inline. Claramente se está tomando información de la base de datos, para transformarla en la información jerárquica requerida.

INPUT

OUTPUT

Data Provider

GetCountries

Source | Rules | Variables | Help | Documentation

```
1  Countries
2  {
3      CountryiesItem
4      {
5          Id = 1
6          Name = "Uruguay"
7          AttractionQuantity = 100
8      }
9      CountryiesItem
10     {
11         Id = 2
12         Name = "France"
13         AttractionQuantity = 200
14     }
15     CountryiesItem
16     {
17         Id = 3
18         Name = "China"
19         AttractionQuantity = 250
20     }
21 }
```

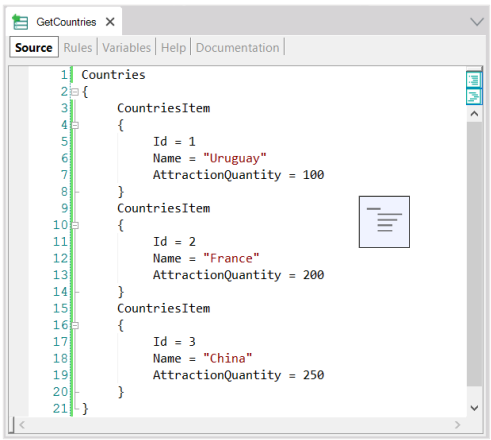
**Countries**

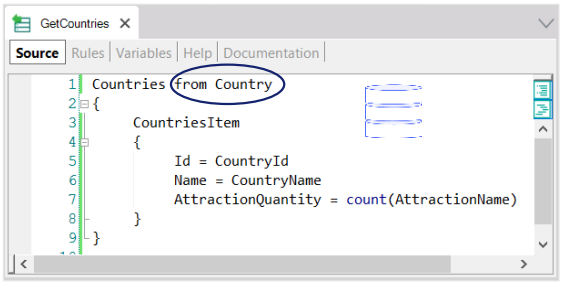
**CountriesItem**
Id: 1
Name: Uruguay
AttractionsQuantity: 100

**CountriesItem**
Id: 2
Name: France
AttractionsQuantity: 200

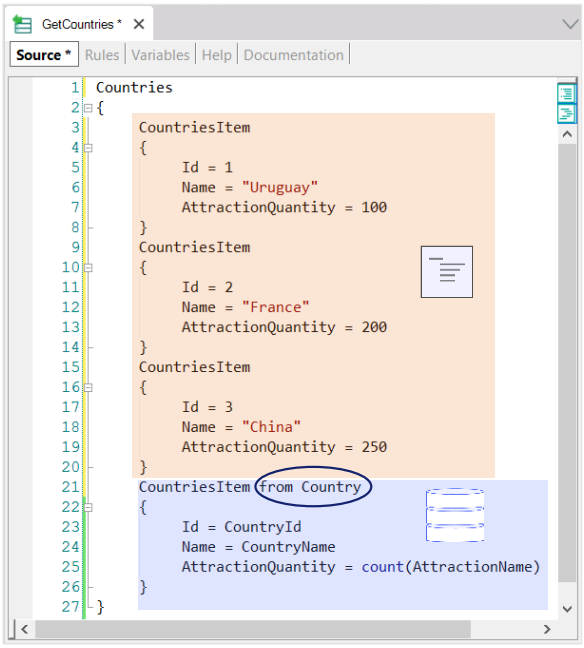
**CountriesItem**
Id: 3
Name: China
AttractionsQuantity: 250

El mismo resultado hubiéramos obtenido si cargábamos de manera estática los datos en el Source, es decir, si el input no se tomaba de la base de datos, sino que se codificaba manualmente. Como vemos, al no haber transacción base ni atributos, GenXus no traerá información de la base de datos.




INPUT MIXTO



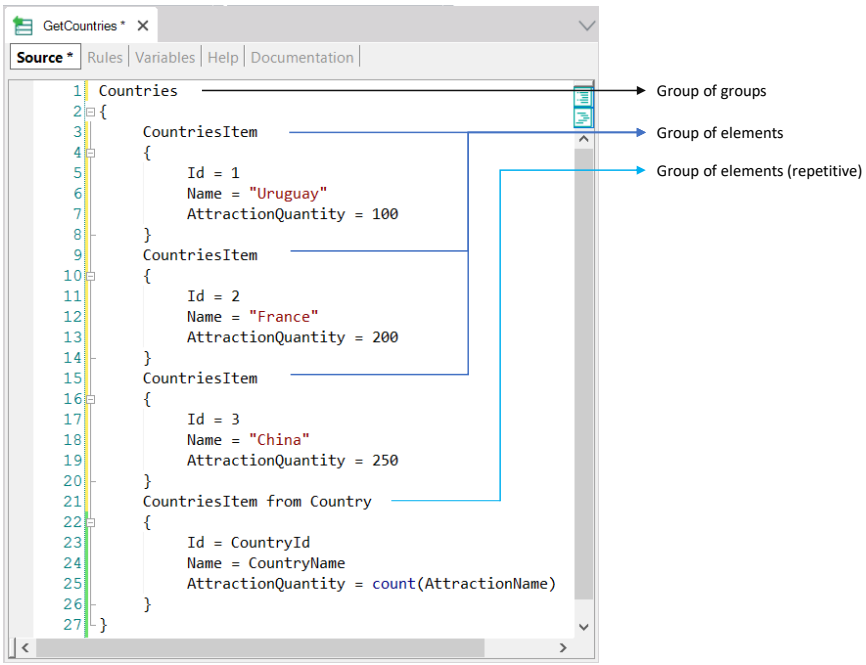

También podemos tener un input mixto: una parte codificada de manera estática y otra tomada de la base de datos.

En este ejemplo, vemos en el Source del Data Provider que presentará en la salida tres ítems de la colección cargados de manera estática, y luego N ítems más cargados de la base de datos a partir de los registros de la tabla Country.

Obsérvese que hemos tenido que mover la cláusula from para que aplique al subgrupo CountriesItem final y no a todos. Ya que como vimos, esta parte estática, codificada manualmente por nosotros, no toma registros de la base de datos.

Data Provider language

- Groups
- Elements
- Variables




```
Countries
{
    CountriesItem
    {
        Id = 1
        Name = "Uruguay"
        AttractionQuantity = 100
    }
    CountriesItem
    {
        Id = 2
        Name = "France"
        AttractionQuantity = 200
    }
    CountriesItem
    {
        Id = 3
        Name = "China"
        AttractionQuantity = 250
    }
    CountriesItem from Country
    {
        Id = CountryId
        Name = CountryName
        AttractionQuantity = count(AttractionName)
    }
}
```

Veremos ahora los componentes principales del lenguaje del Source de un Data Provider.

Tendremos grupos, elementos y también podemos utilizar variables.

Los elementos son los análogos de los miembros de un SDT. Si pensamos a la jerarquía como un árbol, los grupos son las ramas y los elementos las hojas. Es decir, los grupos son elementos compuestos; pueden componerse de otros grupos y/o de elementos.

Los grupos podrán ser estáticos o cargarse de modo dinámico, a éstos les llamamos grupos repetitivos. En nuestro ejemplo, los primeros tres grupos son estáticos, se cargan con datos fijos, mientras que el último es un grupo que tendrá tabla base asociada, y producirá, por tanto, N ítems en la salida, uno por cada registro de la tabla base considerado.

Un grupo con tabla base será equivalente a un comando for each.

## Data Provider language

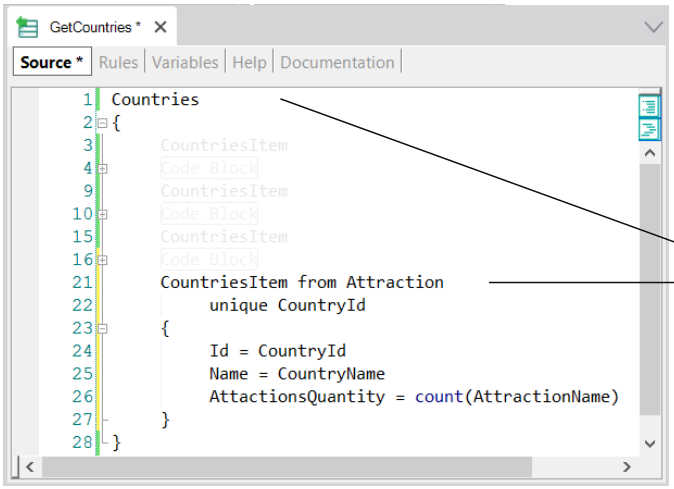

**from** *BaseTransaction*
[**skip** *expr1*] [**count** *expr2*]
[{[**order**] *order_attributesi* [**when** *condi*]}... | [**order none**] [**when** *condx*]]
[**using** *DataSelectorName*([[*parm1* [,*parm2* [, ...] ])]
**unique** $att_1$, $att_2$...,$att_n$
[{**where** {*conditioni***when** *condi*} |
{*attribute* **IN** *DataSelectorName*([[*parm1* [,*parm2* [, ...] ]} }...]

Los grupos permiten especificar transacción base, aunque, a diferencia del for each, aquí se especifica antecediendo el nombre de la transacción base o nivel con la palabra "from".

Observemos que en este ejemplo lo que hemos hecho es, en lugar de recorrer la tabla base COUNTRY, recorrer ATTRACTION, utilizando la cláusula unique para que si hay muchas atracciones de un país se tome en cuenta solo una, y para esa se cuenten todas las demás atracciones que tengan el mismo país. De este modo en la salida solamente se listarán los países con atracciones.

Vale exactamente lo mismo que todo lo visto para el comando for each.

# Data Provider language

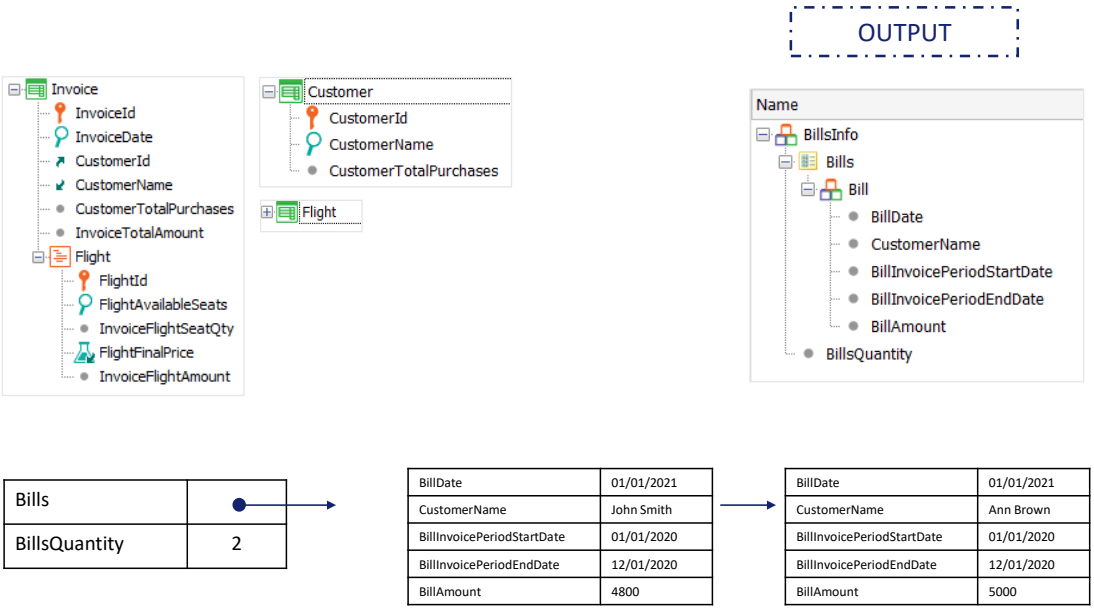



**from** *BaseTransaction*
[**skip** *expr1*] [**count** *expr2*]
[{[**order**] *order_attributesi* [**when** *condi*]}... | [**order none**] [**when** *condx*]]
[**using** *DataSelectorName*([[*parm1* [,*parm2* [, ...] ])]
**unique** $att_1$, $att_2$...,$att_n$
[{**where** {*conditioni***when** *condi*} |
{*attribute* **IN** *DataSelectorName*([[*parm1* [,*parm2* [, ...] ]} }...]

Si los grupos estáticos no se requirieran, entonces hubiera sido lo mismo especificar las cláusulas a nivel del subrupo CountriesItem que del grupo padre, Countries, colección de CountriesItem.

En este caso, entonces, declarar las cláusulas a nivel del grupo padre es equivalente a hacerlo a nivel del grupo hijo.

Data Provider language




| Bills | |
|---|---|
| BillsQuantity | 2 |

| BillDate | 01/01/2021 |
|---|---|
| CustomerName | John Smith |
| BillInvoicePeriodStartDate | 01/01/2020 |
| BillInvoicePeriodEndDate | 12/01/2020 |
| BillAmount | 4800 |

| BillDate | 01/01/2021 |
|---|---|
| CustomerName | Ann Brown |
| BillInvoicePeriodStartDate | 01/01/2020 |
| BillInvoicePeriodEndDate | 12/01/2020 |
| BillAmount | 5000 |

Veamos ahora este otro ejemplo.

Tenemos un Data Provider que devolverá como estructura un SDT, el cual tiene una colección de Bills y un elemento Quantity.

En nuestra aplicación tenemos a la transacción Invoice, que consta de dos niveles y con los siguientes atributos. La transacción Flight independiente, y la transacción Customer.

Lo que queremos es que a partir de las facturas que se le hayan generado a cada cliente entre un par de fechas dadas, se genere un recibo de pago, por el total de todas esas facturas. Puede que entre ese par de fechas de facturación no todos los clientes tengan recibos a ser generados, ya que no tienen por qué habérseles efectuado facturas en ese rango de fechas. En la estructura a ser devuelta se necesita saber cuántos recibos se obtienen del cálculo, y por ello tenemos el miembro BillsQuantity.

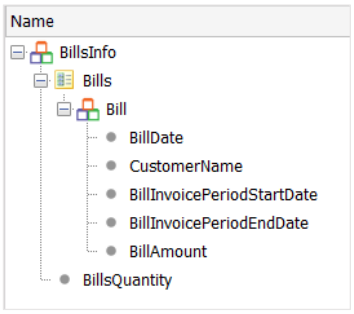
Si en el rango de fechas 01/01/2020 al 12/01/2020 solamente existen facturas para los clientes John Smith y Ann Brown, la salida será como se representa en la imagen: una variable SDT con dos miembros: uno de tipo colección y el otro de tipo Numeric. La colección tendrá dos ítems, como se muestra.

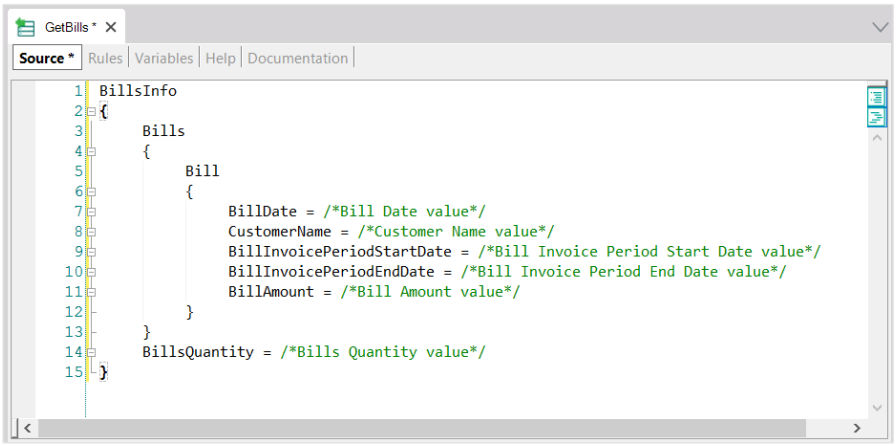
Dicho de otro modo, el DataProvider devolverá una estructura con dos elementos: la colección de recibos, por un lado, y la cantidad de

elementos de esa colección por otro.

# Data Provider language

**Output**: BillsInfo
**Collection**: False



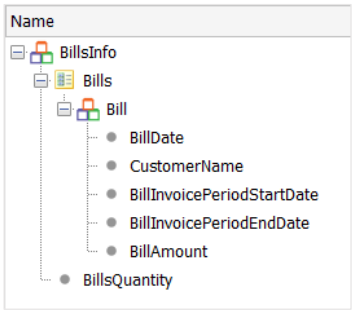



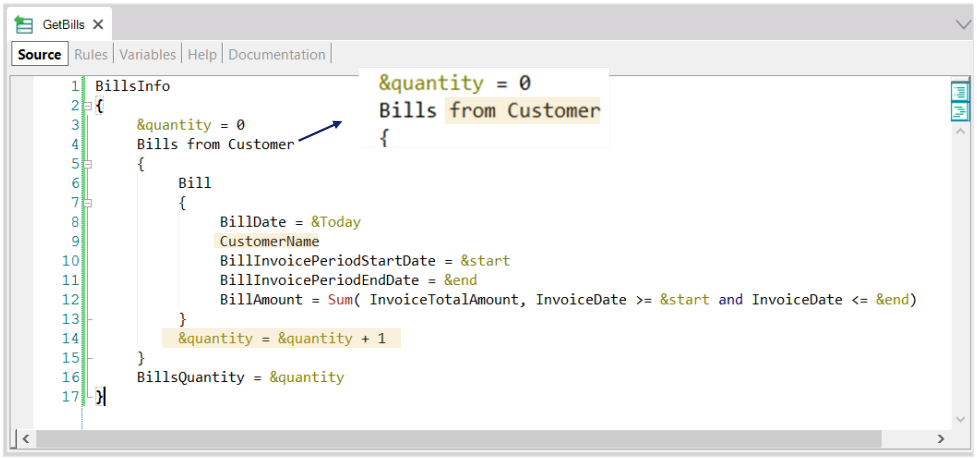

Si arrastramos al Source del Data Provider el SDT, vemos cómo es inicializado, donde a la propiedad Collection se la dejará con su valor default "False". Esto es lo que queremos, porque no vamos a devolver una colección de BillsInfo, sino un solo elemento de ese tipo, que en particular contendrá, entre otras cosas, una colección de Bills.

# Data Provider language

**Output**: BillsInfo
**Collection**: False

**parm**(in: &start, in: &end);




Programemos la regla parm para recibir por parámetro el rango de fechas de facturación.

Y luego al grupo Bills, que representa la colección, le especificamos transacción base.

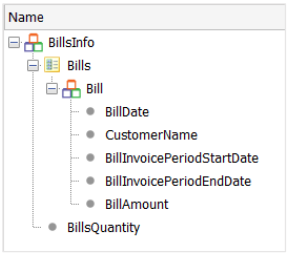
¿Por qué colocamos CustomerName sin asignarle valor? Porque se llama igual que el atributo CustomerName y estamos navegando la tabla Customer. Por lo que podemos utilizar esta notación abreviada. Es equivalente a haber escrito: CustomerName igual a CustomerName, donde el de la izquierda es el miembro del SDT y el de la derecha es el atributo de la tabla Customer.

Observemos que el uso de las variables es igual al que hacemos en un for each.

Tenemos un problema: en este caso vamos a devolver un ítem Bill en la salida incluso para clientes que no tengan facturas en el rango recibido por parámetro. ¿Cómo haríamos para que esto no sucediera?

Data Provider language

**Output**: BillsInfo
**Collection**: False




**parm**(in: &start, in: &end);

GetBills

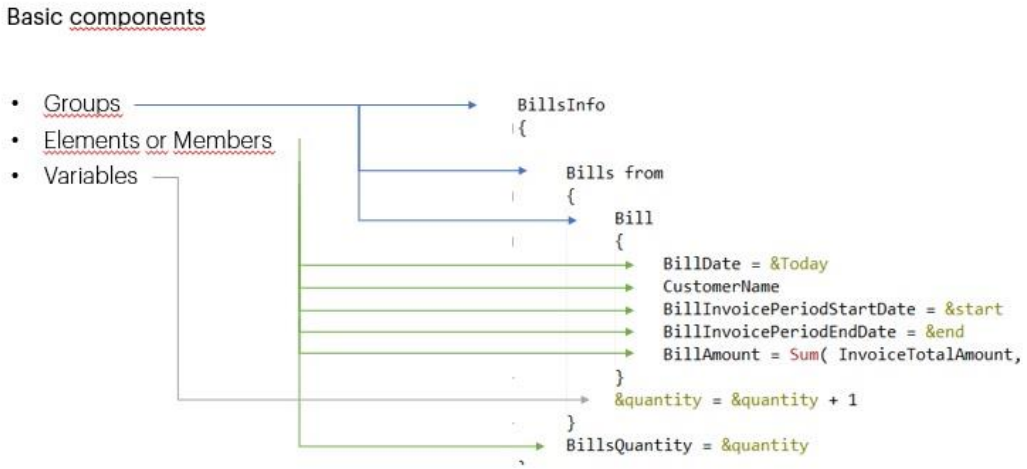
Source | Rules | Variables | Help | Documentation

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

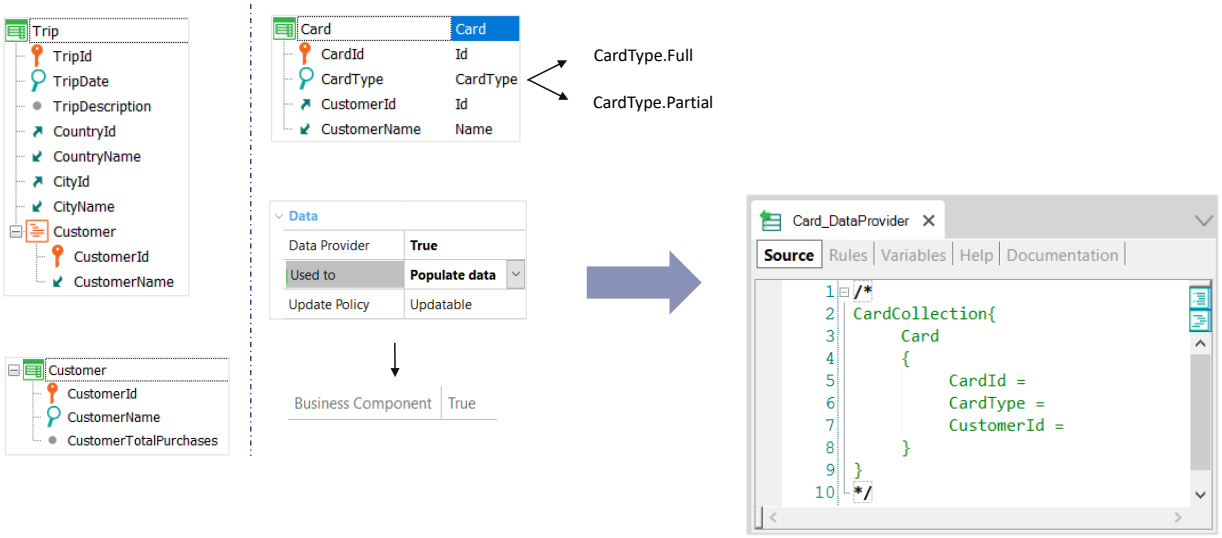
Una forma es cambiando la transacción base por Invoice y quedándonos con los registros de Invoice sin repetir el CustomerId y con fechas en el rango deseado.

Luego en el Sum interno contaremos los totales de todas las facturas del cliente que estén en ese mismo rango de fechas.Como venimos diciendo, es idéntica a la lógica del for each.

Data Provider language




Aquí vemos una vez más los componentes básicos del lenguaje de un Data Provider.




Ahora veamos este otro ejemplo. Queremos poblar con datos la tabla asociada a una nueva transacción de nombre Card, utilizando su Data Provider asociado y a partir de datos de otras tablas de la base de datos.

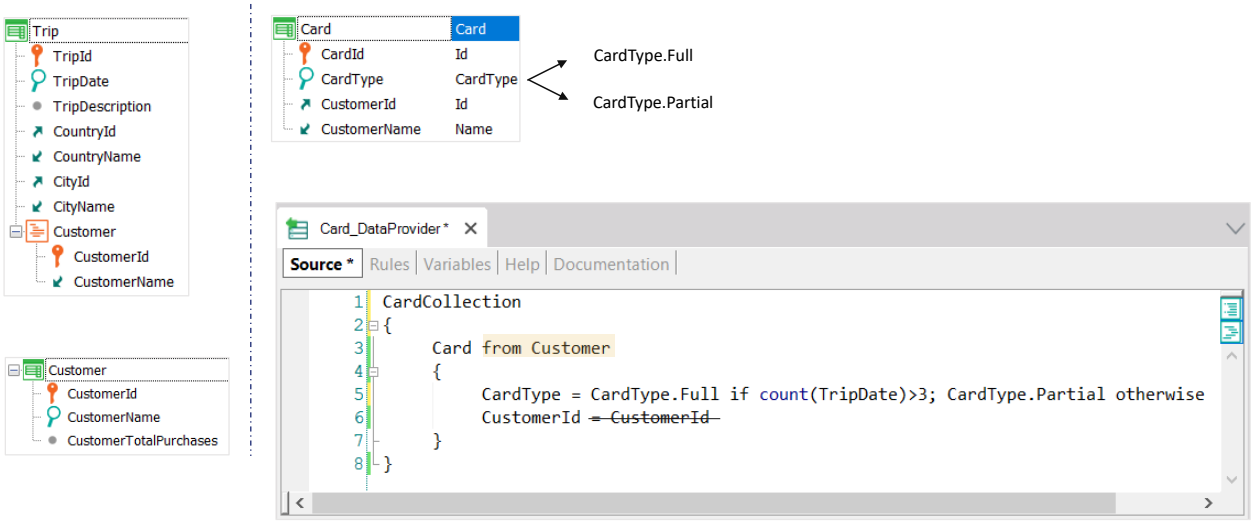
Tenemos la transacción Customer para registrar los clientes de la agencia de viajes y Trip para registrar cada viaje o excursión ofrecido por la agencia a una ciudad determinada. Tenemos un subnivel con los clientes registrados para el viaje.

Supongamos que la agencia de viajes decide que todos los clientes que han contratado más de 3 viajes 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".

Cada tarjeta tiene un identificador que se autonumera, un cliente y un tipo de tarjeta, para el que hemos definido un dominio enumerado que admite solamente los valores "Full" o "Partial".

Ahora bien, queremos que la tabla asociada a la transacción Card se inicialice con la información correcta, por lo que prendemos la propiedad Data Provider y dejamos el valor de Populate data para "Used to", de modo que creará automáticamente el DataProvider que vemos, y prenderá la propiedad Business Component, para insertar las tarjetas que sean devueltas por ese Data Provider al ejecutarse automáticamente en la primera ejecución.

¿Cómo declaramos el Source del Data Provider?




Crearemos un Business Component Card en la colección por cada cliente. Por eso al grupo Card le especificamos transacción base Customer. Sabemos, por tanto, que es un grupo con tabla base.

Quitamos el elemento CardId del grupo Card dado que el dominio Id del atributo CardId de la transacción es autonumerado.

Luego, observemos cómo cargamos el valor del elemento CardType utilizando una fórmula inline condicional. Asumirá el valor del enumerado CardType.Full siempre y cuando el resultado de ejecutar la fórmula inline count(TripDate) sea mayor que 3; de lo contrario se le asignará el valor CardType.Partial.

Esa fórmula irá a contar los registros de la tabla Trip, filtrando por CustomerId.

Y luego al elemento CustomerId, que corresponderá al Business Component, se le asigna el valor del atributo CustomerId de la tabla base del grupo Customer. Podemos utilizar la notación abreviada y quitar la asignación.
Aquí, por tanto, vemos un ejemplo donde el Data Provider devuelve una colección de Business components cuyos datos se obtienen de otra tabla.

Es idéntico, una vez más, al caso de un For each.

GeneXus™

training.genexus.com
wiki.genexus.com