

DATA PROVIDERS

Contributions on their language, and conclusions

GeneXus™ 16

Here we will be introducing new knowledge about the use of Data Providers.

Characteristics of a Data Provider

Output:

- simple SDT
- collection SDT
- simple BC
- collection BC

Parameters:

- Variable
- Attribute

Origin of data:

- fixed data
- data from the DB:
 - To load an SDT: of one or several tables
 - To load a BC:
 - taken from the same table
 - from another table

We should recall that the purpose of a Data Provider is to return a data structure loaded in memory (which may be collection or not). To achieve this, it provides us with a declarative language focused on the output structure, so that we basically will need to indicate how to obtain each of those information elements.

To load, we may use a simple or collection SDT, or a Business Component structure either simple or collection.

Like any other object, a Data Provider may receive parameters (both variables and attributes). However, as opposed to all other objects, and precisely because of the objective to allow developers to focus on the output, it is not declared in the parm rule but rather explicitly as Output property of the object.

We have seen that the data used to load the structures may be fixed data, or variables, taken from one or from several tables of the database. Specifically, when we are loading a structure of the business component type, the data may be from the table associated with the transaction on which the BC was defined, or from another table in the database.

Initialize a table with fixed data

- Creating records in CATEGORY table

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

This group is added for clarity's sake. It is not necessary if we use the property `Collection=True`

The data provider **does not have base table** because the data is fixed

These are not attribute names but names of elements in the business component based on the Category transaction

Here we are initializing a business components structure of Category, for which we are using a data provider.

Note that we repeat the groups, one for each category to be created. We could leave out the definition of the CategoryCollection group because our purpose is to return a collection of Category elements and we have already set up the Collection property of the data provider with True value.

Another thing we should note is that since the data provider will not have to go over any table in order to obtain the data because we are providing the data in the form of fixed values, then this data provider will not have base table and we will not have to use a "from" clause.

And last, it is also important to note that the CategoryName elements on the left of the assignments are not the attributes of the Category transaction, but rather the elements of the business component based on the Category transaction, which we are loading through the data provider.

Data from the database

Loading an SDT of one or several tables: Ranking of countries

The screenshot displays two windows in the GeneXus IDE. The top window, titled 'SDTCountries', shows the 'Structure' view of an SDT. It contains a table with columns 'Name', 'Type', and 'Is Collection'. The 'SDTCountries' SDT is marked as a collection. It contains an 'SDTCountriesItem' which has three properties: 'Id' (Type: Id), 'Name' (Type: Name), and 'CountryAttractionsQuantity' (Type: Numeric(4,0)).

The bottom window, titled 'DPRankingCountriesWithAttractionsQty', shows the 'Source' view of a data provider. The code is as follows:

```

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

```

Two callout boxes provide additional context:

- A callout box points to the 'Country' table in the source code, stating: "The base table of the data provider is: COUNTRY".
- A callout box points to the 'count(AttractionName)' formula, stating: "The table navigated by the formula is: ATTRACTION".

In this example we see how we can load an SDT with data from several tables. Our aim is to build a ranking of countries by the number of attractions in each country.

To do that we define a collection SDT in order to store the identifier, name and number of attractions in each country. To load this SDT we will use a data provider.

In order to obtain the data on the countries, the data provider goes over the COUNTRY table and, for each country, the count formula navigates through the ATTRACTION table to count attractions in that country.

Once the collection is obtained, we may order it in descending order by the number of attractions.

Data from the database

- Load of a business component with a single table

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

The base table of the data provider is: COUNTRY

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

Since the names of the elements in the business components are the same as the names of the attributes, we can use abbreviated notation

In this example we are loading the data of countries in memory.

Data from the database

- Load of a business component with data from a different table

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

The base table of the data provider is: CUSTOMER

The elements correspond to a business component of the SERVICECARD transaction

In this example, a special card (total or partial type) is to be granted to customers who have bought more than 3 trips. To do it we will go over the customers in order to count the number of attractions and assign the corresponding card.

We do this by means of a data provider with which we load a business component structure of the SERVICECARD transaction, to then go over that collection and save that information in the database.

We will now go into the details of this example.

Example

Name	Type
Customer	Customer
CustomerId	Numeric(4,0)
CustomerName	Character(20)
CustomerLastName	Character(20)
CustomerAddress	Address, GeneXus
CustomerPhone	Phone, GeneXus
CustomerEMail	Email, GeneXus
CustomerAddedDate	Date
CustomerMiles	Numeric(4,0)
CustomerFreeTrips	Numeric(4,0)
Trip	Trip
TripId	Id
TripDate	Date
TripDescription	Description
CountryId	Id
CityId	Id
CityName	Name
TripIsFree	Numeric(4,0)
CustomerTripMiles	Numeric(4,0)

Business Component = True

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

Domain: Type

Enum Valuis: { Full
Partial

Requirement: Massive creation of cards (ServiceCard), only for customers who don't have any:

- "Full" → for a customer with more than 3 trips bought
- "Partial" → for other cases

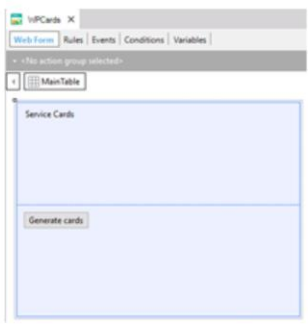
Let's now suppose that the travel agency decides that all customers who have bought more than 3 trips will be receiving a special card of the "Full Services" type which will enable them to enjoy all services for free. For cases where the customer has bought less than 3 trips, a card of the "Partial Services" type will be issued.

The transactions we have are: the "Customer" transaction, and the "ServiceCard" transaction that defines the cards.

Each card has an autonumbered identifier, a customer, and we have defined the ServiceCardType attribute based on the enumerated Type domain (which only allows the "Full" or "Partial" values).

Solution...

1



2 We propose a **Data Provider** to load and return the collection of cards to be generated:

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

We drag the transaction to the source of the data provider

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

They are not attributes but elements in the structure to be loaded in memory.

We have defined the WPCards web panel that just offers a button to trigger the automatic process for generating and viewing new cards.

What should happen when the button is pressed?

A card of the corresponding type is to be created for every customer who has no card issued, always bearing in mind the number of trips bought at the agency.

We will propose a solution where we will **use a Data Provider** to load and return the collection of cards to be generated. **And afterwards we will go over the collection to record the cards in the database.**

How do we do this?

First we configure the SERVICECARD transaction as Business Component, in order to record the cards, for which purpose we apply the Business Component concept.

Then we create a Data Provider object called DPCards and we drag the SERVICECARD transaction to the source of the data provider.

As we saw in previous videos, this will define a structure in memory with items whose name and type are the same as those of the attributes of the transaction defined as Business Components.

Then we must go over the CUSTOMER table, then filter customers for whom cards have not been issued yet, **and add for them** one card in the collection.

The table that will navigate the Data Provider is determined with the transaction base defined in the From clause, which in this case is CUSTOMER.

Note that the table that we will be going over to obtain the data is not the same table associated with the transaction on which we defined the Business Components, that is, SERVICECARD.

Solution...

3

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

→ ServiceCardId based on autonumbered domain

... the structure of the data provider will be as follows...

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

We can also use a procedure that returns a value

base table of the DP:
CUSTOMER

It is clear that we will be assigning the CustomerId attribute to the CustomerId item.

We don't have to assign a specific value to the ServiceCardId item because, as we should recall, the structure to be loaded was dragged from a transaction declared as Business Component, and therefore this item is based on the ServiceCardId attribute that belongs to the Id domain, and its Autonumber property has been set as Yes.

To ServiceCardType we can assign the value returned by a conditional formula. This means that the formula will return the "Full" or "Partial" type according to the number of trips that the customer has bought. We could have also used a procedure to calculate and return that value.

In order to define the base table that will navigate the data provider, GeneXus goes to the transaction base we added with the From clause, so the table that it will go over is the one associated with that transaction, which in this case is the CUSTOMER table.

GeneXus will also verify that the attributes we add to the right of the assignment signs are part of the extended table of CUSTOMER, otherwise, an error will occur, and we will see it reported on the data provider's navigation listing.

Note that the attributes inside the formulas are not considered for this verification, because they are only used to define the table to be navigated by the formula.

It is not necessary to assign a value to the element corresponding to the autonumbered ServiceCardId attribute because it takes it automatically due to the autonumbering of the attribute. Likewise, we can leave unassigned any element in the business component to which we do not wish to load a value, so when the record is created in the table, the value of the attribute will be empty.

In this case, restrictions apply to the foreign keys, to which we must assign a value, unless we have defined the attribute as nullable.

Solution...

... but we only want to go over the customers who still don't have cards...

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

... let's simplify...

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

... let's analyze the output...

Output	
Output	ServiceCard
Collection	True
Collection Name	Cards

← It was configured automatically when the trn was dragged

Remember that we do not want to navigate all customers and load a card for each. We want to navigate only those customers who still do not have cards.

Data providers enable us to include, in their syntax, all the clauses permitted in the For each, so we add the Where clause shown on the slide.

Since the only attribute referenced in the Where is inside a formula, as said, it will not be included in the verification of whether it belongs to the extended table of Customer or not.

Note that, instead of CustomerId = Customer Id, we simply used CustomerId. Remember that the CustomerId to the left of the assignment is the element of the structure that will be loaded in memory, and the one on the right is the attribute that will define its value. Since they have the same name, we may use the **abbreviated notation** and write CustomerId only.

Let's now analyze the output, in other words, what the data provider returns. Since the SERVICECARD transaction was dragged over the source, then the Output property has been automatically associated with the ServiceCard business component associated with the transaction.

And what about the Collection property? The structure we are loading does not represent a collection. It only represents an instance in memory, with the structure of the SERVICECARD transaction. However, we must obtain a **collection of generated cards**, so we set up the Collection property with True value... and we indicate a name for the collection that this data provider will return loaded.

Solution...

Invoking the Data Provider...

Event Enter

&ServiceCardCollection = DPCards()

EndEvent

The screenshot shows the GeneXus IDE interface. The top part displays a web panel titled 'Service Cards' with a 'Generate cards' button. Below the button is a table with the following columns and data:

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

The bottom part of the screenshot shows the 'Variables' tab in the IDE. It displays a list of variables:

Name	Type	Is Collec...	Description
Variables			
Standard Variables			
ServiceCard	ServiceCard	<input checked="" type="checkbox"/>	Service Card

A red arrow points from the 'Generate cards' button in the web panel to the code snippet '&ServiceCardCollection = DPCards()' in the 'Event Enter' section.

Let's now go back to the web panel to invoke the data provider.

In the Enter cards event associated with the button, we assign what the data provider returns to a variable (&ServiceCardCollection) defined as a collection of cards (ServiceCard).

In the form of the web panel, we insert the &ServiceCardCollection variable. Because it is a collection, GeneXus will automatically understand that it must show the content on a grid.

Solution...

Saving the cards...

The screenshot shows the GeneXus IDE interface. At the top, there are tabs for 'Web Form *', 'Rules', 'Events', 'Conditions', and 'Variables *'. Below the tabs, there is a section for 'Service Cards' with a 'Generate cards' button. A red arrow points from this button to a code block on the right. Below the code block, there is a 'Variables' table with columns for Name, Type, Is Collec..., and Description.

```

Event Enter
  &ServiceCardCollection = DPCards()

  For &oneCard in &ServiceCardCollection
    &oneCard.Save()
  EndFor

  Commit

EndEvent

```

Name	Type	Is Collec...	Description
& Variables			
& Standard Variables			
ServiceCard	ServiceCard	<input checked="" type="checkbox"/>	Service Card
OneCard	ServiceCard	<input type="checkbox"/>	One Card

Now, is this enough for the cards returned by the data provider to be actually recorded on the SERVICECARD table associated with the SERVICECARD transaction?

No, it is not enough. For the time being, the cards are loaded in memory and we have shown the contents of the collection.

When we studied the use of business components, we saw that, in order to save we must use the Save method and then execute Commit. So, we still have to go over the collection returned by the data provider and then save each element in the collection as a record in the physical table. And following the saving of all cards, we declare the Commit command.

In order to run through the collection of cards returned by the data provider, we use the **For element in collection** command. This &oneCard variable must be defined as the ServiceCard business component type, and it represents each element from the collection that is iterated.

Better Solution...

Saving the cards...

The screenshot shows the GeneXus IDE interface. At the top, there are tabs for 'Web Form *', 'Rules', 'Events', 'Conditions', and 'Variables'. Below the tabs, there is a 'MainTable' section with a 'Service Cards' table. A 'Generate cards' button is located below the table. A red arrow points from the 'Generate cards' button to a red-bordered box containing the following code:

```

Event Enter
  &ServiceCardCollection = DPCards()
  if &ServiceCards.Insert()
    Commit
  endif
EndEvent

```

Below the 'Service Cards' table, there is a table with columns for 'Service Card Id', 'Service Card Type', 'Customer Id', 'Customer Name', and 'Customer Last Name'. The values in the table are: `&ServiceCard.item().ServiceCardId`, `&ServiceCard.item().ServiceCardType`, `&ServiceCard.item().CustomerId`, `&ServiceCard.item().CustomerName`, and `&ServiceCard.item().CustomerLastName`.

At the bottom, there is a 'Variables' table with columns for 'Name', 'Type', 'Is Collec...', and 'Description'. The table contains the following data:

Name	Type	Is Collec...	Description
& Variables			
Standard Variables			
ServiceCard	ServiceCard	<input checked="" type="checkbox"/>	Service Card
OneCard	ServiceCard	<input type="checkbox"/>	One Card

However, remember that the Insert method of a Business Component collection variable can automatically do what we did manually before.

And not only that, it also returns True if all the insertions in the collection were successful, and False if there were any issues. This way we can Commit if everything was successful. Otherwise, we would have to read the error messages and take the actions we consider suitable.

Solution...

At runtime...

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) Upon pressing the button, **the result is:** the cards generated are shown.

2) Upon pressing the button again, **the result is:** no cards are generated.

Application Name

Recents Customer - WPCards

Service Cards

Generate cards

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

Now the development of what we were asked for is finally complete. We execute the web panel and press the button. On the grid we can see the list of cards generated.

We could wonder what would happen if we press the "Generate Cards" button once again...? Will the cards be created again for the same customers?

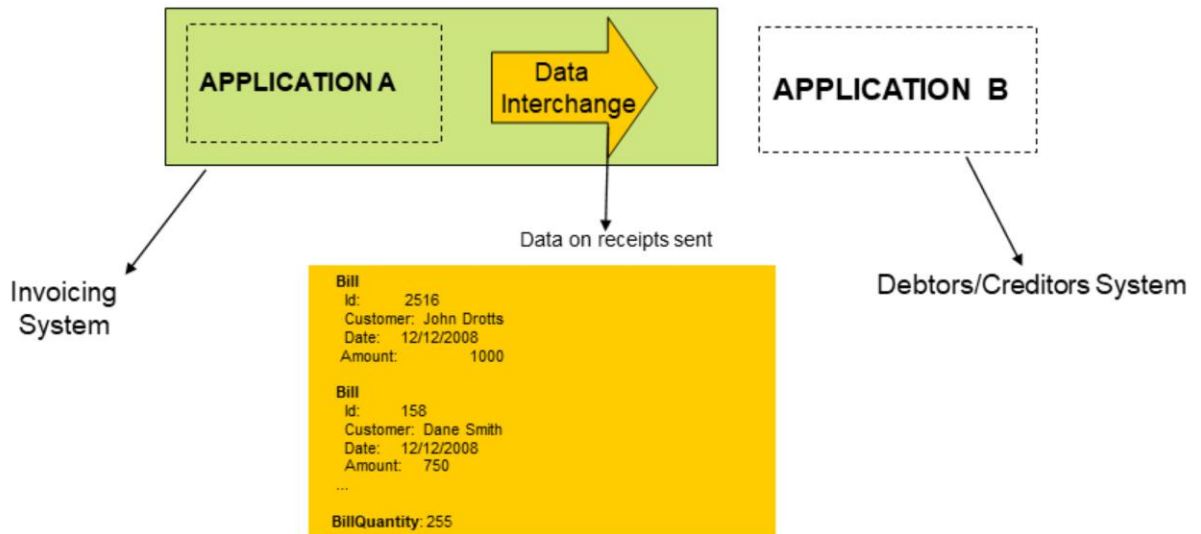
They will not, because, in the data provider, we filtered that we only wanted to navigate customers with no cards.

We should mention that there are other solutions to solve the same requirement in GeneXus. **With this implementation we have used the concept of Business Component to update the database and we have combined its use with the previous loading of a collection structure in memory, with the data to be recorded.**

The use of a Data Provider for this purpose is quite simple and saves us from having to write explicit code.

About the language

- Scenario: exchange of hierarchical information between modules of an application.

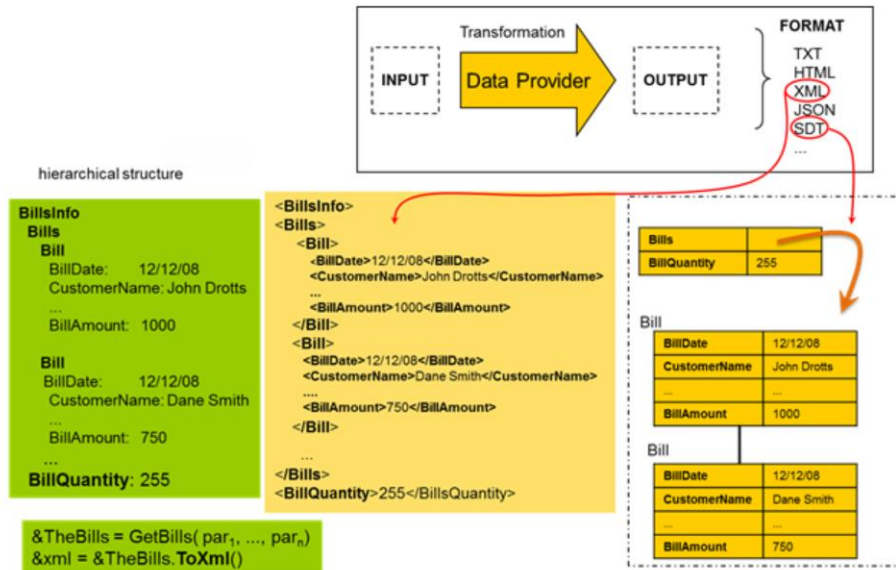


Suppose that the travel agency's system is divided into modules in order to manage the invoicing system and the debtors/creditors system. We want to send a listing, from the invoicing system to the debtors/creditors system, with the receipts corresponding to a given period of invoices (that is, for a given period, we must summarize, for each customer, the total amount invoiced and then generate a receipt).

It is hierarchical information (we will be sending receipt data specific of each receipt document).

The most usual format for exchanging hierarchical information used to be Xml and Json, but there are more.

About the language



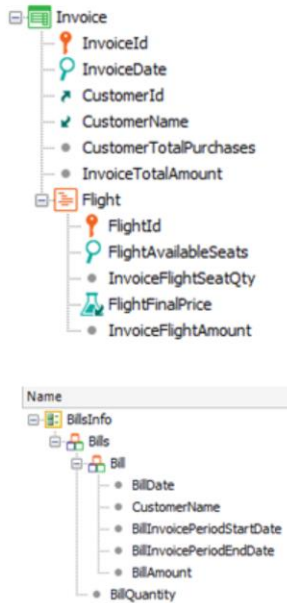
We should recall that, with a Data Provider, the focus is on the output language: the composition of the Output is indicated in a hierarchical structure.

Then, for each element in the hierarchical structure, we will have to indicate -in the Source of the data provider- how it is calculated.

We may represent the same structured information using the various existing formats.

That is the idea behind the data provider. If a new format for representing structured information comes up in the future, then the Data Provider will remain unchanged... GeneXus will implement the transformation method for that format, and we will do is use it.

About the language



```

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

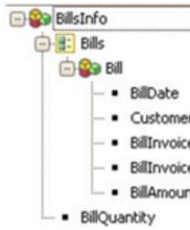
&TheBills = GetBills(&start, &end)

In this case, we are using a more complex structure (an SDT with one Quantity element, and a collection of Bills).

About the language

- Basic components:

- Groups
- Elements
- Variables

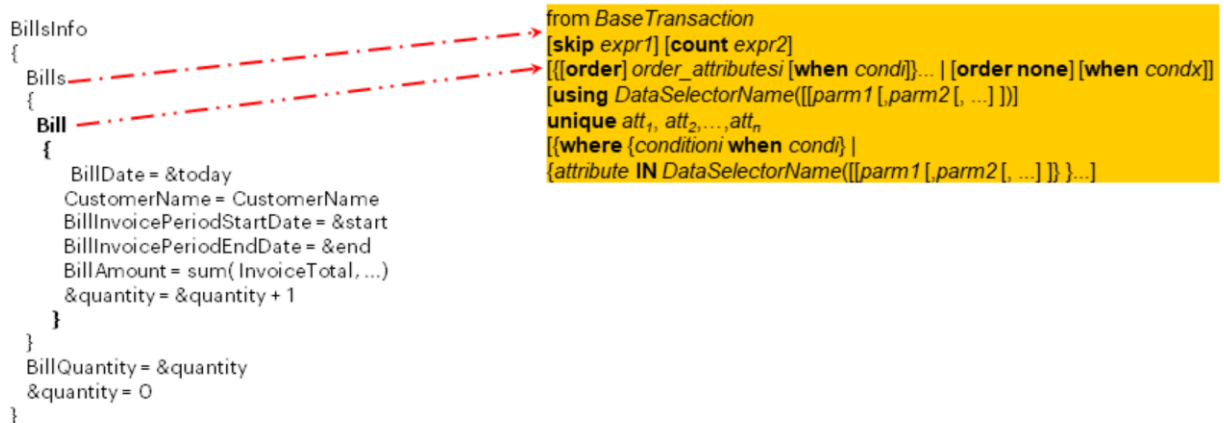


```
BillsInfo
{
  Bills
  {
    Bill
    {
      BillDate = &today
      CustomerName = CustomerName
      BillInvoicePeriodStartDate = &start
      BillInvoicePeriodEndDate = &end
      BillAmount = sum( InvoiceTotal, ... )
      &quantity = &quantity + 1
    }
  }
  BillQuantity = &quantity
}
```

Here we can identify the basic components in the language of Data Providers.

About the language

- A repetitive group is analogous to a *For Each* command:
 - Determines base table (in the same manner as a *For Each* command)
 - It has available the same clauses as a *For Each* command :



In the example, the group with the name *Bill* will be repetitive. Why? We can answer this question with another question: what would happen if it was a *For Each* command, where the elements to the left of the assignments correspond to the various elements of an SDT variable? In such case, the presence of *CustomerName* (to the right of the second assignment) enables us to affirm that there is a base table. We want to iterate on the *CUSTOMER* table, so we write the clause "from Customer" in an analogous manner as we would do in the case of a *For Each* command.

Note that, on the other hand, the group with the name *BillsInfo* will not be repetitive, and it does not have any associated clauses, while the elements contained in it are defined on the basis of variables instead of attributes:

```

BillQuantity = &quantity
&quantity = 0

```

And what about the *Bills* group? Note that, in this case, the group only contains another group. The contained group will be repetitive, so *Bills* will be a collection of *Bill*. Therefore, the *Bill* subgroup may be omitted (leaving only *Bills*) so that it will be implied. This is how the clauses in the group that enable the definition of order and filters may be associated with this group.

About the language

- The groups may be repeated in the Source:

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

The result returned will be a collection with N+1 items: where N is the number of customers in Mexico.

If the condition were placed in the Clients group, then it would apply to the two Client subgroups. And that is why clauses are allowed to operate at the level of the repeated groups (items) instead of only at the level of the group that is collection of items.

Other examples of the language

- Use of paging parameters and clauses

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

- Use of variables, invocation to another Data Provider, Input clause

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

What has been considered in this course is not all that can be said on this subject. For instance, the variables used may be loaded from another Data Provider, and we may use specific clauses, as the Input, among other aspects.

You will find full information on the language of Data Providers at: <http://wiki.genexus.com/commwiki/servlet/wiki?5309,Toc%3AData+Provider+language>

And full documents about this object at: <http://wiki.genexus.com/commwiki/servlet/wiki?5270,Category%3AData+Provider+object>,

GeneXus™

The power of doing.

Videos

training.genexus.com

Documentation

wiki.genexus.com

Certifications

training.genexus.com/certifications