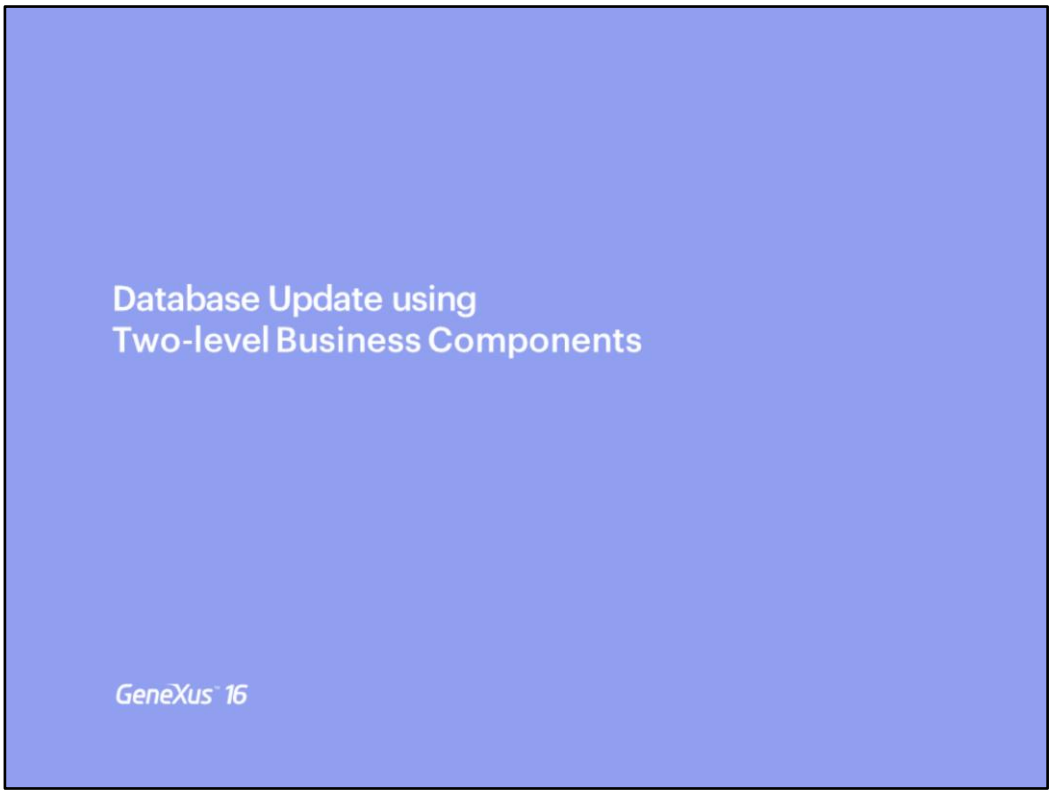
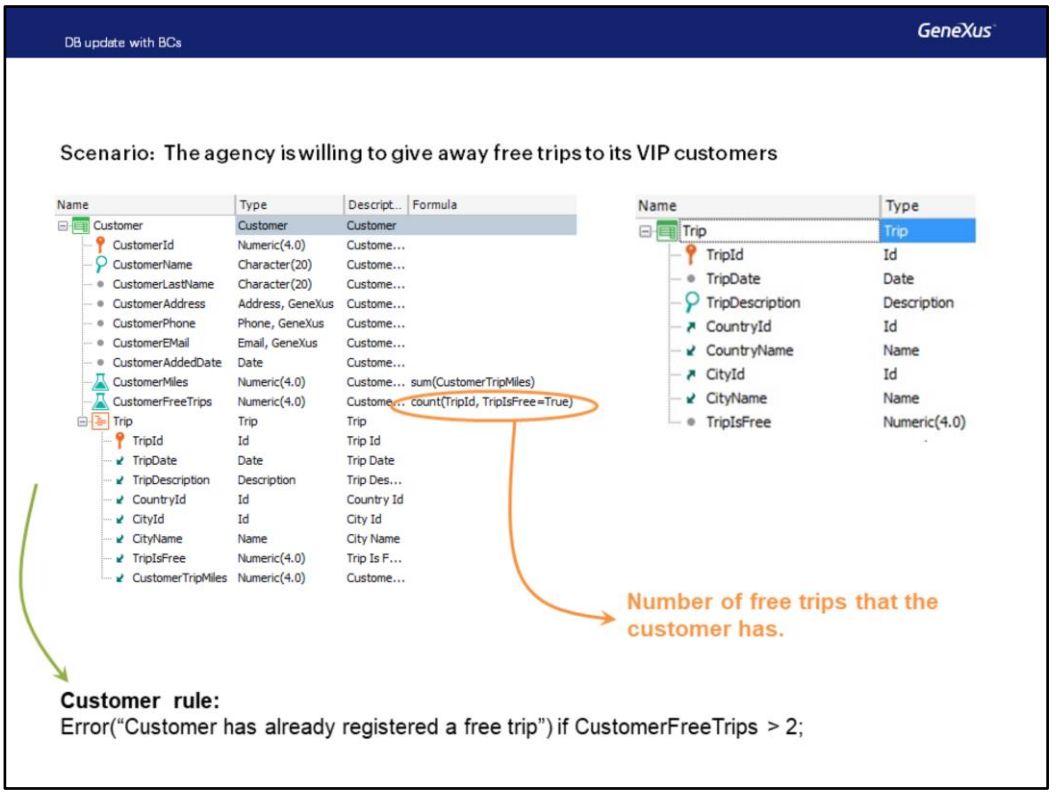



We have already dealt with the general concept of Business Component and how it applies to a single-level transaction.

Now we will see what happens when we create the Business Component for a two-level transaction.



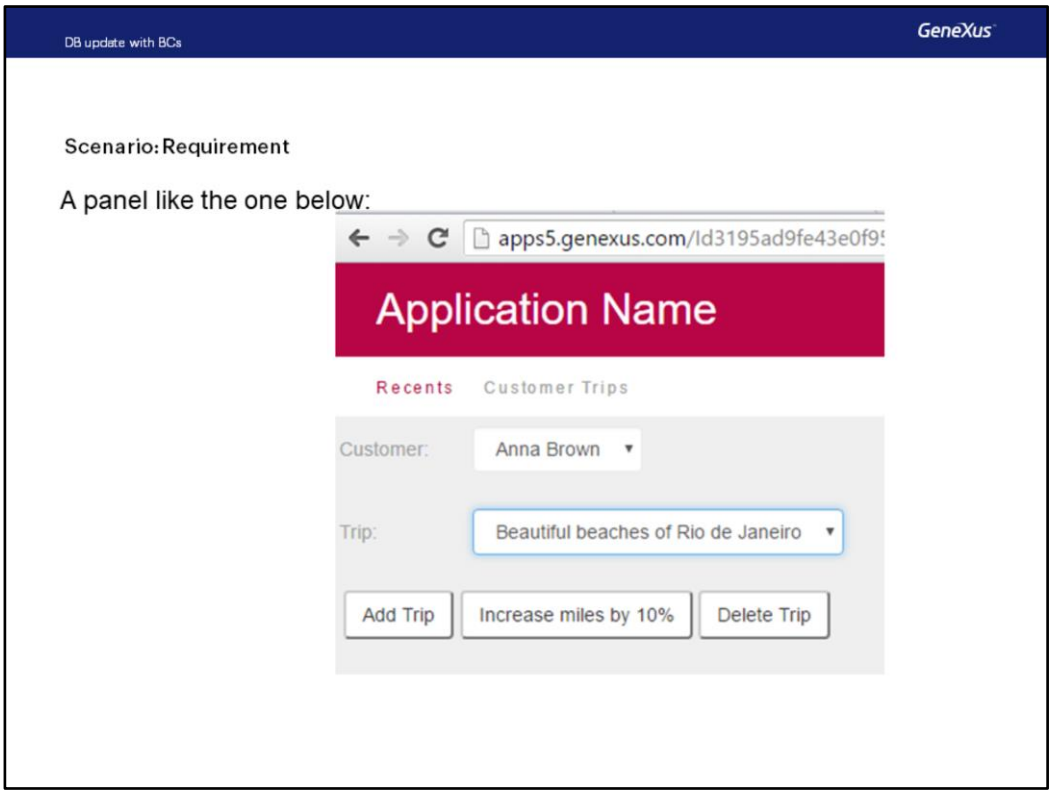

The Agency provides the possibility of trips to different cities offering visits to tourist attractions. Every once in a while, the Agency gives away a free trip to its VIP customers.

For each customer, the system must record the trips that the customer has bought, in addition to the free trips provided as the agency's prize. The agency gives away up to two trips per customer.

In order to record the agency's trips, we will use a Trip transaction to enter information through the Boolean attribute TripIsFree for when the trip is free, or not. To record a customer's trips, we add a second level to the Customer transaction with TripId as identifier, the attributes inferred in the Trip transaction, and CustomerTripMiles as the only secondary attribute, to record the miles accumulated by the customer in that trip.

For controlling that a maximum of two free trips are granted to each customer, we declared the formula attribute CustomerFreeTrips and the following rule:

Error("Customer already has registered a free trip") if CustomerFreeTrips > 2;



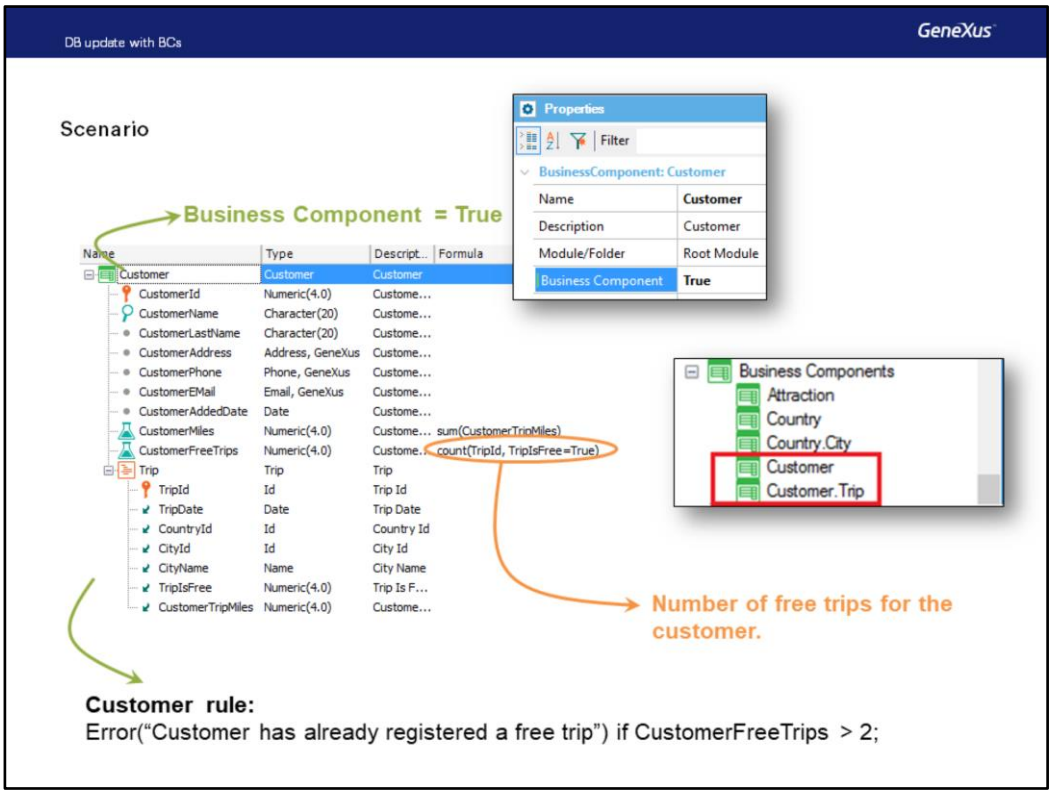

The user will get a panel like the one above allowing the selection of a customer through a dynamic combo, and the selection of a trip through another dynamic combo.

After we select the customer and the trip, we will face three possible operations:

- Add Trip: to try adding the trip selected to the specific customer. If the trip is free of charge and 2 free trips have already been assigned to that customer, then the insertion cannot be allowed, and there should be a screen message telling us about that error.
- Increase miles by 10%: to try increasing mileage by 10% when the trip selected has been recorded for the customer. Otherwise, a screen message will tell us that the operation could not be done.
- Delete Trip: to try deleting the trip selected from the customer record. When the operation cannot be performed (as in the case where that trip is not found for the customer) then a message must inform the user regarding this problem.

The screen shown will be implemented with a web panel, a subject we will be studying later on, including functioning details that we will not be considering now.

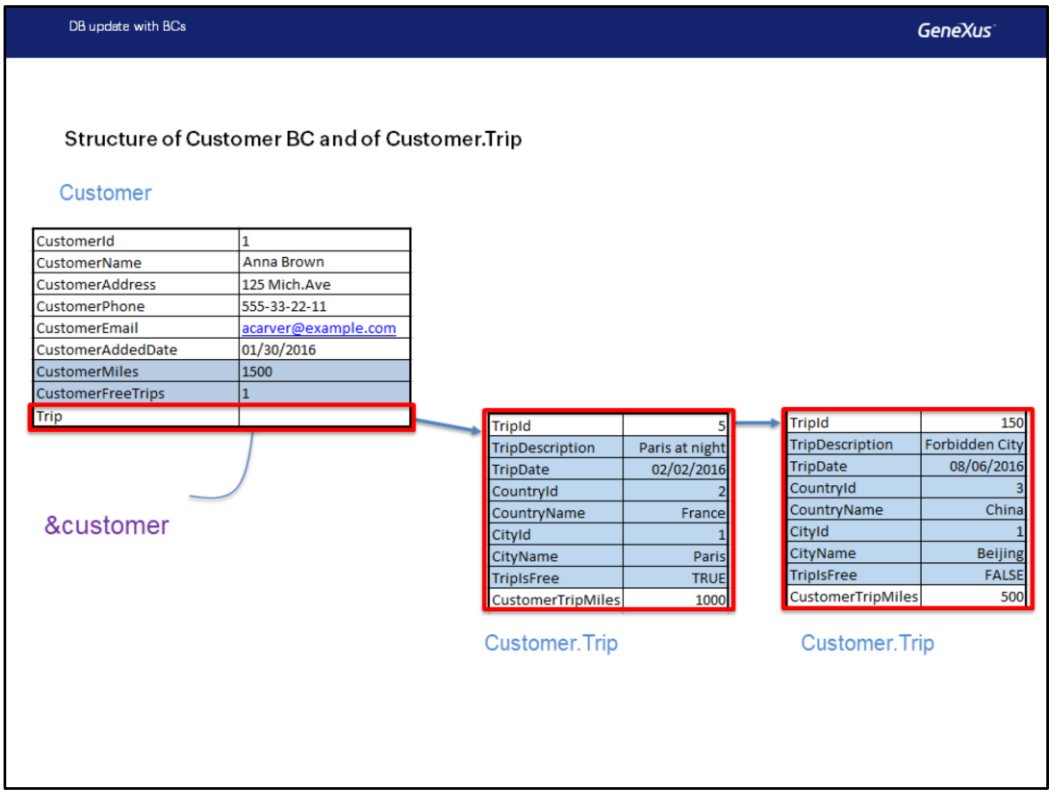
At this time, we will focus on how to program the code corresponding to each button, that is, how to perform the insertion / modification / deletion applicable to a line in the Customer transaction, though without using the transaction.




We create the Business Component associated with the transaction by declaring the value of its Business Component property as True.

We know that, upon saving, GeneXus will create the Customer data type associated with the transaction. In this case, because the transaction has two levels, it will also create the Customer.Trip data type specifically associated with the lines in the second level, which in our example correspond to the customer's trips.

We will find it in the KB as any business component data type. However, the difference with the Business Component data type corresponding to the transaction (Customer in this case) -which enables us to perform operations as if it were the transaction-, the second one only enables us to define the structure of a line in the transaction, but it is not associated with any specific methods such as Insert, Update, Delete, Save, etc.



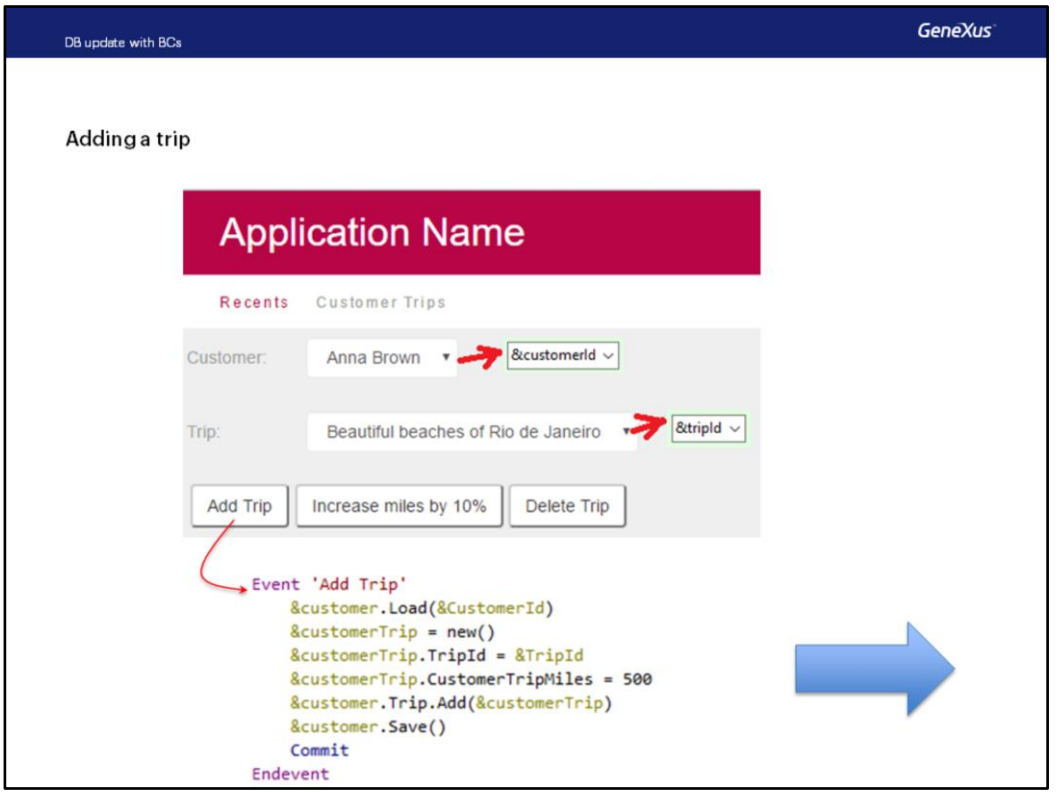

Here, we can see the structure of the Customer Business Component and of Customer.Trip. If we look closely, we will see the &customer variable, of the Customer type, with an element for each attribute in the transaction header, and with an element for the collection of lines. So, we will have the following:

&customer.CustomerId based on the CustomerId attribute, of the Id (Numeric) type
&customer.CustomerName based on the CustomerName attribute, of the Name (Character) type
…
&customer.CustomerAddedDate based on the CustomerAddedDate attribute, of the Date type
&customer.CustomerMiles based on the CustomerMiles attribute, of the Numeric type
&customer.CustomerFreeTrips based on the CustomerFreeTrips attribute, of the Numeric type
&customer.Trip, of the **collection type,** of **Customer.Trip**

Note that the attributes CustomerMiles and CustomerFreeTrips, which in the transaction are formula attributes, are used only for querying information on a customer already loaded in the &customer variable (as in the example shown above).

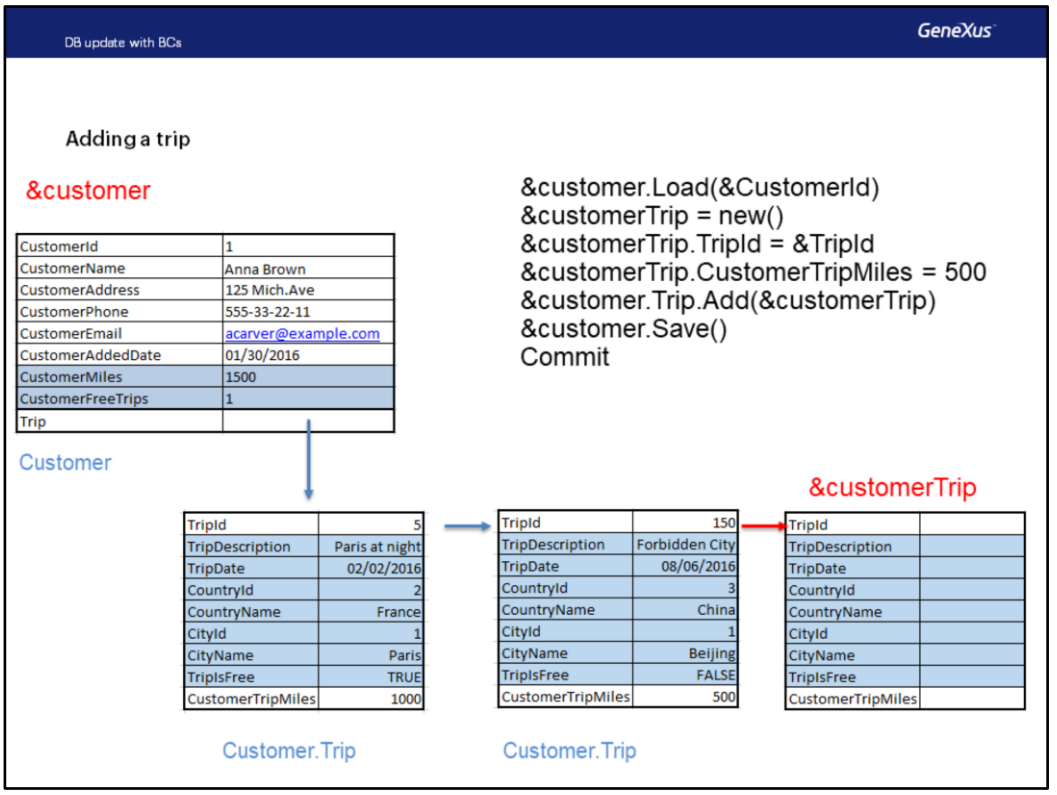
The Customer.Trip data type relates to a structure with the elements corresponding to the Trip level attributes of the transaction. Again, the attributes inferred in that second level of the transaction will be query attributes only.

As we will see ahead, in order to insert a new “line”, or to modify an existing line, we will have to use a variable of the Customer.Trip data type.




Leaving aside details about the web panel, the combo asking the user for the customer will be a &CustomerId variable based on the data type of the CustomerId attribute (which will show the data of the CustomerFullName attribute). And the one asking for the trips will be another variable &TripId, based on the data type of the TripId attribute (which will show, in runtime, the data of the TripDescription attribute).

Let's start by the action associated with the "Add Trip" button. This will be the code, which we will go on to explain now.




If we worked with the transaction, then the first thing to do would be to find the customer with which we will work. To do that, we **load** (with the Load method) the &customer variable (of the Customer Business Component type) with the value selected by the user in the &CustomerId variable on screen in the web panel. In the example, the customer of Id 1.

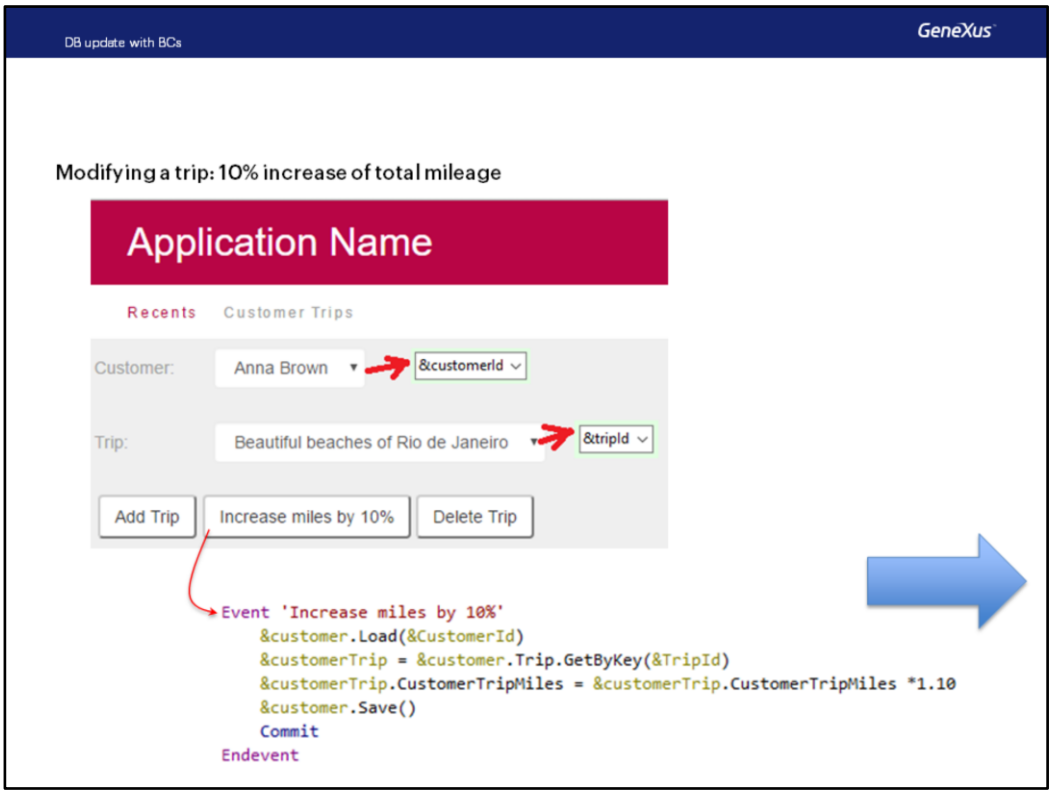
If we worked in the Customer transaction, to add trip we would have to add a line. This is done with the New method which enables us to create the &customerTrip variable with the data type of the items in the line, that is: Customer.Trip. We use it to insert the new trip for that customer. We fill in the values of TripId with the trip selected by the user in the panel, and saved in the &TripId variable, and we assign the value of miles for that trip -500 in this case.

As opposed to when we work with the transaction where we have the lines available on screen, in this case we have to explicitly add that item to the customer's collection of lines. To do this we use the Add method (of collections). We must remember that &customer.Trip is a collection of items of the Customer.Trip type.

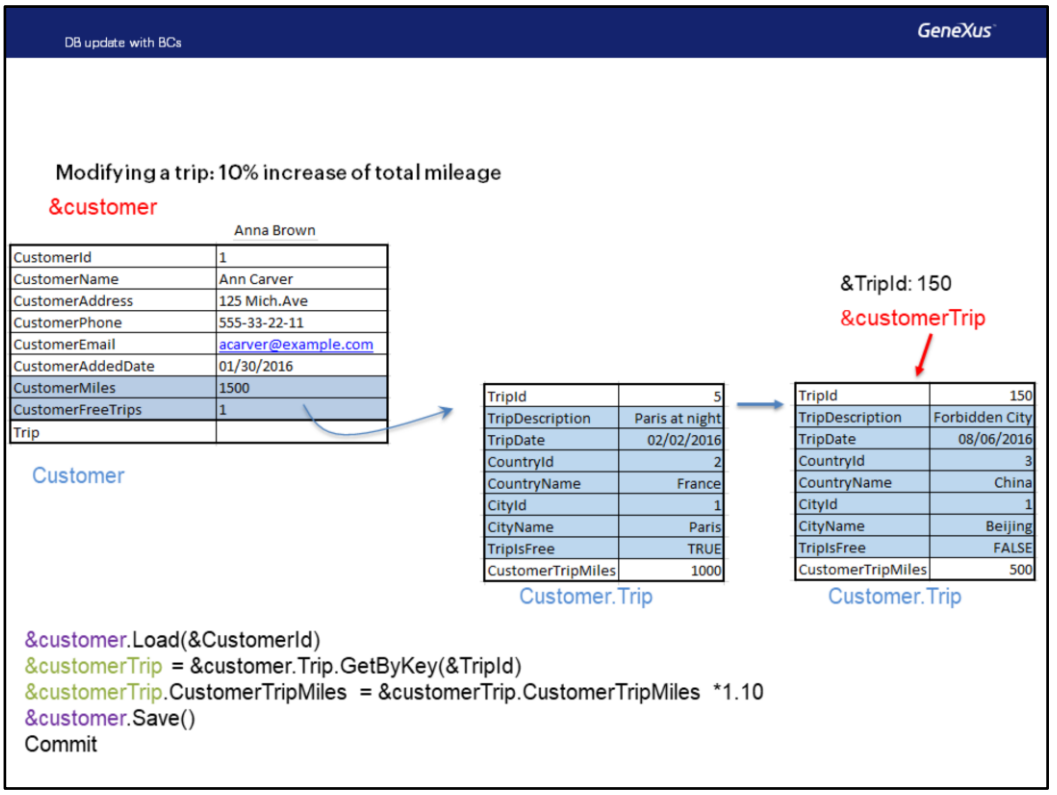
And last, and as we would do in the transaction, we must confirm the operation, for which we add the Save of the Customer business component and the Commit.

**Note**: instead of the Save method, in this case we could have used the Update method, because even when we are inserting a new line, we are in fact doing an update at the customer level. This is why the Insert method would not be the right one, since these methods are being applied to the variable corresponding to the header.

If it is not possible to complete the operation, we will have to see how to manage errors.




Now let's take a look at the action associated with the "Increase miles by 10%" button. Here we have the code that will be explained in the next page.

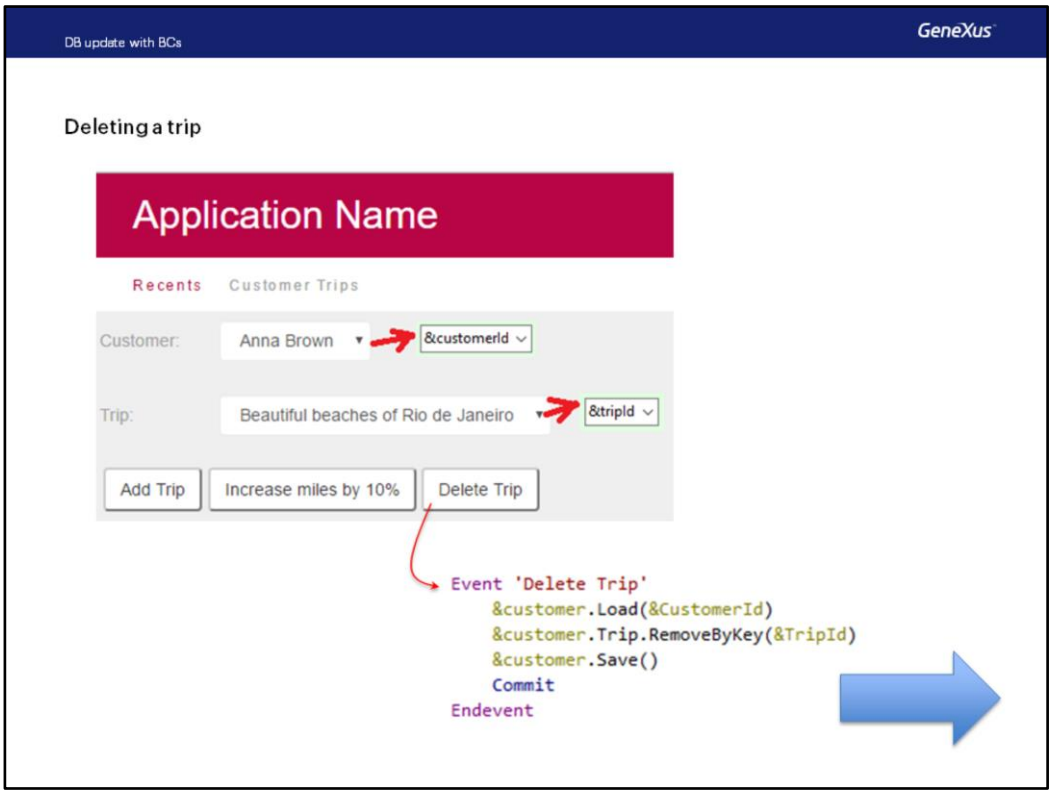



1. As we would do in the transaction, we must load the customer whole mileage we want to update for one of its trips: Load.
2. We must access the trip that we want to modify by using a method from the collection of lines called GetByKey, to which we pass the line identifier and the return is that line in the &customerTrip variable. It is not a copy of the line, but the line itself.
3. Then we modify the value desired – the trip's miles in this case.
4. We Save, and
5. Commit

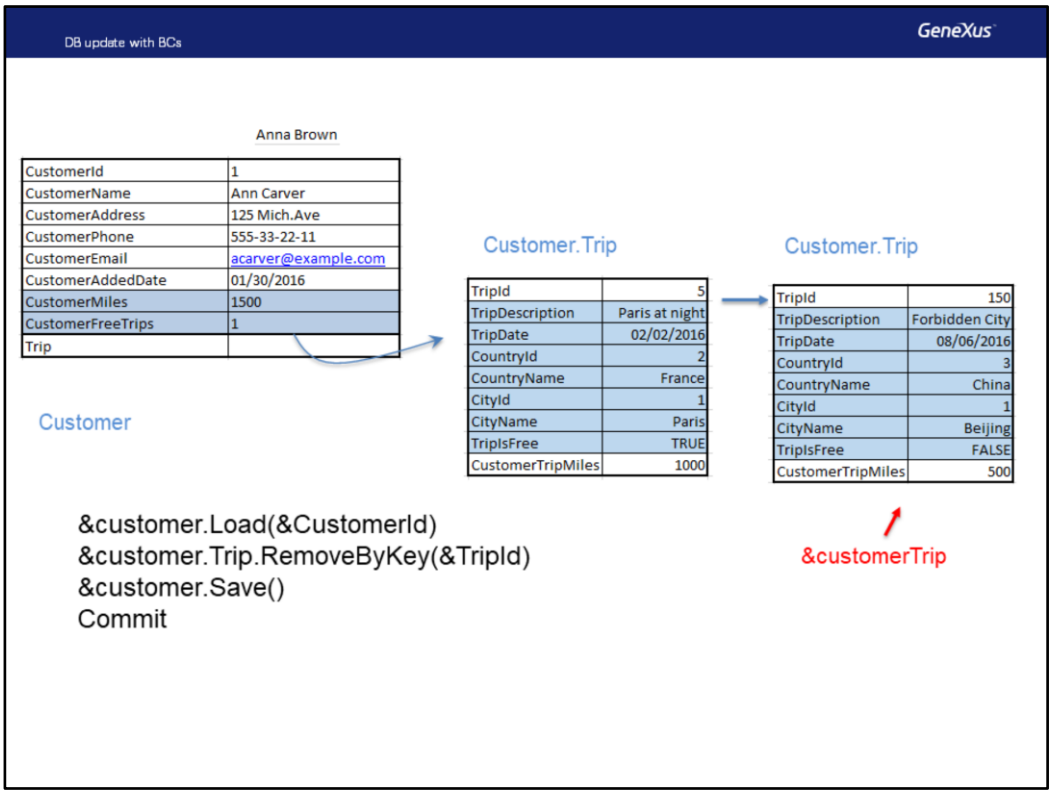
**Note**: if we had any other secondary attribute to which we did not explicitly assign a value, then it will keep the value it had previously. This means that values are assigned only to the elements that we want to modify.

The concepts we considered before in relation to the Insert and Update methods instead of Save also apply here.

We will later see how errors are managed for the case where the operation cannot be completed

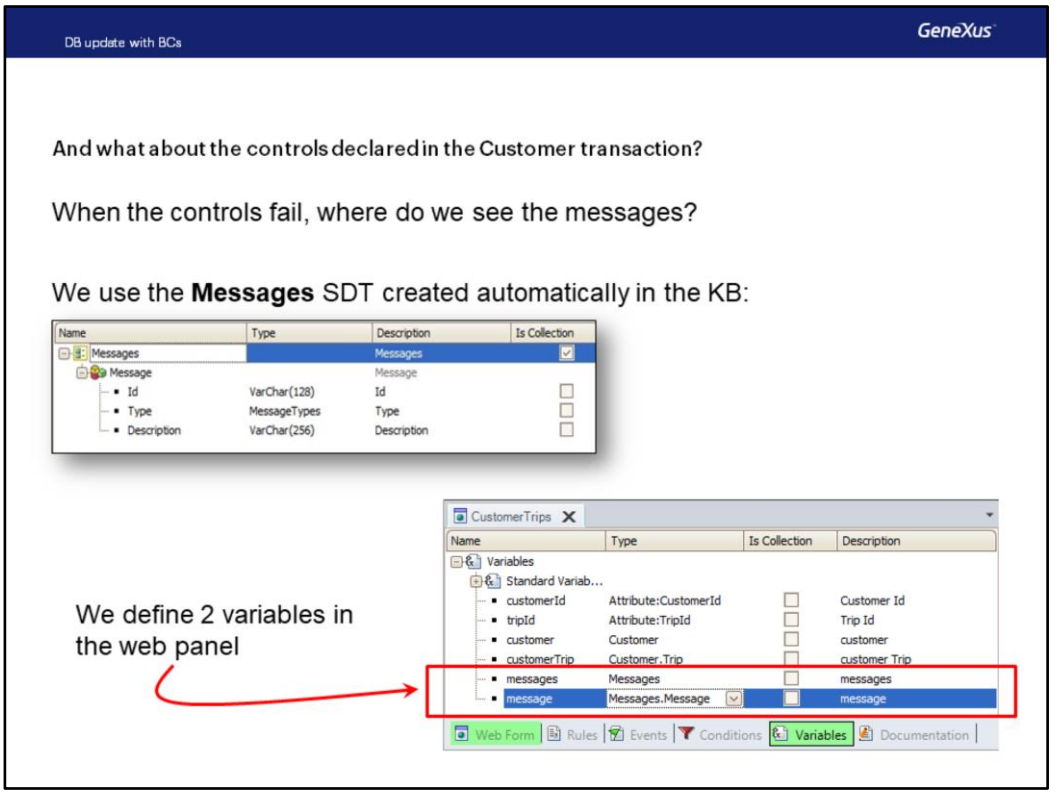



Let's now see the action associated with the "Delete Trip" button. Here we have the code, which is analyzed in the next page.




To delete a line from the transaction we stand on the line and mark it for deletion. In the business component this is done with the RemoveByKey method of the collection of lines. We must pass to this method, by parameter, the identifier of the line that must be deleted.
For the line to be finally deleted in the transaction, we press Confirm. Here, we execute the Save method. Again, as we already saw, we could replace the Save with Update, though not with Delete because these methods apply to the header, which already exists.

We will now see how errors are managed for the case where the operation cannot be completed.




In the action for adding a trip, what would happen if the trip we intend to insert:
1) was already assigned to the customer?
2) is free of charge and the customer already had 2 trips in these conditions?

In both cases the insertion will fail. In case 1) because the BC will verify the unicity of the compound key of the record that we want to insert, and in case 2) because the BC will trigger the error rule that was specified in the transaction to control the number of free trips.

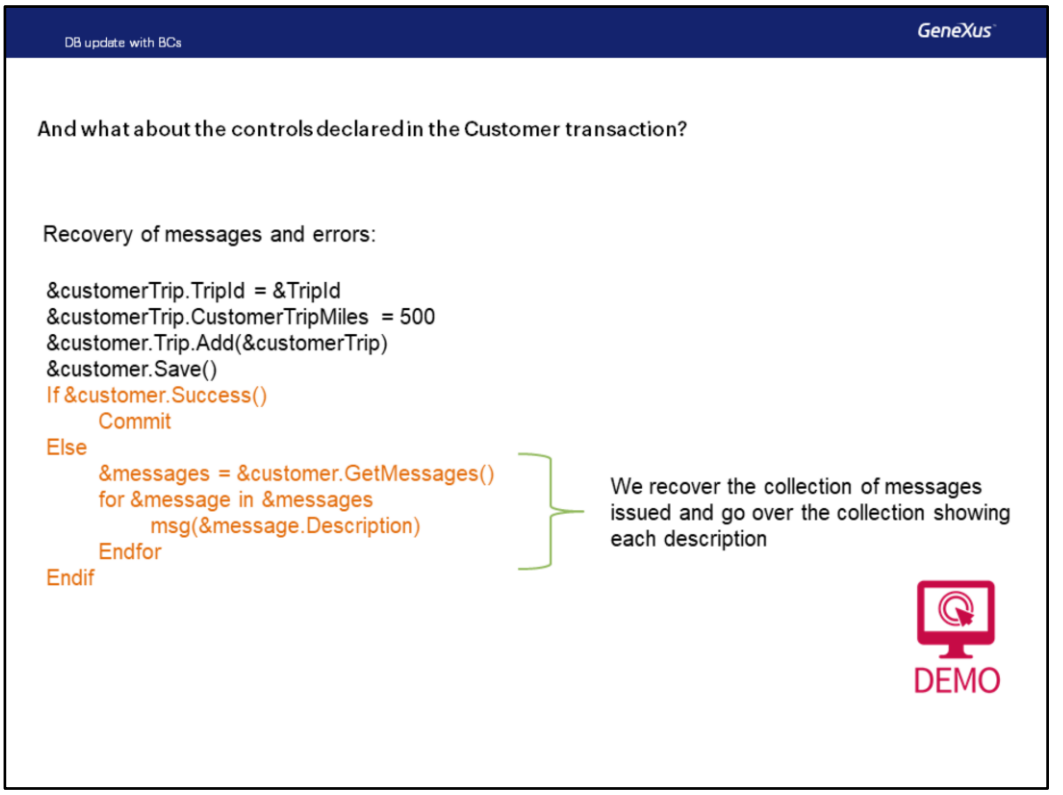
Even when this control is triggered automatically when we apply the Save() method to the variable in the Business Component, if we want the end user to be aware of this, we must obtain and explicitly show the messages issued.

What messages? Those issued by GeneXus due to controls of data consistency, unicity in the values of primary keys, and the messages relative to our own Error and Msg rules.

So, how can we recover the messages occurred?

We use the Messages structured data type that GeneXus defines automatically upon creating a new KB. The purpose of this **collection** data type is to enable our access to the messages issued in the execution of a Business Component.

Therefore, we will be defining two variables:
&messages (of the data type Messages, and collection) and
&message (of the Messages.Message data type, which represents an item in the collection).




When we apply the GetMessages method to the &Customer variable, we obtain the collection of messages that occurred.

And last, by using the "**For item in collection"** structure, we go over each message in the collection of messages variable and deploy its description.

The same colored code would be added to the previous examples when a trip is modified or deleted.

To learn more about using Business Components go to:

- http://wiki.genexus.com/commwiki/servlet/wiki?5846,Toc%3ABusiness+Component,

GeneXus™

The power of doing.

| | |
|---|---|
| Videos | training.genexus.com |
| Documentation | wiki.genexus.com |
| Certifications | training.genexus.com/certifications |