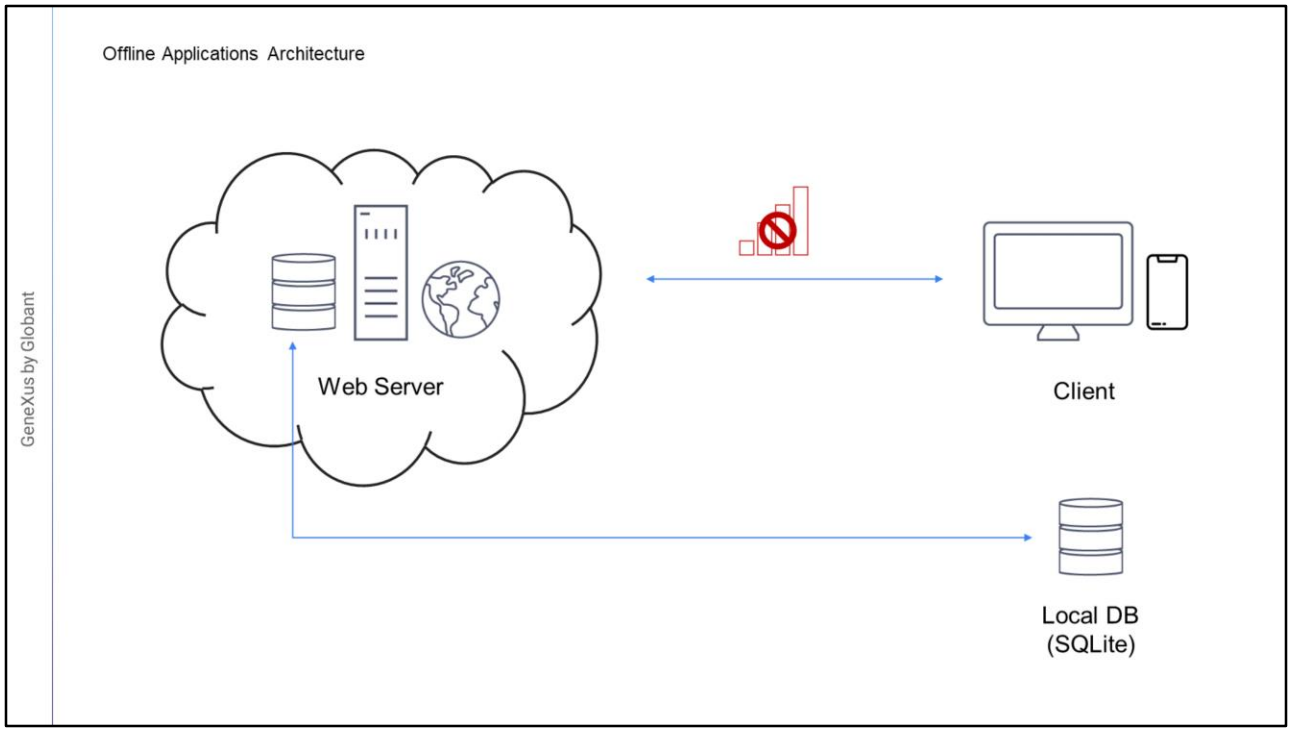
GeneXus by Globant

Offline Applications Architecture

Diego Marranghello



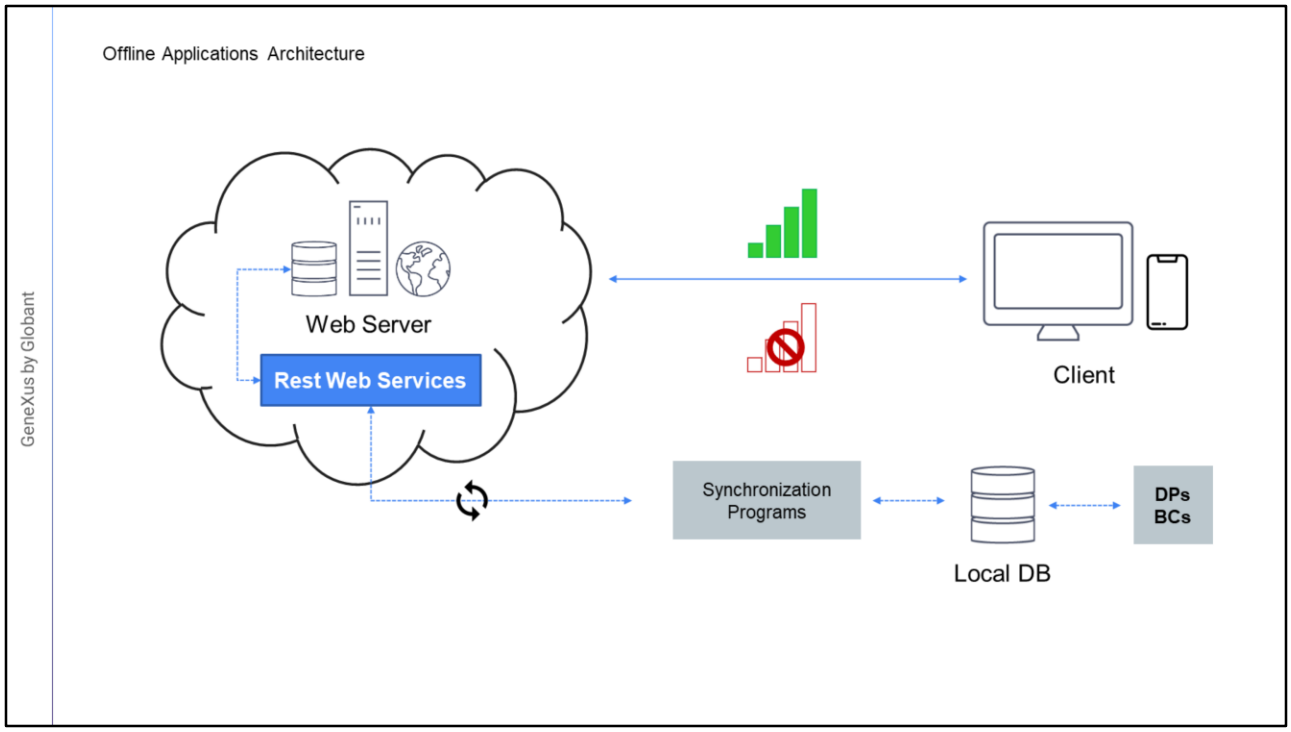
In this video, we'll talk about the architecture of offline applications.

We'll start by talking about offline applications. In this case, we want all the data managed by the mobile application to be accessible even when there is no Internet connection.

Here, the structure of the database centralized on the server that is managed by the mobile application is mirrored on the device. That is to say, a SQLite local database will be created on the device, with the same tables.

However, replicating the entire data set of the centralized database is not mandatory. Instead, a subset of this data can be sent to the device database, according to some condition that can depend on the users, the device itself, etc.

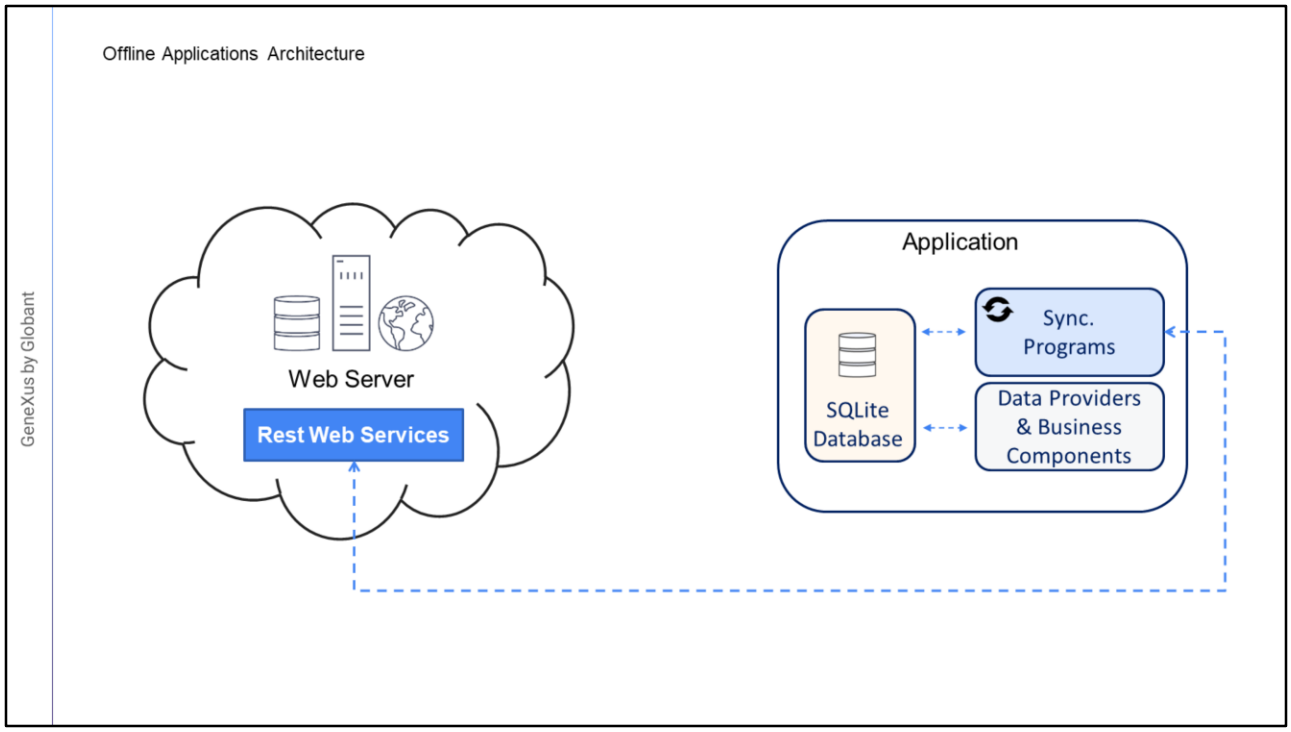
That is to say, there are mechanisms to specify filters on the data to be sent to the local database. In addition, not all the tables will be included, only the necessary ones. We'll talk about all this later when we study the OfflineDatabase object.



In offline applications, in addition to a local database, all the programs (data providers and business components) used to obtain information from the central database are required. Now, they should be programmed in the languages of the platforms for mobile devices, so that they can access the local database.

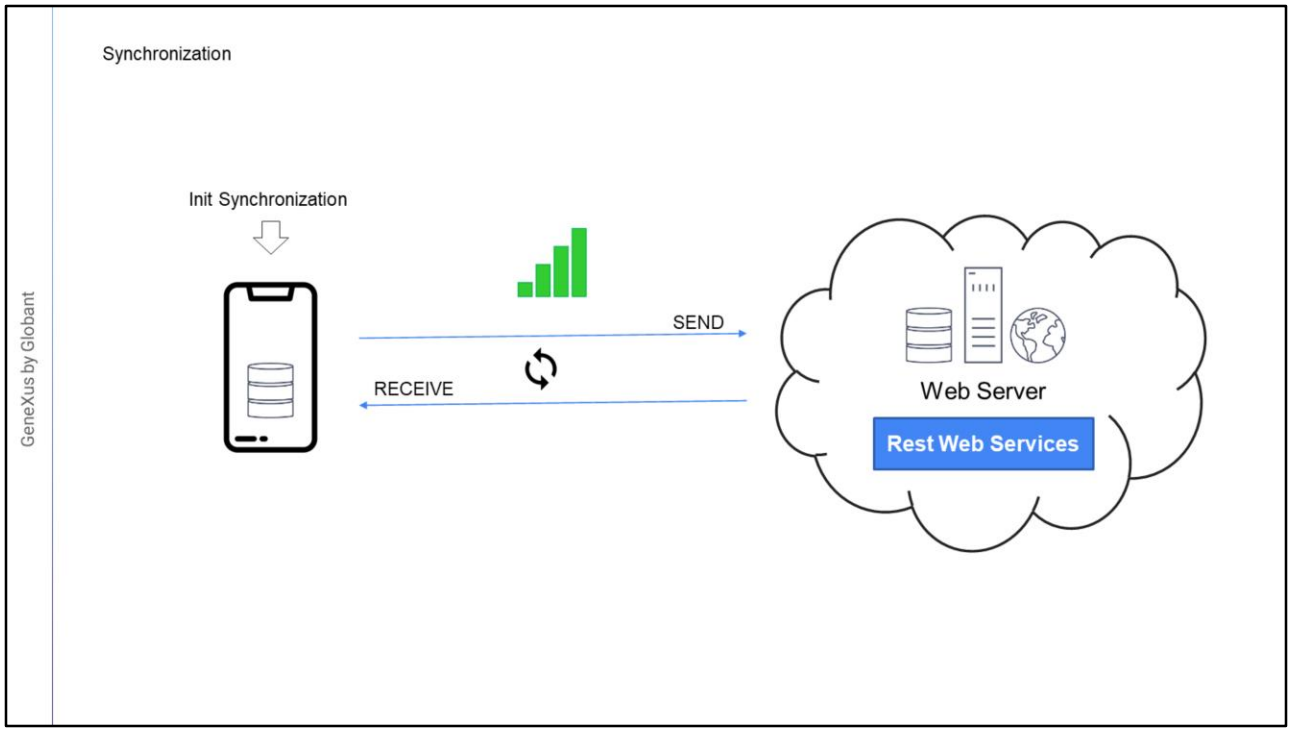
From now on, regardless if there is an Internet connection available, the application will always work on the local database.

The application on the device will access the server only to synchronize data from both databases, which will be achieved using synchronization programs running on the device and using Rest services on the server side. These processes will always start on the device, either to send or receive data from the server.



The entire services layer that was stored in the web server, and which contained the data providers to retrieve the data and the business components to update the data saved in the tables, will now be in the device. They are implemented using the platform's language to access the local database, and compiled in the binary file.

In this way, all CRUD operations will always be performed on the local database. They will never be run on the server database. The application will contact the server only to synchronize.

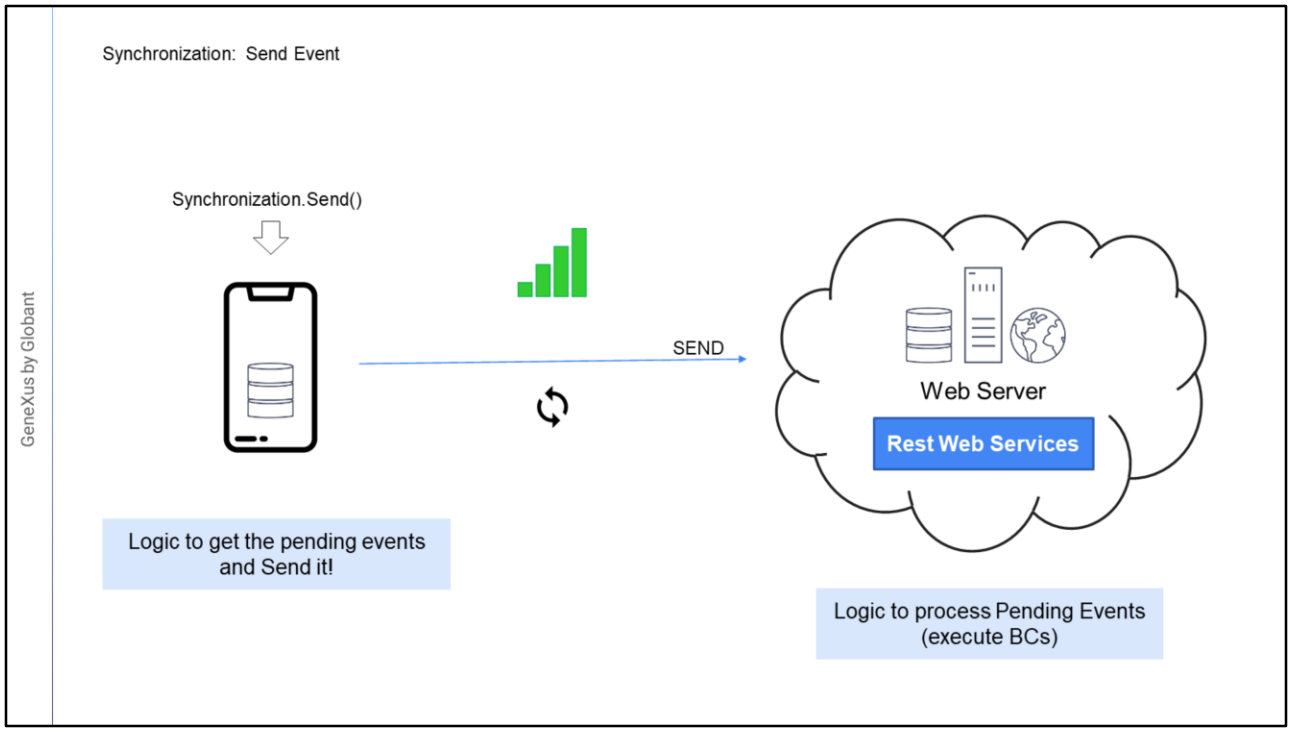


This synchronization will always be started from the device, because the server cannot know when the device is connected.

Optionally, the information locally stored can be synchronized with the data on the server; remember that we may never synchronize or do so on demand.
The process through which the data changed in the device is sent to the server is called Send.

The data on the server that has changed is sent to the device to be updated, at regular intervals or on demand.
The process through which the data changed in the server is sent to the device is called Receive.

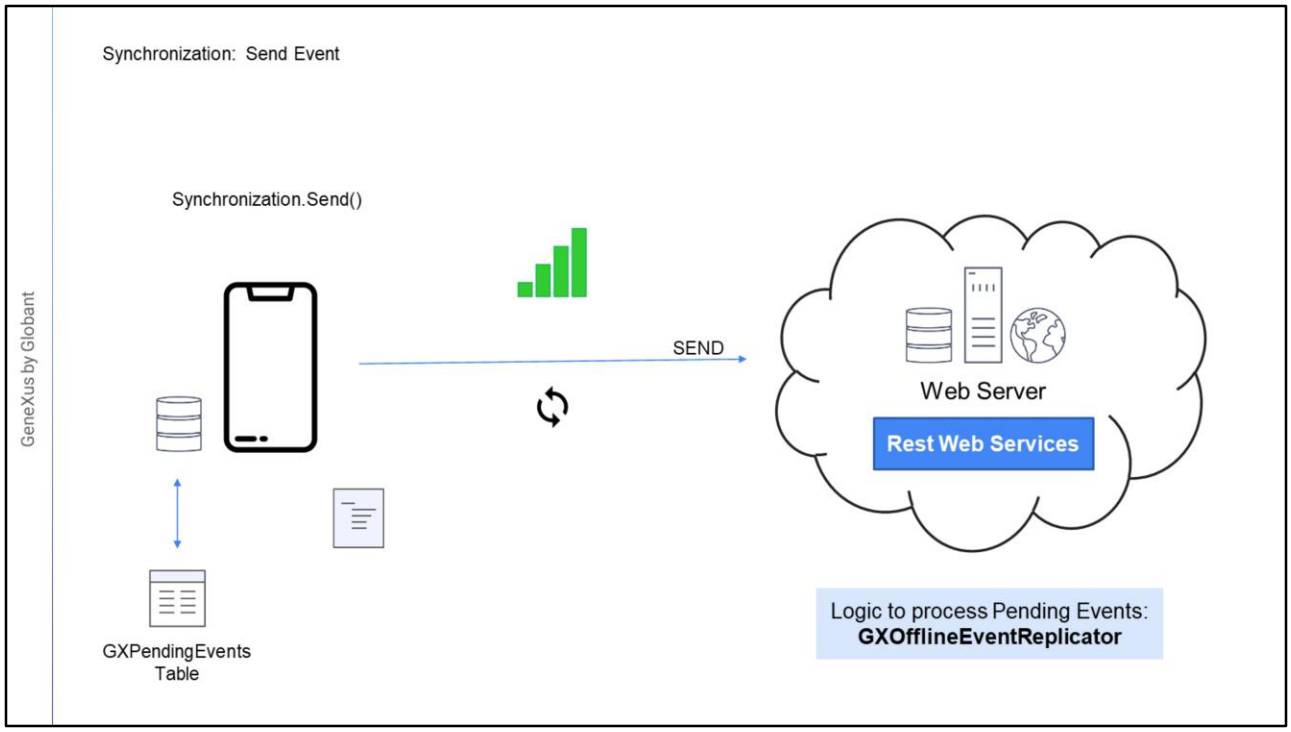
The device and the server, as already mentioned, communicate through the Rest Services layer.



Both Send and Receive operations are implemented with server-side and client-side logic. The objective is to have the client run the least processing possible, because it has less power than the server.

When the device starts the Send operation (which can be started manually when the connection is restored through the Synchronization.Send method or never): it must have built an ordered list of the insert, update, and delete operations performed since the last synchronization. That is to say, those operations in pending status.

This list is sent to the server-side process, which must run through the list in order, and execute the corresponding operation over the database, returning the result to the client-side process.



Remember that regardless if there is an Internet connection available, the client will always work on the local database.

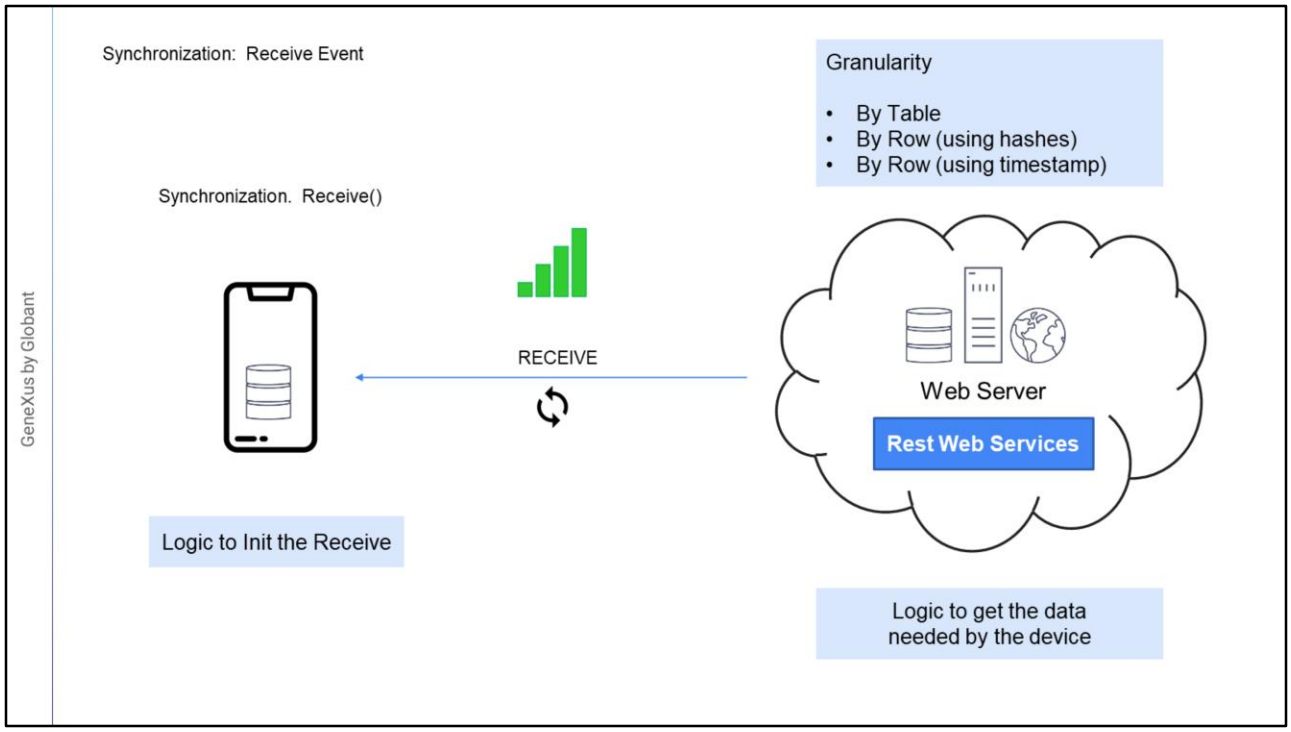
All changes made over the local database are saved as “Synchronization events” in a table for internal use called GXPendingEvents.

This table stores in an ordered manner all the operations executed using Business Components.

The stored details include the name of the BC where the operation was executed, the BC JSON containing the event's data, the type of operation made (addition, deletion, or modification), and its status.

Every time the device runs a business component, the event is stored and is left in “pending synchronization” status.

When the Send operation is started, the client translates the list of all the events with “Pending status” into an SDT with JSON format and sends it to the server. The GXOfflineEventReplicator procedure is programmed in the server, which reads the SDT and performs the Insert, Update and Delete tasks following the order of the operations.

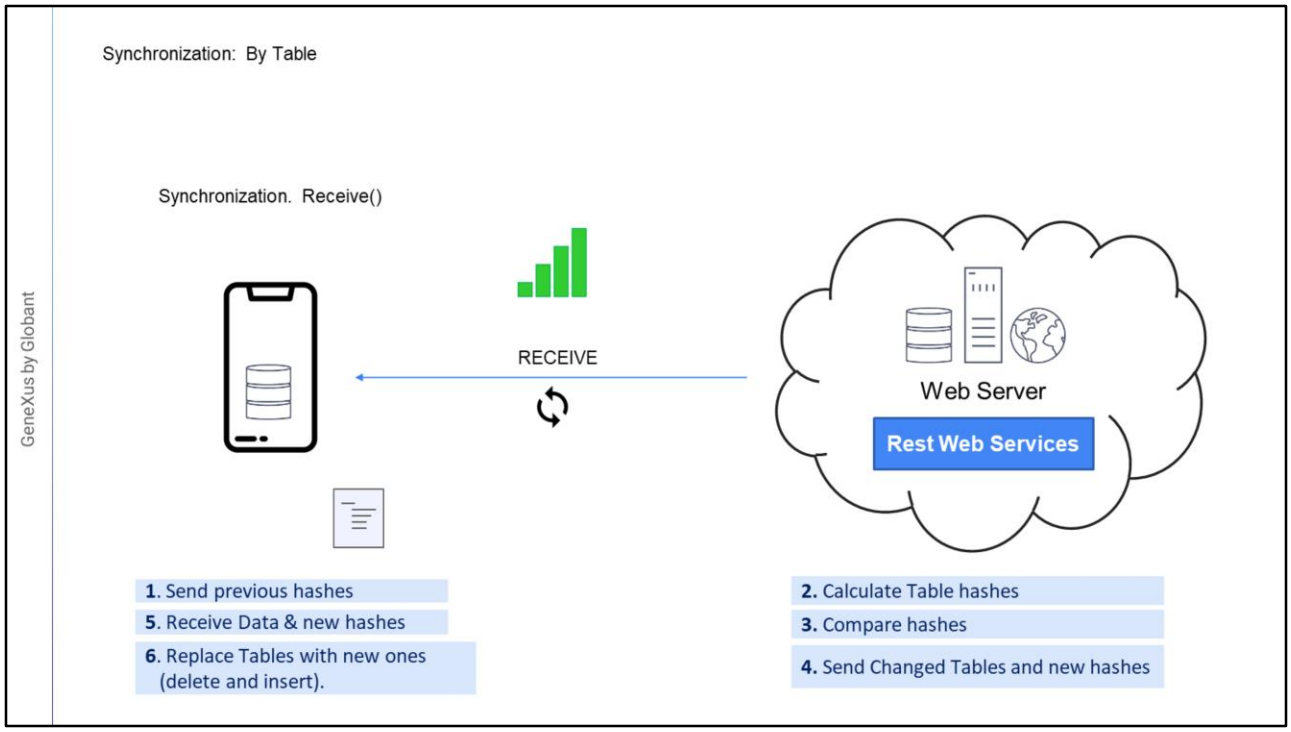


When the device needs to receive the data changed on the server, the Receive process is started by calling a Rest service on the server; then, the device updates the data on the local database.

The synchronization behavior can be configured using certain criteria that determine when the synchronization will be made to receive data.

In addition, the synchronization can be done in two ways: by Table or by Row. When this granularity is set By Table, all the tables changed since the last synchronization are sent to the device. When it is set By Row, only those records changed in each table since the last synchronization are sent to the device. There are two mechanisms, one that uses hashes and another that uses timestamp.

Next, we will look at each of these mechanisms and their differences.



Synchronization by table is useful in scenarios with a small number of records, or when changes are made very frequently, because in this case almost all the data has to be sent in each synchronization.

Its advantage compared to synchronization by row is that it requires significantly less processing on the server side.

To determine which tables were changed and therefore have to be sent to the device, a hash is used; it is the result of calculating a code that “identifies” the data set of each table.

When a client requests a synchronization:

The hashes of each table are sent to the server. They were sent by the server in the previous synchronization.

Next, for each table, the server calculates a new hash with the current data.

Also, it compares the hashes with the ones it currently has.

It sends data from the tables, when it determines that they were modified. If the table wasn't changed since the last synchronization, for this table no action is performed.

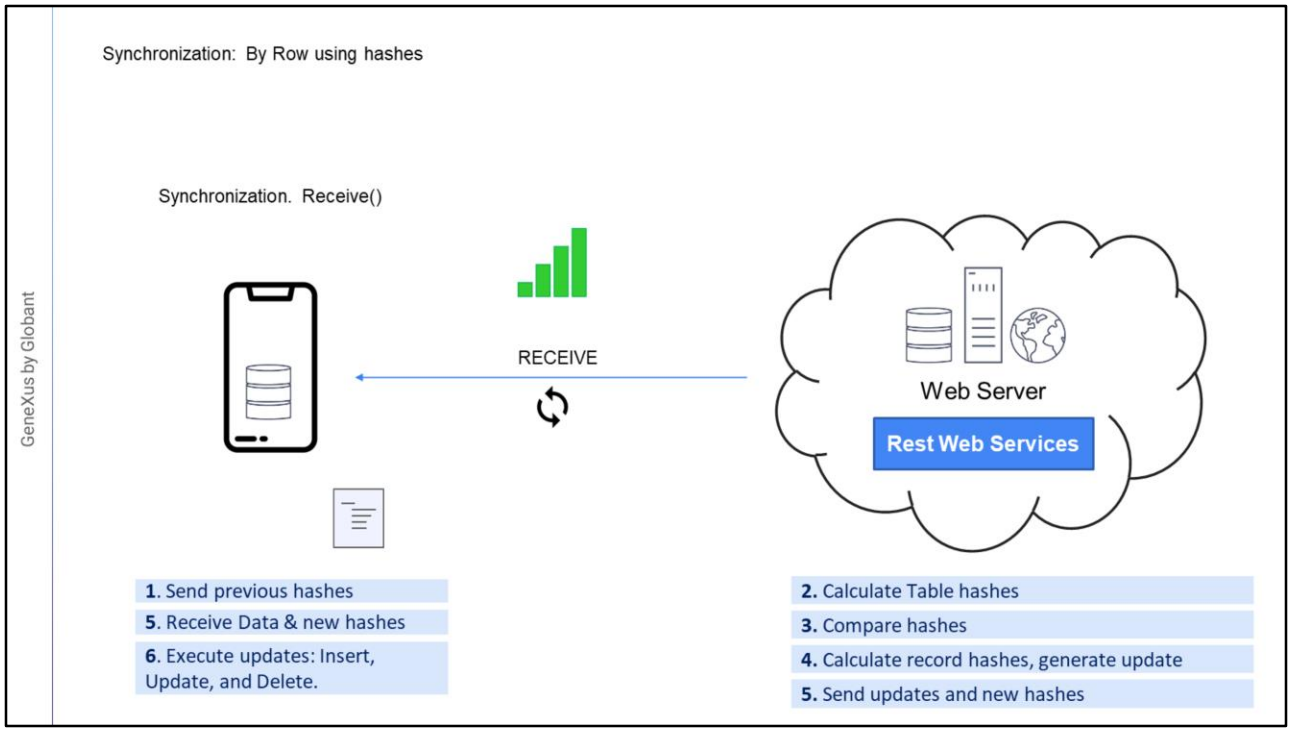
The device receives the data with the new hashes.

Finally, the device replaces the tables that were modified (it deletes the content and generates it again with the new information).

All communication is done using Rest services.

As a special case, in the first synchronization there is no data in the local database. So, all

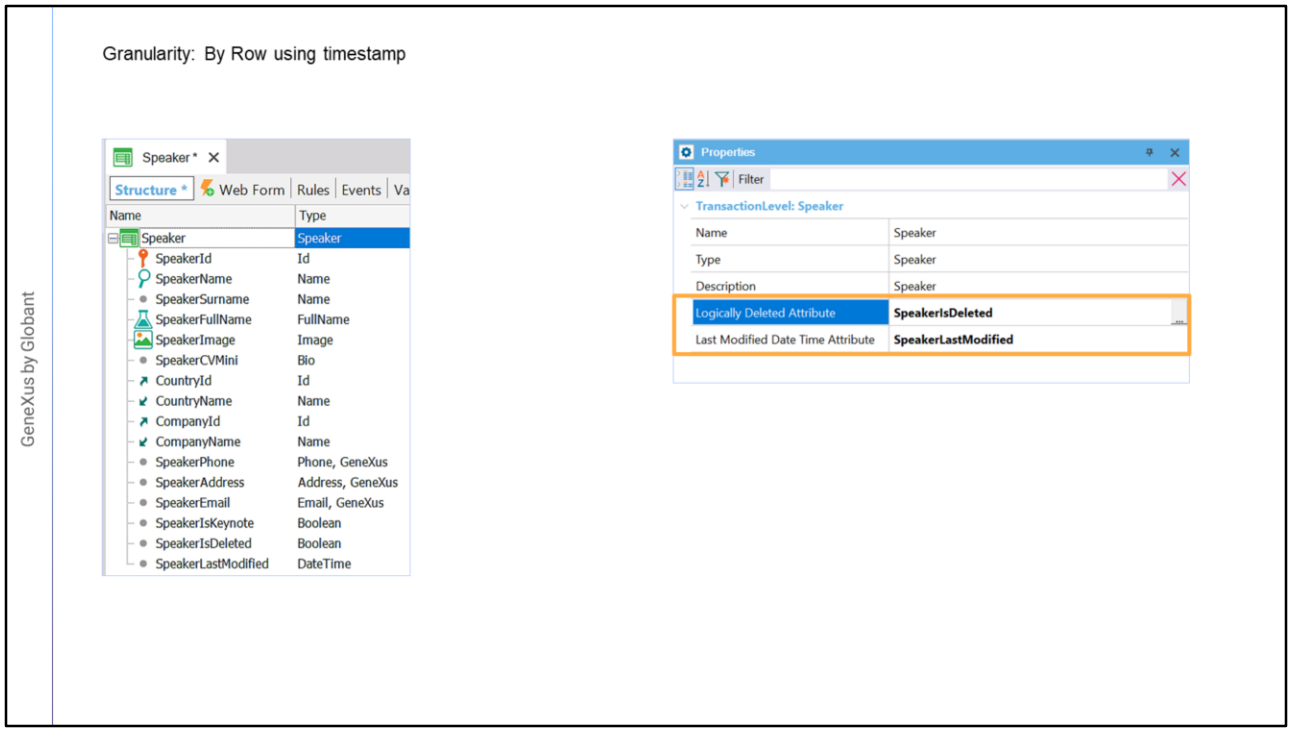
the data of all the tables that comply with the filters in the OfflineDatabase object and the hashes of each one are sent.



Synchronization “by row”, using hashes, only sends to the device those records that were changed since the last synchronization using the calculation of hashes to determine the updates.
The advantage is that less data is transmitted between the device and the server, since it only involves modified records. On the other hand, the disadvantage of this mechanism is that it requires more processing on the server side, especially for large volumes of data.

1. First of all, the device sends the tables' hashes to the server, just like in the synchronization “By Table.”
2. The server determines the data set that must be in the device, according to the filters that may exist, and calculates a hash for each data set.
3. Then, the server compares the new hashes with the current ones and determines which ones must be sent to the device. If there is nothing to send here the processing is finished.
4. For each data set that must be sent, the server calculates a hash for each record and compares them with the current hashes. As a result, the record may be the same, and in this case it will not be sent. If the hash is different, it means that the record has changed and that it must be sent as an update. Also, it may happen that this record doesn't exist in the previous set, so it will be sent as an insert. Lastly, a comparison is made to check which records existed in the current hashes that are not included in the new ones. These will be sent as deletions. For each action to perform –insert, update, and delete– a list is prepared.
5. The server sends the lists with the new data to the device.
6. The device then receives the data and saves the hash of each table.

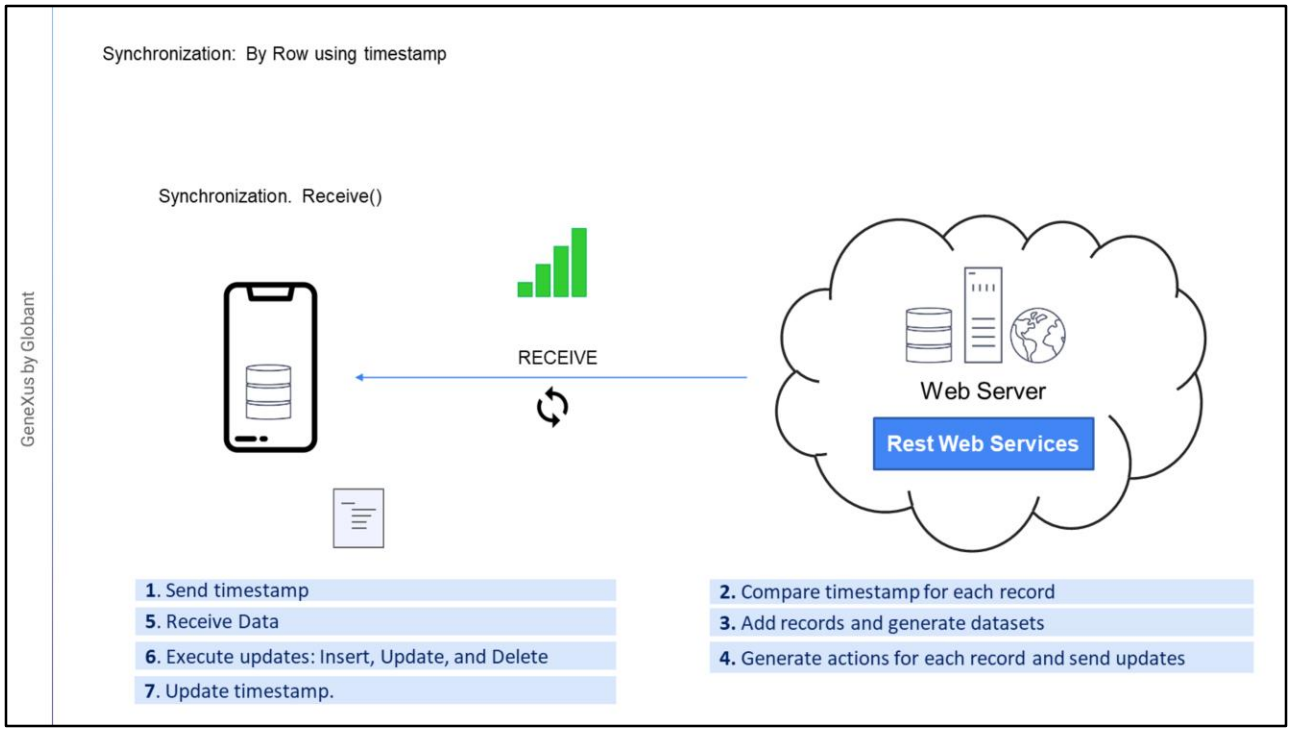
7. Finally, their lists are processed in order. For the new records, an INSERT is made in the database, and if it fails due to a duplicate key, an UPDATE is made. For the records that have been changed, an UPDATE is made, and if there are no records, an INSERT is performed to insert them. A DELETE is run for the deleted records.



Let's see the By Row mechanism using timestamp.

This mechanism uses a different approach. The synchronization is still done by record but no hashes will be used to determine if it is an update or not; instead, we will use the update date and the logical deletion mark.

For this reason, to use this mechanism we must indicate for each level of a transaction what attribute will contain the logical deletion and what attribute will contain the date and time of the last modification.

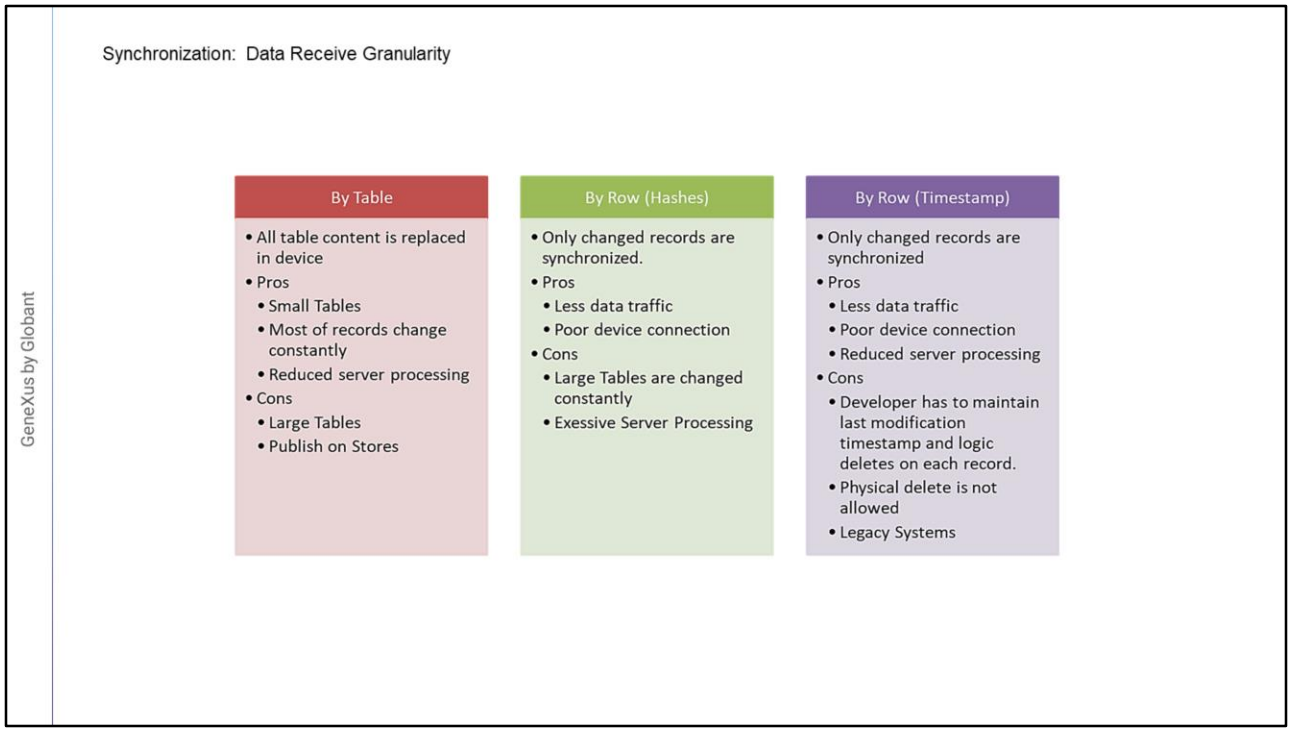


When using this mechanism, in the first synchronization the server will send all the data that meet the conditions, along with the synchronization timestamp. The device will store the timestamp that will be used later in subsequent synchronizations.

Next, in every new synchronization:
The device sends the timestamp of the last synchronization.
The server compares the timestamp received from the device with that of each record using the attribute of each table.
All the records that have been added or inserted after the timestamp of the device are added to a list.
To determine the action to be taken on each record, it will be evaluated whether the record was deleted, using the attribute with the logical deletion mark. If so, the record is marked as a deletion that must be sent to the device, and the rest of the records are marked to be updated. In the device, an “upsert” will be made: that is to say, if the record exists, it will be updated; otherwise, it will be inserted. All this information is sent to the device.
The device receives the data,
performs the operations, and updates the timestamp.

The disadvantages of this mechanism are that the developer is responsible for maintaining the timestamp and the logical deletion mark. In addition, we can't physically delete records in those tables.

This mechanism can be used together with the previous one, by row using hashes.



Let's review the synchronization mechanisms we've just seen.

Regarding By Table synchronization, when the server determines that a table has been modified, that table will be sent to the device and will be entirely replaced there. To determine the tables, a hash will be used for each table and for each device.

When is this mechanism useful? For example, when our tables are small or when most records are frequently changed, it is preferable to handle it in this way. We may use it in an internal mobile system where salespeople have the list of clients to visit, and that list changes every day –today, I'm assigned this list, and tomorrow I'll be assigned a totally different one.

This option has a disadvantage when the tables are too big, because the data traffic will be more significant. It would also not be advisable in cases where the application is going to be massively used; that is to say, when it will be published in a store. For example, if all the data is sent in every synchronization, the user experience is not going to be very good.

Synchronization by records is used to manage these cases.

Here we have two options: to do it using hashes or timestamp.

Let's see the first option.

In this case, the server will determine which hashes must be sent to each device according to the hashes it will calculate for each data set and for each record. Then only the new ones will be sent, only the records that have been modified.

The advantage is that data traffic is considerably reduced, unless a large volume of data is frequently modified.

Returning to the previous example, instead of receiving the entire clients table, we will receive only those that were changed.

Suppose we only receive the active clients; in this case, the device has a hash, that of the last synchronization, and to synchronize it sends that hash to the server, which recalculates a hash for the active clients and compares it with the one sent by the device. If they are different, the server will generate a hash for each record of that query and will compare them with the previous ones; in this way, it can determine exactly what was added, modified, or deleted.

This is the default option when an offline application is created.

This method is also useful when connectivity is not very good because it involves less traffic.

One disadvantage is that much more processing is required on the server side, so it would not be advisable if there are large volumes of data that is frequently modified.

Suppose that in this case our system has a table with thousands of products that we want to be in each device. In addition, there are hundreds of users. Synchronizing by table would not be advisable because the table has a large volume of data. Synchronizing by record using hashes may not be the solution either, because to determine what records must be sent to each device it is necessary to calculate and compare the hash of each one of the thousands of products entered. This, added to the presence of many concurrent users, can generate a server-side processing problem.

So, there isn't an optimal option. We'll see the third option, which involves using a timestamp.

When we use timestamp we will keep the data transfer to a minimum, only for modified records. Also, it involves less server-side processing, since it no longer has to calculate the hashes for the entire query, but can identify them by the date and time of last modification and by the logical deletion mark.

Here we need to maintain these two additional attributes, the logical deletion mark, and the date and time of last modification; the good news is that many systems already implement this data in each record.

The disadvantage of this mechanism is that the developer is responsible for maintaining precisely these two attributes, something that can be complex when there are many different systems involved.

We can also use a combination of the last two mechanisms: if we use By Row and in a transaction we don't configure timestamp and logical deletion attributes, hashes will be used.


GeneXus by Globant
GeneXus™
by Globant
training.genexus.com