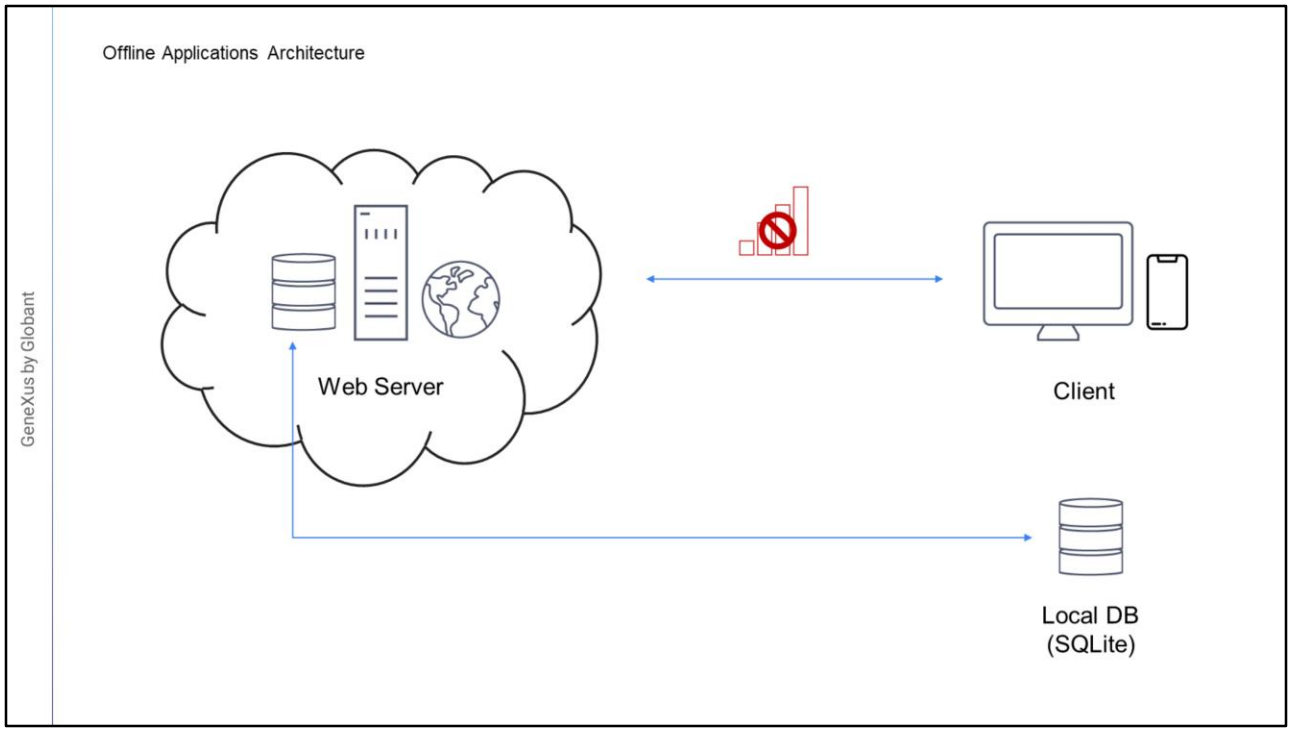
GeneXus by Globant

Offline Applications Architecture

Diego Marranghello



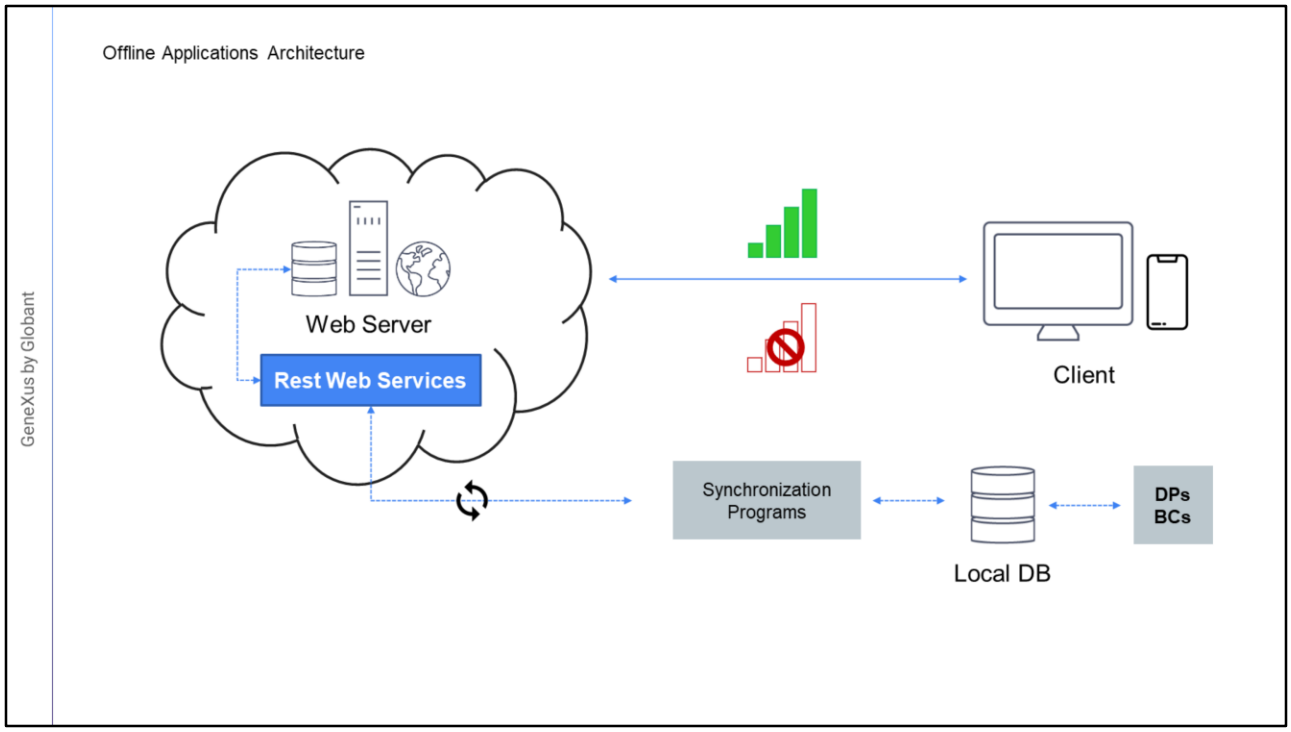
En este video hablaremos sobre la arquitectura de las aplicaciones Offline.

Comencemos por analizar el caso de las aplicaciones desconectadas, en el que queremos que todos los datos manejados por la aplicación móvil sean accesibles incluso cuando no hay conexión.

En este caso, la estructura de la base de datos centralizada en el servidor que es manejada por la aplicación móvil, es espejada en el dispositivo. Es decir, se creará en éste una base de datos local, SQLite con esas mismas tablas.

No obstante, no es obligatorio que se replique todo el conjunto de datos de la base de datos centralizada, sino que podrán enviarse a la base de datos del dispositivo un subconjunto, de acuerdo a alguna condición, que puede tener que ver con usuarios, con el propio dispositivo, etc.

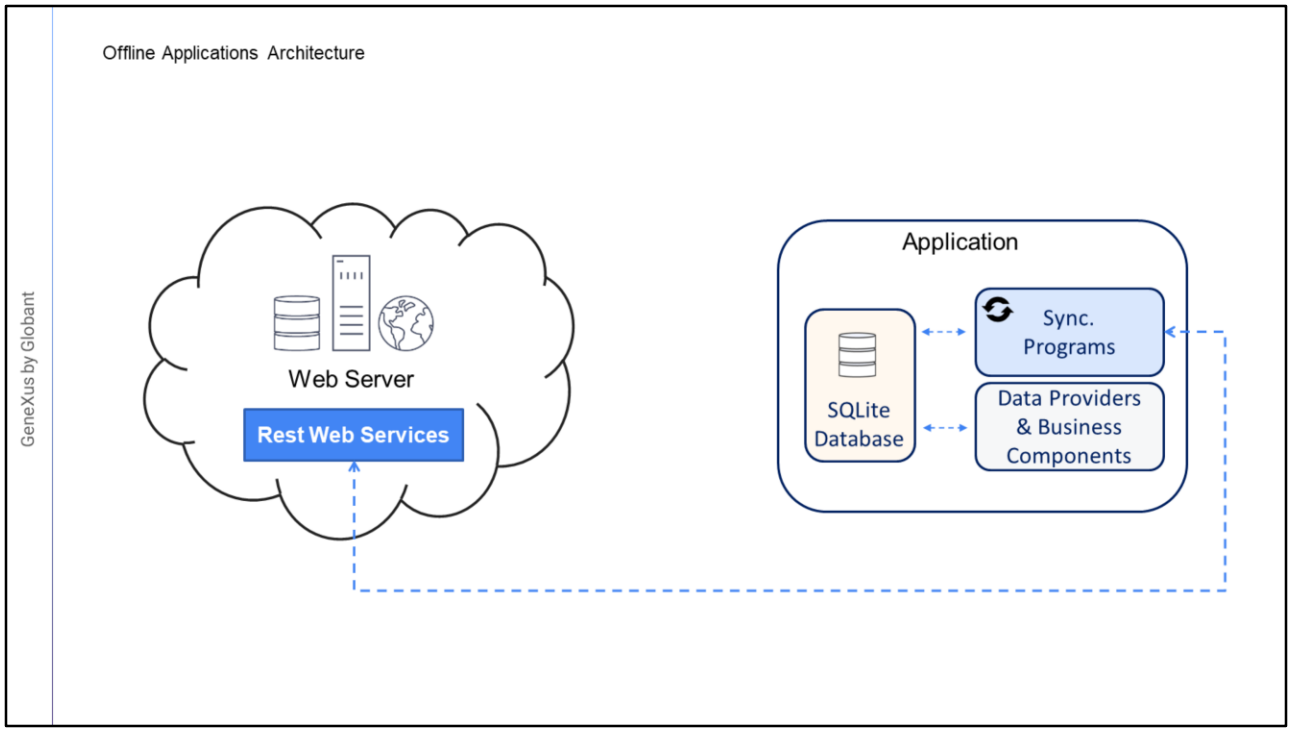
Es decir, hay mecanismos para especificar filtros sobre los datos a enviar a la base de datos local, y tampoco se incluirán todas las tablas, solo las necesarias, todo esto lo veremos mas adelante cuando estudiemos el objeto OfflineDatabase.



En las aplicaciones offline, además de tener la base de datos local, se requieren todos aquellos programas (data providers y business components) que se utilizaban para obtener la información de la base de datos central, los cuales deberán ahora programarse en los lenguajes de las plataformas para dispositivos móviles, de modo tal que accedan a la base de datos local.

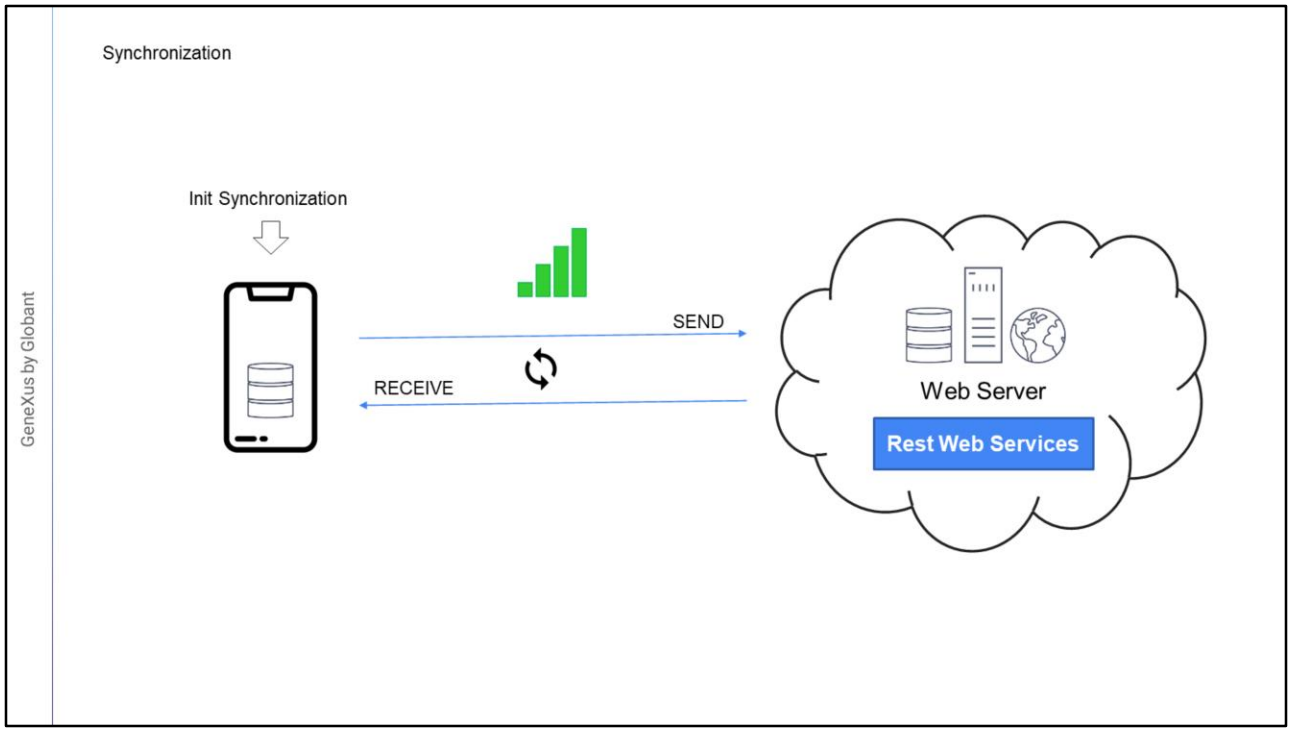
De aquí en más, ya sea que haya conexión, o no la haya, la aplicación siempre trabajará sobre la base de datos local.

La aplicación en el dispositivo no accederá al servidor más que para sincronizar los datos de ambas bases de datos lo que se realizara gracias a programas de sincronización que ejecutaran en el Dispositivo y utilizaran Servicios Rest del lado del Servidor. Estos procesos siempre se iniciaran en el dispositivo, ya sea para realizar el envió o la recepción de datos desde el server.



Toda la capa de servicios que se encontraba en el server web, que contenía los data providers para recuperar los datos y los business components para actualizar los datos de las tablas, estarán ahora en el dispositivo; implementados en el lenguaje de la plataforma, accediendo a la base de datos local, y compilados en el binario.

De esta manera todas las operaciones de CRUD serán siempre sobre la base de datos local y nunca sobre la base de datos de server. El único contacto de la aplicación con el server será para la sincronización.



Siempre la sincronización será iniciada desde el dispositivo, puesto que el servidor no puede saber cuándo el primero obtuvo conexión.

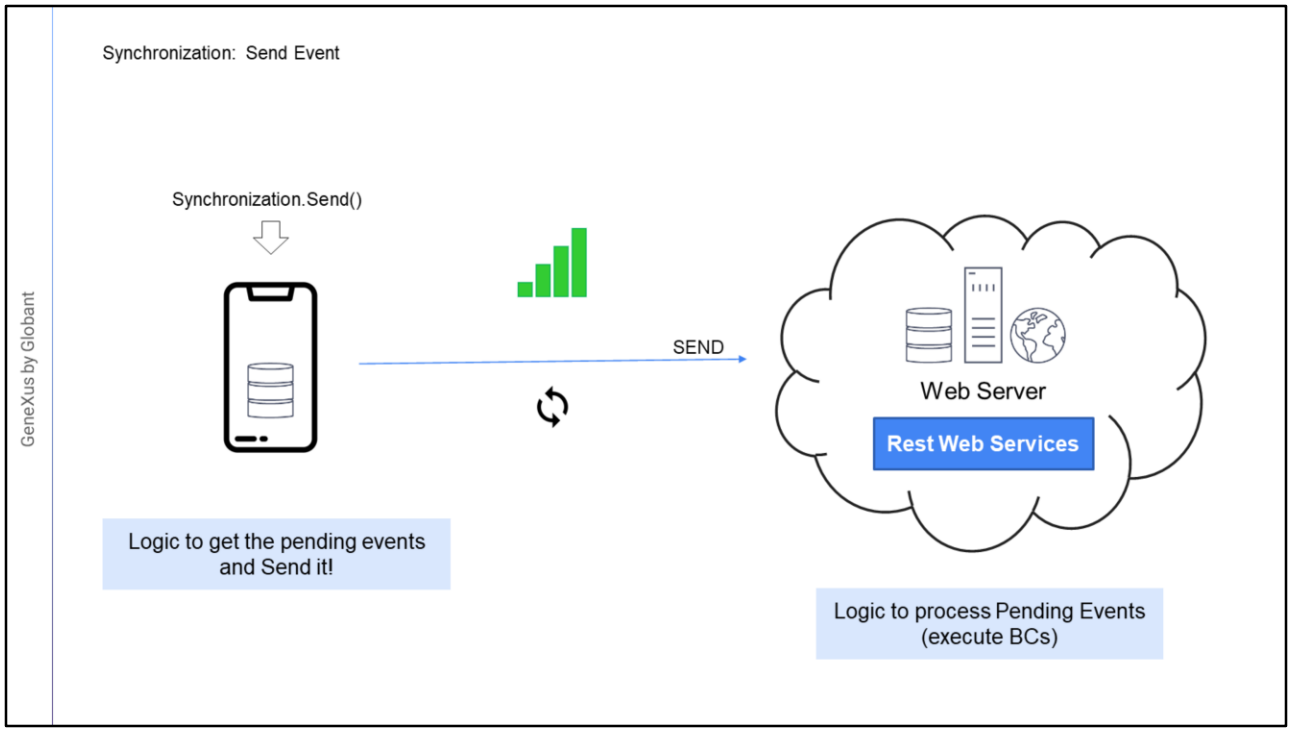
La información almacenada localmente puede sincronizarse con los datos que se encuentran en el servidor si es que así se desea; recordemos que también se puede querer nunca sincronizar o sincronizar a pedido.

El proceso de enviar los datos que cambiaron en el dispositivo hacia el server se denomina: Send

También los datos del servidor que cambiaron se envían al dispositivo para ser actualizados, cada cierto tiempo o a demanda.

El proceso de enviar los datos que cambiaron en el servidor, hacia el dispositivo se denomina: Receive.

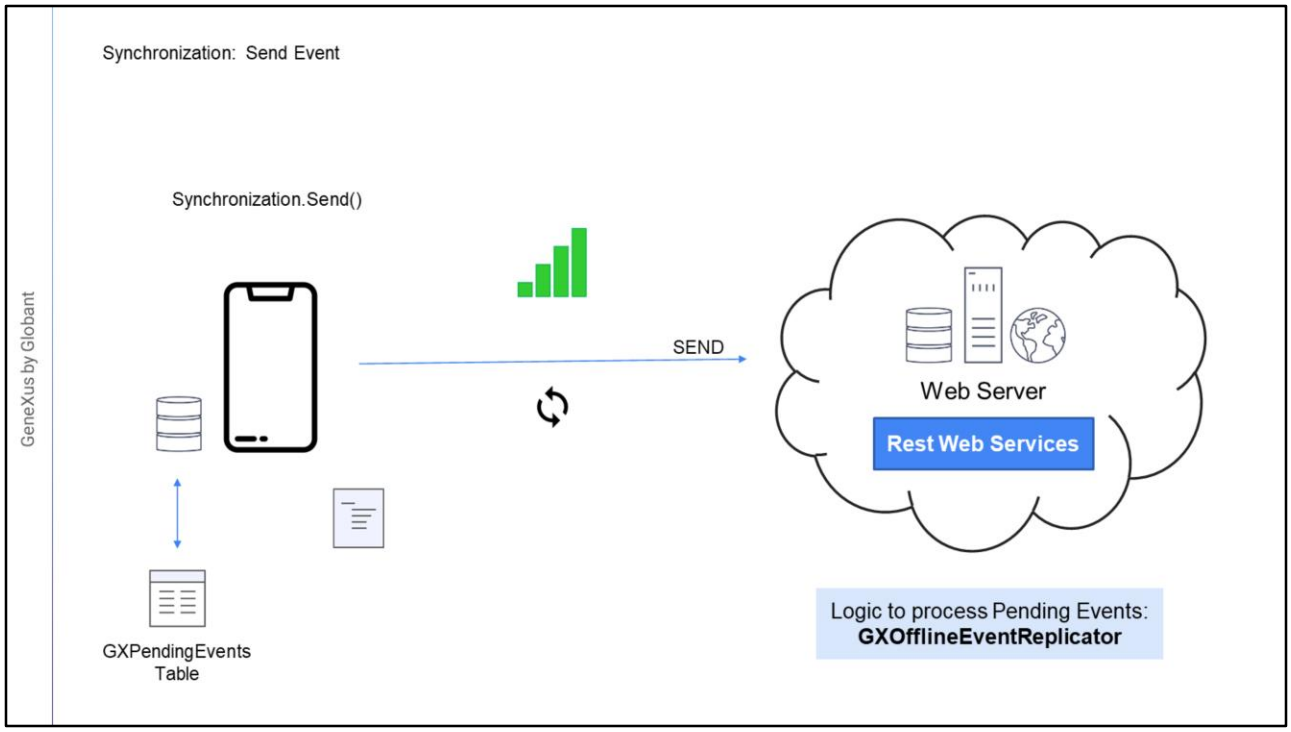
La comunicación entre el dispositivo y el servidor, como ya mencionamos, se realiza mediante la capa de Servicios Rest.



Tanto el Send como el Receive se implementan con lógica del lado del server y lógica del lado del cliente. La idea será que el cliente deba realizar la menor cantidad de procesamiento posible, debido a que su potencia es inferior a la del server.

Cuando el dispositivo inicia el Send (que puede ser iniciado al momento de recuperar la conexión, en forma manual a través del método Synchronization.Send o nunca): debe haber armado una lista ordenada de las operaciones de insert, update y delete que fueron realizadas desde la última sincronización. Es decir, aquellas operaciones que están pendientes.

Esa lista se le envía al proceso del lado del server y este debe recorrer ordenadamente esa lista, y ejecutar la operación correspondiente sobre la base de datos devolviendo al proceso del lado del cliente el resultado.

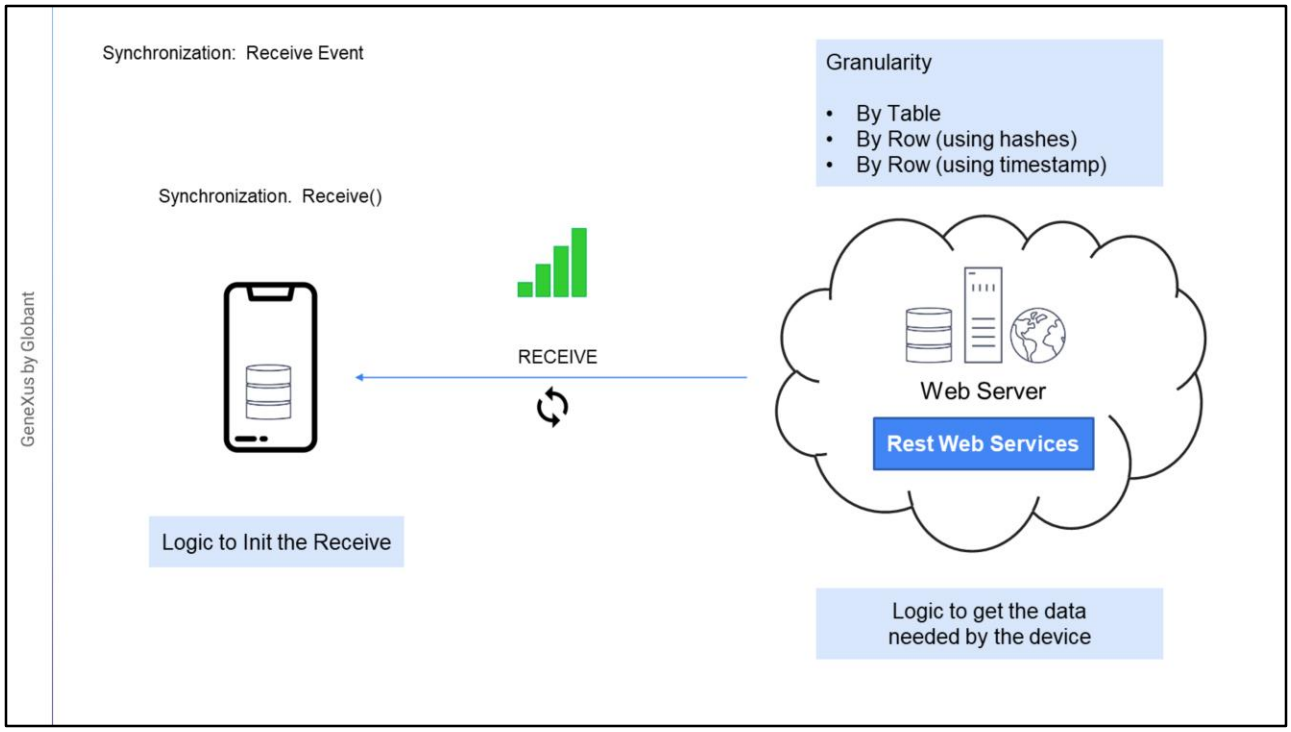


Recordemos que ya sea que tengamos o no conexión, en el cliente siempre se trabajará sobre la base de datos local.

Todas las modificaciones realizadas sobre la base de datos local, son guardadas como "Eventos de sincronización" en una tabla de uso interno llamada GXPendingEvents.

Esta tabla almacena ordenadamente todas las operaciones que se realizaron con Business Components.
Se almacena el nombre del BC donde se realizó la operación, el JSON del BC con los datos del evento, el tipo de operación que se realizó (alta, baja o modificación) y el estado de la misma.
Cada vez que el dispositivo ejecuta un business component, se almacena el evento y queda en estado "pendiente de sincronización".

Cuando se inicia el Send, el cliente traduce la lista de todos los eventos con estado "Pending" en un SDT con formato JSON y lo envía al server. En el server está programado el procedimiento GXOfflineEventReplicator que lee el SDT y realiza las tareas de Inserción, Actualización y Eliminación respetando el orden de las operaciones.

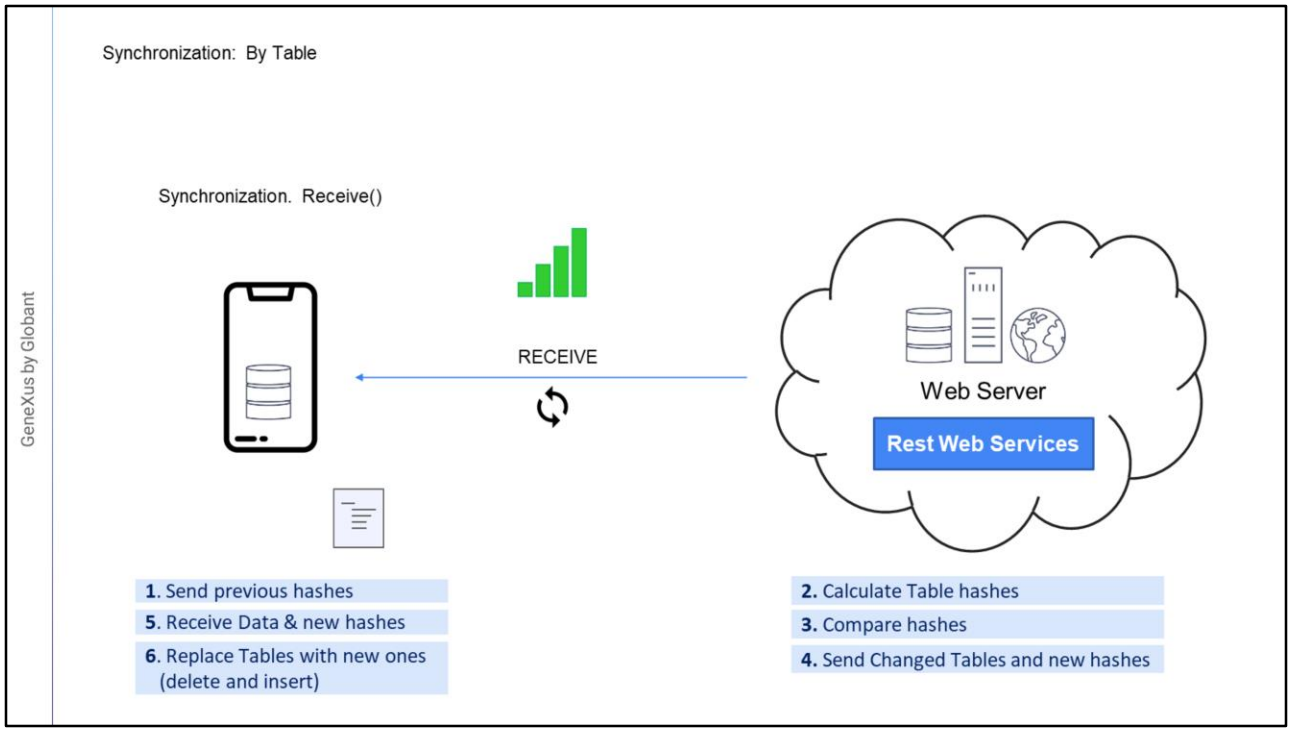


Cuando el dispositivo necesita recibir los datos modificados en el server, inicia el proceso de Receive llamando a un servicio Rest en el server, el dispositivo luego actualiza los datos en la base de datos local.

El comportamiento de la sincronización puede configurarse de acuerdo a varios criterios que determinan cuándo se realizará la sincronización para recibir datos.

Además La sincronización puede hacerse de dos formas: por Tabla o por Fila. Cuando la granularidad es By Table se lleva al dispositivo todas las tablas que fueron modificadas desde la última sincronización. Cuando es By Row, se llevan al dispositivo solamente aquellos registros que cambiaron de cada tabla, desde la última sincronización, y existen dos mecanismos, uno que utiliza hashes y otro que utiliza timestamp.

Veremos a continuación cada uno de estos mecanismos y sus diferencias.



La sincronización “by table” es útil en escenarios donde la cantidad de registros es poca, o cambia con mucha frecuencia, ya que en este último caso es necesario llevar casi todo en cada sincronización.

Tiene la ventaja por sobre la sincronización “by row” de que el procesamiento que requiere del lado del servidor es mucho menor.

Para determinar qué tablas fueron modificadas y por lo tanto deben enviarse al dispositivo, se utiliza un hash, que es el resultado del calculo de un código que “identifica” el juego de datos de cada tabla.

Cuando un cliente pide sincronizar:

Se envía al servidor los hashes de cada tabla, los cuales fueron enviados por el servidor en la sincronización anterior.

Luego para cada tabla, el servidor calcula un nuevo hash con los datos actuales,

Luego compara los hashes con los que tiene actualmente

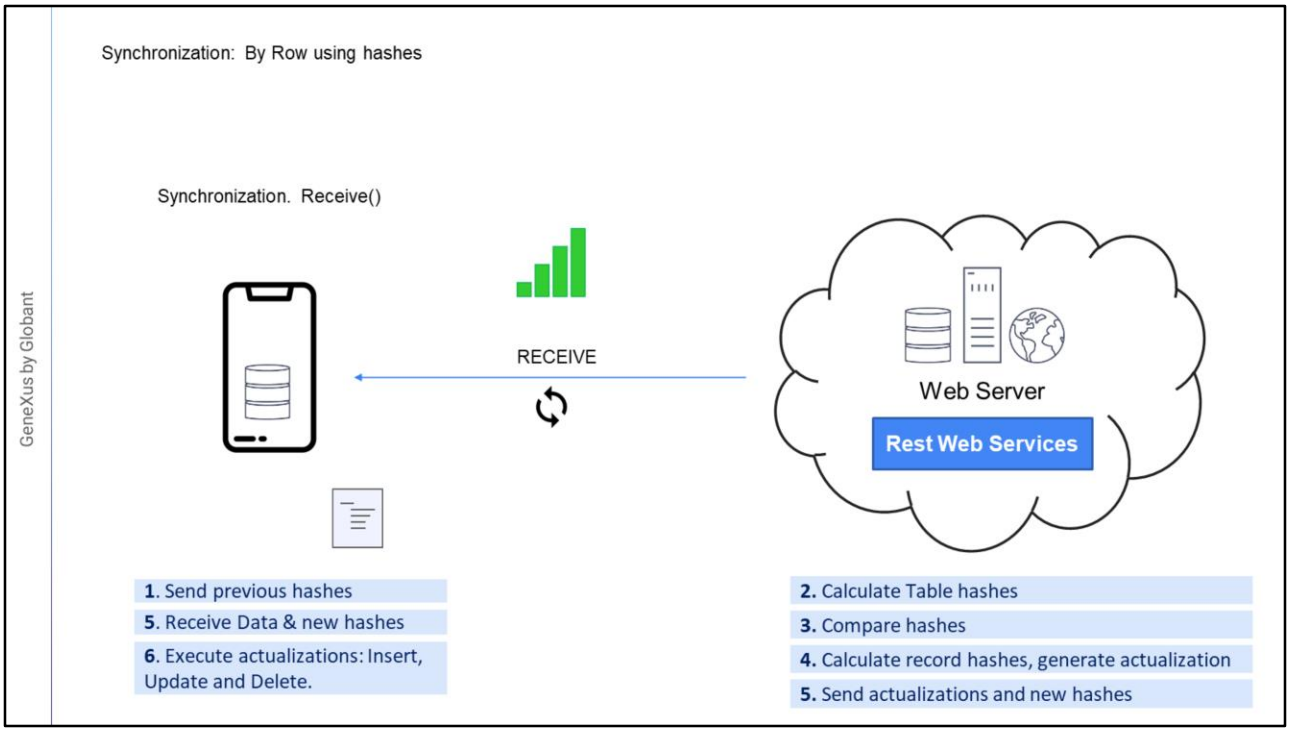
y finalmente envía datos de las tablas, cuando determina que sufrieron modificaciones. Si la tabla no tuvo cambios desde la última sincronización, entonces para esa tabla no se hace nada.

El dispositivo recibe los datos junto con los nuevos hashes.

Por ultimo, el dispositivo reemplaza las tablas que fueron modificadas (borra el contenido y lo vuelve a generar con la nueva información).

Toda esta comunicación se realiza usando servicios Rest.

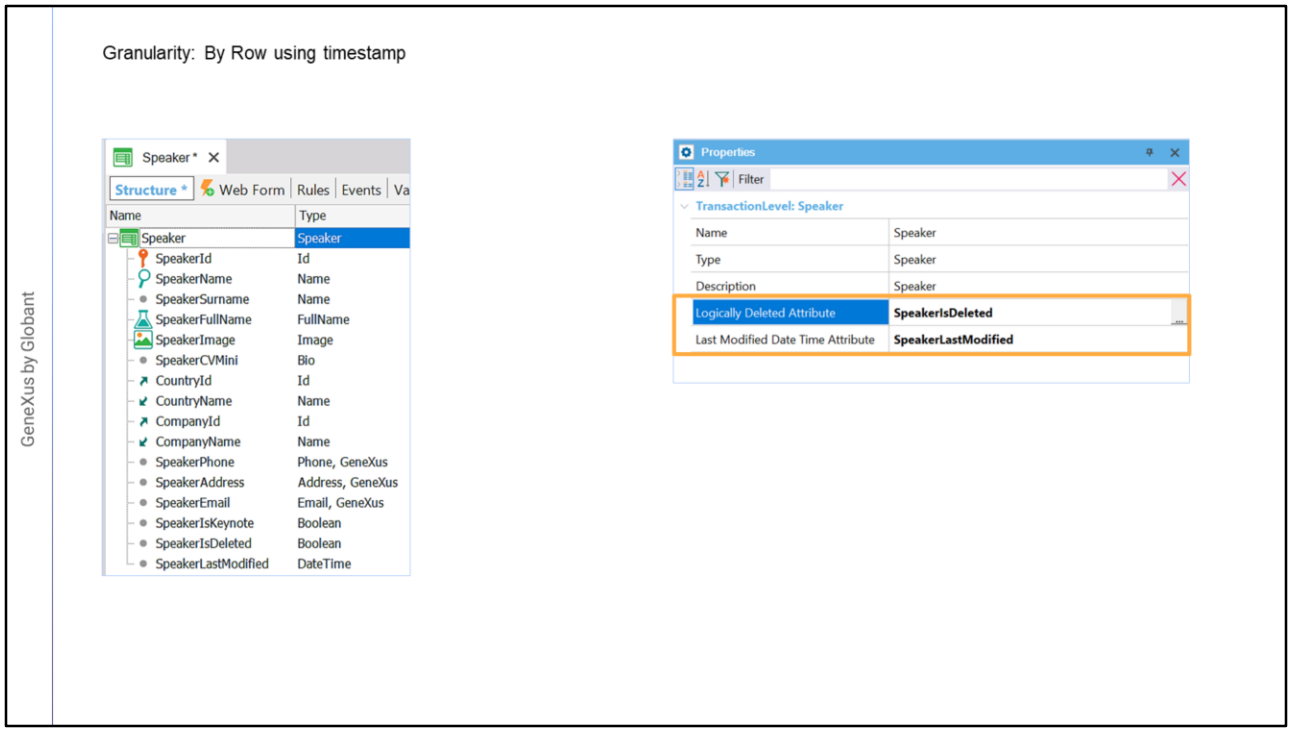
Como caso especial, en la primera sincronización, no hay datos en la base de datos local, por lo que se llevan todos los datos de todas las tablas que cumplan con los filtros en el objeto OfflineDatabase, y los hashes de cada una.



La sincronización “by row”, utilizando hashes, solamente lleva al dispositivo aquellos registros que cambiaron desde la última sincronización utilizando el computo de hashes para determinar las actualizaciones.
La ventaja es que los datos que se transmiten entre el dispositivo y el servidor son menos, ya que solo se trata de los registros modificados, por el otro lado la desventaja de este mecanismo es que requiere de mayor procesamiento del lado del servidor, especialmente con grandes volúmenes de datos.

1. Lo primero que ocurre es que el dispositivo envía al server los hashes de las tablas, al igual que en el caso “By Table”.
2. El server determina que set de datos deben estar en el dispositivo, de acuerdo a los filtros que puedan existir, y calcula un hash para cada set de datos.
3. El server luego compara los nuevos hashes con los actuales y determina cuales deben ser enviados al dispositivo. Si no hay nada que enviar aquí se termina el procesamiento.
4. Para cada set de datos que se debe enviar, el server calcula un hash para cada registro y los compara con los hashes actuales, en base a esto puede resultar que el registro sea el mismo, en ese caso no se enviara, si el hash es distinto significa que ese registro cambio y se deberá enviar como una actualización. También puede ocurrir que ese registro no existe en el set previo, por lo que se enviara como un insert y por ultimo se realiza una comparación para ver que registros existían en los hashes actuales que en los nuevos ya no están, estos serán enviados como eliminaciones. Para cada acción a realizar, insert, update y delete, se prepara una lista.
5. El Server envía las listas con los nuevos datos al dispositivo.
6. El dispositivo luego recibe los datos y guarda el hash de cada tabla.

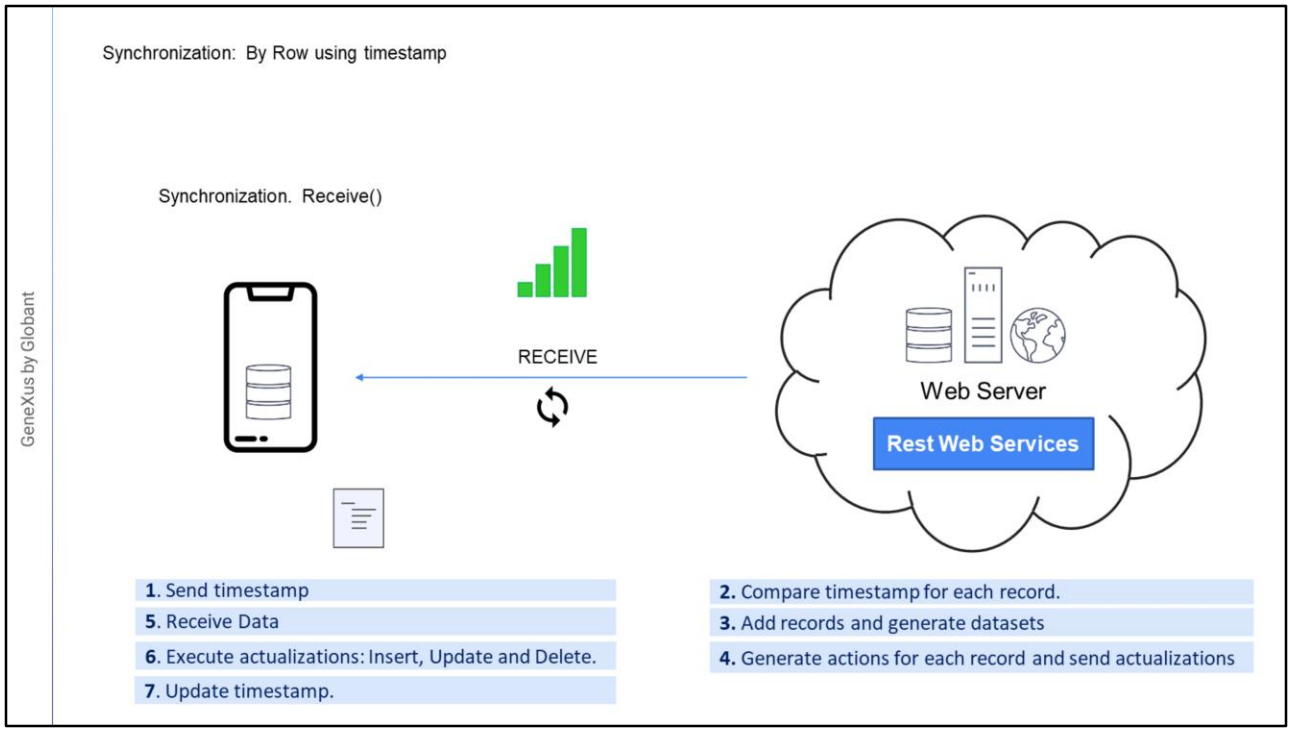
7. Finalmente sus listas se procesan en orden. Para los registros nuevos, se hace un INSERT en la base de datos, y si falla por clave duplicada se hace un UPDATE. Para los registros modificados se hace un UPDATE, y si no existe el registro se hace el INSERT de los mismos. Para los registros eliminados se hace un DELETE.



Veamos el mecanismo By Row utilizando timestamp.

Este mecanismo utiliza un enfoque distinto, la sincronización se sigue realizando por registro pero no se utilizaran hashes para determinar si se trata de una actualización o no, en su lugar se utilizara la fecha de actualización y la marca de eliminación lógica.

Por este motivo para utilizar este mecanismo deberemos indicar para cada nivel de una transacción qué atributo contendrá la eliminación lógica y qué atributo contendrá la fecha y hora de última modificación .

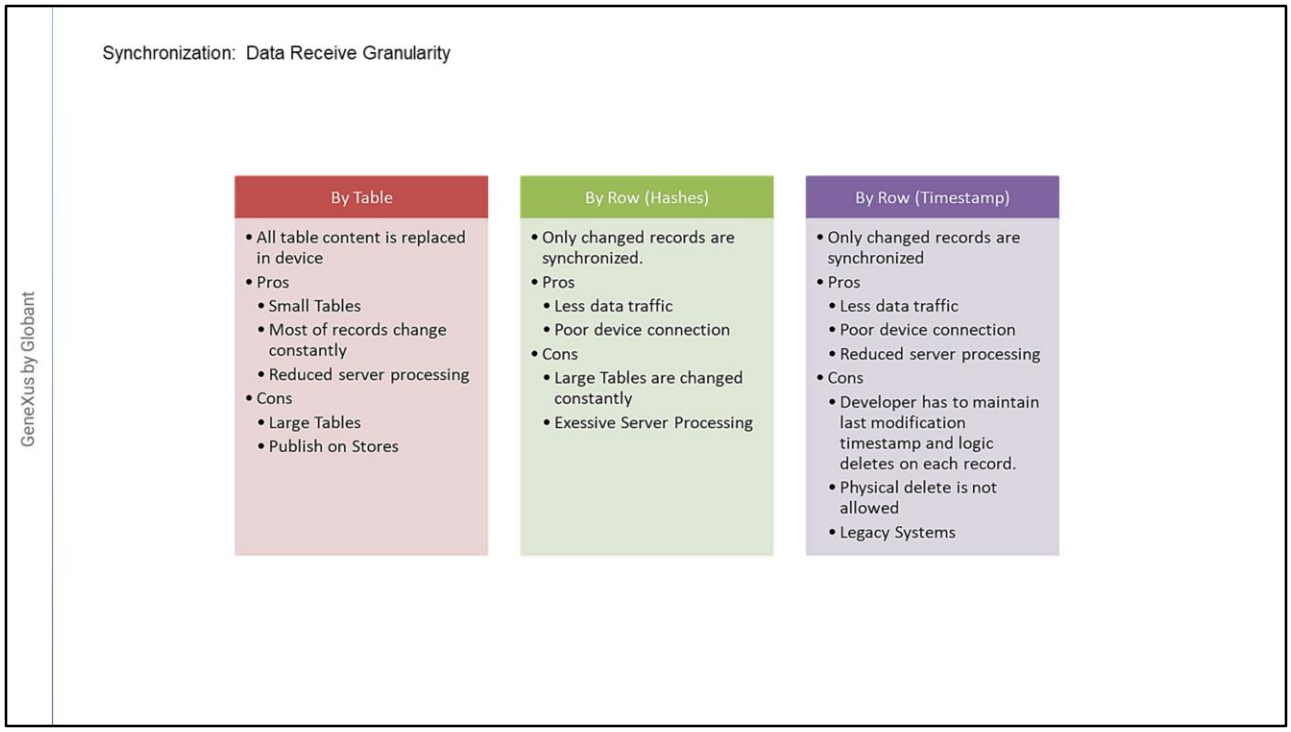


Usando este mecanismo, la primera vez que se debe sincronizar el server enviara todos los datos que cumplan las condiciones, junto con el timestamp de la sincronización, el dispositivo almacenara el timestamp que será utilizado luego en las sucesivas sincronizaciones.

Luego ante cada nueva sincronización.
El dispositivo envía el timestamp de la ultima sincronización.
el server compara el timestamp recibido desde el dispositivo con el de cada registro utilizando el atributo de cada tabla.
Todos los registros que hayan sido agregados o insertados luego del timestamp del dispositivo se agregan a una lista.
Para determinar la acción a tomar sobre cada registro, se evaluara si el registro fue eliminado, utilizando el atributo con la marca de eliminación lógica, en caso afirmativo ese registro se marca como una eliminación que debe ser enviada al dispositivo, el resto de los registros se marca para actualización, en el dispositivo se realizara un "upsert", o sea, se trata de actualizar si existe el registro, de lo contrario se inserta. Toda esta información es enviada al dispositivo.
El dispositivo recibe los datos
Realiza las operaciones y actualiza el timestamp.

Las desventajas de este mecanismo es que el desarrollador es responsable de mantener el timestamp y la marca de eliminación lógica, además no podemos usar eliminación física de registros en esas tablas.

Este mecanismo se puede utilizar en conjunto con el anterior, by row using hashes.



Vamos a hacer un repaso de los mecanismos de sincronización que acabamos de ver.

Respecto de la sincronización By Table, lo que va a suceder es que cuando en el server se determine que una tabla ha sido modificada, esa tabla será enviada al dispositivo y ahí será reemplazada en forma completa. Para determinar las tablas se usara un hash para cada tabla y para cada dispositivo.

¿Cuándo es útil este mecanismo?, por ejemplo cuando nuestras tablas son pequeñas o cuando cambian las mayoría de los registros constantemente, entonces es preferible manejarla de esta forma. Por ejemplo podríamos usarla en un sistema móvil interno donde los vendedores tienen la lista de clientes a visitar y esa lista cambia todos los días, hoy me asignan una lista y mañana es otra completamente distinta.

Cual es la desventaja de esta opción, cuando las tablas son muy grandes, porque el trafico de datos va a ser importante, tampoco seria recomendable en casos donde la aplicación va a ser de uso masivo, o sea que estará publicada en un Store, si cada vez que se va a sincronizar se van a enviar todos los datos la experiencia del usuario no va a ser muy buena.

Bien, para salvar estos casos es que contamos con la sincronización por registros.
Acá tenemos dos opciones, hacerlo usando hashes o por timestamp.

Veamos la primer opción.

En este caso el server va a determinar de acuerdo a los hashes qué va a calcular para cada set de datos y para cada registro, cuales debe enviar a cada dispositivo. Entonces solo se enviaran las novedades, solo los registros que hayan sido modificados.

La ventaja es que el trafico de los datos se reduce considerablemente, a menos que constantemente se modifique un gran volumen de datos.
Volviendo al ejemplo anterior, en vez de recibir toda la tabla de clientes, vamos a recibir solo aquellos que sufrieron alguna modificación.
Supongamos que solo recibimos los clientes activos, entonces el dispositivo tiene un hash, el de la ultima sincronización, y cuando quiere sincronizarse envía ese hash al server, quien vuelve a calcular un hash para los clientes activos y lo compara con el que le envió el dispositivo, si son distintos, entonces el server va a generar un hash para cada registro de esa consulta y va a compararlos con los que tenia anteriormente, de esa forma puede determinar exactamente que se agregó, modificó o eliminó.
Esta es la opción por defecto cuando se crea una aplicación Offline.

Este método también es útil cuando la conectividad no es muy buena, ya que tenemos menos trafico.
Como desventaja vamos a tener que se requiere de mucho mas procesamiento del lado del server y no seria recomendable en el caso de que contemos con grandes volúmenes de datos que son modificados en forma constante.

Supongamos este caso, tenemos en nuestro sistema una tabla con miles de productos que queremos que estén en cada dispositivo, tenemos además cientos de usuarios. Sincronizar por Tabla no seria adecuado dado que es una tabla con muchos datos, y sincronizar por registro usando hashes puede que tampoco sea la solución, puesto que para poder determinar qué registros debe enviar a cada dispositivo, se debe calcular y comparar el hash de cada uno de los miles de productos que tenemos. Esto sumado a la concurrencia de muchos usuarios en simultaneo puede generar un problema de procesamiento del lado del server.
Entonces ninguna opción es optima, veamos entonces la tercer opción, que es usar timestamp.

Lo que vamos a tener usando timestamp es mantener la transferencia de datos al mínimo, solo para los registros modificados, pero a su vez involucrando menos procesamiento del lado del server, puesto que ya no tiene que calcular los hashes para toda la consulta, sino que puede determinar cuales fueron por la fecha y hora de ultima modificación y por la marca de eliminación lógica.
Acá necesitamos mantener estos dos atributos adicionales, la marca de eliminación lógica y la fecha y hora de ultima modificación, lo bueno es que muchos sistemas ya implementan estos datos en cada registro.
La desventaja de este mecanismo es que es una responsabilidad del desarrollador mantener justamente estos dos atributos, algo que puede ser complejo cuando hay muchos sistemas distintos involucrados.

También podemos tener un mix entre los últimos dos mecanismos, si usamos By

Row y en una transacción no configuramos los atributos de timestamp y eliminación lógica, se usaran hashes.


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