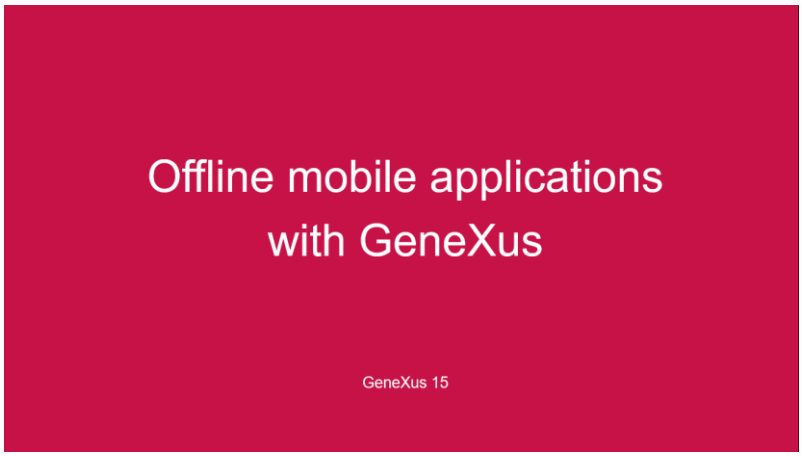
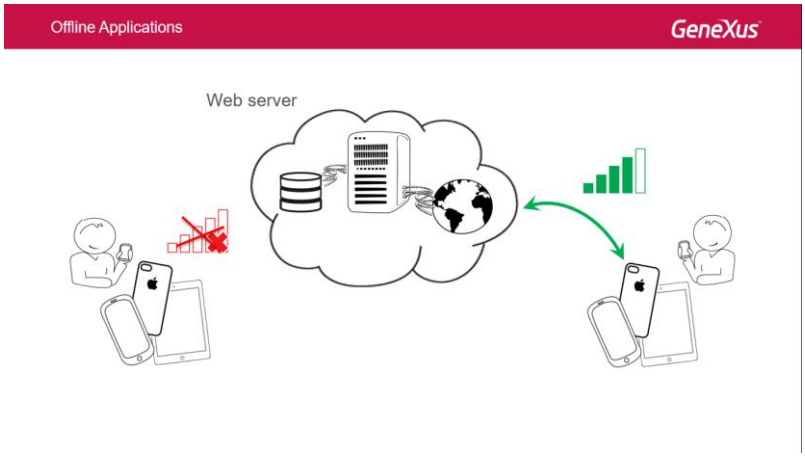
Offline Mobile Applications with GeneXus (Part I)



Hasta ahora hemos asumido que la aplicación móvil debía estar conectada siempre al servidor web para poder funcionar, accediendo a los servicios REST y mediante éstos a la base de datos que está en el server.

Sin embargo, GeneXus nos permite crear aplicaciones móviles que puedan trabajar en forma parcialmente conectada o incluso totalmente desconectada del servidor web.

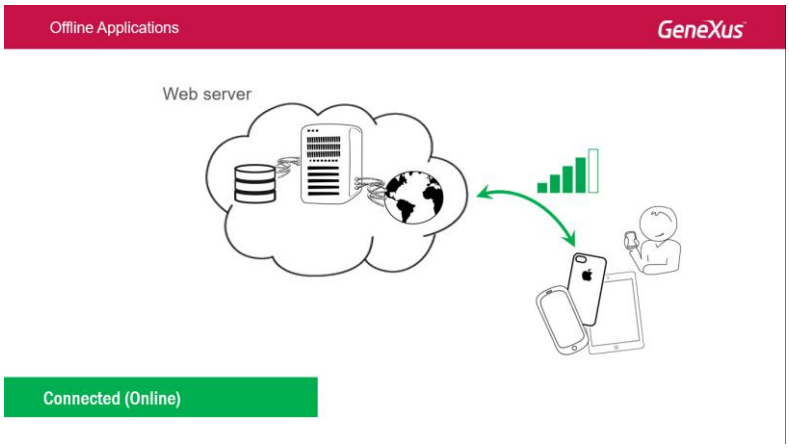


A continuación, estudiaremos cada caso.

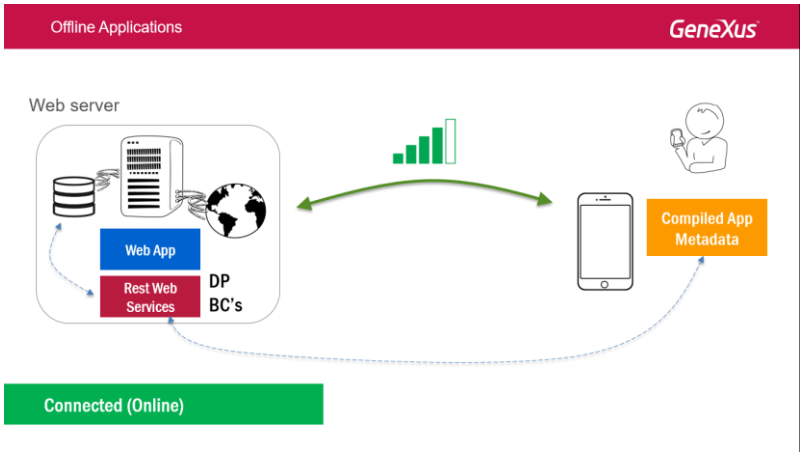
Repasemos los conceptos de las aplicaciones conectadas.

Llamamos aplicaciones conectadas (Online) a aquellas que requieren estar siempre conectadas a Internet para poder recuperar los datos y trabajar con ellos.

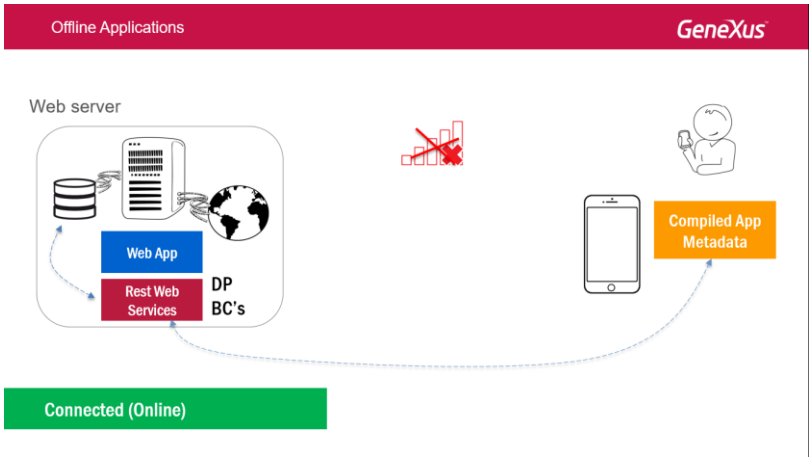
De estar el dispositivo desconectado, sólo podrá trabajar con los datos cacheados, pero no podrá navegar nuevas pantallas, ni podrá actualizar la información.



La aplicación, una vez compilada e instalada en el dispositivo, necesita acceder a la capa de servicios del servidor web para poder ejecutar los data providers que devuelven los datos, y los business components que realizan las operaciones de CRUD sobre la base de datos.



Sin conexión no se pueden obtener los datos, pues no hay una base de datos local, sino que todos los datos están en el servidor.



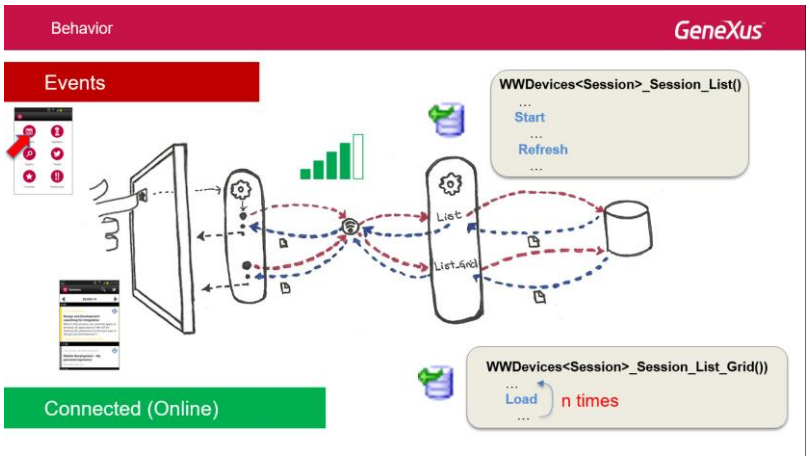
¿Qué pasa si hacemos tap para ver datos de la base de datos en una aplicación online?

Al hacer tap en el dispositivo por ejemplo en icono de las Sesiones, en el Menu, se realiza una invocación al list del WorkWith de Sesiones. El código correspondiente se ejecuta en el dispositivo.

En su lógica interna, este Work With llama al servidor para que éste le devuelva los datos a cargar en la parte fija del WorkWith a través de un servicio Rest, y luego le pide al servidor que ejecute otro servicio Rest, que devuelve los datos del grid.

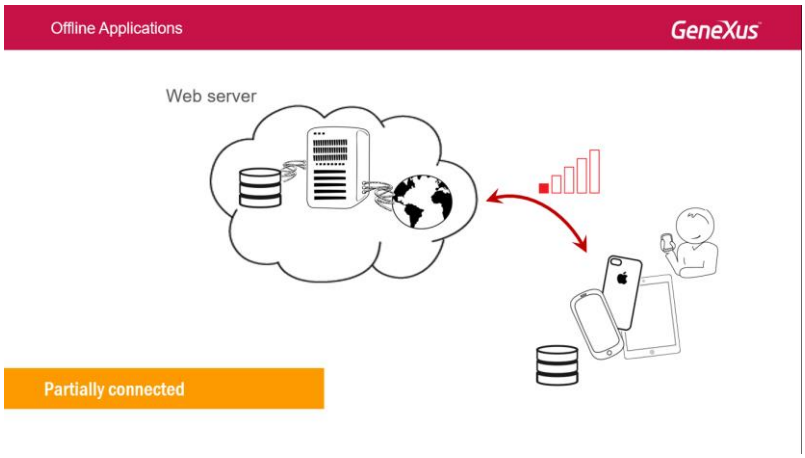
Con los datos devueltos en dos responses, se arma la pantalla (por un lado la parte fija, y por otro lado el grid).

En esta arquitectura de una aplicación online, los servicios Rest, los data providers que acceden a los datos y la base de datos están en el servidor, por lo que solamente pueden accederse a los mismos si hay conexión.



Veamos ahora el caso de aplicaciones parcialmente conectadas.

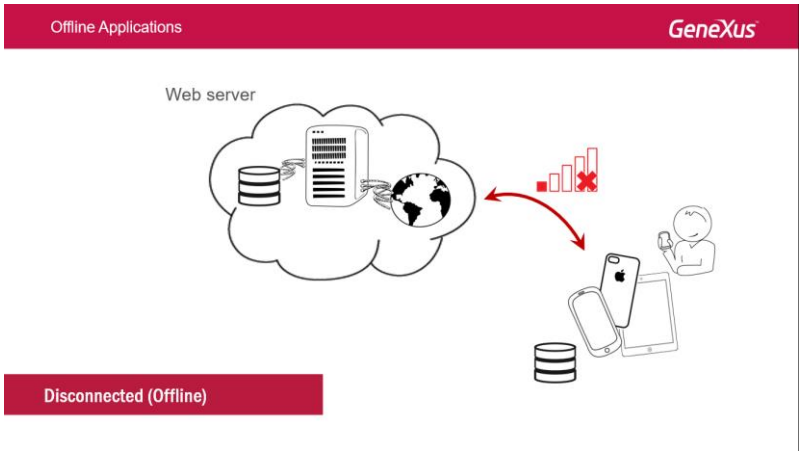
Puede requerirse que haya parte de la aplicación que se siga ejecutando cuando se encuentra desconectada de internet, mientras que otra parte de la misma necesariamente debe tener conexión para poder funcionar.



O inclusive como caso particular puede que no nos interese en absoluto que se produzca la sincronización, sería el caso de una aplicación en la que deseamos independizarnos absolutamente de los datos en el server. Todo se manejaría en el dispositivo, ya que la aplicación en el dispositivo perderá todo contacto con el servidor.

Y otro caso particular es cuando queremos que el dispositivo pueda recibir los cambios efectuados en la base de datos centralizada, pero nunca enviar sus propios cambios, que quedarán en su base de datos local.

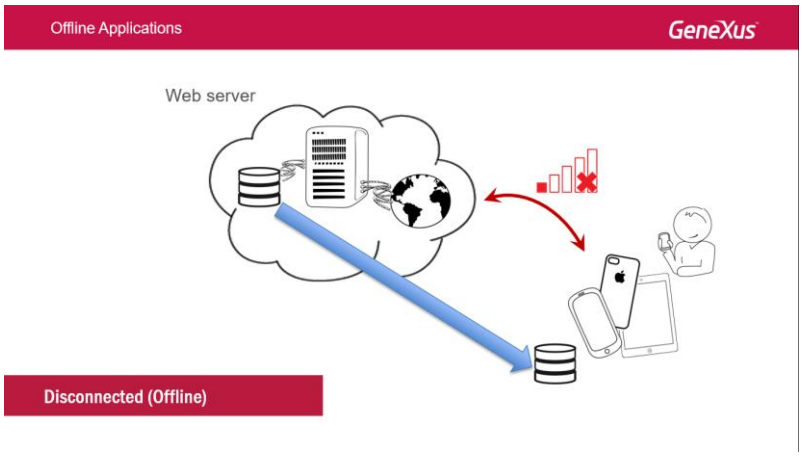
Estas aplicaciones que pueden trabajar sin estar conectadas, las llamamos aplicaciones Offline.



Empecemos 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. Al final abordaremos el caso mixto, en el que una parte de los datos deben seguir siendo manejados exclusivamente en forma centralizada.

En este caso, toda 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 datos; es decir, no todos los datos que están en el server es obligatorio que se envíen a la base de datos del dispositivo. Yo puedo definir ciertos filtros, y hacer que solamente algunos de los datos o de las tablas que están en la base de datos del server pasen a estar almacenadas efectivamente en el dispositivo.

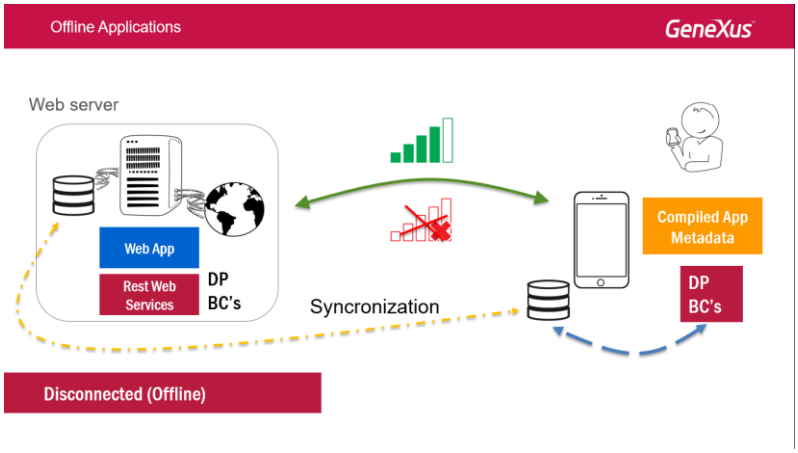


En las aplicaciones offline, además de tener la base de datos local, se requieren todos aquellos programas (data providers y business components) que se utilizaban para obtener la

información de la base de datos central, los cuales deberán ahora programarse en los lenguajes de las plataformas de SD, de modo tal que accedan a la base de datos local.

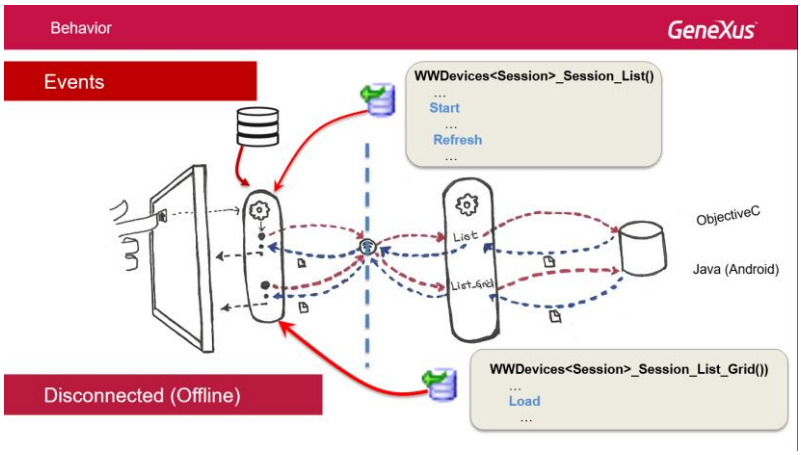
De aquí en más, ya sea que haya conexión, o no la haya, la aplicación siempre trabajará sobre la base de datos local.

La aplicación en el dispositivo no accederá al servidor más que para sincronizar los datos de ambas bases de datos.

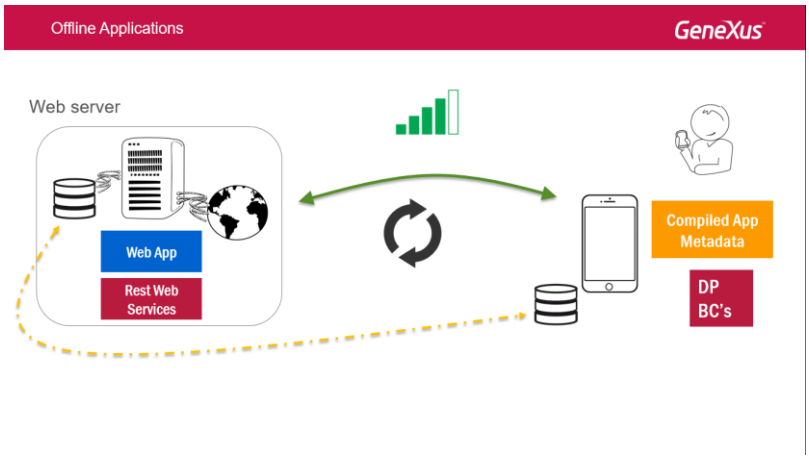


Toda la capa de servicios que se encontraba en el server web, que contenía los data providers para recuperar los datos y los business components para actualizar los datos de las tablas, estarán ahora en el dispositivo; implementados en el lenguaje de la plataforma, accediendo a la base de datos local,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



Cuando se recupera la conectividad hay que sincronizar las bases de datos, es decir enviar y recibir los cambios desde y hacia el dispositivo.



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

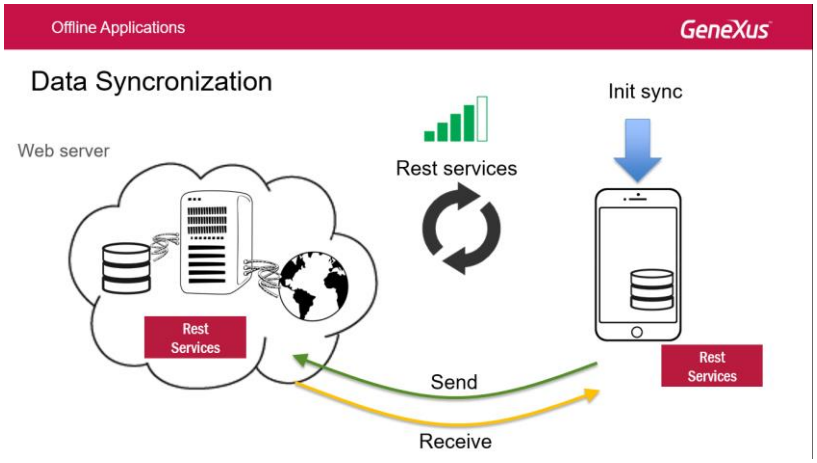
La información almacenada localmente puede sincronizarse con los datos que se encuentran en el servidor (si es que así se desea; recordemos que también se puede querer nunca sincronizar o sincronizar a pedido).

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**.

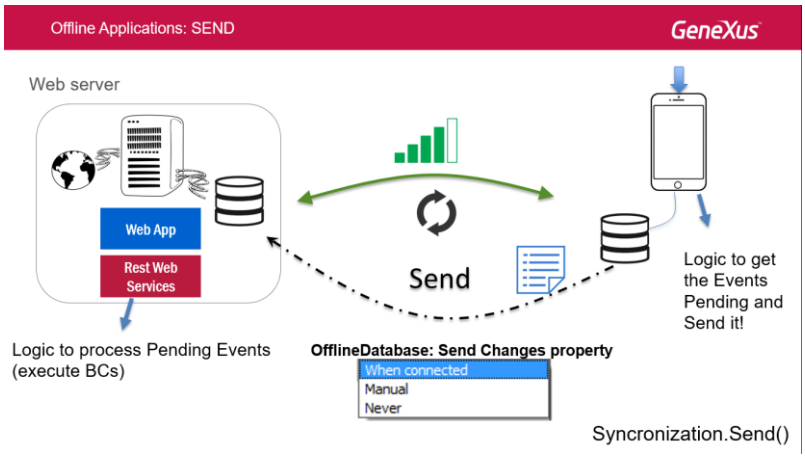
Tanto el dispositivo como el servidor se comunican mediante REST services para la sincronización.



Tanto el Send como el Receive se implementan con lógica del lado del server y con 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 muy inferior que la que podemos tener en el server.

Cuando el dispositivo inicia el **Send** (que puede ser iniciado al momento de recuperar la conexión, en forma manual, a través del método Syncronization.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.

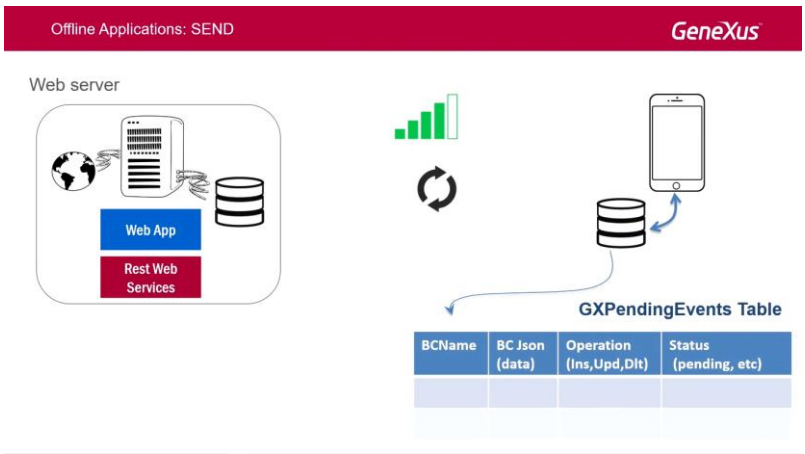
Esa lista se le envía al proceso del lado del server. Este último debe recorrer ordenadamente esa lista, y ejecutar la operación correspondiente sobre la base de datos… devolviendo al proceso del lado del cliente el resultado.



Recordemos que ya sea que haya conexión, o que no la haya, en el dispositivo siempre se trabajará sobre la base de datos local. Solamente se accederá al servidor para la sincronización.

Todas las modificaciones realizadas sobre la base de datos local, son guardadas como “Eventos de sincronización” en una tabla llamada GXPendingEvents

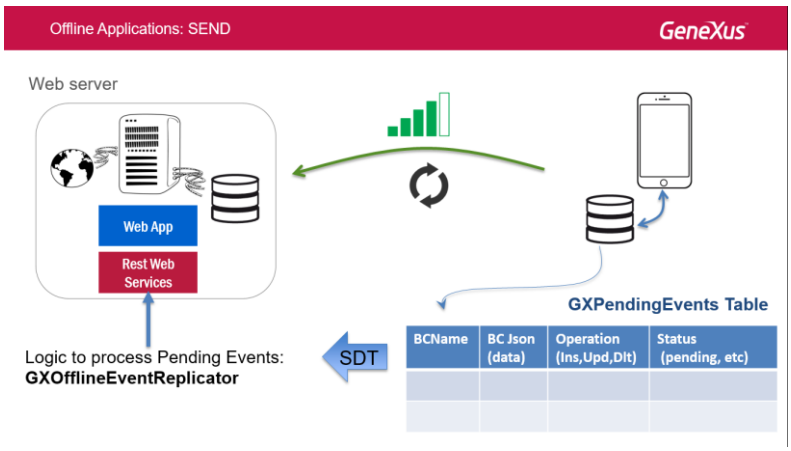
Esta tabla almacena ordenadamente todas las operaciones que se realizaron con Business Components. Se almacena el nombre del BC donde se realizó la operación, el JSON del BC con los datos del evento, el tipo de operación que se realizó (alta, baja, o modificación) y el estado de la misma.



Cada vez que el dispositivo ejecuta un business component, se almacena el evento y queda en estado “pendiente de sincronización”.

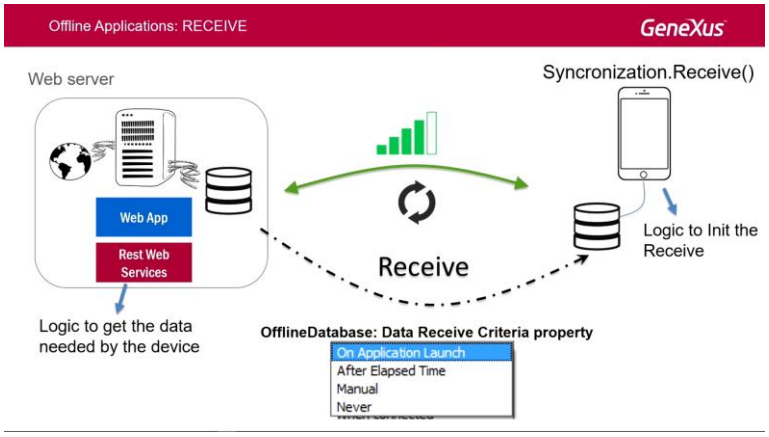
Cuando se inicia el Send, el cliente traduce la lista de todos los eventos con estado “Pending” en un SDT y lo envía al server. En el server está programado el procedimiento GXOfflineEventReplicator que lee el SDT y realiza las tareas de CRUD respetando el orden de las operaciones que vienen en el JSON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

El procedimiento GXOfflineEventReplicator puede llamar a otros procedimientos auxiliares para tratar los errores de sincronización.



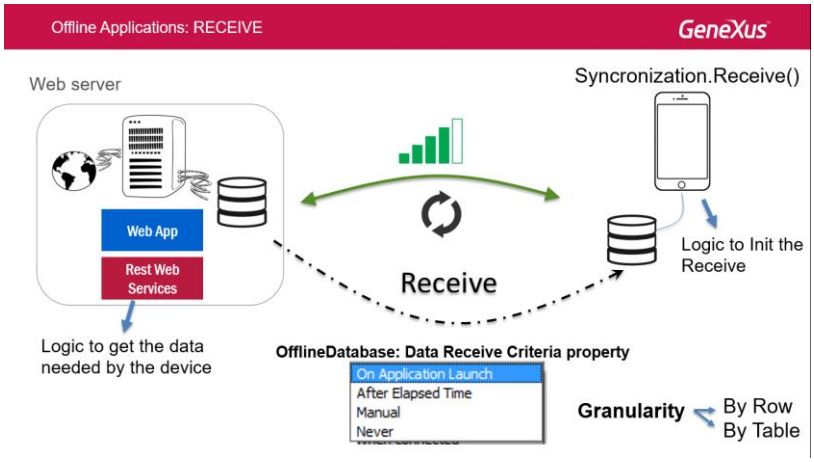
Cuando el dispositivo necesita recibir los datos modificados en el server, inicia el proceso de Receive.

El comportamiento de la sincronización se configura mediante las propiedades del objeto OfflineDatabase, en particular la propiedad **DataReceiveCriteria** que determina cuándo se realizará la sincronización para recibir datos.

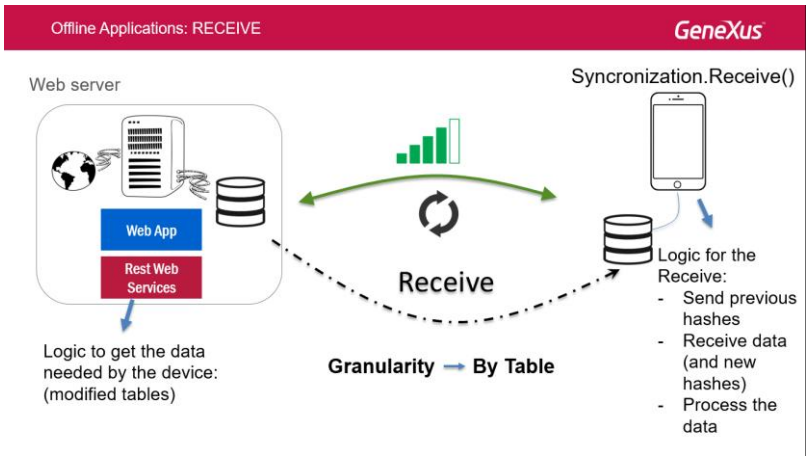


El resultado de la sincronización, (es decir los datos que quedan en el dispositivo) es independiente de la configuración que elija, ya que la misma determina **cómo** se llevan los datos, no **qué** datos se llevan. Los mismos se determinan según los filtros definidos en el objeto OfflineDatabase.

La sincronización que permite al dispositivo recibir datos del server puede hacerse con una granularidad: por Tabla o por Fila. Cuando la granularidad es por Tabla (**By Table**) se lleva al dispositivo todas las tablas que fueron modificadas desde la última sincronización. Cuando es por Fila (**By Row**), se llevan al dispositivo solamente aquellos registros que cambiaron de cada tabla, desde la última sincronización.



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.

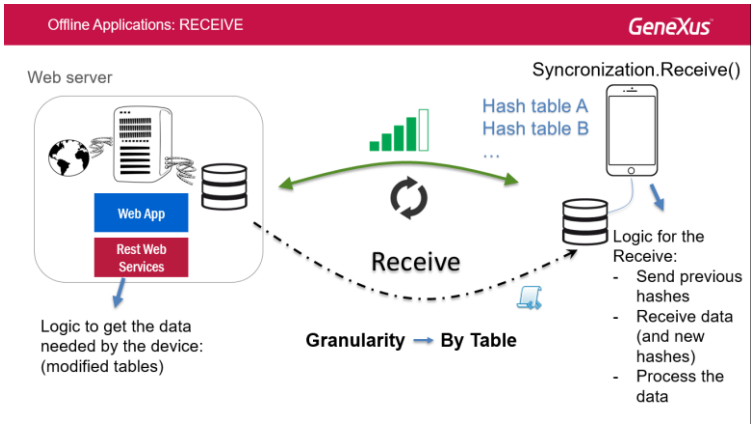


Para determinar qué tablas fueron modificadas y por lo tanto deben enviarse al dispositivo, se utiliza un hash que “identifica” el juego de datos de cada tabla.

Cuando un cliente pide sincronizar, envía al servidor los hashes de cada tabla, los cuales fueron enviados por el servidor en la sincronización anterior. Para cada tabla, el servidor calcula un nuevo hash con los datos actuales (luego de aplicar los filtros que aplican a ese dispositivo), y solo envía datos de la tabla, cuando el nuevo hash es diferente del anterior, es decir, solamente se envían datos de tablas que sufrieron modificaciones. Si la tabla no tuvo cambios desde la última sincronización, entonces para esa tabla no se hace nada. Los mensajes se envían en formato JSON.

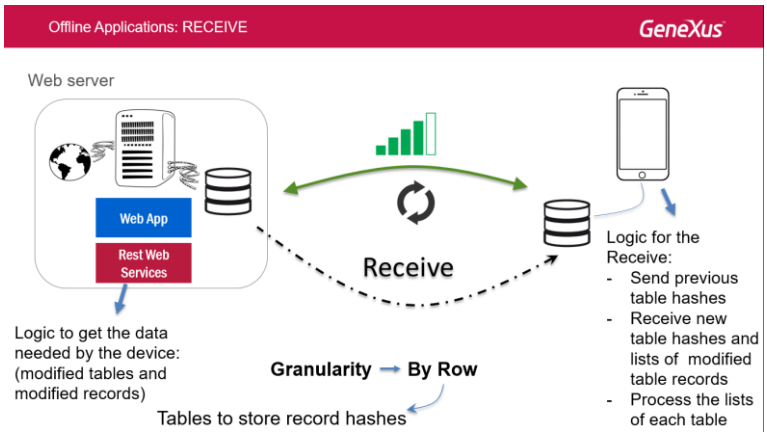
En la primera sincronización, no hay datos en la base de datos local, por lo que se llevan todos los datos de todas las tablas. (Observemos que esto no significa enviar todos los datos que están en el server, porque los datos pueden definirse filtros en el objeto OfflineDatabase, como dijimos antes). Además de guardarse los datos en la tablas, se guardan los hashes de cada tabla.

En las siguientes sincronizaciones, el dispositivo envía los hashes recibidos y el server envía solamente los datos de las tablas que cambiaron. Cuando se reciben, se borra todo el contenido de cada tabla y se reemplaza por el contenido recibido.



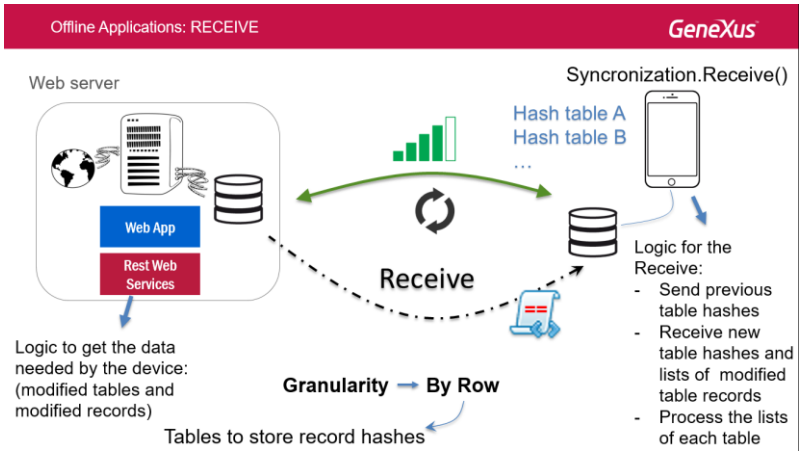
La sincronización “by row” solamente lleva al dispositivo aquellos registros que cambiaron desde la última sincronización, por lo que primero se determina si la tabla cambió o no.

La forma de determinar cuáles tablas fueron modificadas, es exactamente la misma que se utiliza en la sincronización “By Table”.



Una vez que se sabe que una tabla fue modificada, para saber cuáles fueron los registros modificados, se calcula el hash de cada registro y se comparan con el hash correspondiente almacenado previamente. Luego se envía al dispositivo, si la tabla fue modificada su nuevo hash (igual que en la sincronización “By Table”) y tres listas: una para los registros nuevos, otra para los modificados y otra para los eliminados.

Al llegar el JSON, se guarda el hash de cada tabla y 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. Y para los registros eliminados se hace un DELETE.



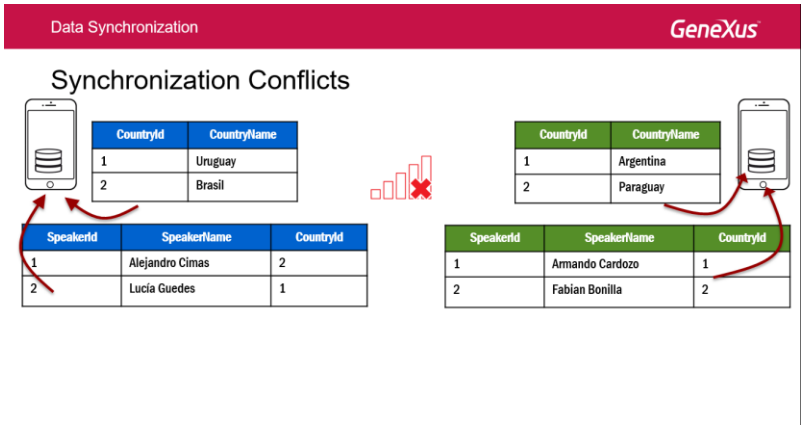
Un problema que la sincronización resuelve de manera automática, es el conflicto por numeración de claves autonumeradas. Por autonumeradas se entiende tanto las claves que son Autonumber, como las claves que se numeran mediante el uso de un Procedure o alguna regla en la transacción.

Tomemos por ejemplo el caso del ejemplo de EventDay, que tenemos la transacción de Country y la transacción de Speaker, en las que ambas claves primarias son autonumeradas. Además, el atributo CountryId es clave foránea de la transacción Speaker.

Supongamos que la aplicación está instalada en 2 dispositivos diferentes, y se está trabajando en forma offline, sin conexión.

Desde el device 1 se insertan los países Uruguay con el Id=1 y Brasil con el Id=2 y desde el device 2 se insertan los países Argentina con el Id=1 y Paraguay con el Id=2.

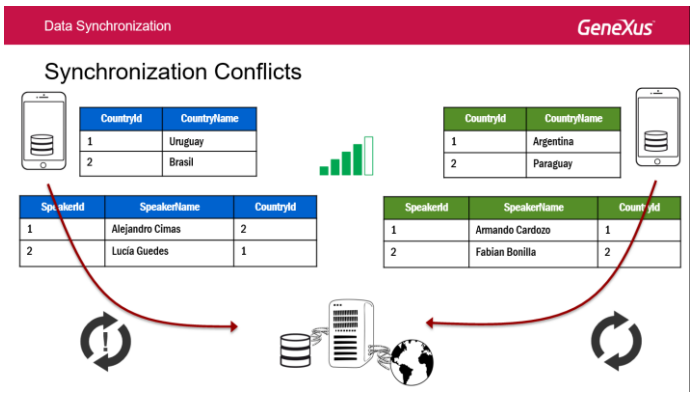
Todos los datos quedan guardados en la base de datos local de cada dispositivo.



Una vez que se obtiene la conexión, uno de los dispositivos se sincroniza con el server, enviando toda las operaciones que realizó a través de un Business Component sobre la base de datos local.

Cuando el segundo dispositivo intenta sincronizar, se produce un conflicto porque se repite la clave del país que está almacenado en el servidor.

Como se usan claves autonumeradas, GeneXus resuelve el conflicto en el mismo servidor y luego se devuelve esta información al dispositivo para que haga localmente este mismo cambio y así quedar consistente con los datos del servidor.

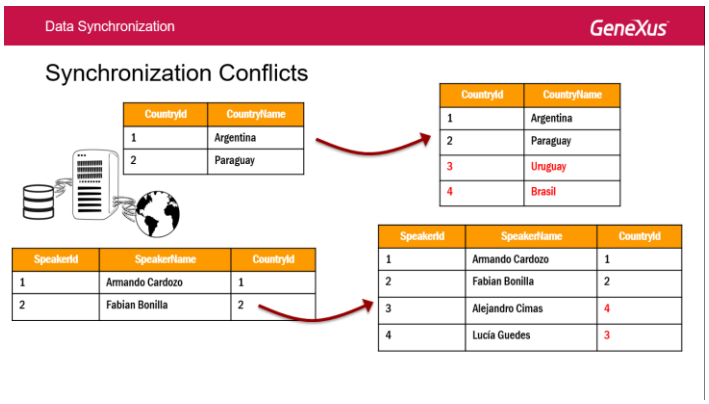


Veamos cómo se hace esto.

Generando una nueva clave para los países que se repiten, y actualizando los datos de la tabla del server con ese Id generado nuevamente. Luego actualiza la tabla de oradores para que las claves foráneas se correspondan.

Por último, actualiza los datos en el dispositivo. Para poder aplicar los cambios de clave en el mismo, el servidor incluye en la respuesta que envía, la correspondencia entre los valores enviados por el dispositivo y los valores con los que quedaron los datos en la base de datos del servidor. Con esta información, el dispositivo actualiza las claves en las tablas locales, de forma que queden coherentes con el servidor.

Los únicos conflictos de sincronización que se resuelven automáticamente son los de claves autonumeradas. En cualquier otro conflicto se sustituye los datos del dispositivo con los datos del servidor.



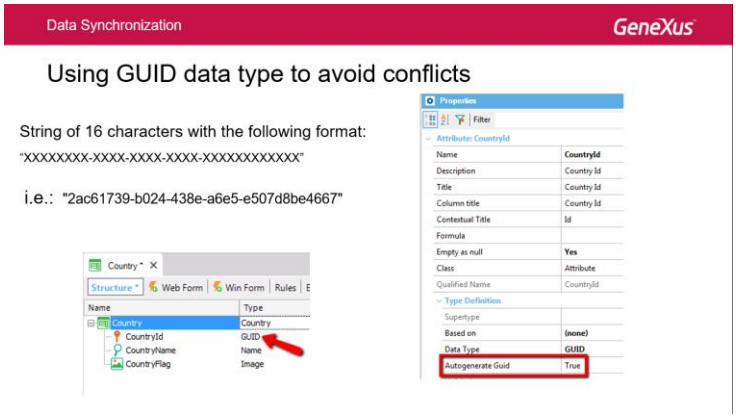
Una forma de minimizar los casos por usar la misma clave, es utilizar un identificador único.

Un GUID (Global Unique Identifier), es un identificador generado en forma pseudo aleatoria. Si bien no se puede garantizar que sea único, la probabilidad de que un número sea generado dos veces es infinitesimalmente baja.

En GeneXus disponemos del tipo de datos GUID para asignar a un identificador. Mediante su propiedad Autogenerate GUID, podemos hacer que se genere automáticamente.

También disponemos de métodos y funciones para operar con este tipo de datos.

Es recomendable utilizarlo en entornos de ejecución distribuida, como es el caso de aplicaciones para Smart devices donde cada usuario puede ingresar datos desde su dispositivo.



En el caso en el que tanto el Receive como el Send se hagan en forma manual, se debe usar la Synchronization API para realizarlos.

Esta API no se encuentra dentro del folder SmartDevicesAPI sino que es parte de la gramática.

Cuenta con los métodos Send y Receive para la sincronización, ServerStatus para determinar el estado del server, y ResetOfflineDatabase que retorna la base de datos local a su estado inicial, ya sea haciendo un Create Database para vaciar las tablas o cargando una base datos pre-cargada.

En el caso de la sincronización manual, por ejemplo, se puede crear un Panel for Smart Devices, donde se programa el send y el receive, como vemos en la imagen.