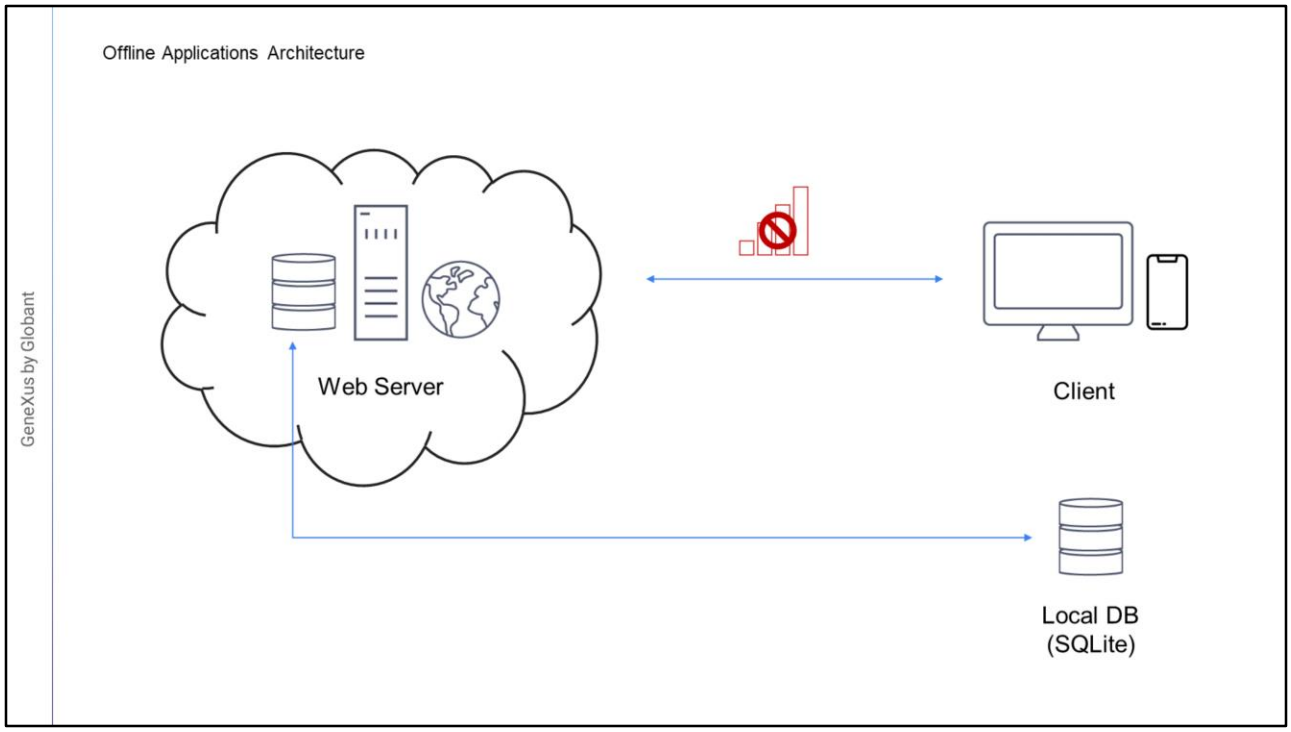
GeneXus by Globant

Offline Applications Architecture

Diego Marranghello



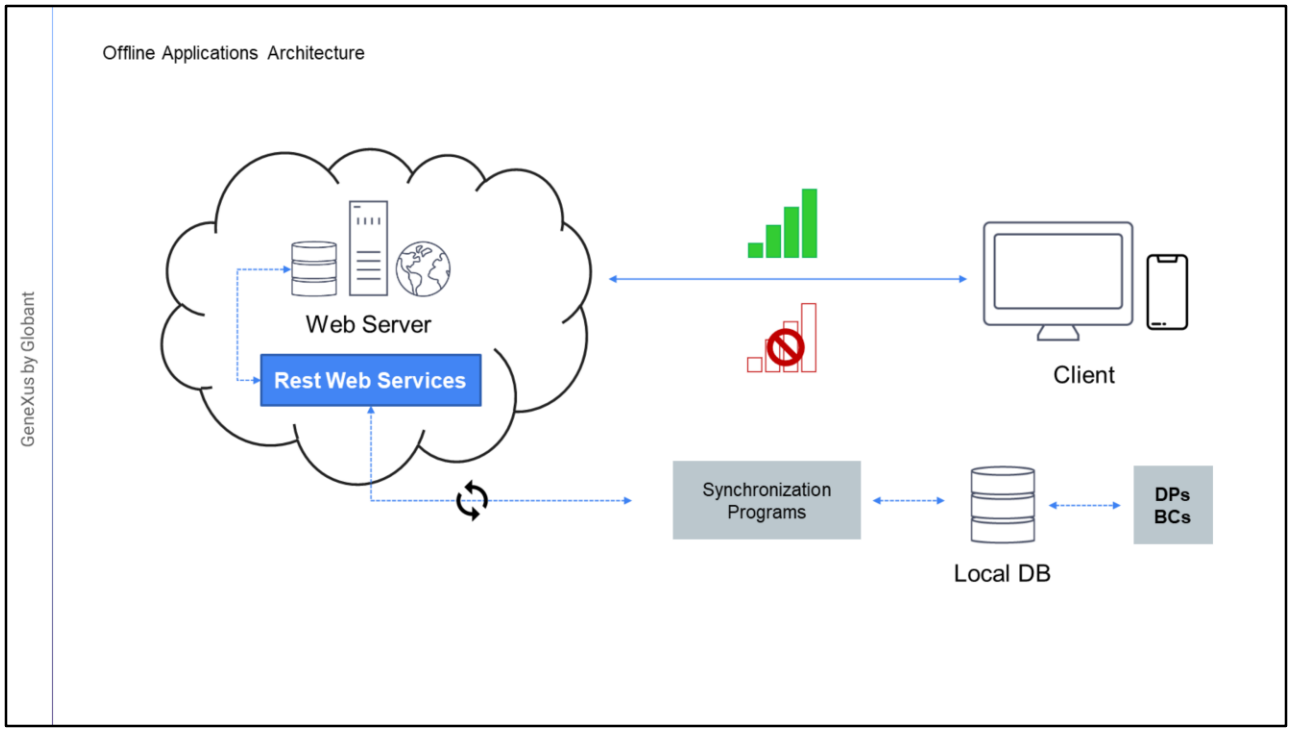
Neste vídeo falaremos sobre a arquitetura das aplicações Offline.

Vamos começar por analisar o caso das aplicações desconectadas, em que queremos que todos os dados tratados pela aplicação móvel sejam acessíveis, mesmo quando não há conexão.

Neste caso, a estrutura da base de dados centralizada no servidor que é manipulada pela aplicação móvel, é espelhada no dispositivo. Ou seja, será criada nele uma base de dados local, SQLite com essas mesmas tabelas.

No entanto, não é obrigatório que seja replicado todo o conjunto de dados da base de dados centralizada, mas poderá ser enviado para a base de dados do dispositivo um subconjunto, de acordo com alguma condição, que pode ter a ver com usuários, com o próprio dispositivo, etc.

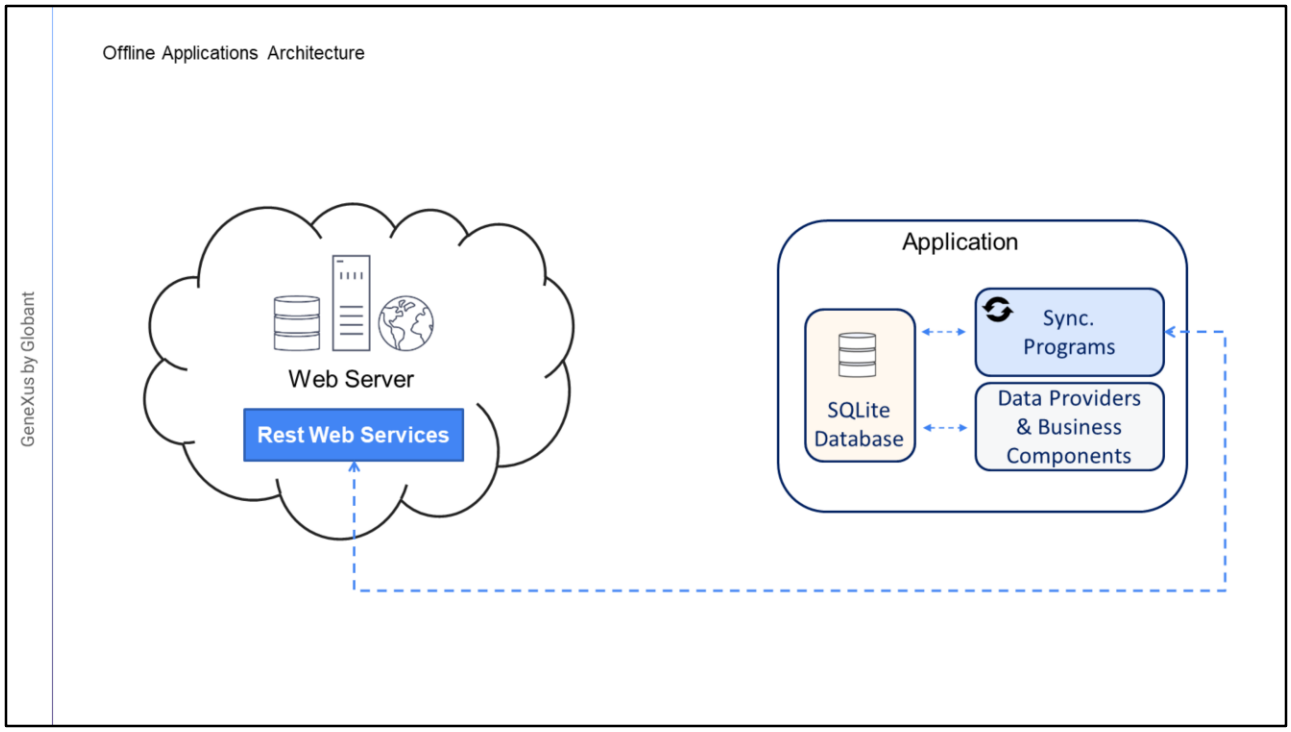
Ou seja, existem mecanismos para especificar filtros sobre os dados a serem enviados à base de dados local, e também não serão incluídas todas as tabelas, apenas as necessárias, veremos tudo isso mais adiante, quando estudarmos o objeto OfflineDatabase.



Nas aplicações offline, além de ter a base de dados local, são necessários todos aqueles programas (data providers e business components) que eram utilizados para obter a informação da base de dados central, que devem agora ser programados nas linguagens das plataformas para dispositivos móveis, de forma que acessem a base de dados local.

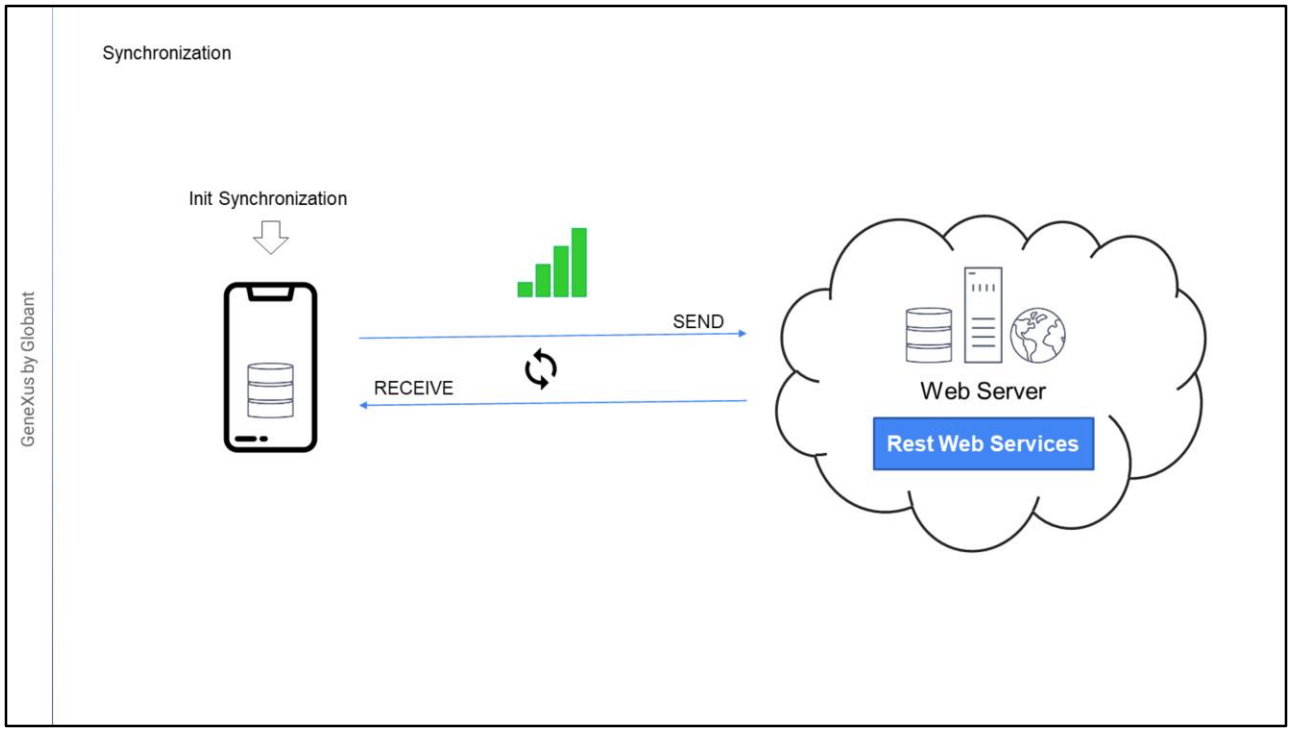
De agora em diante, haja conexão ou não, a aplicação sempre trabalhará sobre a base de dados local.

A aplicação no dispositivo não acessará o servidor exceto para sincronizar os dados de ambas as bases de dados, o que será realizado graças aos programas de sincronização que serão executados no Dispositivo e utilizarão Serviços Rest do lado do Servidor. Esses processos sempre serão iniciados no dispositivo, seja para realizar o envio ou a recepção de dados do server.



Toda a camada de serviços que estava no server web, que continha os data providers para recuperar os dados e os business components para atualizar os dados das tabelas, estarão agora no dispositivo; implementados na linguagem da plataforma, acessando a base de dados local e compilados no binário.

Desta forma todas as operações de CRUD serão sempre sobre a base de dados local e nunca sobre a base de dados do server. O único contato da aplicação com o server será para a sincronização.

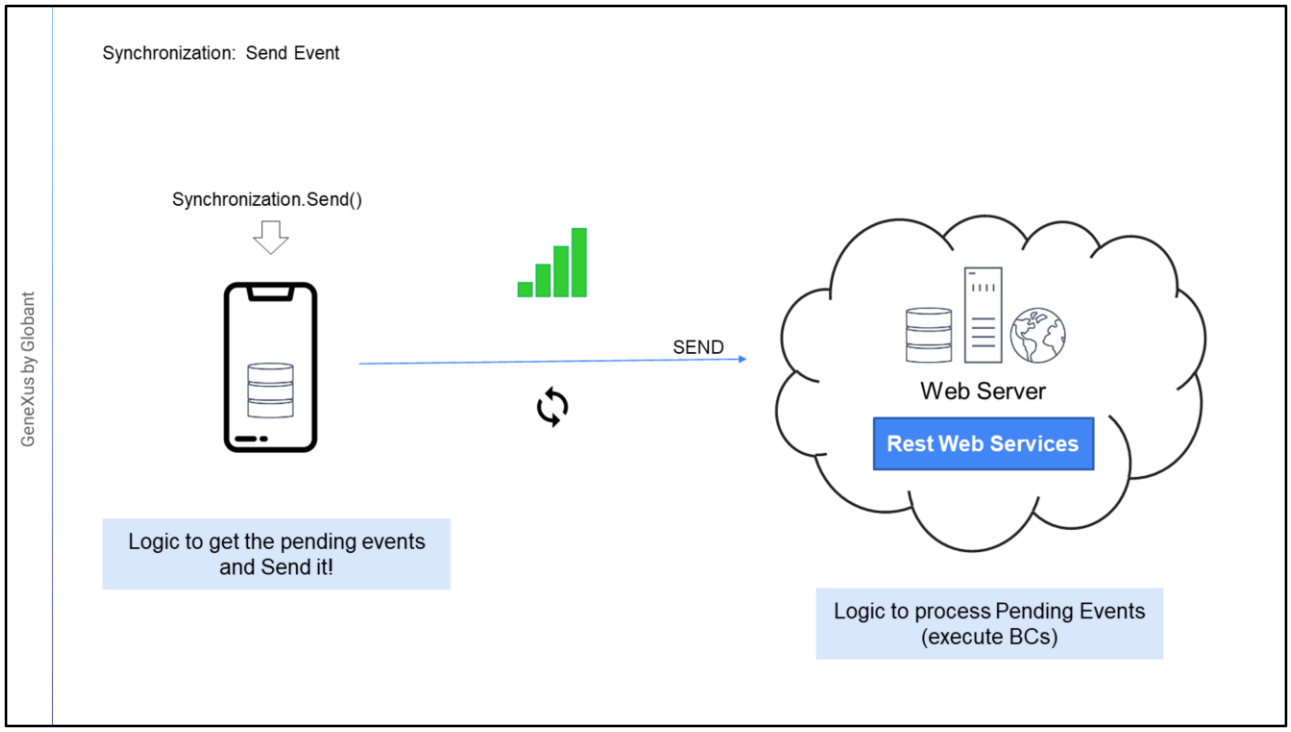


A sincronização sempre será iniciada a partir do dispositivo, pois o servidor não pode saber quando o primeiro obteve conexão.

A informação armazenada localmente pode ser sincronizada com os dados que se encontram no servidor, se desejado; lembramos que você também pode querer nunca sincronizar ou sincronizar sob demanda.
O processo de envio dos dados que foram alterados no dispositivo para o server é denominado: Send

Além disso, os dados do servidor que foram alterados são enviados ao dispositivo para serem atualizados, de tempos em tempos ou a pedido.
O processo de envio dos dados que foram alterados no servidor para o dispositivo é denominado: Receive.

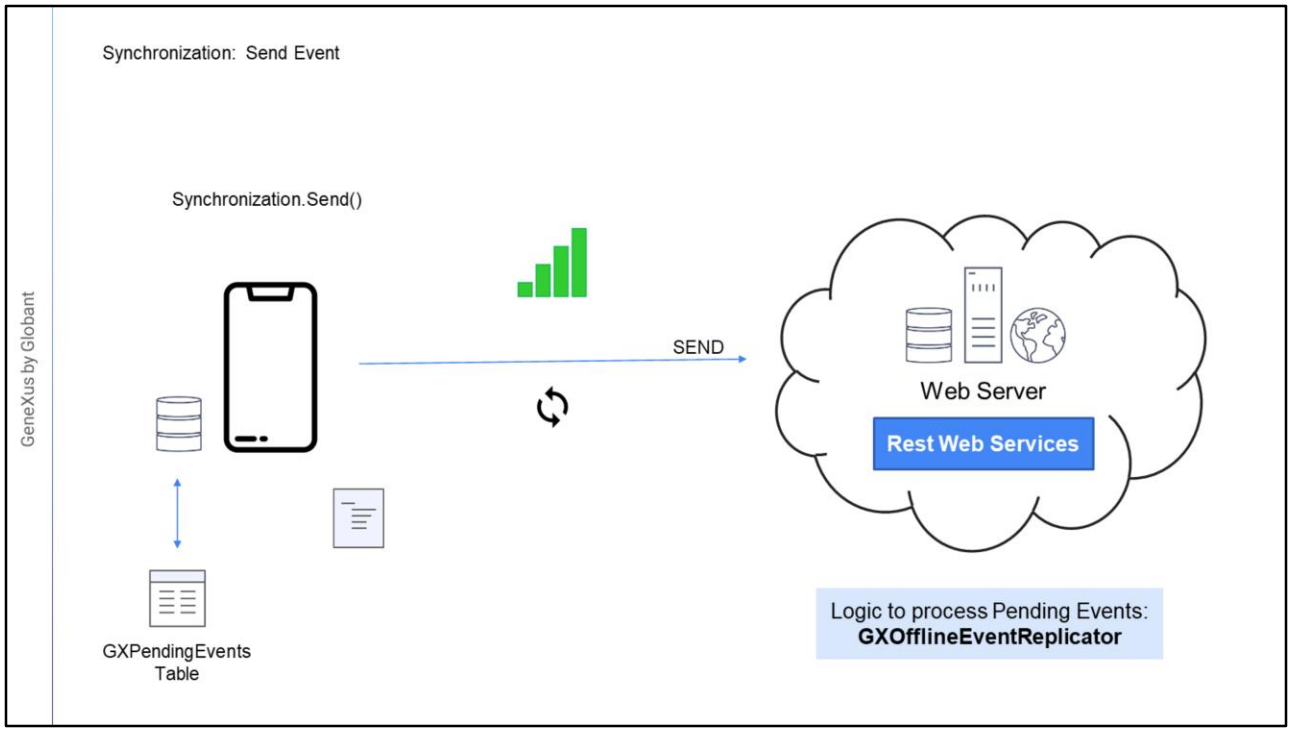
A comunicação entre o dispositivo e o servidor, como já mencionamos, é realizada através da camada de Serviços Rest.



Tanto o Send quanto o Receive são implementados com lógica do lado do server e lógica do lado do cliente. A ideia será que o cliente deva realizar a menor quantidade de processamento possível, pois sua potência é inferior à do server.

Quando o dispositivo inicia o Send (que pode ser iniciado quando for recuperada a conexão, de forma manual através do método Synchronization.Send ou nunca): deve ter montado uma lista ordenada das operações de insert, update e delete que foram realizadas desde a última sincronização. Ou seja, aquelas operações que estão pendentes.

Essa lista é enviada ao processo do lado do server e ele deve percorrer ordenadamente essa lista, e executar a operação correspondente na base de dados, retornando ao processo do lado do cliente o resultado.

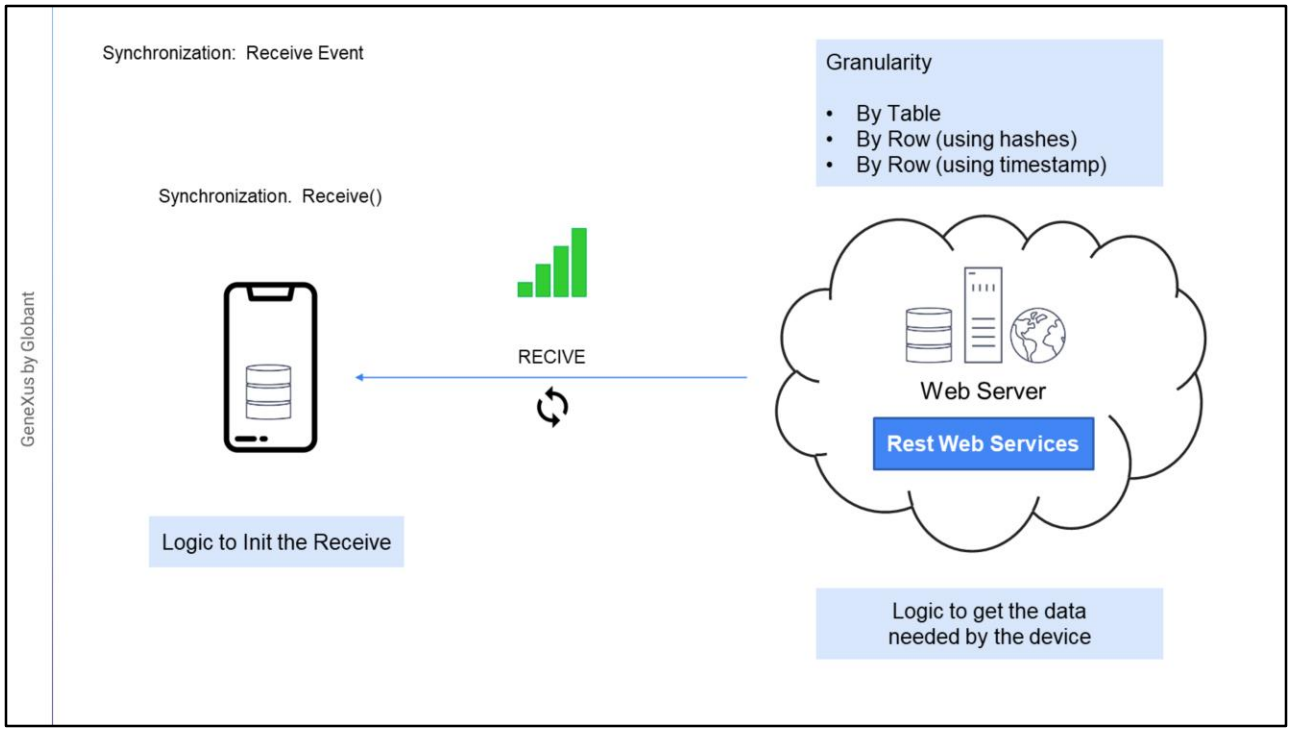


Lembremos que tendo ou não conexão, no cliente se trabalhará sobre a base de dados local.

Todas as modificações realizadas na base de dados local são salvas como “Eventos de sincronização” em uma tabela de uso interno chamada GXPendingEvents.

Esta tabela armazena ordenadamente todas as operações que foram realizadas com Business Components.
São armazenados o nome do BC onde foi realizada a operação, o JSON do BC com os dados do evento, o tipo de operação que foi realizada (entrada, exclusão ou modificação) e o status dela.
Cada vez que o dispositivo executa um business component, é armazenado o evento e permanece em estado “pendente de sincronização”.

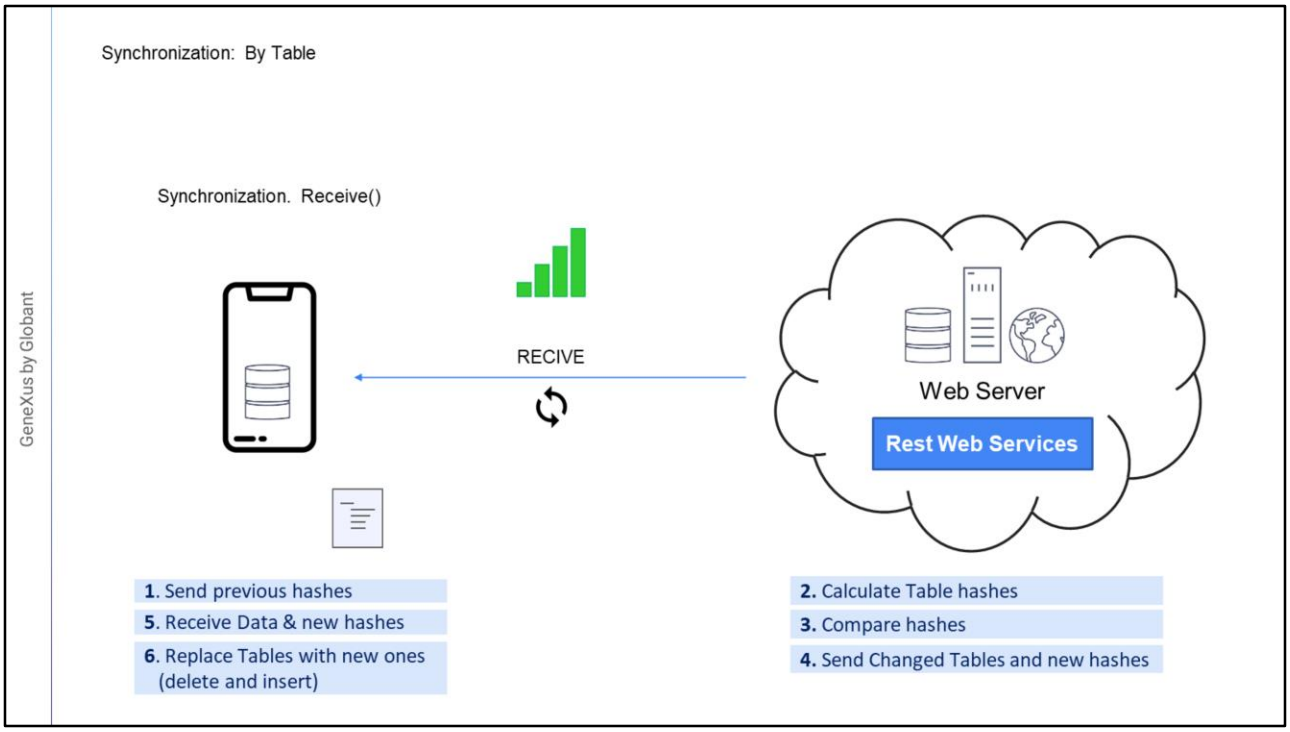
Quando é iniciado o Send, o cliente traduz a lista de todos os eventos com status “Pending” em um SDT com formato JSON e o envia ao server. No server está programado o procedimento GXOfflineEventReplicator, que lê o SDT e realiza as tarefas de Inserção, Atualização e Exclusão, respeitando a ordem das operações.



Quando o dispositivo precisa receber os dados modificados no server, inicia o processo de Receive chamando um serviço Rest no server, o dispositivo então atualiza os dados na base de dados local.

O comportamento da sincronização pode ser configurado de acordo com vários critérios que determinam quando será realizada a sincronização para receber dados.

Além disso, a sincronização pode ser feita de duas formas: por Tabela ou por Linha. Quando a granularidade for By Table, são levadas para o dispositivo todas as tabelas que foram modificadas desde a última sincronização. Quando for By Row, são levados para o dispositivo apenas aqueles registros que foram alterados em cada tabela desde a última sincronização, e existem dois mecanismos, um que utiliza hashes e outro que utiliza timestamp.
Veremos a seguir cada um desses mecanismos e suas diferenças.



A sincronização “by table” é útil em cenários onde a quantidade de registros é pequena, ou muda com muita frequência, pois neste último caso é necessário levar quase tudo em cada sincronização.

Tem a vantagem sobre a sincronização “by row” de que o processamento que requer do lado do servidor é muito menor.

Para determinar quais tabelas foram modificadas e, portanto, devem ser enviadas ao dispositivo, é utilizado um hash, que é o resultado do cálculo de um código que “identifica” o conjunto de dados de cada tabela.

Quando um cliente pede para sincronizar:

São enviados ao servidor os hashes de cada tabela, os quais foram enviados pelo servidor na sincronização anterior.

Então, para cada tabela, o servidor calcula um novo hash com os dados atuais,

Em seguida, compara os hashes com os que possui atualmente

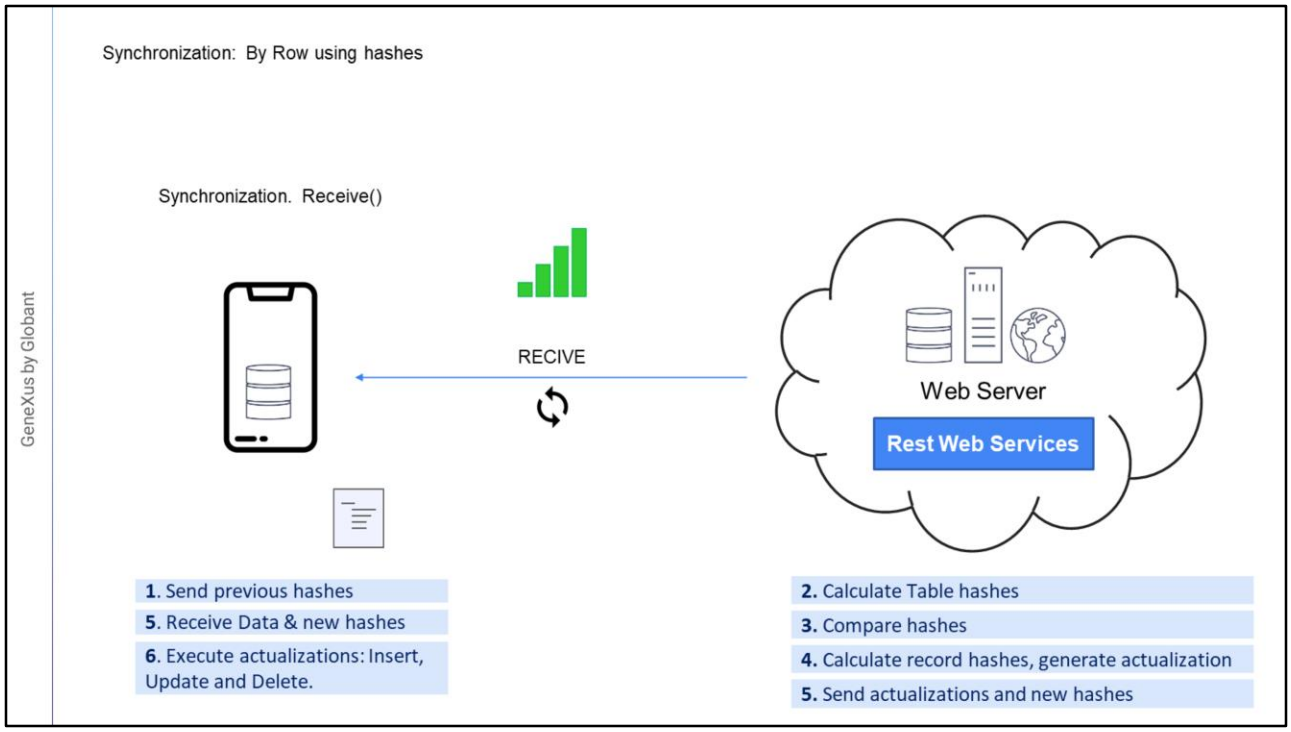
e finalmente envia os dados das tabelas, quando determina que foram modificadas. Se a tabela não foi alterada desde a última sincronização, então, para essa tabela nada é feito.

O dispositivo recebe os dados junto com os hashes.

Por último, o dispositivo substitui as tabelas que foram modificadas (apaga o conteúdo e o gera novamente com a nova informação).

Toda esta comunicação é realizada usando serviços Rest.

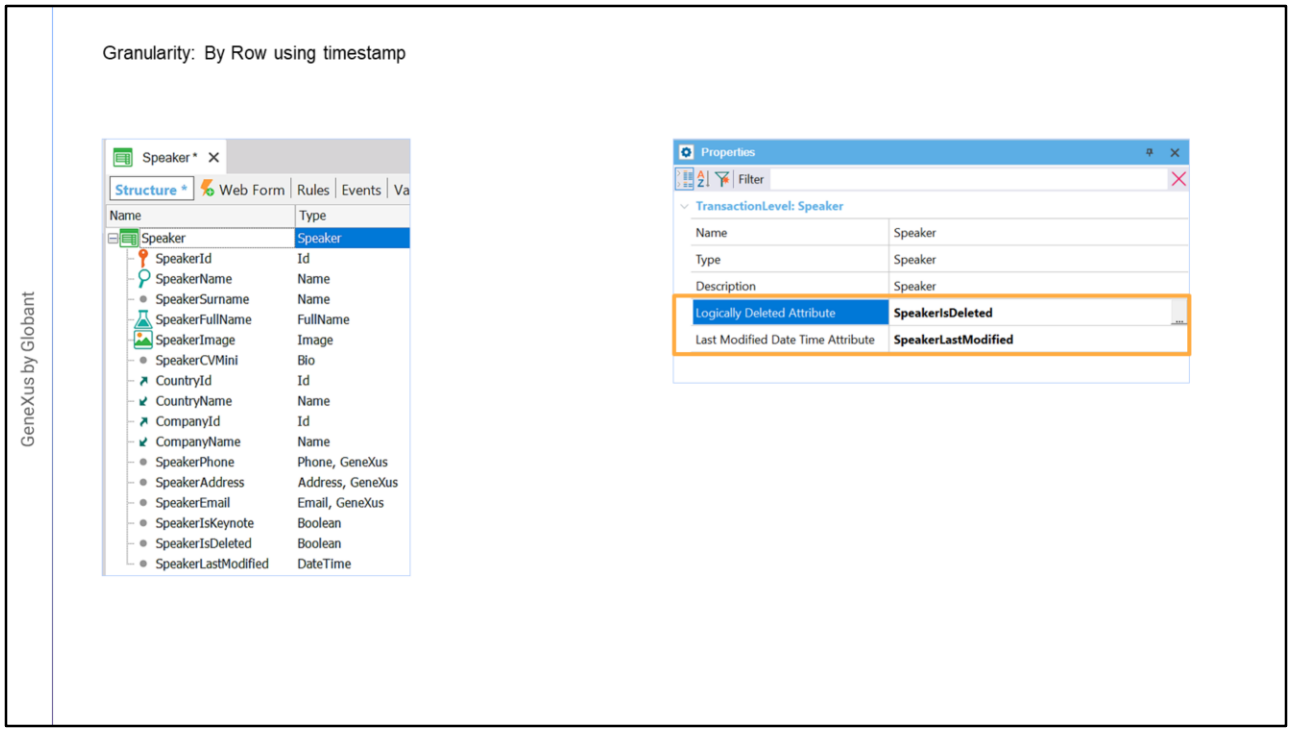
Como caso especial, na primeira sincronização não há dados na base de dados local, portanto são levados todos os dados de todas as tabelas que atendem aos filtros do objeto OfflineDatabase, e os hashes de cada uma.



A sincronização “by row”, utilizando hashes, leva para o dispositivo apenas aqueles registros que foram alterados desde a última atualização utilizando o cálculo de hashes para determinar as atualizações.
A vantagem é que os dados que são transmitidos entre o dispositivo e o servidor são menos, pois são apenas os registros modificados, por outro lado a desvantagem desse mecanismo é que requer maior processamento do lado do servidor, especialmente com grandes volumes de dados.

1. A primeira coisa que acontece é que o dispositivo envia para o server os hashes das tabelas, assim como no caso “By Table”.
2. O server determina qual set de dados deve estar no dispositivo, de acordo com os filtros que possam existir, e calcula um hash para cada set de dados.
3. O server então compara os novos hashes com os atuais e determina quais devem ser enviados ao dispositivo. Se não houver nada para enviar aqui, o processamento é encerrado.
4. Para cada set de dados que deve ser enviado, o server calcula um hash para cada registro e os compara com os hashes atuais, com base nisso pode acontecer que o registro seja o mesmo, nesse caso não será enviado, se o hash é diferente significa que aquele registro mudou e deverá ser enviado como uma atualização. Também pode acontecer que este registro não exista no set anterior, então ele será enviado como um insert e por último é realizada uma comparação para ver quais registros existiam nos hashes atuais e que nos novos não existem mais, estes serão enviados como exclusões. Para cada ação a ser realizada, insert, update e delete, é preparada uma lista.
5. O Server envia as listas com os novos dados ao dispositivo.

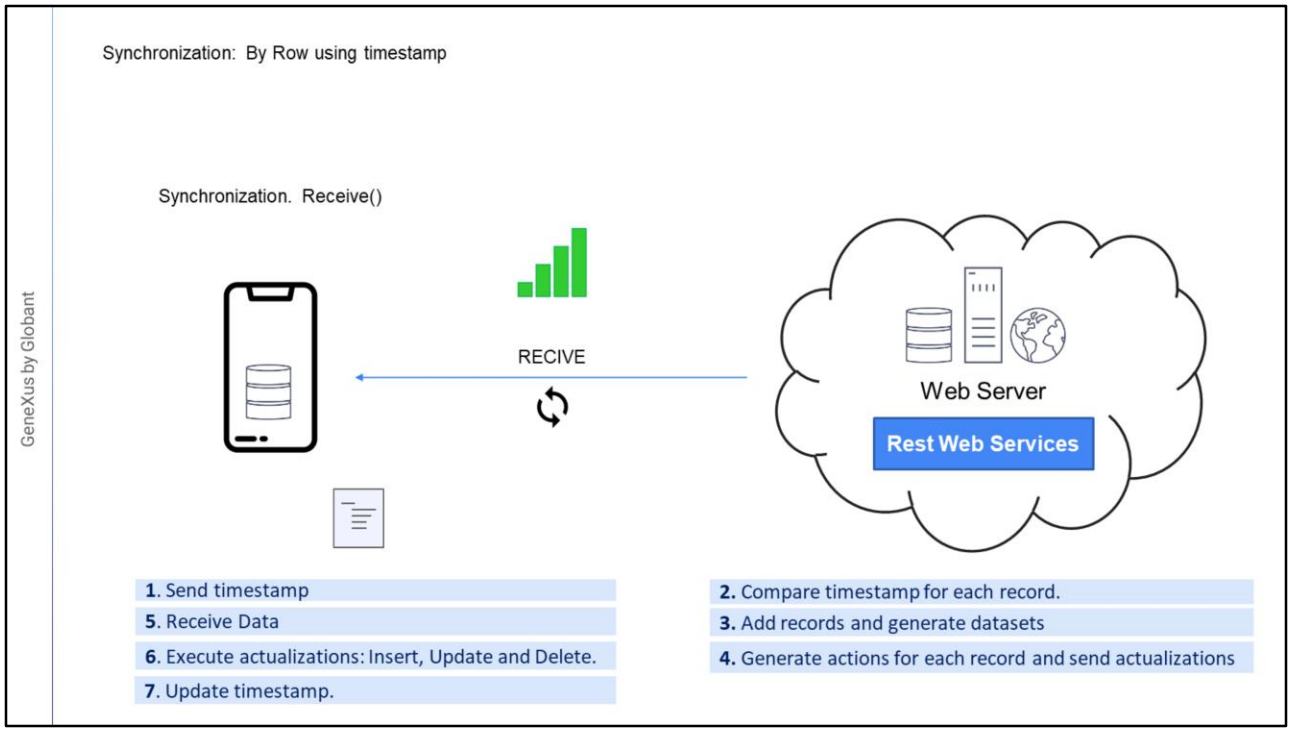
6. O dispositivo então recebe os dados e salva o hash de cada tabela.
7. Finalmente, suas listas são processadas em ordem. Para os registros novos é feito um INSERT na base de dados, e se falha por chave duplicada é feito um UPDATE. Para os registros modificados é feito um UPDATE, e se não existe o registro é feito o INSERT deles. Para os registros excluídos, é feito um DELETE.



Vejamos o mecanismo By Row utilizando timestamp.

Este mecanismo utiliza uma abordagem diferente, a sincronização continua sendo feita por registro, mas não serão utilizados hashes para determinar se é uma atualização ou não, em vez disso será utilizada a data de atualização e a marca de exclusão lógica.

Por este motivo, para utilizar este mecanismo devemos indicar para cada nível de uma transação qual atributo conterá a exclusão lógica e qual atributo conterá a data e hora da última modificação.

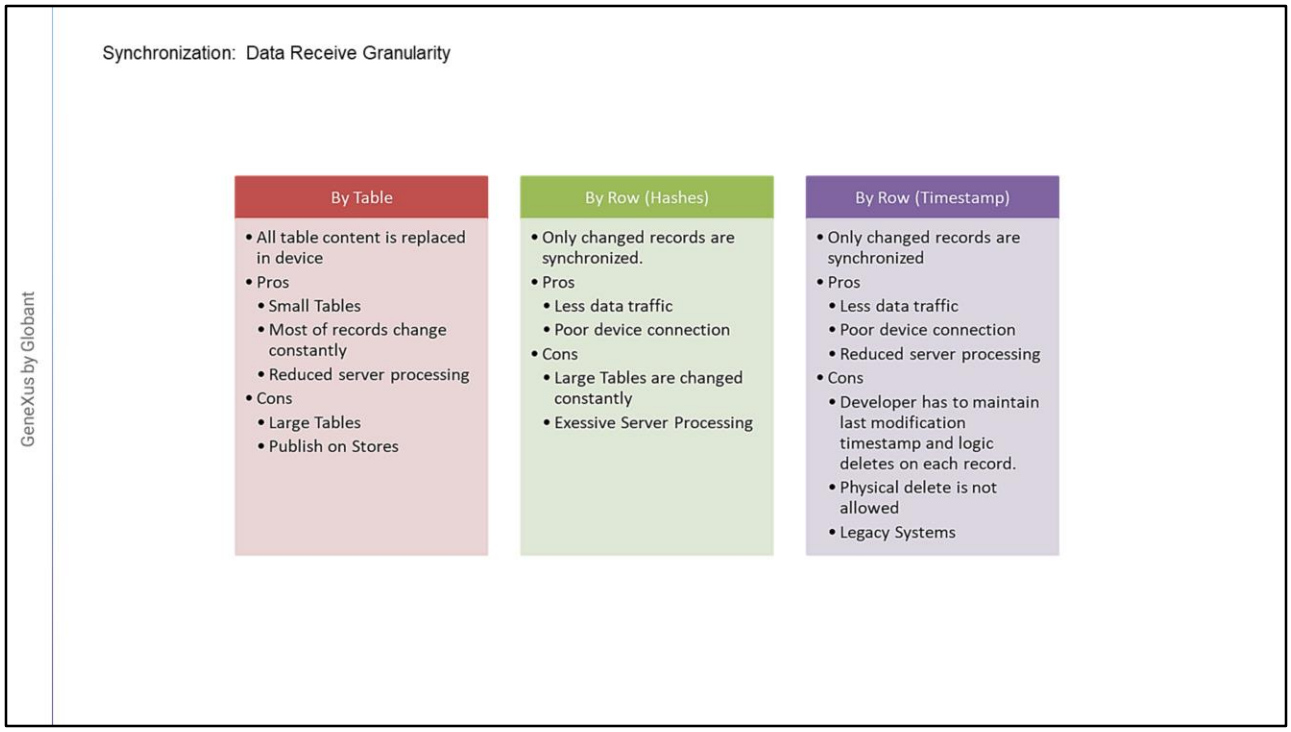


Usando este mecanismo, a primeira vez que é necessário sincronizar o server enviará todos os dados que atendam às condições, junto com o timestamp da sincronização, o dispositivo armazenará o timestamp que será utilizado posteriormente nas sucessivas sincronizações.

Depois, a cada nova sincronização.
O dispositivo envia o timestamp da última sincronização.
o server compara o timestamp recebido do dispositivo com o de cada registro utilizando o atributo de cada tabela.
Todos os registros que foram adicionados ou inseridos após o timestamp do dispositivo são adicionados a uma lista.
Para determinar a ação a ser tomada sobre cada registro, será avaliado se o registro foi excluído, utilizando o atributo com a marca de exclusão lógica, em caso afirmativo, esse registro é marcado como uma exclusão que deve ser enviada ao dispositivo, o restante dos registros é marcado para atualização, no dispositivo será realizado um "upsert", ou seja, trata-se de atualizar se existe o registro, caso contrário é inserido. Toda esta informação é enviada ao dispositivo.
O dispositivo recebe os dados
Realiza as operações e atualiza o timestamp.

As desvantagens deste mecanismo são que o desenvolvedor é responsável por manter o timestamp e a marca de exclusão lógica, além disso, não podemos usar exclusão física de registros nessas tabelas.

Este mecanismo pode ser utilizado em conjunto com o anterior, by row using hashes.



Vamos fazer uma revisão dos mecanismos de sincronização que acabamos de ver.

Em relação à sincronização By Table, o que acontecerá é que quando no server se determine que uma tabela foi modificada, essa tabela será enviada ao dispositivo e lá será substituída de forma completa. Para determinar as tabelas, será utilizado um hash para cada tabela e para cada dispositivo.

Quando é útil esse mecanismo?, por exemplo, quando nossas tabelas são pequenas ou quando muda a maioria dos registros constantemente, então é preferível tratá-la desta forma. Por exemplo, poderíamos utilizá-la num sistema móvel interno onde os vendedores têm a lista de clientes para visitar e essa lista muda todos os dias, hoje me atribuem uma lista e amanhã é outra completamente diferente.

Qual é a desvantagem desta opção, quando as tabelas são muito grandes, pois o tráfego de dados será importante, também não seria recomendável em casos em que a aplicação vai ser de uso massivo, ou seja, que estará publicada em uma Store, se cada vez que for sincronizar, serão enviados todos os dados, a experiência do usuário não será muito boa.

Pois bem, para salvar estes casos é que contamos com a sincronização por registros.

Aqui temos duas opções, fazê-lo usando hashes ou por timestamp.

Vejamos a primeira opção.

Neste caso, o server determinará de acordo com os hashes que irá calcular para cada set de dados e para cada registro, quais deve enviar para cada dispositivo. Então só se enviarão as novidades, apenas os registros que foram modificados.

A vantagem é que o tráfego dos dados é reduzido consideravelmente, a menos que

constantemente seja modificado um grande volume de dados.

Voltando ao exemplo anterior, em vez de receber toda a tabela de clientes, receberemos apenas aqueles que sofreram alguma modificação.

Suponhamos que recebemos apenas os clientes ativos, então o dispositivo tem um hash, aquele da última sincronização, e quando quer sincronizar ele envia esse hash ao server, que recalcula um hash para os clientes ativos e o compara com aquele enviado a ele pelo dispositivo, se forem diferentes, então o server irá gerar um hash para cada registro dessa consulta e irá compará-los com os que tinha anteriormente, dessa forma poderá determinar exatamente o que foi adicionado, modificado ou excluído.

Esta é a opção padrão quando é criada uma aplicação Offline.

Este método também é útil quando a conectividade não é muito boa, pois temos menos tráfego.

Como desvantagem teremos que é exigido muito mais processamento do lado do server e não seria recomendável no caso em que tenhamos grandes volumes de dados que são modificados de forma constante.

Suponhamos este caso, temos em nosso sistema uma tabela com milhares de produtos que queremos que estejam em cada dispositivo, também temos centenas de usuários. Sincronizar por Tabela não seria adequado por se tratar de uma tabela com muitos dados, e sincronizar por registro usando hashes também pode não ser a solução, pois para poder determinar quais registros deve enviar para cada dispositivo, deve ser calculado e comparado o hash de cada um dos milhares de produtos que temos. Isto, somado à concorrência de muitos usuários simultaneamente, pode gerar um problema de processamento do lado do server.

Portanto, nenhuma das opções é ideal, vejamos então a terceira opção, que é usar timestamp.

O que teremos usando o timestamp é manter a transferência de dados ao mínimo, apenas para os registros modificados, mas por sua vez envolvendo menos processamento do lado do server, já que não tem que calcular os hashes para toda a consulta, mas pode determinar quais foram pela data e hora de última atualização e pela marca de exclusão lógica.

Aqui precisamos manter estes dois atributos adicionais, a marca de exclusão lógica e a data e hora de última modificação, o bom é que muitos sistemas já implementam esses dados em cada registro.

A desvantagem deste mecanismo é que é uma responsabilidade do desenvolvedor manter justamente estes dois atributos, algo que pode ser complexo quando há muitos sistemas diferentes envolvidos.

Também podemos ter um mix entre os dois últimos mecanismos, se usamos By Row e em uma transação não configuramos os atributos de timestamp e exclusão lógica, serão utilizados hashes.

Para pesquisar sobre este tema, convidamos você para nossa wiki.


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