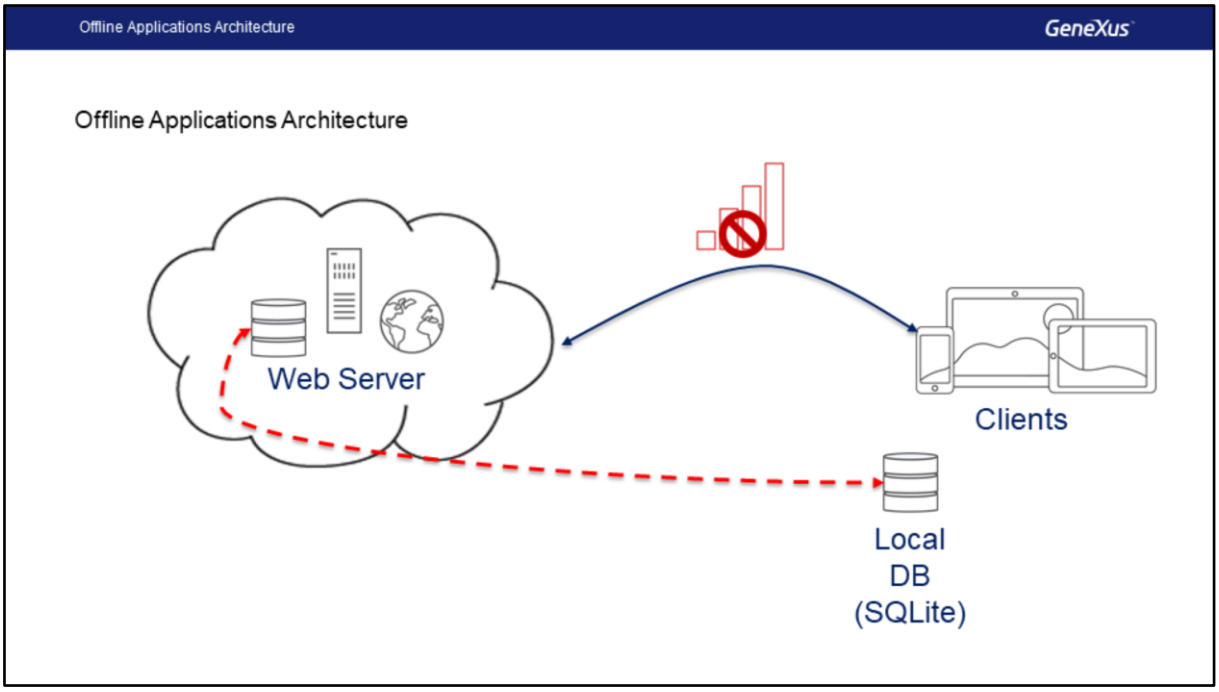

GeneXus™
The power of doing.

Offline Applications Architecture

Development – Offline Applications

GeneXus™ 16

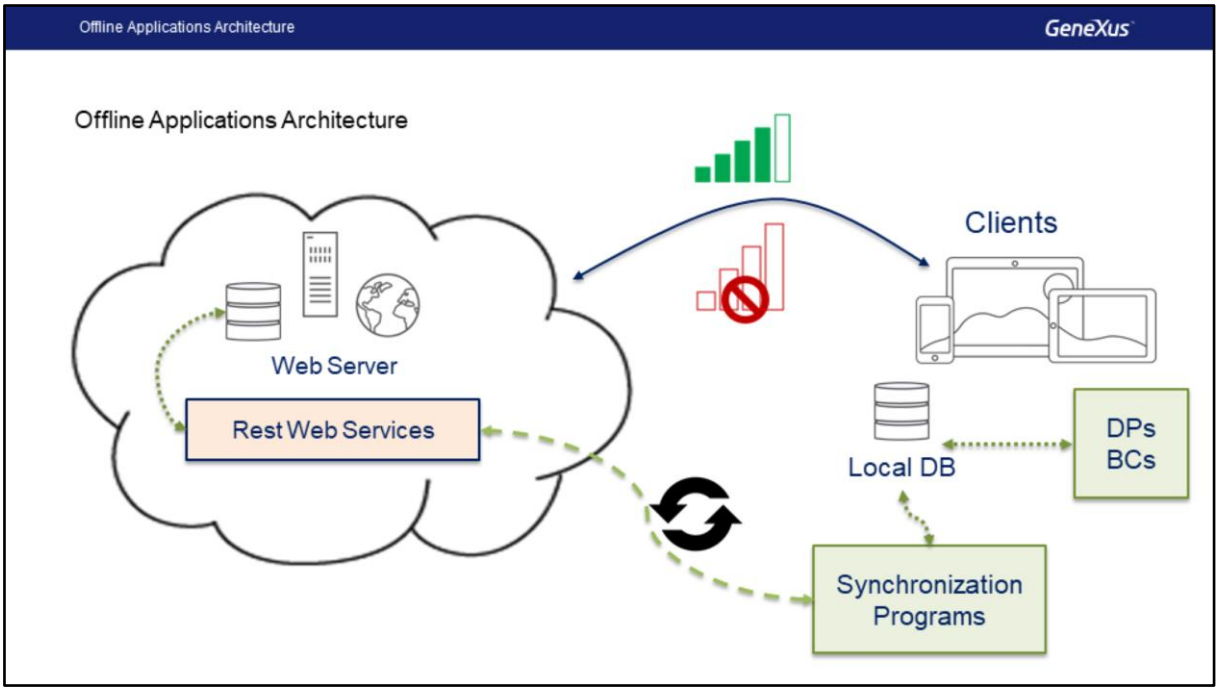


Neste vídeo, falaremos sobre a arquitetura de aplicativos off-line.

Vamos começar analisando o caso de aplicativos desconectados, onde todos os dados manipulados pelo aplicativo móvel estejam acessíveis mesmo quando não há conexão.

Nesse caso, a estrutura do banco de dados centralizada no servidor que é manipulado pelo aplicativo móvel é espelhada no aparelho. Ou seja, um banco de dados local, o SQLite com essas mesmas tabelas será criado no aparelho.

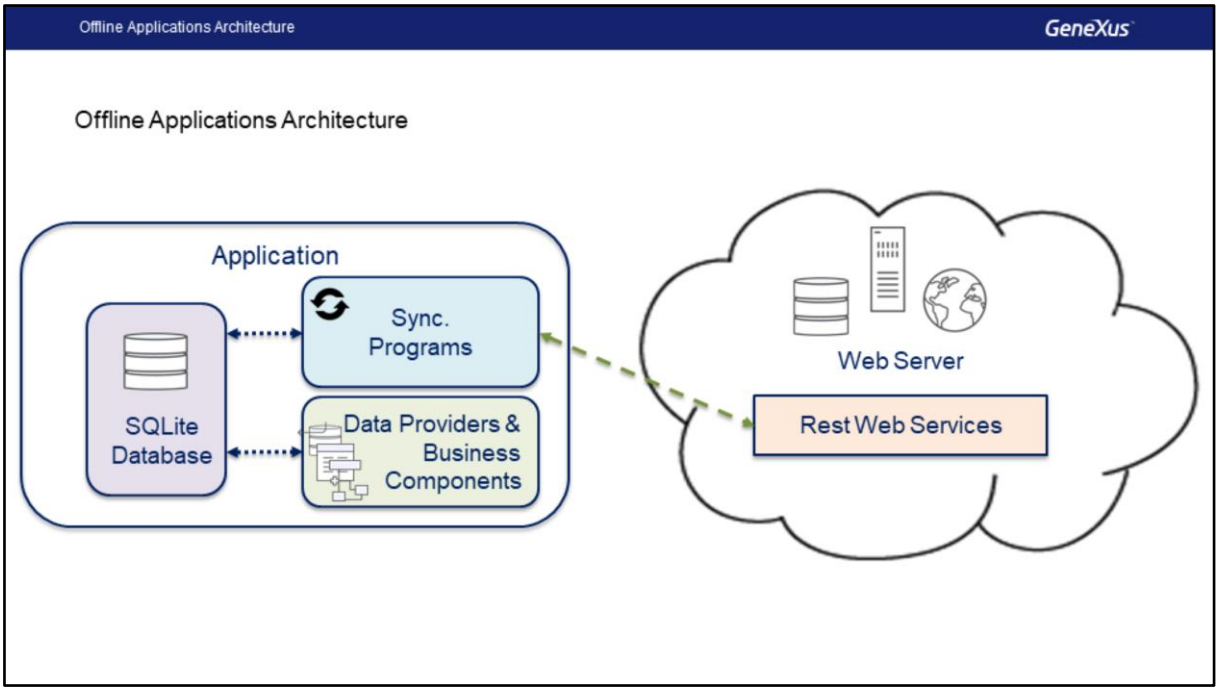
No entanto, não é obrigatório replicar todo o conjunto de dados do banco de dados centralizado, mas um subconjunto pode ser enviado para o banco de dados do dispositivo, de acordo com alguma condição, que pode ter a ver com os usuários, com o próprio dispositivo, etc.
Ou seja, existem mecanismos para especificar filtros nos dados a serem enviados para o banco de dados local, e nem todas as tabelas serão incluídas, apenas as necessárias, tudo isso veremos mais adiante quando estudarmos o objeto OfflineDatabase.



Em aplicativos off-line, além de ter o banco de dados local, e necessário levar também todos os programas (data providers e business components) que foram usados para obter as informações do banco de dados central, e agora devem ser programados na linguagem nativa do Smart Devices, de forma que acessem o banco de dados local.

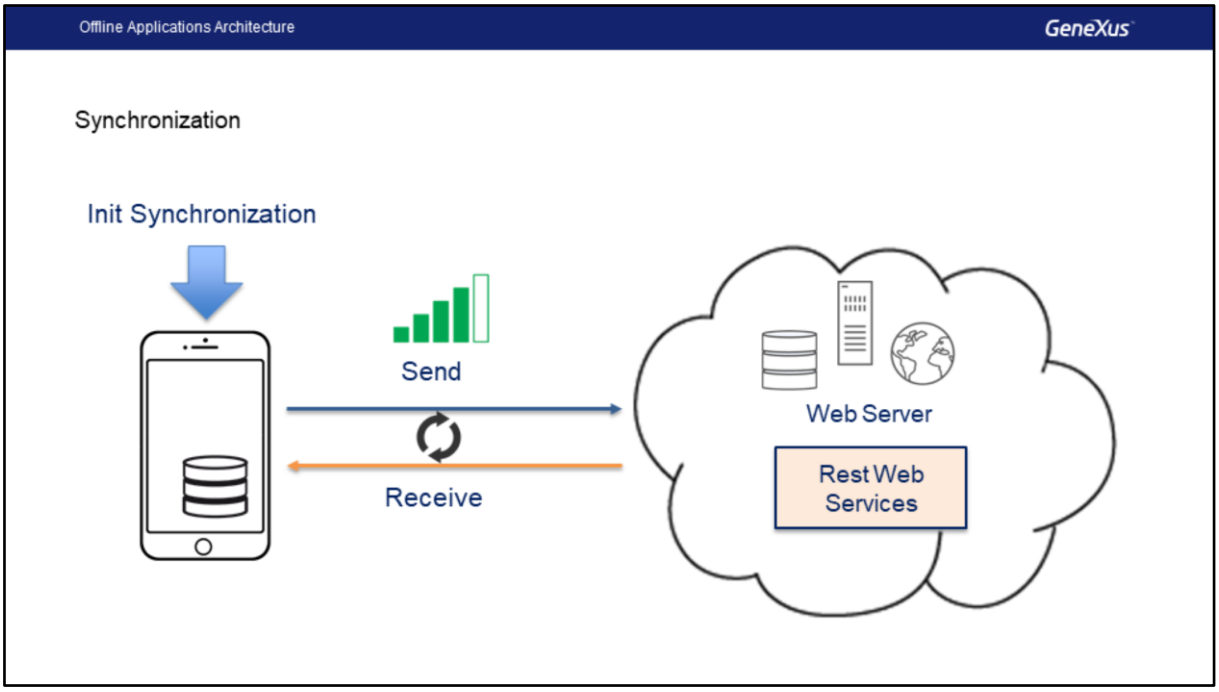
De agora em diante, se houver uma conexão, ou não, o aplicativo sempre trabalhará com o banco de dados local.

O aplicativo no aparelho não acessará o servidor, exceto para sincronizar os dados de ambos os bancos de dados, o que será feito graças aos programas de sincronização que serão executados no aparelho e usarão os serviços Rest no lado do servidor. Esses processos sempre serão iniciados no dispositivo, para enviar ou receber dados do servidor.



Toda a camada de serviço que estava no servidor web, que continha data providers para recuperar os dados e os business components para atualizar os dados das tabelas, agora estarão também no aparelho; implementados na linguagem nativa da plataforma, acessando o banco de dados local e compilados no binário.

Desta forma, todas as operações CRUD serão sempre baseadas em dados locais e nunca no banco de dados do servidor. O único contato do aplicativo com o servidor será para sincronização

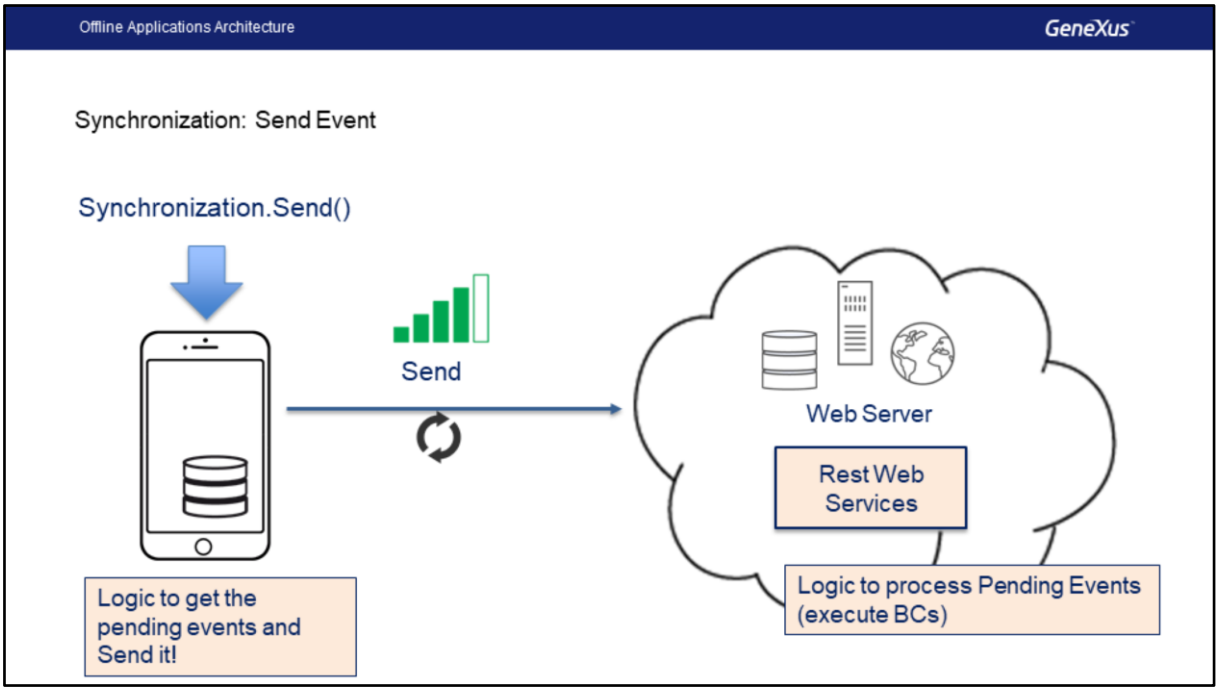


Sempre a sincronização será iniciada a partir do aparelho, uma vez que o servidor não consegue saber quando o aparelho obteve uma conexão.

As informações armazenadas localmente podem ser sincronizadas com os dados que se encontram no servidor (se for esse o caso, lembre-se de que você também pode nunca sincronizar ou sincronizar parcialmente).
O processo de envio dos dados que foram alterados no dispositivo para o servidor é chamado: Send

Além disso, os dados alterados no servidor também são enviados para o aparelho a ser atualizado em um determinado momento ou sob demanda.
O processo de envio dos dados que foram alterados no servidor para o dispositivo é chamado: Receive.

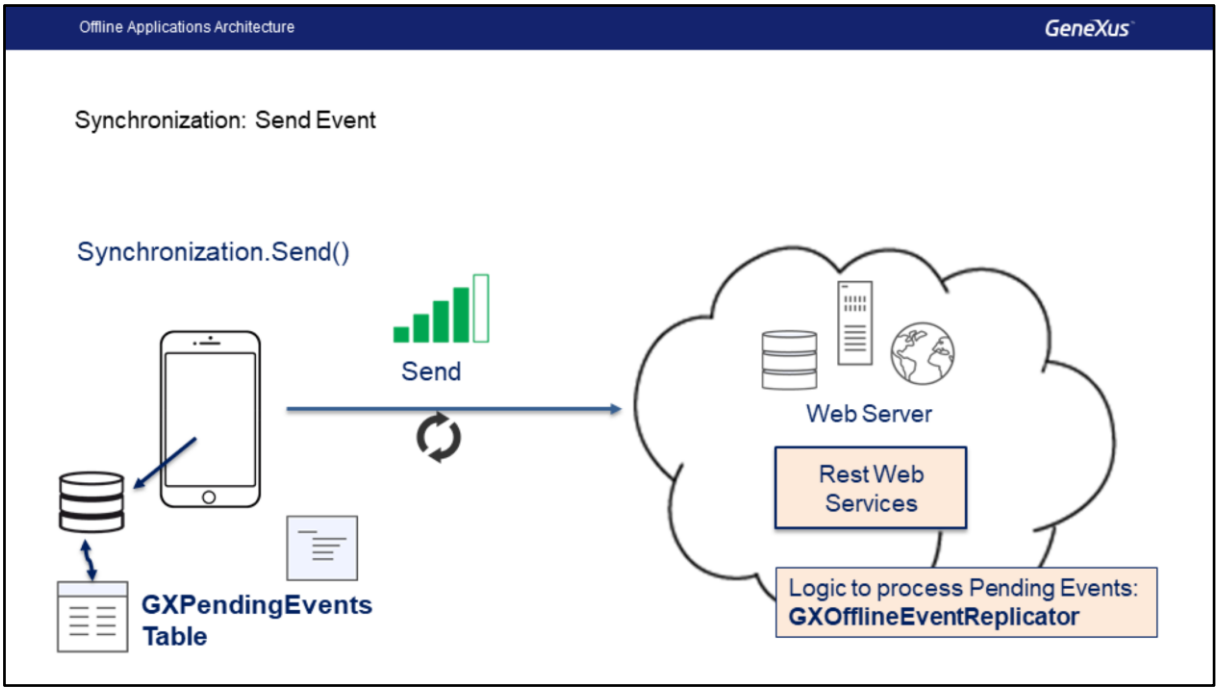
A comunicação entre o dispositivo e o servidor, como já mencionado, é feita através da camada de serviços Rest.



Tanto o Send quanto o Receive são implementados com lógica no lado do servidor e lógica no lado do cliente. A idéia é que o cliente deve executar a menor quantidade possível de processamento, porque sua capacidade de processamento é menor que a do servidor.

Quando o dispositivo inicia o Send (que pode ser iniciado no momento em que se estabelece uma conexão, manualmente através do método Synchronization.Send ou nunca sincronizar): ele deve ter montado uma lista ordenada das operações de inserção, atualização e exclusão executadas desde a última sincronização. Ou seja, as operações que estão pendentes.

Essa lista é enviada para o processo no lado do servidor e este deve percorrer essa lista de maneira ordenada e executar a operação correspondente no banco de dados central, retornando o resultado para o processo no lado do cliente.

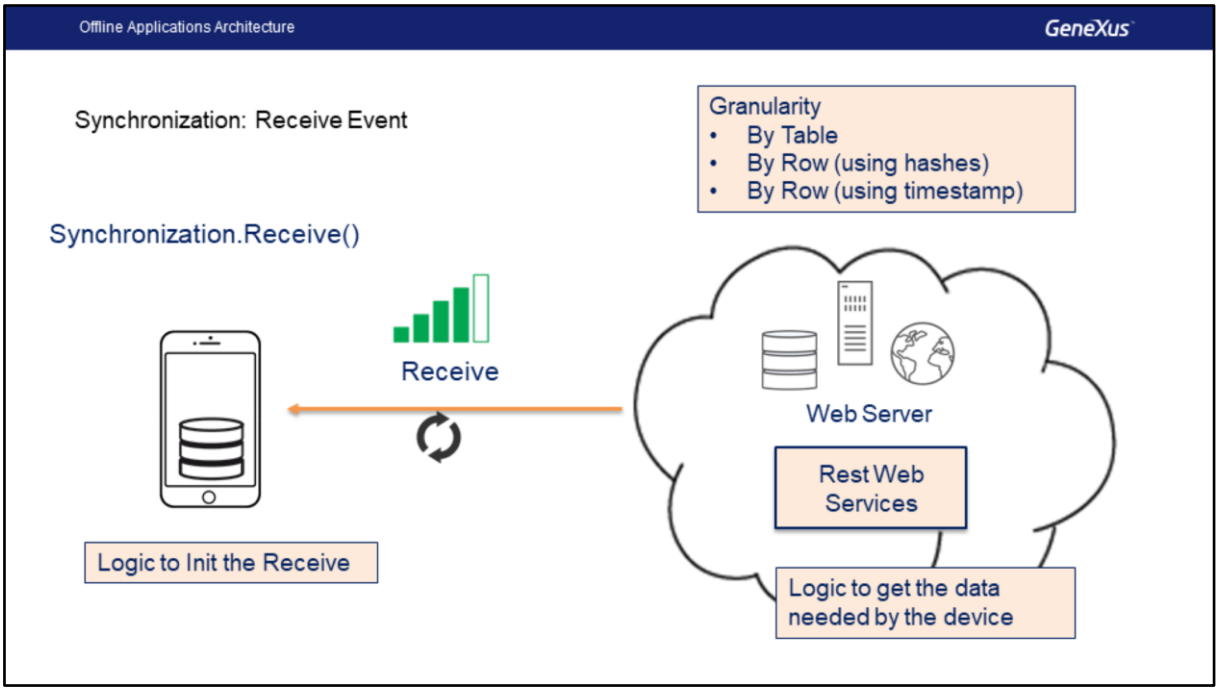


Lembre-se de que, independentemente de termos ou não uma conexão, o Client sempre funcionará no banco de dados local.

Todas as modificações feitas no banco de dados local são salvas como “Eventos de Sincronização" em uma tabela de uso interno chamada GXPendingEvents.

Essa tabela armazena em ordem todas as operações que foram executadas com Business Components.
É armazenado o nome do BC onde a operação foi executada, o JSON do BC com os dados do evento, o tipo de operação que foi executada (inserção, alteração ou modificação) e o estado do mesmo.
Cada vez que o dispositivo executa um business component, o evento é armazenado e fica no estado "pendente de sincronização".

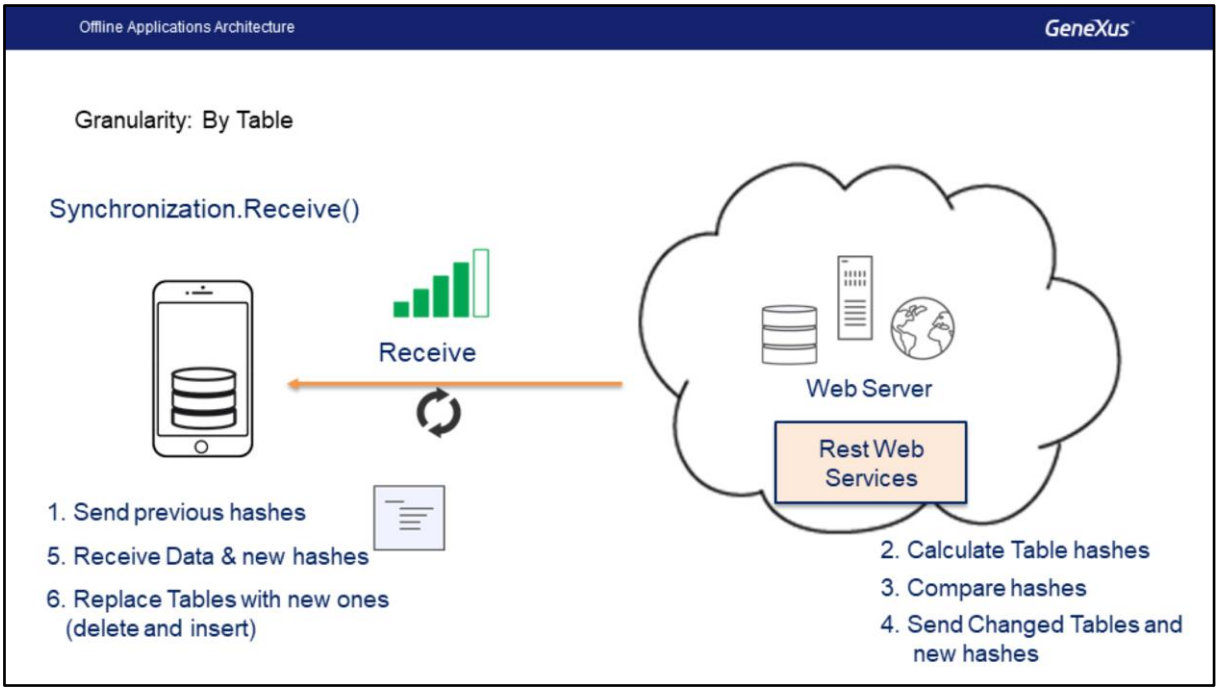
Quando inicia o Send, o cliente converte a lista de todos os eventos com o status "Pending" em uma SDT no formato JSON e envia para o servidor. O procedimento GXOfflineEventReplicator programado no servidor lê a SDT e executa as tarefas Insert, Update e Delete, respeitando a ordem das operações.



Quando o dispositivo precisa receber os dados modificados no servidor, ele inicia o processo de Receive chamando um serviço Rest no servidor e o dispositivo atualiza os dados no banco de dados local.

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

Além disso, a sincronização pode ser feita de duas maneiras: por tabela ou por linha. Quando a granularidade é By Table , todas as tabelas que foram modificadas desde a última sincronização são levadas para o dispositivo. Quando é By Row, somente os registros que sofreram modificações de cada tabela, desde a última sincronização, são transportados para o dispositivo, e existem dois mecanismos, um que usa hashes e outro que usa timestamp.
Vamos ver abaixo cada um desses mecanismos e suas diferenças.



Sincronização "by table" é útil em cenários onde o número de registros é pequeno, ou muda muito freqüentemente, já que neste último caso é necessário levar quase tudo a cada sincronização.
Ele tem a vantagem sobre a sincronização "by row" de que o processamento requerido no lado do servidor é muito menor.

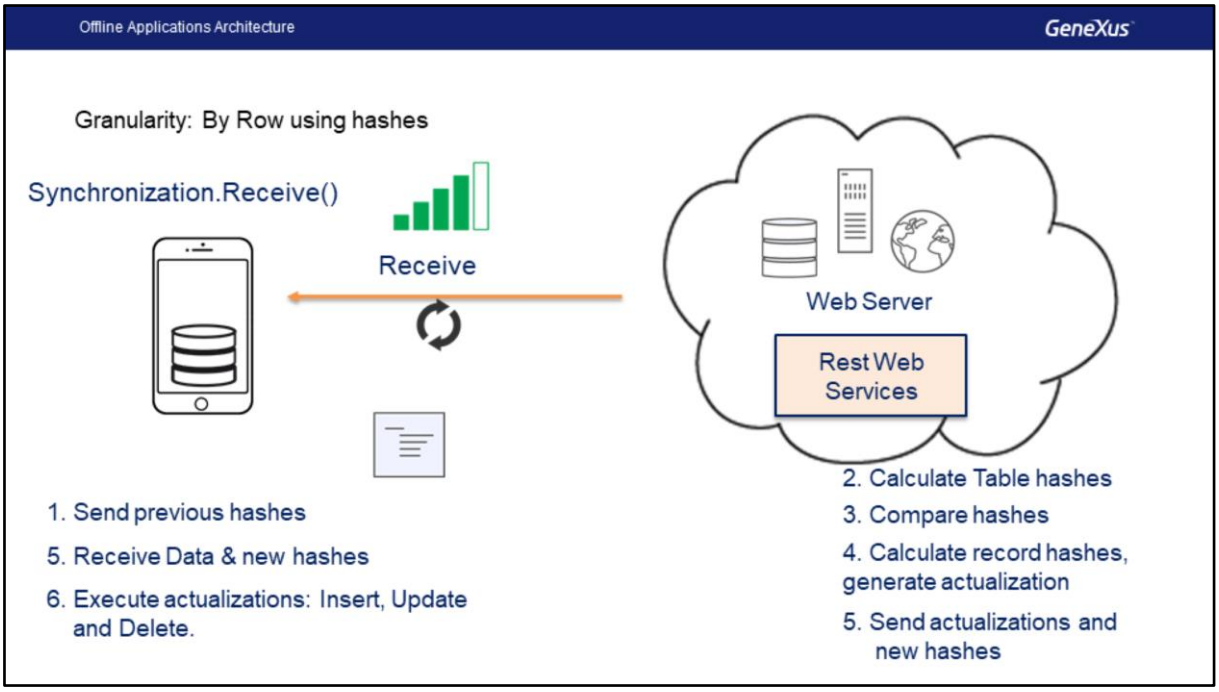
Para determinar quais tabelas foram modificadas e, portanto, devem ser enviadas para o dispositivo, é usado um hash, que é o resultado do cálculo de um código que "identifica" o conjunto de dados de cada tabela.

Quando um cliente pede para sincronizar:

1. Os hashes de cada tabela são enviados para o servidor, que foram enviados pelo servidor na sincronização anterior.
2. Então, para cada tabela, o servidor calcula um novo hash com os dados atuais,
3. Em seguida, compara os hashes com os atuais
4. E finalmente envia os dados das tabelas, quando identifica que sofreram alterações. Se a tabela não foi alterada desde a última sincronização, nada será feito para ela.
5. O dispositivo recebe os dados juntamente com os novos hashes.
6. Finalmente, o dispositivo substitui as tabelas que foram modificadas (exclui o conteúdo e o gera novamente com as novas informações).

Toda essa comunicação é feita usando serviços Rest.

Como exceção, na primeira sincronização, não há dados no banco de dados local, portanto, todos os dados de todas as tabelas que estão em conformidade com os filtros são transportados no objeto OfflineDatabase e nos hashes de cada um.

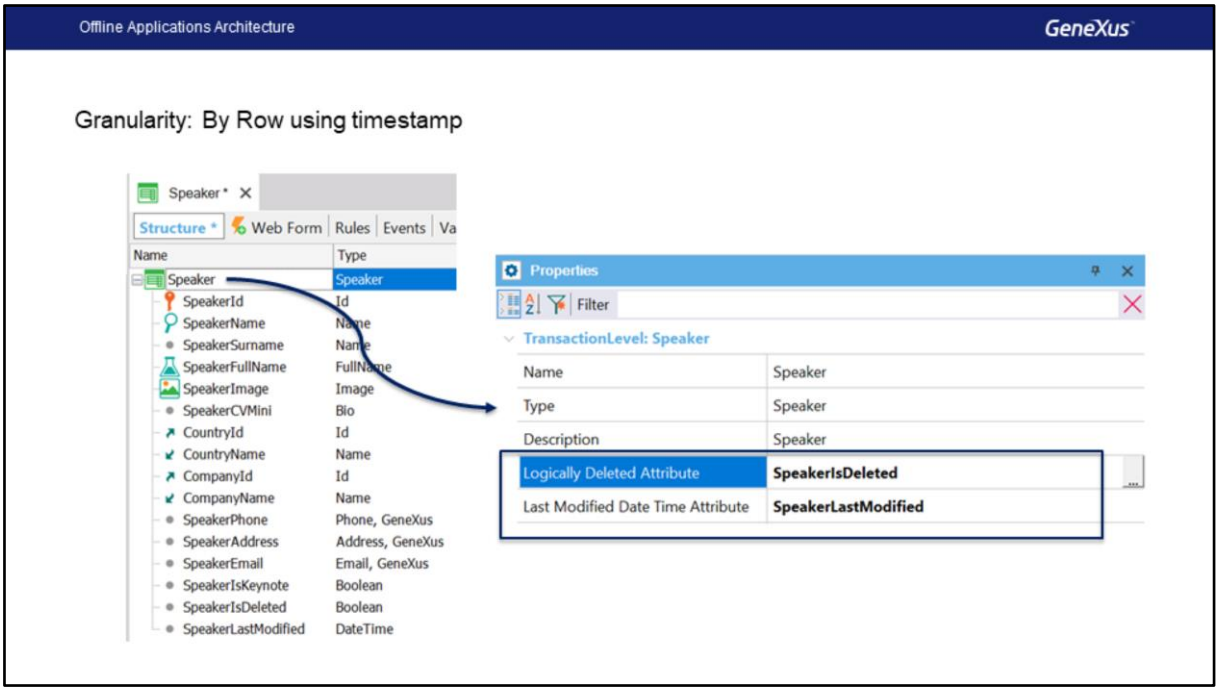


A sincronização "by row" usando hashes leva apenas para o dispositivo os registros que foram alterados desde a última sincronização, usando o processamento hash para determinar as atualizações. A vantagem é que os dados transmitidos entre o dispositivo e o servidor são menores, porque considera apenas os registros modificados. Por outro lado, a desvantagem desse mecanismo é que ele requer mais processamento no lado do servidor, especialmente com grandes volumes de dados.

1. A primeira coisa que acontece é que o dispositivo envia os hashes das tabelas para o servidor, assim como no caso "By Table".
2. O servidor determina qual conjunto de dados deve vir para o dispositivo, de acordo com os filtros que podem existir, e calcula um hash para cada conjunto de dados.
3. O servidor compara os novos hashes com os atuais e determina quais devem ser enviados para o dispositivo. Se não houver nada para enviar, o processamento é finalizado.
4. Para cada conjunto de dados a serem enviados, o servidor calcula um hash para cada registro e compara com hashes atuais. Com base nisso, pode ser que o registro seja o mesmo, nesse caso não será enviado. Se o hash é diferente significa que esse registro foi alterado e deve ser enviado como uma atualização. Também pode acontecer que o registro não existe no set anterior, por isso vai ser enviado como uma inserção. Finalmente, é feita uma comparação para ver quais registros existiam nos hashes atuais

e que já não aparecem nos novos, eles serão enviados como exclusões . Para cada ação a ser executada, inserir, atualizar e excluir, é preparada uma lista.

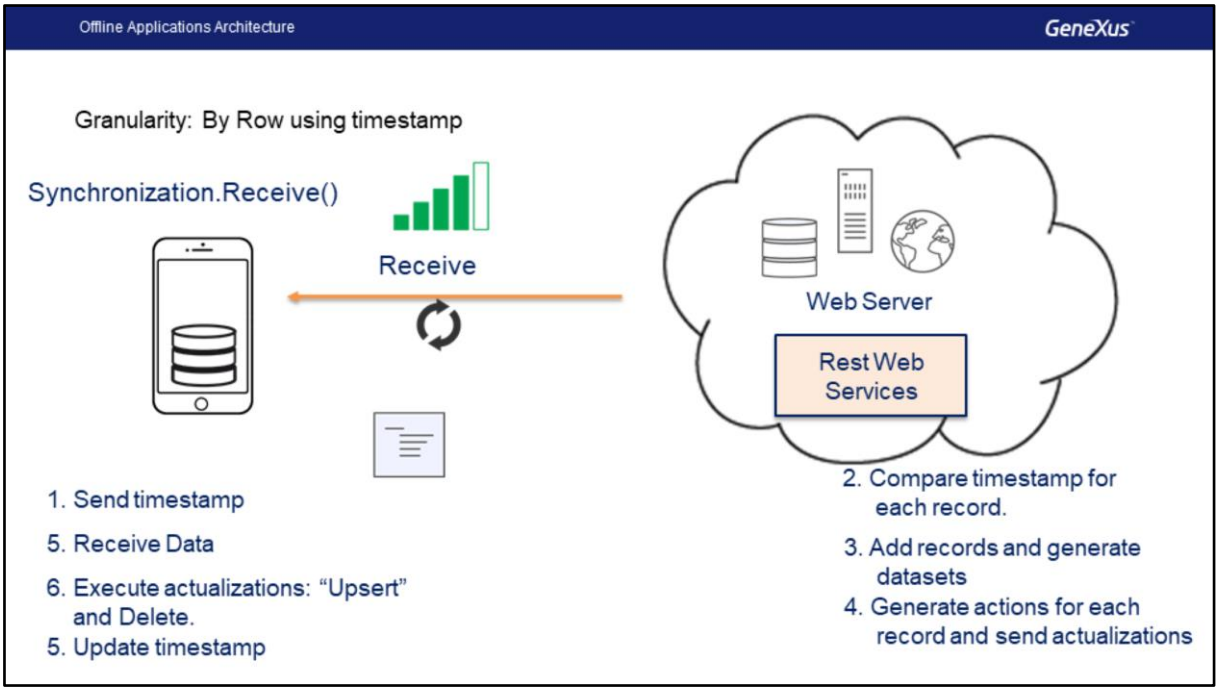
5. O servidor envia as listas com os novos dados para o dispositivo.
6. O dispositivo então recebe os dados e salva o hash de cada tabela.
7. Finalmente, suas listas são processadas em ordem. Para novos registros, um INSERT é feito no banco de dados e, se falhar por causa de chave duplicada, é feito um UPDATE. Para os registros modificados, é feito um UPDATE, e se o registro não existir, é feito um INSERT. É feito um DELETE para os registros excluídos.



Vamos ver o mecanismo By Row usando timestamp.

Esse mecanismo usa uma abordagem diferente, a sincronização ainda é feita por registro, mas os hashes não serão usados para determinar se é uma atualização ou não, em vez disso, serão usadas a data de atualização e a marca de eliminação lógica.

Por essa razão, para usar este mecanismo, devemos indicar para cada nível de uma transação qual atributo irá conter a exclusão lógica e qual irá conter a data e a hora da última modificação.



Usando esse mecanismo, a primeira vez que o servidor for sincronizado, enviará todos os dados que atendem às condições, juntamente com o registro de data e hora da sincronização, o dispositivo armazenará o timestamp que será usado nas sincronizações posteriores.

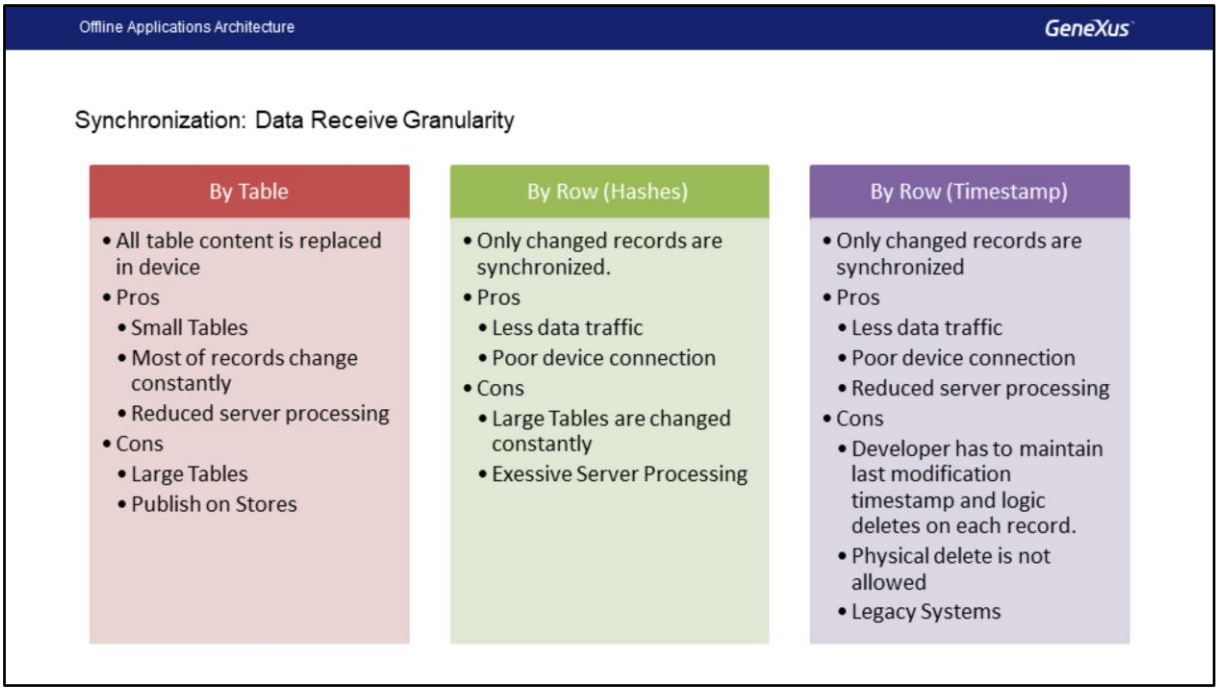
Então, antes de cada nova sincronização.

1. O dispositivo envia o timestamp da última sincronização.
2. O servidor compara o timestamp recebido do dispositivo com o de cada registro usando o atributo de cada tabela.
3. Todos os registros que foram adicionados ou inseridos após o registro de data e hora do dispositivo são adicionados a uma lista.
4. Para determinar a ação a ser realizada em cada registro, será avaliado se o registro foi excluido, usando o atributo com a flag de eliminação lógica, se afirmativo esse registro é marcado como uma exclusão que deve ser enviada para o dispositivo, o restante dos registros é marcado como atualização, o dispositivo irá realizar um "upsert", ou seja, vai atualizar se o registro existir, caso contrário ele é inserido. Toda esta informação é enviada para o dispositivo.
5. O dispositivo recebe os dados
6. Executa operações
7. E atualiza o timestamp.

As desvantagens desse mecanismo é que o desenvolvedor é responsável por manter o registro de data e hora e a marca de eliminação lógica, e não podemos usar a exclusão física de registros nessas tabelas.

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

Com isso terminamos o tópico, no próximo vídeo estudaremos o objeto Offline Database e configuraremos nosso aplicativo para trabalhar off-line.



Vamos rever os mecanismos de sincronização que acabamos de estudar.
Em relação à sincronização By Table, o que acontecerá é que quando o servidor identifica que uma tabela foi modificada, essa tabela será enviada para o dispositivo e lá será substituída completamente. Para determinar as tabelas, será usado um hash para cada tabela e para cada dispositivo.
Quando esse mecanismo é útil: por exemplo, quando nossas tabelas são pequenas ou quando a maioria dos registros muda constantemente, é preferível fazer dessa maneira. Por exemplo, poderíamos usar em um aplicativo mobile interno onde os vendedores têm uma lista de clientes para visitar e a lista muda a cada dia, hoje é uma lista e amanhã é outra completamente diferente.
Qual é a desvantagem desta opção? Quando as tabelas são muito grandes, porque o tráfego de dados será considerável, também não é aconselhável nos casos em que a aplicação será em uso geral e externo, quando são publicados na loja. Se toda vez que for sincronizar, todos os dados forem enviados, a experiência do usuário não será muito boa.

Bem, para evitar esses casos é que temos a sincronização por registros.
Aqui temos duas opções, usando hashes ou timestamp.

Vamos ver a primeira opção.
Nesse caso, o servidor determinará, de acordo com os hashes que serão calculados para

cada conjunto de dados e para cada registro, quais devem ser enviados para cada dispositivo. Então, apenas as alterações serão enviadas, apenas os registros que foram modificados.
A vantagem é que o tráfego dos dados é consideravelmente reduzido, a menos que um grande volume de dados seja constantemente modificado.
Voltando ao exemplo anterior, em vez de recebermos toda a tabela de clientes, receberemos apenas aqueles que sofreram alguma modificação.
Suponha que só iremos receber clientes ativos, então, o dispositivo tem o hash da última sincronização, e quandoquer sincronizar envia esse hash para o servidor, que recalcula um hash para os clientes ativos e compara-o com o que foi enviado pelo dispositivo. Se forem diferentes, o servidor irá gerar um hash para cada registro dessa consulta e irá compará-los com os anteriores, assim poderá determinar exatamente o que foi adicionado, modificado ou excluído.
Esta é a opção padrão ao criar um aplicativo off-line.

Esse método também é útil quando a conexão não é muito boa, pois temos menos tráfego. Como uma desvantagem, teremos que exigir muito mais processamento no lado do servidor e não seria aconselhável no caso de termos grandes volumes de dados que são constantemente modificados.

Vamos supôr que neste caso, temos em nosso sistema uma tabela com milhares de produtos que queremos alimentar em cada dispositivo, também temos centenas de usuários. Sincronizando por Tabela não seria adequado, uma vez que é uma tabela com uma grande quantidade de dados, e sincronizar por registo utilizando hash pode não ser a solução, já que para poder determinar quais registos devem ser enviados para cada dispositivo, é preciso calcular e comparar o hash de cada um dos milhares de produtos que temos. Isso somado à concorrência de muitos usuários simultâneos, pode gerar um problema de processamento do lado do servidor.
Então nenhuma opção é ótima, vamos ver então a terceira opção, que é usar timestamp.

O que conseguimos, utilizando timestamp, é manter a transferência de dados ao mínimo, apenas os registros modificados, mas, ao mesmo tempo envolvendo menos processamento do lado do servidor, uma vez que já não tem de calcular os hashes para cada consulta, mas é possível determinar quais foram, pela data e hora da última modificação e pelo flag de eliminação lógica.
Aqui precisamos manter esses dois atributos adicionais, a flag de eliminação lógica e a data e hora da última modificação, o bom é que muitos sistemas já implementam esses dados em cada registro.
A desvantagem desse mecanismo é que é responsabilidade do desenvolvedor manter esses dois atributos, algo que pode ser complexo quando há muitos sistemas diferentes envolvidos.

Também podemos ter uma mistura entre os dois últimos mecanismos, se usarmos By Row e em uma transação não configurarmos os atributos de data e hora e a eliminação lógica, usando hashes.

Com isso terminamos o tópico, no próximo vídeo estudaremos o objeto Offline Database e configuraremos nosso aplicativo para trabalhar off-line.


GeneXus
GeneXus™
Videos
training.genexus.com
Documentation
wiki.genexus.com
Certificactions
training.genexus.com/certifications