

TVEDT Monica

**Offline Sync**

Azure Mobile Apps

How offline synchronization works

Overview of offline sync Offline sync allows end users to interact with a mobile app - viewing, adding, or modifying data - even when there is no network connection. Changes are stored in a local database. Once the device is back online, these changes are synced with the remote backend.

# How offline synchronization works

When using sync tables, your client code controls when local changes are synchronized with an Azure Mobile App backend. Nothing is sent to the backend until there is a call to push local changes. Similarly, the local store is populated with new data only when there is a call to pull data.

● **Push**: Push is an operation on the sync context and sends all CUD changes since the last push. Note that it is not possible to send only an individual table's changes, because otherwise operations could be sent out of order. Push executes a series of REST calls to your Azure Mobile App backend, which in turn modifies your server database.

● **Pull**: Pull is performed on a per-table basis and can be customized with a query to retrieve only a subset of the server data. The Azure Mobile client SDKs then insert the resulting data into the local store.

● **Implicit Pushes**: If a pull is executed against a table that has pending local updates, the pull first executes a push() on the sync context. This push helps minimize conflicts between changes that are already queued and new data from the server.

● **Incremental Sync**: the first parameter to the pull operation is a query name that is used only on the client. If you use a non-null query name, the Azure Mobile SDK performs an incremental sync. Each time a pull operation returns a set of results, the latest updatedAt timestamp from that result set is stored in the SDK local system tables. Subsequent pull operations retrieve only records after that timestamp.

● To use incremental sync, your server must return meaningful updatedAt values and must also support sorting by this field. However, since the SDK adds its own sort on the updatedAt field, you cannot use a pull query that has its own orderBy clause.

● The query name can be any string you choose, but it must be unique for each logical query in your app. Otherwise, different pull operations could overwrite the same incremental sync timestamp and your queries can return incorrect results.

● If the query has a parameter, one way to create a unique query name is to incorporate the parameter value. For instance, if you are filtering on userid, your query name could be as follows (in C#): await todoTable.PullAsync("todoItems" + userid, syncTable.Where(u => u.UserId == userid));

● If you want to opt out of incremental sync, pass null as the query ID. In this case, all records are retrieved on every call to PullAsync, which is potentially inefficient.

● **Purging**: You can clear the contents of the local store using IMobileServiceSyncTable.PurgeAsync. Purging may be necessary if you have stale data in the client database, or if you wish to discard all pending changes.

● A purge clears a table from the local store. If there are operations awaiting synchronization with the server database, the purge throws an exception unless the force purge parameter is set.

● As an example of stale data on the client, suppose in the “todo list” example, Device1 only pulls items that are not completed. A todoitem "Buy milk" is marked completed on the server by another device. However, Device1 still has the “Buy milk” todoitem in local store because it is only pulling items that are not marked complete. A purge clears this stale item.

Enabling offline sync for your app

# Updating the client app to support offline features

Azure Mobile App offline features allow you to interact with a local database when you are in an offline scenario. To use these features in your app, you initialize a SyncContext to a local store. Then reference your table through the IMobileServiceSyncTable interface. SQLite is used as the local store on the device.

1. Install the **SQLite runtime for the Universal Windows Platform11**.

2. In Visual Studio, open the NuGet package manager for the UWP app project that you completed in the Create a Windows app tutorial. Search for and install the Microsoft.Azure.Mobile.Client.SQLiteStore NuGet package.

3. In Solution Explorer, right-click References > Add Reference&hellip; > Universal Windows > Extensions, then enable both SQLite for Universal Windows Platform and Visual C++ 2015 Runtime for Universal Windows Platform apps.

4. Open the MainPage.xaml.cs file and uncomment the #define OFFLINE\_SYNC\_ENABLED definition.

5. In Visual Studio, press the F5 key to rebuild and run the client app. The app works the same as it did before you enabled offline sync. However, the local database is now populated with data that can be used in an offline scenario.

# Update the app to disconnect from the backend

In this section, you break the connection to your Mobile App backend to simulate an offline situation. When you add data items, your exception handler tells you that the app is in offline mode. In this state, new items added in the local store and will be synced to the mobile app backend when push is next run in a connected state.

1. Edit App.xaml.cs in the shared project. Comment out the initialization of the MobileServiceClient and add the following line, which uses an invalid mobile app URL: public static MobileServiceClient MobileService = new MobileServiceClient("https://your-service.azurewebsites.fail");

1. You can also demonstrate offline behavior by disabling wifi and cellular networks on the device or use airplane mode.

2. Press F5 to build and run the app. Notice your sync failed on refresh when the app launched.

3. Enter new items and notice that push fails with a CancelledByNetworkError status each time you click Save. However, the new todo items exist in the local store until they can be pushed to the mobile app backend. In a production app, if you suppress these exceptions the client app behaves as if it's still connected to the mobile app backend.

4. Close the app and restart it to verify that the new items you created are persisted to the local store.

5. (Optional) In Visual Studio, open Server Explorer. Navigate to your database in Azure->SQL Databases. Right-click your database and select Open in SQL Server Object Explorer. Now you can browse to your SQL database table and its contents. Verify that the data in the backend database has not changed.

6. (Optional) Use a REST tool such as Fiddler or Postman to query your mobile backend, using a GET query in the form https://<your-mobile-app-backend-name>.azurewebsites.net/ tables/TodoItem.

# Update the app to reconnect to the backend

In this section, you reconnect the app to the mobile app backend. These changes simulate a network reconnection on the app.

When you first run the application, the OnNavigatedTo event handler calls InitLocalStoreAsync. This method in turn calls SyncAsync to sync your local store with the backend database. The app attempts to sync on startup.

1. Open App.xaml.cs in the shared project, and uncomment your previous initialization of MobileServiceClient to use the correct the mobile app URL.

2. Press the F5 key to rebuild and run the app. The app syncs your local changes with the Azure Mobile App backend using push and pull operations when the OnNavigatedTo event handler executes.

3. (Optional) View the updated data using either SQL Server Object Explorer or a REST tool like Fiddler. Notice the data has been synchronized between the Azure Mobile App backend database and the local store.

4. In the app, click the check box beside a few items to complete them in the local store.

UpdateCheckedTodoItem calls SyncAsync to sync each completed item with the Mobile App backend. SyncAsync calls both push and pull. However, **whenever you execute a pull against a table that the client has made changes to, a push is always executed automatically**. This behavior ensures all tables in the local store along with relationships remain consistent. This behavior may result in an unexpected push.

# API summary

To support the offline features of mobile services, we used the IMobileServiceSyncTable interface and initialized MobileServiceClient.SyncContext with a local SQLite database. When offline, the normal CRUD operations for Mobile Apps work as if the app is still connected while the operations occur against the local store. The following methods are used to synchronize the local store with the server:

● **PushAsync**: Because this method is a member of IMobileServicesSyncContext, changes across all tables are pushed to the backend. Only records with local changes are sent to the server.

● **PullAsync**: A pull is started from a IMobileServiceSyncTable. When there are tracked changes in the table, an implicit push is run to make sure that all tables in the local store along with relationships remain consistent. The pushOtherTables parameter controls whether other tables in the context are pushed in an implicit push. The query parameter takes an IMobileServiceTableQuery or OData query string to filter the returned data. The queryId parameter is used to define incremental sync.

● **PurgeAsync**: Your app should periodically call this method to purge stale data from the local store. Use the force parameter when you need to purge any changes that have not yet been synced.