hs.data.tablesync

Datei	`hs.data.tablesync.pas`
Typ	Unit
Version	3.2.2
Interfaces	`IhsDataTablesyncRow`, `IhsDataTablesync`
Klassen	`EhsDataTablesyncError`
Geändert	2026-10-29

Synchronisiert Datensätze aus einem TDataset oder einem JSON-Array (ISuperObject) in eine Tabelle einer SQLite-Datenbank (Zugriff über UniDAC), ausgeführt in einem Hintergrundthread. Quellfelder werden über konfigurierbare Zuordnungen (Quellfeld -> Zielspalte) sowie ein oder mehrere Schlüsselpaare auf die Zieltabelle abgebildet. Anhand der Schlüsselpaare wird je Datensatz per Exists-Check entschieden, ob ein INSERT oder ein UPDATE ausgeführt wird (kein SQLite-UPSERT/ON CONFLICT, da die in diesem Projekt eingesetzte UniDAC/SQLite-Engine dieses Sprachfeature nicht unterstützt).

Verwendung

Einsetzen, wenn Daten aus einer beliebigen Quelle (DB-Abfrage als TDataset oder importiertes JSON-Array als ISuperObject) mit einer lokalen SQLite-Tabelle abgeglichen werden sollen, ohne die Programmoberfläche während des Schreibvorgangs zu blockieren. Über OnProgress kann der Fortschritt (z. B. in einer ProgressBar) angezeigt werden, über OnComplete wird der Abschluss inkl. Ergebnisstatistik gemeldet. Beide Events werden bereits im Kontext des Hauptthreads aufgerufen (TThread.Queue) und können daher direkt auf die UI zugreifen. SetOnProgress/SetOnComplete sind überladen und akzeptieren wahlweise eine gebundene Methode (mit „of object“) oder eine anonyme Prozedur (reference to procedure) - beide werden auf demselben Weg per TThread.Queue im Hauptthread ausgeführt, ein anonymer Handler ist also genauso UI-sicher wie ein Methodenzeiger. Wichtig bei anonymen Handlern: Da der Aufruf erst asynchron nach dem Queuing im Hauptthread erfolgt, darf ein darin per Closure erfasstes Objekt (z. B. ein Formular) nicht vorher zerstört werden - vor dem Freigeben des Besitzers ggf. WaitFor/Cancel aufrufen.

Über `SetRunInBackground(False)` laeuft die Synchronisation stattdessen synchron im Aufrufer-Thread ab - Execute blockiert dann, bis alle Zeilen verarbeitet sind, es wird kein TThread erzeugt und OnProgress/OnComplete werden direkt (ohne TThread.Queue) im selben Thread aufgerufen. Sinnvoll für kleine Datenmengen oder wenn der Aufrufer selbst schon in einem Hintergrundthread läuft. Cancel/WaitFor haben in diesem Modus keine Wirkung (es gibt keinen separaten Thread zum Abbrechen/Abwarten). Standardmäßig (True) läuft die Synchronisation im Hintergrundthread wie oben beschrieben.

Über `SetUseOmniThreadLibrary(True)` wird im Hintergrundthread-Modus statt eines klassischen TThread ein OmniThreadLibrary-Task (CreateTask/IOmniTaskControl) verwendet. Beide Implementierungen delegieren dieselbe Datenbankarbeit an `ThsDataTablesyncEngine` und marshalen OnProgress/OnComplete auf demselben Weg (TThread.Queue) in den Hauptthread - nach außen verhält sich beides identisch, es ist lediglich eine alternative Thread-Grundlage zum Ausprobieren, falls der klassische TThread-Pfad im konkreten Projekt/Prozess nicht zuverlässig anläuft. Wirkt nur, wenn `SetRunInBackground(True)` (Standard) aktiv ist.

Über `SetInsertNewRecords(False)` werden ausschließlich bereits vorhandene Datensätze (Schlüssel in der Zieltabelle gefunden) aktualisiert; Datensätze ohne passenden Schlüssel werden übersprungen und in OnComplete als mSkipped gezählt (nicht als mInserted oder mFailed). Standard ist True (fehlende Datensätze werden angelegt).

Über `SetDeleteOrphans(True)` werden nach dem Abgleich verwaiste Datensätze gelöscht, d. h. Zeilen der Zieltabelle, deren Schlüssel in der Quelle nicht (mehr) vorkommt; die Anzahl steht in OnComplete als mDeleted. Standard ist False. ACHTUNG, das Löschen ist endgültig: Die Quelle muss den kompletten Bestand der Zieltabelle abbilden - ein gefiltertes/teilweises Dataset oder Zeilen, die aus anderen Quellen in die Zieltabelle gelangt sind, würden sonst gelöscht. Gelöscht wird nur, wenn der Lauf nicht abgebrochen wurde, die Verbindung stand und die Quelle mindestens eine Zeile enthält (eine leere Quelle löscht nie die ganze Tabelle). Datensätze, die beim Schreiben als Failed gezählt wurden, bleiben erhalten, da ihr Schlüssel in der Quelle vorkommt. Der Schlüsselvergleich läuft in SQLite (TEMP-Tabelle mit den Spalten-Affinitäten der Zieltabelle, NULL-sicher) und damit genau wie der Exists-Check beim Abgleich. Schlägt das Löschen fehl, wird nichts gelöscht (eine Transaktion, inkl. Protokolleinträge) und mSuccess ist False mit „Verwaiste Datensätze: …“. Ist ein Änderungsprotokoll aktiv (SetChangeLogTable), wird pro gelöschter Zeile ein Eintrag geschrieben: column_name = „(Datensatz gelöscht)“, old_value = alle Wertespalten als „spalte=wert; spalte=wert“ (der Schlüssel steht in record_key), new_value = NULL.

Über `SetChangeLogTable('tabellenname')` wird ein Änderungsprotokoll in dieser Tabelle derselben SQLite-Datenbank geführt (wird bei Bedarf angelegt; leer = kein Protokoll, Standard). Protokolliert werden UPDATEs (pro geänderter Spalte ein Eintrag) und mit SetDeleteOrphans gelöschte Zeilen (siehe dort). Ein UPDATE-Eintrag enthält changed_at, table_name, record_key (z. B. „customer_id=5“), column_name, old_value und new_value (NULL, wenn der Wert geleert wurde). Kein Eintrag entsteht, wenn ein leeres Feld (NULL oder Leerstring) einen Wert bekommt oder wenn der Wert unverändert bleibt. UPDATE und zugehörige Protokolleinträge einer Zeile laufen in einer Transaktion. Vergleich: Zahlen numerisch, Datumswerte als „yyyy-mm-dd hh:nn:ss“, alles andere als Text (Groß-/Kleinschreibung zählt). Nicht protokolliert werden Anpassungen, die ein OnRowProcessed-Handler (siehe unten) nachträglich vornimmt.

Über `SetOnRowProcessed(procedure(const mRow: IhsDataTablesyncRow) ...)` wird nach jeder eingefügten oder geänderten Zeile ein Handler aufgerufen, mit dem sich diese Zeile anpassen lässt: mRow.GetAction (raInserted/raUpdated), `mRow.GetValue('spalte')` liefert den Quellwert der gemappten Zielspalte (bzw. den zuvor per SetValue gesetzten Wert), `mRow.SetValue('spalte', Wert)` merkt eine Änderung vor (auch für nicht gemappte Spalten der Zieltabelle, z. B. einen Zeitstempel). Nach Rückkehr des Handlers werden alle vorgemerkten Änderungen in einem UPDATE auf diese Zeile geschrieben. Schlüsselspalten sind nicht änderbar (SetValue löst EhsDataTablesyncError aus). WICHTIG: Der Handler läuft im Kontext der Datenbankarbeit - also im Hintergrundthread (bzw. im Aufrufer-Thread bei `SetRunInBackground(False)`), NICHT im Hauptthread. Kein direkter UI-Zugriff; dafür TThread.Queue verwenden. Wirft der Handler eine Exception, zählt der Datensatz als Failed (die Zeile selbst ist zu diesem Zeitpunkt bereits geschrieben, ihre Anpassung entfällt).

Boolean-Werte: SQLite kennt keinen eigenen Boolean-Typ. Ein Boolean-Quellwert (JSON true/false bzw. ein Boolean-Feld im TDataset) wird deshalb beim Einlesen immer als Integer geschrieben: True = 1, False = 0. Damit genügt z. B. `AddFieldMapping('accountEnabled', 'eid_active')` mit einer Integer-Zielspalte eid_active; eine eigene Konfiguration ist nicht nötig. Die Umwandlung gilt für alle Mappings (auch Schlüsselpaare) und wirkt sich überall gleich aus: INSERT/UPDATE, Änderungsprotokoll (alter/neuer Wert als 1/0) und mRow.GetValue im OnRowProcessed- Handler liefern ebenfalls 1/0. Eine Textspalte erhielte entsprechend '1'/'0' statt 'True'/'False'.

Hinweis: Die Klasse öffnet und schließt pro Lauf eine eigene TUniConnection auf die angegebene Datenbankdatei und aktiviert dafür einen BusyTimeout (5000 ms). Das ist wichtig, weil eine zweite, gleichzeitig offene TUniConnection auf dieselbe Datei (z. B. eine dauerhaft verbundene Connection im Hauptthread, die ein Grid oder das Quell-Dataset speist) zu Sperrkonflikten führen kann. Der BusyTimeout sorgt dafür, dass ein solcher Konflikt spätestens nach der angegebenen Zeit mit einer regulären, abgefangenen Exception endet (der betroffene Datensatz zählt dann als Failed) - statt dass der Aufruf ohne Timeout unbegrenzt (synchron!) blockiert, was sich als Einfrieren der gesamten Programmoberfläche äußert. Ausdrücklich NICHT verwenden: JournalMode=WAL. Das Umschalten in den WAL-Modus verlangt kurzfristig exklusiven Zugriff auf die Datei; ist zu diesem Zeitpunkt bereits eine zweite Connection (z. B. im Hauptthread) auf dieselbe Datei offen, kann der Umschaltversuch selbst unbegrenzt blockieren - in der Praxis sogar zuverlässiger und früher als das ursprüngliche Problem (beobachtet: Hängenbleiben bereits beim ersten Datensatz, statt erst nach 50-100). Ist eine WAL-taugliche Nebenläufigkeit gewünscht, muss die Datenbankdatei stattdessen einmalig (z. B. beim Programmstart, wenn noch keine zweite Connection offen ist) in den WAL-Modus versetzt werden - dieser Zustand ist dauerhaft in der Datei gespeichert und gilt dann für alle künftigen Connections, ohne dass hier erneut umgeschaltet wird.

Beispiel

```
// Quelle: TDataset
var
  lSync: IhsDataTablesync;
begin
  lSync := hsDataTablesync
    .SetDatabaseFile('C:\Data\cache.db')
    .SetTableName('customers')
    .AddKeyMapping('CustomerID', 'customer_id')
    .AddFieldMapping('Name', 'name')
    .AddFieldMapping('City', 'city')
    .SetSourceDataset(lCustomerQuery)
    .SetProgressInterval(100)
    .SetOnProgress(HandleSyncProgress)
    .SetOnComplete(HandleSyncComplete);
  lSync.Execute;
end;

// Quelle: ISuperObject (JSON-Array)
var
  lSync: IhsDataTablesync;
  lJsonArray: ISuperObject;
begin
  lJsonArray := SO('[{"id":1,"name":"Test"}]');
  lSync := hsDataTablesync
    .SetDatabaseFile('C:\Data\cache.db')
    .SetTableName('items')
    .AddKeyMapping('id', 'item_id')
    .AddFieldMapping('name', 'name')
    .SetSourceJsonArray(lJsonArray)
    .SetOnComplete(HandleSyncComplete);
```

```
  lSync.Execute;
end;

// Event-Handler (im Hauptthread aufgerufen):
procedure TForm1.HandleSyncProgress(mCurrent, mTotal: Integer);
begin
  ProgressBar1.Max := mTotal;
  ProgressBar1.Position := mCurrent;
end;

procedure TForm1.HandleSyncComplete(mSuccess: Boolean; mInserted, mUpdated,
mFailed, mSkipped, mDeleted: Integer;
  const mErrorMessage: string);
begin
  if mSuccess then
    StatusBar1.SimpleText := Format('%d neu, %d aktualisiert, %d
fehlgeschlagen, %d übersprungen, %d gelöscht',
      [mInserted, mUpdated, mFailed, mSkipped, mDeleted])
  else
    StatusBar1.SimpleText := 'Fehler: ' + mErrorMessage;
end;

// Alternativ mit anonymer Prozedur statt gebundener Methode:
var
  lSync2: IhsDataTablesync;
begin
  lSync2 := hsDataTablesync
    .SetDatabaseFile('C:\Data\cache.db')
    .SetTableName('customers')
    .AddKeyMapping('CustomerID', 'customer_id')
    .AddFieldMapping('Name', 'name')
    .SetSourceDataset(lCustomerQuery)
    .SetOnProgress(
      procedure(mCurrent, mTotal: Integer)
      begin
        ProgressBar1.Max := mTotal;
        ProgressBar1.Position := mCurrent;
      end)
    .SetOnComplete(
      procedure(mSuccess: Boolean; mInserted, mUpdated, mFailed, mSkipped,
mDeleted: Integer; const mErrorMessage: string)
      begin
        if mSuccess then
          StatusBar1.SimpleText := Format('%d neu, %d aktualisiert',
[mInserted, mUpdated])
        else
          StatusBar1.SimpleText := 'Fehler: ' + mErrorMessage;
      end);
  lSync2.Execute;
end;
```

```
// Synchron im Hauptthread statt im Hintergrundthread (Execute blockiert):
var
  lSync3: IhsDataTablesync;
begin
  lSync3 := hsDataTablesync
    .SetDatabaseFile('C:\Data\cache.db')
    .SetTableName('customers')
    .AddKeyMapping('CustomerID', 'customer_id')
    .AddFieldMapping('Name', 'name')
    .SetSourceDataset(lCustomerQuery)
    .SetRunInBackground(False)
    .SetOnComplete(HandleSyncComplete);
  lSync3.Execute; // blockiert bis fertig, HandleSyncComplete wird direkt
aufgerufen
end;

// Hintergrundthread ueber OmniThreadLibrary statt klassischem TThread:
var
  lSync4: IhsDataTablesync;
begin
  lSync4 := hsDataTablesync
    .SetDatabaseFile('C:\Data\cache.db')
    .SetTableName('customers')
    .AddKeyMapping('CustomerID', 'customer_id')
    .AddFieldMapping('Name', 'name')
    .SetSourceDataset(lCustomerQuery)
    .SetUseOmniThreadLibrary(True)
    .SetOnComplete(HandleSyncComplete);
  lSync4.Execute;
end;

// Nur vorhandene Datensätze aktualisieren, keine neuen anlegen:
var
  lSync5: IhsDataTablesync;
begin
  lSync5 := hsDataTablesync
    .SetDatabaseFile('C:\Data\cache.db')
    .SetTableName('customers')
    .AddKeyMapping('CustomerID', 'customer_id')
    .AddFieldMapping('Name', 'name')
    .SetSourceDataset(lCustomerQuery)
    .SetInsertNewRecords(False)
    .SetOnComplete(HandleSyncComplete);
  lSync5.Execute;
end;

// Boolean-Quellfeld (JSON true/false) in eine Integer-Spalte (1/0)
übernehmen:
var
  lSync9: IhsDataTablesync;
```

```
begin
  lSync9 := hsDataTablesync
    .SetDatabaseFile('C:\Data\cache.db')
    .SetTableName('entra_users')
    .AddKeyMapping('id', 'eid_id')
    .AddFieldMapping('userPrincipalName', 'eid_upn')
    .AddFieldMapping('accountEnabled', 'eid_active') // eid_active INTEGER:
1 = true, 0 = false
    .SetSourceJsonArray(lJsonArray);
  lSync9.Execute;
end;

// Verwaiste Datensätze (Schlüssel nicht mehr in der Quelle) löschen:
var
  lSync8: IhsDataTablesync;
begin
  lSync8 := hsDataTablesync
    .SetDatabaseFile('C:\Data\cache.db')
    .SetTableName('customers')
    .AddKeyMapping('CustomerID', 'customer_id')
    .AddFieldMapping('Name', 'name')
    .SetSourceDataset(lCustomerQuery) // muss den kompletten Bestand
enthalten!
    .SetDeleteOrphans(True);
  lSync8.Execute;
end;

// Änderungen bestehender Werte protokollieren:
var
  lSync7: IhsDataTablesync;
begin
  lSync7 := hsDataTablesync
    .SetDatabaseFile('C:\Data\cache.db')
    .SetTableName('customers')
    .AddKeyMapping('CustomerID', 'customer_id')
    .AddFieldMapping('Name', 'name')
    .SetSourceDataset(lCustomerQuery)
    .SetChangeLogTable('customers_changelog');
  lSync7.Execute;
end;

// Jede geschriebene Zeile nachträglich anpassen (läuft im
Hintergrundthread!):
var
  lSync6: IhsDataTablesync;
begin
  lSync6 := hsDataTablesync
    .SetDatabaseFile('C:\Data\cache.db')
    .SetTableName('customers')
    .AddKeyMapping('CustomerID', 'customer_id')
```

```
    .AddFieldMapping('Name', 'name')
    .SetSourceDataset(lCustomerQuery)
    .SetOnRowProcessed(
      procedure(const mRow: IhsDataTablesyncRow)
      begin
        mRow.SetValue('name', UpperCase(VarToStr(mRow.GetValue('name'))))
            .SetValue('synced_at', Now);
      end);
  lSync6.Execute;
end;
```

Schnittstelle

Interfaces

- `IhsDataTablesyncRow`
- `IhsDataTablesync`

Klassen

- `EhsDataTablesyncError`

Routinen

- `function hsDataTablesync: IhsDataTablesync`

Verwendete Units

`System.SysUtils`, `Data.DB`, `SuperObject`

Änderungshistorie

Datum	Version	Änderung
2026-09-18	1.0.0	Erstellt
2026-09-18	1.1.0	SetOnProgress/SetOnComplete um Overloads für anonyme Prozeduren (reference to procedure) erweitert
2026-09-19	1.2.0	SetRunInBackground hinzugefügt: Synchronisation kann wahlweise im Hintergrundthread (Standard) oder synchron im Aufrufer-Thread laufen; die Datenbankarbeit wurde dafür in eine gemeinsame, threadunabhängige `ThsDataTablesyncEngine` ausgelagert
2026-09-19	1.2.1	Bugfix: `ThsDataTablesyncThread` startete bisher per inherited `Create(False)` als erste Anweisung im Konstruktor, wodurch Execute im Hintergrundthread parallel zum restlichen Konstruktor-Code laufen konnte und dabei z. B. auf ein noch nicht zugewiesenes FEngine traf (Race Condition, führte zu stillem Abbruch ohne OnComplete)

Datum	Version	Änderung
2026-09-19	1.2.2	Bugfix-Korrektur: Der erste Fix (`Create(True) + Start`) löste auf dieser Delphi-Version die Laufzeitfehlermeldung „Start kann für einen laufenden oder unterbrochenen Thread nicht aufgerufen werden“ aus. Stattdessen werden nun alle Felder zugewiesen und erst danach inherited `Create(False)` als letzte Anweisung im Konstruktor aufgerufen - kein Start/Suspended mehr nötig
2026-09-19	1.3.0	SetUseOmniThreadLibrary hinzugefügt: alternative Hintergrundausführung über einen OmniThreadLibrary-Task (`ThsDataTablesyncOmniThread`) statt des klassischen `ThsDataTablesyncThread` (TThread) - beide bleiben nebeneinander bestehen und lassen sich einzeln per SetRunInBackground/ SetUseOmniThreadLibrary auswählen; zusätzlich CodeSite- Diagnosetraces beim Start beider Thread-Varianten ergänzt, um zu erkennen, ob der jeweilige Thread-Body überhaupt erreicht wird
2026-09-20	1.3.1	Bugfix: JournalMode=WAL und BusyTimeout waren auskommentiert (nur LockingMode=lmNormal aktiv) - dadurch konnte die eigene TUniConnection der Engine mit einer parallel im Hauptthread offenen TUniConnection auf dieselbe SQLite-Datei kollidieren und dabei unbegrenzt (ohne Timeout, synchron) blockieren. Das äußerte sich als: Synchronisation verarbeitet je nach Lauf eine unterschiedliche Anzahl an Zeilen, danach friert die komplette Programmoberfläche ein. WAL+BusyTimeout wieder aktiviert
2026-09-20	1.3.2	Bugfix-Korrektur: JournalMode=WAL wieder entfernt - das Umschalten in den WAL-Modus verlangt kurzfristig exklusiven Zugriff und blockierte in diesem Projekt zuverlässig (bei jedem Lauf, schon beim ersten Datensatz), da parallel eine zweite TUniConnection auf dieselbe Datei offen ist. BusyTimeout bleibt aktiv - wandelt Sperrkonflikte in eine abgefangene Exception nach spätestens 5s um, statt unbegrenzt zu blockieren
2026-09-24	1.4.0	SetInsertNewRecords hinzugefügt (Standard True): bei False werden nur vorhandene Datensätze aktualisiert, Datensätze ohne Schlüsseltreffer werden übersprungen
2026-09-24	2.0.0	Breaking Change: OnComplete (Event und Proc) erhält zusätzlich den Parameter mSkipped (Anzahl übersprungener Datensätze bei `SetInsertNewRecords(False)`), Reihenfolge: mSuccess, mInserted, mUpdated, mFailed, mSkipped, mErrorMessage
2026-09-25	2.1.0	SetOnRowProcessed hinzugefügt: anonymer Handler nach jeder eingefügten/geänderten Zeile (`IhsDataTablesyncRow` mit GetAction, GetValue, SetValue), Änderungen werden per UPDATE geschrieben
2026-09-26	2.2.0	SetChangeLogTable hinzugefügt: Änderungsprotokoll (alter/neuer Wert je geänderter Spalte) für UPDATEs; leere Felder, die einen Wert erhalten, werden nicht protokolliert
2026-10-13	3.0.0	Breaking Change: OnComplete (Event und Proc) erhält zusätzlich mDeleted (nach mSkipped, vor mErrorMessage). SetDeleteOrphans hinzugefügt: löscht Zielzeilen, deren Schlüssel in der Quelle nicht mehr vorkommt
2026-10-13	3.1.0	Gelöschte (verwaiste) Zeilen werden im Änderungsprotokoll erfasst (ein Eintrag je Zeile mit allen alten Werten)
2026-10-15	3.2.0	Boolean-Quellwerte (JSON true/false, Boolean-Feld im TDataset) werden beim Einlesen immer als Integer 1/0 übernommen, sodass sie z. B. per `AddFieldMapping('accountEnabled', 'eid_active')` in eine Integer-Spalte geschrieben werden. Gilt für alle Mappings; eine Textspalte erhält statt 'True'/'False' nun '1'/'0'
2026-10-15	3.2.1	OnComplete: Bei mFailed > 0 enthält mErrorMessage jetzt die Meldung des ersten fehlgeschlagenen Datensatzes („Erster Zeilenfehler (Zeile n: …)“), auch wenn mSuccess True bleibt. Bisher landete sie nur in CodeSite

Datum	Version	Änderung
2026-10-15	3.2.2	Ein AddFieldMapping auf dieselbe Zielspalte wie ein AddKeyMapping (z. B. beide 'userPrincipalName' -> 'upn') wird jetzt ignoriert, da es den Spaltennamen/Parameter doppelt im INSERT/UPDATE erzeugte. Das war die Ursache für „Incorrect parameter.“ bei jeder Zeile (SQLite-Provider), bestätigt

From:
https://insights.hschmid-it.de/ - **Insights - ein Programm von hschmid-it.de**

Permanent link:
https://insights.hschmid-it.de/doku.php?id=delphi:claudeai:hs.data.tablesync2

Last update: **2026/11/10 16:30**