Contents

Omnis Synchronisation Server User Guide

JS Client Wrapper version 1.5.0 & SyncServer version 2.3.0 and later

Omnis Software Ltd

V2.0 February 2017

18-062013-04

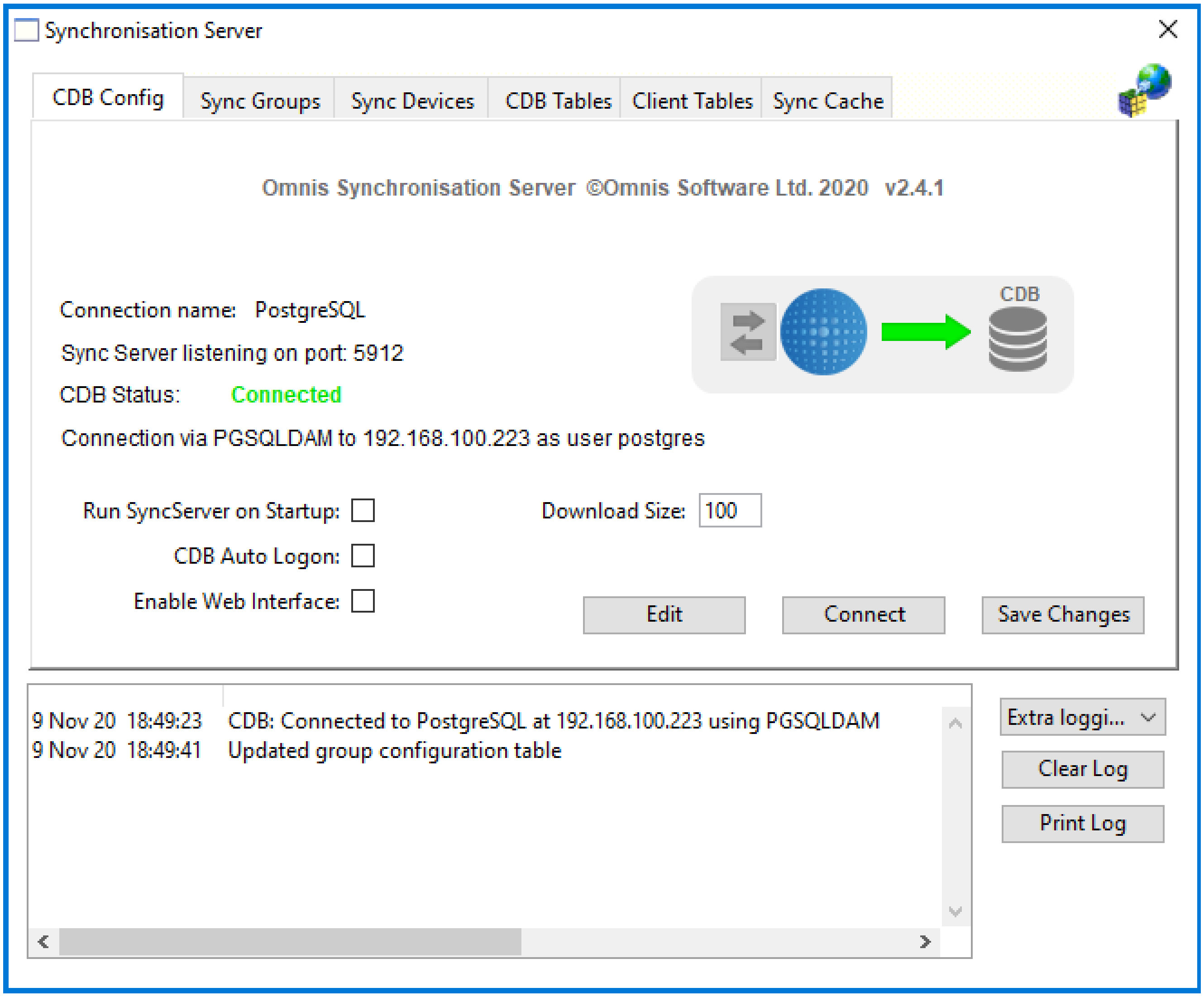
Introduction

The Omnis Synchronisation Server (SyncServer) is designed for use with standalone JavaScript Client applications that are linked with the dbSQLite database library.

It responds to requests from the built-in SQL Object and its two special methods, *$syncinit()* and *$sync()*, providing database synchronisation for iOS, Android and Windows 10 client devices. Database synchronisation is beneficial where devices may have limited or intermittent network connectivity.

This guide will show you how to set up the SyncServer, how to select which database server is used to store the consolidated data (CDB), how to select which tables the client app will use and how data coming from and sent to the SyncServer is handled.

Note: The Omnis Sync Server version 2.3, or above, uses a RESTful interface to communicate with mobile clients. Versions of Omnis Studio prior to Studio 10 may require a Web Services serial number to use the Sync Server in the Development version of Studio 8.0.2 (all Server versions of Omnis include Web Services support). All Development and Server versions of Studio 10, and above, include a Web Services serial number.



Omnis Synchronisation Server ©Omnis Software Ltd. 2020 v2.4.1

Connection name: PostgreSQL

Sync Server listening on port: 5912

CDB Status: Connected

Connection via PGSQLDAM to 192.168.100.223 as user postgres

Run SyncServer on Startup: □

CDB Auto Logon: □

Enable Web Interface: □

Download Size: 100

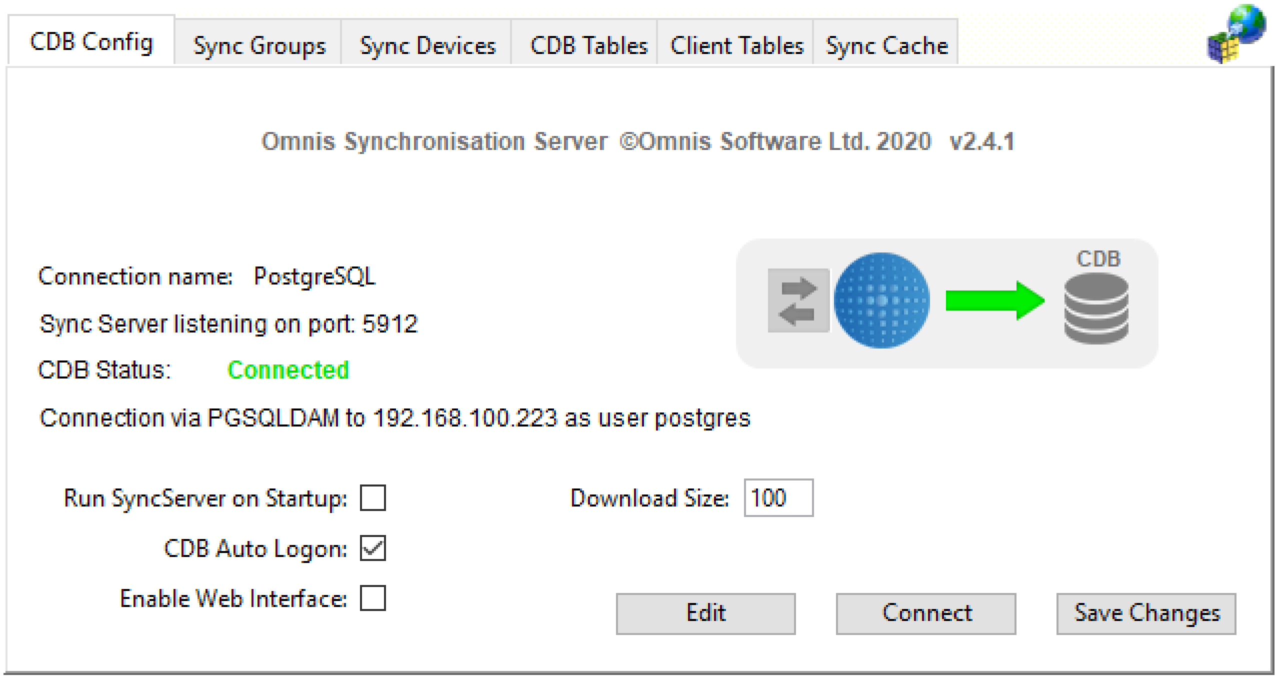
```
9 Nov 20  18:49:23   CDB: Connected to PostgreSQL at 192.168.100.223 using PGSQLDAM
9 Nov 20  18:49:41   Updated group configuration table
```

Figure 1:

Synchronisation Server function

Selecting the Consolidated Database

Connection to the CDB is made by choosing a pre-configured SQL Browser session template from the drop list. If using the Runtime or Server edition of Omnis Studio, it is possible to define and save the database connection parameters here instead. An 'Edit' button will appear when using the Runtime/Server version.



Omnis Synchronisation Server ©Omnis Software Ltd. 2020 v2.4.1

Connection name: PostgreSQL

Sync Server listening on port: 5912

CDB Status: Connected

Connection via PGSQLDAM to 192.168.100.223 as user postgres

Run SyncServer on Startup: □

CDB Auto Logon: ☑

Enable Web Interface: □

Download Size: 100

Figure 2:

When connected to the CDB, the connection diagram displays a green arrow. The two-way arrow indicator briefly illuminates whenever a synchronisation request is received. The log pane will also receive an entry each time a client contacts the SyncServer.

The Omnis $serverport property is displayed along with the CDB connection summary. Clients must be configured to use the same port number in the URL passed to their $syncinit() methods.

The *CDB Auto Logon* option tells the SyncServer to logon automatically next time the library loads.

When selecting the session template or when modifying configuration settings, press *Save Details* to store any changes.

Important note: If you change the CDB Session template to point at a different database, this will invalidate the list of CDB Tables, Client Tables and the Sync Cache information since the existing settings are specific to the existing database. When changing the session template, the synchronisation tables and sync cache will be cleared automatically and CDB Tables will need to be selected for the new session.

Please note that the CDB Config pane also displays the SyncServer version number. This will be required in the event of a technical support enquiry.

The download phase of a $sync() normally involves one or more round trips to the server. The Download Size option governs the number of table rows, inserts, updates and/or deletes that will be sent to a synchronisation client in a single "packet" during synchronisation. Raising this value potentially means sending fewer, larger response packets.

Synchronisation User Groups

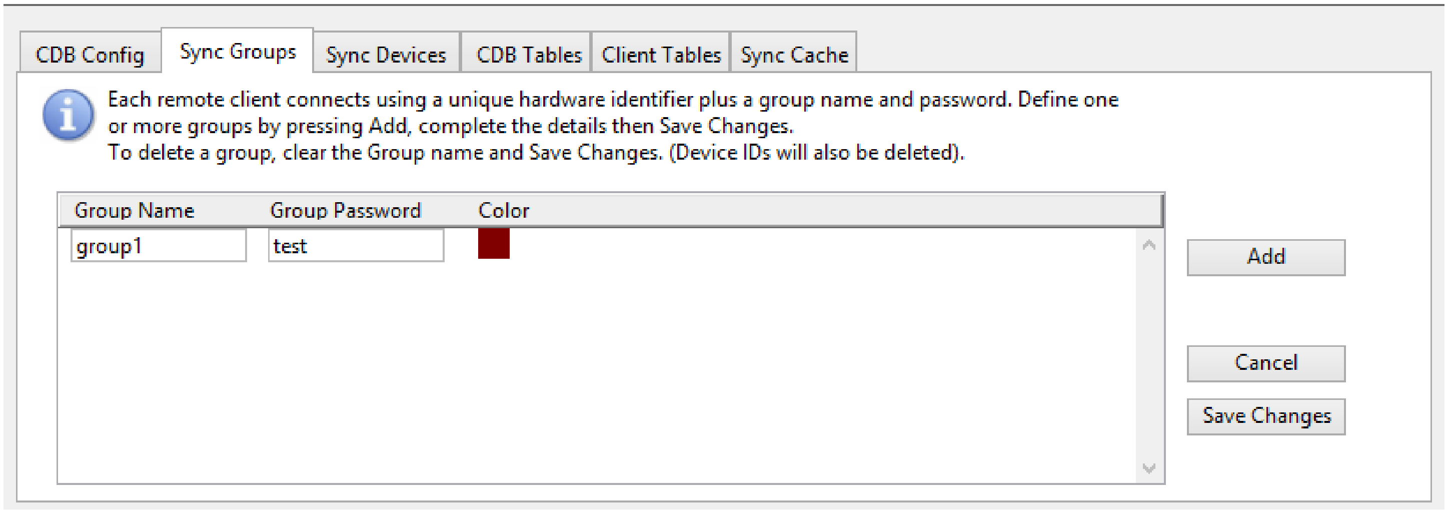
A user group defines a group name and password that clients will subsequently pass during $syncinit() to authenticate with the SyncServer.

Each group can contain different synchronisation types for each database table or can be used to exclude a particular table from the group, in which case it will not be sent to any clients in that group. See *Client Tables* for details.

A user group can support up to 1023 client devices and there can be up to 255 separate groups.

The color box displayed next to each group entry is used to chroma-code log entries for requests received from clients in that group.

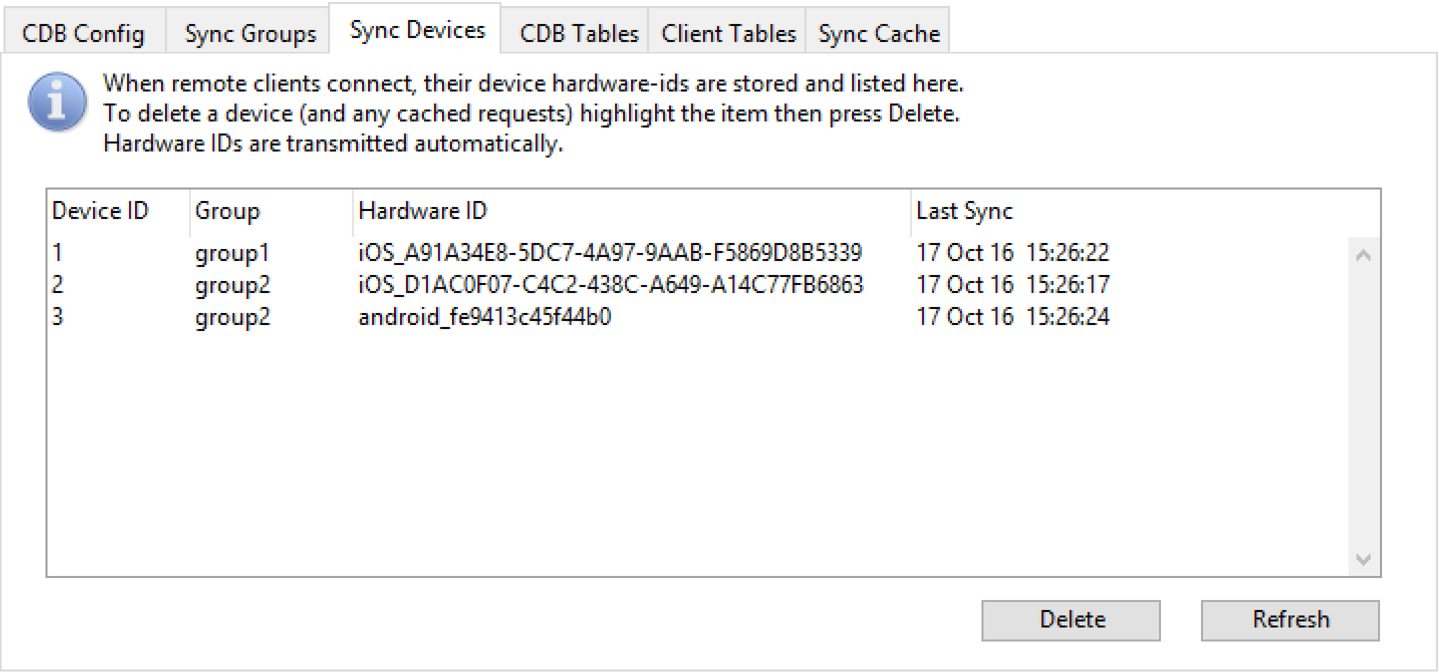
When adding or removing groups or modifying entries, press *Save Changes* to store the values.



Each remote client connects using a unique hardware identifier plus a group name and password. Define one or more groups by pressing Add, complete the details then Save Changes.
To delete a group, clear the Group name and Save Changes. (Device IDs will also be deleted).

Group Name	Group Password	Color
group1	test	🟥

Figure 3:



When remote clients connect, their device hardware-ids are stored and listed here.
To delete a device (and any cached requests) highlight the item then press Delete.
Hardware IDs are transmitted automatically.

Device ID	Group	Hardware ID	Last Sync
1	group1	iOS_A91A34E8-5DC7-4A97-9AAB-F5869D8B5339	17 Oct 16 15:26:22
2	group2	iOS_D1AC0F07-C4C2-438C-A649-A14C77FB6863	17 Oct 16 15:26:17
3	group2	android_fe9413c45f44b0	17 Oct 16 15:26:24

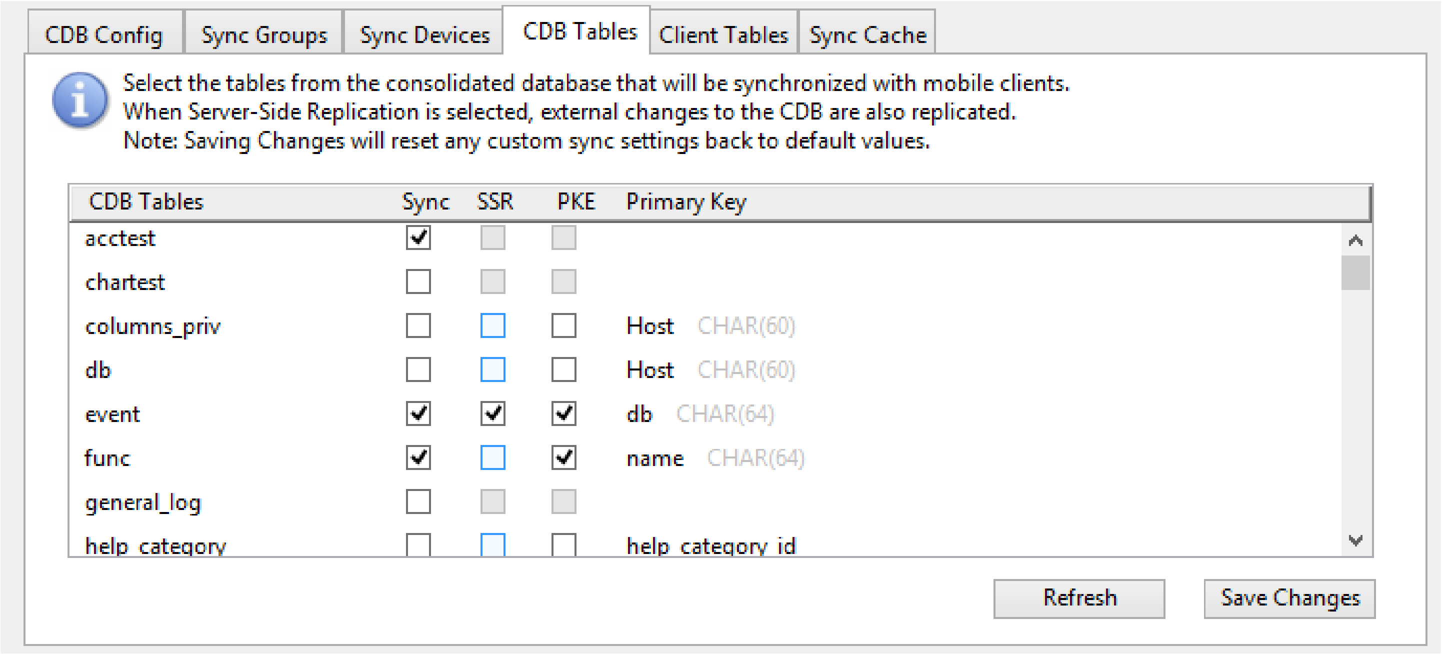
Figure 4:

Synchronisation Devices

This pane provides a log showing individual client devices that have connected to the SyncServer, the device id, which group they authenticated against, the device's unique hardware ID and the timestamp when the device last made a synchronisation request. Values shown are read-only and are assigned automatically.

The *Delete* button is used to delete a selected device and any cached requests/reloads pending for that device.

CDB Tables



Select the tables from the consolidated database that will be synchronized with mobile clients.
When Server-Side Replication is selected, external changes to the CDB are also replicated.
Note: Saving Changes will reset any custom sync settings back to default values.

CDB Tables	Sync	SSR	PKE	Primary Key
acctest	☑	□	□	
chartest	□	□	□	
columns_priv	□	□	□	Host CHAR(60)
db	□	□	□	Host CHAR(60)
event	☑	☑	☑	db CHAR(64)
func	☑	□	☑	name CHAR(64)
general_log	□	□	□	
help_category	□	□	□	help_category_id

Figure 5:

The CDB Tables pane lists all visible tables* in the Consolidated Database. To register a table for synchronisation, check the *Sync* box for that table.

For tables that employ a primary key, the primary key column name plus SQL data type is displayed and the corresponding *SSR* and *PKE* checkboxes are enabled. SSR invokes Server-Side Replication; a system of server-side triggers that enable the SyncServer to track changes made to the CDB independently of the SyncServer and its mobile clients. In other words, if another Omnis library or third-party application executes INSERTs, UPDATEs or DELETEs against any SSR-enabled table, the SyncServer will incorporate these into its download cache the next time a client connects.

PKE stands for Primary Key Enforcement. When enabled (default), mobile clients will override and substitute values inserted into the primary key column with values from a prescribed range (based on the device ID number). This mechanism is intended to prevent conflicting key values being inserted from multiple devices.
If disabled, any values inserted into the primary key column will be retained.

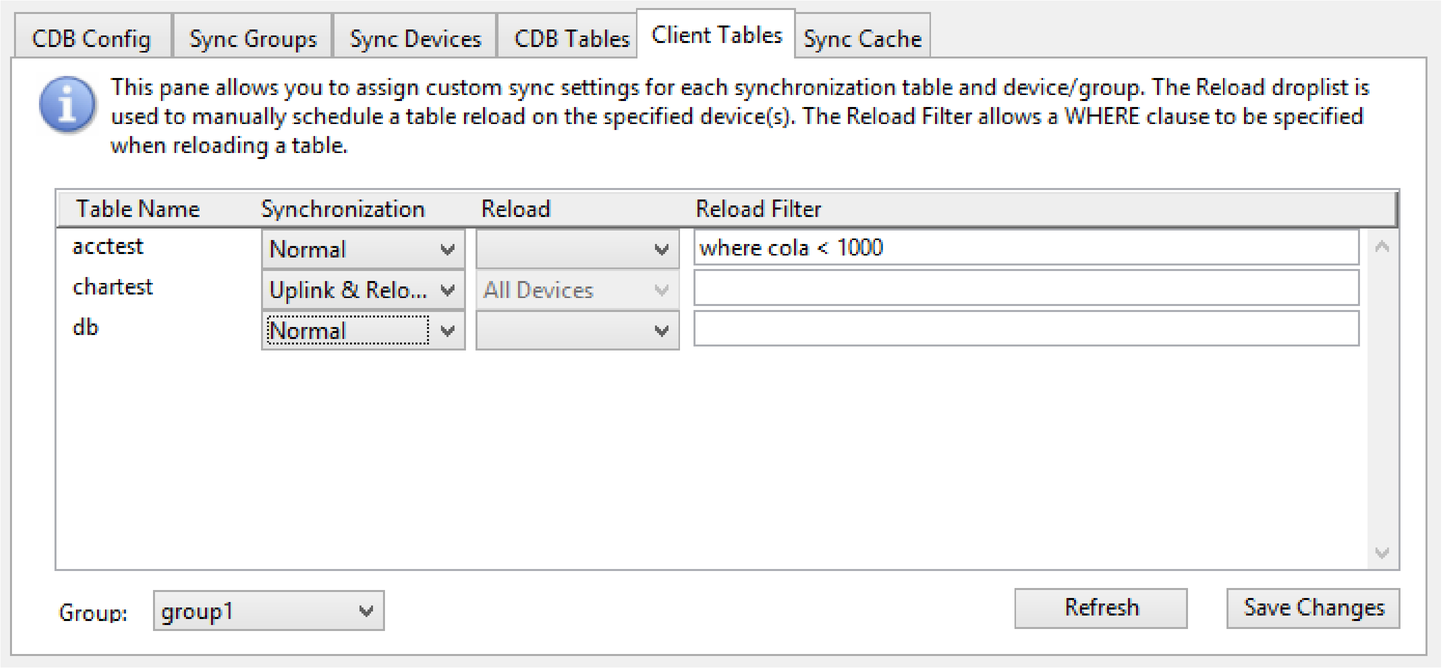
When finished selecting tables for synchronisation, press the *Save Changes* button to store changes.

Note that if the column structure of a CDB table changes it will be necessary to de-select, (*Save Changes*) then re-select (*& Save Changes*) that table. This will allow the SyncServer to build a fresh description of the table ready for transmission to clients. Any cached requests based on the previous table definition may subsequently fail to execute when sent to remote clients.

*The list of tables displayed may exclude certain system tables as well as those used to provide Server-Side Replication.

Client Tables

This pane lists every CDB table selected for synchronisation (via *CDB Tables*) and allows the synchronisation type to be specified for each. Before assigning synchronisation types, ensure that the correct group is selected using the Group drop list. (Groups can use different synchronisation types for each CDB table.)



This pane allows you to assign custom sync settings for each synchronization table and device/group. The Reload droplist is used to manually schedule a table reload on the specified device(s). The Reload Filter allows a WHERE clause to be specified when reloading a table.

Table Name	Synchronization	Reload	Reload Filter
acctest	Normal		where cola < 1000
chartest	Uplink & Relo...	All Devices	
db	Normal		

Group: group1

Figure 6:

The synchronisation type for a given table / group combination can be either *Normal*, *Uplink only*, *Downlink only*, *Uplink and reload*, *Reload only* or *None*. Synchronisation types are explained later.

When a client first connects to the SyncServer, it receives and executes a SQL statement to create each table and then requests a reload in order to populate it. If only part of the table should be sent to the client, the *Reload filter* can be used to specify a WHERE clause that will restrict the number of rows obtained from the CDB table. For example, If *test* has a column named *test_id* then the following could be entered as the reload filter:

```
where test_id >= 1000 and test_id < 100000
```

The *Reload* droplist allows the SyncServer administrator to manually schedule a reload of that table to the individual device specified. This might be necessary if the data in the CDB table has changed significantly, for example. A table reload overrides any pending IUD requests cached for that device since all applicable table rows will be sent to the client. To schedule a reload to several devices, choose the device then press *Save Changes*, and repeat as necessary. To schedule reloads to all devices in the group at once; choose *All Devices* followed by *Save Changes*.

Note: As of SyncServer version 2.4.4, there is an additional DROP checkbox for each table. If a table is selected for manual reload, this option allows the client-side table to be dropped, instead of requesting a reload. This might be necessary if the CDB table definition changes and avoids the need to manually delete tables from each client device (or having to delete and reinstall the app on each device). The client-side table is recreated next time the client issues a $syncinit(), and is reloaded automatically when a $sync() is issued.

Use of Bind Variables

If you want to create a WHERE clause for a specific user or client device, it is possible to pass custom parameters to oSqlObject.$syncinit() and refer to these in the SyncServer using bind variable notation.

For example, to add an additional parameter named ‘MyID’ you would call $syncinit() as follows:

```
Do config.$define(Username, Password, HostString, Timeout, MyID)
Do config.$assigncols(‘user1’,’xxxxxx’,’http://192.168.0.10:7001/ultra?OmnisClass=rtSync&OmnisLibrary=SyncServ
Do oSQL.$syncinit(config) Returns id
```

To use the custom parameter inside the SyncServer *Reload filter*, you could specify the following:

When the client device next reloads this table, the supplied value (*1234* in this example) will be substituted. Note that the client must call $syncinit() with matching parameter name(s), otherwise the table reload will fail.

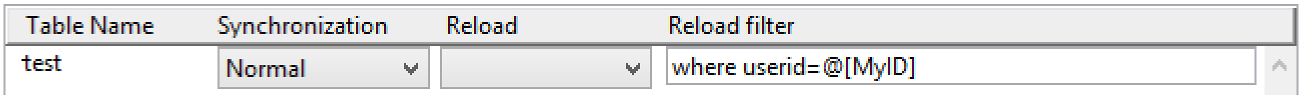


Table Name	Synchronization	Reload	Reload filter
test	Normal		where userid=@[MyID]

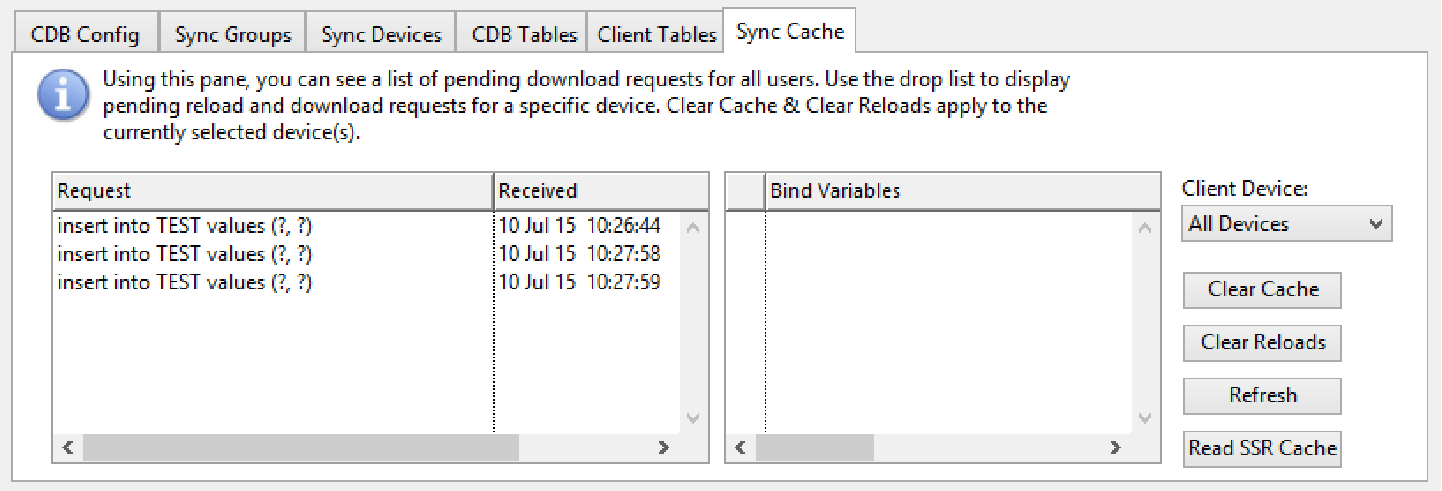
Figure 7:

For further information on the $syncinit() method, please refer to *Synchronisation Initialization*.

Note: As of SyncServer version 2.4.0, the SELECT statement used to fetch rows from the CDB uses prefixed column names and an aliased table name; "a". This change allows a JOIN clause to be used in the reload filter without affecting the result column definitions. The above example therefore becomes:

```
where a.userid=@[MyID]
```

Sync Cache



Using this pane, you can see a list of pending download requests for all users. Use the drop list to display pending reload and download requests for a specific device. Clear Cache & Clear Reloads apply to the currently selected device(s).

Request	Received
insert into TEST values (?, ?)	10 Jul 15 10:26:44
insert into TEST values (?, ?)	10 Jul 15 10:27:58
insert into TEST values (?, ?)	10 Jul 15 10:27:59

Bind Variables

Client Device: All Devices

Figure 8:

The Sync Cache tab contains a log of synchronisation requests that are pending for one or more client devices.

The SQL statement together with the timestamp when the request was received are shown on the left. Bind variable markers are represented by '?'. Where possible, selecting a SQL statement will display the specific bind variable values on the right-hand side.

The *Client Device* droplist can be used to filter requests pending for that device only.

Where a reload is scheduled for a table, the reload information is displayed in the *Request* column.

The *Clear Cache* button will remove pending requests for the selected device. Note that if *All Devices* is selected this will result in the entire cache being cleared.

The *Clear Reloads* button will cause table reload requests for the selected device(s) to be removed.

The *Refresh* button causes the display to update in respect of any synchronisation requests received since opening the Sync Cache tab.

The SyncServer normally reads the Server-Side Replication cache on the CDB when a synchronisation request is received. The *Read SSR Cache* button can be used to manually read and update the sync cache with external changes made to the CDB. This might be useful if no devices have contacted the SyncServer recently or if a lot of external changes have been made.

Synchronisation Types

The Omnis Synchronisation Server supports several synchronisation types that affect how the data from/to a particular client table is handled when the client executes the $sync() method. These are assigned via the SyncServer's *Client Tables* tab.

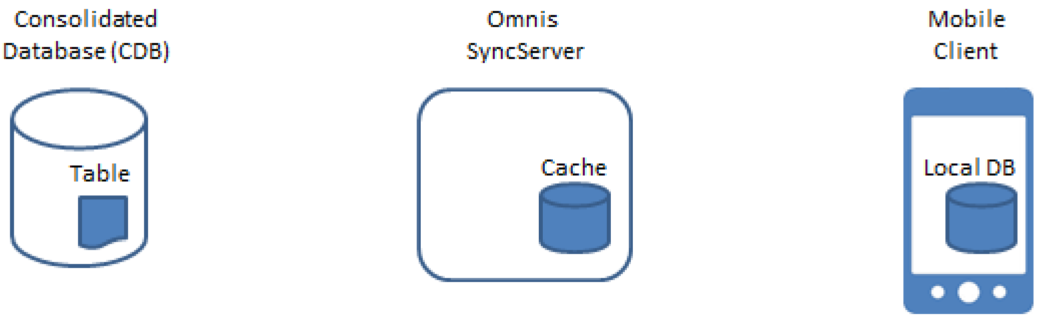


Figure 9:

Consider the scenario above where a client device will use the SyncServer to synchronise one table with the CDB. In the following discussion *uplink* refers to data sent from the mobile client device to the SyncServer. *Downlink* refers to data sent from the SyncServer to the client device.

Uplink only

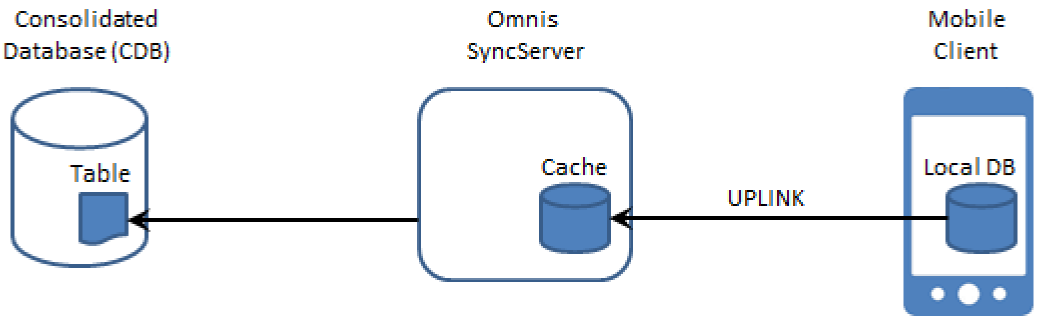


Figure 10:

'Uplink only' means that only uplink synchronisation will be performed, i.e. INSERTs, UPDATEs and DELETEs (IUDs) executed on the client will be sent to the SyncServer, cached for transmission to other clients and executed on the CDB. No IUDs will be sent back to the client for that table. The table will only be populated during a $sync() reload if it does not already exist, or otherwise if a manual reload is scheduled.

Downlink only

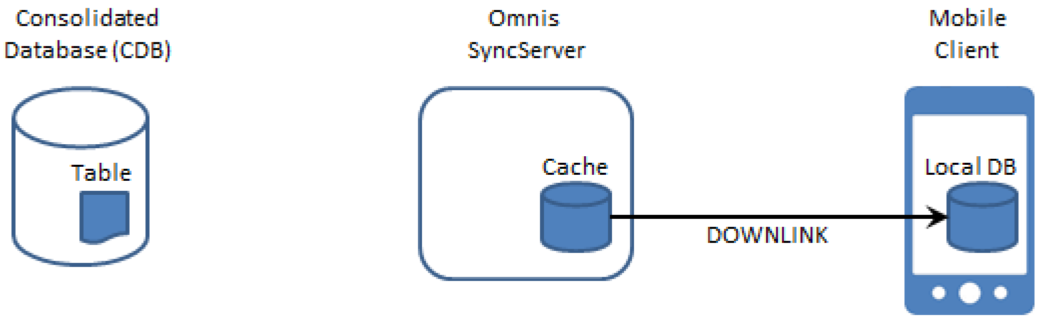


Figure 11:

'Downlink only' means that IUDs executed on the client device will not be transmitted to the SyncServer during $sync(). Changes made to the client-side database are therefore volatile and subject to loss if/when the table gets reloaded.

In response to a $sync(), the client will be sent any cached IUDs received from other clients (incorporating any IUDs read in from the Server-Side cache, if enabled).

Normal Synchronisation

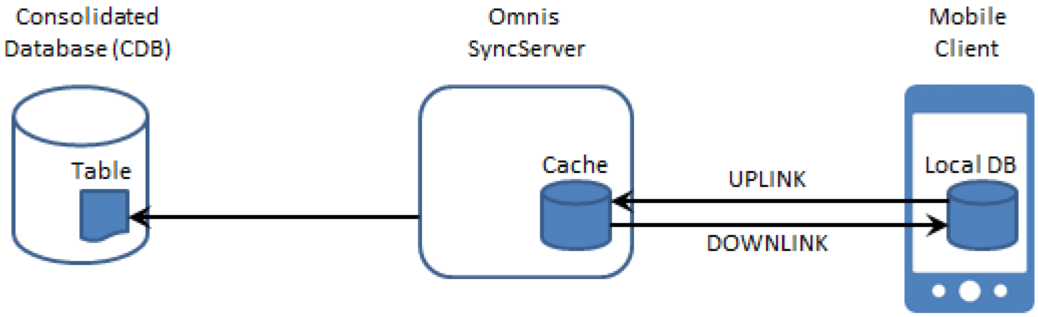


Figure 12:

'Normal synchronisation' is two-way and involves both an uplink and a downlink phase. IUDs cached on the client device are first transmitted to the SyncServer. These are cached and executed on the CDB.

During the downlink phase, one or more responses are generated containing IUDs pending for the client device. Once the client acknowledges receipt of the IUDs the SyncServer removes them from its cache for that device.

Reload only

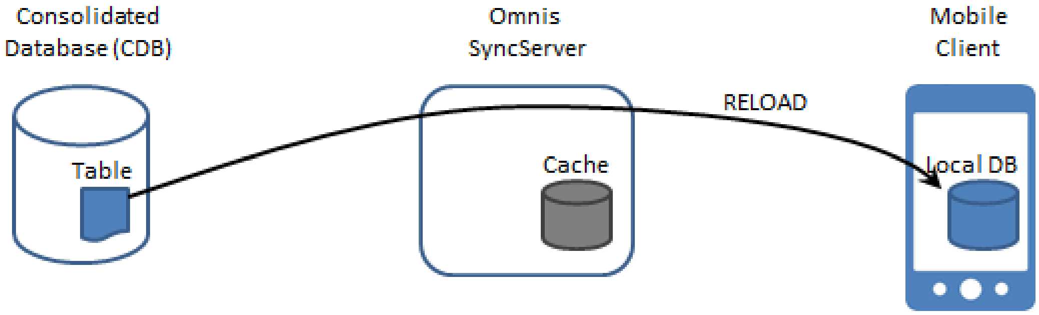


Figure 13:

In 'Reload only' mode, IUDs are not cached on the client device. When the client executes a $sync(), no uplink or downlink synchronisation occurs. Instead, the entire table (or the part specified by the Reload filter) is read and sent to the client device. An initial DELETE statement sent to the client clears any existing data before repopulating the table.

Note that this mode is distinct from the Reload drop list found on the *Client Tables* tab. The drop list schedules a one-time manual reload only, whereas this mode causes the table to be reloaded every time the client synchronises.

Uplink and Reload

In 'Uplink and Reload' mode, the device caches IUDs and sends them to the SyncServer during the uplink phase where they are executed on the CDB and cached ready for other client devices. During the downlink phase however, the cache is ignored and instead the entire table (or the part specified by the Reload filter) is read and sent back the client device.

None

In 'None' mode, the table definition is not sent to the client device and it is not created. In the event that the client app subsequently creates a table by the same name, the SQL object on the client device will not cache IUDs executed against the table. During $sync() no uplink, downlink or reload synchronisation will occur for that table.

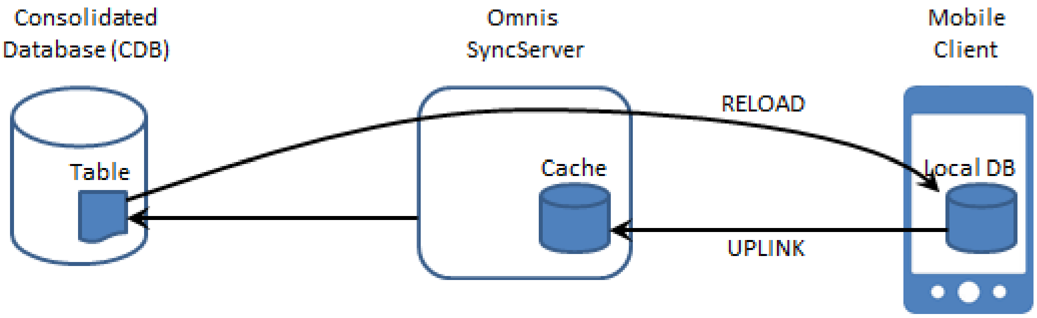


Figure 14:

This mode is useful where you might have two or more groups and each group requires access to different tables.

Note that in this mode, the table will not be deleted if it already exists. To clear the table, use *Reload* or *Uplink and Reload* instead and a WHERE clause similar to 'where 1 = 0'.

Server-Side Replication

The Server-Side Replication (SSR) feature built into the SyncServer allows external changes made to the consolidated database to be tracked and incorporated into the SyncServer's cache.

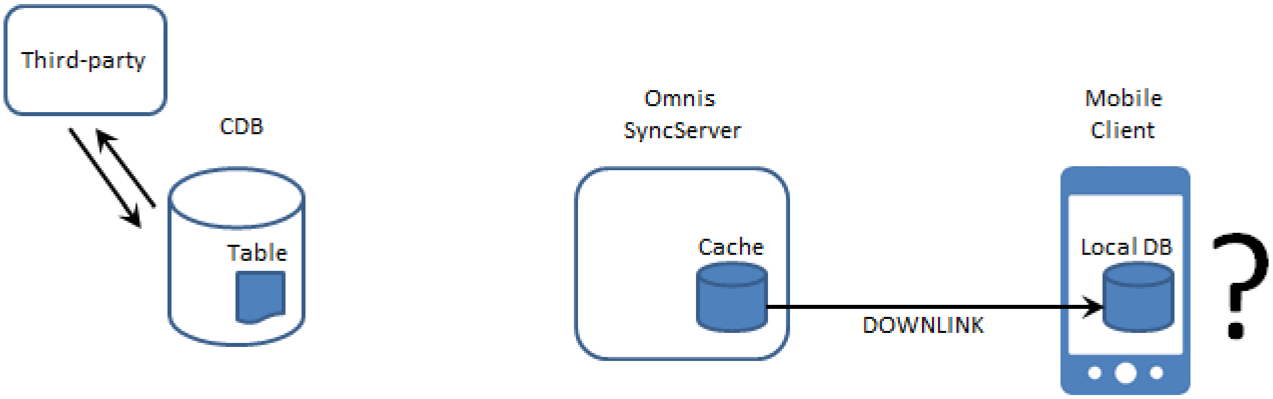


Figure 15:

Without SSR, the SyncServer is oblivious to INSERTs, UPDATEs and DELETEs executed by third-party applications and its cached version of changes made to the database will be out-of-sync with the CDB.

In order for clients to obtain data from tables that are not enabled for SSR, the *Uplink and Reload* or *Reload only* synchronizarion mode should be used. This bypasses the SyncServer cache and obtains a fresh copy of the table data each time the client synchronises.

When enabled for SSR, triggers created on the CDB maintain a cache that tracks IUDs executed by third-party clients. When the client synchronises, the server-side cache is read, purged, and incorporated into the SyncServer's cache. Requests read from the server-side cache are scheduled for transmission to all client devices that subscribe to that table.

Since the server-side triggers track *all* IUDs executed on the CDB, the SyncServer writes "suppression markers" into the server-side cache before and after execution of uplink requests received from each client device. When reading the server-side cache, the SyncServer ignores any requests that occur between the suppression markers. This prevents the possibility of cyclical scheduling of requests.

Tables enabled for SSR should ideally use *Normal* synchronisation mode. Although any of the supported modes will still work, the concept of reloading tables becomes redundant since all changes are now tracked.

The SyncServer supports SSR trigger creation on the following server-side databases:

- MySQL

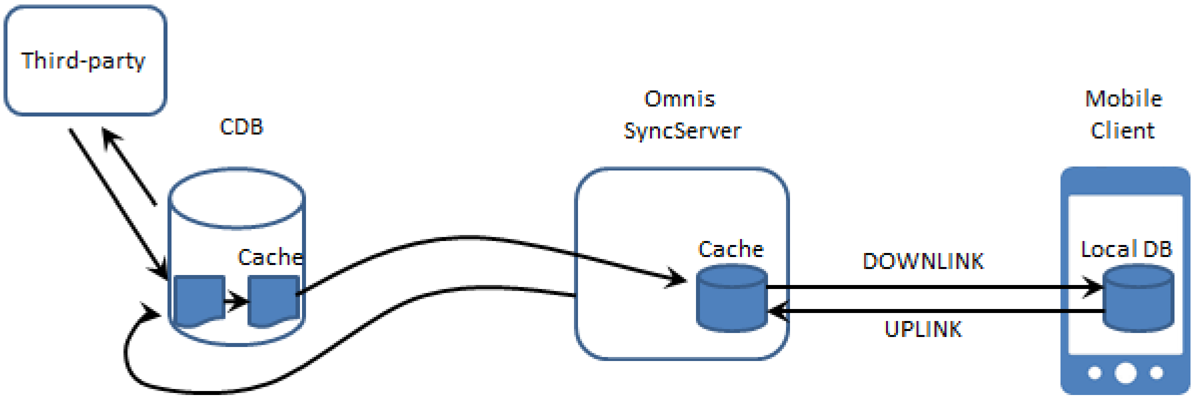


Figure 16:

- PostgreSQL
- Oracle
- SQLite
- Sybase ASE & ASA
- DB2
- MS SQL Server

Handling Primary Keys

Where a consolidated database table is defined with a primary key column, the situation can arise whereby two or more synchronisation clients attempt to upload records containing the same primary key value. This will cause a 'duplicate key' insertion error when the synchronisation server attempts to honor the second and subsequent inserts.

The SyncServer provides a mechanism to avoid this issue by allocating a pool of primary key values to each client device. Assuming an unsigned 32-bit integer for the primary key column, it uses the most-significant 10 bits as a device-id mask and the remaining 22-bits for the identifier value. This provides each client with a maximum of 4194303 unique identifiers per table and also ensures primary key isolation.

Device-id 0x001-3FF	Unique-identifier 0x000000-3FFFFF
(10-bits)	(22-bits)

For example; device ID 1 will be allocated a pool of primary key values in the range 0x400000 to 0x7FFFFF, or 4194304 to 8388607.

It should be noted that since allocated primary key values commence at 4194304, values 0 to 4194303 are available for use by external clients.

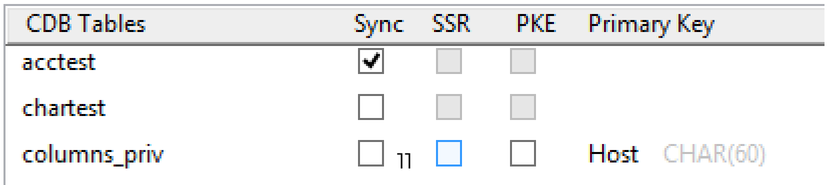
During synchronisation initialization, the next available primary key value is calculated based on the device-id and any values already present in the table.

Subsequently, when the client processes an $insert() for that table, the client substitutes that table's keyval in place of any specified primary key value.

As of wrapper version 1.3.2 and SyncServer v2.2, this mechanism supports both INTEGER and CHAR primary keys (writing integer values into the CHAR columns).

Primary Key Enforcement

To disable the "Primary Key Enforcement" mechanism (PKE), you can uncheck the PKE setting from the SyncServer's CDB Tables tab:



CDB Tables	Sync	SSR	PKE	Primary Key	
acctest	☑	□	□		
chartest	□	□	□		
columns_priv	□	□	□	Host	CHAR(60)

Omnis Running as a Service

When Omnis is running as a service on Windows x86/x64 systems, SyncServer log entries which are normally displayed inside the visual interface are instead written to a log file.

When Omnis is installed in the C:\Program Files (or Program Files (x86)) folder, the log file is created in the installed writable files directory, as returned by sys(115). For example:

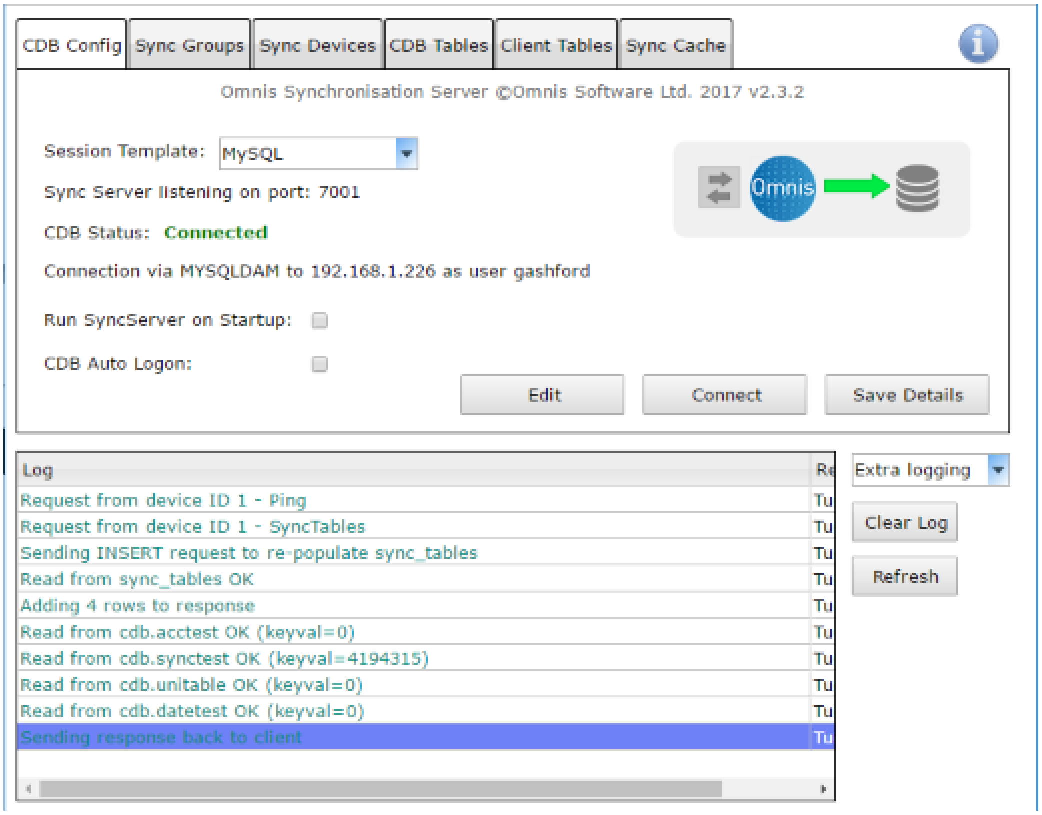
```
C:\Users\....\AppData\Local\Omnis Software\OS8.0.2\syncserver.log
```

The syncserver.log file is automatically truncated so that it never exceeds 64KB in size.

Using the Web Interface

The SyncServer additionally supports remote administration via its web interface. To use the web interface, direct your web browser (or JSWrapper application) to the SyncServer's IP address and server port using a URL similar to:

```
http://127.0.0.1:7001/jschtml/rfSyncServer.htm
```



Omnis Synchronisation Server ©Omnis Software Ltd. 2017 v2.3.2

Session Template: MySQL

Sync Server listening on port: 7001

CDB Status: Connected

Connection via MYSQLDAM to 192.168.1.226 as user gashford

Run SyncServer on Startup: □

CDB Auto Logon: □

Extra logging

Log	Re
Request from device ID 1 - Ping	Tu
Request from device ID 1 - SyncTables	Tu
Sending INSERT request to re-populate sync_tables	Tu
Read from sync_tables OK	Tu
Adding 4 rows to response	Tu
Read from cdb.acctest OK (keyval=0)	Tu
Read from cdb.synctest OK (keyval=4194315)	Tu
Read from cdb.unitable OK (keyval=0)	Tu
Read from cdb.datetest OK (keyval=0)	Tu
Sending response back to client	Tu

Figure 18:

The Remote Form rfSyncServer.htm closely models the desktop interface

To enable remote administration, use the desktop interface and check the 'Enable Web Interface' checkbox on the CDB Config pane. (This is available as of SyncServer version 2.3.2). When you press 'Save Changes' this prompts the SyncServer library to generate the rfSyncServer.htm file, initially inside the Omnis\html folder. The form then opens inside your default web browser.

The remote form can then be moved to your webserver's public html folder as normal if you want to enable SyncServer administration over the internet.

If you uncheck the 'Enable Web Interface' and press 'Save Changes', this prompts the SyncServer library to delete rfSyncServer.htm inside the Omnis\html folder. Please note that it will *not* delete any copy placed in the webserver's public html folder.

Synchronising with the SyncServer

The client-side application accesses an internal SQLite interface using a SQL Object as described in the *Creating Web & Mobile Apps* documentation. (See the online documentation for details).

Example:

```
Do $cinst.$sqlobject Returns oSqlObject
```

The SQL Object provides $insert(), $update and $delete() methods to ensure that INSERT, UPDATE and DELETE SQL statements are cached correctly as well as being executed on the client-side database. Other methods such as $execute(), $selectfetch() and $fetch() are *not* subject to synchronisation.

Mobile client devices connect to the SyncServer using the SQL Object methods $syncinit() and $sync().

$syncinit() expects a number of parameters that tell it how to connect to the SyncServer and needs to be called once when client network access has been restored and before calling $sync().

Once $syninit() has been called and contact is made, the client can execute the $sync() method. $sync() requires no parameters and works using the connection parameters established previously.

Synchronisation Initialization

```
oSqlObject.$syncinit(syncParams) Returns id
```

This method is called with a row variable containing the synchronisation group authentication details (a group name and password), as well as the host URL used to contact the SyncServer and an optional timeout value that will apply to HTTP transmission.

The SQLite module currently recognizes the following parameters:

- **Username** – The synchronisation group name (defined at the Synchronisation Server).
- **Password** – The synchronisation group password (defined at the Synchronisation Server).
- **HostString** – Omnis RESTful URL to the Synchronisation Server.
- **Timeout** – The timeout in seconds for synchronisation operations. (Optional, defaults to 5 seconds).
- **<custom>…** – Zero or more user-defined parameters to be passed to the SyncServer during $sync() requests.

On completion, $sqldone() is called with the following parameters:

- The request id (as returned from $syncinit ()).

Example:

```
Do config.$define(Username, Password, HostString, Timeout, MyParam)     ## define using local variables
Do config.$assigncols(‘user1’,’xxxxxx’,’http://192.168.0.10:7001’, 5, 1234)
Do oSQL.$syncinit(config) Returns id
```

For a direct connection to the built-in Omnis Server, the HostString should be:

```
http://<ipaddress>:<$serverport>
```

If you are connecting through a web server, you need to add the omnisrest… server

plugin to your web server, in the same way as the other server plugins described in

Tech Note: TNJS0003, and connect through that.

The HostString should then be of the form:

```
http://<web server address>/<Omnis rest plugin>/ws/<XXX>
```

Where <XXX> is either:

- <Omnis $serverport> (if Omnis is on the same machine as the web server)
- <Omnis server ipaddress>_<Omnis $serverport>
- <Server Pool>_<Omnis server ipaddress>_<Omnis $serverport>

For example:

```
http://mysite.com/cgibin/omnisrestisapi.dll/ws/192.168.1.14\_7001
```

Where custom parameters are supplied to $syncinit(), these are stored and subsequently passed to the SyncServer during $sync() requests along with the other parameters: the application of custom parameters is described above. (See *Client Tables.*)

Synchronisation Request

```
oSqlObject.$sync()
```

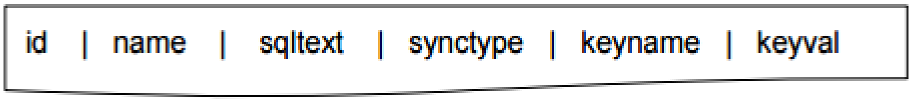
This method invokes uplink synchronisation followed by downlink synchronisation. Only tables previously configured for uplink (or normal) synchronisation will upload IUD requests to the SyncServer. Likewise only tables configured for downlink or (normal) synchronisation will receive IUD requests.

Table reloads are also sent during the downlink phase which may require several round-trips (network transactions) in order to complete depending on the size of the table and any WHERE clause used to filter the data. (See *Client Tables.*)

Client-side Synchronisation Tables

During $syncinit(), three sync-admin tables are created on the client device. The client app should not normally need to interact with these tables although they are described here for your information.

sync_tables - provides information for creating and identifying synchronisation tables.



id | name | sqltext | synctype | keyname | keyval

Figure 19:

id	A unique ID for the table
name	The table name
sqltext	The CREATE TABLE statement that may be used to create the table
synctype	A single character that stores the synchronisation type for the current<br/> group. Key: 0=not sync'd, X=normal sync, U=upl D=downlink only,<br/> R=reload & uplink, V=reload only.The client only uploads SQL statements when a table is flagged as X only receives downloads when the table is flagged as X, D, R or V
keyname	The name of the table's primary key column (if one exists)
keyval	The table's next available integer primary key value (if one exists)

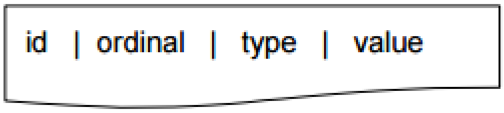
sync_tables is always dropped, created and re-populated during $syncinit().

sync_cache - stores INSERT, UPDATE and DELETE SQL statements to be executed on the CDB.

id	A unique ID for the SQL statement
sqltext	The statement's SQL text, flattened so that '?' characters represent bind variables

id | sqltext

Figure 20:



id | ordinal | type | value

Figure 21:

sync_bind - stores bind variable values for cached statements, one value per row.

id	The statement (sync_cache) ID to which this value applies
ordinal	The order in which the bind variable appears in the SQL statement,1 based
type	An enumerated integer that corresponds to the Omnis data type
value	Stores a bind variable’s value in text format. Binary values are base64- encoded, other values are stored in human-readable fo

Data types used for the *type* column include:

0	A Null value – *value* will be ignored
21	kCharacter – *value* contains character data
22	kBoolean – *value* contains a 1 or 0
23	kDate – *value* contains an ISO datetime
24	kSequence – *value* contains an integer
25	kNumber – *value* contains a floating point number
26	kInteger – *value* contains an integer
28	kBinary – *value* contains a base64-encoded binary value

The sync_cache and sync_bind tables are only created where they do not already exist so as to preserve any previously cached IUD requests. When your app calls $insert(), $update() and $delete(), the resulting SQL statements are stored in these tables ready for transmission to the SyncServer.

FAQs

How many times should I call $syncinit() / $sync()? You only need to call $syncinit() once when the client device comes into network range, but it is not detrimental to call it again. Once initialized, you only need to call $sync() once. This will cause all IUDs cached on the device to be sent to the SyncServer. Following this, all downloads scheduled for the device will be sent in one or more round trips. If further INSERTs, UPDATEs or DELETEs are then executed on the device, $sync() can be called periodically to synchronise these. Note that $sync() will fail unless $syncinit() is called first and is successful.

I called $syncinit() but my tables do not contain any data. Calling $syncinit() will create your client-side tables and issue reload requests where necessary, but they remain empty until you call $sync().

Can a client force a table to be reloaded? Yes, this is done by deleting the table on the client, then executing $syncinit() followed by $sync(). If table creation succeeds during initialization then a reload request is sent to the SyncServer during $sync().

Can client *A* synchronise a different set of tables to client *B*? Yes, this is achieved by creating two (or more) different groups. From the *Client Tables* tab, select the first group and set-up the synchronisation type for each CDB table. Select *None* if you want to exclude that table from the group. Then *Save Changes,* select the second group and repeat. Client A must call $syncinit() using the first group’s credentials. Client B must call $syncinit() using the second group’s credentials.

Can client A and client B both synchronise using the same device? The SyncServer is designed to work with one user/group-id per device. This is because the SyncServer stores cached IUD requests by device-id only. If you change users and re-execute $syncinit() followed by $sync(), you will not receive any IUDs uploaded by the previous user. These IUDs will already be inside the local SQLite database however, unless the settings for the new user cause the table(s) to be cleared and/or reloaded with different data. (The SyncServer stores table reload requests by device-id and group-id).

My Primary Key columns are being changed into large integer values. The wrapper substitutes its own primary key value during sqlobject.$insert(). Referring to the SyncServer's CDB Tables tab, if you deselect the PKE checkbox this will disable Primary Key Enforcement for that table. It should be noted however that the default behaviour is designed to ensure primary key isolation. In overriding this mechanism you should take your own measures to prevent duplicate primary key insertion errors.

How can I find the overridden Primary Key value that got cached for my INSERT? When you execute a sqlobject.$insert() for a table with a primary key, the wrapper library stores the next available key value inside the sync_tables table on the client. You can get it by executing a SQL statement similar to:

```
select keyval-1 as key from sync_tables where name = ‘table_name’
```

Why does SSR require a Primary Key? When a table is enabled for Server-Side Replication, three triggers are created on the CDB that are called whenever an INSERT, UPDATE or DELETE is executed on the table. Strictly speaking, only the *update* and *delete* triggers require a primary key because this is used in a WHERE clause that will uniquely identify the row when the request is executed on a client device. SSR of a table requires all three triggers in order to work correctly.

What if I add or change a CDB table definition? If you add or change a column to a CDB table that is already selected for synchronisation you will need to clear the SyncServer's sync_cache. This will prevent potentially bad IUDs from being sent to clients. You should also uncheck(& *Save Changes*) then re-check (& *Save Changes*) the table via the CDB Tables tab. This prompts the SyncServer to build a new CREATE TABLE statement for transmission to clients. You will need to $execute() a DROP TABLE statement on each client device then call $syncinit() in order to recreate the table using the new definition. Alternatively, delete the App (and database) from each client device and re-install it. As of SyncServer version 2.4.4, the Client Tables tab provides an additional DROP checkbox for each table. Ths forces clients to drop (and recreate the table).