Zumero for SQL Server: Client API

Table of Contents

1. About

This document was generated 2021-03-03 16:35:09.

At a high level view, ZSS, consisting of a server and a client library, replicates and synchronizes data between Microsoft SQL Server and SQLite database files on mobile devices.

The client library contains a function called zumero_sync(), which is the main API used to synchronize data. This document is primarily focused on explaining how to use zumero_sync().

Discussion of how to use SQLite in developing a mobile app is outside the scope of this document. Zumero will bring your data down to a mobile device in the form of a SQLite database file, but you can choose how you want to interact with that file. A wide variety of SQLite APIs and tooling choices are available.

It is also possible to call zumero_sync() from languages other than C, including C#, Objective-C or Java. Discussion of the details of these wrappers is also outside the scope of this document.

2. Basics of zumero_sync()

The header file for the ZSS Client SDK is zumero_client_api.h. The comments in that file contain various details about the use of the API.

The most important functions are `zumero_sync()`, `zumero_sync2()` and `zumero_sync3()`. In many cases, they are the only functions you need (and you're only likely to need one of them).

```
int zumero_sync(
  const char *zFilename,
  const char *zCipherKey,
  const char *zServerUrl,
  const char *zDbfile,
  const char *zAuthScheme,
  const char *zUser,
  const char *zPassword,
  const char *zTempDir,
  char **pzErrorDetails
);

int zumero_sync2(
  const char *zFilename,
  const char *zCipherKey,
  const char *zServerUrl,
  const char *zDbfile,
  const char *zAuthScheme,
  const char *zUser,
  const char *zPassword,
  const char *zTempDir,
  zumero_progress_callback * fnCallback,
  void * pCallbackData,
  char **pzErrorDetails
);

int zumero_sync3(
  const char *zFilename,
  const char *zCipherKey,
  const char *zServerUrl,
  const char *zDbfile,
  const char *zAuthScheme,
  const char *zUser,
  const char *zPassword,
  const char *zTempDir,
  zumero_progress_callback * fnCallback,
  void * pCallbackData,
  const char *jsOptions,
  int *syncId,
  char **pzErrorDetails
);
```

Parameter	**Description**
`zFilename`	the path to the SQLite database file
`zCipherKey`	decryption key, if applicable
`zServerUrl`	https://wherever
`zDbfile`	name of the DBFile on the server
`zAuthScheme`	credentials for this sync. auth scheme string.
`zUser`	credentials for this sync. user name.
`zPassword`	credentials for this sync. password.
`zTempDir`	path for the directory to be used for temporary files
`fnCallback`	Progress callback function
`pCallbackData`	An opaque pointer, which will be passed to the zumero_progress_callback function
`pzErrorDetails`	ptr to receive a string with error details
`jsOptions`	optional JSON string listing sync options

Parameter	Description
`syncId`	if `sync_details` is `true` in `jsOptions`, the id used to retrieve details about this sync's effects

Return value: a result code (see zumero_client_api.h). on success, 0.

Notes:

- If the application has been linked with SQLCipher or SQLite Encryption Extension, zCipherKey will be used (with PRAGMA key) when accessing the file. Pass NULL if this is not needed.
- This function involves network activity and will block until the sync operation is complete. Best practice is to call this function in a background thread.
- zFilename refers to the name or path of the local SQLite file on the client. If it does not exist, it will be created.
- zDbfile is the name you used when creating the DBFile using the ZSS Manager application.
- A DBFile name must begin with a lower-case letter and must contain only lower-case letters, digits, or underscores. Any DBFile name that begins with "zumero_" is reserved for internal use.
- zFilename and zDbfile do not need to be the same name.
- zTempDir is needed only for Android.
- The memory returned in pzErrorDetails must be freed with zumero_free().
- To perform the sync without authenticating, pass NULL for scheme, user, and password.
- The corresponding DBFile on the server must be the same DBFile that the local db has synced with in the past (if applicable).
- If zServerUrl is an internationalized domain name, the Punycoded version of the domain must be used.
- For details on using the progress function, see zumero_client_api.h

Three other functions in the API are commonly used:

Function	Description
`zumero_cancel()`	use this to cancel an ongoing zumero_sync2 operation
`zumero_free()`	use this to free any memory returned by another Zumero API function
`zumero_errstr()`	return the English-language text that describes a Zumero result code

3. Manipulating Data in SQLite

ZSS does not force you to choose a particular SQLite API. The most popular SQLite toolkit varies by platform and you are free to pick one that best suits your needs. But on all client platforms, you'll want to enable two features that are not turned on by default. Each time you open a SQLite handle, execute the pragma statements to enable foreign keys and recursive triggers:

```
PRAGMA foreign_keys = ON;
PRAGMA recursive_triggers = ON;
PRAGMA journal_mode = WAL;
```

Enabling foreign keys means the client database will enforce the foreign key relationships you've defined in you SQL Server database.

Recursive triggers are necessary for Zumero's triggers to correctly handle SQLite INSERT OR REPLACE statements.

WAL mode is strongly recommended for improved concurrency. See Write-Ahead Logging [http://www.sqlite.org/wal.html] for more information.

4. Details for Advanced Users

4.1. Additional Functions in the API

These functions deal with advanced situations. See zumero_client_api.h for details.

Function	Description
`zumero_quarantine_since_last_sync()`	Move unsynched local changes into an isolated holding area
`zumero_sync_quarantine()`	Sync the local database with the server database, including the changes stored in the specified quarantine
`zumero_delete_quarantine()`	Permanently delete quarantined changes

4.2. Progress

If you use zumero_sync2, you can supply a callback function which will be called numerous times to give progress information. The following arguments will be passed to your callback function.

Parameter	Description
`int cancellation_token`	The cancellation token for this sync operation. The cancellation token will remain the same for all progress callbacks for a single sync operation.
`int phase`	The current sync phase. You may receive multiple progress callbacks for the same phase. Some phases may be repeated multiple times for a single sync operation. The phases are: • ZUMERO_PHASE_PREPARING (phase == 1). Examining the local SQLite database to determine which changes need to be uploaded. • ZUMERO_PHASE_UPLOADING (phase == 2). Uploading changes to the server. • ZUMERO_PHASE_WAITING_FOR_RESPONSE (phase == 3). Waiting for the server to apply our changes and calculate the response. • ZUMERO_PHASE_DOWNLOADING (phase == 4). Downloading changes from the server. • ZUMERO_PHASE_APPLYING (phase == 5). Applying the downloaded changes.
`zumero_int64 bytes_so_far`	If the phase is ZUMERO_PHASE_UPLOADING or ZUMERO_PHASE_DOWNLOADING, this argument will be the number of bytes that have been transferred so far.

Parameter	Description
`zumero_int64 bytes_total`	If the phase is ZUMERO_PHASE_UPLOADING or ZUMERO_PHASE_DOWNLOADING, this argument will be the number of bytes that will be transferred in this phase.
`void * data`	An opaque data pointer. You can provide this to zumero_sync2, and Zumero will not modify or use it.

4.3. History

When Zumero is doing synchronization, it sends packages of incremental changes between client and server. In order to facilitate this, Zumero keeps track of changes you have made to the database since your last sync.

For a Zumero table called FOO with id XYZ, under the hood, there are additional tables which keep track of history.

- The zoldXYZ table (the prefix zold, concatenated with the name of your table) contains rows (or versions of rows) that have been deleted since your last sync.

  This table and FOO both contain the same columns. The main difference between them is simply that one contains all your current stuff, and the other contains all of your old stuff.
- The zrvXYZ table gives a distinct identity to each version of each row, whether that version currently resides in FOO or in zoldXYZ.
- The zrdXYZ table contains a record of which row versions were involved in each INSERT, UPDATE and DELETE.

4.4. Sync Details

`zumero_sync3()` can optionally leave behind a list of local database rows affected by the sync call. You'll need to pass a `jsOptions` parameter including a `sync_details: true` value, *e.g.*:

```
zumero_sync3(mySqlFile, NULL, myZssUrl, "dbfile", NULL, NULL, NULL, myTempDir,
NULL, NULL, "{\"sync_details\": true}", &syncid, &errDetails);
```

`syncid` will receive a value with which to query two tables:

4.4.1. zumero_syncs

This table contains a single row per sync, including its `id`, the time at which the sync began, and a `refresh` column, which contains:

- `1` if this was a full-DBFile refresh, as when a DBFile is synched to this device for the first time.
- `0` otherwise.

4.4.2. zumero_sync_<table>

For each table affected by a sync, there will be one or more entries in a table (actually a view) named `"zumero_sync_" + name-of-table`. So if the table "rodents" had some rows modified, added, or deleted, details would be found in "zumero_sync_rodents".

Note

Only rows modified **as a result of the sync** will be recorded in the sync details tables. *i.e.*, rows modified on the server, or rows modified on other client devices, will be logged here when they

are synced to this client device. Rows which were added/modified/deleted on this client device will *not* be logged when they are synced to the server.

In most cases, each added, deleted, or modified row will receive its own row in the details table. Each row contains:

- the sync ID
- an "action" column ('i' for `INSERT`, 'u' for `UPDATE`, 'd' for `DELETE`)
- the old and new primary key column value(s), with names prefixed by `old_` and `new_`
- the current column values.

For deleted rows, the "current" values will all be `null`, as will the `new_...` primary keys.

For inserted rows, the `old_...` keys will be `null`.

If the table is wholly new (just added to the DBFile), or if a filter change has caused all data to be re-sent, a single row is logged instead. The action column will contain `'r'` and the various key and data columns will contain `null`.

4.4.2.1. examples

Assume a table "foo" with columns "a" and "b", where "a" is the primary key. A new row is added on the server, with `a = 1` and `b = "one"`.

We call `zumero_sync3()` and see that our sync ID was `127`. The matching row in `zumero_sync_foo` contains:

syncid	action	old_a	new_a	a	b
127	i	NULL	1	1	one

Another sync, wherein this row is updated (b becomes "two"):

syncid	action	old_a	new_a	a	b
128	u	1	1	1	two

One more, deleting our row and adding another:

syncid	action	old_a	new_a	a	b
129	d	1	null	null	null
129	i	null	2	2	a new entry

4.5. Quarantined Packages

A quarantined package is a collection of changes which have been removed from the database and placed in a waiting area. Typically, the reason something is quarantined is because of conflicts. In most cases, there is no need to quarantine anything. Zumero is designed to manage conflict resolution automatically. However, in some cases, it is appropriate to "undo" some changes to the client instance of the db.

The only way something can get quarantined is when you quarantined something intentionally by calling zumero_quarantine_since_last_sync(). The primary use case for this function is to remove changes from the client db because the server refused to accept them during a zumero_sync() operation.

When a package is quarantined, it is stored in a housekeeping table (named t$q). The rowid of that table is called the "quarantine id". This id can be used to reference the quarantined package when calling

zumero_sync_quarantine(). When you initially call zumero_quarantine_since_last_sync(), it returns the quarantine id through an output parameter.

The quarantine feature is an advanced aspect of Zumero. If you are careful to avoid conflicts during sync, you won't need to use it.

5. Security

5.1. Overview

The Zumero server supports security features which can be used to control who has permission to read or write DBFiles.

- Every request from a client can optionally include credentials for authentication.
- Each dbfile can have an Access Control List containing entries which allow or deny access to an item based on the effective identity resulting from authentication.
- The Zumero server can support multiple authentication schemes.

5.2. Authentication

Every request from a client can optionally include credentials for authentication. A set of credentials includes three things, the specific meanings for which can vary somewhat depending upon the specific authentication scheme being used:

Parameter	**Description**
`scheme`	metadata which describes the other two parameters. can be viewed as a description of the "group" in which the username exists.
`username`	something that identifies one specific user within the "group" specified by scheme
`password`	a password or token which needs to be validated for whatever username means within the context of whatever scheme is

The scheme defines the scope in which the username and password exist. It describes how and where the other two parameters are to be validated.

Under the hood, the scheme is a JSON string. The "scheme_type" key must always be present, and it must contain a recognized name for the type of scheme being described. The JSON object may also contain any additional name/value pairs which are appropriate for that kind of authentication scheme.

In most cases, you should use the "default" scheme:

```
{"scheme_type":"default"}
```

See the ZSS Manager Documentation [http://zumero.com/docs/zumero_for_sql_server_manager.html#users-and-permissions] for details on other possible scheme types.

The effective identity for a request is the pairing of the scheme string and the user name. If the client provides no credentials, the server continues to process the request without authentication. The effective identity is null. We use the word "anonymous" in describing this situation.

Anyway, the server will now proceed to figure out if the effective identity actually has permission to do whatever the client is requesting.

5.3. Permissions

Every request from a client will be denied unless the effective identity has been granted the necessary permission(s).

Note that simply being authenticated grants no permission to do anything. Even after successful authentication, the effective identity of "Madonna in Colorado" will not be able to do anything unless that identity has been granted permissions.

5.3.1. Access Control Lists

Every dbfile can optionally have an Access Control List (ACL) which grants or denies permission to do operations on that dbfile. The ACL can be managed from the ZSS Manager application.