ZSS Manager: User Guide

Copyright © 2013-2018 Zumero LLC

Table of Contents

1. Introduction 1
2. Quick Start 2
  2.1. Connect to ZSS Server's Database 2
  2.2. Create a DBFile 3
  2.3. Create a User and Set Permissions 5
  2.4. Add a Table to the DBFile 7
  2.5. Ready to Sync! 9
3. DBFiles, Defined 11
4. The ZSS Manager Window 12
  4.1. The selected DBFile 12
  4.2. Add a Table to the Current DBFile 12
  4.3. The Synchronized Tables list 12
  4.4. The Test Client Window 15
5. Customizing Synchronizations 16
  5.1. Authentication and Permissions 16
  5.2. Filters 23
  5.3. Constraint Violations and Conflict Resolution 26
6. SQL Server Considerations 34
  6.1. Create DBFile 34
  6.2. Add Tables to DBFile 34
  6.3. Stop Syncing a Table 37
  6.4. Delete DBFile 38
  6.5. Permissions 38
  6.6. SQL Server Triggers 38
  6.7. Database Schema Changes 39
  6.8. Data Type Conversion and Limitations 41
7. Advanced Topics 45
  7.1. Multiple Database Configurations 45
  7.2. Migrating ZSS Configuration between Servers 48
  7.3. Audit Trails 53
  7.4. The Server Log 54
  7.5. Upgrading ZSS 54
  7.6. Editing DBFile Connection Strings 55
  7.7. Recovering from a Database Rollback 57
  7.8. Purging History 58
  7.9. Adding Indexes to "z$old" Tables 60
8. Troubleshooting 60
  8.1. Troubleshooting Error 500 60
  8.2. Troubleshooting License Errors 61

1. Introduction

This document was generated 2018-08-20 14:05:55. It explains how to install and use ZSS Manager to configure SQL Server databases for Zumero synchronization.

ZSS Manager is a component of **Zumero for SQL Server**. It allows you to configure an existing SQL Server database for use with Zumero. In combination with the **ZSS Server** and **ZSS Client SDK**, you can replicate and sync your data among multiple devices.

Using Zumero for SQL Server for the first time typically involves these steps:

1. Install ZSS Manager on a Windows development machine.
2. Use ZSS Manager to configure the SQL Server database.
3. Install the Zumero Server on a Windows Server. Configure it to connect to your database.
4. Create a simple client application, or use the ZSS Manager Test Client to perform a test sync against the server.

This guide focuses on ZSS Manager and step 2.

ZSS Manager works with:

- Windows 7 and later
- SQL Server 2008 R2 and later

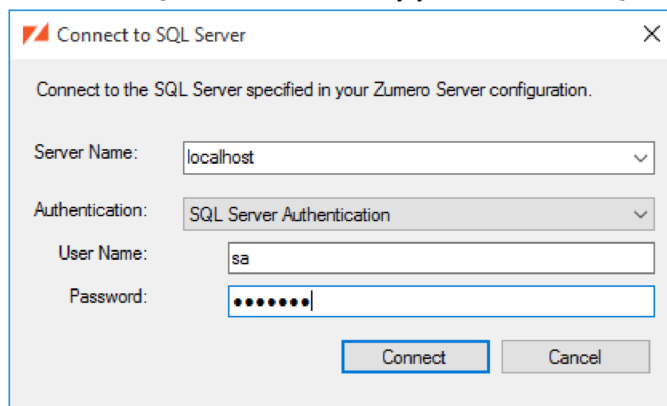
This document assumes the reader has familiarity with SQL Server as a developer or database administrator.

2. Quick Start

Before getting into details, let's take a quick tour of ZSS Manager's features and functionality.

2.1. Connect to ZSS Server's Database

Connect to SQL Server the same way you would with SQL Server Management Studio:

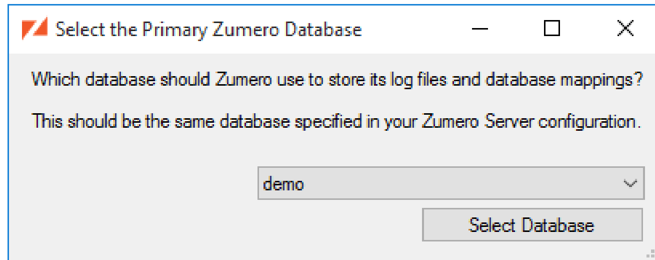


Note

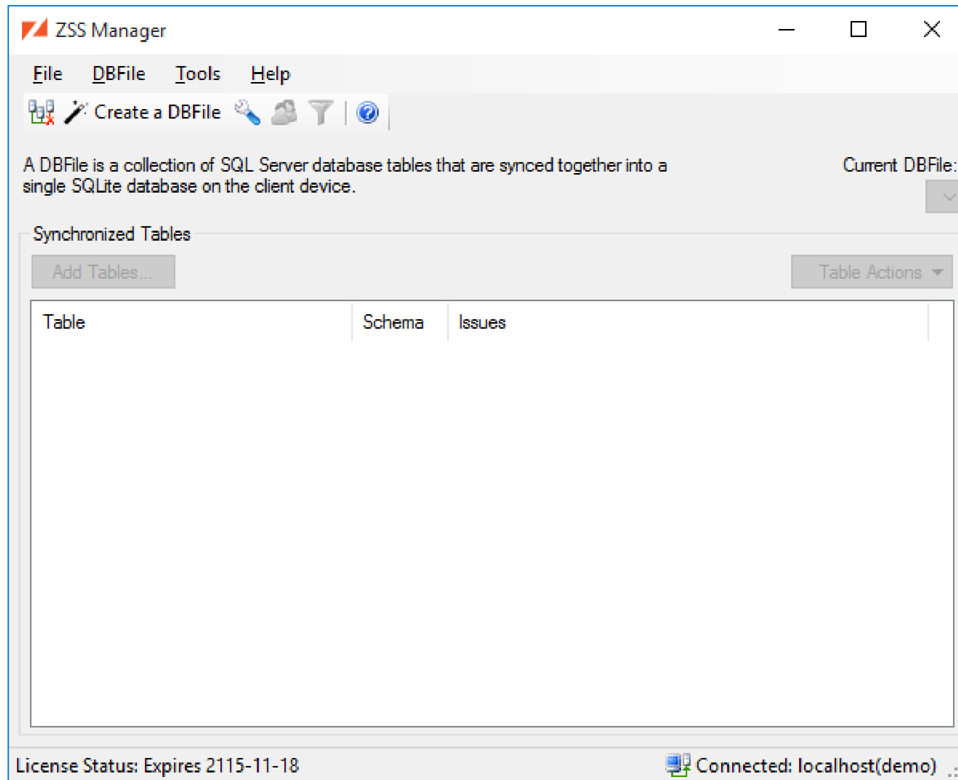
You can always connect to a different ZSS Server via the File / Change Zumero Server... menu. [1]

If this is your first time running ZSS Manager, you'll need to select a database once the connection has been opened. Choose a database from the drop-down list, then click Choose.

[1] See the Multiple Database Configurations section for more details on using ZSS Manager with multiple databases and multiple servers.



You're now greeted by the main window. This window is where you will configure settings for Zumero synchronization.

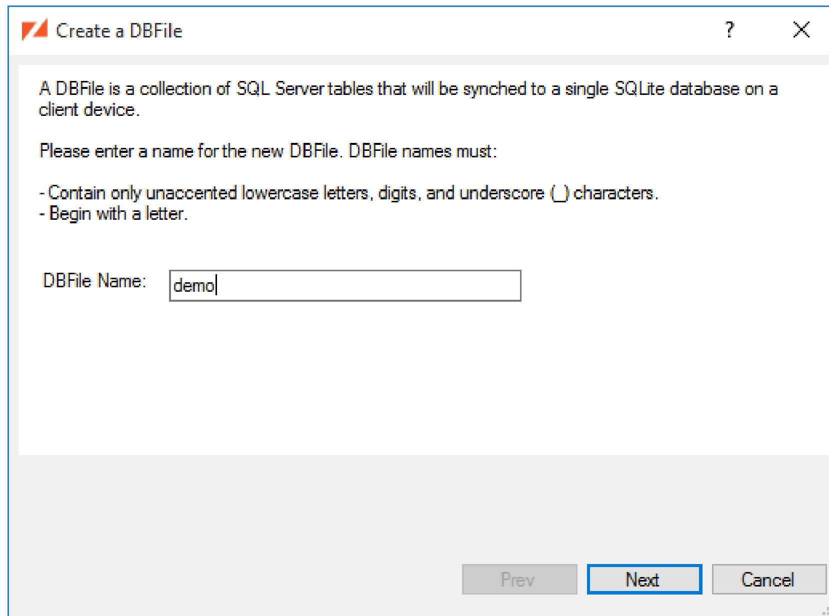


2.2. Create a DBFile

DBFiles are explained in more detail in the DBFile section. For the purposes of this quick start, understand that:

1. A DBFile is a container for tables, and
2. We need at least one DBFile to sync our data via Zumero.

Select Create a DBFile... from the DBFile menu (or click the Create a DBFile toolbar button.

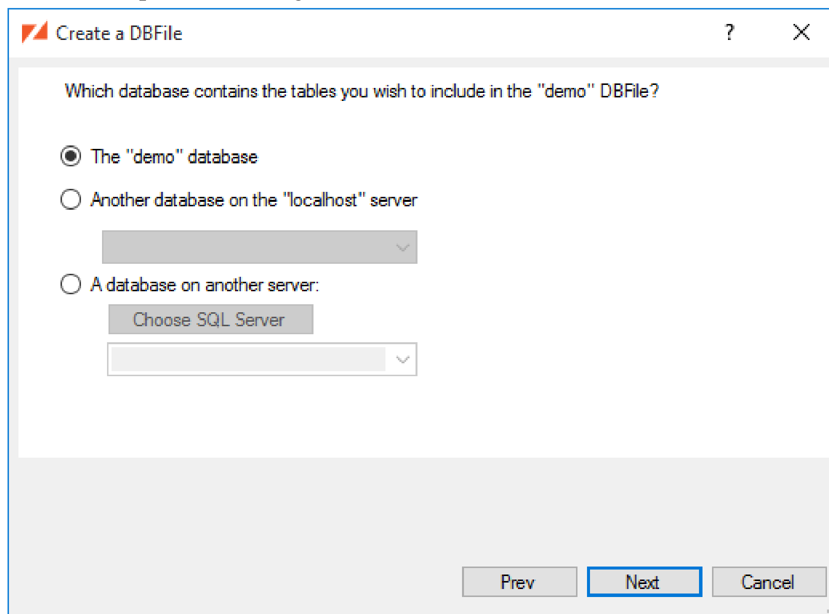


Once you've entered a valid DBFile name, the Next button becomes active.

2.2.1. Choosing the host database

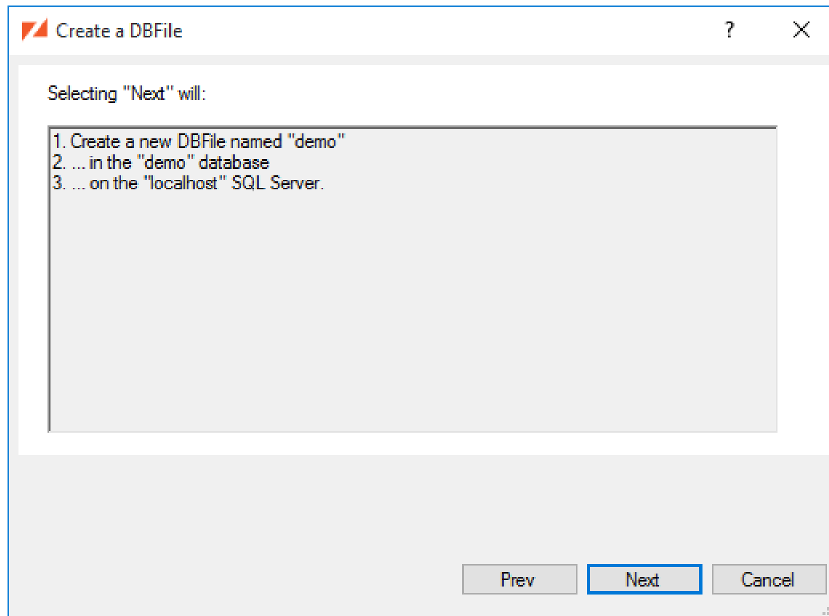
ZSS Manager needs to know what database contains the tables you'll be syncing. The data may reside in the same database as the ZSS Server configuration data, another database on the same server, or on another server altogether.

In this example, we're using data from the same database.

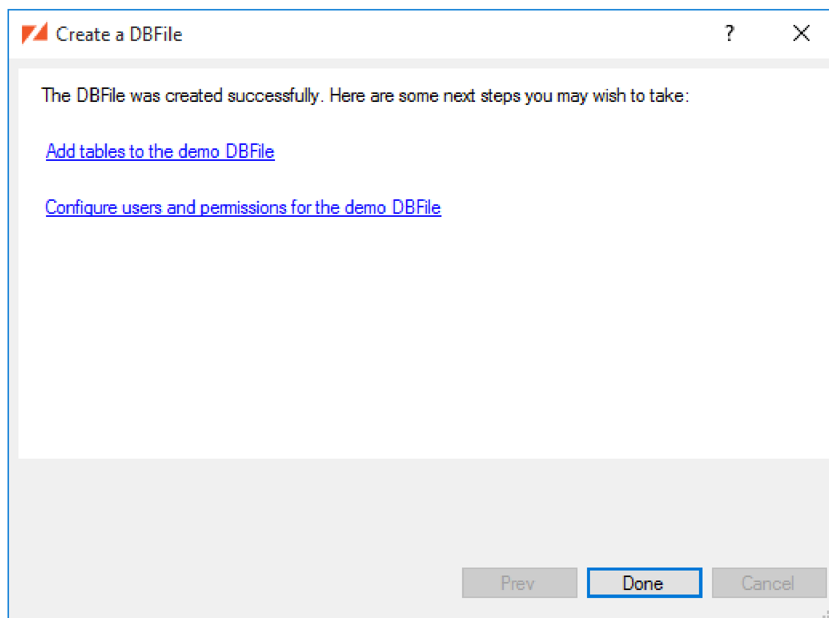


2.2.2. Create the DBFile

The DBFile wizard now reiterates what will happen when you click Next:

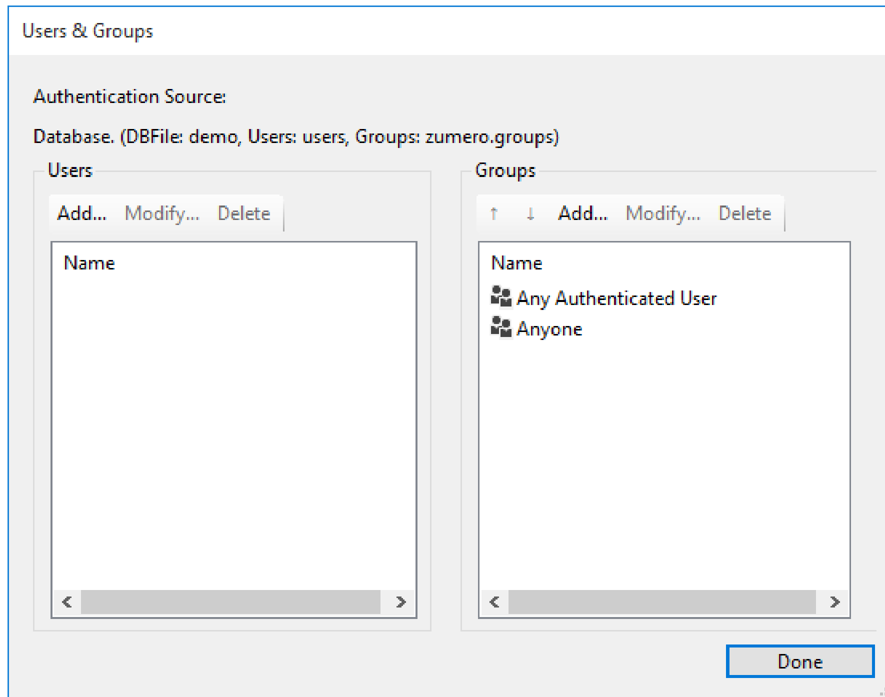


After the DBFile is created, we can next Add a Table or Create a User and Set Permissions:



2.3. Create a User and Set Permissions

Now that we have a DBFile, the User Management button is enabled in the main window. Click it to manage users and set permissions.



The first time you access user management for a given SQL Server database, you'll be prompted to create the users table and the groups table. Click OK to continue.

Below, we simply allow full access to everything, by anyone. For more details on setting up realistic users and permissions, see Authentication, Users, Groups, and Permissions.

To allow full access for anonymous sync requests, we double-click the Anyone group and set all 4 permissions to Allow:

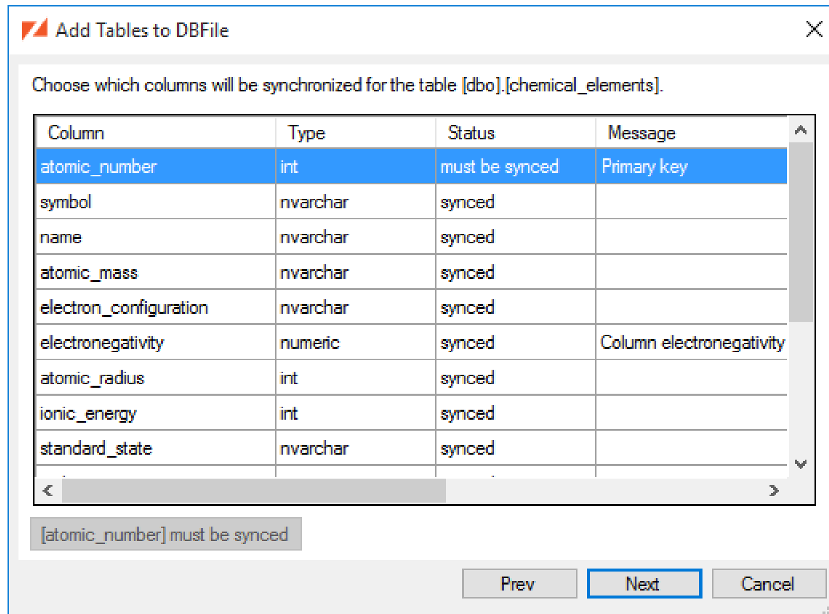
2.4. Add a Table to the DBFile

In the main window, click the Add a Table... button to select the tables that you want to synchronize.

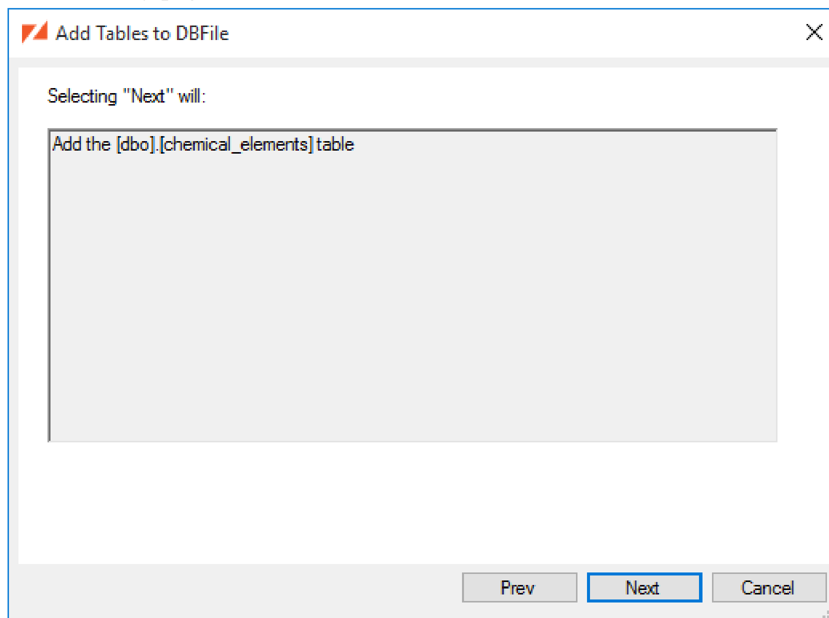
Select the checkbox beside each of the tables that you want to add to the DBFile.

The Table Considerations section of this document has more information about potential warnings and errors.

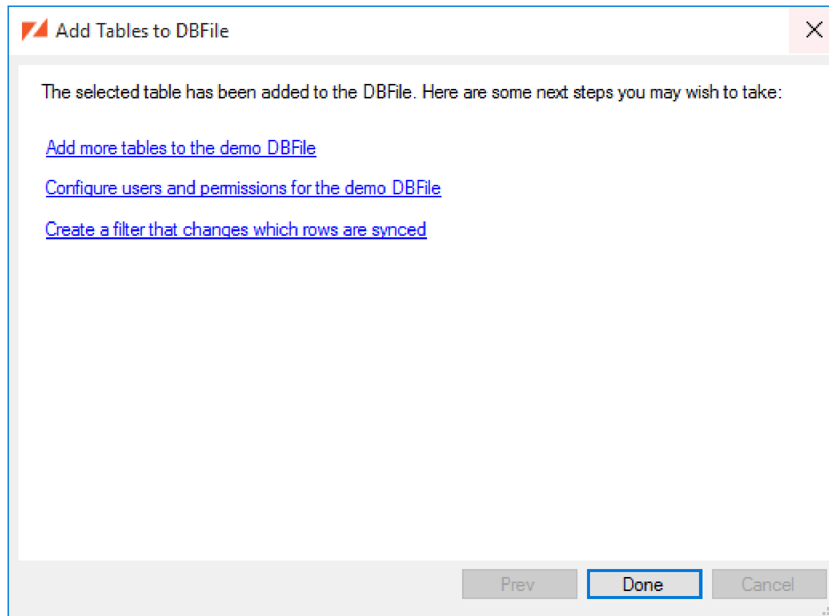
For each table that you have selected to add, you will be presented with a page to select which columns will be synced to the client.



The summary page will describe all of the tables that will be added to the DBFile.

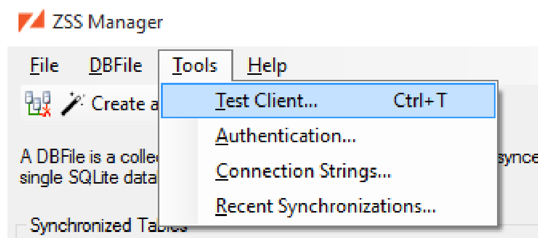


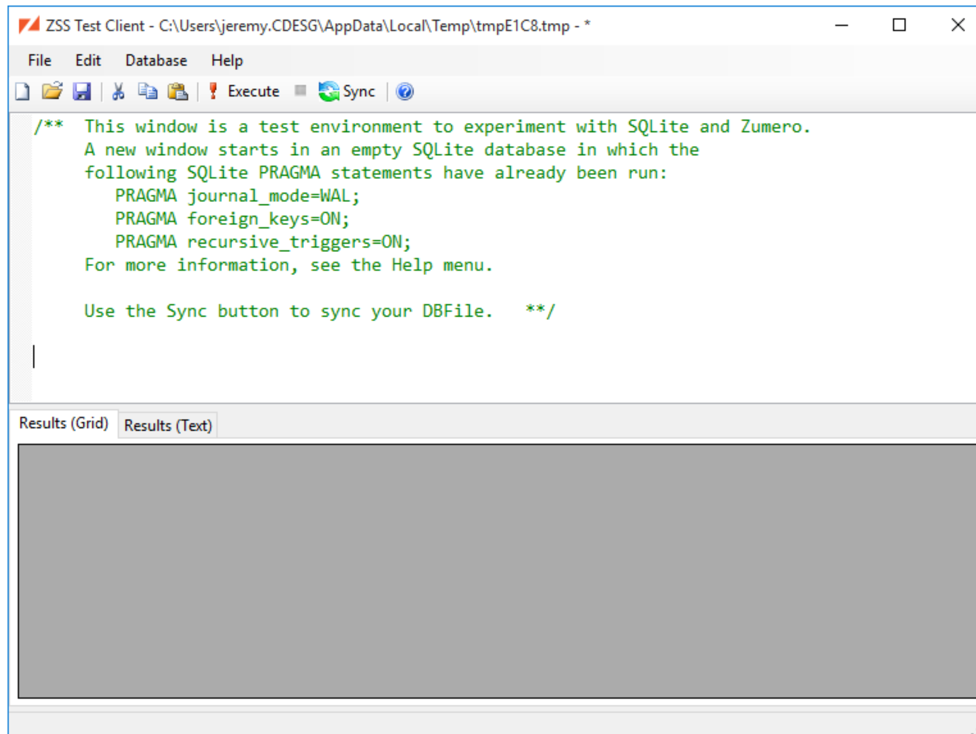
Click the Next button to add the tables.



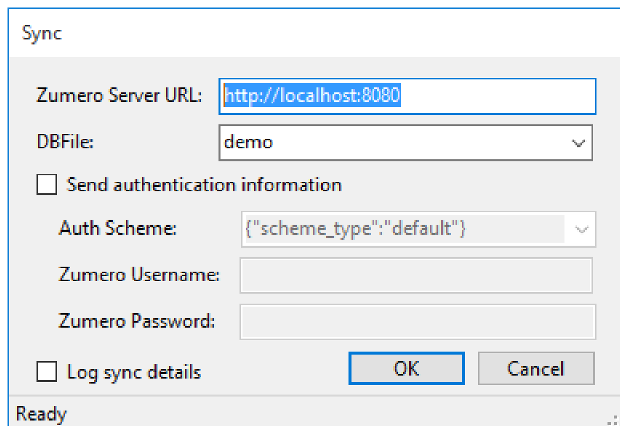
2.5. Ready to Sync!

A Zumero Server configured to connect to the database is now ready to sync DBFile *demo* which has a single table *chemical_elements*. Typically, the other end of the sync will be your mobile app. But ZSS Manager has a Test Client Window to help out:





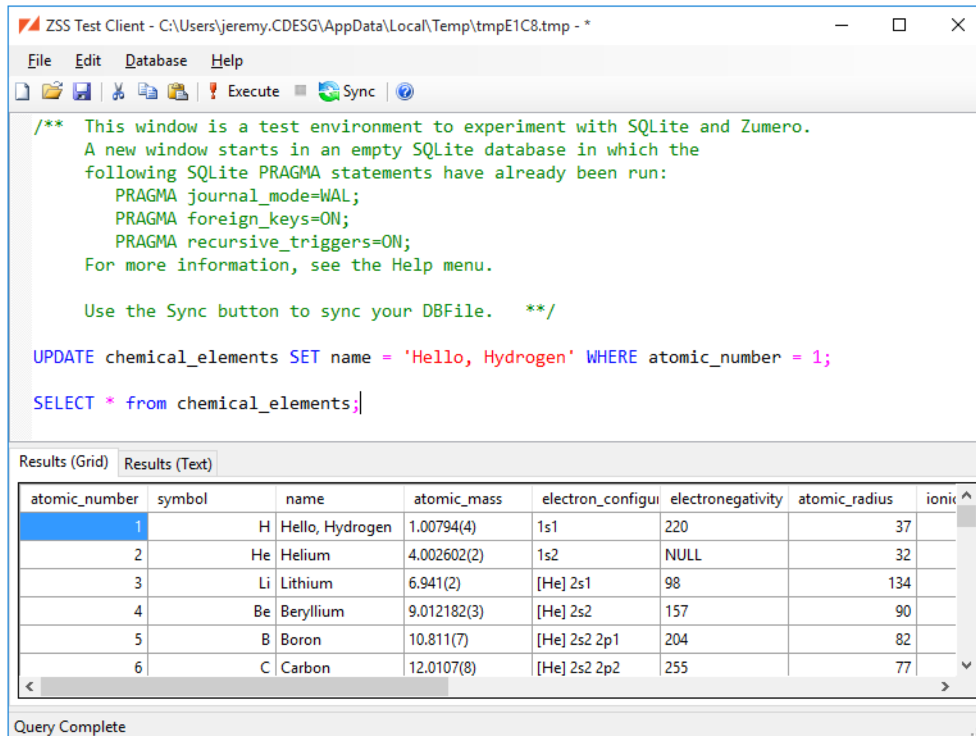
In this window, we start off working with an empty, temporary SQLite database. We can sync our test database into it by clicking the Sync button, which reveals the sync dialog:



Configuring the sync to match our setup:

- Server URL for a local server running on port 8080
- DBFile *demo* matches the DBFile we created.
- Send authentication information should be unchecked, because we set our test server's permissions to allow full anonymous access.

We click OK to perform the sync. Back in the test client's window, verify that the table was synced. Now let's make a change:



Now do another sync, the same way we did above. Examining the chemical_elements table in our SQL Server database, we can see the name for Hydrogen has changed.

3. DBFiles, Defined

A DBFile is a logical grouping of database tables on the server. A DBFile defined on the server with ZSS Manager corresponds to a SQLite database file on a ZSS Client.

- A SQL Server database can have one or more DBFiles, but a DBFile cannot span multiple SQL Server databases.
- A SQL Server table can belong to at most one DBFile.
- Permissions are scoped to DBFiles. A permission may apply to a single table within a DBFile or all tables in a DBFile.
- A DBFile is the smallest sync-able unit in Zumero. Clients specify a DBFile when calling sync, and all tables within it are synced.

Note

Permissions and filters can cause the contents of a DBFile to significantly differ based on the syncing user. On client devices, the SQLite database file corresponding to a Zumero DBFile should be accessed by only one Zumero user.

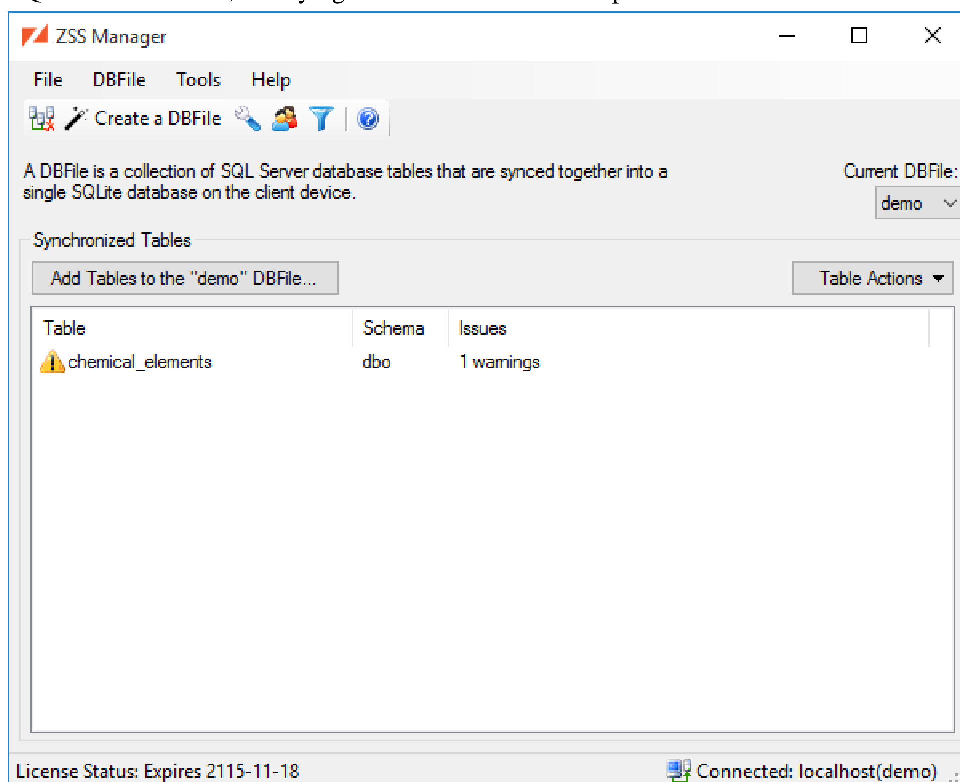
If multiple Zumero users will share a single device the app should maintain separate SQLite database files, opening the correct file based on the logged-in user.

4. The ZSS Manager Window

When starting ZSS Manager, you will be prompted to connect to a SQL Server, and the main window will be shown.

4.1. The selected DBFile

In the top right of the main window, you will see a combobox listing all of the DBFiles that have been created in this primary database. If the DBFile contains tables from a different SQL Server database or SQL Server instance, clarifying details will be shown in parenthesis after the DBFile name.



If you select a DBFile that contains tables from a different SQL Server instance, you will be prompted to supply connection details.

4.2. Add a Table to the Current DBFile

This wizard adds tables to the current DBFile via these steps:

- Select the tables that will be added to the DBFile.
- For each of the selected tables, a wizard page will give you the opportunity to select the columns to sync to the client database.
- Show a summary of the tables and columns that will be synchronized.
- After the database changes are complete, the final page will offer a few possible next steps..

4.3. The Synchronized Tables list

This list shows all of the tables that have been added to the selected DBFile. In addition, this list will show any errors and warnings for the table. The context menu on this list lets you:

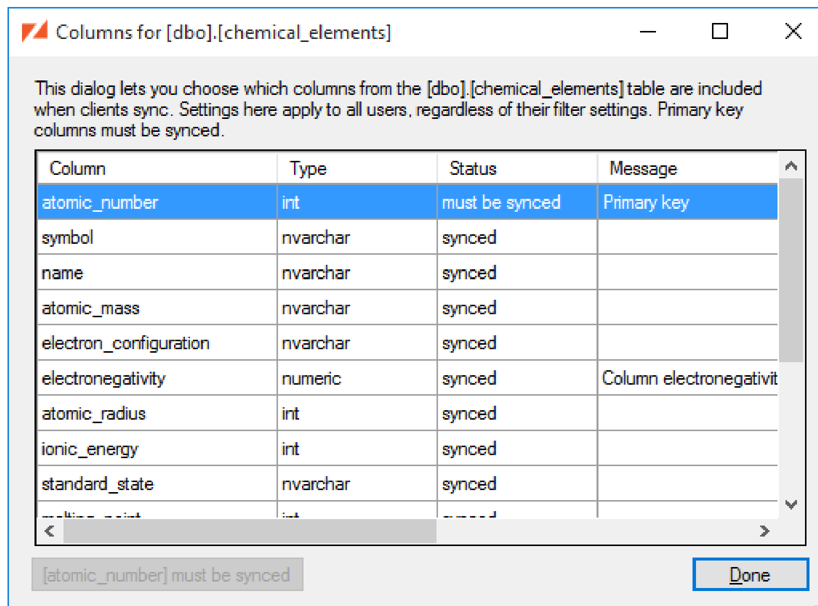
1. Choose Columns to synchronize

2. Configure the Conflict Rules
3. Stop Syncing the table
4. View Warnings and Errors

4.3.1. Choose Columns

If your synchronized tables contain some columns that you don't want to be synced to client databases, you can select the synced columns during the Add Tables to DBFile wizard, or by selecting the table in the ZSS Manager window and picking the Choose Columns entry from the context menu.

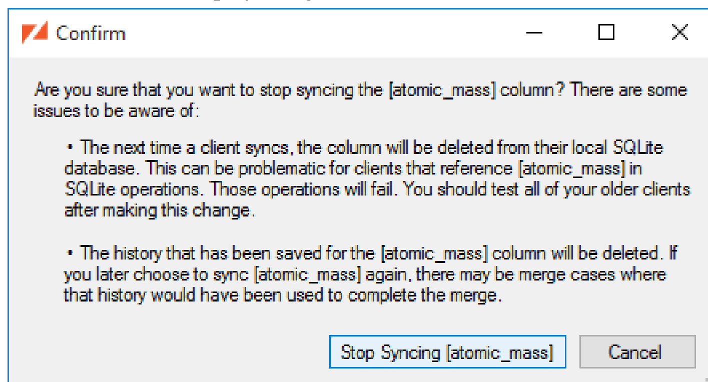
Note that when new columns are added to a table that is already in the DBFile, they are *not* synchronized by default. Use the Choose Columns dialog if you wish to include the new column in the client database.



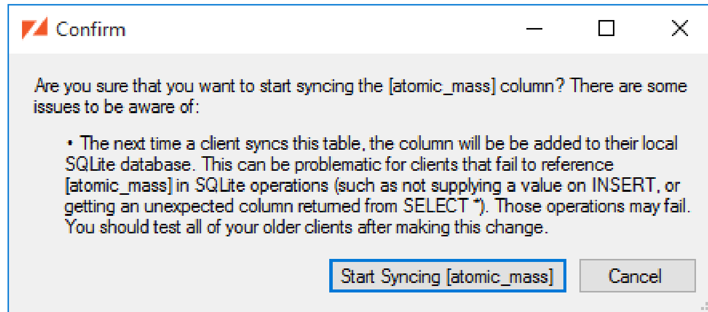
Here we see that:

1. The `atomic_number` column *must* be synced, since it is the table's primary key.
2. The remaining columns are currently part of the sync, but could be excluded.

If we choose to Stop syncing [atomic_mass], we'll be warned of the consequences:



Afterwards, the column is marked as "not synced", and when it's selected we have the option to Start syncing [atomic_mass], just as we would with any unsynced column.



4.3.2. Conflict Rules

This dialog gives you the opportunity to configure the Conflict Resolution settings.

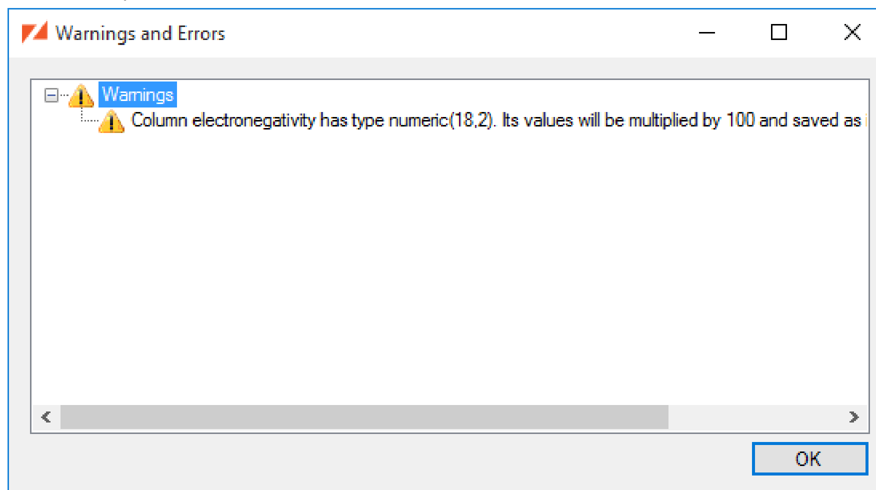
4.3.3. Stop Syncing a Table

If you need to stop syncing a table, you select the table in the ZSS Manager window and pick the "Stop Syncing" entry from the context menu. It should be noted, however, that this is destructive and shouldn't be done on live applications with Zumero clients. Any unsynced changes on client databases will be lost, even if the table is re-added to the DBFile.

- Your data is left alone.
- All Zumero housekeeping data is deleted and the housekeeping tables are dropped.
- Clients will drop their corresponding table on the next sync.

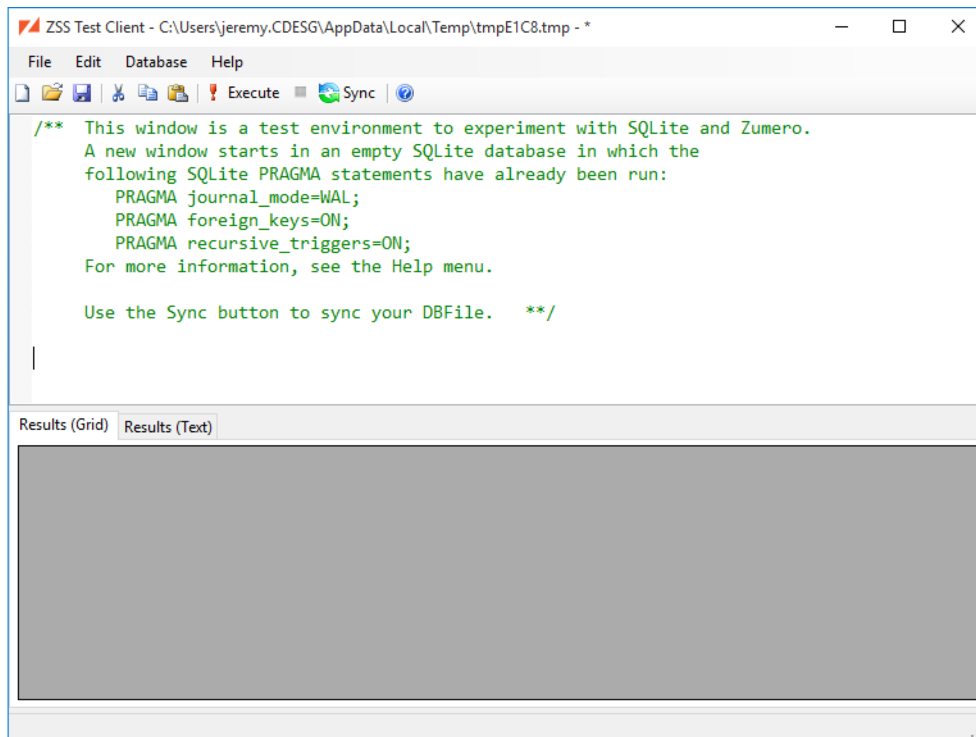
4.3.4. Table Warnings and Errors

The Synchronized Tables list will allow you to see a list of the Warnings and Errors that may impact the table's synchronization. To get a clear view of the issues, select the table and choose View Warnings and Errors entry from the context menu.



Selecting one of the errors or issues, you can choose Copy from the context menu to copy the text to the clipboard.

4.4. The Test Client Window

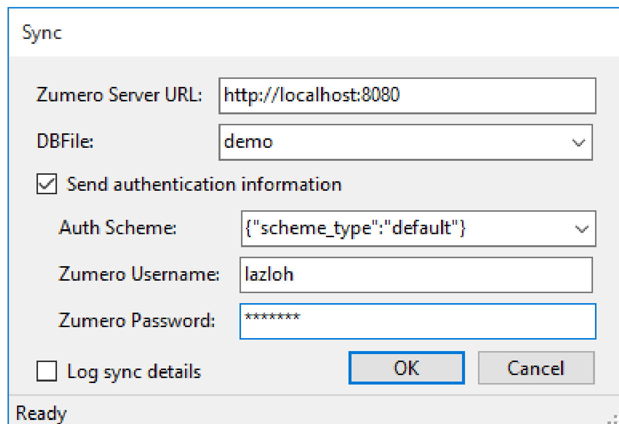


The Test Client window gives you an environment to experiment with the capabilities of SQLite, and also test using Zumero to sync changes between the client and server.

When you open the Test Client window, you will automatically be connected to a new, temporary SQLite database. This database will be deleted when you close the window. If you need to save the SQLite database between sessions, use the Open SQLite Database and Save Database As items in the File menu.

4.4.1. Syncing Changes

Pressing the Sync button in the Test Client window, the Sync Dialog will come up.



There are two syncing modes, authenticated and anonymous. Authenticated syncs validate against a users table. In order for anonymous syncs to succeed, you must enable anonymous access to your DBFile.

4.4.1.1. Logging Sync Details

The **Log sync details** checkbox causes the Test Client to sync with `sync_details` enabled (see the ZSS Client API Documentation for details). After syncing, you can query the `zumero_sync...` tables in the test client window, in addition to your data tables.

4.4.2. Enabling Important SQLite Features

The Test Client Window enables two SQLite features that are off by default. When you open or create a SQLite database, it executes these pragma statements. When writing your own client, you almost certainly want to do the same.

```
PRAGMA foreign_keys = ON;
PRAGMA recursive_triggers = ON;
```

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

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

When creating a new SQLite database, the Test Client Window also enables `WAL` (Write-Ahead Logging) mode.

```
PRAGMA journal_mode = WAL;
```

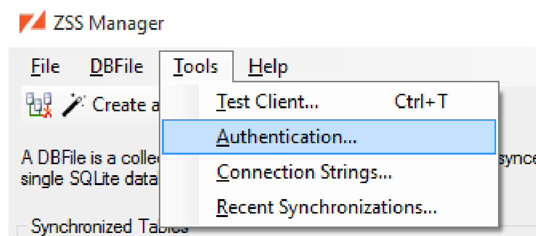
WAL mode is strongly recommended for improved concurrency. See Write-Ahead Logging for more information.

5. Customizing Synchronizations

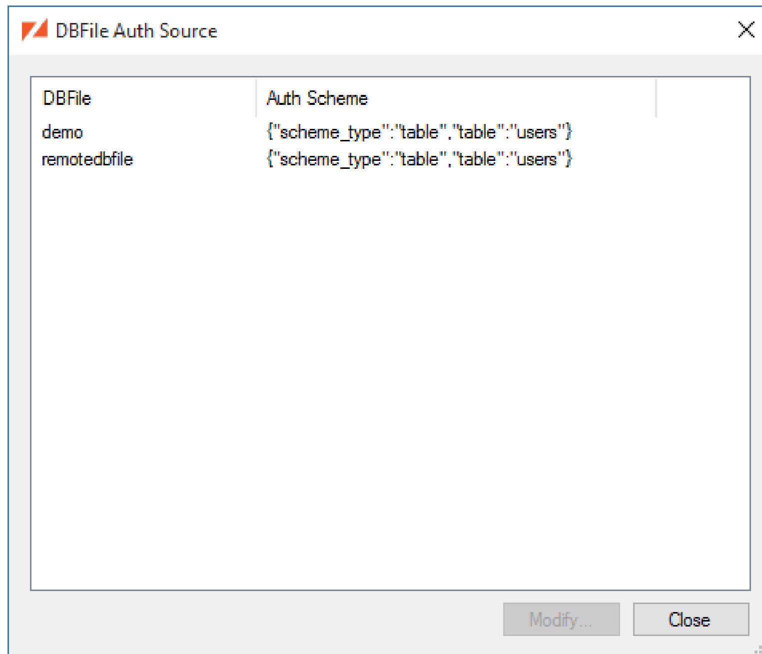
5.1. Authentication and Permissions

Zumero for SQL Server supports two authentication sources. The first and default authentication source is called Database Authentication and it stores users and groups in the database. Secondly, Zumero for SQL Server can use your existing Active Directory as an authentication source.

Each DBFile can have it’s own unique authentication source and by default this source is Database Authentication. If you wish the change the authentication source for a DBFile select DBFile Authentication Source... from the DBFiles menu.



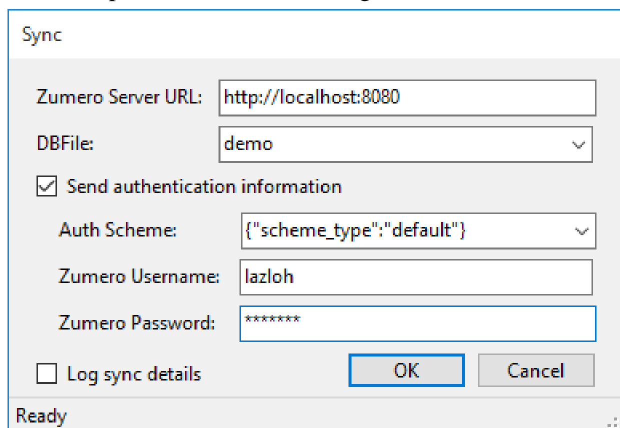
The DBFile Authentication Source dialog allows you to configure the default authentication source for each DBFile. The specific options for each authentication source are covered in more detail in later sections of this guide.



When syncing with the server, clients must provide an authentication scheme string when calling sync(). When the DBFile default is desired, the following scheme string should be used: [2]

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

An Example from the ZSS Manager Test Client Window:



From client code, it would look like this:

```
ZumeroClient.Sync(localFilePath, cipherKey, serverUrl, dbfileName, "{\"scheme_type\":
\"default\"}", username, password);
```

Zumero for SQL Server does not provide features for managing users, groups, and passwords from a Zumero client. The user and group tables are never synced to a client. User, group, and password management occur on the server. ZSS Manager has an interface for managing users and tries to cooperate with existing applications.

First we'll describe how to set up users and groups in the database and authenticate them.

[2]For information on other scheme types, see Non-Default Authentication Schemes

5.1.1. Database Authentication via ZSS Manager

Database Authentication is Zumero's default authentication method. This authentication method stores the users and groups in the database. When first configuring Database Authentication, ZSS Manager will ask if it can create the default backing tables for Database Authentication. If you allow this, they will be created as follows:

```
CREATE TABLE [zumero].[users]
 (
   [id]    int IDENTITY(1,1) PRIMARY KEY,
   [name] nvarchar(100) UNIQUE NOT NULL,
   [pass] nvarchar(100) NOT NULL
 )

 CREATE TABLE [zumero].[groups]
 (
   [username] nvarchar(100) NOT NULL,
   [groupname] nvarchar(100) NOT NULL
 )
```

Users and Groups that you created via the Users and Groups dialog will be added to these tables. The passwords are encrypted with bcrypt. [3]

Database Authentication has a couple of configurable settings. To access these settings use the following steps:

- Select “DBFile Authentication Source…” from the DBFiles menu.
- Select your DBFile from the list and click the “Modify” button.
- If not already selected, change the “Authentication Source” pulldown to “Database”.

Select Authentication Source

Authentication Source:
Database

Locate Via DBFile:
demo
The users table will be located in the database hosting this DBFile.

Users Table:
zumero.users

Groups Table:
Located in zumero.groups

Scheme:
{"scheme_type":"table","table":"users"}
Scheme string to use when syncing

OK Cancel

[3] http://en.wikipedia.org/wiki/Bcrypt

- `Locate Via DBFile:` This option specifies the database in which the users will be stored. If you have a Multiple Database Configuration you can select the DBFile that corresponds to another SQL Server Database in which user and groups have already been configured.
- `Users Table:`This option specified the name of the table the users will be stored in. If you have an existing table containing user information you can change this setting to that table. There are several caveats when doing this, so read the Database Authentication with Your Own Table or View below.

5.1.2. Database Authentication with Your Own Table or View

If you already have users and passwords stored elsewhere in your database, ZSS Manager and the Zumero Server can use them, with some caveats:

- Passwords must be either plain text or bcrypt hashes. (A mix of both in the same table will work.)
- You must create a view, zumero.users, with columns [id], [name], and [pass].
- If you want to use ZSS Manager to alter users and passwords, the view you create must have insert and update triggers that handle the name and pass columns.
- If you change a user's password with ZSS Manager, it will be encrypted with bcrypt. This could cause problems for other applications if they're not expecting passwords in this format.
- If you wish to use existing groups, you must create a view, zumero.groups, with columns username and groupname. This table is used to determine group membership.
- It is strongly recommended that the `[id]` and `[name]` columns be unique. Non-unique users will lead to unpredictable results, or to failed sync.

If you have existing users and passwords in your database but passwords are neither plain text nor bcrypt hashes, you can still use ZSS Manager to set permissions, which is recommended. Create the zumero.users view with id, name and pass columns, but write the insert/update triggers such that only the name column gets updated. This will allow ZSS Manager to change user names and manage permissions, but any password changes are silently ignored. You can manage passwords outside Zumero, as you did before.

Note

It is possible to use any table or view name for client authentication. Simply specify a different table when specifying the authentication source for your DBFile.

5.1.3. Active Directory Authentication

Zumero for SQL Server can also make use of the users and groups in your existing Active Directory setup. This authentication source has the following requirement:

- The machine running the Zumero server and the machine used to run the ZSS Manager application must be members of your Active Directory domain.

Next, you must configure your DBFile to use Active Directory for authentication.

- Select DBFile Authentication Source… from the DBFiles menu.
- Select your DBFile from the list and click the Modify button.
- Change the Authentication Source pulldown to “Active Directory”.
- Enter the windows domain in the Active Directory Domain field.

Note

Use the fully qualified domain name for your domain, do not use the Windows 2000 compatible version.

When using Active Directory as the authentication source, the Users & Groups dialog will not be populated with all of the users and groups from Active Directory. If you wish to add permissions for a user or group you will need to use the “Add…” button to create an ACL entry for the specified Active Directory object.

No changes are necessary for Zumero clients, they should send the username and password as they would with Database Authentication. There is a slight difference on the server that is worth noting. At authentication time the username is converted to the User Principal Name [4] (<username>@<domain>). This is most notable when using the {{ZUMERO_USER_NAME}} token with filters.

5.1.4. Built-In Groups

5.1.4.1. Anyone

Permissions granted to Anyone apply to all requests: authenticated or unauthenticated.

Note that this is different from a sync request that *fails* authentication by providing invalid authentication parameters. Failed authentication requests never have data access of any kind. The only way to explicitly use the Anyone user is to pass null for all three authentication parameters, as we did in the Quick Start.

5.1.4.2. Any Authenticated User

You may want to set common permissions for any user who successfully authenticates. The Any Authenticated User allows you to do that.

When using Active Directory as your authentication source, it is recommended for security reasons that permissions for Any Authenticated User remain set to deny. Instead an AD Group should be created that includes all of the users you wish to have access to the Zumero sync service.

5.1.5. Permissions

Zumero checks permissions from most to least specific. When an authenticated user Winifred syncs, permissions are checked in this order:

- the Winifred user
- Any group of which Winifred is a member
- the Any Authenticated User group
- the Anyone group

As soon as either Allow or Deny is found, the checking stops. If no permission is set, the checking continues. If no match is found the sync is denied.

Note

In the absence of any permissions data, a server is completely locked down: nobody has access to anything. This is nice and secure, but a bit hostile to newbies. Be sure to set up a user with rights when getting started.

Zumero syncs are atomic: they succeed or fail as a single unit. This comes into play when permissions are checked. Say Winifred has modified existing rows and added some new ones. When she syncs, she must have permissions to do both, or the entire sync will be denied.

Zumero for SQL Server has four permissions:

[4] https://msdn.microsoft.com/en-us/library/ms680857(v=vs.85).aspx

5.1.5.1. Pull

Pull is analogous to a read permission. The way to read data from the Zumero Server is to pull it. How that data is accessed on a client application is up to its developer.

The scope of the Pull permission is the entire DBFile. You can pull all of the tables in a DBFile, or none of them.

5.1.5.2. Add

A user with Add permission can add new rows. More precisely, when a client having new rows does a sync, this permission is necessary for that sync request to succeed.

Add may be defined for an entire DBFile, or for a specific table within a DBFile. It's possible, for example, to let Winifred add new rows to the Customers table but not the Sales table.

5.1.5.3. Modify

The Modify permission allows changes to existing rows.

Like Add, Modify can be set for a specific table or an entire DBFile.

5.1.5.4. Delete

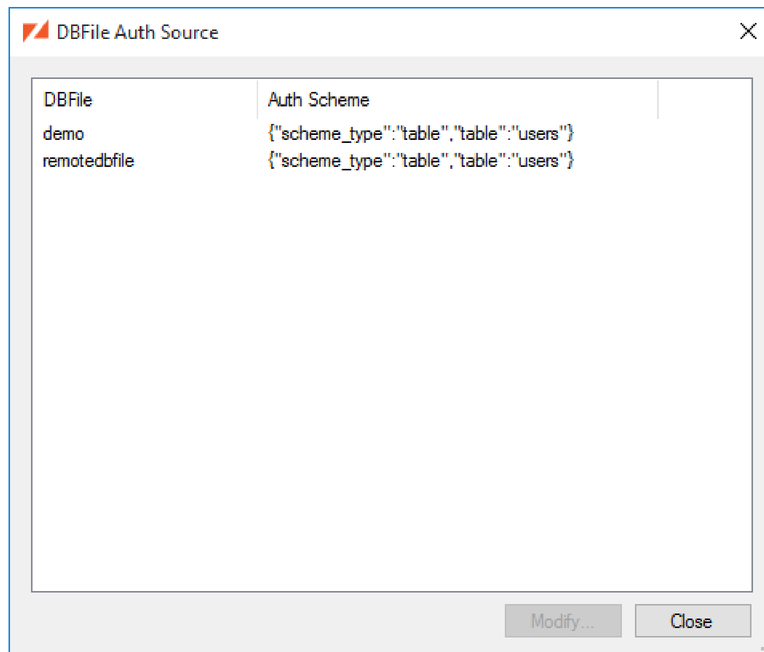
The Delete permission allows deleting existing rows.

Like Add and Modify, Delete can be set for a specific table or an entire DBFile.

5.1.6. Setting up Permissions through Another DBFile (table2 authentication)

By default, authentication is performed by looking up users in the `zumero.users` table *in the same databases* as the DBFile's data. You may wish, however, to authenticate against a table stored elsewhere -- e.g. when sharing a user table among DBFiles in different databases.

In this case, you'll still need a table in one of those databases. Access to "remote" user tables is always done by looking up the connection information for a DBFile which lives in the user table's database. In the Authentication... dialog, you'd select the dbfile and click the Modify button



...which brings up the **Select Authentication Source** dialog:

Here, we've selected the `remotedbfile` DBFile as the link to our user data. The **Table** drop-down now lists user tables found in the database containing `remotedbfile`. We're sticking with that database's `zumero.users` for now.

The **Scheme** box contains an authorization scheme string, ready to be used as the `scheme` parameter in a client application's `sync()` call. Copy and paste this code to avoid manually creating a matching scheme string.

On return to the **Users and Permissions** dialog, and beginning to Add a user permission entry, we now see the users from that remote users table:

5.1.7. Non-Default Authentication Schemes

As a rule, it's best to use the `default` authentication scheme. There are two main advantages:

1. It leaves authentication details in the hands of the server, not the client.
2. It allows the authentication source to *change* (on the server) without affecting sync, configuration, or code on the client side.

If necessary (primarily for backwards-compatibility with older versions of Zumero), you can explicitly specify `table` or `table2` authentication.

5.1.7.1. table Authentication

This is the simplest form of table-based authentication. It uses a named table (or view) in the same SQL Server database as your application data, with `id`, `name` and `pass` columns as described in Database Authentication with Your Own Table or View.

The standard Zumero authentication table (`zumero.users`) would be specified by:

```
{"scheme_type":"table","table":"users"}
```

In most situations, this will be the same authentication source used by `{"scheme_type":"default"}`.

5.1.7.2. table2 Authentication

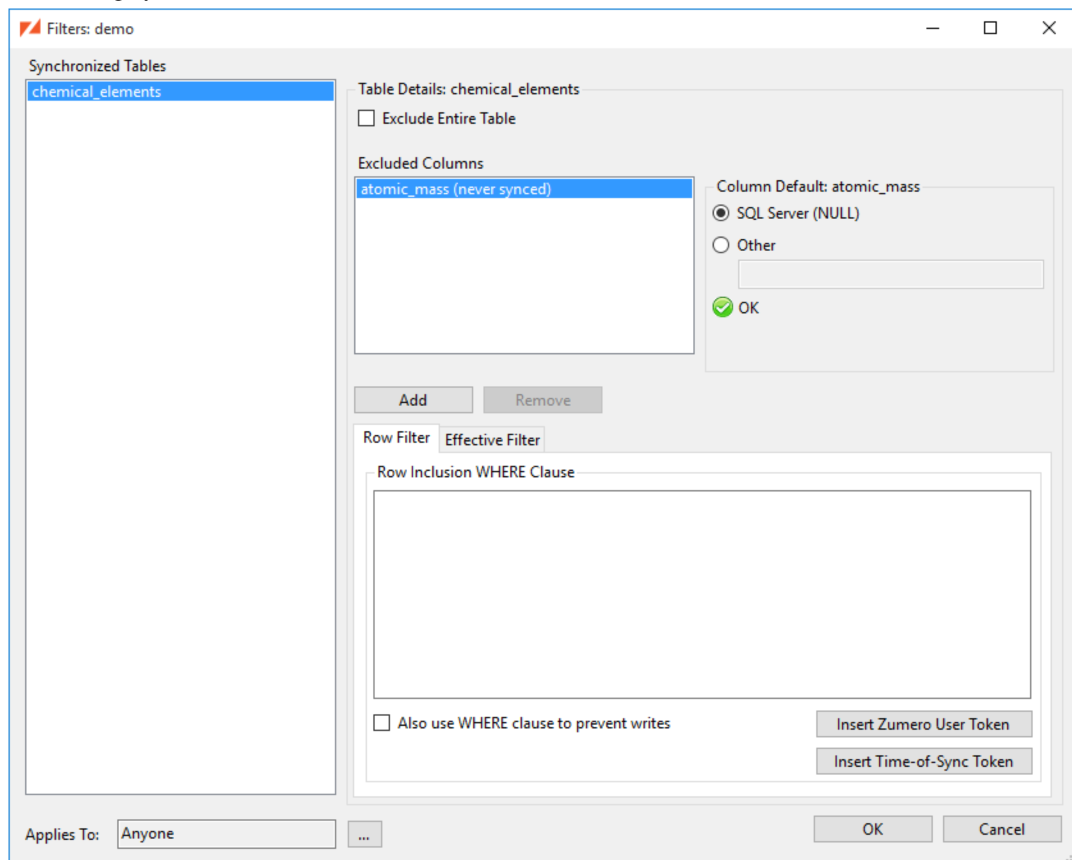
In cases where multiple DBFiles (across multiple databases) will share a single authentication source, you can use `table2` authentication. This works just like table authentication, but adds the name of another DBFile. The authentication table will be found in *that* DBFile's database.

If you were going to authentication against the `zumero.users` table in the database hosting the *central* DBFile, you'd do so via:

```
{"dbfile":"central","scheme_type":"table2","table":"users"}
```

5.2. Filters

ZSS supports filtering tables by row and column based on the syncing user. Filters can be used to reduce the storage size and bandwidth requirements of an app. They can also be used to prevent sensitive data from being synced to mobile devices.



Tables may be filtered in the following ways:

- A synchronized table may be completely excluded.
- Columns within a synchronized table may be excluded.
- Rows within a synchronized table may be excluded.

When a row in a synchronized table is excluded, this automatically causes all rows which reference the excluded row to also be excluded. This feature of Zumero allows for easy filtering of multiple tables and prevents foreign key constraint violations on the client.

5.2.1. Filter Scopes

When choosing a filter to apply, the Zumero server takes these issues into account:

- Like permissions, filters are checked from most to least specific. Only the first matching filter is applied. User-specific filters will take priority over group filters. Group filters will take priority over "Any Authenticated User" and "Anyone" filters. An "Any Authenticated User" filter will take priority over an "Anyone" filter.
- A filter exists within the scope of a DBFile. It may only filter tables that are synchronized with that DBFile.
- You may create filters for the *Anyone* and *Any Authenticated* users.
- A user may appear in only one User filter. If a user appears in a User filter and a Group filter, the group filter will be ignored for that user.
- Filters stand alone. It's not possible to create a filter based on another such that the result is a combination of the two. If you have more than one filter, you may need to duplicate some settings.

5.2.2. Editing a Filter

The Edit Filter dialog contains all of the settings for a single filter. The list on the left shows all of the tables that are synchronized in the current DBFile. If any of the table names are bold, they already have some filter settings applied.

5.2.2.1. Excluding tables and columns

If you need to have some users sync a table or column, while other users do not sync that table or column to their local client database, you will use filters to accomplish that.

- Excluding a table in a filter does not impact the change-tracking applied to the SQL Server host table. If you don't want any users to sync the table, choose to Stop Syncing it instead.
- Excluding a column in a filter does not impact the change-tracking applied to the SQL Server host table. If you don't want any users to sync the column, use the Choose Columns dialog to stop syncing it instead.

The Excluded Columns list will show columns that are specifically excluded by this filter, as well as any that are excluded because of settings in the Choose Columns dialog.

5.2.2.1.1. Column defaults

If you have excluded a column via a filter, or using the Choose Columns dialog, you can provide a default value using the Edit Filter dialog. This will be the value used when inserting any rows that were initially inserted on a client SQLite database. You may choose either the SQL Server default value, or an explicitly specified value.

5.2.2.2. Filtering rows

A filter can reduce the number of rows synced to a client database in two ways.

- **Foreign Key exclusions:** All foreign keys will be enforced for a given table. For example, Table [child] has a foreign key to Table [parent]. If a filter excludes rows from [parent], any rows in [child] which point to excluded [parent] rows will also be excluded. You can use the Effective Filter tab to see all of the foreign key relationships which will be enforced.
- **WHERE clause inclusion:** Explicitly entering a WHERE clause for a table allows you to choose which rows are synced to the client.

5.2.2.2.1. Row Inclusion WHERE Clause

5.2.2.2.1.1. ZUMERO_USER_NAME

When this query is executed by the Zumero server for a syncing client, the `{{{ZUMERO_USER_NAME}}}` token is replaced by the syncing user's `name`. That makes it useful for Any Authenticated User or Group filters. That also means it will not work on filters defined for the Anyone user since the syncing client could be syncing anonymously

5.2.2.2.1.2. TIME_OF_SYNC

In addition to the `{{{ZUMERO_USER_NAME}}}` token mentioned in the previous example, there is also a `{{{TIME_OF_SYNC}}}` token. During a sync, this token is replaced with a datetime object representing the moment when the sync happened. It is the same as using any of the following SQL Server functions, but allows Zumero to use much less bandwidth during sync and also eliminates race conditions that could lead to sync failures:

- `GETUTCDATE()`
- `GETDATE()`
- `CURRENT_TIMESTAMP`

5.2.2.2.1.3. z$this

The alias `z$this` can be used in the WHERE clause to refer to the current table.

5.2.2.2.1.4. Also use WHERE clause to prevent writes

Row Inclusion WHERE clauses affect reads, meaning which rows are sent from the server to the client. By default, they do not prevent writes — meaning, what kinds of inserts, updates, or deletes a client can push to the server. To affect writes too, select the "Also use WHERE clause to prevent writes" check box. In this way, you can use the WHERE clause to configure write permissions at a row-level granularity

If the "Also use WHERE clause to prevent writes" option is selected, all of the following operations are prevented, and will cause the client to receive a "permission denied" error:

- An INSERT of a row that does not match the Row Inclusion WHERE clause.
- An UPDATE that causes a row that matches the WHERE clause to no longer match it.
- An UPDATE of a row that does not match the WHERE clause (even if it causes the row to become a match).
- A DELETE of a row that does not match the WHERE clause.

Note that only the first two operations are even possible under normal circumstances. The last two can only occur when there are existing clients that have already synced without the filter, and the filter is going to be applied starting with their next sync.

5.2.2.2.2. Sample WHERE Clauses

Here are some basic scenarios that you may encounter when configuring a WHERE clause.

A static filter.

```
WHERE region = 'Asia'
```

Filter a table based on matching a column to the Zumero user performing the sync.

```
WHERE name = '{{{ZUMERO_USER_NAME}}}'
```

Join to a table based on the Zumero user performing the sync. This may be redundant if the other table is already filtered. Check the Effective Filter tab.

```
WHERE userid in (SELECT id FROM userstable WHERE name = '{{{ZUMERO_USER_NAME}}}')
```

A slightly more complicated join

```
WHERE regionid in (SELECT id FROM userregions WHERE userid in (SELECT id FROM
userstable WHERE name = '{{{ZUMERO_USER_NAME}}}'))
```

Some tables are write-only, meaning rows can be inserted on the client, but no server rows should ever be synced to client databases. You can accomplish this by using a WHERE clause that always matches zero rows.

```
WHERE 0 = 1
```

5.3. Constraint Violations and Conflict Resolution

When a client invokes `zumero_sync()`, it sends a package to the server. This package contains all changes to the client's copy of the database which have been made since the last time that client was synchronized.

If the server's copy of the database has not been changed in the meantime by other clients, the changes contained in the incoming package will simply be added to the database.

However, if some other client has already sent a package of changes, it is possible that there will be conflicts. The Zumero server is responsible for automatically resolving these conflicts using a set of rules. These rules can be customized.

There are two basic kinds of conflicts that can happen:

- *Row conflicts*. The incoming package is trying to update or delete a row which has already been updated or deleted by another client. By default, row conflicts will not cause the client's sync to be rejected.
- *Constraint violation conflicts*. The incoming package causes the violation of a SQL constraint because of a change that has already been received from another client. By default, any constraint violation conflicts will cause the client's sync to be rejected.

When a client's sync is rejected because of a row conflict or constraint violation of any kind, `zumero_sync()` will return one of the following error codes.

#define name	integer result code	zumero_errstr() string
ZUMERO_CONSTRAINT_ VIOLATION	2760	constraint_violation
ZUMERO_UNIQUE_ CONSTRAINT_VIOLATION	3016	unique_constraint_violation
ZUMERO_CHECK_ CONSTRAINT_VIOLATION	3272	check_constraint_violation
ZUMERO_FOREIGN_KEY_ CONSTRAINT_VIOLATION	3528	foreign_key_constraint_violation
ZUMERO_SYNC_REJECT_ BY_RULE	3784	package_rejected

Any of these error codes mean that the server refused to accept the client's package because one or more database operations contained in the package were problematic. These are the only error codes used for this purpose.

Note

In rare occasions, Zumero app developers have observed SQLite's constraint violation error code (19) arise during a call to sync(). Prior to ZSS version 2.0, this usually indicated that Zumero had

been configured with an unsupported Row Inclusion WHERE clause. Now that Zumero supports arbitrary WHERE clauses, a SQLite error 19 during sync usually indicates the presence of a bug in Zumero. Contact Zumero support if such an error is ever returned by `zumero_sync()`.

The Conflict Rules dialog can be used to adjust, on a per-table basis, how certain conflict situations are handled. It can be opened from the context menu in the main ZSS Manager window. The various different conflict situations and how they can be handled are detailed below.

5.3.1. Attempting to modify a deleted row.

The incoming package is trying to modify a row which has been deleted.

Example 1. A sample modify-after-delete conflict

Consider a case where the table in question has two columns: key, and name. Two clients, John and Paul are both synced with the server, and have the latest contents.

- John deletes a row.

```
DELETE FROM table WHERE key = 4
```

- Paul modifies the same row.

```
UPDATE table SET name='bar' WHERE key = 4
```

- John syncs his change to the server. The sync is successful. The row is deleted.
- Paul syncs his change to the server. The server must use the configured rule to resolve the conflict.

The possible resolutions are:

- Accept the incoming modify operation. The row will be restored, with Paul's values key=4 and name='bar'. *This is the default action.* It is sometimes referred to as an "undelete".
- Ignore the incoming modify operation. The row will remain deleted.
- Reject all changes from Paul. Paul will not be able to perform any sync operations that contain this conflict. His client must call zumero_quarantine_since_last_sync() to remove the modify operation.

5.3.2. Attempting to delete a modified row.

The incoming package is trying to delete a row which has been modified.

Example 2. A sample delete-after-modify conflict

Consider a case where the table in question has two columns: key, and name. Two clients, John and Paul are both synced with the server, and have the latest contents.

- John modifies a row.

```
UPDATE table SET name='bar' WHERE key = 4
```

- Paul deletes the same row.

```
DELETE FROM table WHERE key = 4
```

- John syncs his change to the server. The sync is successful. The row has the values key=4 and name='bar'

- Paul syncs his change to the server. The server must use the configured rule to resolve the conflict.

The possible resolutions are:

- Ignore the incoming delete operation. The row will remain, with the values key=4 and name='bar'. *This is the default action.*
- Accept the incoming delete operation. The row will be deleted.
- Reject all changes from Paul. Paul will not be able to perform any sync operations that contain this conflict. His client must call zumero_quarantine_since_last_sync() to remove the delete operation.

5.3.3. Attempting to modify a modified row.

The incoming package is trying to modify a row which has been modified.

Example 3. A sample modify-after-modify conflict

Consider a case where the table in question has three columns: key, name, and city. Two clients, John and Paul are both synced with the server, and have the latest contents.

- Both John and Paul have the same contents for the row. The current values are key=4, name='foo' and city='Liverpool'
- John modifies the row.

```
UPDATE table SET name='bar' WHERE key = 4
```

- Paul modifies the row.

```
UPDATE table SET city='Hamburg' WHERE key = 4
```

- John syncs his change to the server. The sync is successful. The row has the values key=4, name='bar', city='Liverpool'
- Paul syncs his change to the server. The server must use the configured rule to resolve the conflict.

The possible resolutions are:

- Use the column merge rules to synthesize a new row, combining both Paul and John's changes. Using the default column merge rules, the row will have the values key=4, name='bar' (from John), city='Hamburg' (from Paul) *This is the default action.*
- Accept Paul's version of the row. The row will have the values key=4, name='foo' (unchanged), city='Hamburg' (from Paul)
- Ignore Paul's version of the row. The row will have the values key=4, name='bar' (from John), city='Liverpool' (unchanged)
- Reject all changes from Paul. Paul will not be able to perform any sync operations that contain this conflict. His client must call zumero_quarantine_since_last_sync() to remove the modify operation.

5.3.3.1. Column Merge

When merging two modify operations, one of the possible actions is to merge the columns. This means instead of resolving this conflict for the row as a whole, examine each column individually, and try to merge the two versions of the row on a column-by-column basis.

It is possible to set the default column merge action for all columns, or to specify each column's merge rule individually. You may not set a column merge rule on a primary key column.

Example 4. Example: column merge

Consider a case where the table in question has three columns: key, name, and city. Two clients, John and Paul are both synced with the server, and have the latest contents.

- Both John and Paul have the same contents for the row. The current values are key=4, name='foo' and city='Liverpool'
- John modifies the row.

```
UPDATE table SET name='bar' WHERE key = 4
```

- Paul modifies the row.

```
UPDATE table SET city='Hamburg' WHERE key = 4
```

- John syncs his change to the server. The sync is successful. The row has the values key=4, name='bar', city='Liverpool'
- Paul syncs his change to the server. The server must use the configured rule to resolve the conflict.

Each column can have a different column merge rule. Consider the following actions for the city column:

- Accept Paul's value for city. The value for the city column will be 'Hamburg'. *This is the default action.*
- Ignore Paul's value for city. The value for the city column will be 'Liverpool'.
- Attempt to merge the text of Paul's value with the text of John's value. The two changes are merged using a line-oriented 3-way merge. The technique is identical to what version control tools do when attempting to automerge two changes to a text file.

  This action is only available for text columns

  When choosing a text merge operation, a fallback action will need to be provided, to handle the case where the text merge fails. The default fallback action is to accept the incoming value.
- Reject all changes from Paul. Paul will not be able to perform any sync operations that contain this conflict. His client must call zumero_quarantine_since_last_sync() to remove the modify operation.

5.3.4. Attempting to delete a deleted row

For the sake of completeness, note that the situation where a sync is attempting to delete a row which has already been deleted is not considered a conflict. If everybody wants the row to go away,it does.

The Conflict Rules dialog does not list this as a conflict situation and does not allow you to change this behavior.

5.3.5. UNIQUE Constraint Violations

Consider a scenario involving two clients, Phil and Lew, who are sharing a database on the server.

- The database contains one synchronized table, defined as:

```
TABLE foo: (
  foo_id uniqueidentifier PRIMARY KEY DEFAULT NEWID(),
  a nvarchar(max),
  b int,
  UNIQUE (a,b)
  );
```

- We start the example with Phil, Lew and the server all at version 1 of the database, which contains no rows.
- Phil does:

```
INSERT INTO foo (a,b) VALUES ('rose', 16);
```

- Lew does:

```
INSERT INTO foo (a,b) VALUES ('rose', 16);
```

- Lew does:

```
zumero_sync(...)
```

  The server's database goes to version 2, with his row inserted into the foo table.
- Phil does:

```
zumero_sync(...)
```

Phil's sync will cause a violation of the unique constraint.

Currently, Zumero does not have any mechanism to allow for alternative handling of regular UNIQUE constraint or UNIQUE index violations. (It does have provisions for handling certain PRIMARY KEY constraint conflicts, though, as we'll see in the next section.) So whenever possible, the best way to deal with UNIQUE constraint violations during sync is to carefully design your app to avoid them happening in the first place. Any time you INSERT or UPDATE data in a column which has a UNIQUE constraint, try to make sure that the data is likely to be unique.

5.3.6. PRIMARY KEY constraints

Like UNIQUE constraints, PRIMARY KEY constraints will cause a violation on sync if a client has inserted the same value as a previous client. There are two exceptions to this behavior:

- If a table's primary key is an IDENTITY column, the Zumero server will automatically assign the record a new IDENTITY value. If the value is referenced by FOREIGN KEY constraint from any other records in the package, those references will be updated as well.

  This is the default behavior for all tables with an IDENTITY primary key. It cannot be changed.
- If a table's primary key is not an IDENTITY column, the table can be configured to ignore the row if it's an exact duplicate of an existing row (in other words, if *all* column values that the client sent are an exact match against a single row on the server).

  To enable this feature from the Conflict Resolution dialog, select "ignore the incoming row if it's an exact duplicate" from the dropdown menu for "Sync wants to insert a row" + "Another row with the same primary key already exists". (Note that this options is *only* available if the table's primary key is *not* an IDENTITY column.)

5.3.7. Foreign Key Constraint Violations

5.3.7.1. Non-"Conflict" Violations

First, as an aside, note that since Zumero replicates the SQL Server data to a SQLite database on the client side, it is possible for client apps to have foreign key constraints disabled. We recommend that apps have foreign key constraints turned *ON* at all times, e.g. by using the foreign_keys pragma:

```
PRAGMA foreign_keys = ON;
```

If foreign_keys are turned off, it is possible for a client to make arbitrary changes to their local database in violation of foreign key constraints. Then when they sync, their package will be rejected due to the constraint violations. This kind of failure is a constraint violation, but not a "conflict" in the traditional sense (i.e. where would-be valid operations in the package are conflicting with other changes that have already happened on the server).

5.3.7.2. Violation on INSERT

Consider a scenario involving two clients, Harold and Don, who are sharing a database on the server.

- The database contains two synchronized tables, defined as:

```
TABLE foo: (a nvarchar(max) PRIMARY KEY);
```

```
TABLE bar: (id uniqueidentifier PRIMARY KEY DEFAULT NEWID(), b nvarchar(max)
 REFERENCES foo (a));
```

- We start the example with Harold, Don and the server all at version 1 of the database, which contains the following row:

```
INSERT INTO foo (a) VALUES ('hello');
```

- Harold does:

```
INSERT INTO bar (b) VALUES ('hello');
```

- Don does:

```
DELETE FROM foo;
```

- Don does:

```
zumero_sync(...)
```

  The server's database goes to version 2, and the foo table is now empty.

- Harold does:

```
zumero_sync(...)
```

Harold's sync will cause a violation of the foreign key constraint, because the row he inserted into bar is referencing a row in foo which no longer exists, because Don deleted it.

Harold will not be able to perform any sync operations that contain this conflict. His client must call zumero_quarantine_since_last_sync() to remove the insert operation. *This is the default behavior.*

Alternatively, Zumero can be configured to ignore insert operations that violate a given foreign key constraint. This setting can be adjusted from the Conflict Rules dialog, when looking at the table with the outgoing foreign key constraint. It will mention the foreign key by name. By default SQL Server will give a foreign key constraint a name with the following components, concatenating them together with underscores in-between:

- "FK"
- the referencing table's name

- the referencing column's name
- a hexadecimal number

This "ignore" setting is described in terms of an insert operation, but it has one other affect as well. Besides insert operations, "undelete" operations will also be ignored. In other words, in the conflict situation where the client package contains an update to a row that has already been deleted, the server will favor ignoring the update operation over attempting to restore a row that would violate the foreign key constraint. Again, this behavior is tied to setting for inserts, rather than having a separate configuration.

One final note about the "ignore" setting is that it does not distinguish between true "conflicts" and the more simple and blatant violations that could arise if the client did not have foreign key constraints enabled. All inserts that violate the constraint will simply be ignored.

5.3.7.3. Violation on DELETE

Broadly speaking, there are two ways to violate a foreign key constraint. You can attempt to insert a value into the referencing table which is not present in the referenced table, as we saw in the previous section. Or you can attempt to delete a value from the referenced table that is still present in the referencing table. Let's look at this latter case next.

Consider, again, a scenario involving two clients, Harold and Don, who are sharing a database on the server.

- The database contains two synchronized tables, defined as:

  ```
  TABLE foo: (a nvarchar(max) PRIMARY KEY);
  ```

  ```
  TABLE bar: (id uniqueidentifier PRIMARY KEY DEFAULT NEWID(), b nvarchar(max)
   REFERENCES foo (a));
  ```

- We start the example with Harold, Don and the server all at version 1 of the database, which contains the following row:

  ```
  INSERT INTO foo (a) VALUES ('hello');
  ```

- Don does:

  ```
  DELETE FROM foo;
  ```

- Harold does:

  ```
  INSERT INTO bar (b) VALUES ('hello');
  ```

- Harold does:

  ```
  zumero_sync(...)
  ```

  The server's database goes to version 2, and the bar table now references the row in foo.

- Don does:

  ```
  zumero_sync(...)
  ```

Don's sync will cause a violation of the foreign key constraint because the row he deleted from foo is being referenced by a row in bar which was added by Harold.

Don will not be able to perform any sync operations that contain this conflict. His client must call zumero_quarantine_since_last_sync() to remove the delete operation. *This is the default behavior.*

While this behavior cannot be adjusted by Zumero, it can be adjusted using referential constraint actions in SQL directly. For example, if the constraint above had specified `ON DELETE CASCADE` then Don's package would have imported successfully and deleted the row in "bar" that had been added by Harold.

5.3.8. CHECK Constraint Violations

Consider a scenario involving two clients, Nancy and Ann, who are sharing a database on the server.

- The database contains one synchronized table, defined as:

  ```
  TABLE foo: (
    id uniqueidentifier PRIMARY KEY DEFAULT NEWID(),
    a int,
    b int,
    c int,
    CHECK (c > (a + b))
   );
  ```

  And one record:

  ```
  INSERT INTO items (a,b,c) VALUES (10, 20, 50);
  ```

- We start the example with Nancy, Ann and the server all at version 1 of the database.
- Nancy does:

  ```
  UPDATE items SET a=25;
  ```

  The row is now (25, 20, 50). Since 50 > (20 + 25), the CHECK constraint is satisfied.
- Ann does:

  ```
  UPDATE items SET b=35;
  ```

  The row is now (10, 35, 50). Since 50 > (10 + 35), the CHECK constraint is satisfied.
- Ann does:

  ```
  zumero_sync(...)
  ```

  The server's database goes to version 2.
- Nancy does:

  ```
  zumero_sync(...)
  ```

Nancy's sync will cause a violation of the CHECK constraint, because column merge will result in a record (25, 35, 50), which is a violation because 50 is not > (25 + 35).

Nancy will not be able to perform any sync operations that contain this conflict. Her client must call zumero_quarantine_since_last_sync() to remove the update operation. *This is the default behavior.* It cannot be changed.

Also, note that currently ZSS does not replicate any of the server's CHECK constraints to the client. App developers using Zumero should have logic in place at the higher layers which will ensure the validity of the data being inserted (though this would be true even if the CHECK constraints were replicated, to avoid client-side SQL constraint violation errors). That being said, it is still technically possible that, like with FOREIGN KEY constraints, a client package could encounter a CHECK constraint violation on the server, even if there were no other "conflicting" changes involved.

6. SQL Server Considerations

Zumero tries hard to tread lightly as a participant in your SQL Server infrastructure.

6.1. Create DBFile

The first time this command is run in a SQL Server database, Create DBFile creates the `zumero` schema. All new objects created by ZSS Manager are created under this schema to logically separate them from your database.

Note

Do *not* create tables in the `zumero` schema. That schema is reserved for use by ZSS; data in the `zumero` schema may be dropped or modified without warning.

Each time a new DBFile is added, ZSS Manager performs these additional steps:

- Create a handful of housekeeping tables for the DBFile, some with triggers on them. These tables keep track of transactions, data changes, schema information, merge resolution rules, and user permissions.
- Create a stored procedure which determines the current SQL Server transaction and returns a corresponding Zumero transaction ID.
- Create a stored procedure which marks a new set of transactions to be exported to Zumero clients.

6.2. Add Tables to DBFile

In terms of database changes, adding a table will:

- Create a view which the Zumero Server uses to query and modify the selected table when a Zumero client syncs. Because this view is also used for changes coming from a Zumero client, there are insert, update, and delete triggers on this view.
- Create insert, update, and delete triggers on the selected table allowing Zumero to keep track of changes made by other applications.
- Save the table's current schema.
- Save transaction data, effectively pretending that all existing rows in the selected table were inserted in a Zumero transaction.

6.2.1. Primary Keys

Zumero's triggers depend on a table's primary key to uniquely identify rows. A side-effect of this requirement is that **in a synchronized table, primary key values can no longer be changed.** If you try, the transaction will fail with a foreign key constraint violation from a Zumero housekeeping table which looks like this:

```
The UPDATE statement conflicted with the REFERENCE constraint "fk_test$z$x$5975771".
The conflict occurred in database "test", table "zumero.test$z$x$5975771", column
 'test'.
```

Zumero creates an `AFTER UPDATE` trigger on your table. In it, Zumero needs to be able to correlate a row from the implicit `DELETED` table with its counterpart in the implicit `INSERTED` table. Unfortunately SQL Server provides no way to do this beyond a set of columns which uniquely identifies the row and doesn't change. By preventing updates to primary key values, we always have this.

For this reason and others, changing primary key values in SQL Server is often cited as a bad practice. If you have existing applications that rely on changing primary key values, you may want to change them whether or not you use Zumero. If you must change a row's primary key values, you will need to delete the row and add it back with the desired changes.

6.2.2. Requirements

ZSS Manager will refuse to add a table unless it meets certain requirements. To be added, a table must...

- have a primary key
- not have any columns whose types are unsupported. Where possible, unsupported columns will be excluded and not synced to the client; however, unsupported primary key columns, etc. can't be excluded. Tables with such columns cannot be synchronized.
- not have an identity column unless it is the (one and only) primary key column
- not have a non-integer identity column
- all foreign key references must point to a table already added to the DBFile. Alternatively, you can choose to skip the foreign key reference column.
- not use any reserved column names
- not include foreign key references to other columns in the *same* table, if an identity primary key is present
- not be part of a circular foreign key reference (other than allowable references to columns in the same table)

6.2.2.1. Unsupported Data Types

Zumero for SQL Server does not support the following SQL Server data types:

- `datetimeoffset`
- `decimal/numeric with precision greater than 18`
- `image`
- `ntext`
- `sql_variant`
- `text`
- `timestamp`
- `xml`

6.2.2.2. User Defined Types

Zumero for SQL Server supports any user-defined type that is based on a supported system type.

6.2.2.3. Reserved Column Names

The following column names are reserved for use by Zumero or SQLite:

- `_rowid_`
- `z_del_txid`
- `z_gen_add`
- `z_gen_del`
- `z_rid`
- `z_rv`
- `z_txid`

6.2.3. Warnings

ZSS Manager will display warnings for certain tables. These warnings don't prevent you from adding a table but usually indicate the table has characteristics which won't be replicated to a client SQLite database.

The following table characteristics will not be automatically replicated to client databases:

- Non-unique indexes
- check constraints
- filtered unique constraints
- unique constraints with `ignore_dup_key` enabled
- triggers (see SQL Server Triggers below as well)
- computed columns

6.2.3.1. Primary Keys

Adding a table will prevent future changes to data in that table's primary key column(s). The reasons behind this are described above. ZSS Manager will warn for any non-IDENTITY primary key columns (because IDENTITY columns are already immutable).

6.2.3.2. Indexes

Because they serve a data integrity purpose, all unique indexes (or constraints) will be replicated to the client database. Non-unique indexes, however, are deliberately left behind. Indexes that are designed to optimize performance on your server are often not the same indexes you want for a single app's use on a mobile device. For this reason, Zumero does not automatically replicate them. You should add to the client database only the indexes that are appropriate to your mobile app.

6.2.3.3. Decimal, Numeric, Money and Smallmoney types

The exact SQL Server floating point types also yield a warning. Because SQLite has no exact floating point type, these values are multiplied by `10^scale` and stored as integers in the client database. `Money` and `smallmoney` always have scale 4 and are multiplied by 10^4 or 10,000. For example, a SQL Server `smallmoney` value of 12.34 is saved as 123400 in the client database. In a client app, you should multiply or divide the integer by the defined scale when saving or retrieving the value, respectively.

Note

Zumero does not support decimal or numeric columns with precision greater than 18. 64-bit integers are used in the client SQLite database, and precisions greater than 18 will overflow them.

6.2.3.4. Default Values

ZSS supports some SQL Server default value definitions. Most literals and the `GETUTCDATE` function are supported and will be automatically included in the client database. A warning will appear for any unsupported defaults. Unsupported defaults are not included in the client database.

6.2.3.5. Small Identity Columns

ZSS Manager will warn you about any `IDENTITY` columns smaller than 32-bits. Identity values created on the client can be considered temporary: when syncing, the server will insert "official" values and replicate the changes back to the client. For this reason, the client can't know if it is overflowing an identity column. Only the server knows for sure. If you were to make a `tinyint` column an `IDENTITY(1,1)`, you'd only get 255 rows before an overflow. ZSS Manager warns you to help prevent this scenario.

6.2.3.6. Non-Unicode Text Columns

The client SQLite database has only Unicode text storage. All text in the client database is UTF-8 encoded and no attempt is made to exclude non-ASCII text. For this reason, ZSS Manager will warn about `char`

and `varchar` fields, which do not support Unicode. You should take care in your client to write only ASCII data in these fields. Non-ASCII data may be mangled when synced to the server.

6.2.3.7. MAX columns

ZSS Manager will warn about the lengths of any `varchar(max)`, `nvarchar(max)`, or `varbinary(max)` columns. In SQL Server, the maximum length for these columns is 2 gigabytes. In the client SQLite database the maximum length is 1 billion bytes, which is a bit less than a gigabyte. (This is unlikely to affect a mobile app. You don't want to transfer multi-gigabyte fields over mobile networks for reasons that have nothing to do with Zumero: mobile networks aren't that fast and mobile device storage isn't that plentiful.)

6.2.3.8. Unchecked foreign keys

ZSS Manager will warn about unchecked foreign keys. In SQL Server, it is possible (usually via `ALTER TABLE... WITH NOCHECK`) to have a foreign key relationship where the values are not actually checked against the foreign tables. This can lead to foreign key constraint failures on the client side when non-matching rows are included in the synced data.

6.2.3.9. Columns named "rowid" or "oid"

The column names `"rowid"` and `"oid"` can have special meaning in SQLite[5]. If a table has a column named "rowid" or "oid" (or one of its variants - "RowID", "OID", etc.) Zumero will create and synchronize this column correctly in the client database, however some third-party ORMs or libraries might not work correctly with it. For this reason, ZSS Manager will issue a warning for such a column.

6.2.4. SQL Server Schemas

ZSS supports SQL Server schemas. Any table belonging to a schema other than `dbo` will appear in the client database prefixed by the schema name and an underscore:

SQL Server Table	Client Database Table
dbo.Categories	Categories
sales.Categories	sales_Categories
region.Categories	region_Categories

Note

You must *not* create tables in the `zumero` schema. That schema is reserved for use by ZSS; data in the `zumero` schema may be dropped or modified without warning.

6.3. Stop Syncing a Table

If you need to stop syncing a table, you select the table in the ZSS Manager window and pick the "Stop Syncing" entry from the context menu. It should be noted, however, that this is destructive and shouldn't be done on live applications with Zumero clients. Your data is left alone, but all Zumero housekeeping data is deleted and the housekeeping tables are dropped.

Clients will drop their corresponding table on the next sync. This behavior has noteworthy consequences:

- **Any unsynced changes to the table in client databases are lost.**
- **The previous statement is true even if the table is re-added before a client has synced and dropped the table.** All Zumero housekeeping data is deleted, so clients must still drop and recreate the table.

[5] http://www.sqlite.org/lang_createtable.html#rowid

6.4. Delete DBFile

Deleting a DBFile should be considered destructive. It is tremendously useful during development, but should be avoided in production. Deleting a DBFile won't delete your data from the server database, but it will invalidate any client databases that have synced against it. Even if you replace a deleted DBFile with another of the same name, corresponding client databases need to be deleted and re-synced. Also note that any user permissions associated with the DBFile will be permanently deleted.

6.5. Permissions

The Zumero Server needs the same read/write permissions that you have configured for other applications that use the data. You should give it access to only the data you expect your Zumero clients to modify.

VIEW SERVER STATE

Zumero requires that the `VIEW SERVER STATE` permission is granted for both the Zumero Server and for any other applications that will modify synchronized tables. This allows Zumero to correlate SQL Server transaction IDs with Zumero transaction IDs.

Zumero installs triggers that call `sys.dm_tran_current_transaction` to retrieve the ID of the current SQL Server transaction. Because these triggers will be fired by anyone who changes data in a synchronized table, it's important that you grant this permission for all users and groups who will alter data in synchronized tables.

IDENTITY_INSERT

If any synchronized table has an IDENTITY column for its primary key, the Zumero Server will issue `SET IDENTITY_INSERT` commands on that table in a few special situations, detailed below. The permissions required to perform `SET IDENTITY_INSERT` state that the user "must own the table or have ALTER permission on the table."

The first situation where `SET IDENTITY_INSERT` is used is as follows. Two clients are both in sync. One client deletes a row (in a table with an IDENTITY primary key) and syncs. The other client modifies the same row and syncs. In this situation, the Zumero Server's default behaviour is to "restore the row with the new contents" (though this can be changed from within the Conflict Rules dialogue). When the Zumero Server restores the row, it will use `SET IDENTITY_INSERT` in order to restore it with the same IDENTITY value that it had before.

The other situation where `SET IDENTITY_INSERT` may be used can arise on a table that has both an IDENTITY primary key and a foreign key constraint on a NOT NULL column. If a client deletes a row in the parent table but does not delete its children in the aforementioned child table (instead having assigned them a new parent) and then syncs, when the Zumero Server imports this change it will DELETE the child row(s) and re-INSERT them with the new parent. Once again, it uses `SET IDENTITY_INSERT` in order to retain the original IDENTITY value(s).

6.6. SQL Server Triggers

As noted in the Warnings section, ZSS will not replicate SQL Server triggers to SQLite on the client devices. Furthermore, some care must be taken to ensure that triggers on synced tables will be compatible with Zumero.

6.6.1. Trigger Order

When adding a table to the DBFile, ZSS Manager calls `sp_settriggerorder` to set Zumero's update trigger as "First". If this step fails, the table will still be added, but a warning is issued: *"Unable to set the Zumero trigger as First. If any update trigger on this table changes the table before the Zumero trigger*

runs, serious data corruption may occur. Check your trigger order, and make sure that any update triggers that change the data in the table are run after the Zumero trigger.”

The data corruption in the table arises when there is an update trigger on the table that performs another update operation within the table (for example, to keep a timestamp column up-to-date). If you do not have any update triggers on the table that update the table (and know that you never will) this warning can be ignored.

If, on the other hand, you have any update triggers that update the table, these triggers *must* be set to execute *after* Zumero's trigger. If a non-Zumero update trigger executes before Zumero's update trigger and then performs an update within the table, Zumero will store the wrong versions the updated rows in its record of the table's history. This corrupt historical data could then be used when performing conflict resolution (e.g. in a Column Merge), which then produces corrupt data in the live table in surprising ways.

6.6.2. Trigger Output

Triggers that issue counts or PRINT statements have been known to interfere with the Zumero server's connection to the database.

Make sure that a `SET NOCOUNT ON` statement is executed at the beginning of each trigger, and remove any PRINT statements.

6.6.3. Incompatible Triggers

Some triggers that change the effect of the executed SQL may prove to be incompatible with Zumero. In particular, watch out for following situations:

- INSTEAD OF INSERT triggers must insert the same number of rows as the original INSERT, and the rows must have the same primary key values as were originally being inserted.
- AFTER INSERT triggers must not delete any of the inserted rows.
- INSTEAD OF UPDATE triggers must update the exact same set of rows as the original UPDATE.
- INSTEAD OF DELETE triggers must delete the exact same set of rows as the original DELETE.

6.7. Database Schema Changes

Once a table has been added to a Zumero DBFile, most schema changes will be detected automatically, and propagated to the client on the next sync.

Tables may be renamed; columns may be added, dropped or renamed; foreign keys may be added, dropped and modified; constraints and data types may be changed, etc.

6.7.1. Deployment Tips

As is the case for any database-backed app, schema changes are a pain. Generally speaking, you have to deploy the new schema and the app changes to support them simultaneously. ZSS has no silver bullet for this issue, but we've attempted to make schema changes in Zumero-enabled databases no *more* difficult than they already were. When you change the schema of a synchronized table, clients pull down the new schema on their next sync. So your client app needs to be capable of dealing with the new schema before it's applied to the production database. For a simple change, you'd do something like this:

- Make your schema changes in your development database.
- Update your app to handle the new schema using the development database. You have several options here. The best choice depends on the scope of the schema changes and your app deployment scenario:
  - Make the app capable of dealing with both the old and the new schema. Deploy the app first, then run the schema change scripts in production.

- Make the app capable of dealing with only the new schema. Deploy the app and the schema changes "simultaneously," where the definition of simultaneous fits your deployment scenario.
- Add schema versions to your database (e.g. simply a table with a schema version number that you create) so the app can make more sophisticated decisions based on the schema it sees.

6.7.2. Renaming a Table

The standard `sp_rename` procedure cannot be used to rename synchronized tables. If attempted, SQL will issue an error message stating *"Object '[dbo].[foo]' cannot be renamed because the object participates in enforced dependencies."* To work around this limitation, Zumero provides a stored procedure `zumero.RenamePreparedTable` which can be used to do the rename. For example, to rename table `[foo]` to `[bar]` in the `[dbo]` schema, issue the command:

```
EXEC [zumero].[RenamePreparedTable] 'dbo', 'foo', 'bar';
```

6.7.3. Adding a Column

When you add a new column to a table, Zumero will not add that column to client devices unless you explicitly tell it to start syncing the column. This can be done from ZSS Manager's "Choose Columns..." dialog for that table. You can also tell Zumero to start syncing a column by calling a Zumero-provided SQL stored procedure. This could be called directly from within the same SQL script being used to add the column. For example:

```
BEGIN TRANSACTION;
 ALTER TABLE [dbo].[foo] ADD [c] int;
 EXEC [zumero].[StartSyncingColumn] 'dbo', 'foo', 'c';
 COMMIT TRANSACTION;
```

Similarly, Zumero provides a `StopSyncingColumn` stored procedure which allows you stop syncing a certain column without removing it from the server database. When this is used, the column will be removed from all client devices, and Zumero's history and housekeeping data for that column will be removed from the server as well.

6.7.4. Warnings

Many schema change operations will cause Zumero to drop and re-create its triggers on the host table. When this happens (as when first adding the table to the dbfile) `sp_settriggerorder` is used to set Zumero's update trigger as "First". If `sp_settriggerorder` fails, a warning is issued saying *“Zumero Warning: Unable to set trigger as First. If any update trigger on this table changes the table before the Zumero trigger runs, serious data corruption may occur. Check your trigger order, and make sure that any update triggers that change the data in the table are run after the Zumero trigger.”*

This warning message is referring to the fact that an update trigger that performs updates within the table will cause data corruption if it executes before Zumero's trigger. This is described in more detail in the Trigger Order section.

Additionally, certain other warnings may be issued similar to the warnings when adding the table, in particular when altering a column of a data type that may require special handling by Zumero clients.

6.7.5. Problematic Schema Changes

In a synchronized table, the references and triggers created by Zumero will prevent certain schema changes from succeeding, including:

- Altering or dropping the primary key column.
- Removing the table's primary key constraint.
- Dropping the synchronized table. (In order to drop a synchronized table, you must first use ZSS Manager to stop syncing the table.)

There are other changes which Zumero does not prevent from happening, but which put the database into a state that Zumero does not support. When this happens, all Zumero clients will receive a "server_misconfiguration" error every time they try to sync with the affected dbfile. Schema changes in this category include:

- Changing the type of a Zumero-synchronized column to one of the unsupported types.
- Creating a circular foreign key dependency between synchronized tables. (For example, Table-A has a foreign key reference into Table-B and Table-B has a foreign key reference into Table-A.)
- Adding a same-table reference constraint in a synchronized table whose primary key is an identity column.

Finally, certain other schema changes are supported, but can cause trouble if you're not careful. These include the following.

- If you drop or rename a column that is referenced by a filter's Row Exclusion WHERE Clause, the users specified for that filter will be unable to sync until the WHERE clause is updated. They will receive an "http_500" error (Internal Server Error).
- It is possible to add a foreign key constraint on a syncing column which references a column that is not synced. When this is the case, the client has no way to enforce the constraint and will receive a "foreign_key_constraint_violation" error on sync if it has inserted an invalid value.
- If you add a NOT NULL column that does not have a default value, clients will not be able to push inserts into the table until you have either started syncing the column with Zumero or else given it a default value.
- When changing a column's default value, it is important to note that while most default values that are simple expressions will be synced to the client, some default values will not. ZSS Manager's Errors and Warnings dialog will show warnings about default values that won't sync, but when altering them using SQL no warning will be issued.

Most of these issues will be listed on the Errors and Warnings dialog.

6.8. Data Type Conversion and Limitations

In order to maintain compatibility between the SQL Server and mobile (SQLite) copies of your data, certain column types involve type conversions and/or content limitations to ensure that:

- SQL Server rows will arrive intact on the mobile side
- SQLite rows will arrive intact in SQL Server
- Data will "round trip" properly between the two ends
- Equivalent column values added on both ends will be equal to each other

SQL Server Type	SQLite Type	Conversions	Restrictions
`CHAR`, `NCHAR`	`text`	Values added from SQLite will be padded to the full column width	

SQL Server Type	SQLite Type	Conversions	Restrictions
		(with the exception of NULL values). SQL Server will automatically pad strings to match the fixed column width, so Zumero mimics this behavior on the mobile side. This applies to fixed-width fields only; VARCHAR and NVARCHAR will not be automatically padded.	
MONEY, SMALLMONEY	int	SQL Server stores money values as ten-thousandths of a monetary unit. The mobile data will reflect that ten-thousandth value, not the whole. For example, assuming dollars as the monetary unit, adding $2.75 on the SQL Server side would result in a column containing 27,500 in the mobile database. Similarly, adding 10,000 in SQLite would result in $1.00 being stored in SQL Server.	
DECIMAL, NUMERIC	int	DECIMAL and NUMERIC types are converted to integers when saved in SQLite by multiplying by 10^scale. The integers are divided by the 10^scale for conversion back to SQL Server. For example suppose we have a DECIMAL(10,5) column (5 is the scale.) A value of 12.34 in SQL Server would be stored as 1,234,000 in SQLite.	Zumero does not support decimal or numeric columns with precision greater than 18. 64-bit integers are used in the client SQLite database, and precisions greater than 18 will overflow them.

SQL Server Type	SQLite Type	Conversions	Restrictions
DATE	text		In order for date values to be interpreted identically on both the SQL Server and SQLite sides, when adding dates to mobile databases, you *must* use YYYY-MM-DD format. All other (non-NULL) values will be rejected at INSERT/UPDATE time.
TIME	text		In order for time values to be interpreted identically on both the SQL Server and SQLite sides, when adding times to mobile databases, you *must* use one of the following time formats: • Whole seconds: HH:MM:SS • Fractional seconds, from 3 to 7 digits: HH:MM:SS.sss[s{1,4}] All other (non-NULL) values will be rejected at INSERT/UPDATE time.
DATETIME	text		In order for datetime values to be interpreted identically on both the SQL Server and SQLite sides, when adding times to mobile databases, you *must* use one of the following formats: • Minutes only: YYYY-MM-DD HH:MM or YYYY-MM-DDTHH:MM • Whole seconds: YYYY-MM-DD HH:MM:SS or YYYY-MM-DDTHH:MM:SS • Fractional seconds, from 3 to 7 digits: HYYYY-MM-DD H:MM:SS.sss[s{1,4}]or HYYYY-MM-DDTH:MM:SS.sss[s{1,4}] **Note** In all cases, the "T" is a literal letter "T", which may be used interchangeably with " " as the date/time separator. For example, 12:55pm on January 10, 1982 may be represented as

SQL Server Type	SQLite Type	Conversions	Restrictions
			either `1982-01-10 12:55` or `1982-01-10T12:55`. All other (non-`NULL`) values will be rejected at `INSERT`/`UPDATE` time.
`UNIQUEIDENTIFIER`	`blob`	The `uniqueidentifier` type is stored as a 16-byte blob in SQLite (matching the 16-byte value stored on SQL Server).	
`GEOMETRY`, `GEOGRAPHY`, `HIERARCHYID`	`blob`	These .net object types will be stored in their default binary representation. Interpretation of these values is up to the client application.	Do *not* insert arbitrary binary data, or modify synced data, unless you are absolutely *certain* you know what you are doing. Data which cannot be properly deserialized will cause sync operations to fail. If columns of these types aren't strictly needed on the client, it is safest to exclude those columns from sync.

6.8.1. Working with uniqueidentifier Columns in Client Databases

A `uniqueidentifier` column is stored in the client SQLite database as 16-byte binary values, just like in SQL Server.

You can use SQLite's `randomblob` function to create new uniqueidentifier values:

```
insert into my_table (my_guid_col, my_text_col) values (randomblob(16), 'my data');
```

However, SQL Server developers are accustomed to seeing uniqueidentifier values (GUIDs) in their standard textual representation. For example: EA296149-4E9A-4DB7-9944-F3459931140C. For the purposes of display and literal values with the uniqueidentifier data type, SQL Server automatically converts to/from this textual representation.

But SQLite does not have built-in support for this textual representation of a 16-byte blob. If you ask SQLite to SELECT the value of 16-byte blob, you will get 16 binary bytes. It will not automatically convert to the textual representation typically used to display a GUID.

SQLite does have a `hex` function which can be used to get a textual representation in hexadecimal:

```
select hex(my_guid_col) from my_table;
```

However, this will result in a textual representation which is different: 496129EA9A4EB74D9944F3459931140C. The dashes are missing. And the byte order is different as well.

Microsoft's GUID format uses little endian byte order for the first three fields of the GUID[6]. In the first three fields, the order of hexadecimal pairs is reversed as follows:

[6] http://en.wikipedia.org/wiki/Globally_unique_identifier

SQL Server GUID	Raw Hex
AABBCCDD-EEFF-GGHH-IIJJ-KKLLMMNNOOPP	DDCCBBAAFFEEHHGGIIJJKKLLMMNNOOPP
EA296149-4E9A-4DB7-9944-F3459931140C	496129EA9A4EB74D9944F3459931140C

The differing textual representations also come into play when using a uniqueidentifier value as a literal. If you try to do this the normal SQL Server way:

```
insert into my_table (my_guid_col) values ('EA296149-4E9A-4DB7-9944-F3459931140C');
```

... you will get an error. For SQLite, the GUID value shown is a string literal, not a GUID value, and SQLite will not automatically convert to a 16-byte blob.

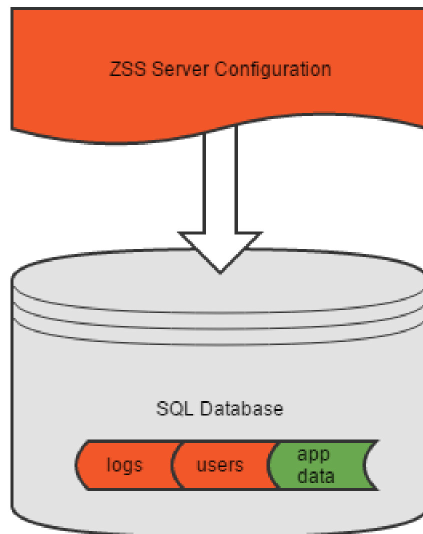
Use SQLite's binary literal syntax to specify a uniqueidentifier value:

```
select my_text_col from my_table
where my_guid_col = x'496129EA9A4EB74D9944F3459931140C';

insert into my_table (my_guid_col, my_text_col)
values (x'496129EA9A4EB74D9944F3459931140C', 'my data');

update my_table set my_text_col='test123'
where my_guid_col = x'496129EA9A4EB74D9944F3459931140C';
```

7. Advanced Topics

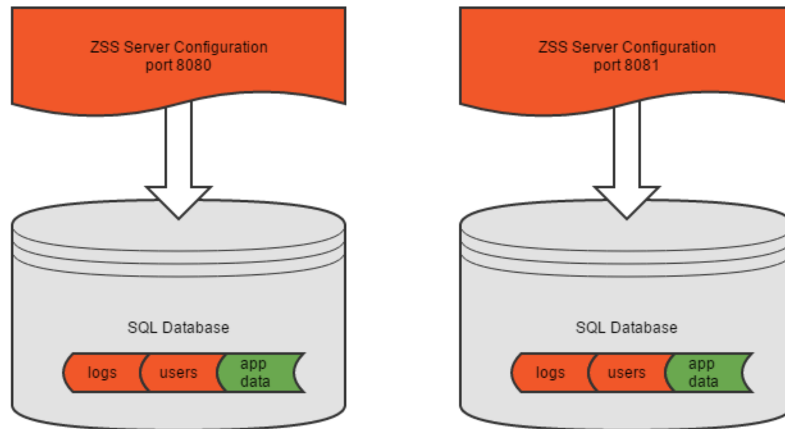
7.1. Multiple Database Configurations

In the simplest configuration, ZSS server will point to a single database containing both its configuration data and your application data.



The database specified during ZSS configuration will be your application's data store. Your Zumero log files, users, and configuration and housekeeping data will be stored here.

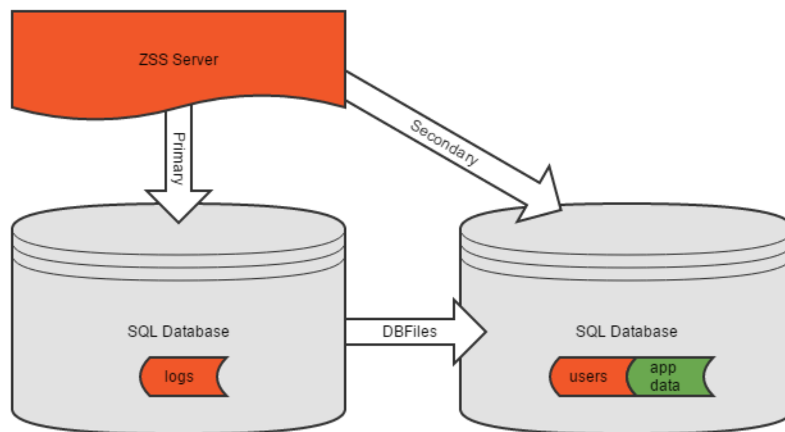
Another possible configuration is multiple ZSS server instances, each pointing at its own database.



This would be configured through the ZSS Server Configuration tool.

Each instance runs on a separate port, and behaves just as the simple version above: one server instance, one database, all your data and Zumero's data in the same place.

If your application's data spans multiple databases, or if you simply wish to keep Zumero's configuration separate from the application data, you may choose to use a multiple database configuration.



In this case you will again configure ZSS Server to point at a single database. This database may or may not contain application data as well. It *will* contain Zumero configuration data, including pointers to the databases and tables where your data is stored.

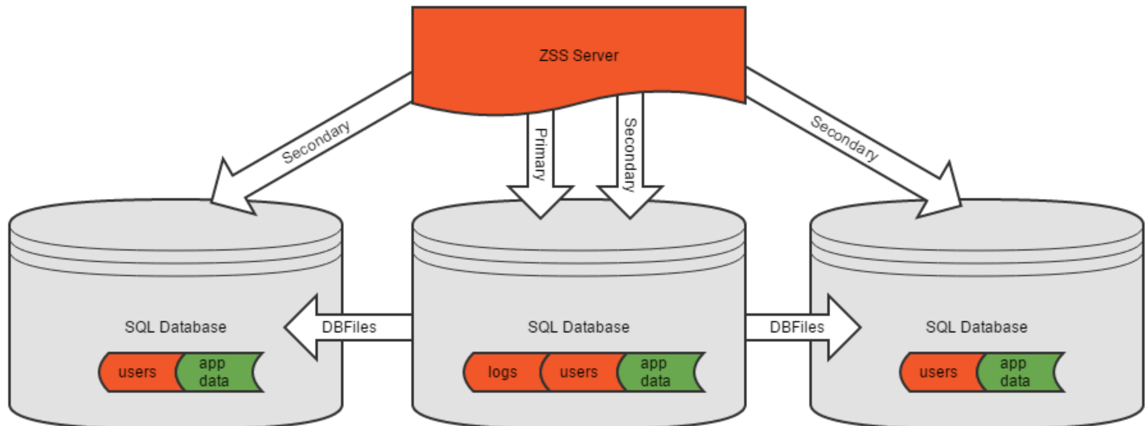
This database is referred to as the **Primary** ZSS database. ZSS Server is always connected to this database. ZSS manager will also always be connected to this database.

Both ZSS Server and ZSS manager may also connect to other databases where your application data are stored. These data may exist solely in one or more other databases, or some may exist in the primary database as well.

In all cases, when referring to the databases storing your application's data, we refer to these as **Secondary** databases. They contain application data, Zumero historical information, and optional user information; but they do not contain Zumero configuration or logs (unless also acting as a Primary).

Above we see a sample database structure consisting of a single primary database with configuration data only, and a single secondary database containing application data. Note that the ZSS server has been configured to connect to the primary database; it finds the secondary databases via the Primary.

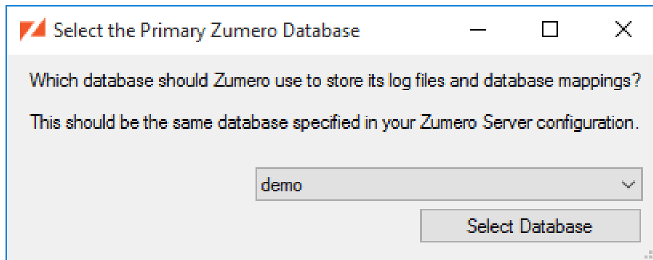
Secondary databases may reside on the same SQL Server as the Primary database, on another SQL Server, or both.



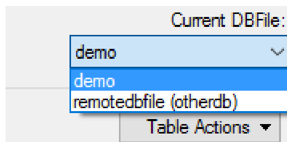
In this more-complex scenario we have a primary database which also includes application data. This may have started as a simple, single-database ZSS installation. An additional secondary database has been added.

Note that when creating a DBFile, or adding tables to that DBFile, we will always be working with a single secondary connection. DBFiles cannot span databases.

Let's look at how this configuration would behave in ZSS Manager.



Here, we are connecting ZSS manager to our primary database -- the same database configured for ZSS Server. At the moment, we're also connected to that same database as our Secondary connection.

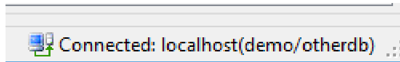


Looking at the list of available DBFiles, we see that some of them are "remote" -- they live in a secondary database ("otherdb") to which we're not connected at the moment.

When we select one of these DBFiles, ZSS Manager's Secondary connection is changed automatically to the database housing that DBFile's data.

The DBFile list now shows this DBFile as "local", and our previous DBFile as remote.

The status bar shows which server(s) and database(s) you are currently connected to. The primary connection is listed first, and secondary (if any) is listed second:



7.2. Migrating ZSS Configuration between Servers

7.2.1. What is Migration?

For most projects, you'll have (at least) a development Zumero server and a production server. You'll test things out, create and change DBFiles, tweak filters, find just the right back-end schema, etc. in the development environment. At some point, though, you need to move that setup into production — preferably without having to go through all of the Zumero configuration steps again. ZSS Migration handles that.

7.2.2. What does Migration do?

ZSS can migrate, on a DBFile-at-a-time basis:

- The definition of the DBFile itself
- The tables synchronized with that DBFile
- Row and column merge conflict rules defined for that DBFile
- Column and table exclusions
- Where-clause filters
- Users defined in `zumero.users`
- Permissions associated with migrated users
- Permissions associated with "Anyone" or "Any Authenticated Users"
- "table2" authorization dependencies on other DBFiles

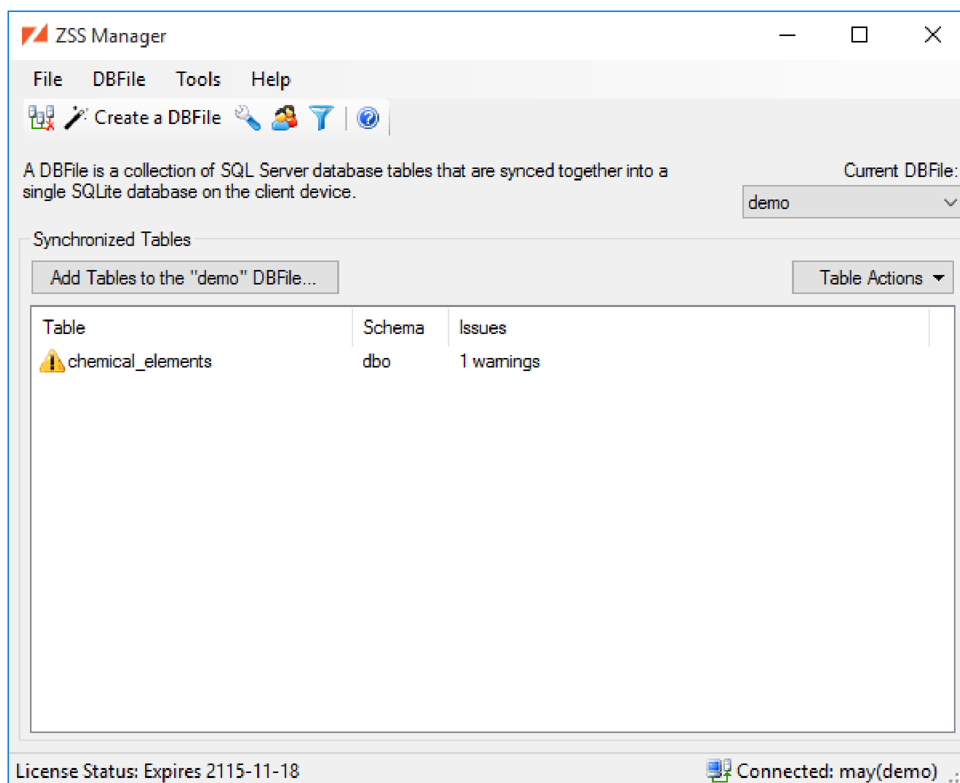
7.2.3. What *doesn't* Migration do?

ZSS Migration (like ZSS in general) doesn't create or alter your underlying data. In particular, Migration will not:

- Create your tables
- Add columns missing from the destination tables
- Change data types to match the source tables
- Start Synchronizing tables that are missing columns synced in the source DBFile
- Migrate or create user tables *other* than `zumero.users`

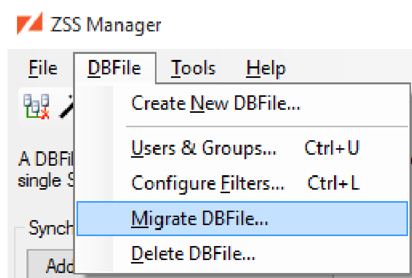
7.2.4. Using ZSS Migration

Select the DBFile that you wish to migrate. From the menubar, choose DBFile -> Migrate DBFile

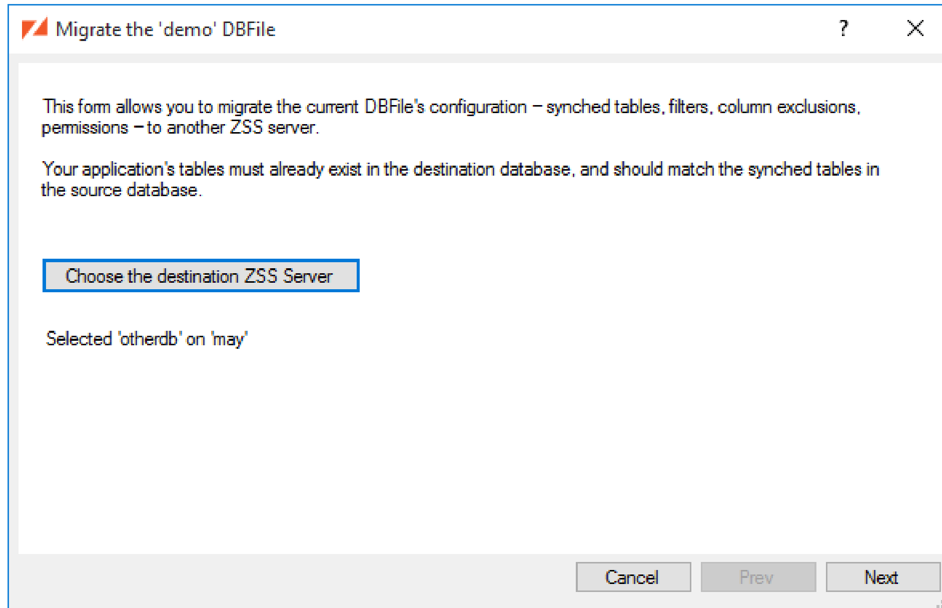


In this example, we're migrating the `demo` DBFile in the `demo` database on `may`

To begin, select Migrate the current DBFile... from the DBFile menu.

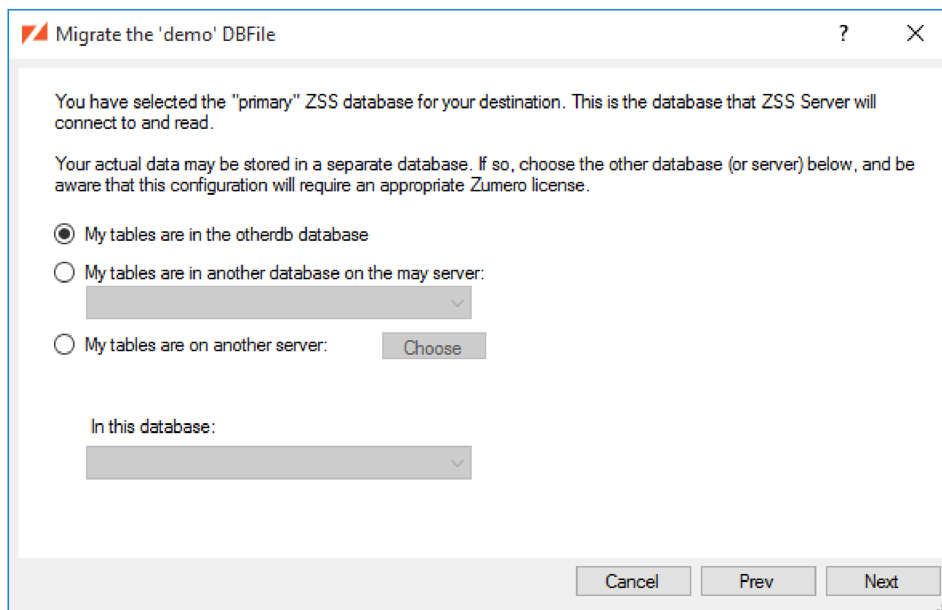


Our first step is to select the destination database for the migrated DBFile. This would be the primary database, e.g. the one that your production ZSS Server is configured to use.



We've chosen the `otherdb` database.

Next, we need to select the database where your data lives in the destination environment (i.e. the secondary database)



We have an all-in-one setup here, but you could choose any supported multi-database configuration.

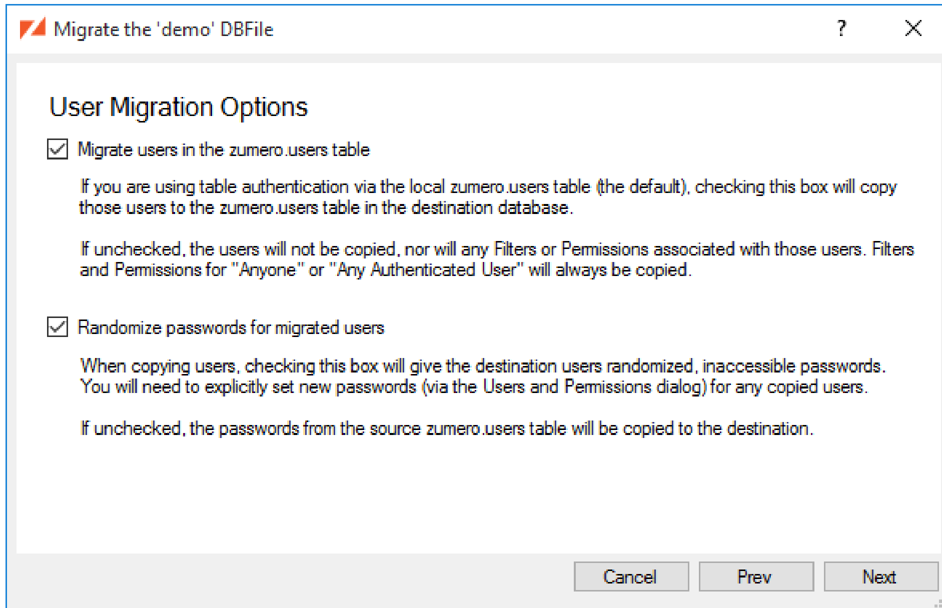
By default, if a `zumero.users` table is found alongside your data tables, ZSS Manager will migrate those users to the matching `zumero.users` table at the destination. You can choose *not* to do this if, for example, you're moving from a development environment full of dummy users into a production environment where things need to be more locked-down.

If you choose to migrate these users, you can either:

- Migrate the passwords along with the usernames, thus allowing the same login credentials in both environments. Or...

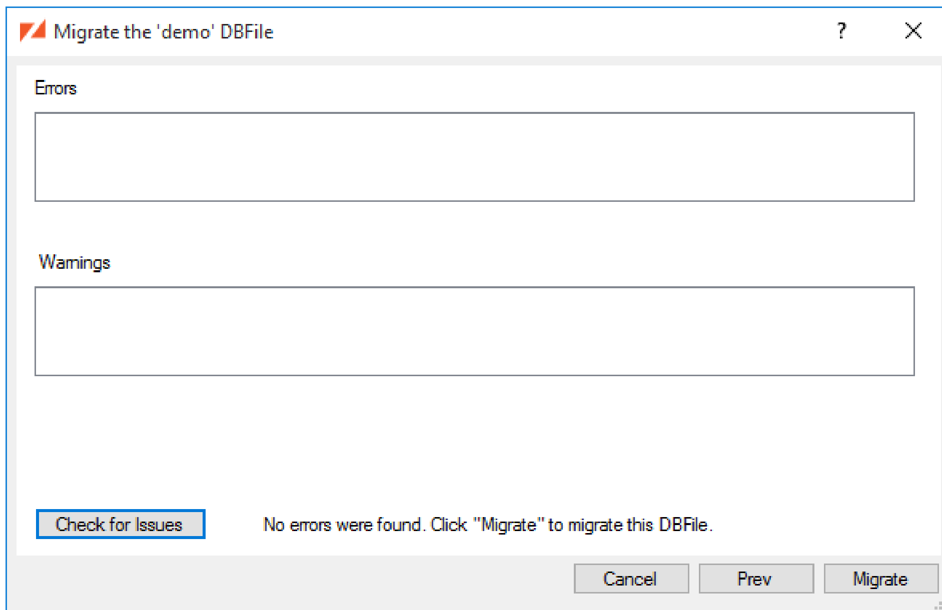
- Use scrambled passwords in the new environment. In this case, you'll need to explicitly change the passwords on any migrated users that will actually be authenticating and syncing.

Randomized passwords spare you from re-creating your users (and their associated permissions) without compromising security. Only those users you specifically edit will be allowed to actually *do* anything.



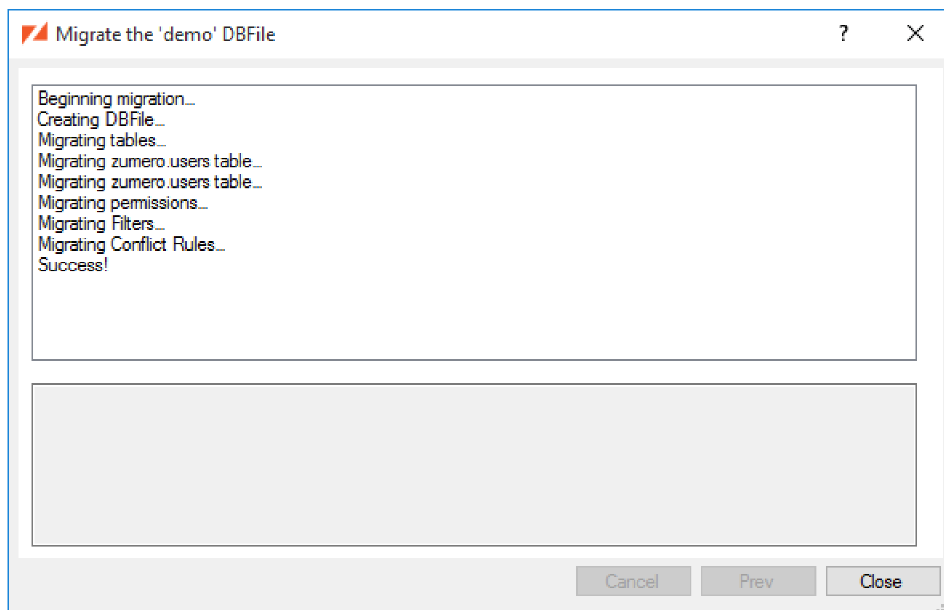
Here, we're migrating users for convenience, but scrambling their passwords for safety.

We're almost ready to migrate our configuration, but first we need to be sure we *can* migrate safely. To do so, click Check for Issues. If any errors are found (missing columns, etc.) migration won't proceed until they are corrected.



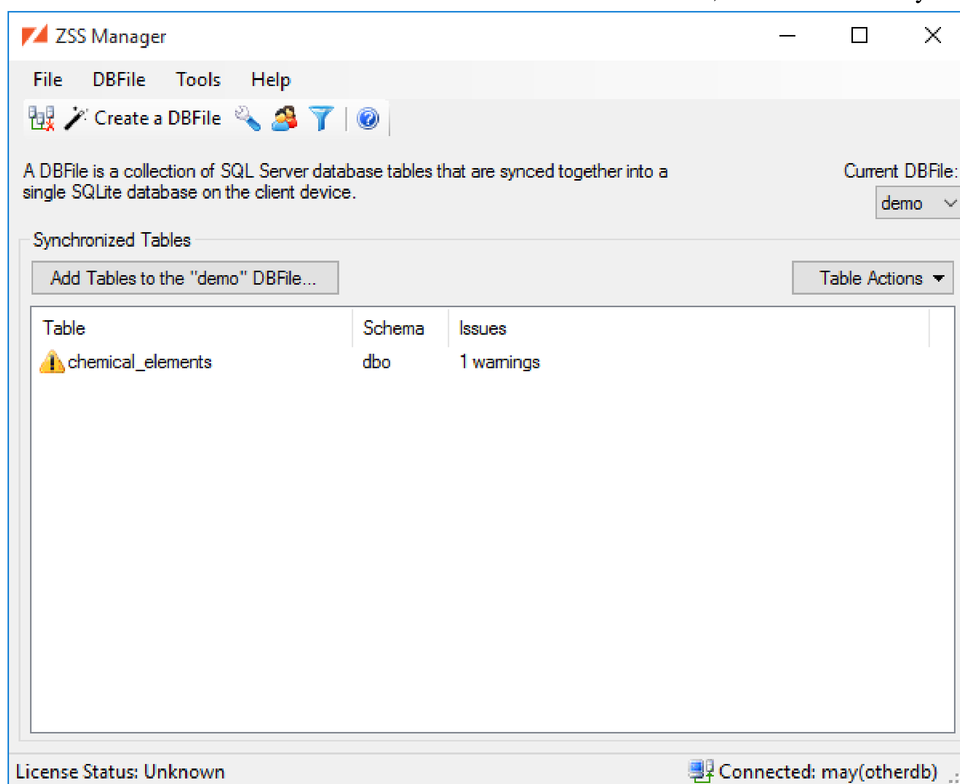
No problems here. We're ready to go.

We click Migrate, and see a series of status messages as the migration proceeds.

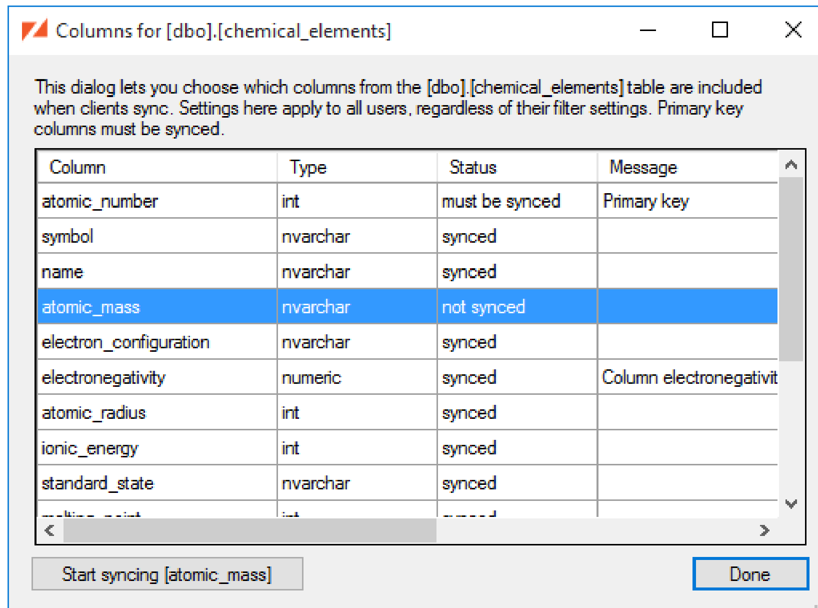


This migration completed successfully.

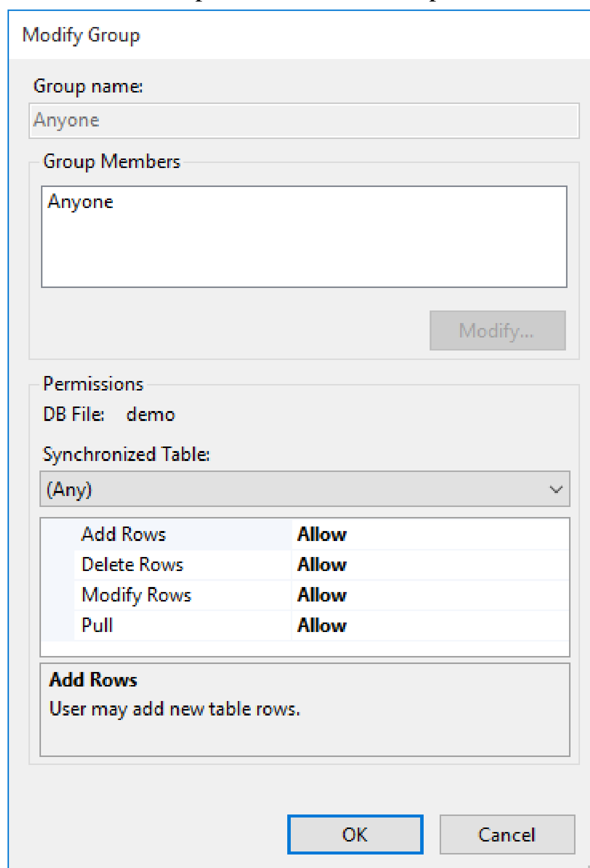
If we connect to the otherdb database and load the demo DBFile, we see the same synchronized table...



Including a column exclusion carried over from the source...



Users, filters and permissions are all in place, as well.



7.3. Audit Trails

Whenever Zumero uses rules to resolve a conflict, it alters data. For diagnostic purposes, it preserves an audit trail of all such changes. This information can be used to verify that your conflict resolution rules are behaving in the manner you expect.

The audit trail is stored in the `zumero.audit` table, located in the same database as the DBFile being synced.

Each row of the audit table contains the object_id of the host table plus four versions of the conflicting row, each one serialized as JSON.

Column	Description
`tbl`	the object_id of the host table
`ancestor`	the row as it appeared before the conflict happened
`already`	the row modified by the first client
`incoming`	the row as modified by the second client
`result`	the row with its conflict resolved

The audit trail table is designed only for the purpose of having a place for the Zumero server to keep a chronicle of any changes it makes during conflict resolution. It will INSERT rows, but it does not UPDATE or DELETE. You should not INSERT or UPDATE anything in `zumero.audit`. You may, however, DELETE rows if you want to save space.

7.4. The Server Log

The Zumero server logs all client requests into the `zumero.log` table, located in the **primary** ZSS Server database.

Column	Description
`url`	The URL requested by the client.
`ip_address`	The IP address of the client making the request
`unix_time`	The time of the request as *unix time* [a]
`request_size`	The size of the package being pushed from the client, in bytes
`response_size`	The size of the response package being sent back down to the client, in bytes
`status`	The HTTP status code of the response • 200 -- successful request, response package sent back to the client • 204 -- successful request, no response package sent back • 304 -- /pull, but nothing new to send back • 401 -- authentication failed • 403 -- permission denied • 406 -- database constraint violation
`elapsed`	The amount of time necessary to process the request, in milliseconds

[a] http://en.wikipedia.org/wiki/Unix_time

7.5. Upgrading ZSS

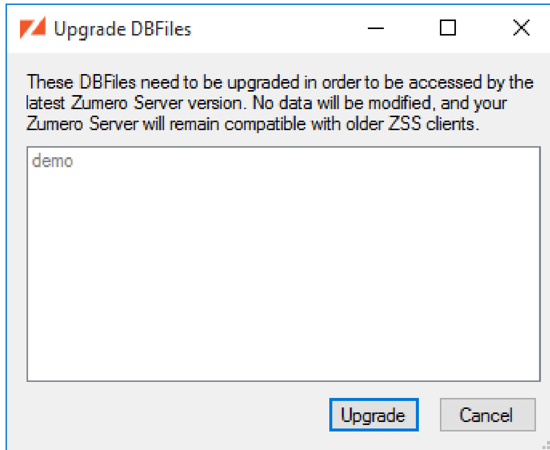
The recommended order in which to upgrade ZSS is:

1. Upgrade ZSS Manager

2. Connect to your SQL Server database with ZSS Manager, and follow the upgrade prompts (see below). If you have multiple primary databases, you may want to examine them all for upgrades.
3. Upgrade your ZSS Server.
4. Replace the client libraries that you're using. Rebuild your app and redeploy. When the client does the first sync, it should upgrade the client database.

Older clients *will* sync with a newer server, but the reverse is not true.

When you first connect your upgraded ZSS Manager to a database that was previously used with ZSS, you'll may see the Upgrade DBFiles dialog:

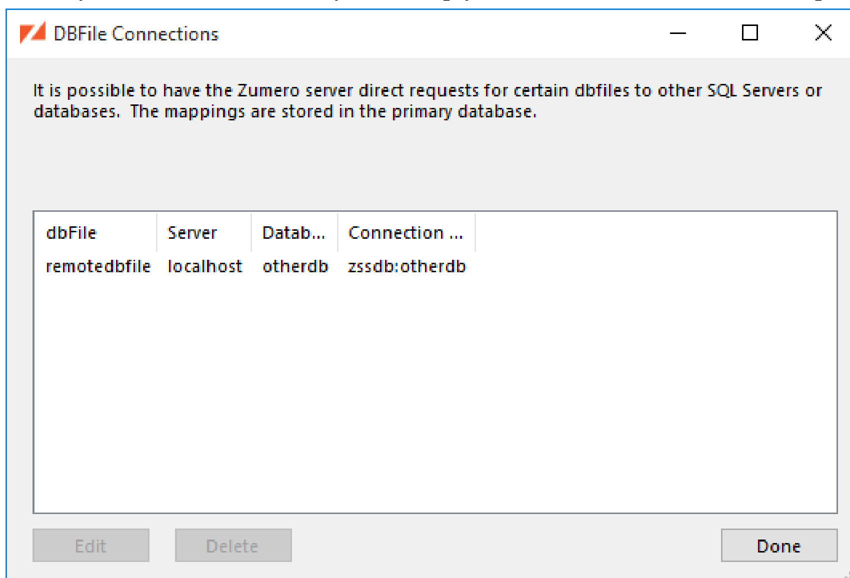


Clicking Upgrade will begin a table-by-table upgrade process, as ZSS Manager updates each table's triggers and housekeeping data. Once this process is complete, the tables are ready for use with your upgraded ZSS Server.

7.6. Editing DBFile Connection Strings

To edit a DBFile's connection information, select DBFile Connection Strings... from the DBFiles menu.

You'll see a list of DBFiles and their connection information. DBFiles in the same database as the ZSS Primary connection will usually have empty connection details; this is as expected.



Double-click a list item to edit it. There are two variations of connection details.

7.6.1. Use the Primary Database Connection

This is for DBFiles that live on the same database server as the Primary ZSS database. In these cases, you only need to edit the name of the database. ZSS Server will connect using the primary connection string, then switch to the Secondary database.

DBFile Connection

DBFile Name: remotedbfile

Use Primary Database Connection String — Connect via the Primary ZSS connection.

Use Custom Connection Details — Connect using a specific ODBC string.

Connection:

SQL Server: localhost

Database: otherdb

SQL Server Authentication

Username

Password

Connection String:

Driver={SQL Server Native Client 11.0};Server=localhost;Database=otherdb

Test | OK | Cancel

7.6.2. Custom Connection Details

This is for DBFiles that live on a different database server than the Primary ZSS database, or any DBFile for which you need to use different credentials. Here you'll build a full ODBC connection string.

DBFile Connection

DBFile Name: remotedbfile

Use Primary Database Connection String — Connect via the Primary ZSS connection.

Use Custom Connection Details — Connect using a specific ODBC string.

Connection:

SQL Server: otherserver

Database: otherdb

Windows Authentication

Username

Password

Connection String:

Driver={SQL Server Native Client 11.0};Server=otherserver;Database=otherdb;Trusted_Connec

Test | OK | Cancel

You can use the Test button to try connecting using the connection string before clicking OK to save it. Be aware that, depending on your system configuration, the connection string needed by the ZSS Server may not work from the system on which ZSS Manager is running.

ZSS Manager will still allow you to save the connect information without testing, or when the test fails.

7.7. Recovering from a Database Rollback

If your Zumero-enabled SQL Server database is ever rolled back to a previous state (e.g. by being restored from a backup), then it is possible that some of the Zumero clients will have changes that used to be on the server but have been rolled back. Furthermore, such clients may have made additional changes which were never pushed to the server.

When a call to `zumero_sync()` on a client device detects that the server was rolled back to a point in time before the client's baseline, it will send a "recovery package" to the server. By default, the server will reject the recovery package and the sync will fail with a `server_rollback_detected` error. Obviously, this situation is not ideal since the client will never be able to synchronize that local copy of the database again. Their only option would be to start a brand new client database and sync from scratch. ZSS Manager can be used to change this behavior so that the client's sync can succeed, and possibly even restore lost changes back into the server database's host tables. The settings that affect how the server and client behave after a server rollback can be accessed in ZSS Manager using the "DBFile" menu's "Database Rollback Recovery..." item.

A successful import of a recovery package will apply all rolled back INSERT, UPDATE, and DELETE operations that the client knew about in all tables in the DBFile. All of the recovery operations are applied within a single transaction and committed as an atomic operation. If that commit succeeds and the client also had additional never-pushed changes, those changes will be attempted in a second transaction.

Note

When recovering INSERT operations in a table with an IDENTITY primary key, the identity values of the inserted rows may be different than they were before the rollback.

7.7.1. Permissions

The first setting in the Rollback Recovery dialog is a permissions setting. In order for the server to import a recovery package from a client, authenticated users must be granted permission to recover the rolled back changes.

Note

Because the permissions setting in the "Rollback Recovery" dialog allows the client to modify the host tables, it is best to revert the setting (and thus deny the permission) after all rolled back changes have been recovered.

Note, though, that even if the client has permission to perform data recovery, the recovery could still fail. This is because the client might only have a filtered view of the rolled back changes, and because there might have been new changes to the database after the rollback. Thus recovery could fail due to a constraint violation or a conflict rule setting.

7.7.2. Client Sync Retry Rule

The second setting in the Rollback Recovery dialog controls what will happen whenever a recovery package is rejected or fails to import for any reason. By default, the server will send a `server_rollback_detected` error to the client, and report details about why their particular recovery package failed in the Event Viewer entry for the failed sync request. This allows the clients to preserve their local changes and retry again later.

Ideally, after you have given users permission to recover changes, then all clients will have a successful sync after all the changes have been recovered. Unfortunately, there are cases where, due to conflicts or missing changes, some clients are not able to recover. When this happens, you can adjust the client sync retry rule to "abandon changes on the client device and re-sync from the restore point." This will allow their syncs to succeed, and they will no longer be stuck getting an error.

It is recommended that when using this option, users are still given permission to recover rolled-back changes to Zumero. Without this permission, there will be no record of what changes on the client were rolled back. If, on the other hand, the user has permission to report rolled-back changes, then raw information from their failed recovery package will be left in the dbfile's `r$dump` and `r$dump_rows` tables. These rows contain JSON-serialized data from the client. Columns requiring conversion will be in their client-side SQLite format. Data in the `r$dump` and `r$dump_rows` tables is never used by Zumero itself, and can be deleted at any time.

7.7.3. Extra Client Data Cap

There is one more setting worth mentioning related to recovering from a database rollback. In order to recover from a rollback, the client needs to store a (usually) small amount of extra data in the client-side database. By default, the client will cap this extra data at "two month's worth of data" (actually sixty-five days). In other words, clients will be able to recover if the database is rolled back to a restore point up to sixty-five days in the past. If the server is rolled back to a point earlier than this "cap", clients will not be able to recover any data, and will not be able to get out of the "server_rollback_detected" error state.

There is no way to modify this cap from ZSS Manager, however administrators wanting to modify it may do so by inserting (or modifying) the rule directly in the database. The rule is stored in the `t$rules` table. Its "situation" number is one, and the "action" represents the number of days clients will store.

For example, an administrator desiring to be "safer" and have clients store 100 days worth of data for the DBFile named "my_dbfile" would issue the command:

```
INSERT INTO zumero.my_dbfile$t$rules (sit, act) VALUES (1, 100);
```

On the other hand, an administrator who is confident that 20 days is enough, and wants to save a little bit of bandwidth and client storage space would issue the command:

```
INSERT INTO zumero.my_dbfile$t$rules (sit, act) VALUES ('my_dbfile', 1, 20);
```

Finally, an administrator who does not want clients to be able to recover from a rollback and wants to use the minimum amount of bandwidth and storage could effectively turn off the feature by issuing:

```
INSERT INTO zumero.rules_dbfile (dbfile, sit, act) VALUES ('my_dbfile', 1, -1);
```

7.8. Purging History

When a table is added to a DBFile, Zumero uses the table's object_id to identify the table. A table's object_id can be found by using the `OBJECT_ID` function. For example, to find the object_id of the [items] table in the [dbo] schema you could issue the following SQL command:

```
SELECT OBJECT_ID ('dbo.items', 'U');
```

Each of Zumero's housekeeping tables for a synchronized table will be in the [zumero] schema and have the DBFile name and the table's object_id as a part of its name. One of these housekeeping tables is the `z$old` table. Each synchronized table has a `z$old` table, named DBFILE_NAME + $z$old$ + OBJECT_ID. So, for example, if our "items" table's object_id is 12345 and it is synchronized in the "foo" DBFile, it will have a `z$old` table at [zumero].[foo$z$old$12345].

The `z$old` table contains past versions of rows. Every SQL transaction that updates or deletes a row will add an entry to the `z$old` table with what the row used to be.

Rows in `z$old` are never automatically deleted by Zumero, and will persist in the database indefinitely. In order trim down the size of the database, administrators may find it necessary to delete rows out of the `z$old` table that will no longer be needed. The main difficulty is in determining which rows will no longer

be needed. The basic rule is that a row in z$old will be needed again in the future if there is any client device which currently still has that row version in its live dataset. Thus there are two basic approaches to determining what can be deleted: a date-based approach and a filter-based approach.

7.8.1. Date-Based Purging

Zumero provides a stored procedure [zumero].[PurgeDeletedRowVersionsBeforeTimestamp] which will perform date-based purging. This stored procedure will iterate over all z$old tables in the DBFile, purging all row versions that became "old" before the specified date. Here is an example of how to execute this stored procedure to purge history more than 18 months old.

```
DECLARE @t datetime2;
SELECT @t = DATEADD(month, -18, CURRENT_TIMESTAMP);
EXEC [zumero].[PurgeDeletedRowVersionsBeforeTimestamp] @dbfile='foo', @t=@t;
```

By selecting a certain date and calling PurgeDeletedRowVersionsBeforeTimestamp, you are effectively choosing a point in time at which: "All Zumero clients that have synced more recently than this date are safe, and if there are any Zumero clients that have not synced since before this date then they are at risk of having their next sync run inefficiently or get rejected entirely." A Zumero client that has not synced since before that date will only have its next sync rejected if it is trying to push an UPDATE to a row that was also modified or deleted on the server earlier than the cutoff date.

7.8.2. Filter-Based Purging

Filter-based purging only applies in situations where there is row-based filtering using one or more row inclusion WHERE clauses. Furthermore, it is only applicable when certain rows in a table are *never* sent to *any* Zumero clients. The approach is to identify which rows are never sent to any Zumero clients, and periodically delete them from z$old. Thus, this approach can be used to save space in a heavily-filtered table where most of the rows are never sent to any Zumero clients.

Again, you should only use this method to delete rows that have *never* been sent to Zumero clients. If you have a filter that does date-based calculations against a date/time column in the database, you should use date-based purging as described in the previous section, i.e. by using the PurgeDeletedRowVersionsBeforeTimestamp procedure. You should not use the filter-based approach to manually delete z$old rows that have moved out of the filterset with the passing of time.

7.8.3. Side Effects of Purging

To summarize what we have said so far, "purging history" in Zumero refers to deleting rows from z$old housekeeping tables. These rows represent past versions of rows from the host table, after they have been deleted or replaced (by an UPDATE). Deleting a given z$old row will have no side effects whatsoever as long as that version of the row does not exist on any client, which will be the case if all clients have synced recently enough to delete it or replace it with the latest (thus allowing for date-based purging) or if it was never sent to any clients at all due to row inclusion WHERE clauses (thus allowing for filter-based purging).

So what happens if a purged row version *does* exist on a client?

There are three situations that the client could be in. It could have updated its own local copy of the row, it could have deleted the row, or it could have done nothing with it.

- Even if the client did not modify or delete the row, its next sync will not be as efficient as a normal sync because the server will need to re-send the entire host table rather than just sending a delta of the incremental changes. Depending on the size of the table, this could transfer significantly more bytes across the network. (Note that this inefficiency will only occur if the table is filtered by a row inclusion WHERE clause. If, on the other hand, the table is not filtered, then a delta will still be used.)

- If the client deleted the row, there will not be any further affects beyond the inefficiency. When it pushes the delete operation to the server, the delete will be processed according to the normal conflict resolution rules (and will either be accepted and performed on the server or else ignored).
- If the client updated the row, then client's next sync will be rejected and the client will return a "purged_data_referenced" error. (Note, the Zumero client SDK did not have this error code prior to version 2.2, and a "package_rejected" error will be returned instead.)

If a client is in a state where it is getting "purged_data_referenced" errors when it tries to sync, there are a few different ways to proceed.

- The client could quarantine the change using `zumero_quarantine_since_last_sync()`. While this would allow the client to sync again, it would effectively abandon the change, as any attempt to use `zumero_sync_quarantine()` to re-send it later would fail with the same error.
- Of course there is always the most blunt tool in the toolbox, which is to have the client delete their client-side database and re-sync from scratch. This is less efficient than using `zumero_quarantine_since_last_sync()` and also abandons the entire change, but it works.
- If it is known which rows are causing the error and it is safe to delete them, the client could delete those rows, which would change the update operations into delete operations, thus allowing the push to succeed.
- Finally, special conflict resolution rules can be set on a per-table basis so that the server will ignore updates to purged rows while accepting the rest of the client's changes. There is no ZSS Manager user interface to adjust these rules, so the [zumero].[rules_record] table must be accessed directly. The magic numbers to use are "6" for the "update to purged row" situation, and "4" for "ignore". So, for example, the following SQL command could be used to set the "ignore" rule for our "items" table:

```
INSERT INTO [zumero].[rules_record] (tbl_id, sit, act, extra) VALUES (12345, 6,
 4, NULL);
```

  We advise against using this setting because its use can — without warning or error — cause loss of data (i.e. the client's change) depending on what history has been purged (which from the client's point of view is unknown and fairly arbitrary).

7.9. Adding Indexes to "z$old" Tables

When using a SQL Server profiler to examine Zumero's sync performance, your profiling tool may suggest that adding a certain index to a `z$old` table will help. Adding your own index to a `z$old` table is OK. The only caveat is that such an index may block certain schema changes on the host table, namely DROP COLUMN changes. This is because the Zumero DDL triggers will attempt to drop the same column in the `z$old` table as well, which will fail if they are included in the index. Because of this potential conflict with host table operations, we recommend that you keep custom indexes on `z$old` tables *only* if they yield a measurable or noticeable speed improvement.

8. Troubleshooting

8.1. Troubleshooting Error 500

A 500 error means that something unexpected happened on the Zumero server. The best way to find out more information about the error is to log onto the server computer and open Event Viewer. Click on the *Applications and Services Logs -> Zumero -> Admin* node. You should be able to find an event that corresponds to the 500 error. The General tab should contain more information on the error. Some common problems that can cause 500 errors

8.1.1. Mis-configured SQL Server connection

An incorrect or missing ODBC connection string will cause the server to return 500 errors. To change it, perform the following steps:

1. On the server computer, use the Start Menu to open ZSS Server Configuration.
2. Enter the connect string in the ODBC Connection String field. Clicking "Build..." will prompt you for these values and generate an ODBC string automatically.

   Example 5. An example connect string

   ```
   Driver={SQL Server Native Client 11.0};Database={YOUR DATABASE};Server={YOUR
    SQL SERVER};UID={YOUR MSSQL USER};PWD={YOUR MSSQL PASSWORD}
   ```

   Note

   Please note: the driver is important, and will almost always be `Driver={SQL Server Native Client 11.0};`

   Also note that the curly braces are necessary for fields having spaces.
3. Click the Apply button. There should be no need to restart the Zumero server

8.1.2. Insufficient permissions to the SQL Server database

If you are attempting to use Trusted_Connection in your connection string, it is possible that the permissions are not set to allow the Zumero server to connect and make changes to your database. During testing, it is often easier to use the `sa` account, as illustrated in the above example connect string.

8.2. Troubleshooting License Errors

The Zumero server can return a number of licensing errors. These errors will be returned in situations where the license or activation keys are invalid or when the server usage has exceeded limits imposed by the license key. The license errors will be returned to the clients in the sync response and will be logged in the Windows Event Viewer. To determine the exact nature of the error, log onto the server computer and open Event Viewer. Click on the *Applications and Services Logs -> Zumero -> Admin* node. You should be able to find an event that corresponds to the license error. The General tab should contain more information on the error.

8.2.1. Invalid Key Errors

These are a series of errors indicate that the license or activation key provided in the IIS site's web.config file is invalid or malformed. The most likely cause is that the license key was copy and pasted incorrectly. Locate the original copy of your license key and follow these steps:

1. On the server computer, use the Start Menu to open ZSS Server Configuration.
2. Select the IIS site the error originated from.
3. Enter the license key in the License Key field.
4. Click the Apply button. There should be no need to restart the Zumero server

If the error was not resolved by these steps contact support@zumero.com

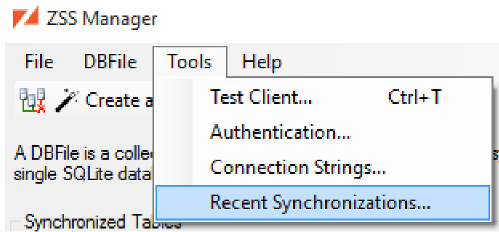
8.2.2. License User Limit Exceeded Error

This error indicates that the number of users connecting to the Zumero server exceeds the limit set in your license key. The Zumero server determines the number of users by examining recent sync activity. In most

cases this error indicates that a new license key needs to be purchased for the increased activity. To do so contact sales@zumero.com.

It is also possible that the Zumero server is still considering users who are no longer active syncing against the Zumero server. This could happen if you recently deleted users, decommissioned client devices, or changed authentication methods. In this case Zumero for SQL Server Manager provides a mechanism for removing these users from your active user list.

1. Open the Zumero for SQL Server Manager application and connect to your primary database.
2. Select *Recent Synchronizations* from the *Tools* menu.



3. Select the user you wish for Zumero to forget in the dialog and click Delete