SQL Connections

The MBS Filemaker Plugin contains functions to access SQL database servers directly. This functions are based on SQLAPI++, a C library. You can connect and run queries, execute SQL statements and work with stored procedures. We support this database types: Oracle, SQL Server, DB2, Sybase, Informix, InterBase, SQLBase, MySQL, PostgreSQL, ODBC and SQLite

By using native APIs of target DBMS, the applications developed with this plugin run swiftly and efficiently. The product also provides a low-level interface that allows developers to access database-specific features. By encapsulating a vendor's API, the plugin acts as middleware and delivers database portability. See details on supporting SQL database servers on different platforms:

Server	Windows	Mac OS X
Oracle Database Server	Supported (OCI)	Supported (OCI)
Microsoft SQL Server	Supported (DB-Library, OLE DB)	FreeTDS ODBC driver should be used
Sybase	Supported (Open Client, ASE & ASA)	Supported (Open Client, ASE & ASA)
DB2	Supported (DB2 CLI)	Supported (DB2 CLI)
Informix	Supported (Informix CLI)	Supported (Informix CLI)
InterBase/ Firebird	Supported	Supported
Centura (formerly Gupta) SQLBase	Supported (CAPI)	Supported (CAPI)
MySQL	Supported (MySQL C API)	Supported (MySQL C API)
PostgreSQL	Supported (libpq)	Supported (libpq)
ODBC	Supported	Supported (iODBC, see www.iodbc.org)
SQLite	Supported	Supported

The plugin incudes the SQLAPI library. You may need database access libraries from the database vendors.

For your information:
The MBS Filemaker Plugin license includes a cross-platform unlimited license of SQLAPI++ which costs 299 USD for a C/C++ developer. Of course this bundled license works only inside the plugin. If you want to develop with SQLAPI in C/C++, please order a separate license from SQLAPI.com website.

Steps to connect

In order to connect to a database, you need to create a new connection. For this you call **MBS("SQL.NewConnection")**. Our examples store the returned connection reference number in a variable called $Connection.

Once you have a connection, you can set options to define which database client you use and what client libraries the plugin should use. For the libraries, you normally pass file paths. Native file pathes for Mac and Windows. For example, you can use **MBS("SQL.SetConnectionOption"; $Connection; "SQLITE.LIBS"; "/usr/lib/libsqlite3.dylib")** to set client library for SQLite on Mac. Or **MBS("SQL.SetConnectionOption"; $Connection; "SQLITE.LIBS"; "c:\sqlite\sqlite3.dll")** for the same on Windows. The paths are of course different for you, so please adjust them for our example databases or your solutions.

You can tell the plugin with Connect or SetClient function about what database client to use. So you can call **MBS("SQL.SetClient"; $Connection; "SQLite")** before a connect to do other functions like querying client library version.

Now you want to connect and call the SQL.Connect function: **MBS("SQL.Connect"; $Connection; $database; $name; $pass; $client)**. Depending on what database you use, please check the server specific guides. Normally you have a database connection string which for SQLite is simply the path to the database file. Also you often have credentials which you pass for optional username and password parameters. Some databases have them inside the connection string. Also pass the client type as last parameter if you didn't call setClient before.

SQL Execute and Selects

Now you have a connection and you want to run commands on it. In order to execute commands, please create a command object with SQL.NewCommand. You can pass command here or later set it with SQL.SetCommandText. The command string can include parameters. This way you can avoid SQL injection attacks as your parameters are

probably escaped so they are not used as SQL commands by mistake. For example with an insert command like this: **MBS("SQL.NewCommand"; $Connection; "INSERT INTO 'Test' (FirstName, LastName, Birthday, NumberOfOrders, TotalSales) VALUES (:1,:2,:3,:4,:5)")** you have 5 parameters with IDs from 1 to 5. This way you can fill them with our set parameter commands like SQL.SetParamAsText.

To execute the command, simply call SQL.Execute. If your call is successful and a select command, SQL.isResultSet. In that case you can read results.

To read results, you navigate with Fetch commands through the result table. For example you can call SQL.FetchNext in a loop. As long as you get result 1 from this function, you have another row in your table. With GetField functions you can read the values. For example **MBS("SQL.GetFieldAsText"; $command; "FirstName")** reads the field named "FirstName" from the result set as text.

After you are done with your command and you don't need it again to execute another command, you can free it from memory with SQL.FreeCommand. Once you are done with the database connection, please free the connection with SQL.FreeConnection.

If you change data, please don't forget to commit data with SQL.Commit.