DB Lib

DB Lib

8 Using the Data Storage add-on Library

Getting Started

Setting the default connection id

DB Lib usually works on a default connection id. In this lesson we learn how to set it up.

Opening your database connection

DB Lib tries not to reinvent the wheel. Instead of a new fancy way to open a database connection, we just use the normal calls that we're used to. This library was created when I felt the need for better tools to work with SQLite on mobile but except for a single function, all the other methods are standard SQL and should work with any type of database.

Open your connection using **revOpenDatabase()**

```
get revOpenDatabase("sqlite", "contacts.sqlite",,,)
```

Set the default connection id

If you could sucessfuly open the connection using **revOpenDatabase()**, then you can set this connection as default using **dbSetDefaultConnectionID**

```
get revOpenDatabase("sqlite", "contacts.sqlite",,,)
if it is a number then
  dbSetDefaultConnectionID it
else
  answer error it
end if
```

After that, all the database touching functions such as dbGet, dbInsert, dbUpdate and dbDelete will default to that connection.

Selecting Records

Using dbGet() to retrieve records

This lesson describes how to use the dbGet function to retrieve records from the database

Retrieving records

The dbGet function is a database touching function, this means that it access the database. Other functions are not database touching and just set parameters to be used by the database touching functions.

Consider the following SQLite schema for a table called "contacts"

```
id INTEGET
first_name TEXT
last_name TEXT
email TEXT
```

And consider it has the following records in it:

```
1  andre  garzia  andre@andregarzia.com
2  artie  nielsen  artie@example.com
3  claudia  donovan  claude@example.com
```

If you use the following code:

```
put dbGet("contacts") into tRecordsA
```

You will end up with the following data in tRecordsA

```
tRecordsA[1]["id"] = 1
tRecordsA[1]["first_name"] = andre
tRecordsA[1]["last_name"] = garzia
tRecordsA[1]["email"] = andre@andregarzia.com

tRecordsA[2]["id"] = 2
tRecordsA[2]["first_name"] = artie
tRecordsA[2]["last_name"] = nielsen
tRecordsA[2]["email"] = artie@example.com

tRecordsA[3]["id"] = 1
tRecordsA[3]["first_name"] = claudia
tRecordsA[3]["last_name"] = donovan
```

tRecordsA[3]["email"] = claude@example.com

So ***dbGet(tableName)*** returns a nested array where the first level is a number that will go from 1 to the number of records retrieved. The second level is an array of the field names from the schema and their values.

TIP: This array structure is the same as the one used by the datagrid, basically you can set the dgData of a datagrid straight out of the output from a **dbGet** call.

This function works on the default connection unless you pass an extra parameter that is the id from another connection for example:

```
put dbGet("contacts", mySecondConnectionID) into tRecordsA
```

Will retrieve the records from another database connection.

REMEMBER: If you don't use a refinement command such as **dbWhere**, **dbGet** will retrieve all the records from the database. Check out the documentation for those commands. Thats where this library shines.

Querying a different database connection

All the database touching functions work on the default connection specified by dbSetDefaultConnectionId command but all those calls accept an extra parameter at the end that is a different database connection id. This way you can have a default connection or specify a custom one. As you can see in:

```
put dbGet("contacts") into tRecordsA
repeat with x = 1 to the number of keys in tRecordsA
  put tRecordsA[x] into tContactA
  get dbInsert("contacts", myOtherConnectionID)
end repeat
```

The first **dbGet** call picked the whole contacts table then we looped all the results callind **dbInsert** to insert each record on a different database as specified by the connection id on the variable *myOtherConnectionID*. Of course both databases need to have the same table called contacts with compatible schemas.

Using dbWhere to refine your query

In this lesson we learn how to use **dbWhere** to refine our queries. This command is used to specify filters for **dbGet** command so that it doesn't get all records from the database. It is also used by **dbDelete** and **dbInsert** to specify what are the targeted records for those calls.

Using dbWhere

The thinking behind DB Lib is that you can decompose a SQL query into various easy to plan commands and functions. This way instead of writing SQL statements by hand or using complex LiveCode commands with 15 parameters, you just use simple commands and functions that in the end generate the correct query for you.

If you know your SQL standard, then the easiest way to explain **dbWhere** is to tell you that it writes the WHERE clause for you. Lets see how that works, consider the following schema for a table called contacts:

```
id INTEGER
first_name TEXT
last_name TEXT
email TEXT
age INTEGER
country TEXT
```

And consider the following records from this table:

```
1  andre  garzia  andre@andregarzia.com  32  Brazil
2  artie  nielsen  artie@example.com  42  US
3  claudia  donovan  claude@example.com  19  US
```

As you've seen on Using dbGet() to retrieve records if you use **dbGet** function, you will end up with an array will all the records but if you want to retireve a subset of the records from the database, you can use **dbWhere** before the **dbGet** call. Lets see some examples:

```
dbWhere "country", "Brazil"
put dbGet("contacts") into tRecordsA
```

The first dbWhere call tells DB Lib that the next call to a database touching function should only match records with the *country* fields set to *Brazil*. Basically, using a **dbGet** with no **dbWhere** before it is the same as this:

```
put dbGet("contacts") into tRecordsA
```

executes the following SQL:

```
SELECT * FROM contacts;
```

While using the code:

```
dbWhere "country", "Brazil"
put dbGet("contacts") into tRecordsA
```

executes the following SQL:

```
SELECT * FROM contacts WHERE country = 'Brazil';
```

Considering the mock table data use used above, the result from this query would be:

```
tRecordsA[1]["id"] = 1
tRecordsA[1]["first_name"] = andre
tRecordsA[1]["last_name"] = garzia
tRecordsA[1]["email"] = andre@andregarzia.com
tRecordsA[1]["age"] = 32
tRecordsA[1]["country"] = Brazil
```

Think of the **dbWhere** command as setting parameters for the next SQL call. This way, you can just use simple commands that are easy to understand to build complex queries.

The default operator for **dbWhere** is the equal sign, so it matches exactly what you pass, if you need other operators, check out the next section. If you need to match partial strings then check out **dbLike** command.

Changing the operator for dbWhere

The default format for the **dbWhere** call is **dbWhere columnName, valueToMatch**. Sometimes you want to use a different operator. For example, considering the data set on the section above this one, how would one go to match all the adults? The answer is pretty simple, just add the desired operator to the *columnName parameter* as an extra word. Like this:

```
dbWhere "age >", "21"
put dbGet("contacts") into tRecordsA
```

This will execute the following SQL:

```
SELECT * FROM contacts WHERE age > 21;
```

And the result from that call will be:

```
tRecordsA[1]["id"] = 1
tRecordsA[1]["first_name"] = andre
tRecordsA[1]["last_name"] = garzia
tRecordsA[1]["email"] = andre@andregarzia.com
tRecordsA[1]["age"] = 32
tRecordsA[1]["country"] = Brazil

tRecordsA[2]["id"] = 2
tRecordsA[2]["first_name"] = artie
tRecordsA[2]["last_name"] = nielsen
tRecordsA[2]["email"] = artie@example.com
tRecordsA[2]["age"] = 42
tRecordsA[2]["country"] = US
```

Multiple dbWhere commands

The beauty of this library is the ability to decompose a complex query into a series of simple calls. You can pile up as many **dbWhere** commands as you need and when you finally call a database touching function, they will all be in the query. In the example above, we found the adults from the data set. Now suppose you want to find the adults who are from the US. The following code will do that:

```
dbWhere "age >", "21"
dbWhere "country", "US"
put dbGet("contacts") into tRecordsA
```

This will execute the following SQL:

```
SELECT * FROM contacts WHERE age > 21 AND country = 'US';
```

And the result from that call will be:

```
tRecordsA[2]["id"] = 2
```

tRecordsA[2]["first_name"] = artie
tRecordsA[2]["last_name"] = nielsen
tRecordsA[2]["email"] = artie@example.com
tRecordsA[2]["age"] = 42
tRecordsA[2]["country"] = US

By stacking commands such as **dbWhere, dbLike, dbLimit, dbGroupBy, dbOrderBy**, etc, you can create complex queries with ease.

Using dbLike to match partial strings

dbLike is similar to dbWhere but matching part of strings.

Using dbLike

In Using dbWhere to refine your query we learned how to change the target rows for a future query. dbLike is similar to dbWhere, they both write the WHERE clause of a SQL statement but while dbWhere is used to match exact values, dbLike is used to match part of strings.

Consider the following schema for a table called contacts:

```
id INTEGER
first_name TEXT
last_name TEXT
email TEXT
age INTEGER
country TEXT
```

And consider the following records from this table:

```
1  andre  garzia  andre@andregarzia.com  32  Brazil
2  artie  nielsen  artie@example.com  42  US
3  claudia  donovan  claude@example.com  19  US
```

If we want to get all records with emails containing example.com, we would use the following code:

```
dbLike "email", "example.com"
put dbGet("contacts") into tRecordsA
```

This will translate to the following SQL:

```
SELECT * FROM contacts WHERE email LIKE '%example.com%'
```

The tRecordsA array will contain:

```
tRecordsA[1]["id"] = 2
tRecordsA[1]["first_name"] = artie
tRecordsA[1]["last_name"] = nielsen
tRecordsA[1]["email"] = artie@example.com
tRecordsA[1]["age"] = 42
tRecordsA[1]["country"] = US
```

```
tRecordsA[2]["id"] = 2
tRecordsA[2]["first_name"] = claudia
tRecordsA[2]["last_name"] = donovan
tRecordsA[2]["email"] = claude@example.com
tRecordsA[2]["age"] = 19
tRecordsA[2]["country"] = US
```

As you've seen from the generated SQL, wildcards will be put around the value. In the case that you want the wildcard just before or just after the value, you can use a third parameter such as:

```
  dbLike "email", "example.com", "after"
```

will generate

```
 ... WHERE email LIKE 'example.com%'
```

or

```
  dbLike "email", "example.com", "before"
```

will generate

```
 ... WHERE email LIKE '%example.com'
```

In our example, using '*before*' is better because that will match emails endind in example.com and not emails containing it such as something@example.com.br

Like **dbWhere**, you can use as many **dbLike** commands as you want before actually calling the database. For example:

```
  dbLike "email", "example.com", "before"
  dbWhere "age >", "21"
  put dbGet("contacts") into tRecordsA
```

This will translate to the following SQL:

```
 SELECT * FROM contacts WHERE email LIKE 'example.com%' AND age > 21;
```

The tRecordsA array will contain:

```
tRecordsA[1]["id"] = 2
```

```
tRecordsA[1]["first_name"] = artie
tRecordsA[1]["last_name"] = nielsen
tRecordsA[1]["email"] = artie@example.com
tRecordsA[1]["age"] = 42
tRecordsA[1]["country"] = US
```

Ordering with dbOrderBy

The dbOrderBy command will set the ORDER BY part of the SQL statement. You can use that to set custom ordering schemes for your queries. In this little lesson we'll explore it

Basic dbOrderBy usage

Consider the following schema for a table called contacts:

```
id INTEGER
first_name TEXT
last_name TEXT
email TEXT
age INTEGER
country TEXT
```

And consider the following records from this table:

```
1  andre  garzia  andre@andregarzia.com  32  Brazil
2  artie  nielsen  artie@example.com  42  US
3  claudia  donovan  claude@example.com  19  US
```

If we use a dbGet with no refinement, our result array will reflect the order above. If we wanted to order the results by age, we would use something like:

```
dbOrderBy "age"
put dbGet("contacts") into tDataA
```

Our SQL statement will be:

```
SELECT * FROM contacts ORDER BY age;
```

And our result array will be:

```
tRecordsA[1]["id"] = 3
tRecordsA[1]["first_name"] = claudia
tRecordsA[1]["last_name"] = donovan
tRecordsA[1]["email"] = claude@example.com

tRecordsA[2]["id"] = 1
tRecordsA[2]["first_name"] = andre
tRecordsA[2]["last_name"] = garzia
tRecordsA[2]["email"] = andre@andregarzia.com
```

```
tRecordsA[3]["id"] = 2
tRecordsA[3]["first_name"] = artie
tRecordsA[3]["last_name"] = nielsen
tRecordsA[3]["email"] = artie@example.com
```

The default sort order is ascending. If you want to change it, just add DESC after the column name, such as:

```
dbOrderBy "age DESC"
put dbGet("contacts") into tDataA
```

Our SQL statement will be:

```
SELECT * FROM contacts ORDER BY age DESC;
```

And our result array will be:

```
tRecordsA[1]["id"] = 2
tRecordsA[1]["first_name"] = artie
tRecordsA[1]["last_name"] = nielsen
tRecordsA[1]["email"] = artie@example.com

tRecordsA[2]["id"] = 1
tRecordsA[2]["first_name"] = andre
tRecordsA[2]["last_name"] = garzia
tRecordsA[2]["email"] = andre@andregarzia.com

tRecordsA[3]["id"] = 3
tRecordsA[3]["first_name"] = claudia
tRecordsA[3]["last_name"] = donovan
tRecordsA[3]["email"] = claude@example.com
```

Limiting the number of results with dbLimit

There are times when we want to limit the amount of results from a database. When this needs arise, we use dbLimit

Using dbLimit to limit the amount of results

The dbLimit command will write the LIMIT part of the SQL statement.

Consider the following schema for a table called contacts:

```
id INTEGER
first_name TEXT
last_name TEXT
email TEXT
age INTEGER
country TEXT
```

And consider the following records from this table:

```
1  andre  garzia  andre@andregarzia.com  32  Brazil
2  artie  nielsen  artie@example.com  42  US
3  claudia  donovan  claude@example.com  19  US
```

If we wanted our **dbGet()** call to return just the first two records, we'd use:

```
dbLimit "2"
put dbGet("contacts") into tDataA
```

Our SQL statement will be:

```
SELECT * FROM contacts LIMIT 2;
```

And our result array will be:

```
tRecordsA[1]["id"] = 1
tRecordsA[1]["first_name"] = andre
tRecordsA[1]["last_name"] = garzia
tRecordsA[1]["email"] = andre@andregarzia.com

tRecordsA[2]["id"] = 2
tRecordsA[2]["first_name"] = artie
tRecordsA[2]["last_name"] = nielsen
```

tRecordsA[2]["email"] = artie@example.com

Just the first two items are returned.

Choosing what columns are returned with dbColumns

Sometimes you don't need all the columns from a table. You can specify what columns are returned by using dbColumns

Using dbColumns

Consider the following SQLite schema for a table called "contacts"

```
id INTEGET
first_name TEXT
last_name TEXT
email TEXT
```

And consider it has the following records in it:

```
1  andre  garzia  andre@andregarzia.com
2  artie  nielsen  artie@example.com
3  claudia  donovan  claude@example.com
```

If you use the following code:

```
put dbGet("contacts") into tRecordsA
```

You will end up with the following data in tRecordsA

```
tRecordsA[1]["id"] = 1
tRecordsA[1]["first_name"] = andre
tRecordsA[1]["last_name"] = garzia
tRecordsA[1]["email"] = andre@andregarzia.com

tRecordsA[2]["id"] = 2
tRecordsA[2]["first_name"] = artie
tRecordsA[2]["last_name"] = nielsen
tRecordsA[2]["email"] = artie@example.com

tRecordsA[3]["id"] = 1
tRecordsA[3]["first_name"] = claudia
tRecordsA[3]["last_name"] = donovan
tRecordsA[3]["email"] = claude@example.com
```

Now, suppose, all you want is the emails and ids. You can use the following code:

```
dbColumns "id,email"
put dbGet("contacts") into tRecordsA
```

dbColumns receives a comma separated list of columns. The generates SQL code looks like:

```
SELECT id,email FROM contacts;
```

The result array from that dbGet call will be:

```
tRecordsA[1]["id"] = 1
tRecordsA[1]["email"] = andre@andregarzia.com

tRecordsA[2]["id"] = 2
tRecordsA[2]["email"] = artie@example.com

tRecordsA[3]["id"] = 1
tRecordsA[3]["email"] = claude@example.com
```

Customizing the SQL statement with dbSetSQL

Sometimes you need to execute a complex SQL statement that is not covered by this library such as when you need to do a JOIN or execute some function call from the database engine itself. For all those needs, there is dbSetSQL.

Overriding the generated SQL with dbSetSQL

This command allows you to specify the SQL statement to use in the next function that touches the database.
Sometimes you need to write a complex SQL statement that is beyond what we offer with routines such as dbWhere, dbLike, dbLimit, in this cases you can still use our handy database functions but specify the SQL statement yourself.

For example:
dbSetSQL "SELECT * FROM page, tags WHERE tags.page_id = page.id"
put dbGet() into tPagesAndTagsArray

Our commands and functions cover most of the common uses for application database usage but if you need more, you can always write your own SQL. The golden rule is: if you know what a join is, then you can write it better than a library.

Inserting Records

Inserting new records with dbInsert()

In this little lesson we show how to insert new records in the database

Basic dbInsert() usage

Consider the following SQLite schema for a table called "contacts"

```
id INTEGET
first_name TEXT
last_name TEXT
email TEXT
```

And consider it has the following records in it:

```
1 andre garzia andre@andregarzia.com
2 artie nielsen artie@example.com
3 claudia donovan claude@example.com
```

The dbInsert() function inserts a new record into the database.
It uses an array where each element is a field value with the same keys as the field names
on the database schema.

```
put "ned" into tDataA["first_name"]
put "stark" into tDataA["last_name"]
put "ned@winterfell.com" into tDataA["email"]
put dbInsert("contacts", tDataA) into tResult
```

Will insert a new record with the values from the array. After calling this function, our table will be:

```
1 andre garzia andre@andregarzia.com
2 artie nielsen artie@example.com
3 claudia donovan claude@example.com
4 ned stark ned@winterfell.com
```

The result from **dbInsert()** is the number of affected rows or an error (its the same result as the one from revExecuteSQL).

REMEMBER: This function works on the default connection id unless
you specify an extra parameter with the desired connection id.

Updating Records

Updating records with dbUpdate()

dbUpdate is the functions that inserts a new record based on the elements of the given array

Using dbUpdate()

Consider the following SQLite schema for a table called "contacts"

```
id INTEGET
first_name TEXT
last_name TEXT
email TEXT
```

And consider it has the following records in it:

```
1  andre  garzia  andre@andregarzia.com
2  artie  nielsen  artie@example.com
3  claudia  donovan  claude@example.com
```

The dbUpdate() function updates an existing record in the database.
It uses an array where each element is a field value with the same keys as the field names
on the database schema.

You must use a dbWhere or a dbLike before calling dbUpdate() or the call will fail with an error. This behavior is there to protect you from updating all your records at once because you forgot to specify which one.

```
put "claudia@example.com" into tDataA["email"]
dbWhere "id", "3"
put dbUpdate("contacts", tDataA) into tResult
```

This code will generate the following SQL:

```
UPDATE contacts SET email = 'claudia@example.com' WHERE id = 3;
```

Will update a new record with the values from the array. After calling this function, our table will be:

```
1  andre  garzia  andre@andregarzia.com
2  artie  nielsen  artie@example.com
3  claudia  donovan  claudia@example.com
```

The result from **dbUpdate()** is the number of affected rows or an error (its the same result as the

one from revExecuteSQL).

REMEMBER: This function works on the default connection id unless you specify an extra parameter with the desired connection id.

Deleting Records

Deleting records with dbDelete()

dbDelete is the function that is used in conjunction to dbWhere or dbLike to remove records from the database

Basic dbDelete() usage

Consider the following SQLite schema for a table called "contacts"

```
id INTEGET
first_name TEXT
last_name TEXT
email TEXT
```

And consider it has the following records in it:

```
1  andre  garzia  andre@andregarzia.com
2  artie  nielsen  artie@example.com
3  claudia  donovan  claude@example.com
```

The dbDelete() function removes an existing record in the database.

You must use a dbWhere or a dbLike before calling dbUpdate() or the call will fail with an error. This behavior is there to protect you from deleting all your records at once because you forgot to specify which one.

```
dbWhere "id", "3"
put dbDelete("contacts", tDataA) into tResult
```

This code will generate the following SQL:

```
DELETE FROM contacts WHERE id = 3;
```

Will update a new record with the values from the array. After calling this function, our table will be:

```
1  andre  garzia  andre@andregarzia.com
2  artie  nielsen  artie@example.com
```

The result from **dbDelete()** is the number of affected rows or an error (its the same result as the one from revExecuteSQL).

REMEMBER: This function works on the default connection id unless you specify an extra parameter with the desired connection id.

Data Binding

Moving data from array to card with dbArrayToCard

Easy set fields, buttons and group data from an array

Using dbArrayToCard

This command will loop the keys of an array looking for controls with the same name in the current card. If it finds a field, button or group with the same name, it will try to replace the current value for the control with the value from the array.

consider the following array

tDataA["first_name"] = andre
tDataA["last_name"] = garzia
tDataA["country"] = Brazil

If you have a card with a field called "first_name", a field called "last_name" and a menu button called "country" which are all fields on your contacts table and you use:

```
dbArrayToCard tDataA
```

it is the same as writing:

```
set the unicodetext of field "firstName" to uniencode(tDataA["first_name"], "utf8")
set the unicodetext of field "lastName" to uniencode(tDataA["last_name"], "utf8")
set the label of button "country" to tDataA["country"]
```

This command follows these rules:
1 - it looks for a field, it there is one, then it sets the unicodetext property.
2 - it looks for a button and sets the label.
3 - it looks for a group and sets the value from the custom property dbvalue.

REMEMBER: if you're using groups for your mobile controls, just script a setprop dbvalue and a getprop dbvalue for the group to be able to exchange data with this command.

Like other database touching functions, you can use a different database connection by passing an extra parameter with the desired connection id.

Moving data from card to array with dbCardToArray

Using the schema from a table, look for card controls with names similar to the fields in the table and bundle them in an array

Using dbCardToArray

This command will look into the current card for fields, buttons and groups
with the same name as the fields on a given database table. If it finds the correct controls
it picks their values and assemble an array to be used by the database touching functions.

consider the following schema for a table called contents:

```
id INTEGER
first_name TEXT
last_name TEXT
email TEXT
country TEXT
```

If you have a card with a field called "first_name", a field called "last_name" and a menu button called "country" which are all fields on your contacts table and you use:

```
put dbCardToArray("contacts") into tDataA
```

it is the same as writing:

```
put unidecode(the unicodetext of field "first_name", "utf8") into tDataA["first_name"]
put unidecode(the unicodetext of field "last_name", "utf8") into tDataA["last_name"]
put the label of button "country" into tDataA["country"]
```

This command follows these rules:
1 - it looks for a field, it there is one, then it picks the unicodetext property and unidecodes it into the array.
2 - it looks for a button and places the label into the array.
3 - it looks for a group and places the value from the custom property dbvalue into the array.

REMEMBER: if you're using groups for your mobile controls, just script a setprop dbvalue and a getprop dbvalue for the group to be able to exchange data with this command.

Like other database touching functions, you can use a different database connection by passing an extra parameter with the desired connection id.

Avoiding Side Effects

Reseting, saving and restoring query parameters with dbResetQuery, dbPreserveQueryParameters and dbRestoreQueryParameters

Sometimes you need to clear all those dbWhere, dbLike, dbLimit and other parameter setting calls and begin with a blank slate. When you need to clear all the stored parameters for the next query, use dbResetQuery

Clearing all query parameters with dbResetQuery

By now you're pretty familiar with how the query parameter commands such as dbWhere and dbLike are used in conjunction with the database touching functions like dbGet() to manipulate your records. You may be thinking, what if I want to create a function that does something with the database but somewhere earlier in the code, someone left a hanging dbWhere, this will break my code? How can I clear those commands? The answer is dbResetQuery.

Consider the following script:

```
function getAmericans
  dbWhere "country", "US"
  put dbGet("contacts") into tReturnValueA
  return tReturnValueA
end getAmericans
```

You may think that this function is safe to use and will return the americans in the database. But what if the complete script was this one:

```
on mouseUp
  dbWhere "age >", "21"
  put getAmericans() into tAmericansA
end mouseUp

function getAmericans
  dbWhere "country", "US"
  put dbGet("contacts") into tReturnValueA
  return tReturnValueA
end getAmericans
```

Now, if that mouseUp handler is executed, it will cause a side-effect on your getAmericans call. Both dbWhere commands will be in effect, as you remember, they are only cleared after a database touching function such as dbGet. So if a dbWhere or similar parameter setting command is used elsewhere, it may still be in effect.

The solution is reset the query before executing your getAmericans() call, like this:

```
on mouseUp
  dbWhere "age >", "21"
  put getAmericans() into tAmericansA
end mouseUp

function getAmericans
  dbResetQuery
  dbWhere "country", "US"
  put dbGet("contacts") into tReturnValueA
  return tReturnValueA
end getAmericans
```

Now, the previous dbWhere call is ignored. Any parameters set before dbResetQuery are cleared.

Saving and restoring query parameters

If you want to create a self contained library that uses DB Lib, you cannot simply go reseting queries as you please. Your client/developer may have parameters set that he doesn't want to have cleared. Just imagine building your nice query workflow and in the middle of it calling some function that resets all your hard work.

Thats why there is a way to save and restore query parameters, so that you can create self contained libraries that will not cause side-effects. Check the final code here:

```
on mouseUp
  dbWhere "age >", "21"
  put getAmericans() into tAmericansA
end mouseUp

function getAmericans
  put dbPreserveQueryParameters() into tQueryParamsA
  dbResetQuery
  dbWhere "country", "US"
  put dbGet("contacts") into tReturnValueA
  dbRestoreQueryParameters tQueryParamsA
  return tReturnValueA
end getAmericans
```

Now first we save all settings for the next query to a variable tQueryParamsA, then we reset everything and do our call to find the americans. After that we restore the query to its previous state with dbRestoreQueryParameters.

With those changes, the function getAmericans() is now safe to be called anywhere and it will not cause side-effects in other database calls.

Using the Data Storage add-on Library

Introduction: A NoSQL solution for LiveCode

A bried introduction to the thinking behind the DataStorage Library

What is the Data Storage Library

This is an add-on library that builds upon DB Lib and SQLite to offer NoSQL features for LiveCode Developers. With this library, developers can store and retrieve any kind of structured information from an storage file without the need for SQL.

This is a NoSQL local solution. It doesn't offer the all the features of an RDBMS but it is very useful for storing and retrieving structured document-like information.

What NoSQL means?

The definition below is edited from wikipedia picking parts that I think apply to what the Data Storage Lib does

In computing, NoSQL (mostly interpreted as "not only SQL") is a broad class of database management systems identified by its non-adherence to the widely used relational database management system model, that is NoSQL databases are not primarily built on tables, and as a result, generally do not use SQL for data manipulation.

The following characteristics are often associated with a NoSQL database:

- It does not use SQL as its query language
- NoSQL database systems are developed to manage data that do not necessarily follow a fixed schema.
- It does not give full ACID guarantees

NoSQL database systems are often highly optimized for retrieve and append operations and often offer little functionality beyond record storage (e.g. key-value stores). The reduced run time flexibility compared to full SQL systems is compensated by significant gains in scalability and performance for certain data models.

In short, NoSQL database management systems are useful when working with a huge quantity of data and the data's nature does not require a relational model for the data structure. The data could be structured, but it is of minimal importance and what really matters is the ability to store and retrieve great quantities of data, and not the relationships between the elements. For example,

to store millions of key-value pairs in one or a few associative arrays or to store millions of data records. This is particularly useful for statistical or real-time analyses for growing list of elements (such as Twitter posts or the Internet server logs from a big group of users).

What Can I Store with Data Storage Library?

This library is optmized to store and retrieve multi-dimensional arrays. You organize your data into the elements of an array in any way you desire and tell the database to store this array. It will do so and will report back a key that you can use to retrieve that array back again.

With auxiliary routines to list keys in storage, you can quickly create map/reduce algorithms to work thru your storage cherry picking what items you need.

You system will not have the same performance as one using a hand tailored SQL database but it will be very easy to maintain and work with and since this library is just for local storage access, the performance hit will be very small since computers are fricking fast today.

Using the Data Storage add-on Library

In this lesson we do a quick review of the steps, commands and functions of the Data Storage Library

Start using both DB Lib and Data Storage Lib

The Data Storage Lib uses DB Lib so you must put both into use before trying to use the NoSQL routines. You can do that with a command similar to:

```
start using stack "aagDBLib.livecode"
start using stack "aagDataStorageLib.livecode"
```

If you never used libraries before, please take a look at Extending The Message Path article by Richard Gaskin and the documentation for the start using command.

After those two libraries are in the stacksInUse we're ready to proceed.

Opening a storage

Each storage is a file on disk. The library creates this file for you if it doesn't exists. Each storage file is a SQLite file and is saved in a safe place for each possible operating system. To open a storage file you need to pass your storage name and your application bundle id. This bundle id is used to create a folder to save the storage files with the same name in some operating systems (Mac OS X, Windows, Linux).

To open a storage you use the dsOpen command, like this:

```
dsOpen "tests", "com.andregarzia.dataStorageLib"
```

On my Mac OS X this will create the following file **~/Library/Application Support/com.andregarzia.dataStorageLib/tests.sqlite** and in this file, the items will be saved. If there is an error opening the storage, a string is returned beginning with "**dserr,**".

After your storage is open, you can refer to it by name on the other calls.

Saving an item

Each item saved into the storage has a corresponding key. When you save an item, if everything went fine, you will receive a key back. If there is an error, you will receive an error beginning with "dserr" or "dberr" depening if the error happened on the Data Storage Lib or DB Lib.

To save an item, assemble your data array and pass it to the dsSave command. Like this:

```
put "blue" into tA["color"]
put "32" into tA["age"]
dsSave "tests", tA
put the result into tLastKey
```

The key is auto-generated for you and looks like "item_80770293450_5534". You don't need to remember these keys because there are routines to list all keys on storage.

If you don't want an auto-generated key, you can pass an element called "key" on your array and then that key will be used. Be aware that keys need to be unique. If you pass a key that is already on the database, the item will be replaced.

Retrieving an item

To retrieve an item you use the **dsGet()** function. You just pass the storage name and the key and you will receive your data array back if the key is found.

```
put dsGet("tests", tLastKey) into tA
```

In the example above, the tA array would contain the elements saved on the previous section.

Listing keys

As you add items to storage, you will not remember the keys and soon you will need to list them. To solve that you use the **dsKeys()** function. This function returns a return delimited list of keys. As in:

```
put dsKeys("tests") into tKeys
if tLastKey is among the lines of tKeys then
  put "found our key"
else
  put "where is the data we just inserted?!."
end if
```

This function is useful to build loops with the repeat command. These loops can build a map/reduce like workflow where you loop getting all items in your storage and then cherry pick what you need.

Checking if a key exists

You can use the dsKeyExists() function to check if a given key is present on storage or not. At first glance this is not that useful in a world of auto-generated keys but if you mix your auto-generated keys with hard coded ones, this function becomes really useful.

For example, suppose you want to save preferences for your application. Just assemble an array with all your preferances and save it with a key called "preferences". Then in other places you can check to see if preferences have been set by using dsKeyExists().

```
put "14" into tPreferencesA["font_size"]
put "Arial" into tPreferencesA["font_name"]
put "preferences" into tPreferencesA["key"]
dsSave "tests", tPreferencesA
---
--- Later in the software
---
if dsKeyExists("tests", "preferences") then
  -- preferences have been set
else
  -- preferences have NOT been set
end if
```

Emptying the storage

If you want to destroy all items in a storage, you call dsNuke. Everything will be deleted.

```
dsNuke "tests"
```

Inserting a batch of items

You can use dsBatchSave command to to batch save operations. You pass the storage name and a batch data array. This array needs to have on the first level, numeric keys going from 1 to N where N is the number of records and on the second level the items you want to save.

For example:

```
put empty into tA
repeat with x = 1 to 10
  put "item_" & x into tA[x]["key"]
  put "this is item" & x into tA[x]["data"]
```

```
end repeat
dsBatchSave "tests", tA
```

If there is an error inserting one of the items, the batch save operation will abort and an error beginning with **dsErr** will be in **the result.**

Getting all items in storage

If you're building repeat loops going thru all items in a storage, there is a way that is easier than calling dsKeys() and looping thru them. You can use dsGetAll() to retrieve all items in a given storage.

```
put dsGetAll("tests") into tA
put the keys of tA into tKeys
sort numeric ascending tKeys
repeat for each line x in tKeys
  if tA[x]["key"] is not ("item_" & x) or tA[x]["data"] is not ("this is item" & x) then
    put "Retrieved wrong record back, record #" & x
  else
    put "dsGetAll: record #" & x && "OK"
  end if
end repeat
```