LUA - DATABASE ACCESS

http://www.tutorialspoint.com/lua/lua_database_access.htm Copyright © tutorialspoint.com

For simple data operations, we may use files, but, sometimes, these file operations may not be efficient, scalable, and powerful. For this purpose, we may often switch to using databases. LuaSQL is a simple interface from Lua to a number of database management systems. LuaSQL is the library, which provides support for different types of SQL. This include,

- SQLite
- Mysql
- ODBC

In this tutorial, we will be covering database handling of MySQL an SQLite in Lua. This uses a generic interface for both and should be possible to port this implementation to other types of databases as well. First let see how you can do the operations in MySQL.

MySQL db Setup

In order to use the following examples to work as expected, we need the initial db setup. The assumptions are listed below.

- You have installed and setup MySQL with default user as root and password as '123456'.
- You have created a database test.
- You have gone through MySQL tutorial to understand **MySQL Basics.**

Importing MySQL

We can use a simple **require** statement to import the sqlite library assuming that your Lua implementation was done correctly.

```
mysql = require "luasql.mysql"
```

The variable mysql will provide access to the functions by referring to the main mysql table.

Setting up Connection

We can set up the connection by initiating a MySQL environment and then creating a connection for the environment. It is shown below.

```
local env  = mysql.mysql()
local conn = env:connect('test','root','123456')
```

The above connection will connect to an existing MySQL file and establishes the connection with the newly created file.

Execute Function

There is a simple execute function available with the connection that will help us to do all the db operations from create, insert, delete, update and so on. The syntax is shown below −

```
conn:execute([[ 'MySQLSTATEMENT' ]])
```

In the above syntax, we need to ensure that conn is open and existing MySQL connection and replace the 'MySQLSTATEMENT' with the correct statement.

Create Table Example

A simple create table example is shown below. It creates a table with two parameters id of type integer and name of type varchar.

```
mysql = require "luasql.mysql"

local env  = mysql.mysql()
local conn = env:connect('test','root','123456')

print(env,conn)

status,errorString = conn:execute([[CREATE TABLE sample2 (id INTEGER, name TEXT);]])
print(status,errorString )
```

When you run the above program, a table named sample will be created with two columns namely, id and name.

```
MySQL environment (004BB178) MySQL connection (004BE3C8)
0 nil
```

In case there is any error, you would be returned an error statement instead of nil. A simple error statement is shown below.

```
LuaSQL: Error executing query. MySQL: You have an error in your SQL syntax; check the
manual that corresponds to your MySQL server version for the right syntax to use near
'"id INTEGER, name TEXT)' at line 1
```

Insert Statement Example

An insert statement for MySQL is shown below.

```
conn:execute([[INSERT INTO sample values('11','Raj')]])
```

Update Statement Example

An update statement for MySQL is shown below.

```
conn:execute([[UPDATE sample3 SET name='John' where id ='12']])
```

Delete Statement Example

A delete statement for MySQL is shown below.

```
conn:execute([[DELETE from sample3 where id ='12']])
```

Select Statement Example

As far as select statement is concerned, we need to loop through each of the rows and extract the required data. A simple select statement is shown below.

```
cursor,errorString = conn:execute([[select * from sample]])
row = cursor:fetch ({}, "a")

while row do
   print(string.format("Id: %s, Name: %s", row.id, row.name))
   -- reusing the table of results
   row = cursor:fetch (row, "a")
end
```

In the above code, conn is an open MySQL connection. With the help of the cursor returned by the execute statement, you can loop through the table response and fetch the required select data.

A Complete Example

A complete example including all the above statements is given below.

```
mysql = require "luasql.mysql"

local env  = mysql.mysql()
local conn = env:connect('test','root','123456')
print(env,conn)

status,errorString = conn:execute([[CREATE TABLE sample3 (id INTEGER, name TEXT)]])
print(status,errorString )

status,errorString = conn:execute([[INSERT INTO sample3 values('12','Raj')]])
print(status,errorString )

cursor,errorString = conn:execute([[select * from sample3]])
print(cursor,errorString)

row = cursor:fetch ({}, "a")

while row do
   print(string.format("Id: %s, Name: %s", row.id, row.name))
   row = cursor:fetch (row, "a")
end

-- close everything
cursor:close()
conn:close()
env:close()
```

When you run the above program, you will get the following output.

```
MySQL environment (0037B178) MySQL connection (0037EBA8)
0 nil
1 nil
MySQL cursor (003778A8) nil
Id: 12, Name: Raj
```

Performing Transactions

Transactions are a mechanism that ensures data consistency. Transactions should have the following four properties −

- **Atomicity** − Either a transaction completes or nothing happens at all.
- **Consistency** − A transaction must start in a consistent state and leave the system in a consistent state.
- **Isolation** − Intermediate results of a transaction are not visible outside the current transaction.
- **Durability** − Once a transaction was committed, the effects are persistent, even after a system failure.

Transaction starts with START TRANSACTION; and ends with commit or rollback statement.

Start Transaction

In order to initiate a transaction, we need to execute the following statement in Lua, assuming conn is an open MySQL connection.

```
conn:execute([[START TRANSACTION;]])
```

Rollback Transaction

We need to execute the following statement to rollback changes made after start transaction is executed.

```
conn:execute([[ROLLBACK;]])
```

Commit Transaction

We need to execute the following statement to commit changes made after start transaction is executed.

```
conn:execute([[COMMIT;]])
```

We have known about MySQL in the above and following section explains about basic SQL operations. Remember transactions, though not explained again for SQLite3 but the same statements should work for SQLite3 as well.

Importing SQLite

We can use a simple require statement to import the SQLite library assuming that your Lua implementation was done correctly. During installation, a folder libsql that contains the database related files.

```
sqlite3 = require "luasql.sqlite3"
```

The variable sqlite3 will provide access to the functions by referring to the main sqlite3 table.

Setting Up Connection

We can set up the connection by initiating an SQLite environment and then creating a connection for the environment. It is shown below.

```
local env  = sqlite3.sqlite3()
local conn = env:connect('mydb.sqlite')
```

The above connection will connect to an existing SQLite file or creates a new SQLite file and establishes the connection with the newly created file.

Execute Function

There is a simple execute function available with the connection that will help us to do all the db operations from create, insert, delete, update and so on. The syntax is shown below −

```
conn:execute([[ 'SQLite3STATEMENT' ]])
```

In the above syntax we need to ensure that conn is open and existing sqlite3 connection and replace the 'SQLite3STATEMENT' with the correct statement.

Create Table Example

A simple create table example is shown below. It creates a table with two parameters id of type integer and name of type varchar.

```
sqlite3 = require "luasql.sqlite3"

local env  = sqlite3.sqlite3()
local conn = env:connect('mydb.sqlite')
print(env,conn)

status,errorString = conn:execute([[CREATE TABLE sample ('id' INTEGER, 'name' TEXT)]])
print(status,errorString )
```

When you run the above program, a table named sample will be created with two columns namely, id and name.

```
SQLite3 environment (003EC918) SQLite3 connection (00421F08)
0 nil
```

In case of an error, you would be returned a error statement instead of nil. A simple error statement is shown below.

```
LuaSQL: unrecognized token: """'id' INTEGER, 'name' TEXT)"
```

Insert Statement Example

An insert statement for SQLite is shown below.

```
 conn:execute([[INSERT INTO sample values('11','Raj')]])
```

Select Statement Example

As far as select statement is concerned, we need to loop through each of the rows and extract the required data. A simple select statement is shown below.

```lua
cursor,errorString = conn:execute([[select * from sample]])
row = cursor:fetch ({}, "a")

while row do
   print(string.format("Id: %s, Name: %s", row.id, row.name))
   -- reusing the table of results
   row = cursor:fetch (row, "a")
end
```

In the above code, conn is an open sqlite3 connection. With the help of the cursor returned by the execute statement, you can loop through the table response and fetch the required select data.

A Complete Example

A complete example including all the above statements is given below.

```lua
sqlite3 = require "luasql.sqlite3"

local env  = sqlite3.sqlite3()
local conn = env:connect('mydb.sqlite')
print(env,conn)

status,errorString = conn:execute([[CREATE TABLE sample ('id' INTEGER, 'name' TEXT)]])
print(status,errorString )

status,errorString = conn:execute([[INSERT INTO sample values('1','Raj')]])
print(status,errorString )

cursor,errorString = conn:execute([[select * from sample]])
print(cursor,errorString)

row = cursor:fetch ({}, "a")

while row do
   print(string.format("Id: %s, Name: %s", row.id, row.name))
   row = cursor:fetch (row, "a")
end

-- close everything
cursor:close()
conn:close()
env:close()
```

When you run the above program, you will get the following output.

```
SQLite3 environment (005EC918) SQLite3 connection (005E77B0)
0 nil
1 nil
SQLite3 cursor (005E9200) nil
```

```
Id: 1, Name: Raj
```

We can execute all the available queries with the help of this libsql library. So, please don't stop with these examples. Experiment various query statement available in respective MySQL, SQLite3 and other supported db in Lua.