Lua-PgSQL

A lightweight binding of the libpq client library for PostgreSQL

2011.07.31

License

Lua-PgSQL is provided under the **simplified BSD license**:

Overview

Lua-PgSQL is a purpose built Lua module designed for the fast and efficient manipulation of a **PostgreSQL** database using the official **libpq** client libraries. As this is a lightweight binding, some of this documentation is sourced directly from the PostgreSQL manual. The current version can be downloaded from here:

http://luaforge.net/projects/luapgsql/

The base functions only serve to configure the client libraries and provide a **connection object** that is used to prepare data and execute SQL commands. Once a command is executed, a **result object** is created. This object is used to perform all of the data retrieval operations.

The object methods are arranged where more functionality is provided the further up the hierarchy you go. This allows for identical libraries to be made for almost any database. One could imagine a Lua-MySQL binding built from these sources with relatively minor modifications...

Requirements:

- Lua 5.1+
- PostgreSQL 8.2+ Client

Base functions

pg.connect(*connection string*)

Makes a new connection to the database server.

Returns a connection object and a error message. If there is an error the connection object will be **nil** and the error message will be in plain text.

This function opens a new database connection using the parameters taken from the connection string. The passed string can be empty to use all default parameters, or it can contain one or more parameter settings separated by whitespace. Each parameter setting is in the form `keyword = value`. (To write an empty value or a value containing spaces, surround it with single quotes, e.g., `keyword = 'a value'`. Single quotes and backslashes within the value must be escaped with a backslash, i.e., `\'` and `\\`.) Spaces around the equal sign are optional.

```
con, errmsg = pg.connect("host=localhost dbname=foobar user=foo password=bar")
if not con then
      print(string.format("Connection Error: %s", errmsg))
end
```

The currently recognized parameter key words are:

`host`

> Name of host to connect to. If this begins with a slash, it specifies Unix-domain communication rather than TCP/IP communication; the value is the name of the directory in which the socket file is stored. The default is to connect to a Unix-domain socket in `/tmp`.

`hostaddr`

> Numeric IP address of host to connect to. This should be in the standard IPv4 address format, e.g., `172.28.40.9`. If your machine supports IPv6, you can also use those addresses. TCP/IP communication is always used when a nonempty string is specified for this parameter.
>
> Using `hostaddr` instead of `host` allows the application to avoid a host name look-up, which may be important in applications with time constraints. However, Kerberos authentication requires the host name. The following therefore applies: If `host` is specified without `hostaddr`, a host name lookup occurs. If `hostaddr` is specified without `host`, the value for `hostaddr` gives the remote address. When Kerberos is used, a reverse name query occurs to obtain the host name for Kerberos. If both `host` and `hostaddr` are specified, the value for `hostaddr` gives the remote address; the value for `host` is ignored, unless Kerberos is used, in which case that value is used for Kerberos authentication. (Note that authentication is likely to fail if libpq is passed a host name that is not the name of the machine at `hostaddr`.) Also, `host` rather than `hostaddr` is used to identify the connection in `$HOME/.pgpass`.
>
> Without either a host name or host address, libpq will connect using a local Unix domain socket.

`port`

> Port number to connect to at the server host, or socket file name extension for Unix-domain connections.

`dbname`

> The database name. Defaults to be the same as the user name.

`user`

PostgreSQL user name to connect as.

`password`

Password to be used if the server demands password authentication.

`connect_timeout`

Maximum wait for connection, in seconds (write as a decimal integer string). Zero or not specified means wait indefinitely. It is not recommended to use a timeout of less than 2 seconds.

`options`

Command-line options to be sent to the server.

`sslmode`

This option determines whether or with what priority an SSL connection will be negotiated with the server. There are four modes: `disable` will attempt only an unencrypted SSL connection; `allow` will negotiate, trying first a non-SSL connection, then if that fails, trying an SSL connection; `prefer` (the default) will negotiate, trying first an SSL connection, then if that fails, trying a regular non-SSL connection; `require` will try only an SSL connection.

If PostgreSQL is compiled without SSL support, using option `require` will cause an error, and options `allow` and `prefer` will be tolerated but libpq will be unable to negotiate an SSL connection.

`service`

Service name to use for additional parameters. It specifies a service name in `pg_service.conf` that holds additional connection parameters. This allows applications to specify only a service name so connection parameters can be centrally maintained. See *`PREFIX`*`/share/pg_service.conf.sample` for information on how to set up the file.

If any parameter is unspecified, then the corresponding environment variable is checked. If the environment variable is not set either, then built-in defaults are used.

Connection Object

con:*escape(string)*

Escapes a string for safe use in SQL queries.

Returns a properly escaped string.

Use this method to escape a string that you will be using in a SQL query without parameters. Note that using parameters in your queries is **highly** recommended as it is more efficient and secure. **Do not** use this method when executing a SQL command with parameters.

con:exec(*command, {parameters}, {param count}*)

Submits a command to the server and waits for the result, with the optional ability to pass parameters separately from the SQL command text.

Returns a result object and a error message. If there is an error the connection object will be **nil** and the error message will be in plain text.

When not using parameters, it is allowed to include multiple SQL commands (separated by semicolons) in the command string. Multiple queries sent in a single call are processed in a single transaction, unless there are explicit `BEGIN/COMMIT` commands included in the query string to divide it into multiple transactions. Note however that the returned result object includes only the result of the last command executed from the string. Should one of the commands fail, processing of the string stops with it and the returned error message describes the error condition.

When using parameters, PostgreSQL allows at most one SQL command in the given string. (There can be semicolons in it, but not more than one nonempty command.) This is a limitation of the underlying protocol, but has some usefulness as an extra defense against SQL-injection attacks. Parameters are referred to in the command string as `$1`, `$2`, etc. If you specify the parameter count, then SQL NULLS will be used when Lua **nils** are passed as parameter values.

```
-- simple query with no parameters
rs, errmsg = con:exec("select * from foobar")
if not rs then
      print(string.format("SQL Error: %s", errmsg))
end
```

```
-- query with parameters
rs, errmsg = con:exec("select * from foobar where foo = $1 and bar = $2", {3, "B"})
if not rs then
      print(string.format("SQL Error: %s", errmsg))
end
```

con:notifywait({timeout})

Wait for any NOTIFY message from server. Timeout (in seconds) can be specified.

Returns count of received notifications from server.

Example, listener process:

```
con:exec("LISTEN somenotify")
-- wait indefinitely
con:notifywait()
```

Example, notifier process:

```
con:exec("NOTIFY somenotify")
```

More information about NOTIFY:

http://www.postgresql.org/docs/current/interactive/sql-notify.html

con:close()

Closes the connection to the server. Also frees memory used by the libpq library.

This method is optional and the use of it is **discouraged**, as a semaphore-based invocation of this is performed within Lua's standard garbage collection. This function will be automatically called when the connection object goes out of scope or is set to **nil**.

Result Object

rs:count()

Returns the rows found/affected by the SQL command.

If the SQL command that generated the result was a `SELECT`, this will return the number of rows in the query result. If the SQL command was `INSERT`, `UPDATE`, `DELETE`, `MOVE`, or `FETCH`, this returns the number of rows affected. If the command was anything else, it returns 0.

rs:fetch()

Retrieves the current row from the result set.

A table with the data for the current row indexed by both column number and column name is returned. Result data is returned in native Lua types as follows: null = nil, bool = boolean, int2/int4/int8/float4/float8/numeric = number, all else = string using the current database and connection default formatting. If there are no more rows available, **nil** is returned.

```
d = rs:fetch()
while d then
      print(string.format("foo = %s, bar = %s", d[2], d[2]))
      print(string.format("foo = %s, bar = %s", d.foo, d.bar))
      d = rs:fetch()
end
```

rs:cols()

Result object column iterator for use in generic "for" loops.

This is an iterator that is used to retrieve the column schema for a result set. A numeric column index (starting with 1) and column name is returned. Column names are returned as Lua strings, using the current database and connection default formatting.

```
for i, n in rs:cols() do
      print(string.format("column number = %d, column name = %d", i, n))
end
```

rs:rows()

Result object row iterator for use in generic "for" loops.

This is an iterator that is used to retrieve the rows for a result set. A table with the data for the current row indexed by both column number and column name is returned. Result data is returned in native Lua types as follows: null = nil, bool = boolean, int2/int4/int8/float4/float8/numeric = number, all else = string using the current database and connection default formatting.

```
-- access by column number
for d in rs:rows() do
      for i, v in ipairs(d) do
            print(string.format("column number = %d, value = %d", i, v))
      end
end
```

```
-- access by column name
for d in rs:rows() do
      print(string.format("foo = %s, bar = %s", d.foo, d.bar))
end
```

rs:clear()

Frees the storage associated with a result set.

This method is optional and the of it use is **discouraged**, as a semaphore-based invocation of this is performed within Lua's standard garbage collection. This function will be automatically called when the result object goes out of scope or is set to **nil**.