LinuxFocus article number 331
http://linuxfocus.org

by Diego Alberto Arias Prad
<dariapra(at)yahoo.es>

About the author:

I am a Telecommunication engeneer based in Lugo, Spain. I cannot exactly remember when I started to use Linux, it was in 1995 or 1996. Before that, I was a Microsoft Windows user and I even didn't know that Linux existed. The first time I saw a computer running Linux was at the university. It looked very interesting to me and soon I installed it on my computer at home; I remember that my first Linux distro was Slackware.

In all these years I have had a lot of fun using other Linux distros and some BSD "flavours", using programming languages like Java or Tcl, using database, web and application servers... Linux

An introduction to the TclMySQL library

Abstract:

In this article you will learn how to install and use MySQLTcl, a Tcl library which makes possible to do SQL queries (select, insert, delete...) to a MySQL database server from Tcl scripts. The versions of Tcl, MySQL server and the MySQLTcl library covered in this article are respectively 8.4.2, 4.0.15 and 2.40.

Tcl stands for Tool Command Language and was invented by John Ousterhout [1]. Tcl is actually two things: a scripting language and an interpreter. Tcl is a structured programming language which uses three basic data structures: strings, lists and arrays. Features of Tcl include regular expressions [2], third party Tcl extention libraries and Tk, a toolkit for writing graphical applicactions in Tcl.

MySQL is a very popular database server in the open software comunity which I think it needs no presentation.

MySQLTcl is a Tcl library which allows querying a MySQL database server from a Tcl script. Currently, the authors and mantainers of this Tcl library are Paolo Brutti (Paolo.Bruti at tlsoft.it), Tobias Ritzau (tobri at ida.liu.se) and Artur Trzewick (mail at xdobry.de).

has not just meant joy for
me, I have had also the
chance of using Linux when
I worked at Telefónica I+D.

MySQLTcl library installation

If your Linux distro or your *BSD operating system has support for packages (like RPM or DEB, for example) or ports (Crux Linux or FreeBSD, for example), you can use the package or ports system to install the MySQLTcl library and skip this section.

If it is not the case or you simply prefer to install "by hand", in the following lines I show the steps I followed. These lines should be viewed as a guideline and not a step-by-step installation manual. In a Linux Mandrake *distro* (version 9.2), from bash:

```
$ ./configure --with-mysql-lib=/usr/lib
$ make
$ make install
```

If something goes wrong during the "configure" step, the error information will provided clues to fix the problem. Usually, the problem is that the *configure* script is unable to find certain directories or files. In cases like this one, you can "play" with this script by passing it parameters telling where the missing files or directories are. Let’s see another example. When I used FreeBSD 5.0, I installed the MySQLTcl with these options:

```
$ export CPP=/usr/bin/cpp
$ ./configure --with-tcl=/usr/lib/local/tcl8.3
  --with-tclinclude=/usr/local/include/tcl8.3
  --with-mysql-include=/usr/local/include/mysql
  --with-mysql-lib=/usr/local/lib/mysql
$ make
$ make install
```

As you must have noted, in this second example the version of Tcl was 8.3; besides, the version of the MySQLTcl library was 2.15 and MySQL database server version was 3.23.54.

Tcl Basics

In this section I briefly introduce some Tcl basics for readers interested in this article who cannot program in Tcl. If you are already a Tcl programer, you can skip this section.

You can reproduce the examples shown in this (and also the following) sections from the Tcl shell (tclsh).

Variables. Command and variable substitution.

Tcl variables are created with the command *set*. Let's see some examples:

```
% set address {Edison Avenue, 83}
Edison Avenue, 83
% set zip_code 48631
48631
```

In these two examples, we have created two variables, named address and zip_code. The values contained by these variables are, respectevely, *Edison Avenue*, 83 and *48361*; both values are strings. Note that for creating the variable address, curly braces have been used because the string has white spaces. Variable values can be retrieved using the *set* command:

```
% set address
Edison Avenue, 83
% set zip_code
48631
```

Supose we want to print on the screen the value of the variable address. This can be done with the command *puts*:

```
% puts stdout [set address]
Edison Avenue, 83
```

The parameter *stdout* is passed to the *puts* command. This parameter tells the command *puts* to print on the standard output, in our case the screen. The second parameter passed to the *puts* command is *[set address]*. The square brackets in the second parameter inform the Tcl interpreter that the date inside the brackets is another Tcl command which must be executed by the Tcl interpreter before the *puts* command; this is called command substitution. The same can be done in another way:

```
% puts stdout $address
Edison Avenue, 83
```

In this example what we have done is called variable substitution: the characer *$* preceding a variable name makes variable substitution happen.

In a previous example we have seen that by using curly braces words separated by white spaces could be grouped in a string. Another form of grouping is using double quotes (character "). However, these two forms of grouping don't work exactly the same way. Let's see an example:

```
% puts stdout "the zip code is [set address]"
the zip code is 48631
% puts stdout "the zip code is $address"
the zip code is 48631
% puts stdout {the zip code is [set address]}
the zip code is [set address]
% puts stdout {the zip code is $address}
the zip code is $address
```

In this example you can see that using curly braces for grouping command and variable substitution will not happen; however, they will happen if double quotes are used for grouping.

Varibles are deleted with the *unset* command:

```
% unset address
```

```
% set address
can’t read "address": no such variable
% unset zip_code
% set zip_code
can’t read "zip_code": no such variable
```

Tcl Strings

The string data structure is one of the three basic Tcl data structures. A string is a set of characters. A string can be created with the *set* command.

```
% set surname Westmoreland
Westmoreland
% set number 46.625
46.625
```

Both *surname* and *number* variables are strings. Strings can be manipulated by using the *string* command. The general syntax of the *string* command is *string operation stringvalue otherargs*. Let’s see some examples that show how this command can be used:

```
% string length $surname
12
% set surname [string range $surname 0 3]
West
% puts stdout $surname
West
% string tolower $surname
west
```

Unlike Java or Pascal, Tcl is not a strong-typed programming language. The following example shows this:

```
% set number [expr $number + 24.5]
70.125
% string range $number 2 5
.125
```

With the command *expr*, the value of the variable *number* was increased 24.5. After that, with the command *string* the variable *number* was treated as a string and then the last four characters were shown.

More string operations than shown in the previous example are possible.

Tcl Lists

Tcl lists are a special case of strings in which list elements are separated by white spaces and have an special interpretation.

```
% set friends_list {Fany John Lisa Jack Michael}
Fany John Lisa Jack Michael
```

```
% set friends_list [string tolower $friends_list]
fany john lisa jack michael
% set friends_list
fany john lisa jack michael
```

There are a number of Tcl commands that allow list manipulation. Let’s see some examples:

```
% lindex 2 $friends_list
lisa
% lrange $friends_list 2 4
lisa jack michael
% set friends_list [lsort -ascii $friends_list]
fany jack john lisa michael
% set friends_list
fany jack john lisa michael
% set friends_list [linsert $friends_list 2 Peter]
fany jack Peter john lisa michael
% string toupper $friends_list
FANY JACK PETER JOHN LISA MICHAEL
```

The last example shows that strings and lists are actually the same data structure.

Tcl Arrays

An array can be seen as a list with a string-valued index. An array can be created like shown in the following example:

```
% set phone_numbers(fany) 629
629
% set phone_numbers(michael) 513
513
% set phone_numbers(john) 286
286
```

Array values can be retrieved using the *set* command and variable substitution; as shown in the previous example, the string-valued index is delimited by parenthesis.

```
% set $phone_numers(michael)
513
```

The command *array* returns information about an array variable. Let’s see some examples that show what this command can do:

```
% array size phone_numbers
3
% array names phone_numbers
fany john michael
```

Database connection

Before making any query to a database from a Tcl script, it is necessary to establish a connection to the database server. In this section we will see how to do it and how to handle errors that can happen while trying to establish the database connection.

Establising a database connection

Let's see an example of a Tcl script that establishes a connection to a MySQL database server:

```
0:  #!/usr/bin/tclsh8.4
1:
2:  package require mysqltcl
3:  global mysqlstatus
4:
5:  set port {3306}
6:  set host {127.0.0.1}
7:  set user {john_smith}
8:  set password {jsmith_password}
9:
10: set mysql_handler [mysqlconnect -host $host
    -port $port -user $user -password $password]
11:
12: mysqlclose $mysql_handler
```

Please, note that the left-column numbers and its following colon aren't part of the Tcl script; its only purpose is labeling the Tcl script lines. Also note that depending on the Linux distro you use, you may have to modify the line #0 to set a correct path to the Tcl shell.

The line #0 tells the shell that this file is a Tcl script and where to find the Tcl interpreter.

The line #2 tells the Tcl interpreter to search in the library MySQLTcl when executing the commands of the script. For example, in the line #10 we can see the command *mysqlconnect*; if the line #2 were not included in the script, then the Tcl interpreter executing line #10, the command *mysqlconnect*, would give a command not found error.

In the lines #5 and #6 the port and host the Tcl script will try to connect to are set. In this script the port number is 3306 (the default port on which a MySQL server is listening) and the host is the same machine on which the Tcl script is executed.

In the lines #7 and #8 the database user and database user's password are set.

At line #10 the database connection is actually established. The output of the command *mysqlconnect* is stored in the variable that we have named *mysql_handler*. This variable will be the handler of the database connection. This handler is used for querying the database and also for closing the database connection, as shown on line #12.

Handling errors

In the previous subsection, the line #3 of the script was not explained. We will do it now.

MySQLTcl library commands might raise errors. If an error is not caught, the script will abort and this effect might interest us or not. Let's modify the script presented in the previous subsection in this way:

```
10: if [catch {mysqlconnect -host $host -port $port
    -user $user -password $password} mysql_handler] {
11:     puts stderr {error, the database connection could not be established}
12:     exit
13: } else {
14:     mysqlclose mysql_handler
15: }
```

If the command *set mysql_handler [mysqlconnect -host $host...* raises an error, the error will be caught by the command *catch*. The command *catch* returns 1 if the executed command inside the curly braces raises an error or 0 if no error is raised. The variable *mysql_handler* stores the output of the executed command inside the curly braces.

If there is an error, a message is printed on the standard error output (stderr), which in our case is the screen. There are a number of causes that raise an error when attemping to establish a database connection: wrong password, incorrect host or port number... In this case, getting more information than given by a simple message like "there was an error" can be useful.

On line #3, the variable *mysqlstatus* was declared global. A global variable is one that is accessible from any part of a Tcl script; this is related to Tcl variable scope, a topic not covered in this article. The library MySQLTcl creates and mantains a global array named *mysqlstatus*. This array has the following elements:

element	meaning
code	If there were no error then *mysqlstatus(code)* equals to zero; else, *mysqlstatus(code)* is set to the MySQL server error code. If there were an error not raised by the MySQL server, then *mysqlstatus(code)* is set to -1. The value of *mysqlstatus(code)* is updated after the execution of any MySQLTcl library command.
command	The last MySQLTcl library command that raised an error is stored in *mysqlstatus(command)*. The value of *mysqlstatus(command)* is updated after every unsuccessful execution of any MySQLTcl command; thus, *mysqlstatus(command)* is **not** updated after the successful execution of any MySQLTcl command.
message	The value of *mysqltcl(message)* is updated after every unsuccessful execution of any MySQLTcl command with a string containing a message error. Like *mysqlstatus(command)*, mysqlstatus(message) is **not** updated after the successful execution of any MySQLTcl command.

There is another element in the global array *mysqlstatus* not related with error handling:

element	meaning

nullvalue String used to represent the null value when showing SQL query results. By default an empty string is used; *musqlstatus(nullvalue)* can be set to any string value.

Thus, using the global array *mysqlstatus*, the previous piece of code could be rewritten in this way so more information is given in case there is an error while attemping to establish a database connection:

```
10: catch {mysqlconnect -host $host -port $port
    -user $user -password $password} mysql_handler
11: if {$mysqltatus(code) != 0} {
12:     puts stderr $mysqlstatus(message)
13: } else {
14:     mysqlclose mysql_handler
15: }
```

Obviously, the global array *mysqlstatus* can be used for handling more errors than raised when attemping to establish a database connection.

Basic MySQLTcl library commands

In this section the most basic MySQLTcl library commands are presented and I will show how to use them with some examples. For a complete reference, see the MySQLTcl library man page.
The commands covered in this section are shown in the following table. Parameters are underlined. If a parameter is between two ? characters, it means that it is optional; the character | means "or".

command	short description
mysqlconnect ?option value ...?	connects to a database; a connection handler that must be used by other mysqltcl commands is returned
mysqluse handle dbname	associates a MySQL handler with the specified database
mysqsel handle sql_statement ?-list | -flatlist?	sends a select SQL statement to the database
mysqlexec handle sql_statement	sends a non-select SQL statement to the database
mysqlclose handle	closes a database connection

mysqlconnect

This command was already discussed in the section "Database connection". This command accepts an extra parameter not shown before: -db. With the parameter -db, the database that will be used in future SQL statements is fixed. Let’s see an example:

```
% set mysql_handler [mysqlconnect -h 127.0.0.1 -p 3306 \\
  -user john_smith -password jsmith_password -db jsmith_database]
```

The "target" database of MySQLTcl commands using the *mysql_handler* database handler will be the one named *jsmith_database*.

(Please, note that the characters \\ are not part of the command; they mean that the command continues on the following line.)

mysqluse

This command allows changing the database associated to the MySQL handler which this command gets as first parameter.

mysqlsel

This command sends a SQL select statement to the database associated with the MySQL handler. If the parameter sql_statement is not a SQL select statement, then there will be an error.

There is a third optional parameter, which can be list or flat_list. Let's see in an example how this parameter affects the output of this command. Supose that in the database associated with the MySQL handler we have a table named *people* like shown below:

id	first_name	last_ name	phone
26	Karl	Bauer	8245
47	James	Brooks	1093
61	Roberto	Castro Portela	6507

We use the *mysqlsel* command to send a SQL select statement to the database:

```
% mysqlsel $mysql_handler {select first_name, last_name from people order by id asc}
{Karl Bauer} {James Brooks} {Roberto {Castro Portela}}
% mysqlsel $mysql_handler {select first_name, last_name from people order by id asc}
Karl Bauer James Brooks Roberto {Castro Portela}
```

In the first example (*-list parameter*), the command returns a list whose elements are lists. In the second example (*-flatlist parameter*), the command returns a single list in which all the elements have been concatenated.

What happens if the *mysqlsel* command doesn't get a third parameter? In this case, the output of the *mysqlsel* command is the number of rows returned by the query:

```
% mysqlsel $mysql_handler {select first_name, last_name from people order by id asc}
3
```

mysqlexec

The *mysqlexec* command sends a SQL non-select statement to the database associated with the MySQL

handler. If the parameter *sql_statement* is a SQL select statement, there will not be an error but nothing will be done.

Let's take the example shown in the previuos subsection:

```
% mysqlexec $mysql_handler {delete from people where id=26}
1
% mysqlsel $mysql_handler {select first_name, last_name, phone from people order by
{James Brooks 1093} {Roberto {Castro Portela} 6507}
% mysqlexec $mysql_handler \\
  {insert into people (id, first_name, last_name, phone) values (58, "Carla", "di Be
1
% mysqlsel $mysql_handler {select first_name, last_name, phone from people order by
{James Brooks 1093} {Carla {di Bella} 8925} {Roberto {Castro Portela} 6507}
```

Of course, other SQL queries than delete ot insert can be sent to the database with this command. For example, a row can be updated:

```
% mysqlexec $mysql_handler {update people set phone=3749 where first_name="James"}
1
% mysqlsel $mysql_handler {select first_name, last_name, phone from people order by
{James Brooks 3749} {Carla {di Bella} 8925} {Roberto {Castro Portela} 6507}
```

The command *mysqlexec* returns the number of rows affected by the SQL non-select statement sent to the database if executed successfully.

mysqlclose

As seen previously, the command *mysqlclose* closes a database connection. The parameter to this command is the MySQL handler of the database connection we want to close.

Other MySQLTcl library commands

The MySQLTcl library has more commands than the five shown in this section. These commands allow retrieving information about the databases, escaping strings so they are suitable for queries, making nested queries... A good reference is the own MySQLTcl man page which is part of the installation of the MySQLTcl library.

References

[1] a slightly skeptical view on John K. Ousterhout and Tcl:
http://www.softpanorama.org/People/Ousterhout/index.shtml

[2] a tutorial on Tcl regular expressions:

http://www.mapfree.com/sbf/tcl/book/select/Html/7.html

TclTutor is a free and interactive application for learning Tcl:
http://www.msen.com/~clif/TclTutor.html

MySQL documentation, in various formats (HTML, PDF...):
http://www.mysql.com/documentation/index.html

Webpages maintained by the LinuxFocus Editor team © Diego Alberto Arias Prad "some rights reserved" see linuxfocus.org/license/ http://www.LinuxFocus.org	Translation information: en --> -- : Diego Alberto Arias Prad <dariapra(at)yahoo.es>

2005-01-14, generated by lfparser_pdf version 2.51