LiveCode Server for HTTP Server built with Node.js

The official instructions only explain how to install "LiveCode Server" [1] as a CGI processor for the Apache web server - although the package may also be used behind other servers without any problems. If you prefer light-weight solutions, you may build your own server using Node.js [2] and Express.js [3] together with the SimpleCGI "Middleware", which forwards CGI requests to "LiveCode Server".

Read here how to put together your own server in this way.

Overview

A simple Server for "LiveCode Server"

If you plan to work with "LiveCode Server", terms such as "Node.js" and "Express.js" may not sound familiar to you - but setting up a CGI-enabled server is still pretty easy.

As a preliminary information: using "Node.js" (together with additional packages such as "Express.js" and "SimpleCGI") it is possible to create extremely light-weight yet still powerful servers which are programmed in JavaScript.

A simple HTTP server built with these systems, that receives requests for files with the extension ".lc" and forwards them to "LiveCode Server", looks as follows:

```
#!/usr/bin/env node

  var express   = require('express');
  var SimpleCGI = require('simplecgi');

  var oneDay = 24*60*60*1000;

  var WebServer = express();
    WebServer.use(express.compress());
    WebServer.use(express.staticCache());

    WebServer.all(/^.+[.]lc$/, SimpleCGI(
      '/usr/local/bin/livecode-server', __dirname + '/www', /^.+[.]lc$/
    ));

    WebServer.use(express.static(__dirname + '/www', { maxAge:oneDay }));

    WebServer.use(express.errorHandler());
  WebServer.listen(8080);                          // actually starts the server
```

More is not required!

The first line is called a "shebang" line, and allows Mac OS X and Linux users to directly invoke the file containing this script - under Windows, this line should be omitted.

The lines

```
    WebServer.all(/^.+[.]lc$/, SimpleCGI(
      '/usr/local/bin/livecode-server', __dirname + '/www', /^.+[.]lc$/
    ));
```

recognize a CGI-based request by looking for extension ".lc" in the URL and directs them to "LiveCode Server". In this particular example, it is assumed that "LiveCode Server" is installed as a "command line processor" (as described elsewhere). Windows users should modify the installation path accordingly.

Both static files as well as the scripts for "LiveCode Server" are expected in subdirectory "www" of the same directory in which the HTTP server was set up. If you prefer, you may adjust the terms `__dirname + '/www'` to your own taste.

Server Installation

The complete installation of the described HTTP server is done as follows:

1. If not already done, download an installer for your operating system from nodejs.org and install Node.js on your computer;
2. open a terminal window;
3. create a directory for your HTTP server

   ```
   mkdir WebServer
   ```
4. change to that directory

   ```
   cd WebServer
   ```
5. install Express.js

   ```
   npm install express
   ```
6. install SimpleCGI

   ```
   npm install simplecgi
   ```
7. open a text editor and create a file called `WebServer` in the current directory. The content of this file is the script shown above - if necessary with any changes you want
8. users of Mac OS X or Linux should now mark the script as executable

   ```
   chmod +x WebServer
   ```

   (Windows users should skip this step)
9. create a subdirectory for the files the WebServer should deliver:

   ```
   mkdir www
   ```

That's all: users of Mac OS X or Linux may run the script file directly:

```
./WebServer
```

Windows users should enter

```
node WebServer
```

instead.

The newly started server is now waiting for requests on port 8080 - any errors are output to the still open terminal window.

A few Sample Scripts

To test the functionality of the server you just created, the following short scripts may be used. Just follow these steps::

1. create a file with the specified name in the subdirectory www of your server (e.g., `www/00_SmokeTest.lc`)
2. invoke this script from your browser as follows:

   `http://127.0.0.1/00_SmokeTest.lc`
3. the browser should now display the output generated by this script

00_SmokeTest.lc

This very first test will only display a simple text to demonstrate the functioning of HTTP server and "LiveCode Server":

```
<?lc
  put header "Content-Type: text/plain" & NumToChar(13)

  write "LiveCode Server is alive" & LF to stdout  # due to LiveCode Server bug!
?>
```

Anyone familiar with "LiveCode Server", will probably wonder about two specialities:

1. `put header ... & NumToChar(13)`
   "LiveCode Server" does not terminate HTTP headers with the character sequence CR-LF (as actually required by the standard), but only with LF - explicitly appending `NumToChar(13)` (not the LiveCode constant `CR`!) to the actual header avoids this problem - albeit with the side effect of a blank line at the beginning of the actual script output;
2. `write ... & LF to stdout`
   as it has already been described elsewhere, text output using `put` does not work correctly from within "LiveCode Server". You should prefer `put binary ...` or `write ... to stdout` instead

01_EnvironmentVariables.lc

The second script displays the contents of any environment variables relevant for CGI scripts:

```
<?lc
  put header "Content-Type: text/plain" & NumToChar(13)

  local KeyList; put the keys of $_SERVER into KeyList
    sort lines of KeyList

  local maxKeyLength; put 0 into maxKeyLength
  repeat for each line Key in KeyList
    local KeyLength; put the number of chars of Key into KeyLength
    if (KeyLength > maxKeyLength) then; put KeyLength into maxKeyLength; end if
  end repeat

  repeat for each line Key in KeyList
    put the number of chars of Key into KeyLength

    write Key to stdout                          # due to a LiveCode Server bug
      repeat with i = KeyLength to maxKeyLength; put " "; end repeat
    write "= " & quote & $_SERVER[Key] & quote & LF to stdout             # dto.
  end repeat
?>
```

02_SmokeTest.lc

Of course, you can also create HTML pages, as shown by the following script:

```
<?lc
  put header "Content-Type: text/html" & NumToChar(13)
?>
<!DOCTYPE HTML>
<html>
  <head>
    <title>LiveCode Server is alive!</title>
  </head>
  <body>
    <h1>LiveCode Server is alive!</h1>
  </body>
</html>
```

03_ErrorInScript.lc

Even scripts for "LiveCode Server" may contain errors. The next script shows what happens (without further precautions) in such a case:

```
<?lc
  put header "Content-Type: text/plain" & NumToChar(13)

  throw "Exception thrown in Script"
?>
```

As can be seen easily, the script output is not very helpful...

04_ErrorInScript.lc

In general, redirecting error output to stderr helps - although our server only logs them in the terminal window: nothing will be displayed in the browser (which may possibly even be beneficial in the production case):

```
<?lc
  set the ErrorMode to "stderr"                     # reports any errors on stderr

  put header "Content-Type: text/plain" & NumToChar(13)

  write "Please, look into the server's log file to see the error message" && \
    "issued by this script" & LF to stdout          # due to LiveCode Server bug!

  throw "Exception thrown in Script"
?>
```

05_ErrorInScript.lc

The most useful solution is probably to provide a special procedure for processing (otherwise uncaught) errors:

```
<?lc
--------------------------------------------------------------------------------
-- ScriptExecutionError            catches and reports an error in the script --
--------------------------------------------------------------------------------

  on ScriptExecutionError ErrorStack, FileList
    write "Error in Command-Line Script:"         & LF to stdout
    write "ErrorStack = " & ErrorStack            & LF to stdout
    write "FileList   = " & FileList              & LF to stdout
    write "Context    = " & the ExecutionContexts & LF to stdout

    exit to top
  end ScriptExecutionError
  put header "Content-Type: text/plain" & NumToChar(13)

  throw "Exception thrown in Script"
?>
```

How to handle errors in the "best" way, depends on your particular application: in principle, logging to the terminal window (via stderr) should always be useful - on the other hand, a display in text or html form will, at least, inform the user that an error occurred.

Have fun with "LiveCode Server" and this HTTP server!

Bibliography

[1] (RunRev Ltd.)
LiveCode | LiveCode Server Guide
(see http://livecode.com/developers/guides/server/)
The LiveCode Server is an interpreter for LiveCode scripts that is started from the command line (does not offer any graphical user interface) and is intended primarily as a CGI processor (this way, web pages can be processed with LiveCode - thus, you do not necessarily have to learn PHP any longer).

[2] Joyent Inc.
node.js
(see http://nodejs.org)
Node.js is first and foremost a platform for extremely powerful network applications written in JavaScript. However, in combination with other technologies (such as Node-WebKit) Node.js may also be used for more than just HTTP servers.

[3] Tj Holowaychuk
Express - node.js web application framework
(see http://expressjs.com)
Express is a lightweight web application framework for Node.js. Thanks to its modularity, Express allows for rapid development of HTTP servers based on Node.js.