

# Server Deployment Release Notes

## Table of Contents

Overview.....	3
Installation.....	3
Installing with Apache.....	3
Installing on OS X.....	5
Installing on Windows.....	6
Installing on Linux.....	7
Installing via .htaccess.....	7
Global Script.....	7
CGI Mode.....	8
Command-line Mode.....	8
Put Extensions.....	9
Error Handling.....	9
Stack Support.....	9
Externals.....	10
SDK.....	10
Syntax.....	10
\$_SERVER.....	10
\$_GET.....	11
\$_GET_RAW.....	11
\$_GET_BINARY.....	11
\$_POST.....	11
\$_POST_RAW.....	11
\$_POST_BINARY.....	11
\$_FILES.....	12
stdin / stdout / stderr.....	12
\$_COOKIE.....	12
put [secure] [httponly] cookie <name> [for <path>] [on <domain>] with <value> [until <expiry>].....	13
start session.....	13
stop session.....	13
delete session.....	13
\$_SESSION.....	14
the sessionSavePath.....	14
the sessionLifeTime.....	14
the sessionID.....	14
the sessionCookieName.....	15
include.....	15
require.....	15
put.....	15
put [ new ] header <header>.....	16
put [ unicode ] <string>.....	16
put binary <string>.....	16
put [ unicode ] markup <string>.....	16

put [ unicode ] content <string>.....	16
the errorMode.....	17
the outputTextEncoding.....	17
the outputLineEndings.....	17
scriptExecutionError.....	18
Changes Compared to revServer.....	18
Change Logs and History.....	19
Engine Change History.....	19
Document History.....	19

## Overview

The server engine is a separate build of the LiveCode engine with specific syntax and functionality that makes it suitable for use in command-line contexts and, in particular, as a CGI processor. In terms of interaction, it borrows heavily from PHP.

The principal difference between the server engine and desktop/mobile engines is that it is able to process scripts from text files in global scope. Such a script consists of content to output directly interspersed with code segments. Code segments are enclosed in processing instructions '<?... ?>'. This global script is naturally integrated into the existing language structure by being represented as an implicitly created 'Home' stack that sits at the root of the message path - identical to the Home stack in the IDE engine.

## Installation

For each supported platform, LiveCode Server is distributed as a single .zip file containing the server engine, the drivers and externals and these release notes. The engine can be run in two separate modes, command line mode or CGI mode (see the appropriate sections later in this document for more information). To run the engine in command line mode, unzip the appropriate archive for your platform, then execute the livecode-server binary (livecode-server.exe on Windows) from the command line, passing the initial script as the first argument. For example:

```
[user@~/LiveCodeServer/]$ livecode-server my_script.lc
```

To run the in CGI mode, the engine needs to be integrated with the web server software running on your machine. There are various web server packages available, most of which should be compatible with LiveCode Server. In this document, we describe how to setup for the most popular package, Apache.

### Installing with Apache

The simplest way to get LiveCode Server running as a CGI is to add the required directives to your Apache configuration, mapping LiveCode script files to the livecode-server binary. If you do not have access to your machine's Apache configuration, see the section Installing via .htaccess.

In order to set up LiveCode Server as a CGI with Apache, the following three modules must be enabled:

- mod\_actions: [http://httpd.apache.org/docs/2.0/mod/mod\\_actions.html](http://httpd.apache.org/docs/2.0/mod/mod_actions.html)
- mod\_cgi: [http://httpd.apache.org/docs/2.0/mod/mod\\_cgi.html](http://httpd.apache.org/docs/2.0/mod/mod_cgi.html)
- mod\_alias: [http://httpd.apache.org/docs/2.0/mod/mod\\_alias.html](http://httpd.apache.org/docs/2.0/mod/mod_alias.html)

To begin, unzip the appropriate LiveCode Server archive for your platform to a suitable location on your machine. Your system may already have a pre-configured directory where CGI executables are stored, in which case you may choose to install the server engine and its associated files there. To determine if your system has been set up with such a central location, search your httpd.conf file for an entry like the following (for more information on the location of your httpd.conf file, see the installation section for your platform):

```
ScriptAlias /cgi-bin/ /usr/lib/cgi-bin/
<Directory "/usr/lib/cgi-bin">
```

```

    AllowOverride None
    Options +ExecCGI -MultiViews +SymLinksIfOwnerMatch
    Order allow,deny
    Allow from all
</Directory>

```

These directives determine that by default, all requests to `/cgi-bin/` will be mapped to the directory `/usr/lib/cgi-bin/`.

If you wish to limit the use of the LiveCode server engine to your user account only or do not have access to a shared CGI directory, then you can install the server within your home directory (or any other location).

Once unzipped, you need to map requests for LiveCode scripts to the server engine. You can do this by adding the appropriate directives to your `httpd.conf` file. Your `httpd.conf` file should have an entry similar to the following:

```
DocumentRoot /var/www
```

This directive determines where the web server will look for files. For example, in the above situation, if you were to request `http://localhost/index.lc`, the web server will serve the file located at `/var/www/index.lc`.

You need to tell the server to map all `.lc` files within the document root and its sub-directories to the LiveCode Server engine. To do that, you need to add extra entries to the `<Directory>` directive for the document root. Locate the `<Directory>` directive for your document root. It should look something like the following:

```

<Directory "/var/www/">
    Options Indexes FollowSymLinks MultiViews
    AllowOverride None
    Order allow,deny
    Allow from all
</Directory>

```

To this directive, add the following two lines:

```

AddHandler livecode-script .lc
Action livecode-script /cgi-bin/livecode-server

```

The first line tells the web server to match files with the `.lc` extension (the extension can be anything and can be used to map particular extensions to a given version of the LiveCode Server engine) with the type `livecode-script`. The second line maps files of type `livecode-script` to the CGI script `livecode-server` in the directory `cgi-bin`. Once these two lines have been added, your `<Directory>` directive should look something like this:

```

<Directory "/var/www/">
    Options Indexes FollowSymLinks MultiViews
    AllowOverride None
    Order allow,deny
    Allow from all
    AddHandler livecode-script .lc
    Action livecode-script /cgi-bin/livecode-server

```

```
</Directory>
```

If a default cgi-bin directory has been set up and you have installed the LiveCode server engine and associated files directly within that directory, then setup is complete. If not, you must add a ScriptAlias directive that will determine the location of our LiveCode server engine. If, for example, you have installed the LiveCode Server engine and associated files in the directory /home/username/LiveCodeServer/4\_6\_3/, then this directive will be as follows:

```
ScriptAlias /cgi-bin/livecode-server
/home/username/LiveCodeServer/4_6_3/livecode-server
```

If a default CGI directory has been set up but you wish to install your LiveCode server files elsewhere, you will either need to remove that alias or ensure that the name of the new alias you create is different to that of the default directory. The first ScriptAlias directive Apache finds for a given alias will overwrite all others. For example, if the default CGI directory has the /cgi-bin/, the alias of the directory where you will store the LiveCode Server files must be something different (unless of course you remove the default alias). In this case, your additions may look something like the following:

```
AddHandler livecode-script .lc
Action livecode-script /livecode-cgi/livecode-server
```

```
ScriptAlias /livecode-cgi/livecode-server
/home/username/LiveCodeServer/4_6_3/livecode-server
```

Finally, you must make sure that this directory has permissions to execute CGI scripts. You do this by adding the following <Directory> directive.

```
<Directory "/home/username/LiveCodeServer/">
    Options ExecCGI
    Order allow,deny
    Allow from all
</Directory>
```

Setup is now complete. All you need to do now is restart Apache.

## Installing on OS X

OS X comes with Apache installed by default. Most of the relevant files can be found in the directory /etc/apache2/ which can be reached using Finder via the Go to Folder option from the Go menu. To enable Apache, you need to turn on Web Sharing from within the Sharing pane of System Preferences.

By default, each user has been set up with their own sub-site that can be accessed by the following link:

<http://localhost/~username/>

The files for each user's site are served from the directory /Users/username/Sites/ with the Apache config file for that sub-site located at /etc/apache2/users/username.conf. If you want to set up LiveCode Server for a specific user, you need to modify the users Apache .conf file to look the following:

```
<Directory "/Users/username/Sites/">
    Options Indexes MultiViews
    AllowOverride None
```

```

    Order allow,deny
    Allow from all
    AddHandler livecode-script .lc
    Action livecode-script /livecode-cgi/livecode-server
</Directory>
<Directory "/Users/username/LiveCodeServer/">
    Options ExecCGI
    Order allow,deny
    Allow from all
</Directory>
ScriptAlias /livecode-cgi/livecode-server
/Users/username/LiveCodeServer/4_6_3/livecode-server

```

This assumes that the LiveCode Server engine and associated files have been installed in the directory `/Users/username/LiveCodeServer/4_6_3/`.

If you want to install LiveCode Server for all users, you can install the LiveCode Server engine and associated files in the directory `/Library/WebServer/CGI-Executables/`. This is the default location for CGI executables, as specified in the file `/etc/apache2/httpd.conf`. You then need to add the following two entries to the `<Directory "/Library/WebServer/Documents">`:

```

AddHandler livecode-script .lc
Action livecode-script /cgi-bin/livecode-server

```

Once complete, you can restart Apache either by disabling and enabling Web Sharing from System Preferences or by executing the following command from the terminal:

```
[user@~]$ sudo apachectl restart
```

## Installing on Windows

Apache installers for Windows can be found here: <http://httpd.apache.org/download.cgi#apache22>. By default, (assuming `C:\` is your main hard disk) Apache installs at `C:/Program Files/Apache Software Foundation/Apache2.2/`, with the `httpd.conf` file located in the `conf/` sub-directory. Files are served from the `htdocs/` sub-directory.

To get LiveCode Server up and running, add the following entries to the `<Directory "C:/Program Files/Apache Software Foundation/Apache2.2/htdocs">` directive in the `httpd.conf` file.

```

AddHandler livecode-script .lc
Action livecode-script /cgi-bin/livecode-server.exe

```

If you install LiveCode Server in the directory `C:/Program Files/Apache Software Foundation/Apache2.2/cgi-bin/`, all you need to do now is restart Apache, using the tools available from the Start menu or the System Tray.

If you wish to install LiveCode Server in a custom location (for example `C:/LiveCodeServer/4_6_3/`), you will need to add the following lines to your `httpd.conf` file before restarting Apache:

```

<Directory "C:/LiveCodeServer/4_6_3/">
    Options ExecCGI
    Order allow,deny
    Allow from all

```

```
</Directory>
ScriptAlias /cgi-bin/livecode-server.exe
"C:/LiveCodeServer/4_6_3/livecode-server.exe"
```

As noted in the section Installing with Apache, you will need to make sure your new alias does not clash with any existing defaults.

Apache can also be installed as part of a WAMP package (Windows Apache MySQL PHP). See this Wikipedia page for a comparison of WAMP packages available: [http://en.wikipedia.org/wiki/Comparison\\_of\\_WAMPs](http://en.wikipedia.org/wiki/Comparison_of_WAMPs). Most WAMP packages will use a standard Apache configuration, with a similar directory and httpd.conf structure as detailed above.

## Installing on Linux

Most Linux distributions come with Apache installed by default. If not, you can either install it using the appropriate package manager for your system, download pre-built binaries for your system or build it from source. The most common location for Apache is /etc/apache2/, with the httpd.conf file residing within that directory. Locate your main httpd.conf file and add the directives detailed in the section Installing With Apache, before restarting apache.

## Installing via .htaccess

If you are attempting to install LiveCode Server on a shared host, it will often be the case that you will not have access to the Apache configuration files. If your host allows configurable .htaccess files, LiveCode Server can be configured as follows:

Unzip the LiveCode Server archive into the cgi-bin folder. Then, in the 'public\_html' directory, create a .htaccess file along the following lines:

```
Options ExecCGI
AddHandler livecode-script .lc
Action livecode-script /cgi-bin/livecode-server
```

This file will tell Apache to map files with a .lc extension to livecode-script files, which should then get executed by the LiveCode Server CGI.

## Global Script

At start up, a 'Home' stack is created which serves as the container for the global script. This stack sits at the root of the message path and works in much the same way as the IDE home stack - it sits after all mainstacks in the message path, but before library stacks and backscripts.

Global script execution begins with either the file that was specified on the command-line (non-CGI mode), or as part of the PATH\_TRANSLATED environment variable (CGI mode). Further scripts can be executed via the include or require commands, which always affect the global script regardless of where they are executed from.

Scripts are parsed in full before being executed, with any handler and variable definitions being added to the home stack environment before any commands placed at global scope are executed. The latter is ordered by encounter in the file. As include and require are commands, the parse-before-execute effect only extends as far as the end of the current file.

A script consists of an alternating sequence of content blocks and code blocks. A code block is contained between <?rev ... ?>, <?lc ... ?> or <?livecode ...?> tags with the ?> being an implicit

command terminator / delimiter. Any newline that is present after the `?>` tag is ignored and is not considered part of the following content block. Each content block is implicitly converted to a 'put binary' command either in global scope, or within an enclosing handler, should there be one. The equating content blocks with the binary form of put essentially means that the encoding of any content must match the output text encoding.

An important limitation in the current implementation is that the encoding of text inside code blocks is always considered to be that of the native platform. This means that if a file is encoded as UTF-8 then text within code blocks must be limited to ASCII text.

## CGI Mode

If, on startup, the engine finds the `GATEWAY_INTERFACE` environment variable then it initializes itself in CGI mode. In this mode, various `$` variables are made available to script for easy interaction with the CGI environment.

The initial script to execute is taken from the `PATH_TRANSLATED` environment variable, and headers will be output before the first explicitly put or write to stdout.

In this mode, scripts can access the CGI variables passed to it via the `$_SERVER` variable and any form / query parameters via either `$_GET` or `$_POST`.

The GET data can either be fetched using `$_GET`, as binary data (not converted to the native character set) via `$_GET_BINARY`, as raw data via `$_GET_RAW`, or directly from the `QUERY_STRING`.

The POST data can either be fetched using `$_POST` (if it is encoded in a form the engine understands), as binary data (not converted to the native character set) via `$_POST_BINARY`, as raw data via `$_POST_RAW`, or directly by reading from stdin. When `$_POST` is fetched for the first time, the `$_POST_RAW` and `$_POST_BINARY` data will also be fetched. The same is true of `$_POST_RAW` and `$_POST_BINARY`. If script has already read from stdin, `$_POST`, `$_POST_BINARY` and `$_POST_RAW` will be empty.

Any CGI headers to be output should be specified using 'put [new] header ...'. This either replaces an existing header or appends a new header. The 'new' adjective is needed if multiple headers with the same tag are needed. At a minimum, the engine will always generate an appropriate 'Content-Type' header if it has not been overridden by a 'put header' instruction.

At the moment the engine assumes that the 'Content-Type' is html if no header has been generated. In this case, an appropriate 'charset' is appended depending on the setting of 'the `outputTextEncoding`'.

## Command-line Mode

If no `GATEWAY_INTERFACE` environment variable is present, the engine loads the first command-line argument as the initial script file.

In this mode, no special `$` variables are created except for the normal import of environment variables that occurs in other engine modes. Also, there is no special handling of `stdio` or `stdout`.



## Put Extensions

The server engine has a number of extensions to 'put' in order to ease development of CGI and command-line script applications.

CGI headers can be output using 'put [new] header ...' - the command taking a standard header spec '<key>:<value>' and either adding to the list of headers, or replacing an existing header.

Text to be interpreted as HTML content can be written out using 'put [ unicode ] content'. Any text provided has full entity substitution performed (including quotes and angled brackets) as appropriate to the outputTextEncoding (i.e. using named / numeric entities instead of encoded values).

Text to be interpreted as HTML markup can be written out using 'put [unicode ] markup'. In this case, the engine assumes any quotes and angled brackets have been substituted and only ensures that encoding of characters is appropriate to the outputTextEncoding.

If data is to be output without any automatic processing, then the 'put binary' form can be used. This writes out a given string verbatim with no conversion or transcoding performed.

Otherwise, 'put' and 'put unicode' convert the given string appropriately depending on the current 'outputTextEncoding' with unknown characters being replaced with '?'. Additionally, line encoding conversion is performed, changing the engine's internal universal LF line engine to LF, CR or CRLF depending on the setting of the outputLineEndings.

## Error Handling

If an error occurs which propagates back to the global script environment, then a 'scriptExecutionError' message will be sent to the global script (Home stack). If this is unhandled a default error output handler will be invoked which uses 'the errorMode' to produce appropriate output. After this, script execution is terminated.

## Stack Support

The LiveCode Server engine supports loading stacks in a similar manner to that of the desktop engines. Stacks provide the programmer with an additional means to store data and manage code. However, visual and graphical commands are not supported (e.g. export snapshot). See the section Global Script for more details of how the message path works in the server context.

To load a library stack use the start using command:

```
start using stack "<path to stack>"
```

As with the start using command in other environments, the libraryStack message will be sent to the newly loaded library stack and its the stack script of the will sit behind the Home stack.

The go stack command is also valid:

```
go stack "<path to stack>"
```

Here, the newly loaded stack will sit in front of the home stack and will be sent the standard initialisation messages (preopenstack, openstack etc).

You can also send messages directly to stacks using the following form:

```
send "<message name>" to stack "<path to stack>"
```

Stacks and objects can also be created using the standard create commands.

## Externals

Included in the LiveCode Server distribution are the revZip, revXML and revDB externals, with database drivers for MySQL, ODBC, Oracle PostgreSQL and SQLite provided. The browser, font, speech and video grabber externals do not apply to the Sever platform.

## SDK

Externals created with the desktop externals SDK can be used with LiveCode Server. The only difference is with OS X externals, which must be compiled as dylibs rather than bundles. To do this, add a new dylib target to your Xcode project.

## Syntax

### **\$\_SERVER**

Available when running CGI mode.

It is an array variable, containing the CGI interface related variables, along with any HTTP\_\* variables that are available.

The list of CGI variables is:

- GATEWAY\_INTERFACE
- SERVER\_ADDR
- SERVER\_NAME
- SERVER\_SOFTWARE
- SERVER\_PROTOCOL
- REQUEST\_METHOD
- REQUEST\_TIME
- QUERY\_STRING
- DOCUMENT\_ROOT
- HTTPS
- REMOTE\_USER
- REMOTE\_ADDR
- REMOTE\_HOST
- REMOTE\_PORT
- REDIRECT\_REMOTE\_USER
- SERVER\_ADMIN
- SERVER\_PORT
- SERVER\_SIGNATURE
- PATH\_TRANSLATED
- REQUEST\_URI
- PATH\_INFO
- SCRIPT\_NAME
- SCRIPT\_FILENAME

- CONTENT\_TYPE
- CONTENT\_LENGTH

`$_SERVER` can be modified but doing so has no effect and should be avoided (indeed, it might become read-only in future).

## **`$_GET`**

Available when running in CGI mode.

It is an array variable, translated from the `QUERY_STRING`. It assumes the query string is encoded as url-form-encoded data.

The data will be converted to the native character set from the character set defined in the `outputTextEncoding`.

## **`$_GET_RAW`**

Available when running in CGI mode.

A binary string variable, identical in content to that of the `QUERY_STRING`.

## **`$_GET_BINARY`**

Available when running in CGI mode.

It is an array variable, translated from the `QUERY_STRING`. It assumes the query string is encoded as url-form-encoded data.

No character set conversion is carried out on the data.

## **`$_POST`**

Available when running in CGI mode.

It is an array variable, formed from reading stdin and translating the data. If the data is not url-encoded or multipart, then it is empty.

The data will be converted to the native character set from the character set defined in the `outputTextEncoding`.

For multi-part forms, when `$_POST` is fetched, `$_POST_BINARY` and `$_FILES` (if necessary) will also be fetched.

## **`$_POST_RAW`**

Available when running in CGI mode.

A binary string variable, formed by reading the content of stdin.

## **`$_POST_BINARY`**

Available when running in CGI mode.

It is an array variable, formed from reading stdin and translating the data. If the data is not url-encoded or multipart, then it is empty.

No character set conversion is carried out on the data.

For multi-part forms, when `$_POST_BINARY` is fetched, `$_POST` and `$_FILES` (if necessary) will also be fetched.

## **\$\_FILES**

Available when running in CGI mode.

`$_FILES` is an array variable formed by reading the content of stdin. If the POST form data is multi-part and includes file uploads, each file uploaded will have an entry containing the following information:

- name: The original name of the uploaded file
- type: The mime type of the uploaded file
- size: The size in bytes of the uploaded file
- filename: The temporary filename of the file in which the uploaded file was stored on the server.
- error: Any error that occurred during the upload. Will be one of
  - upload stopped: Upload was only partially completed
  - upload failed: No file was uploaded
  - no upload folder: Missing temporary folder
  - i/o error: Failed to write file to disk

Uploaded files will be stored in a temporary folder. Once the executing script has completed, all temporary uploaded files created by way of the script will be removed.

The array is constructed on demand, and will be empty if stdin has been touched before it is used.

When `$_FILES` is fetched, `$_POST` and `$_POST_BINARY` will also be fetched.

## **stdin / stdout / stderr**

In CGI mode, stdin is assumed to contain post data.

In CGI mode, writing to stdout for the first time will trigger any headers to be generated.

If 'the errorMode' is stderr and default error handler is triggered then stderr is used as the target for default error messages.

## **\$\_COOKIE**

Available when running in CGI mode.

`$_COOKIE` is a read only array variable storing the contents of any cookies passed to the server engine.

The data will be converted to the native character set from the character set defined in the `outputTextEncoding`.

The array is constructed on demand.

## **put [secure] [httponly] cookie <name> [for <path>] [on <domain>] with <value> [until <expiry>]**

Use the put cookie command to create a new cookie or delete an existing cookie.

After headers have been output, the put cookie variant has no further effect.

- *name* - The name by which the cookie is identified.
- *value* - The contents of the cookie. This will be urlencoded.
- *expiry* - The time at which the cookie expires, expressed as seconds since the UNIX epoch. If unspecified or empty, cookie will expire when the client browser closes. To delete a cookie, set the expiry time to a value in the past.
- *path* - The path on the server at which the cookie will be available.
- *domain* - The domain at which the cookie is available.
- *secure* - If true, the browser will only send the cookie if communicating over a secure connection (https).
- *httponly* - If true, the cookie will only be available from the browser when communicating with the server, and hidden from client-side scripting

## **start session**

Use the start session command to begin a server session. The contents of the \$\_SESSION array is restored if available.

Start session must be called before the headers are sent.

The location where the \$\_SESSION array is stored on the server is configured using the sessionSavePath property.

If no session identifier is specified by the sessionId property or contained within the session cookie, then a new session will be created with a unique identifier, and the contents of the session cookie set to that identifier.

A session is stored until it expires or is deleted using the delete session command. The maximum age of a session is configured using the sessionLifetime property.

Start session is only available when running in CGI mode.

## **stop session**

Use the stop session command to end the session explicitly and save the contents of the \$\_SESSION variable.

Stop session is only available when running in CGI mode.

## **delete session**

Use the delete session command to end the current session, expiring the session cookie and discarding any values contained in the \$\_SESSION variable.

Delete session is only available when running in CGI mode.

## **\$\_SESSION**

\$\_SESSION is an array variable that contains the data for the current active session.

Use the \$\_SESSION keyword to access the array containing the data for the current active session.

\$\_SESSION is only available when running in CGI mode.

## **the sessionSavePath**

Specifies where on the filesystem session data is stored.

Use the sessionSavePath property to set the path to the location on the server filesystem where session data is stored. If the sessionSavePath property is not set then the standard temporary folder is used.

To set the sessionSavePath back to the default set the sessionSavePath to empty.

The sessionSavePath is only effective before a call to stop session. Any changes made after a call to stop session will be ignored.

The sessionSavePath is only available when running in CGI mode.

## **the sessionLifetime**

Specifies the maximum duration in seconds for which session data is retained between uses.

Use the sessionLifetime property to set the duration, in seconds, for which session data is retained. The sessionLifetime specifies how long a session will remain active, accessing the session data resets the countdown to expiry to the sessionLifetime.

For example if the sessionLifetime is 360 seconds (10 minutes) and 10 minutes pass without the session being used the session will expire. A session can last indefinitely as long as the session data is accessed at regular intervals no further apart than the sessionLifetime.

If the sessionLifetime is not specified the default of 24 minutes is used.

The sessionLifeTime is only effective before a call to start session. Any changes made after a call to start session will be ignored.

The sessionLifeTime is only available when running in CGI mode.

## **the sessionID**

Specifies the identifier of the session to be started.

If the sessionID property is not set before starting a session, and the session cookie does not contain a session id, then a session id is automatically created.

If the current session has an automatically assigned id you can retrieve it by getting the sessionID property.

The sessionID is only effective before a call to start session. Any changes made after a call to start session will be ignored.

The sessionID is only available when running in CGI mode.

## **the sessionCookieName**

Specifies the name of the cookie used to store the session id.

Use the sessionCookieName property to set the name of the cookie used to store the session id.

When a session is created on the server, a cookie is created in the browser to identify the ongoing session. Setting the sessionCookieName is useful if you want to have multiple session running a the same time for a given website.

If the sessionCookieName is not specified then the default, LCSESSION, is used.

The sessionCookieName is only effective before a call to start session. Any changes made after a call to start session will be ignored.

The sessionCookieName is only available when running in CGI mode.

## **include**

Executes the given script in the context of the global environment.

The script is first loaded into memory and parsed, any variable and handler definitions being added to the global (script) environment. Then, each command that is present is executed in order as it was encountered in the file.

The include command only works when running in the server environment, invocation of the command in other environments will throw an error.

The behavior of the include command is identical regardless of where it is run from - e.g. if it is run from a handler in a stack, it will still only affect the global script environment (home stack).

## **require**

Executes the given script in the context of the global environment, but only if it has not been previously included/required.

This is the same as the 'include' command in operation, except it makes it easy to implement 'include-once' files and is designed primarily for library scripts.

Require and include are distinct in the sense that if you require a file and then include, the second include will execute the file.

## **put**

The put command has been enhanced with a number of undirected forms making it easier to perform certain output operations. All undirected forms of put cause output to 'stdout', and thus are affected by the 'headers-first' semantic in CGI mode.

**put [ new ] header <header>**

Replaces or adds a CGI output header to the current list that will be generated the first time any data is output to stdout.

The <header> string should be of the form 'header: value'.

If new is not specified, any existing header of the same name has its value replaced by the action. If new is specified, or an existing header is not found, a new entry is made at the end of the header list.

After headers have been output, the put header variant has no further effect.

**put [ unicode ] <string>**

Write the (unicode) string to stdout.

If unicode is not specified, <string> is considered to be in the native text encoding for the platform and will be automatically converted to match the current 'outputTextEncoding' setting.

If unicode is specified, <string> is considered to be a binary string containing UTF-16 encoded text; similarly, this will be automatically converted to match the current 'outputTextEncoding' setting.

When converting, any characters that are encountered which are not representable in the output text encoding will be output as '?'.

In both cases, the internal line ending character 'LF' will be transformed on output to match the setting of 'the outputLineEndings' property.

**put binary <string>**

Write the binary string to stdout. No processing is done on the string and it is written directly to stdout with no intervening processing.

**put [ unicode ] markup <string>**

Write the given (unicode) string to stdout, processing it for suitable output in an SGML markup context.

If 'unicode' is not specified, then <string> is considered to be in the native text encoding for the platform. If 'unicode' is specified, then <string> is considered to be in UTF-16.

Upon output the text is converted to match the setting of the current 'outputTextEncoding' property, with any unrepresentable characters being output using a decimal character entity reference &#dddddd;.

Additionally, the internal line ending character 'LF' will be transformed on output to match the setting of 'the outputLineEndings' property.

**put [ unicode ] content <string>**

Write the given (unicode) string to stdout, processing it for suitable output in an SGML content context.

This functions in an identical way to 'put markup' except that '<', '>', '&' and '"' are encoded as named entities - <, >, & and ".



## the `errorMode`

Determines the action the engine takes when an error occurs and a custom `scriptExecutionError` handler has not been provided.

This can be one of:

- `debugger`
- `inline`
- `stderr`
- `quiet`

The '`debugger`' setting is informational only and indicates that the script is being run in 'remote debug' mode (only relevant to on-rev engine).

The '`inline`' setting means that the error should be output into the stdout stream. In this case, the engine assumes that the output is HTML and puts the error messages in a '`pre`' block.

The '`stderr`' setting means that the error should be written out to stderr.

The '`quiet`' setting means that nothing is output anywhere when an error occurs.

## the `outputTextEncoding`

Determines what text conversion to perform when writing text strings to stdout.

It can be one of the following:

- `windows-1252`: use the Windows 'Latin-1' encoding (codepage 1252) [ this is the native text encoding for the Windows engine ]
- `macintosh`: use the MacRoman encoding [ this is the native text encoding of the Mac engine ]
- `iso-8859-1`: use the ISO-8859-1 encoding [ this is the native text encoding of the Linux engine ]
- `utf-8`: use the UTF-8 encoding

The naming of the encodings here corresponds directly to the IANA assigned charset names which is why they are perhaps slightly different from text encoding naming elsewhere in the engine.

The following synonyms exist:

- `windows == windows-1252`
- `mac == macintosh`
- `linux == iso-8859-1`
- `macroman == macintosh`
- `utf8 == utf-8`

Additionally, '`native`' can be specified to indicate that the `outputTextEncoding` should be the native one for the platform the engine is running on. This is the default.

## the `outputLineEndings`

Determines what line ending conversion to perform on text output.

This can be one of the following values:

- `lf`

- cr
- crlf

The quoted literals *\*must\** be used when setting this property - the property expects a name not a sequence of bytes to use as the line-ending. The reason behind this is two-fold - (1) it is more 'correct' from the point of view it is identifying the style of line-ending rather than the sequence of bytes to use (2) 'cr' and 'lf' are defined as the same numToChar(10) constant on all platforms.

## scriptExecutionError

The 'scriptExecutionError' message is sent to the global script (home stack) when an uncaught error is encountered. It has the following signature:

```
scriptExecutionError pErrorStack, pFilesList
```

Here 'pErrorStack' is the standard engine error stack listing, detailing the errors that occurred at each stage of the stack being unwound and 'pFilesList' is the list of all files that have been 'included' / 'required' which the error stack references if an error occurred in the context of a file script rather than an object script.

## Changes Compared to revServer

There are a number of critical changes/improvements that have been made since revServer:

- Script files are now parsed before being executed - this allows the placement of handlers anywhere within the script regardless of where they are called from. (Variables still need to be declared before they are referenced - just like in object scripts - this is because variable scope is not dynamic and is determined at parse-time in order of encounter).
- The 'put' command now assumes it is being given text strings and will convert them according the outputTextEncoding. If binary data needs to be output then the 'put binary' form must be used. (No processing is done if outputTextEncoding matches the native text encoding which is the default case so existing scripts should continue to work).
- If scripts are using 'put' to write UTF-8 encoded text, then this should be done using 'put unicode uniEncode(<string>, "UTF8")'.
- The 'put content/markup' commands now do entity substitution for unrepresentable characters. This may break existing scripts that are using them to output utf-8. Such uses should be changed to 'put unicode content/markup uniEncode(<string>, "UTF8")'
- The 'global script environment' is now a 'home' stack that sits at the root of the message path. This should not cause any problems with existing scripts as previously there was no interaction with the message path at all, so they would not be using it.
- The server engine build on Mac is now 'proper' Mac rather than Darwin, this affects various system functions (such as date/time, filename handling, text conversion etc. - these implementations are taken from the Desktop Mac engine) as well as meaning the native text encoding is MacRoman and default line endings is CR. The latter two can be changed to match the revServer engine via explicit setting of 'the outputTextEncoding' and 'the outputLineEndings' at the beginning of the file.
- The engine version has jumped from 3.5 up to 4.6.3 - meaning any bug-fixes and features pertinent to server that have been added since 3.5 are now available. (Print to pdf is not yet supported)

## Change Logs and History

### Engine Change History

4.6.3-dp-2 (2011-07-11)	MM See section Changes Compared to revServer.
4.6.3-dp-3 (2011-07-13)	MM Fixed bug with seek relative behaving differently to the desktop platforms (9350). Updated the platform string to be consistent with desktop engines. Made \$_POST and \$_POST_BINARY no longer mutually exclusive (bug 9616). Converted data in \$_GET and \$_POST to native character set from the outputTextEncoding. Added new deferred variables \$_GET_RAW, \$_GET_BINARY and \$_POST_BINARY.
4.6.3-rc-1 (2011-07-15)	MM Fixed bug with loading stacks (9619).
4.6.3-gm-1 (2011-07-19)	MM No changes.
4.6.3-gm-2 (2011-07-26)	MM No changes.
4.6.4-dp-1 (2011-08-10)	MM Added support for multi-part POST forms and file uploads.
4.6.4-dp-2 (2011-08-16)	MM Fixed bug with put new header.
4.6.4-dp-3 (2011-08-22)	MM Added cookie support.
4.6.4-rc-1 (2011-08-26)	MM Removed setCookie function and replaced with put cookie variant. Updated stdin so that it can be read in conjunction with \$_POST. Fixed an assortment of bugs relating to unquoted literals.
4.6.4-rc-2 (2011-09-02)	MM No changes.
4.6.4-gm-1 (2011-09-06)	MM Added /tmp fallback if TMPDIR environment variable is not set when requiring temp folder.
4.6.4-gm-1 (2011-09-09)	MM Added REDIRECT_REMOTE_USER to \$_SERVER array.
4.6.4-gm-3 (2011-09-15)	MM No changes.
5.0.0-dp-2 (2011-09-22)	MM Added support for sessions.
5.0.0-gm-1 (2011-10-10)	MM No changes.
5.0.2-gm-1 (2011-12-12)	MM No changes.
6.0.0-dp-5 (2013-03-01)	MM Fixed bug – FTP not enabled on Linux (10573). Fixed bug - include and require now search for files relative to the folder of the file the include/require command is in (10657). Fixed bug – path environment variables incorrect in OS X Lion and later (9869).
6.0.0-rc-1(2013-03-08)	MM Fixed bug – Windows PostgreSQL driver is corrupt (9958). Fixed bug – Linux engine incompatible with file systems using 64-bit inodes (10673).

### Document History

Revision 1 (2011-07-11)	MM Initial version.
Revision 2 (2011-07-13)	MM Updated Installation section. Added Stack Support section. Updated syntax details for \$_GET and \$_POST to note character set.

		Updated syntax details for \$_POST and \$_POST_RAW to note they are no longer mutually exclusive. Updated CGI Mode section to reflect the changes in CGI variables. Added syntax details for \$_GET_RAW, \$_GET_BINARY and \$_POST_BINARY.
<i>Revision 3 (2011-07-15)</i>	MM	Removed entry in section Changes Compared to revServer stating \$_POST and \$_POST_RAW are mutually exclusive. Added Externals section.
<i>Revision 4 (2011-07-19)</i>	MM	No changes.
<i>Revision 5 (2011-07-26)</i>	MM	No changes.
<i>Revision 6 (2011-08-10)</i>	MM	Added section on \$_FILES. Updated \$_POST and \$_POST_BINARY sections.
<i>Revision 7 (2011-08-16)</i>	MM	No changes.
<i>Revision 8 (2011-08-22)</i>	MM	Added the Syntax sub-sections \$_COOKIES and setCookie
<i>Revision 9 (2011-08-26)</i>	MM	Updated the setCookie section to put cookie. Updated sections on \$_POST, \$_POST_BINARY, \$_POST_RAW and stdin.
<i>Revision 10 (2011-09-02)</i>	MM	No changes.
<i>Revision 11 (2011-09-06)</i>	MM	No changes.
<i>Revision 12 (2011-09-09)</i>	MM	Added REDIRECT_REMOTE_USER to \$_SERVER section.
<i>Revision 13 (2011-09-15)</i>	MM	No changes.
<i>Revision 14 (2011-09-22)</i>	MM	Added sections “start session”, “stop session”, “delete session”, “\$_SESSION”, “the sessionID”, “the sessionSavePath”, “the sessionLifeTime” and the “sessionCookieName”
<i>Revision 15 (2011-10-10)</i>	MM	No changes.
<i>Revision 16 (2011-12-12)</i>	MM	No changes.
<i>Revision 17 (2012-03-01)</i>	MM	No changes.
<i>Revision 18 (2012-03-08)</i>	MM	No changes.