LuaFlare Documentation

2.7.21

January 19, 2015

Contents

1 Command Line Arguments, Options, & Environment Variables

1.1 Arguments

1.1.1 `listen`

Start listening for connections.

1.1.2 `mount path name`

Mount `path` to `name` inside of `/etc/luaflare/sites/`. This will change the path's files group to the group by the same name as the user running LuaFlare (by default, `www-data`).

1.1.3 `unmount name`

Remove a mounted path from `/etc/luaflare/sites/`.

1.2 Options

1.2.1 `--port=number`

Port to bind to (default `8080`).

1.2.2 `--threads=number`

Number of threads to create (default `2`).

1.2.3 `--threads-model=string`

The threading method to use (default `coroutine`).
Valid values for string:

- coroutine
- pyrate

1.2.4 `--host=string`

The host to bind to (default `*`).

1.2.5 `-l`, `--local`

Bind to only this machine; equivalent to `--host=localhost`.

1.2.6 `-t`, `--unit-test`

Perform unit tests.

1.2.7 `-h`, `--help`

Show the help screen.

1.2.8 `-v`, `--version`

Show the version.

1.2.9 `--no-reload`

Do not automatically reload Lua scripts when they've changed.

1.2.10 `--max-etag-size=size`

Specifies the maximum size to generate ETags for.

Supported notation (`K` can also be either: `M`, `G`, `T`, `P`):

- 1B
  - 1 byte
- 1K
  - 1024 bytes
- 1KB
  - 1000 bytes
- 1KiB
  - 1024 bytes

1.2.11 `--reverse-proxy`

Tell LuaFlare that we won't be running stand alone. The client's peer will be set to `X-Real-IP`.

Only inbound connections from trusted sources are allowed.

1.2.12 `--trusted-reverse-proxies=string`

Comma delimited list of trusted reverse proxy addresses.

Each element in the list can be one of:

- IP address
  - `127.0.0.1`
- IP address with mask
  - `192.168.0.0/16`
- Domain
  - `myserver.domain.net`

1.2.13 `--x-accel-redirect=path`

Let LuaFlare use accelerated sending of files with the `X-Accel-Redirect` header. `path` is what to prepend to the URL so that the reverse proxy redirects to the internal section (by default, this is `/./`).

1.2.14 `--x-sendfile`

Let LuaFlare use accelerated sending of files with the `X-Sendfile` header.

1.2.15 `--chunk-size=number`

The number of bytes to send per coroutine yield (default `131072`).

1.2.16 `--scheduler-tick-rate=number`

The tick rate to resort to if the schedule did not specify one (default is `60`).

1.2.17 `--max-post-length=number`

Max number of bytes that can be received in a POST request.

1.2.18 `--systemd`

Enable systemd facilities, such as the heartbeat and notifying systemd on startup completion.

1.2.19 `--out-pid=path`

Upon startup completion, write our PID to `path`.

1.2.20 `--keepalive-time`

Maximum number of seconds a connection may be kept alive (default is `2`).

1.2.21 `--session-tmp-dir=path`

Where to store session (textfiles) files (default: /tmp/luaflare-sessions-XXXXXX).

1.2.22 `--disable-expects`

Disable type checking for performance.

1.2.23 `--socket-backend=string`

The backend to use for sockets (default is “luasocket”).

1.3 Environment Variables

Environment variables may be set either before you call `luaflare` (such as `NAME="value" luaflare ...`), or by placing them inside of `/etc/default/luaflare` – which is loaded automatically, both by `/usr/bin/luaflare` or their daemon scripts.

The following environment variables are recognized by LuaFlare.

1.3.1 `BOOTSTRAP_LOG`

Should we write the bootstrap log to stdout? Defaults to 0.

1.3.2 `LUAFLARE_HOOK_PERFCOUNT_DISABLE`

Set this to 1 to disable the hook performance counter.

1.3.3 `LUAFLARE_CFG_DIR`

Points to where the LuaFlare configuration directory is. Will usually be “/etc/luaflare” or “.”.

1.3.4 `LUAFLARE_LIB_DIR`

Points to where the LuaFlare library directory is. Will usually be “/usr/lib/luaflare” or “.”.

2 Install LuaFlare on Debian based distros

This simple guide will show you how to install LuaFlare on a fresh install of Debian 7, compatible with Ubuntu 12.04 and up.

The sysvinit service uses a newer syntax, so if your system uses sysvinit, the dependency `sysvinit-utils (>= 2.88dsf-50)` must be satisfiable; in Debian, this is satisfied at 7 (Jessie), and with Ubuntu, 15.04 (Vivid). By default, Debian (>= 7) uses systemd by default, and Ubuntu (>= 12.04) uses upstart.

In Ubuntu 14.04 and older, luaflare-service is not installable as the dependency `init-system-helpers (>= 1.18~)` is not satisfiable (this is automatically added via `dh_installinit`). You may have to install the service files yourself by checking out the source (`apt source luaflare`), running the configure script, and then copying `thirdparty/luaflare.upstart.post` to `/etc/init/luaflare.conf`.

2.1 Either install via APT (my repo @ kateadams.eu/debian/)

May not be bleeding edge, but is updated via APT.

2.1.1 sources.list

Open `/etc/apt/sources.list.d/kateadams.list` as a root, and set it's contents to:

```
deb http://kateadams.eu/ debian/
deb-src http://kateadams.eu/ debian/
```

2.1.2 apt keys

```
sudo apt-key adv --keyserver keys.gnupg.net --recv 0B7BD0AD
```

2.1.3 install

```
sudo apt update
sudo apt install luaflare
```

2.2 Or install via git (Makefile)

Bleeding edge, must be updated manually.

2.2.1 Install git, nginx, lua, and LuaFlare's lua dependencies

```
sudo apt install git
sudo apt install nginx-full
sudo apt install lua5.2 lua-bitop lua-socket lua-posix lua-filesystem lua-md5 unaccent
```

2.2.2 Download and install LuaFlare

```
git clone https://github.com/KateAdams/LuaFlare
cd LuaFlare/thirdparty/
#some arguments you may want for configuring: --prefix=/usr/local, --no-nginx (default
    if nginx is not installed), --lua=lua|luajit|lua5.1|lua5.2
./configure
sudo make install
```

2.2.3 Enable the Nginx site

2.2.4 Remove nginx's default site

LuaFlare uses Nginx as a reverse proxy on port 80 to 8080. The default site listens on port 80, and therefore must be removed.

If you wish to keep your current Nginx configs, you can merge `/etc/nginx/sites-available/luaflare` with your own site config after install.

```
sudo rm /etc/nginx/sites-enabled/default
```

2.2.4.1 Enabling LuaFlare's site

```
sudo ln -s /etc/nginx/sites-available/luaflare /etc/nginx/sites-enabled/luaflare
```

2.3 Others/Help

2.3.1 Old method to import keys from gpg

```
gpg --recv-keys 0B7BD0AD
gpg -a --export 0B7BD0AD | sudo apt-key add -
```

2.3.2 Alternate method to get the keys for apt

```
curl kateadams.eu/debian/key | sudo apt-key add -
```

3 Internal Workings

3.1 Entry point

The entry point of main_loop is responsible for loading all the autorun scripts via safely the hook ReloadScripts, once all the autorun files are loaded, the hook Loaded is unsafely called. The Loaded hook is responsible for parsing things such as (in order):

- Parse reverse proxies, mime types, etc…
- Notify the daemon manager by outputting the PID (--out-pid=file), or reporting to systemd (--systemd).

Once loaded, main_loop will enter an infinite loop. The infinite loop works by first attempting to accept a TCP client. If there is clients still connected/in the queue (thread pool), then the accept function will not attempt to wait; however if there are no active connections, then accept will attempt to wait until the next scheduled task is ready to run. Before enqueueing the client, if --no-reload is not set, then any autorun scripts (/lua/ar_*.lua) that have changed (or are new) will be re-executed.

Now the client will be enqueued, the thread-pool ran which processes the connections, and then finally the scheduler will resume.

3.2 Processing the connection

The thread pool responsible for processing the connections will call handle_client(client), where it will attempt to construct a Request object, and keep trying until it either the connection is closed, the Request constructor fails (and returns nil, errstr), the connection has been upgraded, or the keep-alive timeout is reached.

Once the request and response objects have been constructed, the hook Request is safely called. By default, the Request hook is processed by hosts.process_request.

The first thing hosts.process_request() will attempt is to upgrade the connection (check Upgrading for further detail). If the request does not want to be upgraded, then we attempt to locate a host for the request via pattern matching against all hosts, falling back to hosts.any if none is found; if we find more than one host that can take said request, then a 409 Conflict response is sent.

Now that we have a valid host object, we attempt to find the page assigned to it. If a page was not found (404 Not Found) and options.no_fallback is falsy, then an attempt to match against host.any is made.

If a page has still not been found, then halt() is called with the error code and error reason (i.e., a conflict between pages, or a 404); otherwise the page callback will be invoked with the arguments request, response, ..., where ... is either, the captures from the page pattern, or the whole URL (no captures).

3.3 Upgrading

To test whether or not a request wants it's connection to be upgraded, the header Connection is checked to see if upgrade is present, and then the Upgrade header is checked to exist. If both of these are true, then an attempt to upgrade the connection is made by checking the that hosts.upgrades[x] exists, where x is the value of the Upgrade header. In the case of the upgrade function not being found, then LuaFlare will respond to the request with a 404 Not Found with the message "Upgrade not found" and return.

Now that we have our function that is responsible for upgrading the request (upgrader), it will be invoked. The upgrader is responsible for calling request:set_upgraded(); this ensures that both, the connection is not closed, and no more requests are attempted to be read from this connection.

3.4 Pseudocode

```
main_loop():
    safehook ReloadScripts
    hook Loaded
    while true:
        if threadpool_isdone:
            wait til the next scheduled task is to be ran
        else:
            no waiting, pop one if there, but do not wait

        safehook ReloadScripts

        handle_client(client): --returning true = keep connection open
            Response(client)
            safehook Request(req, res):
                default: hosts.process_request
                    if hosts.upgrade_request(): return --ie, websockets
                    host = hosts.match(req:hosts())
                    if not host: generate conflict page
                    page = host:match
                    if 404: try the same, but with hosts.any if host.options.no_fallback
                         is not truthy.
                    if still not page: show error

                    page.callback(req, res, args...)
            return client:is upgraded() or keepalive --keep the connection alive if we
                upgraded or keepalived
        if handle_client did not return true:
            client:close()

        run an interation of the scheduler

upgrade_request():
    if not connection has upgrade part: return false
    get upgrade func from Connection header
    if not upgrade func:
        halt("invalid upgrade")
    else:
        upgrade_func(req, res)
            upgrade_func is responsible for setting
            is_upgraded, to prevent the connection from
            being closed
    return true
```

4 Bootstrapping

LuaFlare provides it's own flavour of Lua, coming with type-checking and default values for function arguments. To upgrade the current Lua state, some bootstrapping must be done first.

4.1 Syntax Extensions

LuaFlare provides some type-checking syntax. Before loading any code, either by `require()`, `dofile()`, or `include()`, LuaFlare will process the file, and translate the type information to an immediate call to `expects()`.

4.1.0.1 type arg `function(type arg)`

Tests `arg` against `"type"`.

4.1.0.2 :: `function meta::func()`

Test `self` against `meta`.

4.1.0.3 meta& arg `function(meta& arg)`

Tests `arg` against `meta`.

4.1.0.4 arg=default `function(msg="hello")`

Set `arg` to `default` if `arg == nil` (placed before `expects()`).

4.1.1 How `expects()` works

`expects()` will examine the stack, and compare it with the arguments that have been passed to it.

If the type passed type is a string, it will check it against the function `expects_types[typestr](value)` if it exists, else `type(value) == typestr`. The type string `"any"` will just check against a non-nil value.

If the passed type is nil, it will ignore this argument.

If the passed type is a table, it will ensure the value table contains the same functions (via `metatable_compatible()`).

`expects()` also checks against too many arguments being passed to it. So this will throw an error: `function(a) expects("string", "number")`

4.1.2 Examples, along with translations

4.1.2.1 Standard

```
function(string a, number b)
```

```
function(a, b) expects("string", "number")
```

4.1.2.2 `self` checking.

```
function meta::func()
```

```
function meta:func() expects(meta)
```

4.1.2.3 Metatable

```
function(meta& a)
```

```
function(a) expects(meta)
```

4.1.2.4 Complex

```
function meta::dosomething(string arg, meta& other, string message = "hello")
```

```
function meta:dosomething(arg, other, message) if message == nil then message =
"hello" end expects(meta, "string", meta, "string")
```

4.2 Bootstrap table

The bootstrap table contains a number of functions that may be used early in the LuaFlare boot process.

4.2.1 `bootstrap.pack(...)`

The same as `table.pack`, but guaranteed to be there.

4.2.2 `bootstrap.unpack(...)`

The same as `table.unpack`, but guaranteed to be there.

4.2.3 `bootstrap.options`

The options table that was passed when initializing the bootstrap process.

4.2.4 `bootstrap.log_buffer`

The log output generated by the bootstrap process.

4.2.5 `bootstrap.log_depth`

The current depth of the log (indentation).

4.2.6 `bootstrap.log_deeper()`

Increase the log depth.

4.2.7 `bootstrap.log_shallower()`

Decrease the log depth.

4.2.8 `bootstrap.log(string format, ...)`

Output logging information.

4.2.9 `bootstrap.fatal(string format, ...)`

Called upon an unrecoverable error, or a state that results in bootstrapping failing.

4.2.10 `bootstrap.loadfile(string path, ...)`

Load the file `path` in the "bootstrap/" directory.

4.2.11 `bootstrap.module(string name, string path)`

Loads a module from `path` and installs it as `name`.

Future calls to require(`name`) will return `path`'s return value.

4.2.12 `bootstrap.extend(string name, string path)`

Installs some functions/variables into the table `_G[name]`, with what `path` returns.

4.2.13 `bootstrap.level_string`

A human-readable version of `level_cache`.

4.2.14 `bootstrap.level_cache`

A table in the format of `[lvl] = true` where `lvl` is a name of a level. Used to ensure the right extensions are available.

4.2.15 `bootstrap.level(string name)`

If the level `name` has not been reached, error and exit.

4.2.16 `bootstrap.set_level(string name)`

Report that the level `name` has been reached.

4.3 Bootstrap Process

1. Ensure compatibility with old versions of Lua.
2. Install all the modules required by the extensions and/or translator.
    1. Install `hook.lua` into `luaflare.hook`.
    2. Install `stringreader.lua` into `luaflare.util.luaparser.stringreader`.
    3. Install `luaparser.lua` into `luaflare.util.luaparser`.
    4. Install `stack.lua` into `luaflare.util.stack`.
3. Extend Lua's default libraries.
    1. Install global type checking functions.
    2. Extend `string`.
    3. Extend `table`.
    4. Extend `math`.
    5. Extend `os`.
4. Setup the translator.
    1. Install `translate-luacode.lua` into `luaflare.util.translate_luacode`.
    2. Detour functions that need a translator (i.e. `require()`), and install `include()`.
5. Ensure forwards Lua compatibility.
6. Ensure backwards compatibility with old versions of LuaFlare.
7. Set the process name.

4.4 Automatic Circular Requires

Very often, a module may have a circular dependency, such as module “a” requires “b”, and module “b” requires “a”. In C/C++, such dependencies are resolved by using header guards. Such a system is not compatible with Lua as there is no such thing as a definition. Often, providing a reference to the table that is yet to be populated is enough, as the require-er does not need to use the library right away.

In standard Lua, an early reference that subsequent requires will return is set by filling in the `package.loaded` field for your module.

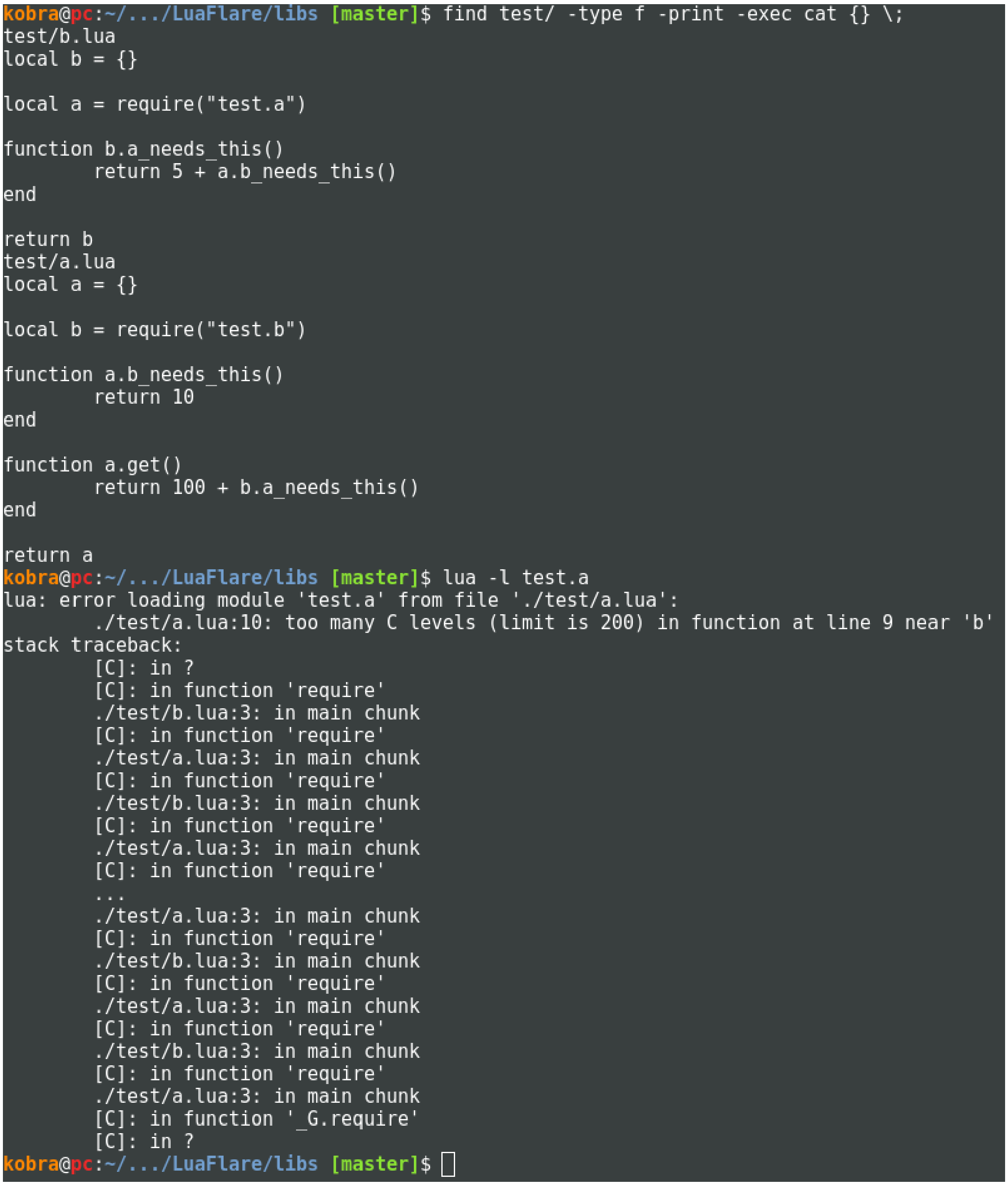
The module name is passed as the first argument, so you may fill in the package.loaded field via the following code:

```
local mod_name = {}
package.loaded[...] = mod_name
```

When doing this, the `return mod_name` at the end of the file is no longer necessary, but is kept for semantic purposes.

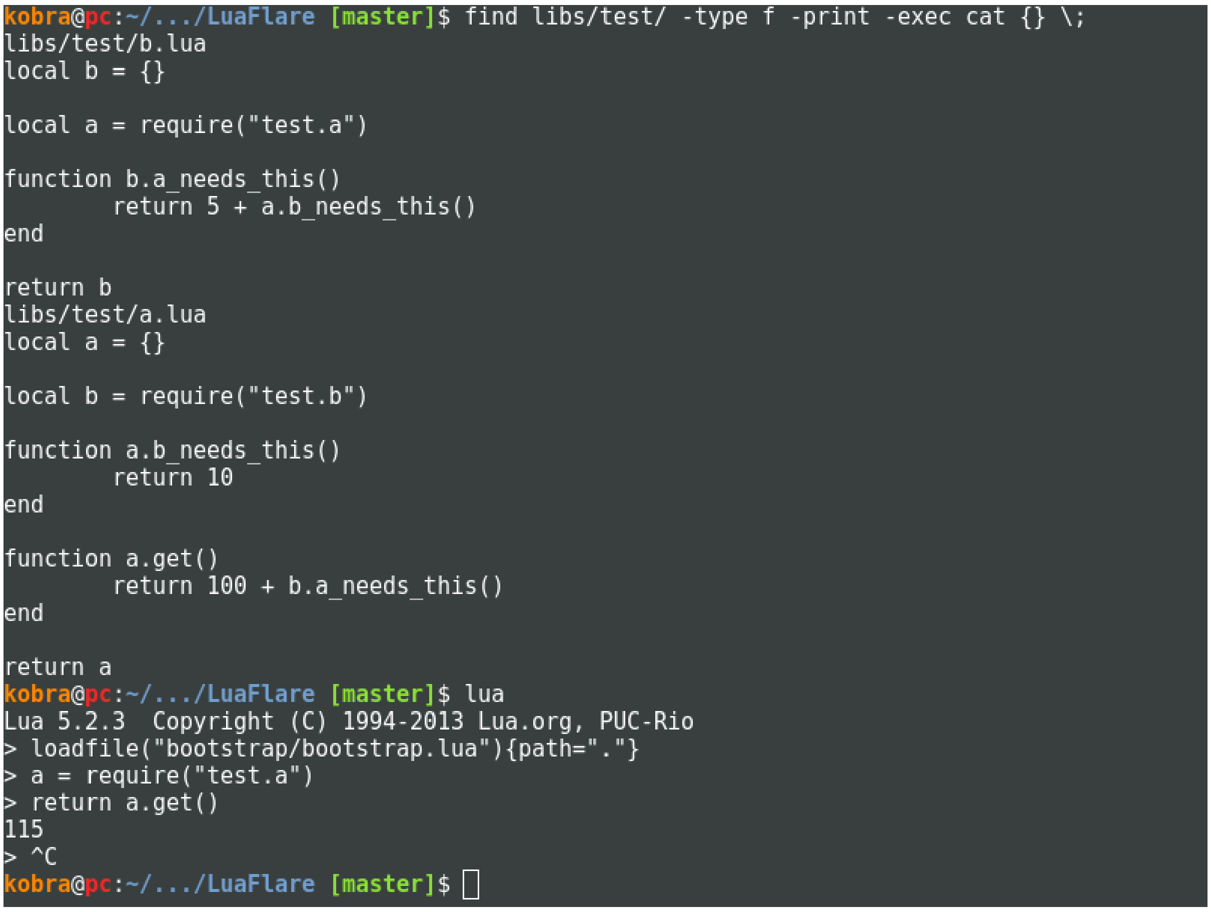
The bootstrap’s require will automatically detect modules where the first (meaningful) tokens are defining an empty table, *and* the file’s last tokens are returning the same table, then a reference right after the table has been created is installed into the package.loaded table. This allows, for 99% of cases for circular requires to be resolved.

This works because nearly all modules will only use the required modules after they’ve *returned* from the main function. In other words, the main ‘file function’ only populates the module table, loads & saves references to other modules, and then returns. The actual modules needed are often only called once everything has been populated. For these reasons, it is discouraged to use the ‘population’ function as a means to do work. If you need to do work on load, please use the hook named “Loaded”, which will be called after LuaFlare is started up.



```
kobra@pc:~/.../LuaFlare/libs [master]$ find test/ -type f -print -exec cat {} \;
test/b.lua
local b = {}

local a = require("test.a")

function b.a_needs_this()
        return 5 + a.b_needs_this()
end

return b
test/a.lua
local a = {}

local b = require("test.b")

function a.b_needs_this()
        return 10
end

function a.get()
        return 100 + b.a_needs_this()
end

return a
kobra@pc:~/.../LuaFlare/libs [master]$ lua -l test.a
lua: error loading module 'test.a' from file './test/a.lua':
        ./test/a.lua:10: too many C levels (limit is 200) in function at line 9 near 'b'
stack traceback:
        [C]: in ?
        [C]: in function 'require'
        ./test/b.lua:3: in main chunk
        [C]: in function 'require'
        ./test/a.lua:3: in main chunk
        [C]: in function 'require'
        ./test/b.lua:3: in main chunk
        [C]: in function 'require'
        ./test/a.lua:3: in main chunk
        [C]: in function 'require'
        ...
        ./test/a.lua:3: in main chunk
        [C]: in function 'require'
        ./test/b.lua:3: in main chunk
        [C]: in function 'require'
        ./test/a.lua:3: in main chunk
        [C]: in function 'require'
        ./test/b.lua:3: in main chunk
        [C]: in function 'require'
        ./test/a.lua:3: in main chunk
        [C]: in function '_G.require'
        [C]: in ?
kobra@pc:~/.../LuaFlare/libs [master]$
```

Figure 1: Circular require in Lua



```
kobra@pc:~/.../LuaFlare [master]$ find libs/test/ -type f -print -exec cat {} \;
libs/test/b.lua
local b = {}

local a = require("test.a")

function b.a_needs_this()
        return 5 + a.b_needs_this()
end

return b
libs/test/a.lua
local a = {}

local b = require("test.b")

function a.b_needs_this()
        return 10
end

function a.get()
        return 100 + b.a_needs_this()
end

return a
kobra@pc:~/.../LuaFlare [master]$ lua
Lua 5.2.3  Copyright (C) 1994-2013 Lua.org, PUC-Rio
> loadfile("bootstrap/bootstrap.lua"){path="."}
> a = require("test.a")
> return a.get()
115
> ^C
kobra@pc:~/.../LuaFlare [master]$
```

Figure 2: Circular require in bootstrapped Lua

5 LuaFlare global functions

5.1 expects_types

A table of type checkers for `expects()`, where key = typename, value = function.

5.2 valid, reason metatable_compatible(table base, value)

Returns whether `value` is compatible with the metatable `base`, plus an error reason if it is not.

5.3 expects(...)

Checks the caller's arguments against `...`, where each argument in `...` is of either:

- `"any"`: Anything but `nil`.
- `nil`: Argument not tested.
- `string`: Checks `expects_types[str](arg)` or `str == type(arg)`.
- `table`: Checks `tbl == arg` or `metatable_compatible(tbl, arg)`.

5.4 print_table(table tbl[, done[, depth]])

Prints a table.

5.5 number table.count(table t)

Counts the total number of elements in a table.

5.6 table.remove_value(table t, any value)

Removes all values from a table.

5.7 boolean table.is_empty(table t)

Returns whether or not the table is empty.

5.8 boolean table.has_key(table t, any key)

Checks to see if `t` has a key-value pair with the key of `key`.

5.9 string table.to_string(table t)

Returns a nice string representation of `t`.

5.10 boolean string.begins_with(string what, string with)

5.10.1 boolean string.starts_with(string what, string with)

Checks whether `what` begins with `with`.

5.11 `boolean string.ends_with(string what, string with)`

5.11.1 `boolean string.stops_with(string what, string with)`

Checks whether `what` ends with `with`.

5.12 `string string.replace(string in, string what, string with)`

Replaces all occurrence of `what` with `with` in `in`.

5.13 `string string.replace_last(string in, string what, string with)`

Replace the last occurrence of `what` with `with` in `in`.

5.14 `string string.path(string)`

TODO: remove this

5.15 `string string.trim(string in)`

Returns `in` without any white space padding.

5.16 `table string.split(string in, string delimiter[, table options])`

Turn `in` into a list separated by `delimiter`.

Valid options:

- `boolean remove_empty`

5.17 `number math.round(number in, number quantum_size = 1)`

Rounds a number to the smallest unit (`quantum_size`).

5.17.1 Example

```
math.round(1.55, 0.25) == 1.5
math.round(1.7, 0.25) == 1.75
math.round(5.5) == 6
math.round(math.pi, 0.001) == 3.142
```

5.18 `number math.secure_random(number min, number max)`

Returns a secure random number between (inclusive) `min` and `max`.

TODO: currently reads from /dev/urandom, should it read from /dev/random (tho, urandom is seeded by random).

5.19 `output, err_code os.capture(string cmd[, table options])`

Run a command and return the result.

Valid options:

- `boolean stdout`
- `boolean stderr`

5.20 `string name, number version os.platform()`

Returns the lower-case platform name, along with the version.

5.21 `warn(fmt[, ...])`

Dispatch a warning.

5.22 `... include(string path[, ...])`

Includes a file relative the the current file's directory.

Varargs are passed the file as arguments.

Returns the file's returns.

6 LuaFlare hook library

```
local hook = require("luaflare.hook")
```

6.1 *`hook.invalidate(any name)`*

Rebuilds the `callorder` table. Called automatically by `hook.add()` and `hook.remove()`.

- `name`: The hook name.

6.2 `hook.add(any name, any id, function callback, number priority = 0)`

Adds a hook. Returning a none-nil value will prevent callbacks yet-to-be-called from being invoked.

- `name`: The hook name.
- `id`: A unique ID for this hook.
- `callback`: The function to invoke upon the hook being called.
- `priority`: Hooks with a lower priority are called first (default `0`).

6.3 `hook.remove(any name, any id)`

Removes a hook.

- `name`: The hook `id` belongs to.
- `id`: The ID of the hook.

6.4 `... hook.call(any name, ...)`

Invokes all functions subscribed to this hook.

- `name`: The hook name.
- `...`: The arguments to the hook.
- returns: nil, unless a hook returned a none-nil value (meaning not all hooks were called).

6.5 `... hook.safe_call(any name, ...)`

- Same as `hook.call()`, except any errors are caught, and attempts to show the Lua error page.

7 LuaFlare hosts library

```
local hosts = require("luaflare.hosts")
```

7.1 `hosts.any`

The fallback (wildcard) site.

7.2 `hosts.developer`

The developers site. It is recommended you disable this in a production enviroment.

7.3 `host hosts.get(string pattern[, table options])`

Returns a host with the input pattern. If it does not already exist, then it will be created, and it's options set.

- `pattern`: A Lua pattern (along with wildcard support) to test incomming connections `Host` header against.
- `options`: A table of host options.
  - `no_fallback`: Don't fall back to `hosts.any` if a page could not be matched.
- returns: The host.

7.4 `host[, err] hosts.match(string host)`

Gets the host that matches `host`.

- `host`: The host to test against.
- returns: Either the matched host, or nil plus an error string.

7.5 *`hosts.process_request(request, response)`*

Finds the correct host and page, and invokes it. Will handle HTTP upgrades too.

7.6 *`hosts.upgrade_request(request, response)`*

Checks to see if this request should be upgraded.

- returns: true if it has eaten the request, else false.

7.7 `host:addpattern(string pattern, function callback)`

Add a route that matches `pattern`. Captures from the pattern are passed to callback after the request and response objects.

- `pattern`: The URL pattern.
- `callback`: The function; should be in the format `function(request, response, ...)` where ... are the captures from `pattern`.

7.7.1 Example

```
local function hello(req, res, msg)
    res:append(msg)
end
hosts.any:addpattern("/hello/(.+)", hello)
```

7.8 `host:add(string url, function callback)`

Adds a direct link to a function, no pattern matching is done.

- `url`: The URL to add.
- `callback`: The function; should be in the format `function(request, response, url)`.

7.9 `page, args[, errcode[, errstr]] host:match(string url)`

- `url`: The URL to test against.
- returns:
  - `page`: The page table. Is nil on error.
  - `args`: The array of arguments to pass to `page.callback`. Is nil on error.
  - `errcode`: The HTTP error code to send.
  - `errstr`: The reason for the error.

8 LuaFlare httpstatus library

Serves to translate between HTTP status codes and HTTP status messages.

```
local httpstatus = require("luaflare.httpstatus")
```

8.1 `httpstatus.known_statuses`

A table in of known HTTP statuses, where the key is the status number, and the value is the canonicalized status message.

8.2 `string httpstatus.tostring(number)`

Attempt to convert a status number to a string.

8.3 `number httpstatus.fromstring(string)`

Attempt to find a HTTP status code from a string.

9 LuaFlare mimetypes library

```
local mimetypes = require("luaflare.mimetypes")
```

Translate file extensions to mime types. Has basic types inbuilt, and loads the rest from `/etc/mime.types`

9.1 `mimetypes.types`

The loaded data. Key is file extension, value is mimetype.

9.2 `mimetypes.guess(string path)`

Returns the mimetype associated with `path`, or nil.

10 LuaFlare request object

```
local request = _G.Request(client)
```

The object that represents a request.

10.1 `string request:method()`

Returns the HTTP method used.

10.2 `table request:params()`

Returns a table of query parameters.

10.3 `table request:post_params()`

Returns a table of post parameters (query string like).

10.4 `string request:post_string()`

Returns the raw post data sent with this request.

10.5 `table request:headers()`

Returns a table of headers.

10.6 `string request:url()`

Returns the URL, without any query string.

10.7 `string request:full_url()`

Returns the URL, including the query string.

10.8 `table request:parsed_url()`

Returns all the parts of the URL in a table.

10.9 `tcpclient request:client()`

Returns the underlying tcpclient.

10.10 `number request:start_time()`

Returns when the request was created.

10.11 `number request:total_time()`

Returns the seconds passed since the request was created.

10.12 `string request:peer()`

Returns the IP address, following X-Real-IP if a reverse proxy is being used.

10.13 `string request:host()`

Returns the host the request is using.

HTTP/1.2 does not require the Host header to be set, if the first HTTP line specified it. LuaFlare sets the host for compatibility anyway, but you should still use this.

10.14 `request:parse_cookies()`

Parses the cookies. You shouldn't need to call this, `get_cookie()` and `cookies()` will call this automatically.

10.15 `string request:get_cookie(string name)`

Returns the value of the `name` cookie.

10.16 `table request:cookies()`

Returns a table of cookies.

10.17 `boolean request:is_upgraded()`

Returns true if this request has been upgraded to another type of connection.

10.18 `request:set_upgraded()`

Tell the request that it has been upgraded.

Once this has been called, LuaFlare *forgets* about this connection (does not close it). As well as avoiding to keep the connection alive (`Connection: keep-alive`).

11 LuaFlare response object

`local response = _G.Request(request)`

The object that represents a response.

11.1 Hooks

11.1.1 `"ListDirectory" request, response, string path[, table options]`

Called by `response:set_file()` if `can_list_directory` is truthy, and the target is a directory. The function is expected to write to the response.

11.1.2 `"Error" {type = code, message = reason}, request, response`

11.2 `request response:request()`

Returns the request we're responding to.

11.3 `tcpclient response:client()`

Returns the underlying tcpclient.

11.4 `response:set_status(number what)`

Sets the HTTP status to `what`.

11.5 `response:set_reply(string reply)`

Sets the response buffer, clearing it, if it wasn't already empty.

11.6 `response:append(string data)`

Append to the response buffer.

11.7 `string response:reply()`

Get the current response buffer.

11.8 `number response:reply_length()`

Returns the current length of the response buffer.

11.9 `response:clear()`

Clear everything (reset to default).

11.10 `response:clear_headers()`

Clear the headers.

11.11 `response:clear_content()`

Clear the content; retains the status code.

11.12 `response:halt(number code, reason)`

Report an error, and call the appropriate hooks for an error.

11.12.1 Example

```
response:halt(403, "Not logged in")
```

11.13 `response:set_file(string path[, table options])`

Sets the file to send.

If `X-Accel-Redirect` or `X-Sendfile` is on, it will use these to serve the file.

Valid options:

- `can_list_directory`: If the target is a directory, should we list the directory?

11.14 `response:set_header(string name, any value)`

Sets the header to a value.

11.15 `response:remove_header(string name)`

Removes a header completely.

11.16 `response:set_cookie(string name, string value[, string path[, string domain[, number lifetime]]])`

Adds cookies to be sent.

11.17 `response:etag()`

Returns a hash/mostly-unique string that changes when the content does.

11.18 `response:use_etag()`

Returns whether we should use an etag.

Reasons they may not be used are: - Exceeds file-size limit. - Etags turned off.

11.19 `response:send()`

Sends the request.

Once this has been sent, future calls to `send()` will do nothing.

12 LuaFlare scheduler library

`local scheduler = require("luaflare.scheduler")`

Allows you to periodically run tasks.

12.1 `scheduler.newtask(string name, function func)`

Creates a new scheduled task. Return from `func` to exit the task, and yield (`coroutine.yield()`) the number of seconds you want to wait.

12.2 `scheduler.run()`

Resumes all scheduled tasks. Any tasks that take longer than half a second to either yield or return results in a warning, as during the time your scheduled task is running, LuaFlare is hung.

12.3 `number scheduler.idletime()`

Returns the number of seconds until the next scheduled task is to be resumed; `-1` if complete.

12.4 `boolean scheduler.done()`

Returns true if all tasks are complete.

13 LuaFlare session library

`local session = require("luaflare.session")`

Provides session information for a request.

13.1 Hooks

13.1.1 `session "GetSession" request, response, string name, string id`

Used to create a session object.

13.1.1.1 `default textfile session`

This is the default handler for this hook; it save to small textfiles in `$configdir/sessions/`.

This hook has a priority of 1. To override it, make sure that your hook's priority is less than 1.

13.2 `session.valid_chars`

When generating a new session ID, use these characters to do it.

13.3 `session session.get(request, response, string name = "session")`

Returns a session object that matches the session name.

If the session does not exist, it is created, along with setting the response cookies for said session.

13.4 *`session:construct(request, response, name, id)`*

Internal function to construct a textfile session. Used by the hook "default textfile session" via "GetSession".

13.5 `session:save()`

Saves any changes to the session.

13.6 `string session:id()`

Return the ID of the session.

13.7 `table session:data()`

Return loaded data. If you make changes, save them with `session:save()`.

14 LuaFlare socket API

LuaFlare itself does not implement sockets directly, but instead offers an API that backends can implement.

`local socket = require("luaflare.socket")`

The socket API version this document targets is 0.1.

14.1 Functions with timeout arguments

If a function has a timeout argument, this means it supports the standard socket timeout behaviour. This is where if the timeout value (`t`) is less than 0, the function may wait forever; if `t` is equal to 0, the function inhibits non-blocking behaviour; and if `t` is above 0, the function may wait either for a maximum of `t` seconds, or until there has been no activity for `t` seconds.

14.1.1 Future

In future, -1 may mean forever, and -2 to use the client's `:set_timeout()` value, although this is currently not in this API version.

14.2 Top level functions or values

14.2.1 `socket.backend`

The name of the socket backend. The CLI option `--socket-backend` is used to decide which backend to use.

14.2.2 `socket.api_version`

The API this backend implements.

This version is taken from the latest LuaFlare version at which these are still compatible. The API version can be read in "libs/luaflare/socket/none.lua", or by running `print(require("luaflare.socket.none").api_version)`.

14.2.3 `listener[, err] socket.listen(number port = 0, string address = "*")`

Bind to an address (start listening for connections).

If `port` is 0, then the operating system will choose it's own port, usually only temporary (ephemeral). Address is the address to listen from, "*" will listen to all addresses on all local interfaces.

Returns either the listening object, or `nil` plus an error string.

14.2.4 `client[, err] socket.connect(string host, number port)`

Connect to a remote host on the specified port. The host may be a hostname or an IP address.

Returns client or `nil` plus error string.

14.3 Listener functions

14.3.1 `client[, err] listener:accept(number timeout = -1)`

Accept a client from the queue.

14.3.2 `number listener:port()`

Returns the port number we are/were listening on.

14.3.3 `listener:close()`

Stop listening.

14.4 Client functions

14.4.1 `type, backend, version client:type()`

Returns the type of connection, the backend name, and the backend API version.

14.4.2 `string client:ip()`

Returns the IP address of the client.

14.4.3 `number client:port()`

Gets the port this client is connecting on.

14.4.4 `boolean client:connected()`

Returns whether or not this client is connected.

14.4.5 `data[, err] client:read(string format = "a", number length = 0, number timeout = -1)`

Read up to `length` bytes from the stream (infinite if `length` is 0), or until the format condition is met; whichever comes first.

The valid formats are either “a”, to the end of the stream; and “l”, to the end of the line. The formats may be prefixed with an asterisk (*) to maintain API semantics with Lua 5.2 and below; this behaviour was deprecated in Lua 5.3 and above.

14.4.6 `boolean[, err] client:write(string data)`

Write data to the socket; returns `true` if it succeeds, or `false` plus an error string if it fails.

14.4.7 `boolean[, err] client:flush(number timeout = -1)`

Flush all written data; returns `true` if it succeeds, or `false` plus an error string if it fails.

14.4.8 `client:close()`

Closes the connection. Does not error if it has already been closed.

15 LuaFlare tags library

`local tags = require("luaflare.tags")`

Provides HTML generation with automatic escaping.

Use like:

```
tags.div {attrib1 = "value", attrib2 = value}
{
   tags.em { "Children" }
}.to_(html|response)
```

15.1 `string tbl.to_html(section = 0)`

Returns the HTML tbl represents.

15.2 `string tbl.print(section = 0)`

Writes the HTML to stdout.

15.3 `string tbl.to_response(section = 0)`

Writes the HTML to the response.

15.4 Valid Tags

Name	Options	Special Function
SECTION		Mark a section; no output
NOESCAPE		Don't escape the next element.
html	pre text = "<!DOCTYPE html>\n"	
head		
body		
script	escaper = striptags	
style	escaper = striptags	
link	empty element	
meta	empty element	
title	inline	
div		
header		
main		
footer		
br	inline, empty element	
img	empty element	
image	empty element	
a	inline	

Name	Options	Special Function
p	inline	
span	inline	
code	inline	
h1	inline	
h2	inline	
h3	inline	
h4	inline	
h5	inline	
h6	inline	
b	inline	
i	inline	
em	inline	
u	inline	
center	inline	
pre		
table		
ul		
li	inline	
tr		
td		
tc		
form		
input		
textarea		

16 LuaFlare threadpool library

```
local threadpool = require("luaflare.threadpool")
```

16.1 `pool threadpool.create(number threads, function func)`

Creates and returns a threadpool.

`func(obj)` is called for each `enqueue()`ed object.

16.2 `pool:enqueue(any object)`

Adds an object to the queue.

16.3 *`object pool:dequeue()`*

Removes and returns the first item from the queue.

16.4 `pool:done()`

Returns whether or not the queue is empty.

16.5 `pool:step()`

Resume all threads.

17 LuaFlare canonicalize_header function

```
local canonicalize_header = require("luaflare.util.canonicalize_header")
```

17.1 `header canonicalize_header(string header)`

Returns the conanical form of `header`. Such as `"host"` to `"Host"`, or `"content-length"` to `"Content-Length"`.

18 LuaFlare escape library

`local escape = require("luaflare.util.escape")`

Provides methods to escape strings to their safe(er) forms.

18.1 `string escape.pattern(string input)`

Escapes a Lua pattern.

- `(` -> `%(`
- `)` -> `%)`
- `.` -> `%.`
- `%` -> `%%`
- `+` -> `%+`
- `-` -> `%-`
- `*` -> `%*`
- `?` -> `%?`
- `[` -> `%[`
- `]` -> `%]`
- `^` -> `%^`
- `$` -> `%$`

18.2 `string escape.html(string input)`

Escapes a HTML string.

18.3 `string escape.striptags(string input)`

Strips all tags from a string.

18.4 `string escape.sql(string input)`

Returns a safe string to use in SQL queries.

18.5 `string escape.mysql(string input)`

Returns a safe string to use in MySQL queries.

18.6 `string escape.argument(string input, boolean quotify = true)`

If `quotify`, then the string will be encapsulated in double quotes with a couple special chars escaped; otherwise, special chars are prefixed with a backslash.

Escapes a Unix shell argument.

19 LuaFlare unescape library

```
local unescape = require("luaflare.util.unescape")
```

Turns strings into their more litteral sense.

19.1 `string unescape.sql(string input)`

Unescape an SQL escaped string.

20 LuaFlare luaparser library

`local parser = require("luaflare.util.luaparser")`

Tokenize Lua code. Used for syntax extensions.

20.1 `parser.strict`

Should the parser `error()` on an issue, or try to resume?

20.2 `parser.problem(str, depth)`

Used when parsing Lua code to report a problem.

20.3 `... parser.assert(check, [msg | ...])`

If check is falsy, and strict then error. else `return check, ...`.

20.4 `parser.keywords`

A list of keywords in the format of `["keyword"] = true`.

Used to mark identifieres as keywords in the parser.

20.5 `parser.tokenchars_joinable`

A list of chars that “compress” together to form one token.

20.6 `parser.valid_tokens`

A list of valid tokens.

20.7 `parser.operator_precedence`

A table that stores the priority (precedence) of operators.

20.8 `parser.escapers`

A list of valid string escapers in the form `pattern = function`.

20.9 `parser.brackets_create`

Brackets that increase the bracket depth.

20.10 `parser.brackets_destroy`

Brackets that decrease the bracket depth.

20.11 `tokens parser.tokenize(string code)`

Turns `code` into a list of tokens. Tokens are in the format of:

```
{
   type = string --[[ hashbang, comment, string,
                    number, newline, whitespace,
                    token, keyword, identifier,
                    unknown, or more... ]]
   value = string,
   range = {from, to},
   chunk = string,
}
```

20.12 `parser.scope_create`

A list of keywords (`["keyword"] = true`) that create scopes.

20.13 `parser.scope_destroy`

A list of keywords (`["keyword"] = true`) that destroy scopes.

20.14 `rootscope parser.read_scopes(table tokens)`

Attempts to match scopes to the tokens.

Scopes are in the format:

```
{
   starts = number,
   ends = number,
   starttoken = token,
   locals = table,
   children = table
}
```

21 LuaFlare stringreader object

`local stringreader = require("luaflare.util.luaparser.stringreader")`

A helper library used by `luaparser`.

21.1 `reader stringreader.new(string data)`

Constructs a reader object.

21.2 `string reader:read(number count = 1)`

Reads `count` bytes from the stream, and increases the position.

21.3 `string reader:peek(number count = 1)`

Reads `count` bytes from the stream. Does not increase the position.

21.4 `string reader:peekat(number offset, number count = 1)`

Peeks `count` bytes at position `offset`.

21.5 `string reader:peekmatch(string pattern)`

Returns the match at the current position.

21.6 `string reader:readmatch(string pattern)`

Returns the match at the current position, along with increasing the position.

21.7 `boolean reader:eof()`

Returns whether the end has been reached.

22 LuaFlare util library

```
local util = require("luaflare.util")
```

22.1 `number util.time()`

Returns accurate time.

22.2 `util.iterate_dir(string dir, boolean recursive, function callback, ...)`

Iterates a directory, calling `callback` for each path.

22.3 `boolean util.dir_exists(string dir)`

Returns whether or not a directory is valid and exists.

22.4 `table util.dir(string basedir, boolean recursive = false)`

Returns the files in a directory, recursively.

22.5 `boolean util.ensure_path(string path)`

Checks to see if a directory exists. If it does not, then it is created.

Returns true if it exists, false, if it couldn't be created.

23 LuaFlare script library

```
local script = require("luaflare.util.script")
```

23.1 `script.options`

A table of options parsed with `script.parse_options()`.

23.2 `script.arguments`

An array of arguments that were parsed with `script.parse_options()`

23.3 `script.cfg_blacklist`

A list of options that should *not* be saved to disk (i.e. `--help` and `--version`).

23.4 `script.parse_arguments(table args, table shorthands, boolean nosave = false)`

Parse the arguments table into options and arguments. Shorthands are in the form `["x"] = "big-x"`. If `nosave` is true, then the configuration file will not be saved.

23.5 `integer script.pid()`

Returns the PID of LuaFlare.

23.6 `string script.instance()`

Returns instance information for the caller.

23.7 `string script.instance_info()`

Returns a complete string used to identify this instance.

24 LuaFlare slug library

Generate human-readable IDs from an input string (usually a title).

```
local slug = require("luaflare.util.slug")
```

24.1 `script.readable_chars`

A table of all readable chars.

24.2 `script.aliases`

A table of chars and their new values.

24.3 `string script.slug_char(character x, number depth = 0)`

Turns the character `x` into a slug part (excluding spaces).

24.4 `string slug.generate(string input)`

Turns the input string into an ID-safe and human-readble string.

24.4.1 Examples

- `"Hello, world!"` -> `"hello-world"`
- `"Abc:  does foo & bar cause pootis?"` -> `"abc-does-foo-and-bar-cause-pootis"`
- `"10 things for $10"` -> `"10-things-for-usd10"`

25 LuaFlare virtualfilesystem library

```
local vfs = require("luaflare.virtualfilesystem")
```

25.1 `string vfs.locate(string path, boolean fallback = false)`

Translates site relative file locations relative to the current working directory.

25.2 `array vfs.ls(string path, table options = {})`

Returns a list of files and folders.

List of valid options:

- `type`: The type (mode) of the file **must** match this. The type of file may be of either `directory`, `file`, `link`, `socket`, `named pipe`, `char device`, `block device`, or `other`.
- `recursive`: When encountering another directory, should we recurse into it?
- `tester`: A function to test each file. The arguments passed are: `string file, table options, table attributes, boolean default`.

26 LuaFlare websocket library

```
local websocket = require("luaflare.websocket")
```

26.1 Example

The following is an example echo server that sends all received messages to all connected clients.

```
local echo = websocket.register("/echo", "echo")
function echo:on_message(client, message)
   self:send(string.format("%s: %s", client.peer, message))
end
```

26.2 `websocket.registered`

The websockets that have been registered mapped by path and protocol (`[path][protocol]`).

26.3 `hosts.upgrades.websocket(request, response)`

The function responsible for upgrading a HTTP request to a websocket connection.

26.4 `wsserver websocket.register(string path, string protocol)`

Registers a websocket.

Valid callbacks:

- `wsserver:on_connect(client)`
- `wsserver:on_message(client, message)`
- `wsserver:on_disconnect(client)`

26.5 `wsserver:send(string message[, client])`

Sends a message to `client` (or all connected if `client` is absent).

26.6 `wsserver:wait()`

Yeild (via scheduler) with an appropriate number of seconds.