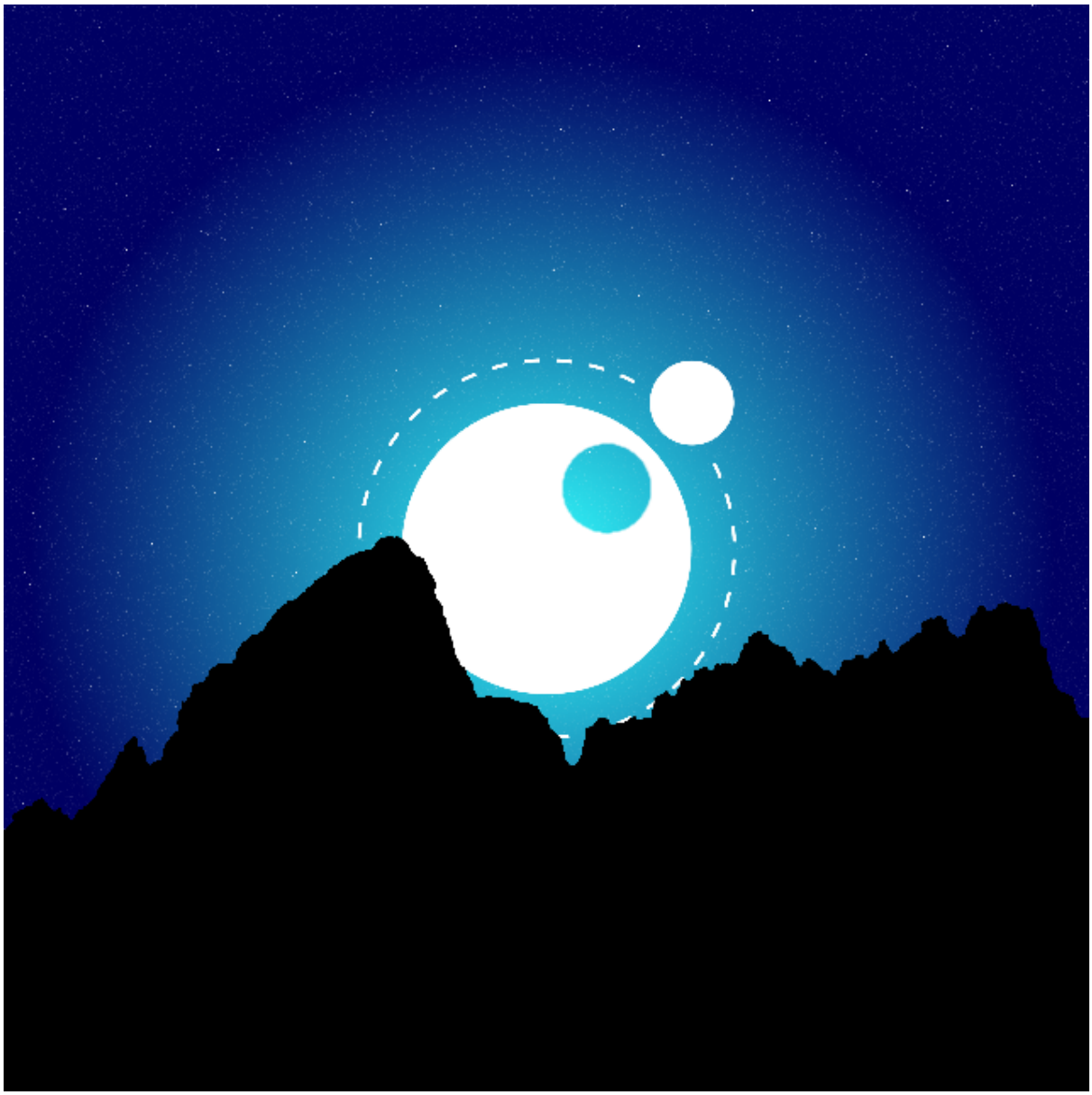
LuaFlare Documentation

Kate Adams <self@kateadams.eu>

November 29, 2014



Contents

1 Install LuaFlare on Debian based distros

This simple guide will show you how to install LuaFlare on a fresh install of Debian, compatible with Ubuntu/Mint.

1.1 Either install via apt-get (my repo @ kateadams.eu)

May not be bleeding edge, but is updated via apt-get.

1.1.1 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/
```

1.1.2 apt keys

```
sudo apt-key adv --keyserver keys.gnupg.net --recv ED672012
```

1.1.3 install

```
sudo apt-get update
sudo apt-get install luaflare luaflare-service luaflare-reverseproxy-nginx
```

Please continue to Enable the Nginx site.

1.1.4 enable service

1.1.4.1 systemd (Debian)

```
sudo systemctl enable luaflare
```

1.1.4.2 upstart or sysvinit (Ubuntu)

```
sudo update-rc.d luaflare enable
```

1.2 Or install via git (Makefile)

Bleeding edge, must be updated manually.

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

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

1.2.2 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
```

1.3 Enable the Nginx site

1.3.1 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
```

1.3.1.1 Enabling LuaFlare's site

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

1.4 Others/Help

1.4.1 Old method to import keys from gpg

```
gpg --recv-keys ED672012
gpg -a --export ED672012 | sudo apt-key add -
```

1.4.2 Alternate method to get the keys for apt

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

2 Internal Workings

2.1 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.

2.2 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).

2.3 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[...]` exists, where `...` 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.

2.4 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
```

3 LuaFlare global functions

3.1 `expects_types`

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

3.2 `valid, reason metatable_compatible(table base, value)`

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

3.3 `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)`.

3.4 `print_table(table tbl[, done[, depth]])`

Prints a table.

3.5 `number table.count(table t)`

Counts the total number of elements in a table.

3.6 `table.remove_value(table t, any value)`

Removes all values from a table.

3.7 `boolean table.is_empty(table t)`

Returns whether or not the table is empty.

3.8 'boolean table.has_key(table t, any key)

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

3.9 `string table.to_string(table t)`

Returns a nice string representation of `t`.

3.10 `boolean string.begins_with(string what, string with)`

3.10.1 `boolean string.starts_with(string what, string with)`

Checks whether `what` begins with `with`.

3.11 boolean string.ends_with(string what, string with)

3.11.1 boolean string.stops_with(string what, string with)

Checks whether `what` ends with `with`.

3.12 string string.replace(string in, string what, string with)

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

3.13 string string.replace_last(string in, string what, string with)

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

3.14 string string.path(string)

TODO: remove this

3.15 string string.trim(string in)

Returns `in` without any white space padding.

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

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

Valid options:

- `boolean remove_empty`

3.17 number math.round(number in, number quantum_size = 1)

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

3.17.1 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
```

3.18 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).

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

Run a command and return the result.

Valid options:

- `boolean stdout`
- `boolean stderr`

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

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

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

Dispatch a warning.

3.22 `... 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.

4 LuaFlare hook library

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

4.1 *hook.invalidate(any name)*

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

- `name`: The hook name.

4.2 hook.add(any name, any id, function callback[, number priority])

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`).

4.3 hook.remove(any name, any id)

Removes a hook.

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

4.4 ... 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).

4.5 ... hook.safe_call(any name, ...)

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

5 LuaFlare hosts library

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

5.1 hosts.any

The fallback (wildcard) site.

5.2 hosts.developer

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

5.3 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.

5.4 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.

5.5 *hosts.process_request(request, response)*

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

5.6 *hosts.upgrade_request(request, response)*

Checks to see if this request should be upgraded.

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

5.7 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`.

5.7.1 Example

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

5.8 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)`.

5.9 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.

6 LuaFlare httpstatus library

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

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

6.1 `httpstatus.know_statuses`

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

6.2 `string httpstatus.tostring(number)`

Attempt to convert a status number to a string.

6.3 `number httpstatus.fromstring(string)`

Attempt to find a HTTP status code from a string.

7 LuaFlare Lua 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()`.

7.1 Syntax

7.1.1 type arg `function(type arg)`

Tests `arg` against `"type"`.

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

Test `self` against `meta`.

7.1.3 meta& arg `function(meta& arg)`

Tests `arg` against `meta`.

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

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

7.2 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 none-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")"`

7.3 Examples, along with translations

7.3.1 1 - Standard

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

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

7.3.2 2 - `self` checking.

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

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

7.3.3 3 - Metatable

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

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

7.3.4 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")
```

8 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`

8.0.1 `mimetypes.types`

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

8.0.2 `mimetypes.guess(string path)`

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

9 LuaFlare request object

`local request = _G.Request(client)`

The object that represents a request.

9.0.1 `string request:method()`

Returns the HTTP method used.

9.0.2 `table request:params()`

Returns a table of query parameters.

9.0.3 `table request:post_params()`

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

9.0.4 `string request:post_string()`

Returns the raw post data sent with this request.

9.0.5 `table request:headers()`

Returns a table of headers.

9.0.6 `string request:url()`

Returns the URL, without any query string.

9.0.7 `string request:full_url()`

Returns the URL, including the query string.

9.0.8 `table request:parsed_url()`

Returns all the parts of the URL in a table.

9.0.9 `tcpclient request:client()`

Returns the underlying tcpclient.

9.0.10 `number request:start_time()`

Returns when the request created.

9.0.11 `number request:total_time()`

Returns the seconds passed since the request was created.

9.0.12 `string request:peer()`

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

9.0.13 `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.

9.0.14 `request:parse_cookies()`

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

9.0.15 `string request:get_cookie(string name)`

Returns the value of the `name` cookie.

9.0.16 `table request:cookies()`

Returns a table of cookies.

9.0.17 `boolean request:is_upgraded()`

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

9.0.18 `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`).

10 LuaFlare response object

`local response = _G.Request(request)`

The object that represents a response.

10.1 Hooks

10.1.1 "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.

10.2 request response:request()

Returns the request we're responding to.

10.3 tcpclient response:client()

Returns the underlying tcpclient.

10.4 response:set_status(number what)

Sets the HTTP status to `what`.

10.5 response:set_reply(string reply)

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

10.6 response:append(string data)

Append to the response buffer.

10.7 string response:reply()

Get the current response buffer.

10.8 number response:reply_length()

Returns the current length of the response buffer.

10.9 response:clear()

Clear everything (reset to default).

10.10 response:clear_headers()

Clear the headers.

10.11 `response:clear_content()`

Clear the content; retains the status code.

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

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

10.12.1 Example

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

10.13 `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?

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

Sets the header to a value.

10.15 `response:remove_header(string name)`

Removes a header completely.

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

Adds cookies to be sent.

10.17 `response:etag()`

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

10.18 `response:use_etag()`

Returns whether we should use an etag.

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

10.19 `response:send()`

Sends the request.

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

11 LuaFlare scheduler library

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

Allows you to periodically run tasks.

11.1 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.

11.2 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.

11.3 number scheduler.idletime()

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

11.4 boolean scheduler.done()

Returns true if all tasks are complete.

12 LuaFlare session library

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

Provides session information for a request.

12.1 Hooks

12.1.1 `session, function save_func "GetSession" string name, string id`

Used in session loading. Replaces the default save function with `save_func`.

12.2 `session.valid_chars`

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

12.3 `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.

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

Loads the data either by an answered `hook.call("GetSession")` **(TODO)**, or from disk using `table.load`.

12.5 `session:save()`

Saves any changes to the session.

12.6 `string session:id()`

Return the ID of the session.

12.7 `table session:data()`

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

13 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)
```

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

Returns the HTML tbl represents.

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

Writes the HTML to stdout.

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

Writes the HTML to the response.

13.4 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		

14 LuaFlare threadpool library

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

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

Creates and returns a threadpool.

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

14.2 `pool:enqueue(any object)`

Adds an object to the queue.

14.3 *`object pool:dequeue()`*

Removes and returns the first item from the queue.

14.4 `pool:done()`

Returns whether or not the queue is empty.

14.5 `pool:step()`

Resume all threads.

15 LuaFlare canonicalize_header function

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

15.1 `header canonicalize_header(string header)`

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

16 LuaFlare escape library

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

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

16.1 `string escape.pattern(string input)`

Escapes a Lua pattern.

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

16.2 `string escape.html(string input)`

Escapes a HTML string.

16.3 `string escape.striptags(string input)`

Strips all tags from a string.

16.4 `string escape.sql(string input)`

Returns a safe string to use in SQL queries.

16.5 `string escape.argument(string input)`

Escapes a Unix shell argument.

17 LuaFlare unescape library

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

Turns strings into their more litteral sense.

17.1 `string unescape.sql(string input)`

Unescape an SQL escaped string.

18 LuaFlare luaparser library

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

Tokenize Lua code. Used for syntax extensions.

18.1 parser.strict

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

18.2 parser.problem(str, depth)

Used when parsing Lua code to report a problem.

18.3 ... parser.assert(check, [msg | ...])

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

18.4 parser.keywords

A list of keywords in the format of `["keyword"] = true`.
Used to mark identifieres as keywords in the parser.

18.5 parser.tokenchars_joinable

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

18.6 parser.tokenchars_unjoinable

A list of chars that each represent one token.

18.7 parser.escapers

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

18.8 parser.brackets_create

Brackets that increase the bracket depth.

18.9 parser.brackets_destroy

Brackets that decrease the bracket depth.

18.10 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,
}
```

18.11 parser.scope_create

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

18.12 parser.scope_destroy

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

18.13 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
}
```

19 LuaFlare stringreader object

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

A helper library used by `luaparser`.

19.1 `reader stringreader.new(string data)`

Constructs a reader object.

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

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

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

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

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

Peeks `count` bytes at position `offset`.

19.5 `string reader:peekmatch(string pattern)`

Returns the match at the current position.

19.6 `string reader:readmatch(string pattern)`

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

19.7 `boolean reader:eof()`

Returns whether the end has been reached.

20 LuaFlare util library

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

20.1 number util.time()

Returns accurate time.

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

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

20.3 boolean util.dir_exists(string dir)

Returns whether or not a directory is valid and exists.

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

Returns the files in a directory, recursively.

20.5 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.

21 LuaFlare virtualfilesystem library

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

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

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

22 LuaFlare websocket library

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

22.1 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
```

22.2 `websocket.registered`

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

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

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

22.4 `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)`

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

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

22.6 `wsserver:wait()`

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