Turbo.lua Documentation

Release 2.1.1

John Abrahamsen

Aug 31, 2017

Contents

Turbo.lua is a framework built for LuaJIT 2 to simplify the task of building fast and scalable network applications. It uses a event-driven, non-blocking, no thread design to deliver excellent performance and minimal footprint to high-load applications while also providing excellent support for embedded uses. The toolkit can be used for HTTP REST API's, traditional dynamic web pages through templating, open connections like WebSockets, or just as high level building blocks for native speed network applications.

First and foremost the framework is aimed at the HTTP(S) protocol. This means web developers and HTTP API developers are the first class citizens. But the framework contains generic nuts and bolts such as; a I/O loop, IO Stream classes, customizeable TCP (with SSL) server classes giving it value for everyone doing any kind of high performance network application.

CHAPTER 1

Hello World

The traditional and mandatory 'Hello World'

```
local turbo = require("turbo")

local HelloWorldHandler = class("HelloWorldHandler", turbo.web.RequestHandler)

function HelloWorldHandler:get()
    self:write("Hello World!")
end

turbo.web.Application({
    {"/hello", HelloWorldHandler}
}):listen(8888)
turbo.ioloop.instance():start()
```

LuaJIT 2 is REQUIRED, PUC-RIO Lua is unsupported.

Git repository is available at https://github.com/kernelsauce/turbo.

It's main features and design principles are:

- Simple and intuitive API (much like Tornado).
- Low-level operations is possible if the users wishes that.
- Implemented in straight Lua and LuaJIT FFI on Linux, so the user can study and modify inner workings without too much effort. The Windows implementation uses some Lua modules to make compability possible.
- Good documentation
- Event driven, asynchronous and threadless design
- Small footprint
- SSL support (requires OpenSSL or LuaSec module for Windows)

Travis Linux CI Appveyor Windows CI

CHAPTER 2

Supported Architectures

x86, x64, ARM, PPC, MIPSEL

CHAPTER 3

Supported Operating Systems

Linux distros (x86, x64), OSX (Intel-based) and Windows x64. Possibly others using LuaSocket, but not tested or supported.

CHAPTER 4

Installation

You can use LuaRocks to install Turbo on Linux.

```
luarocks install turbo
```

If installation fails make sure that you have these required pacakages:

```
apt-get install luajit luarocks git build-essential libssl-dev
```

For Windows use the included install.bat. This will install all dependencies: Visual Studio, git, mingw, gnuwin, openssl using Chocolatey. LuaJIT, the LuaRocks package manager and Turbo will be installed at C:\turbo.lua. It will also install LuaSocket, LuaFileSystem and LuaSec with LuaRocks. The Windows environment will be ready to use upon success.

Try: `luajit C:\turbo.lua\src\turbo\examples\helloworld.lua`

If any of the .dll or. so's are placed at non-default location then use environment variables to point to the correct place:

E.g: `SET TURBO_LIBTFFI=C:\turbo.lua\src\turbo\libtffi_wrap.dll` and `SET TURBO_LIBSSL=C:\Program Files\OpenSSL\libeay32.dll`

Applies for Linux based OS and OSX only:

Turbo.lua can also be installed by the included Makefile. Simply download and run `make install` (requires root priv). It is installed in the default Lua 5.1 and LuaJIT 2.0 module directory.

You can specify your own prefix by using `make install PREFIX=<prefix>`, and you can specify LuaJIT version with a `LUAJIT_VERSION=2.0.0` style parameter.

To compile without support for OpenSSL (and SSL connections) use the make option SSL=none. To compile with axTLS support instead of OpenSSL use the make option SSL=axTLS.

In essence the toolkit can run from anywere, but is must be able to load the libtffi_wrap.so at run time. To verify a installation you can try running the applications in the examples folder.

CHAPTER 5

Object oriented Lua

Turbo.lua are programmed in a object oriented fashion. There are many ways to do object orientation in Lua, this library uses the Middleclass module. Which is documented at https://github.com/kikito/middleclass/wiki. Middleclass is being used internally in Turbo Web, but is also exposed to the user when inheriting from classes such as the `turbo.web.RequestHandler` class. Middleclass is a very lightweight, fast and very easy to learn if you are used to Python, Java or C++.

CHAPTER 6

Packaging

The included Makefile supports packaging the current tree as a versioned tar.gz file. This file will include only the neccessary bits and pieces for Turbo to run. The files will be built as bytecode (luajit -b -g) with debug info. This reduces size drastically. Suitable for embedded devices with limited storage... It also reduces the startup time.

Use `make package`. Only tested on Linux and OSX.

this results in a turbo.x.x.x.tar.gz file and a package/ directory being created.

CHAPTER 7

Dependencies

All of the modules of Turbo.lua are made with the class implementation that Middleclass provides. https://github.com/kikito/middleclass.

The HTTP parser by Ryan Dahl is used for HTTP parsing. This is built and installed as part of the package.

OpenSSL is required for SSL support. It is possible to run without this feature, and thus not need an SSL library.

CHAPTER 8

License

CHAPTER 9

Tutorials

Get Started With Turbo

A set of simple examples to get you started using Turbo.

Installing Turbo

Turbo needs LuaJIT to run, because it uses the LuaJIT FFI library it will not run on 'official/normal/vanilla' Lua. Quick install on Debian/Ubuntu (you may need to add sudo or run these as root user):

```
$ apt-get install luajit luarocks git build-essential libssl-dev
$ luarocks install turbo
```

You can also install Turbo use the included Makefile in the project source:

```
$ git clone https://github.com/kernelsauce/turbo.git
$ cd turbo && make install
```

You can provide a PREFIX argument to the make which will install Turbo in a specified directory.

For Windows users it is recommended to use the included install.bat file or running the one-line command below from a administrative command line. Beware that this will compile and install all dependencies: Visual Studio, git, mingw, gnuwin, openssl using Chocolatey. LuaJIT, the LuaRocks package manager and Turbo will be installed at C:\turbo.lua. Unfortunately it is required to have Visual Studio to effectively use LuaRocks on Windows. LuaRocks will be used to install LuaSocket and LuaFileSystem. The Windows environment will be ready to use upon success, and the luajit and luarocks commands will be in your Windows environment PATH.

```
powershell -command "& { iwr https://raw.githubusercontent.com/kernelsauce/turbo/
↪master/install.bat -OutFile t.bat }" && t.bat
```

Hello World

The traditional and mandatory 'Hello World'

```
-- Import turbo,
local turbo = require("turbo")

-- Create a new requesthandler with a method get() for HTTP GET.
local HelloWorldHandler = class("HelloWorldHandler", turbo.web.RequestHandler)
function HelloWorldHandler:get()
    self:write("Hello World!")
end

-- Create an Application object and bind our HelloWorldHandler to the route '/hello'.
local app = turbo.web.Application:new({
    {"/hello", HelloWorldHandler}
})

-- Set the server to listen on port 8888 and start the ioloop.
app:listen(8888)
turbo.ioloop.instance():start()
```

Save the file as helloworld.lua and run it with `luajit helloworld.lua`.

Request parameters

A slightly more advanced example, a server echoing the request parameter 'name'.

```
local turbo = require("turbo")

local HelloNameHandler = class("HelloNameHandler", turbo.web.RequestHandler)

function HelloNameHandler:get()
    -- Get the 'name' argument, or use 'Santa Claus' if it does not exist
    local name = self:get_argument("name", "Santa Claus")
    self:write("Hello " .. name .. "!")
end

function HelloNameHandler:post()
    -- Get the 'name' argument, or use 'Easter Bunny' if it does not exist
    local name = self:get_argument("name", "Easter Bunny")
    self:write("Hello " .. name .. "!")
end

local app = turbo.web.Application:new({
    {"/hello", HelloNameHandler}
})

app:listen(8888)
turbo.ioloop.instance():start()
```

Routes

Turbo has a nice routing feature using Lua pattern matching. You can assign handler classes to routes in the turbo.web.Application constructor.

```
local turbo = require("turbo")

-- Handler that takes no argument, just like in the hello world example
local IndexHandler = class("IndexHandler", turbo.web.RequestHandler)
function IndexHandler:get()
    self:write("Index..")
end

-- Handler that takes a single argument 'username'
local UserHandler = class("UserHandler", turbo.web.RequestHandler)
function UserHandler:get(username)
    self:write("Username is " .. username)
end

-- Handler that takes two integers as arguments and adds them..
local AddHandler = class("AddHandler", turbo.web.RequestHandler)
function AddHandler:get(a1, a2)
    self:write("Result is: " .. tostring(a1+a2))
end

local app = turbo.web.Application:new({
    -- No arguments, will work for 'localhost:8888' and 'localhost:8888/'
    {"/$", IndexHandler},

    -- Use the part of the url after /user/ as the first argument to
    -- UserHandler:get
    {"/user/(.*)$", UserHandler},

    -- Find two int's separated by a '/' after /add in the url
    -- and pass them as arguments to AddHandler:get
    {"/add/(%d+)/(%d+)$", AddHandler}
})

app:listen(8888)
turbo.ioloop.instance():start()
```

Serving Static Files

It's often useful to be able to serve static assets, at least for development purposes. Turbo makes this really easy with the built in turbo.web.StaticFileHandler, just specify a directory, and it will do the heavy lifting, as well as cache your files for optimal performance.

```
local turbo = require("turbo")

app = turbo.web.Application:new({
    -- Serve static files from /var/www using the route "/static/(path-to-file)"
    {"/static/(.*)$", turbo.web.StaticFileHandler, "/var/www/"}
})

app:listen(8888)
turbo.ioloop.instance():start()
```

JSON Output

Turbo has implicit JSON coversion. This means that you can pass a JSON-serializable table to self:write and Turbo will set the 'Content-Type' header to 'application/json' and serialize the table for you.

```
local turbo = require("turbo")

-- Handler that responds with '{"hello":"json"}' and a Content-Type of application/
↪json
local HelloJSONHandler = class("HelloJSONHandler", turbo.web.RequestHandler)
function HelloJSONHandler:get()
    self:write({hello="json"})
end

local application = turbo.web.Application:new({
    {"/hello", HelloJSONHandler}
})

application:listen(8888)
turbo.ioloop.instance():start()
```

Asynchronous modules

Using native modules instead of generic Lua modules are important when using sockets to communicate with e.g databases to get the best performance, since the generic modules will block during long operations. If the operations you are doing are relatively fast then you may get away with using a generic module. This is something you have to benchmark yourself in your specific cases.

Creating modules for Turbo using the highly abstracted IOStream classes is real easy. If there is not a driver for e.g a database etc available then please do try to make it.

Overview

There are many ways of implementing async modules in Turbo.lua. To list the most apparent:

- Using callbacks
- Using Coroutine wrappable functions
- Using Coroutines and the CoroutineContext class

Which one suits your module best is up to your. Know that at the time of writing, no functions within the Lua Coroutine namespace is inlined by LuaJIT. Instead it fallbacks to the very fast interpreter. So if every little performance matters then callbacks are the way to go. All of the modules in the core framework uses Coroutine wrappable functions, except the HTTPClient, which uses coroutines and the CoroutineContext class.

Callback based modules is probably the flavour that most have used before. Basically you take in a callback (and maybe a callback argument/userdata) as argument(s) for the functions of your module. This function is then called when I/O has been completed. This means the user of your module must either split his program flow into seperate functions (seemingly in parts) or create closures inside functions.

Coroutine wrappable functions means that the functions of your API strictly adhers to the convention where the last two arguments of a functions always are a callback AND a callback argument (the argument is passed as first argument into the provided callback when it is called on I/O completion). If, and only if these requirements are met, the users of your module may use the `turbo.async.task` function to wrap the function and use the builtin Lua yield functionality. These functions then supports both callback-style and yield-style programming.

Coroutine directly with the CoroutineContext class does not offer a callback compatible API, and the users of the module must always yield to the I/O loop. This has some advantages in that it creates a 100% "I don't care about this async stuff" environment.

Programmatically this can be illustrated as follows:

"I don't care about this async stuff" module:

```
local turbo = require "turbo"
local turboredis = require "turbo-redis"
local rconn = turboredis.connect("127.0.0.1", 1337)

local ExampleHandler = class("ExampleHandler", turbo.web.RequestHandler)
function ExampleHandler:get()
    self:write("The value is " .. rconn:get("myvalue"))
end

local application = turbo.web.Application({
    {"^/$", ExampleHandler}
})
application:listen(8888)
turbo.ioloop.instance():start()
```

Callback-type module:

```
local turbo = require "turbo"
local turboredis = require "turbo-redis"
local rconn = turboredis.connect("127.0.0.1", 1337)

local ExampleHandler = class("ExampleHandler", turbo.web.RequestHandler)
function ExampleHandler:get()
    local _self = self
    rconn:get("myvalue", function(val)
        _self:write("The value is " .. rconn:get("myvalue"))
    end)
end

local application = turbo.web.Application({
    {"^/$", ExampleHandler}
})
application:listen(8888)
turbo.ioloop.instance():start()
```

Callback-type module with callback argument and no closures, probably well known for those familiar with Python and Tornado.

```
local turbo = require "turbo"
local turboredis = require "turbo-redis"
local rconn = turboredis.connect("127.0.0.1", 1337)

function ExampleHandler:_process_request(data)
    self:write("The value is " .. data)
end

local ExampleHandler = class("ExampleHandler", turbo.web.RequestHandler)
function ExampleHandler:get()
    rconn:get("myvalue", ExampleHandler._process_request, self)
end

```

```
local application = turbo.web.Application({
    {"^/$", ExampleHandler}
})
application:listen(8888)
turbo.ioloop.instance():start()
```

Coroutine wrappable Callback-type module:

```
local turbo = require "turbo"
local turboredis = require "turbo-redis"
local task = turbo.async.task
local yield = coroutine.yield

local rconn = turboredis("127.0.0.1", 1337)

local ExampleHandler = class("ExampleHandler", turbo.web.RequestHandler)
function ExampleHandler:get()
    self:write("The value is " .. yield(task(turboredis.get(rconn, "myvalue"))))
end

local application = turbo.web.Application({
    {"^/$", ExampleHandler}
})
application:listen(8888)
turbo.ioloop.instance():start()
```

The easiest to use is probably the first, where the program flow and code paths are more easily followed. The builtin HTTPClient uses this style of API... It is probably also a good choice for database queries etc, so you can keep your logic clean and easy to follow.

All callbacks added to the I/O loop are executed in its own coroutine. The callback functions can yield execution back to the I/O loop. Lua yield's can return a object as it return to where the coroutine where started... This is utilized in the Turbo.lua I/O loop which will treat yields different based on what they return as they yield. The I/O Loop supports these returns:

- A function, that will be called on next iteration and its results returned when resuming the coroutine thread.
- Nil, a empty yield that will simply resume the coroutine thread on next iteration.
- A CoroutineContext class, which acts as a reference to the I/O loop which allow the coroutine thread to be managed manually and resumed on demand.

Example module

So bearing this in mind let us create a CouchDB module.

We will create this one with a API that supports the business as usual programming style where the programmer does not yield or control this flow by himself. Note that this is in no way a complete and stable module, it is only meant to give some pointers:

```
local turbo = require "turbo"

-- Create a namespace to return.
local couch = {}

-- CouchDB class, it is obviously optional if you want to use object orientation or
↪not.
couch.CouchDB = class("CouchDB")
```

```
8
9
10  -- Init function to setup connection to CouchDB and more.
11  function couch.CouchDB:initialize(addr, ioloop)
12      assert(type(addr) == "string", "addr argument is not a valid address.")
13      self.ioloop = ioloop
14      local sock, msg = turbo.socket.new_nonblock_socket(
15          turbo.socket.AF_INET,
16          turbo.socket.SOCK_STREAM,
17          0)
18      if sock == -1 then
19          error("Could not create socket.")
20      end
21      self.sock = sock
22
23      self.iostream = turbo.iostream.IOStream(
24          self.sock,
25          self.io_loop,
26          1024*1024)
27
28      local hostname, port = unpack(util.strsplit(addr, ":"))
29      self.hostname = hostname
30      self.port = tonumber(port)
31      self.connected = false
32      self.busy = false
33      local _self = self
34
35      local rc, msg = self.iostream:connect(
36          self.hostname,
37          self.port,
38          turbo.socket.AF_INET,
39          function()
40              -- Set a connected flag, so that we know we can process requests.
41              _self.connected = true
42              turbo.log.success("Couch Connected!") end,
43          function() turbo.log.error("Could not connect to CouchDB!") end)
44      if rc ~= 0 then
45          error("Host not reachable. " .. msg or "")
46      end
47      self.iostream:set_close_callback(function()
48          _self.connected = false
49          turbo.log.error("CouchDB disconnected!")
50          -- Add reconnect code here.
51      end)
52  end
53
54  function couch.CouchDB:get(resource)
55      assert(self.connected, "No connection to CouchDB, can not process request.")
56      assert(not self.busy, "Connection is busy, try again later.")
57      self.busy = true
58
59      self.headers = turbo.httputil.HTTPHeaders()
60      self.headers:add("Host", self.hostname)
61      self.headers:add("User-Agent", "Turbo Couch")
62      self.headers:set_method("GET")
63      self.headers:set_version("HTTP/1.1")
64      self.headers:set_uri(resource)
65      local buf = self.headers:stringify_as_request()
```

```
    -- Write request HTTP header to stream and wait for finish using the simple way with turbo.async.task wrapper
    -- function.
    coroutine.yield (turbo.async.task(self.iostream.write, self.iostream, buf))

    -- Wait until end of HTTP response header has been read.
    local res = coroutine.yield (turbo.async.task(self.iostream.read_until_pattern, self.iostream, "\r?\n\r?\n"))

    -- Decode response header.
    local response_headers = turbo.httputil.HTTPParser(res, turbo.httputil.hdr_t["HTTP_RESPONSE"])

    -- Read the actual body now that we know the size of body.
    local body = coroutine.yield (turbo.async.task(
            self.iostream.read_bytes,
            self.iostream,
            tonumber((response_headers:get("Content-Length")))))

    -- Decode JSON response body and return it to caller.
    local json_dec = turbo.escape.json_decode(body)
    return json_dec
end

--- Add more methods :)

return couch
```

Usage from a turbo.web.RequestHandler:

```
local turbo = require "turbo"
local couch = require "turbo-couch"

-- Create a instance.
local cdb = couch.CouchDB("localhost:5984")

local ExampleHandler = class("ExampleHandler", turbo.web.RequestHandler)
function ExampleHandler:get()
    -- Write response directly through.
    self:write(cdb:get("/test/toms_resource"))
end

turbo.web.Application({{"^/$", ExampleHandler}}):listen(8888)
turbo.ioloop.instance():start()
```

CHAPTER 10

API documentation

Turbo.lua API Versioning

Preliminaries

All modules are required in turbo.lua, so it's enough to

```
local turbo = require('turbo')
```

All functionality is placed in the "turbo" namespace.

Module Version

The Turbo Web version is of the form *A.B.C*, where *A* is the major version, *B* is the minor version, and *C* is the micro version. If the micro version is zero, it's omitted from the version string.

When a new release only fixes bugs and doesn't add new features or functionality, the micro version is incremented. When new features are added in a backwards compatible way, the minor version is incremented and the micro version is set to zero. When there are backwards incompatible changes, the major version is incremented and others are set to zero.

The following constants specify the current version of the module:

turbo.MAJOR_VERSION, turbo.MINOR_VERSION, turbo.MICRO_VERSION Numbers specifiying the major, minor and micro versions respectively.

turbo.VERSION A string representation of the current version, e.g `"1.0.0"` or `"1.1.0"`.

turbo.VERSION_HEX A 3-byte hexadecimal representation of the version, e.g. `0x010201` for version 1.2.1 and `0x010300` for version 1.3.

turbo.web – Core web framework

The Turbo.lua Web framework is modeled after the framework offered by Tornado (http://www.tornadoweb.org/), which again is based on web.py (http://webpy.org/) and Google's webapp (http://code.google.com/appengine/docs/python/tools/webapp/) Some modifications has been made to make it fit better into the Lua eco system. The web framework utilizes asynchronous features that allow it to scale to large numbers of open connections (thousands). The framework support comet polling.

Create a web server that listens to port 8888 and prints the canoncial Hello world on a GET request is very easy:

```
local turbo = require('turbo')

local ExampleHandler = class("ExampleHandler", turbo.web.RequestHandler)
function ExampleHandler:get()
        self:write("Hello world!")
end

local application = turbo.web.Application({
        {"^/$", ExampleHandler}
})
application:listen(8888)
turbo.ioloop.instance():start()
```

RequestHandler class

Base RequestHandler class. The heart of Turbo.lua. The usual flow of using Turbo.lua is sub-classing the RequestHandler class and implementing the HTTP request methods described in self.SUPPORTED_METHODS. The main goal of this class is to wrap a HTTP request and offer utilities to respond to the request. Requests are deligated to RequestHandler's by the Application class. The RequestHandler class are implemented so that it should be subclassed to process HTTP requests.

It is possible to modify self.SUPPORT_METHODS to add support for more methods if that is wanted.

Entry points

RequestHandler:on_create(kwargs)
Redefine this method if you want to do something straight after the class has been initialized. This is called after a request has been received, and before the HTTP method has been verified against supported methods. So if a not supported method is requested, this method is still called.

Parameters **kwargs** (*Table*) – The keyword arguments that you initialize the class with.

RequestHandler:prepare()
Redefine this method if you want to do something after the class has been initialized. This method unlike on_create, is only called if the method has been found to be supported.

RequestHandler:on_finish()
Called after the end of a request. Useful for e.g a cleanup routine.

RequestHandler:set_default_headers()
Redefine this method to set HTTP headers at the beginning of all the request received by the RequestHandler. For example setting some kind of cookie or adjusting the Server key in the headers would be sensible to do in this method.

Subclass RequestHandler and implement any of the following methods to handle the corresponding HTTP request. If not implemented they will provide a 405 (Method Not Allowed). These methods receive variable arguments, depending

on what the Application instance calling them has captured from the pattern matching of the request URL. The methods are run protected, so they are error safe. When a error occurs in the execution of these methods the request is given a 500 Internal Server Error response. In debug mode, the stack trace leading to the crash is also a part of the response. If not debug mode is set, then only the status code is set.

RequestHandler:get(...)
HTTP GET reqests handler.

Parameters **...** – Parameters from matched URL pattern with braces. E.g /users/(.*)$ would provide anything after /users/ as first parameter.

RequestHandler:post(...)
HTTP POST reqests handler.

Parameters **...** – Parameters from matched URL pattern with braces.

RequestHandler:head(...)
HTTP HEAD reqests handler.

Parameters **...** – Parameters from matched URL pattern with braces.

RequestHandler:delete(...)
HTTP DELETE reqests handler.

Parameters **...** – Parameters from matched URL pattern with braces.

RequestHandler:put(...)
HTTP PUT reqests handler.

Parameters **...** – Parameters from matched URL pattern with braces.

RequestHandler:options(...)
HTTP OPTIONS reqests handler.

Parameters **...** – Parameters from matched URL pattern with braces.

Input

RequestHandler:get_argument(name, default, strip)
Returns the value of the argument with the given name. If default value is not given the argument is considered to be required and will result in a raise of a HTTPError 400 Bad Request if the argument does not exist.

Parameters

- **name** (*String*) – Name of the argument to get.
- **default** (*String*) – Optional fallback value in case argument is not set.
- **strip** (*Boolean*) – Remove whitespace from head and tail of string.

Return type String

RequestHandler:get_arguments(name, strip)
Returns the values of the argument with the given name. Should be used when you expect multiple arguments values with same name. Strip will take away whitespaces at head and tail where applicable. Returns a empty table if argument does not exist.

Parameters

- **name** (*String*) – Name of the argument to get.
- **strip** (*Boolean*) – Remove whitespace from head and tail of string.

Return type Table

RequestHandler:get_json(force)
Returns JSON request data as a table. By default, it only parses request with "application/json" as content-type header.

Parameters **force** (*Boolean*) – If force is set to true, all request will be parsed regardless of content-type header.

Return type Table or nil

RequestHandler:get_cookie(name, default)
Get cookie value from incoming request.

Parameters

- **name** (*String*) – The name of the cookie to get.
- **default** (*String*) – A default value if no cookie is found.

Return type String or nil if not found

RequestHandler:get_secure_cookie(name, default, max_age)
Get a signed cookie value from incoming request.

If the cookie can not be validated, then an error with a string error is raised.

Hash-based message authentication code (HMAC) is used to be able to verify that the cookie has been created with the "cookie_secret" set in the Application class kwargs. This is simply verifing that the cookie has been signed by your key, IT IS NOT ENCRYPTING DATA.

Parameters

- **name** (*String*) – The name of the cookie to get.
- **default** (*String*) – A default value if no cookie is found.
- **max_age** (*Number*) – Timestamp used to sign cookie must be not be older than this value in seconds.

Return type String or nil if not found

RequestHandler:set_cookie(name, value, domain, expire_hours)
Set a cookie with value to response.

Note: Expiring relies on the requesting browser and may or may not be respected. Also keep in mind that the servers time is used to calculate expiry date, so the server should ideally be set up with NTP server.

Parameters

- **name** (*String*) – The name of the cookie to set.
- **value** (*String*) – The value of the cookie:
- **domain** (*String*) – The domain to apply cookie for.
- **expire_hours** (*Number*) – Set cookie to expire in given amount of hours.

RequestHandler:set_secure_cookie(name, value, domain, expire_hours)
Set a signed cookie value to response.

Hash-based message authentication code (HMAC) is used to be able to verify that the cookie has been created with the "cookie_secret" set in the Application class kwargs. This is simply verifing that the cookie has been signed by your key, IT IS NOT ENCRYPTING DATA.

Note: Expiring relies on the requesting browser and may or may not be respected. Also keep in mind that the servers time is used to calculate expiry date, so the server should ideally be set up with NTP server.

Parameters

- **name** (*String*) – The name of the cookie to set.
- **value** (*String*) – The value of the cookie:
- **domain** (*String*) – The domain to apply cookie for.
- **expire_hours** (*Number*) – Set cookie to expire in given amount of hours.

RequestHandler.request `turbo.httpserver.HTTPRequest` class instance for this request. This object contains e.g `turbo.httputil.HTTPHeader` and the body payload etc. See the documentation for the classes for more details.

Output

RequestHandler:write(chunk)
Writes the given chunk to the output buffer. To write the output to the network, call the `turbo.web.RequestHandler:flush()` method. If the given chunk is a Lua table, it will be automatically stringifed to JSON.

Parameters **chunk** (*String*) – Data chunk to write to underlying connection.

RequestHandler:finish(chunk)
Finishes the HTTP request. This method can only be called once for each request. This method flushes all data in the write buffer.

Parameters **chunk** (*String*) – Final data to write to stream before finishing.

RequestHandler:flush(callback)
Flushes the current output buffer to the IO stream.

If callback is given it will be run when the buffer has been written to the socket. Note that only one callback flush callback can be present per request. Giving a new callback before the pending has been run leads to discarding of the current pending callback. For HEAD method request the chunk is ignored and only headers are written to the socket.

Parameters **callback** (*Function*) – Function to call after the buffer has been flushed.

RequestHandler:clear()
Reset all headers, empty write buffer in a request.

RequestHandler:add_header(name, value)
Add the given name and value pair to the HTTP response headers.

Parameters

- **name** (*String*) – Key string for header field.
- **value** (*String*) – Value for header field.

RequestHandler:set_header(name, value)
Set the given name and value pair of the HTTP response headers. If name exists then the value is overwritten.

Parameters

- **name** (*String*) – Key string for header field.
- **value** (*String*) – Value for header field.

RequestHandler:get_header(name)
Returns the current value of the given name in the HTTP response headers. Returns nil if not set.

Parameters **name** (*String*) – Name of value to get.

Return type String or nil

RequestHandler:set_status(code)
Set the status code of the HTTP response headers. Must be number or error is raised.

Parameters **code** (*Number*) – HTTP status code to set.

RequestHandler:get_status()
Get the curent status code of the HTTP response headers.

Return type Number

RequestHandler:redirect(url, permanent)
Redirect client to another URL. Sets headers and finish request. User can not send data after this.

Parameters

- **url** (*String*) – The URL to redirect to.
- **permanent** (*Boolean*) – Flag this as a permanent redirect or temporary.

Misc

RequestHandler:set_async(bool)
Set handler to not call finish() when request method has been called and returned. Default is false. When set to true, the user must explicitly call finish.

Parameters **bool** (*Boolean*) –

HTTPError class

This error is raisable from RequestHandler instances. It provides a convinent and safe way to handle errors in handlers. E.g it is allowed to do this:

```
function MyHandler:get()
    local item = self:get_argument("item")
     if not find_in_store(item) then
         error(turbo.web.HTTPError(400, "Could not find item in store"))
     end
     ...
end
```

The result is that the status code is set to 400 and the message is sent as the body of the request. The request is always finished on error.

HTTPError(*code*, *message*)
Create a new HTTPError class instance.

Parameters

- **code** (*Number*) – The HTTP status code to send to send to client.
- **message** (*String*) – Optional message to pass as body in the response.

StaticFileHandler class

A static file handler for files on the local file system. All files below user defined _G.TURBO_STATIC_MAX or default 1MB in size are stored in memory after initial request. Files larger than this are read from disk on demand. If TURBO_STATIC_MAX is set to -1 then cache is disabled.

Usage:

```
local app = turbo.web.Application:new({
        {"^/$", turbo.web.StaticFileHandler, "/var/www/index.html"},
        {"^/(.*)$", turbo.web.StaticFileHandler, "/var/www/"}
})
```

Paths are not checked until intial hit on handler. The file is then cached in memory if it is a valid path. Notice that paths postfixed with / indicates that it should be treated as a directory. Paths with no / is treated as a single file.

RedirectHandler class

A simple redirect handler that simply redirects the client to the given URL in 3rd argument of a entry in the Application class's routing table.

```
local application = turbo.web.Application({
    {"^/redirector$", turbo.web.RedirectHandler, "http://turbolua.org"}
})
```

Application class

The Application class is a collection of request handler classes that make together up a web application. Example:

```
1     local application = turbo.web.Application({
2             {"^/static/(.*)$", turbo.web.StaticFileHandler, "/var/www/"},
3             {"^/$", ExampleHandler},
4             {"^/item/(%d*)", ItemHandler}
5     })
```

The constructor of this class takes a "map" of URL patterns and their respective handlers. The third element in the table are optional parameters the handler class might have. E.g the `turbo.web.StaticFileHandler` class takes the root path for your static handler. This element could also be another table for multiple arguments.

The first element in the table is the URL that the application class matches incoming request with to determine how to serve it. These URLs simply be a URL or a any kind of Lua pattern.

The ItemHandler URL pattern is an example on how to map numbers from URL to your handlers. Pattern encased in parantheses are used as parameters when calling the request methods in your handlers.

Note: Patterns are matched in a sequential manner. If a request matches multiple handler pattern's only the first handler matched is delegated the request. Therefore, it is important to write good patterns.

A good read on Lua patterns matching can be found here: http://www.wowwiki.com/Pattern_matching.

Application (*handlers*, *kwargs*)
Initialize a new Application class instance.

Parameters

- **handlers** (*Table*) – As described above. Table of tables with pattern to handler binding.
- **kwargs** (*Table*) – Keyword arguments

Keyword arguments supported:

•"default_host" (String) - Redirect to this URL if no matching handler is found.

•"cookie_secret" (String) - Sequence of bytes used to sign secure cookies.

Application:add_handler(pattern, handler, arg)
Add handler to Application.

Parameters

- **pattern** (*String*) – Lua pattern string.
- **handler** (*RequestHandler based class*) –
- **arg** – Argument for handler.

Application:listen(port, address, kwargs)
Starts an HTTP server for this application on the given port. This is really just a convinence method. The same effect can be achieved by creating a `turbo.httpserver.HTTPServer` class instance and assigning the Application instance to its request_callback parameter and calling its listen() method.

Parameters

- **port** (*Number*) – Port to bind server to.
- **address** (*String or number.*) – Address to bind server to. E.g "127.0.0.1".
- **kwargs** (*Table*) – Keyword arguments passed on to `turbo.httpserver.HTTPServer`. See documentation for available options. This is used to set SSL certificates amongst other things.

Application:set_server_name(name)
Sets the name of the server. Used in the response headers.

Parameters **name** (*String*) – The name used in HTTP responses. Default is "Turbo vx.x"

Application:get_server_name()
Gets the current name of the server. :rtype: String

Mustache Templating

Turbo.lua has a small and very fast Mustache parser built-in. Mustache templates are logic-less templates, which are supposed to help you keep your business logic outside of templates and inside "controllers". It is widely known by Javascript developers and very simple to understand.

For more information on the Mustache markup, please see this: http://mustache.github.io/mustache.5.html

Mustache.**compile**(*template*)
Compile a Mustache highlighted string into its intermediate state before rendering. This function does some validation on the template. If it finds syntax errors a error with a message is raised. It is always a good idea to cache pre-compiled frequently used templates before rendering them. Although compiling each time is usally not a big overhead. This function can throw errors if the template contains invalid logic.

Parameters **template** – (String) Template in string form.

Return type Parse table that can be used for Mustache.render function

Mustache.**render**(*template*, *obj*, *partials*, *allow_blank*)
Render a template. Accepts a parse table compiled by Mustache.compile or a uncompiled string. Obj is the table with keys. This function can throw errors in case of un-compiled string being compiled with Mustache.compile internally.

Parameters

- **template** – Accepts a pre-compiled template or a un-compiled string.
- **obj** (*Table*) – Parameters for template rendering.
- **partials** (*Table*) – Partial snippets. Will be treated as static and not compiled...
- **allow_blank** – Halt with error if key does not exist in object table.

Example templating:

```
<body>
    <h1>
            {{heading }}
    </h1>
    {{!
        Some comment section that
        even spans across multiple lines,
        that I just have to have to explain my flawless code.
    }}
    <h2>
        {{{desc}}} {{! No escape with triple mustaches allow HTML tags! }}
        {{&desc}} {{! No escape can also be accomplished by & char }}
    </h2>
    <p>I am {{age}} years old. What would you like to buy in my shop?</p>
    {{  #items }}  {{! I like spaces alot!          }}
        Item: {{item}}
        {{#types}}
                {{! Only print items if available.}}
                Type available: {{type}}
        {{/types}}
        {{^types}}  Only one type available.
        {{! Apparently only one type is available because types is not set,
        determined by the hat char ^}}
        {{/types}}
    {{/items}}
    {{^items}}
            No items available!
    {{/items}}
</body>
```

With a render table likes this:

```
{
    heading="My website!",
    desc="<b>Big important website</b>",
    age=27,
    items={
              {item="Bread",
                  types={
                      {type="light"},
                      {type="fatty"}
                  }
          },
          {item="Milk"},
          {item="Sugar"}
   }
}
```

Will produce this output after rendering:

```
<body>
    <h1>
        My%20website%21
    </h1>
    <h2>
        <b>Big important website</b>
        <b>Big important website</b>
```

```
    </h2>
    <p>I am 27 years old. What would you like to buy in my shop?</p>
        Item: Bread
            Type available: light
            Type available: fatty
        Item: Milk
            Only one type available.
        Item: Sugar
            Only one type available.
</body>
```

Mustache.TemplateHelper class

A simple class that simplifies the loading of Mustache templates, pre-compile and cache them for future use.

`Mustache.`**`TemplateHelper`**`(`*base_path*`)`
Create a new TemplateHelper class instance.

Parameters **`base_path`** (*`String`*) – Template base directory. This will be used as base path when loading templates using the load method.

Return type `Mustache.TemplateHelper class`

`Mustache.TemplateHelper:load(template)`
Pre-load a template.

Parameters **`template`** (*`String`*) – Template name, e.g file name in base directory.

`Mustache.TemplateHelper:render(template, table, partials, allow_blank)`
Render a template by name. If called twice the template will be cached from first time.

Parameters

- **`template`** (*`String`*) – Template name, e.g file name in base directory.
- **`obj`** (*`Table`*) – Parameters for template rendering.
- **`partials`** (*`Table`*) – Partial snippets. Will be treated as static and not compiled...
- **`allow_blank`** – Halt with error if key does not exist in object table.

turbo.websocket – WebSocket server and client

The WebSocket modules extends Turbo and offers RFC 6455 compliant WebSocket support.

The module offers two classes:

- `turbo.websocket.WebSocketHandler`, WebSocket support for `turbo.web.Application`.
- `turbo.websocket.WebSocketClient`, callback based WebSocket client.

Both classes uses the mixin class `turbo.websocket.WebSocketStream`, which in turn provides almost identical API's for the two classes once connected. Both classes support SSL (wss://).

NOTICE: _G.TURBO_SSL MUST be set to true and OpenSSL or axTLS MUST be installed to use this module as certain hash functions are required by the WebSocket protocol.

A simple example subclassing `turbo.websocket.WebSocketHandler`.:

```
_G.TURBO_SSL = true
local turbo = require "turbo"

local WSExHandler = class("WSExHandler", turbo.websocket.WebSocketHandler)

function WSExHandler:on_message(msg)
    self:write_message("Hello World.")
end

turbo.web.Application({{"^/ws$", WSExHandler}}):listen(8888)
turbo.ioloop.instance():start()
```

WebSocketStream mixin

WebSocketStream is a abstraction for a WebSocket connection, used as class mixin in `turbo.websocket.WebSocketHandler` and `turbo.websocket.WebSocketClient`.

`WebSocketStream:write_message(msg, binary)`
Send a message to the client of the active Websocket. If the stream has been closed a error is raised.

Parameters

- **`msg`** (*`String`*) – The message to send. This may be either a JSON-serializable table or a string.
- **`binary`** (*`Boolean`*) – Treat the message as binary data (use WebSocket binary opcode).

`WebSocketStream:ping(data, callback, callback_arg)`
Send a ping to the connected client.

Parameters

- **`data`** (*`String`*) – Data to pong back.
- **`callback`** (*`Function`*) – Function to call when pong is received.
- **`callback_arg`** – Argument for callback function.

`WebSocketStream:pong(data)`
Send a pong to the connected server.

Parameters **`data`** (*`String`*) – Data to pong back.

`WebSocketStream:close()`
Close the connection.

`WebSocketStream:closed()`
Has the stream been closed?

WebSocketHandler class

The WebSocketHandler is a subclass of `turbo.web.RequestHandler`. So most of the methods and attributes of that class are available. However, some of them will not work with WebSocket's. It also have the mixin class `turbo.websocket.WebSocketStream` included. Only the official WebSocket specification, RFC 6455, is supported.

For a more thorough example of usage of this class you can review the "chatapp" example bundled with Turbo, which uses most of its features to create a simple Web-based chat app.

Subclass WebSocketHandler and implement any of the following methods to handle the corresponding events.

WebSocketHandler:open()
Called when a new WebSocket request is opened.

WebSocketHandler:on_message(msg)
Called when a message is received.

Parameters **msg** (*String*) – The received message.

WebSocketHandler:on_close()
Called when the connection is closed.

WebSocketHandler:on_error(msg)

Parameters **msg** (*String*) – A error string.

WebSocketHandler:prepare()
Called when the headers has been parsed and the server is about to initiate the WebSocket specific handshake. Use this to e.g check if the headers Origin field matches what you expect. To abort the connection you raise a error. `turbo.web.HTTPError` is the most convinient as you can set error code and a message returned to the client.

WebSocketHandler:subprotocol(protocols)
Called if the client have included a Sec-WebSocket-Protocol field in header. This method will then receive a table of protocols that the clients wants to use. If this field is not set, this method will never be called. The return value of this method should be a string which matches one of the suggested protcols in its parameter. If all of the suggested protocols are unacceptable then dismissing of the request is done by either raising error (such as `turbo.web.HTTPError`) or returning nil.

Parameters **protocols** (*Table of protocol name strings.*) – The protocol names received from client.

WebSocketClient class

A async callback based WebSocket client. Only the official WebSocket specification, RFC 6455, is supported. The WebSocketClient is partly based on the `turbo.async.HTTPClient` using its HTTP implementation to do the initial connect to the server, then do the handshake and finally wrapping the connection with the `turbo.websocket.WebSocketStream`. All of the callback functions receives the class instance as first argument for convinence. Furthermore the class can be initialized with keyword arguments that are passed on to the `turbo.async.HTTPClient` that are being used. So if you are going to use the connect to a SSL enabled server (wss://) then you simply refer to the documentation of the HTTPClient and set "priv_file", "cert_file" keys properly. Some arguments are discared though, such as e.g "method".

A simple usage example of `turbo.websocket.WebSocketClient`.:

```
_G.TURBO_SSL = true -- SSL must be enabled for WebSocket support!
local turbo = require "turbo"

turbo.ioloop.instance():add_callback(function()
    turbo.websocket.WebSocketClient("ws://127.0.0.1:8888/ws", {
        on_headers = function(self, headers)
            -- Review headers received from the WebSocket server.
            -- You can e.g drop the request if the response headers
            -- are not satisfactory with self:close().
        end,
        modify_headers = function(self, headers)
            -- Modify outgoing headers before they are sent.
            -- headers parameter are a instance of httputil.HTTPHeader.
        end,
        on_connect = function(self)
```

```
            -- Called when the client has successfully opened a WebSocket
            -- connection to the server.
            -- When the connection is established you can write a message:
            self:write_message("Hello World!")
        end,
        on_message = function(self, msg)
            -- Print the incoming message.
            print(msg)
            self:close()
        end,
        on_close = function(self)
            -- I am called when connection is closed. Both gracefully and
            -- not gracefully.
        end,
        on_error = function(self, code, reason)
            -- I am called whenever there is a error with the WebSocket.
            -- code are defined in ``turbo.websocket.errors``. reason are
            -- a string representation of the same error.
        end
    })
end):start()
```

WebSocketClient uses error codes to report failure for the `on_error` callback.

errors

Numeric error codes set as first argument of `on_error`:

`INVALID_URL` - URL could not be parsed.

`INVALID_SCHEMA` - Invalid URL schema

`COULD_NOT_CONNECT` - Could not connect, check message.

`PARSE_ERROR_HEADERS` - Could not parse response headers.

`CONNECT_TIMEOUT` - Connect timed out.

`REQUEST_TIMEOUT` - Request timed out.

`NO_HEADERS` - Shouldn't happen.

`REQUIRES_BODY` - Expected a HTTP body, but none set.

`INVALID_BODY` - Request body is not a string.

`SOCKET_ERROR` - Socket error, check message.

`SSL_ERROR` - SSL error, check message.

`BUSY` - Operation in progress.

`REDIRECT_MAX` - Redirect maximum reached.

`CALLBACK_ERROR` - Error in callback.

`BAD_HTTP_STATUS` - Did not receive expected 101 Upgrade.

`WEBSOCKET_PROTOCOL_ERROR` - Invalid WebSocket protocol data received.

WebSocketClient(address, kwargs):

Create a new WebSocketClient class instance.

Parameters

- **address** (*String*) – URL for WebSocket server to connect to.

- **kwargs** (*Table*) – Optional keyword arguments.

Return type Instance of `turbo.websocket.WebSocketClient`

Available keyword arguments:

- `params` - Provide parameters as table.
- `cookie` - The cookie to use.
- `allow_redirects` - Allow or disallow redirects. Default is true.
- `max_redirects` - Maximum redirections allowed. Default is 4.
- `body` - Request HTTP body in plain form.
- `request_timeout` - Total timeout in seconds (including connect) for request. Default is 60 seconds. After the connection has been established the timeout is removed.
- `connect_timeout` - Timeout in seconds for connect. Default is 20 secs.
- `auth_username` - Basic Auth user name.
- `auth_password` - Basic Auth password.
- `user_agent` - User Agent string used in request headers. Default is `Turbo Client vx.x.x`.
- `priv_file` - Path to SSL / HTTPS private key file.
- `cert_file` - Path to SSL / HTTPS certificate key file.
- `ca_path` - Path to SSL / HTTPS CA certificate verify location, if not given builtin is used, which is copied from Ubuntu 12.10.
- `verify_ca` - SSL / HTTPS verify servers certificate. Default is true.

Description of the callback functions

modify_headers (*self*, *headers*)

Modify OUTGOING HTTP headers before they are sent to the server.

Parameters

- **self** (*turbo.websocket.WebSocketClient*) – The WebSocketClient instance calling the callback.
- **headers** (`turbo.httputil.HTTPHeader`) – Headers ready to be sent and possibly modified.

on_headers (*self*, *headers*)

Review HTTP headers received from the WebSocket server. You can e.g drop the request if the response headers are not satisfactory with self:close().

Parameters

- **self** (*turbo.websocket.WebSocketClient*) – The WebSocketClient instance calling the callback.
- **headers** (`turbo.httputil.HTTPHeader`) – Headers received from the client.

on_connect (*self*)

Called when the client has successfully opened a WebSocket connection to the server.

Parameters **self** (*turbo.websocket.WebSocketClient*) – The WebSocketClient instance calling the callback.

on_message (*self*, *msg*)
Called when a message is received.

Parameters

- **self** (*turbo.websocket.WebSocketClient*) – The WebSocketClient instance calling the callback.
- **msg** (*String*) – The message or binary data.

on_close (*self*)
Called when connection is closed. Both gracefully and not gracefully.

Parameters **self** (*turbo.websocket.WebSocketClient*) – The WebSocketClient instance calling the callback.

on_error (*self*, *code*, *reason*)
Called whenever there is a error with the WebSocket. code are defined in `turbo.websocket.errors`. reason are a string representation of the same error.

Parameters

- **self** (*turbo.websocket.WebSocketClient*) – The WebSocketClient instance calling the callback.
- **code** (*Number*) – Error code defined in `turbo.websocket.errors`.
- **reason** (*String*) – String representation of error.

on_ping (*self*, *data*)
Called when a ping request is received.

Parameters

- **self** (*turbo.websocket.WebSocketClient*) – The WebSocketClient instance calling the callback.
- **data** (*String*) – The ping payload data.

turbo.iosimple – Simple Callback-less asynchronous sockets

A simple interface for the IOStream class without the callback spaghetti, but still the async backend (the yield is done internally):

```
turbo.ioloop.instance():add_callback(function()
    local stream = turbo.iosimple.dial("tcp://turbolua.org:80")
    stream:write("GET / HTTP/1.0\r\n\r\n")

    local data = stream:read_until_close()
    print(data)

    turbo.ioloop.instance():close()
end):start()
```

iosimple.**dial** (*address*, *ssl*, *io*)
Connect to a host using a simple URL pattern.

Parameters

- **address** (*String*) – The address to connect to in URL form, e.g: `"tcp://turbolua.org:80"`.

- **ssl** (*Boolean or Table*) – Option to connect with SSL. Takes either a boolean true or a table with options, options described below.
- **io** (*IOLoop object*) – IOLoop class instance to use for event processing. If none is set then the global instance is used, see the ioloop.instance() function.

Available SSL options:

Boolean, if ssl param is set to `true`, it will be equal to a SSL option table like this `{verify=true}`. If not argument is given or nil then SSL will not be used at all.

A table may be used to give additional options instead of just a "enable" button:

- `key_file` (String) - Path to SSL / HTTPS key file.
- `cert_file` (String) - Path to SSL / HTTPS certificate file.
- `ca_cert_file` (String) - Path to SSL / HTTPS CA certificate verify location, if not given builtin is used, which is copied from Ubuntu 12.10.
- `verify` (Boolean) SSL / HTTPS verify servers certificate. Default is true.

IOSimple class

A alternative to the IOStream class that were added in version 2.0. The goal of this class is to further simplify the way that we use Turbo. The IOStream class is based on callbacks, and while this to some may be the optimum way it might not be for others. You could always use the `async.task()` function to wrap it in a coroutine and yield it. To save you the hassle a new class has been made.

All functions may raise errors. All functions yield to the IOLoop internally. You may catch errors with xpcall or pcall.

IOSimple(*stream*)

Wrap a IOStream class instance with a simpler IO. If you are not wrapping consider using `iosimple.dial()`.

Parameters **stream** (`IOStream object`) – A stream already connected. If not consider using `iosimple.dial()`.

Return type `IOSimple object`

IOSimple:read_until(delimiter)

Read until delimiter. Delimiter is plain text, and does not support Lua patterns. See read_until_pattern for that functionality. read_until should be used instead of read_until_pattern wherever possible because of the overhead of doing pattern matching.

Parameters **delimiter** (*String*) – Delimiter sequence, text or binary.

Return type String

IOSimple:read_until_pattern(pattern)

Read until pattern is matched, then return with received data. If you only are doing plain text matching then using read_until is recommended for less overhead.

Parameters **pattern** (*String*) – Lua pattern string.

Return type String

IOSimple:read_bytes(num_bytes)

Read the given number of bytes.

Parameters **num_bytes** (*Number*) – The amount of bytes to read.

Return type String

IOSimple:read_until_close()
Reads all data from the socket until it is closed.

Return type String

IOSimple:write(data)
Write the given data to this stream. Returns when the data has been written to socket.

IOSimple:close()
Close this stream and its socket.

IOSimple:get_iostream()
Returns the IOStream instance used by the IOSimple instance.

Return type `IOStream object`

turbo.iostream – Callback based asynchronous streaming sockets

The turbo.iostream namespace contains the IOStream and SSLIOStream classes, which are abstractions to provide easy to use streaming sockets. All API's are callback based and depend on the `turbo.ioloop.IOLoop` class.

IOStream class

The IOStream class is implemented through the use of the IOLoop class, and are utilized e.g in the RequestHandler class and its subclasses. They provide a non-blocking interface and support callbacks for most of its operations. For read operations the class supports methods suchs as read until delimiter, read n bytes and read until close. The class has its own write buffer and there is no need to buffer data at any other level. The default maximum write buffer is defined to 100 MB. This can be defined on class initialization.

IOStream (*fd*, *io_loop*, *max_buffer_size*, *kwargs*)
Create a new IOStream instance.

Parameters

- **fd** (*Number*) – File descriptor, either open or closed. If closed then, the `turbo.iostream.IOStream:connect()` method can be used to connect.
- **io_loop** (*IOLoop object*) – IOLoop class instance to use for event processing. If none is set then the global instance is used, see the `ioloop.instance()` function.
- **max_buffer_size** (*Number*) – The maximum number of bytes that can be held in internal buffer before flushing must occur. If none is set, 104857600 are used as default.
- **kwargs** (*Table*) – Keyword arguments

Return type IOStream object

Available keyword arguments:

•`dns_timeout` - (Number) Timeout for DNS lookup on connect.

IOStream:connect(address, port, family, callback, fail_callback, arg)
Connect to a address without blocking. To successfully use this method it is neccessary to use a success and a fail callback function to properly handle both cases.

Parameters

- **host** (*String*) – The host to connect to. Either hostname or IP.
- **port** (*Number*) – The port to connect to. E.g 80.

- **family** – Socket family. Optional. Pass nil to guess.
- **callback** (*Function*) – Optional callback for "on successfull connect"
- **fail_callback** (*Function*) – Optional callback for "on error". Called with errno and its string representation as arguments.
- **arg** – Optional argument for callback. callback and fail_callback are called with this as first argument.

IOStream:read_until(delimiter, callback, arg)

Read until delimiter, then call callback with received data. The callback receives the data read as a parameter. Delimiter is plain text, and does not support Lua patterns. See read_until_pattern for that functionality. read_until should be used instead of read_until_pattern wherever possible because of the overhead of doing pattern matching.

Parameters

- **delimiter** (*String*) – Delimiter sequence, text or binary.
- **callback** (*Function*) – Callback function. The function is called with the received data as parameter.
- **arg** – Optional argument for callback. If arg is given then it will be the first argument for the callback and the data will be the second.

IOStream:read_until_pattern(pattern, callback, arg)

Read until pattern is matched, then call callback with received data. The callback receives the data read as a parameter. If you only are doing plain text matching then using read_until is recommended for less overhead.

Parameters

- **pattern** (*String*) – Lua pattern string.
- **callback** (*Function*) – Callback function. The function is called with the received data as parameter.
- **arg** – Optional argument for callback. If arg is given then it will be the first argument for the callback and the data will be the second.

IOStream:read_bytes(num_bytes, callback, arg, streaming_callback, streaming_arg)

Call callback when we read the given number of bytes. If a streaming_callback argument is given, it will be called with chunks of data as they become available, and the argument to the final call to callback will be empty.

Parameters

- **num_bytes** (*Number*) – The amount of bytes to read.
- **callback** (*Function*) – Callback function. The function is called with the received data as parameter.
- **arg** – Optional argument for callback. If arg is given then it will be the first argument for the callback and the data will be the second.
- **streaming_callback** (*Function*) – Optional callback to be called as chunks become available.
- **streaming_arg** – Optional argument for callback. If arg is given then it will be the first argument for the callback and the data will be the second.

IOStream:read_until_close(callback, arg, streaming_callback, streaming_arg)

Reads all data from the socket until it is closed. If a streaming_callback argument is given, it will be called with chunks of data as they become available, and the argument to the final call to callback will contain the final chunk. This method respects the max_buffer_size set in the IOStream object.

Parameters

- **callback** (*Function with one parameter or nil.*) – Function to call when connection has been closed.
- **arg** – Optional argument for callback. If arg is given then it will be the first argument for the callback and the data will be the second.
- **streaming_callback** – Function to call as chunks become available.
- **streaming_arg** – Optional argument for callback. If arg is given then it will be the first argument for the callback and the data will be the second.

IOStream:write(data, callback, arg)

Write the given data to this stream. If callback is given, we call it when all of the buffered write data has been successfully written to the stream. If there was previously buffered write data and an old write callback, that callback is simply overwritten with this new callback.

Parameters

- **data** (*String*) – The chunk to write to the stream.
- **callback** (*Function*) – Function to be called when data has been written to stream.
- **arg** – Optional argument for callback. If arg is given then it will be the first argument for the callback.

IOStream:write_buffer(buf, callback, arg)

Write the given `turbo.structs.buffer` to the stream.

Parameters

- **buf** (`turbo.structs.buffer` class instance) – The buffer to write to the stream.
- **callback** (*Function*) – Function to be called when data has been written to stream.
- **arg** – Optional argument for callback. If arg is given then it will be the first argument for the callback.

IOStream:write_zero_copy(buf, callback, arg)

Write the given buffer class instance to the stream without copying. This means that this write MUST complete before any other writes can be performed, and that the internal buffer has to be completely flushed before it is invoked. This can be achieved by either using `IOStream:writing` or adding a callback to other write methods callled before this. There is a barrier in place to stop this from happening. A error is raised in the case of invalid use. This method is recommended when you are serving static data, it refrains from copying the contents of the buffer into its internal buffer, at the cost of not allowing more data being added to the internal buffer before this write is finished. The reward is lower memory usage and higher throughput.

Parameters

- **buf** (`turbo.structs.buffer`) – The buffer to send. Will not be modified, and must not be modified until write is done.
- **callback** (*Function*) – Function to be called when data has been written to stream.
- **arg** – Optional argument for callback. If arg is given then it will be the first argument for the callback.

IOStream:set_close_callback(callback, arg)

Set a callback to be called when the stream is closed.

Parameters

- **callback** (*Function*) – Function to call on close.

- **arg** – Optional argument for callback.

IOStream:set_max_buffer_size(sz)
Set the maximum amount of bytes to be buffered internally in the IOStream instance. This limit can also be set on class instanciation. This method does NOT check the current size and does NOT immediately raise a error if the size is already exceeded. A error will instead occur when the IOStream is adding data to its buffer on the next occasion and detects a breached limit.

Parameters **sz** (*Number*) – Size of max buffer in bytes.

IOStream:close()
Close the stream and its associated socket.

IOStream:reading()
Is the stream currently being read from?

Return type Boolean

IOStream:writing()
Is the stream currently being written to?

Return type Boolean

IOStream:closed()
Has the stream been closed?

Return type Boolean

SSLIOStream class

The class is a extended IOStream class and uses OpenSSL for its implementation. All of the methods in its super class IOStream, are available. Obviously a SSL tunnel software is a more optimal approach than this, as there is quite a bit of overhead in handling SSL connections. For this class to be available, the global `_G.TURBO_SSL` must be set.

SSLIOStream(*fd*, *ssl_options*, *io_loop*, *max_buffer_size*)
Create a new SSLIOStream instance. You can use:

- `turbo.crypto.ssl_create_client_context`
- `turbo.crypto.ssl_create_server_context`

to create a SSL context to pass in the ssl_options argument.

ssl_options table should contain:

- "_ssl_ctx" - SSL_CTX pointer created with context functions in crypto.lua.
- "_type" - Optional number, 0 or 1. 0 indicates that the context is a server context, and 1 indicates a client context. If not set, it is presumed to be a server context.

Parameters

- **fd** (*Number*) – File descriptor, either open or closed. If closed then, the `turbo.iostream SSLIOStream:connect()` method can be used to connect.
- **ssl_options** (*Table*) – SSL arguments.
- **io_loop** (*IOLoop class instance*) – IOLoop class instance to use for event processing. If none is set then the global instance is used, see the `ioloop.instance()` function.
- **max_buffer_size** (*Number*) – The maximum number of bytes that can be held in internal buffer before flushing must occur. If none is set, 104857600 are used as default.

Return type SSLIOStream object

SSLIOStream:connect(address, port, family, verify, callback, errhandler, arg)
Connect to a address without blocking. To successfully use this method it is neccessary to check the return value, and also assign a error handler function. Notice that the verify arugment has been added as opposed to the `SSLIOStream:connect` method.

Parameters

- **host** (*String*) – The host to connect to. Either hostname or IP.
- **port** (*Number*) – The port to connect to. E.g 80.
- **family** – Socket family. Optional. Pass nil to guess.
- **verify** (*Boolean*) – Verify SSL certificate chain and match hostname in certificate on connect. Setting this to false is only recommended if the server certificates are self-signed or something like that.
- **callback** (*Function*) – Optional callback for "on successfull connect"
- **errhandler** (*Function*) – Optional callback for "on error". Called with errno and its string representation as arguments.
- **arg** – Optional argument for callback. callback and errhandler are called with this as first argument.

Return type Number. -1 + error message on error, 0 on success.

turbo.ioloop – Main I/O Loop

Single threaded I/O event loop implementation. The module handles socket events and timeouts and scheduled intervals with millisecond precision

The inner working are as follows:

- Set iteration timeout to 3600 milliseconds.
- If there exists any timeout callbacks, check if they are scheduled to be run. Run them if they are. If timeout callback would be delayed because of too long iteration timeout, the timeout is adjusted.
- If there exists any interval callbacks, check if they are scheduled to be run. If interval callback would be missed because of too long iteration timeout, the iteration timeout is adjusted.
- If any callbacks exists, run them. If callbacks add new callbacks, adjust the iteration timeout to 0.
- If there are any events for sockets file descriptors, run their respective handlers. Else wait for specified interval timeout, or any socket events, jump back to start.

Note that because of the fact that the server itself does not know if callbacks block or have a long processing time it cannot guarantee that timeouts and intervals are called on time. In a perfect world they would be called within a reasonable time of what is specified.

Event types for file descriptors are defined in the ioloop module's namespace: `turbo.ioloop.READ`, `turbo.ioloop.WRITE`, `turbo.ioloop.PRI`, `turbo.ioloop.ERROR`

`ioloop.instance()`

Create or get the global IOLoop instance. Multiple calls to this function returns the same IOLoop.

Return type IOLoop class.

IOLoop class

IOLoop is a level triggered I/O loop, with additional support for timeout and time interval callbacks. Heavily influenced by ioloop.py in the Tornado web framework. *Note: Only one instance of IOLoop can ever run at the same time!*

IOLoop()
Create a new IOLoop class instance.

IOLoop:add_handler(fd, events, handler, arg)
Add handler function for given event mask on fd.

Parameters

- **fd** (*Number*) – File descriptor to bind handler for.
- **events** (*Number*) – Events bit mask. Defined in ioloop namespace. E.g `turbo.ioloop.READ` and `turbo.ioloop.WRITE`. Multiple bits can be AND'ed together.
- **handler** (*Function*) – Handler function.
- **arg** – Optional argument for function handler. Handler is called with this as first argument if set.

Return type Boolean

IOLoop:update_handler(fd, events)
Update existing handler function's trigger events.

Parameters

- **fd** (*Number*) – File descriptor to update.
- **events** (*Number*) – Events bit mask. Defined in ioloop namespace. E.g `turbo.ioloop.READ` and `turbo.ioloop.WRITE`. Multiple bits can be AND'ed together.

IOLoop:remove_handler(fd)
Remove a existing handler from the IO Loop.

Parameters **fd** (*Number*) – File descriptor to remove handler from.

IOLoop:add_callback(callback, arg)
Add a callback to be called on next iteration of the IO Loop.

Parameters

- **callback** (*Function*) – A function to be called on next iteration.
- **arg** – Optional argument for callback. Callback is called with this as first argument if set.

Return type IOLoop class. Returns self for convinience.

IOLoop:add_timeout(timestamp, func, arg)
Add a timeout with function to be called in future. There is given no gurantees that the function will be called on time. See the note at beginning of this section.

Parameters

- **timestamp** (*Number*) – Timestamp when to call function, based on Unix CLOCK_MONOTONIC time in milliseconds precision. E.g util.gettimemonotonic() + 3000 will timeout in 3 seconds. See `turbo.util.gettimemonotonic()`.
- **func** (*Function*) – A function to be called after timestamp is reached.
- **arg** – Optional argument for func.

Return type Unique reference as a reference for this timeout. The reference can be used as parameter for `IOLoop:remove_timeout()`

`IOLoop:remove_timeout(ref)`
Remove a scheduled timeout by using its reference.

Parameters **`identifer`** (*`Number`*) – Identifier returned by `IOLoop:add_timeout()`

Return type Boolean

`IOLoop:set_interval(msec, func, arg)`
Add a function to be called every milliseconds. There is given no guarantees that the function will be called on time. See the note at beginning of this section.

Parameters

- **`msec`** (*`Number`*) – Milliseconds interval.
- **`func`** (*`Function`*) – A function to be called every msecs.
- **`arg`** – Optional argument for func.

Return type Unique numeric identifier as a reference to this interval. The refence can be used as parameter for `IOLoop:clear_interval()`

`IOLoop:clear_interval(ref)`
Clear a interval.

Parameters **`ref`** (*`Boolean`*) – Reference returned by `IOLoop:set_interval()`

`IOLoop:add_signal_handler(signo, handler, arg)`
Add a signal handler. If handler already exists for signal it is overwritten. Calling of multiple functions should be handled by user.

Parameters

- **`signo`** (*`(Number) Signal number, defined in turbo.signal, e.g turbo.signal.SIGINT.`*) – The signal number(s) too handle.
- **`handler`** (*`Function`*) – Function to handle the signal.
- **`arg`** – Optional argument for handler function.

`IOLoop:remove_signal_handler(signo)`
Remove a signal handler for specified signal number.

Parameters **`signo`** (*`(Number) Signal number, defined in turbo.signal, e.g turbo.signal.SIGINT.`*) – The signal number to remove.

`IOLoop:start()`
Start the IO Loop. The loop will continue running until `IOLoop.close` is called via a callback added.

`IOLoop:close()`
Close the I/O loop. This call must be made from within the running I/O loop via a callback, timeout, or interval. Notice: All pending callbacks and handlers are cleared upon close.

`IOLoop:running()`
Is the IO Loop running?

Return type Boolean

turbo.async – Asynchronous clients

Utilities for coroutines

task (*func*, ...)

A wrapper for functions that always takes callback and callback argument as last arguments to be able to yield and resume execution of function when callback is called from another function.

No callbacks required, the arguments that would normally be used to call the callback is put in the left-side result.

Usage: Consider one of the functions of the IOStream class which uses a callback based API: IOStream:read_until(delimiter, callback, arg)

```
local res = coroutine.yield(turbo.async.task(
        stream.read_until, stream, "\r\n"))

-- Result from read_until operation will be returned in the res variable.
```

A HTTP(S) client - HTTPClient class

Based on the IOStream/SSLIOStream and IOLoop classes. Designed to asynchronously communicate with a HTTP server via the Turbo I/O Loop. The user MUST use Lua's builtin coroutines to manage yielding, after doing a request. The aim for the client is to support as many standards of HTTP as possible. However there may be some artifacts as there usually are many compability fixes in equivalent software such as curl. Websockets are not handled by this class. It is the users responsibility to check the returned values for errors before usage.

When using this class, keep in mind that it is not supported to launch muliple :fetch()'s with the same class instance. If the instance is already in use then it will return a error.

Usage inside a `turbo.web.RequestHandler` method:

```
    local res = coroutine.yield(
            turbo.async.HTTPClient():fetch("http://domain.com/latest"))
    if res.error then
            self:write("Could not get latest from domain.come")
    else
            self:write(res.body)
    end
```

HTTPClient (*ssl_options*, *io_loop*, *max_buffer_size*)

Create a new HTTPClient class instance. One instance can serve 1 request at a time. If multiple request should be sent then create multiple instances.

Parameters

- **ssl_options** (*Table*) – SSL keys, verify certificate, CA path etc.
- **io_loop** (*IOLoop class instance.*) – Provide a IOLoop instance or global instance is used.
- **max_buffer_size** (*Number*) – Maximum response buffer size in bytes.

Available SSL options:

•`priv_file` (String) - Path to SSL / HTTPS private key file.

•`cert_file` (String) - Path to SSL / HTTPS certificate key file.

- `ca_path` (String) - Path to SSL / HTTPS CA certificate verify location, if not given builtin is used, which is copied from Ubuntu 12.10.
- `verify_ca` (Boolean) SSL / HTTPS verify servers certificate. Default is true.

errors

Numeric error codes set in the HTTPResponse returned by `HTTPClient:fetch`:

`INVALID_URL` - URL could not be parsed.

`INVALID_SCHEMA` - Invalid URL schema

`COULD_NOT_CONNECT` - Could not connect, check message.

`PARSE_ERROR_HEADERS` - Could not parse response headers.

`CONNECT_TIMEOUT` - Connect timed out.

`REQUEST_TIMEOUT` - Request timed out.

`NO_HEADERS` - Shouldn't happen.

`REQUIRES_BODY` - Expected a HTTP body, but none set.

`INVALID_BODY` - Request body is not a string.

`SOCKET_ERROR` - Socket error, check message.

`SSL_ERROR` - SSL error, check message.

`BUSY` - Operation in progress.

`REDIRECT_MAX` - Redirect maximum reached.

HTTPClient:fetch(url, kwargs)

Parameters

- **url** (*String*) – URL to fetch.
- **kwargs** (*Table*) – Keyword arguments

Return type `turbo.coctx.CoroutineContext` class instance. Resumes coroutine with `turbo.async.HTTPResponse`.

Available keyword arguments:

- `method` - The HTTP method to use. Default is `GET`
- `params` - Provide parameters as table.
- `keep_alive` - Reuse connection if scenario supports it.
- `cookie` - The cookie to use.
- `http_version` - Set HTTP version. Default is HTTP1.1
- `use_gzip` - Use gzip compression. Default is true.
- `allow_redirects` - Allow or disallow redirects. Default is true.
- `max_redirects` - Maximum redirections allowed. Default is 4.
- `on_headers` - Callback to be called when assembling request HTTPHeaders instance. Called with `turbo.httputil.HTTPHeaders` as argument.
- `body` - Request HTTP body in plain form.
- `request_timeout` - Total timeout in seconds (including connect) for request. Default is 60 seconds.

- `connect_timeout` - Timeout in seconds for connect. Default is 20 secs.
- `auth_username` - Basic Auth user name.
- `auth_password` - Basic Auth password.
- `user_agent` - User Agent string used in request headers. Default is `Turbo Client vx.x.x`.

HTTPResponse class

Represents a HTTP response by a few attributes. Returned by `turbo.async.HTTPClient:fetch`.

error (Table) Table with code and message members. Possible codes is defined in `async.errors`. Always check if the error attribute is set, before trying to access others. If error is set, then all of the other attributes, except request_time is nil.

request (HTTPHeaders class instance) The request header sent to the server.

code (Number) The HTTP response code

headers (HTTPHeader class instance) Response headers received from the server.

body (String) Body of response

url (String) The URL that was used for final resource.

request_time (Number) msec used to process request.

turbo.thread – Threads with communications

Thread class

The Thread class is implemented with C fork, and allows the user to create a seperate thread that runs independently of the parent thread, but can communicate with each other over a AF_UNIX socket. Useful for long and heavy tasks that can not be yielded. Also useful for running shell commands etc.

```
local turbo = require "turbo"

turbo.ioloop.instance():add_callback(function()

    local thread = turbo.thread.Thread(function(th)
        th:send("Hello World.")
        th:stop()
    end)

    print(thread:wait_for_data())
    thread:wait_for_finish()
    turbo.ioloop.instance():close()

end):start()
```

All functions may raise errors. All functions yield to the IOLoop internally. You may catch errors with xpcall or pcall. Make sure to do proper cleanup of threads not being used anymore, they are not automatically stopped and collected.

Thread(*func*)

Create a new thread. Start running in provided func.

Parameters **func** – Function to call when thread has been created. Function is called with the childs Thread object, which contains its own IOLoop e.g: "th.io_loop".

Return type `Thread object`

`Thread:stop()`
Stop and cleanup pipe. Can be called from both parent and child thread.

`Thread:send(data)`
Called by either parent or child thread to send data to each other. If you are calling from parent thread, make sure to call wait_for_pipe() first.

Parameters **`data`** – String to be sent.

`Thread:wait_for_data()`
Wait for data to become available from other thread. May be called by parent or child thread.

Parameters **`num_bytes`** (*`Number`*) – The amount of bytes to read.

Return type String

`Thread:wait_for_finish()`
Wait for child process to stop itself and end thread.

`Thread:wait_for_pipe()`
Wait for thread pipe to be connected. Must be used by main thread before attempting to send data to child. Only callable from parent thread.

`IOSimple:get_pid()`
Get PID of child.

Return type Number

turbo.escape – Escaping and JSON utilities

JSON conversion

`json_encode`(*t*)
JSON stringify a table. May raise a error if table could not be decoded.

Parameters **`t`** – Value to JSON encode.

Return type String

`json_decode`(*s*)
Decode a JSON string to table.

Parameters **`s`** (*`String`*) – JSON enoded string to decode into Lua primitives.

Return type Table

Escaping

`unescape`(*s*)
Unescape a escaped hexadecimal representation string.

Parameters **`s`** (*`String`*) – String to unescape.

Return type String

`escape`(*s*)
Encodes a string into its escaped hexadecimal representation.

Parameters **`s`** (*`String`*) – String to escape.

Return type String

html_escape(*s*)
Encodes the HTML entities in a string. Helpfull to avoid XSS.

Parameters **s** (*String*) – String to escape.

Return type String

String trimming

trim(*s*)
Remove trailing and leading whitespace from string.

Parameters **s** – String

Return type String

ltrim(*s*)
Remove leading whitespace from string.

Parameters **s** – String

Return type String

rtrim(*s*)
Remove trailing whitespace from string.

Parameters **s** – String

Return type String

turbovisor – Application supervisor

Turbovisor is an application management tool which detects file changes and restart application on the fly. There are several parameters to control its behavior.

Options

-w, –watch Specify files or directories to watch. If directory is given, all its sub-directories will be monitored as well. By default, turbovisor will monitor current directory.

-i, –ignore Specify files or directories to ignore. If directory is given, all its sub-directories will be ignored as well. This is uesfull for auto-generated files or temporary files. By default, no file is ignored.

Examples

Suppose we have the following directory tree, and turbovisor is invoked in the app's root directory

```
MyApp
|-- doc
|   |-- doc1.rst
|   |-- doc2.rst
|-- main.lua
|-- model.lua
|-- templates
```

```
|   |-- view.lua
|-- static
|   |-- files
|       |-- file1
|       |-- file2
|   |-- images
|       |-- image1.jpg
|       |-- image2.jpg
|   |-- sounds
|       |-- sound1.mp3
|       |-- sound2.mp3
|-- test.lua
```

turbovisor main.lua start application, detect any changes in the application

turbovisor main.lua -w model.lua start application, only monitor file model.lua

turbovisor main.lua –watch model.lua main.lua start application, only monitor file model.lua and main.lua

turbovisor main.lua -i static start application, detect any changes in the application, except static directory and its sub-directories

turbovisor main.lua –watch static –ignore static/images static/sounds/sound2.mp3 start application, monitor static directory, but ignore its images sub-directory and sound2.mp3 file

turbo.httputil – Utilities for the HTTP protocol

The httputil namespace contains the HTTPHeader class and POST data parsers, which is a integral part of the HTTPServer class.

HTTPParser class

Used to parse HTTP headers. Parsing is done through Ryan Dahls HTTP Parser via the FFI. It is very fast and contains various protection against attacks. The .so is compiled when Turbo is installed with `make install`. Note that this class has sanity checking for input parameters. If they are of wrong type or contains bad data they will raise a error.

`HTTPParser`(*hdr_str*, *hdr_t*)

Create a new HTTPParser class instance.

hdr_t available:

- `turbo.httputil.hdr_t.HTTP_RESPONSE`,
- `turbo.httputil.hdr_t.HTTP_REQUEST`,
- `turbo.httputil.hdr_t.HTTP_BOTH`

Parameters

- **`hdr_str`** (*`String`*) – (optional) Raw header, including ending double CLRF, if you want the class to parse headers on construction.
- **`hdr_t`** (*`Number`*) – (optional) If hdr_str is defined this should also be defined. It is used to specify what kind of header you want to parse.

Return type HTTPParser object

HTTPParser:get_url()
Get URL.

Return type String or nil

HTTPParser:get_url_field(UF_prop)
Get specified URL segment. If segment does not exist, nil is returned. Will throw error if no URL has been parsed. UF_prop available:

- `turbo.httputil.UF.SCHEMA`,
- `turbo.httputil.UF.HOST`,
- `turbo.httputil.UF.PORT`,
- `turbo.httputil.UF.PATH`,
- `turbo.httputil.UF.PATH`,
- `turbo.httputil.QUERY`,
- `turbo.httputil.UF.FRAGMENT`,
- `turbo.httputil.UF.USERINFO`

Parameters **UF_prop** (*Number*) – Segment to return, values defined in `turbo.httputil.UF`.

Return type String or nil

HTTPParser:parse_url(url)
Parse standalone URL and populate class instance with values. `HTTPParser.get_url_field` must be used to read out value.

Parameters **url** (*String*) – URL string.

HTTPParser:get_method()
Get current URL request method.

Return type String or nil

HTTPParser:get_status_code()
Get the current HTTP status code.

Return type Number or nil

HTTPParser:get(key, caseinsensitive)
Get given key from header key value section.

Parameters

- **key** (*String*) – Value to get, e.g "Content-Encoding".
- **caseinsensitive** (*Boolean*) – If true then the key will be matched without regard for case sensitivity.

Return type The value of the key in String form, or nil if not existing. May return a table if multiple keys are set.

HTTPParser:get_argument(name)
Get a argument from the query section of parsed URL. (e.g ?param1=myvalue) Note that this method only gets one argument. If there are multiple arguments with same name use `HTTPParser.get_arguments()`

Parameters **name** (*String*) – The name of the argument.

Return type String or nil

HTTPParser:get_arguments()
Get all URL query arguments of parsed URL. Support multiple values with same name.

Return type Table

HTTPParser:parse_response_header(raw_headers)
Parse HTTP response headers. Populates the class with all data in headers. Will throw error on parsing failure.

Parameters **raw_headers** (*String*) – Raw HTTP response header in string form.

HTTPParser:parse_request_header(raw_headers)
Parse HTTP request headers. Populates the class with all data in headers. Will throw error on parsing failure.

Parameters **raw_headers** (*String*) – Raw HTTP request header in string form.

HTTPHeaders class

Used to compile HTTP headers. Note that this class has sanity checking for input parameters. If they are of wrong type or contains bad data they will raise a error.

HTTPHeaders()
Create a new HTTPHeaders class instance.

Return type HTTPHeaders object

Manipulation

HTTPHeaders:set_uri(uri)
Set URI. Mostly usefull when building up request headers, NOT when parsing response headers. Parsing should be done with HTTPHeaders:parse_url.

Parameters **uri** (*String*) – URI string to set.

HTTPHeaders:get_uri()
Get current URI.

Return type String or nil

HTTPHeaders:set_method(method)
Set URL request method. E.g "POST" or "GET".

Parameters **method** (*String*) – Method to set.

HTTPHeaders:get_method()
Get current URL request method.

Return type String or nil

HTTPHeaders:set_version(version)
Set HTTP protocol version.

Parameters **version** (*String*) – Version string to set.

HTTPHeaders:get_version()
Get current HTTP protocol version.

Return type String or nil

HTTPHeaders:set_status_code(code)
Set HTTP status code. The code is validated against all known.

Parameters **code** (*Number*) – The code to set.

HTTPHeaders:get_status_code()
Get the current HTTP status code.

Return type Number or nil

HTTPHeaders:get(key, caseinsensitive)
Get given key from header key value section.

Parameters

- **key** (*String*) – Value to get, e.g "Content-Encoding".
- **caseinsensitive** (*Boolean*) – If true then the key will be matched without regard for case sensitivity.

Return type The value of the key in String form, or nil if not existing. May return a table if multiple keys are set.

HTTPHeaders:add(key, value)
Add a key with value to the headers. Supports adding multiple values to one key. E.g mutiple "Set-Cookie" header fields.

Parameters

- **key** (*String*) – Key to add to headers. Must be string or error is raised.
- **value** (*String*) – Value to associate with the key.

HTTPHeaders:set(key, value, caseinsensitive)
Set a key with value to the headers. Overwiting existing key.

Parameters

- **key** (*String*) – The key to set.
- **value** (*String*) – Value to associate with the key.
- **caseinsensitive** (*Boolean*) – If true then the existing keys will be matched without regard for case sensitivity and overwritten.

HTTPHeaders:remove(key, caseinsensitive)
Remove a key value combination from the headers.

Parameters

- **key** (*String*) – Key to remove.
- **caseinsensitive** (*Boolean*) – If true then the existing keys will be matched without regard for case sensitivity and overwritten.

Stringifiers

HTTPHeaders:stringify_as_request()
Stringify data set in class as a HTTP request header.

Return type String. HTTP header string excluding final delimiter.

HTTPHeaders:stringify_as_response()
Stringify data set in class as a HTTP response header. If not "Date" field is set, it will be generated automatically.

Return type String. HTTP header string excluding final delimiter.

HTTPHeaders:__tostring()
Convinience method to return HTTPHeaders:stringify_as_response on string conversion.

Return type String. HTTP header string excluding final delimiter.

Functions

parse_multipart_data(*data*)
Parse multipart form data.

Parameters **data** (*String*) – Multi-part form data in string form.

Return type Table of keys with corresponding values. Each key may hold multiple values if there were found multiple values for one key.

parse_post_arguments(*data*)
Parse ? and & separated key value fields.

Parameters **data** (*String*) – Form data in string form.

Return type Table of keys with corresponding values. Each key may hold multiple values if there were found multiple values for one key.

turbo.httpserver – Callback based HTTP Server

A non-blocking HTTPS Server based on the TCPServer class. Supports HTTP/1.0 and HTTP/1.1. Includes SSL support.

HTTPServer class

HTTPServer based on TCPServer, IOStream and IOLoop classes. This class is used by the `turbo.web.Application` class to serve its RequestHandlers. The server itself is only responsible for handling incoming requests, no response to the request is produced, that is the purpose of the request callback given as argument on initialization. The callback receives the HTTPRequest class instance produced for the incoming request and can by data provided in that instance decide on how it want to respond to the client. The callback must produce a valid HTTP response header and optionally a response body and use the `turbo.web.HTTPRequest:write` method. The server supports SSL, HTTP/1.1 Keep-Alive and optionally HTTP/1.0 Keep-Alive if the header field is specified.

Only use this class if you wish to have full control of things. Otherwise use the wrapper `turbo.web.Application`!

Example usage of HTTPServer:

```
local httpserver = require('turbo.httpserver')
local ioloop = require('turbo.ioloop')
local ioloop_instance = ioloop.instance()

function handle_request(request)
    local message = "You requested: " .. request.path
    request:write("HTTP/1.1 200 OK\r\nContent-Length:" .. message:len() .. "\r\n\r\n")
    request:write(message)
    request:finish()
end

http_server = httpserver.HTTPServer:new(handle_request)
http_server:listen(8888)
ioloop_instance:start()
```

HTTPServer(*request_callback*, *no_keep_alive*, *io_loop*, *xheaders*, *kwargs*)
Create a new HTTPServer class instance.

Parameters

- **request_callback** (*Function*) – Function to be called when requests are received by the server. The callback receives the HTTPRequest class instance produced for the incoming request as first argument. See the HTTPRequest documentation.
- **no_keep_alive** (*Boolean*) – If clients request to use Keep-Alive is to be ignored.
- **io_loop** (`turbo.ioloop.IOLoop` class instance.) – The IOLoop instance you want to use, if not defined the global instance is used.
- **xheaders** (*Boolean*) – Care about X-* header fields or not. If set to true the remote_ip attribute in self reflects the X-Real-Ip or X-Forwarded-For HTTP header value received.
- **kwars** – Optional keyword arguments

Available keyword arguments:

- `read_body` - Automatically read, and parse any request body. Default is true. If set to false, the user must read the body from the connection himself. Not reading a body in the case of a keep-alive request may lead to undefined behaviour. The body should be read or connection closed.
- `max_header_size` - The maximum amount of bytes a header can be. If exceeded, request is dropped.
- `max_body_size` - The maxium amount of bytes a request body can be. If exceeded, request is dropped. HAS NO EFFECT IF read_body IS FALSE.
- **ssl_options :** `key_file` - SSL key file if a SSL enabled server is wanted, `cert_file` - Certificate file.

General note regarding callbacks for all write methods: If you do writes before the previous callback has been called it is replaced with the new callback. If there is no callback defined in consequtive calls, the old callback is simply removed.

HTTPRequest class

Represents a HTTP request to the server.

This class has some attributes that can be accessed:

method (String) HTTP Request method. Also available in the `headers` attribute.

version (String) HTTP Version

uri (String) Uniform Resource Identifier

path (String) Request URL

headers `turbo.httputil.HTTPHeaders` class instance of the request headers.

body (String) The raw payload of the request, if any.

connection `turbo.httpserver.HTTPConection` class instance for this request.

files (Table) Files sent, file name is key and contents is its value.

host (String) Host IP

arguments (Table) Raw arguments table. The `turbo.web.RequestHandler` have convience methods for accessing these.

HTTPRequest:request_time()
Return the time used to handle the request or the time up to now if request not finished.

Return type (Number) Milliseconds the request took to finish, or up until now if not yet completed.

HTTPRequest:full_url()
Return the full URL that the user requested.

Return type String

HTTPRequest:write(chunk, callback, arg)
Writes a chunk of output to the stream.

Parameters

- **chunk** (*String*) – Data chunk to write to underlying IOStream.
- **callback** (*Function*) – Optional function called when buffer is fully flushed.
- **arg** – Optional first argument for callback.

HTTPRequest:write_buffer(buf, callback, arg)
Write the given `turbo.structs.buffer` to the underlying stream.

Parameters

- **buf** (`turbo.structs.buffer` class instance) – The buffer to write to the stream.
- **callback** (*Function*) – Optional function called when buffer is fully flushed
- **arg** – Optional argument for callback. If arg is given then it will be the first argument for the callback.

HTTPRequest:write_zero_copy(buf, callback, arg)
Write a Buffer class instance without copying it into the underlying IOStream's internal buffer. Some considerations has to be done when using this. Any prior calls to HTTPConnection:write or HTTPConnection:write_buffer must have completed before this method can be used. The zero copy write must complete before any other writes may be done. Also the buffer class should not be modified while the write is being completed. Failure to follow these advice will lead to undefined behaviour.

Parameters

- **buf** – Buffer class instance
- **callback** (*Function*) – Optional function called when buffer is fully flushed
- **arg** – Optional first argument for callback.

HTTPConnection:finish()
Finishes request.

HTTPRequest:supports_http_1_1()
Returns true if requester supports HTTP 1.1

Return type Boolean

HTTPConnection class

Represents a live connection to the server. Basically a helper class to HTTPServer. It uses the IOStream class's callbacks to handle the different sections of a HTTP request.

HTTPConnection:write(chunk, callback, arg)
Writes a chunk of output to the stream.

Parameters

- **chunk** (*String*) – Data chunk to write to underlying IOStream.
- **callback** (*Function*) – Optional function called when buffer is fully flushed.
- **arg** – Optional first argument for callback.

HTTPConnection:write_buffer(buf, callback, arg)
Write the given turbo.structs.buffer to the underlying stream.

Parameters

- **buf** (turbo.structs.buffer class instance) – The buffer to write to the stream.
- **callback** (*Function*) – Optional function called when buffer is fully flushed
- **arg** – Optional argument for callback. If arg is given then it will be the first argument for the callback.

HTTPConnection:write_zero_copy(buf, callback, arg)
Write a Buffer class instance without copying it into the underlying IOStream's internal buffer. Some considerations has to be done when using this. Any prior calls to HTTPConnection:write or HTTPConnection:write_buffer must have completed before this method can be used. The zero copy write must complete before any other writes may be done. Also the buffer class should not be modified while the write is being completed. Failure to follow these advice will lead to undefined behaviour.

Parameters

- **buf** – Buffer class instance
- **callback** (*Function*) – Optional function called when buffer is fully flushed
- **arg** – Optional first argument for callback.

HTTPConnection:finish()
Finishes request.

turbo.tcpserver – Callback based TCP socket Server

A simple non-blocking extensible TCP Server based on the IOStream class. Includes SSL support. Used as base for the Turbo HTTP Server.

TCPServer class

A non-blocking TCP server class. Users which want to create a TCP server should inherit from this class and implement the TCPServer:handle_stream() method. Optional SSL support is provided.

TCPServer(*io_loop*, *ssl_options*, *max_buffer_size*)
Create a new TCPServer class instance. If the SSL certificates is provided and can not be loaded, a error is raised.

Parameters

- **io_loop** (turbo.ioloop.IOLoop instance) – Provide specific IOLoop class instance. If not provided the global instance is used.
- **ssl_options** (*Table*) – Optional SSL parameters.
- **max_buffer_size** (*Number*) – The maximum buffer size of the server. If the limit is hit, the connection is closed.

Return type TCPServer class instance

Available ssl_options keys:

- "key_file" (String) - Path to SSL key file if a SSL enabled server is wanted.
- "cert_file" (String) - Path to certificate file. key_file must also be set.

TCPServer:handle_stream(stream, address)
This method is called by the class when clients connect. Implement this method in inheriting class to handle new connections.

Parameters

- **stream** (`turbo.iostream.IOStream` instance) – Stream for the newly connected client.
- **address** (*String*) – IP address of newly connected client.

TCPServer:listen(port, address, backlog, family)
Start listening on port and address. When using this method, as oposed to TCPServer:bind you should not call TCPServer:start. You can call this method multiple times with different parameters to bind multiple sockets to the same TCPServer.

Parameters

- **port** (*Number*) – The port number to bind to.
- **address** (*Number*) – The address to bind to in unsigned integer hostlong format. If not address is given, `turbo.socket.INADDR_ANY` will be used, binding to all addresses.
- **backlog** (*Number*) – Maximum backlogged client connects to allow. If not defined then 128 is used as default.
- **family** (*Number*) – Optional socket family. All socket familys are defined in `turbo.socket` module. If not defined AF_INET is used as default.

TCPServer:add_sockets(sockets)
Add multiple sockets in a table that should be bound on calling start. Use the `turbo.sockutil.bind_sockets` function to create sockets easily and add them to the sockets table.

Parameters **sockets** (*Table*) – 1 or more socket fd's.

TCPServer:add_socket(socket)
Single socket version of TCPServer:add_socket.

Parameters **socket** (*Number*) – Socket fd.

TCPServer:bind(port, address, backlog, family)
Bind this server to port and address. User must also call TCPServer:start to start listening on the bound socket.

Parameters

- **port** (*Number*) – The port number to bind to.
- **address** (*Number*) – The address to bind to in unsigned integer hostlong format. If not address is given, `turbo.socket.INADDR_ANY` will be used, binding to all addresses.
- **backlog** (*Number*) – Maximum backlogged client connects to allow. If not defined then 128 is used as default.
- **family** (*Number*) – Optional socket family. All socket familys are defined in `turbo.socket` module. If not defined AF_INET is used as default.

TCPServer:start()
Start the TCPServer, accepting conncetions on bound sockets.

TCPServer:stop()
Stop the TCPServer. Closing all the sockets bound to it. Before restarting the TCPServer, the socket must be readded.

turbo.structs – Data structures

Usefull data structures implemented using Lua and the LuaJIT FFI.

deque, Double ended queue

A deque can insert items at both the beginning and then end in constant time, "O(1)" If you are going to insert things regurarly to the front it is wise to use this class instead of the standard Lua table. Keep in mind that inserting to the back is still slower than a Lua table.

deque()
Create a new deque class instance.

Return type Deque class instance

deque:append(item)
Append elements to tail.

deque:appendleft(item)
Append element to head.

deque:peeklast()
Returns element at tail.

deque:peekfirst()
Returns element at front.

deque:pop()
Removes element at tail and returns it.

deque:popleft()
Removes element at head and returns it.

deque:not_empty()
Check if deque is empty.

Return type Boolean

deque:size()
Returns the amount of elements in the deque.

deque:strlen()
Find length of all elements in deque combined. (Only works if the elements have a :len() method)

deque:concat()
Concat elements in deque. Only works if the elements in the deque have a :__concat() method.

deque:getn(pos)
Get element at position.

Return type Element or nil if not existing.

buffer, Low-level mutable buffer

Can be used to replace plain Lua strings where it is of importance to not create temporary strings, and there is little help in the Lua string interning. It is mutable and allows preallocations to be done on intialization. The data stored in a buffer is not handled by the LuaJIT 2.0 GC which in turn circumvents the memory limit.

Keep in mind that this class i "low-level" and giving the wrong arguments to its methods may cause memory segmentation fault. It is NOT protected.

Buffer(*size_hint*)
Create a new buffer. May raise error if there is not enough memory available.

Parameters **size_hint** (*Number*) – The buffer is preallocated with this amount (in bytes) of storage.

Return type Buffer class instance.

Buffer:append_right(data, len)
Append data to buffer. Keep in mind that defining a length longer than the actual data, might lead to a segmentation fault.

Parameters

- **data** (*char **) – The data to append in char * form.
- **len** (*Number*) – The length of the data in bytes.

Buffer:append_luastr_right(str)
Append Lua string to buffer.

Parameters **str** (*String*) – The data to append.

Buffer:append_left(data, len)
Prepend data to buffer.

Parameters

- **data** (*char **) – The data to prepend in char * form.
- **len** (*Number*) – The length of the data in bytes.

Buffer:append_luastr_left(str)
Prepend Lua string to the buffer.

Parameters **str** (*String*) – The data to prepend.

Buffer:pop_left(sz)
Pop bytes from left side of buffer. If sz exceeds size of buffer then a error is raised. Note: does not release memory allocated.

Parameters **sz** (*Number*) – Bytes to "pop".

Buffer:pop_right(sz)
Pop bytes from right side of the buffer. If sz exceeds size of buffer then a error is raised. Note: does not release memory allocated.

Parameters **sz** (*Number*) – Bytes to "pop".

Buffer:get()
Get internal buffer pointer. Must be treated as a const value. Keep in mind that the internal pointer may or may not change when calling its methods.

Return type Two values: const char * to data and current size in bytes.

Buffer:copy()
Create a "deep" copy of the buffer.

Return type Buffer class instance

Buffer:shrink()
Shrink buffer memory (deallocate) usage to its minimum.

Buffer:clear(wipe)
Clear buffer. Note: does not release memory allocated.

Parameters **wipe** (*Boolean*) – Optional switch to zero fill allocated memory range.

Buffer:len()
Get current size of the buffer.

Return type Number. Size in bytes.

Buffer:mem()
Get the total number of bytes currently allocated to this instance.

Return type Number. Bytes allocated.

Buffer:__tostring()
Convert to Lua type string using the tostring() builtin or implicit conversions.

Buffer:__eq(cmp)
Compare two buffers by using the == operator.

Buffer:__concat(src)
Concat by using the .. operator, Lua type strings can be concated also. Please note that all concatination involves deep copying and is slower than manually building a buffer with append methods.

turbo.hash – Cryptographic Hashes

Wrappers for the OpenSSL crypto library.

SHA1 class

SHA1 (*str*)
Create a SHA1 object. Pass a Lua string with the initializer to digest it.

Parameters **str** (*String or nil*) – Lua string to digest immediately. Note that you cannot call `SHA1.update` or `SHA1.final` afterwards as the digest is already final.

SHA1:update(str)
Update SHA1 context with more data

Parameters **str** – String

hash.SHA1:final()
Finalize SHA1 context

Return type (char*) Message digest.

SHA1:__eq(cmp)
Compare two SHA1 contexts with the equality operator ==.

Return type (Boolean) True or false.

SHA1:hex()
Convert message digest to Lua hex string.

Return type String

HMAC(*key*, *digest*)
Keyed-hash message authentication code (HMAC) is a specific construction for calculating a message authentication code (MAC) involving a cryptographic hash function in combination with a secret cryptographic key.

Parameters

- **key** (*String*) – Sequence of bytes used as a key.
- **digest** (*String*) – String to digest.

Return type String. Hex representation of digested string.

turbo.util Common utilities

The util namespace contains various convinience functions that fits no where else. As Lua is fairly naked as standard. Just the way we like it.

Table tools

strsplit(*str*, *sep*, *max*, *pattern*)
Split a string into a table on given seperator. This function extends the standard string library with new functionality.

Parameters

- **str** (*String*) – String to split
- **sep** (*String*) – String that seperate elements.
- **max** (*Number*) – Max elements to split
- **pattern** (*Boolean*) – Separator should be treated as a Lua pattern. Slower.

Return type Table

join(*delimiter*, *list*)
Join a table into a string.

Parameters

- **delimiter** (*String*) – Inserts this string between each table element.
- **list** (*Table*) – The table to join.

Return type String

is_in(*needle*, *haystack*)
Search table for given element.

Parameters

- **needle** (*Any that supports == operator.*) – The needle to find.
- **haystack** (*Table*) – The haystack to search.

tablemerge(*t1*, *t2*)
Merge two tables together.

Parameters

- **t1** (*Table*) – First table.
- **t2** (*Table*) – Second table.

Return type Table

Low level

mem_dump(*ptr*, *sz*)
Dump memory region to stdout, from ptr to given size. Usefull for debugging Luajit FFI. Notice! This can and will cause a SIGSEGV if not being used on valid pointers.

Parameters

- **ptr** (*cdata*) – A cdata pointer (from FFI)
- **sz** (*Number*) – Length to dump contents for.

TBM(*x*, *m*, *y*, *n*)
Turbo Booyer-Moore memory search algorithm. Search through arbitrary memory and find first occurence of given byte sequence. Effective when looking for large needles in a large haystack.

Parameters

- **x** (*char**) – Needle memory pointer.
- **m** (*int*) – Needle size.
- **y** (*char**) – Haystack memory pointer.
- **n** (*int*) – Haystack size.

Return type First occurence of byte sequence in y defined in x or nil if not found.

Misc

file_exists(*name*)
Check if file exists on local filesystem.

Parameters **path** (*String*) – Full path to file.

Return type Boolean

hex(*num*)
Convert number value to hexadecimal string format.

Parameters **num** (*Number*) – The number to convert.

Return type String

gettimeofday()
Returns the current time in milliseconds precision. Unlike Lua builtin which only offers granularity in seconds.

Return type Number

gettimemonotonic()
Returns milliseconds since arbitraty start point, doesn't jump due to time changes.

Return type Number

turbo.sockutil – Socket utilites and helpers

bind_sockets (*port*, *address*, *backlog*, *family*)

Binds sockets to port and address. If not address is defined then * will be used. If no backlog size is given then 128 connections will be used.

Parameters

- **port** – (Number) The port number to bind to.
- **address** – (Number) The address to bind to in unsigned integer hostlong format. If not address is given, INADDR_ANY will be used, binding to all addresses.
- **backlog** – (Number) Maximum backlogged client connects to allow. If not defined then 128 is used as default.
- **family** – (Number) Optional socket family. Defined in Socket module. If not defined AF_INET is used as default.

Return type (Number) File descriptor

add_accept_handler (*sock*, *callback*, *io_loop*, *arg*)

Add accept handler for socket with given callback. Either supply a IOLoop object, or the global instance will be used...

Parameters

- **sock** – (Number) Socket file descriptor to add handler for.
- **callback** – (Function) Callback to handle connects. Function receives socket fd (Number) and address (String) of client as parameters.
- **io_loop** – (IOLoop instance) If not set the global is used.
- **arg** – Optional argument for callback.

turbo.log – Command-line log helper

A simple log writer implementation with different levels and standard formatting. Messages written is appended to level and timestamp. You can turn off unwanted categories by modifiying the table at log.categories.

For messages shorter than 4096 bytes a static buffer is used to improve performance. C time.h functions are used as Lua builtin's is not compiled by LuaJIT. This statement applies to all log functions, except log.dump.

Example output: `[S 2013/07/15 18:58:03] [web.lua] 200 OK GET / (127.0.0.1) 0ms`

To enable or disable categories, modify the table in `turbo.log.categories`.

As default, it is declared as such:

```
log = {
    ["categories"] = {
        -- Enable or disable global log categories.
        -- The categories can be modified at any time.
        ["success"] = true,
        ["notice"] = true,
        ["warning"] = true,
        ["error"] = true,
        ["debug"] = true,
        ["development"] = false
```

```
    }
}
```

success(*str*)
Log to stdout. Success category. Use for successfull events etc. Messages are printed with green color.

Parameters **str** (*String*) – Log string.

notice(*str*)
Log to stdout. Notice category. Use for notices, typically non-critical messages to give a hint. Messages are printed with white color.

Parameters **str** (*String*) – Log string.

warning(*str*)
Log to stderr. Warning category. Use for warnings. Messages are printed with yellow color.

Parameters **str** (*String*) – Log string.

error(*str*)
Log to stderr. Error category. Use for critical errors, when something is clearly wrong. Messages are printed with red color.

Parameters **str** (*String*) – Log string.

debug(*str*)
Log to stdout. Debug category. Use for debug messages not critical for releases.

Parameters **str** (*String*) – Log string.

devel(*str*)
Log to stdout. Development category. Use for development purpose messages. Messages are printed with cyan color.

Parameters **str** (*String*) – Log string.

stringify(*t*, *name*, *indent*)
Stringify Lua table.

Parameters

- **t** (*Table*) – Lua table
- **name** (*String*) – Optional identifier for table.
- **indent** (*Number*) – Optional indent level.

Return type String

Index

S

T

U

W