Tarantool Server Gateway Interface

Rationale

Unifying the function handler for HTTP Requests has the following benefits:

1. *Decoupling Web Servers from HTTP Routers*.
   This will allow us to switch between Web Servers in existing application without the need to rewrite the HTTP Router source code.

2. *Enable HTTP Middleware reuse between HTTP Routers*.
   See https://wsgi.readthedocs.io/en/latest/libraries.html for examples.

Inspiration

The specification is almost entirely based on Ruby Rack specification. Which in turn is almost entirely based on WSGI.

Short Example

```
function handle(env)
   local method = env.REQUEST_METHOD
   local content_type = env.HEADER_CONTENT_TYPE
   local body = env[‘tsgi.input’]:read()
   …
   local response = {
      status = 200,
      body = '{name = "John Doe"}',
      headers = {[‘Content-Type'] = 'application/json'}
   }
   return response
end
```

Request Handler

HTTP request handling function must take one argument of table type: **env.**

CGI-like Keys

env Key	Value	Description
HTTP_ Variables	string	Variables corresponding to

		the client-supplied HTTP request headers (i.e., variables whose names begin with HTTP_). The presence or absence of these variables should correspond with the presence or absence of the appropriate HTTP header in the request. See RFC3875 section 4.1.18 for specific behavior.
REQUEST_METHOD	string	The HTTP request method, such as "GET" or "POST". This cannot ever be an empty string, and so is always required.
SCRIPT_NAME	string	RESERVED.
PATH_INFO	string	The remainder of the request URL's “path”, designating the virtual “location” of the request's target within the application. This may be an empty string, if the request URL targets the application root and does not have a trailing slash. This value may be percent-encoded when originating from a URL.
QUERY_STRING	string	The portion of the request URL that follows the ?, if any. May be empty, but is always required!
SERVER_NAME, SERVER_PORT	string	When combined with SCRIPT_NAME and PATH_INFO, these variables can be used to complete the URL. Note, however, that HTTP_HOST, if present, should be used in preference to SERVER_NAME for

		reconstructing the request URL. SERVER_NAME and SERVER_PORT can never be empty strings, and so are always required.

TSGI Specific Keys

env Key	Value	Description
tsgi.version	string	TSGI version that Web Server supports.
tsgi.url_scheme	string	“http” or “https”.
tsgi.input	table See Input Stream Interface below.	Used for reading request body.
tsgi.errors	table	RESERVED.
tsgi.hijack	nil or function See Hijacking below.	If present, allows “stealing” TCP connection.

Input Stream Interface

Lua-table **input_stream** implements Input Stream Interface, if it has the methods described below.
Used for reading HTTP Request Body.

- **str = input_stream:read(n)**
  str = input_stream:read()

  Returns at most n bytes from stream.
  If n is not specified, returns all bytes.
  n must be non-negative or nil.

- **input_stream:rewind()**

  Rewinds the input stream back to the beggining.

Hijacking

Let **hijack = env[‘tsgi.hijack’]**.

If **hijack** is **nil**, Web Server doesn’t support Upgrading.
Otherwise, when invoked **hijack()** must close the current HTTP connection, and return its’ (still open) raw TCP connection object.

Used for upgrading from HTTP to other procotols.

Response

Request handling function must return a table with the following keys:

Key	Type	Description
status	number	HTTP status.
headers	stateless iterable Must behave as table with string keys (header names) and strings or array of strings values (header values)	HTTP headers.
body	string or Wrapped Iterator See Wrapped Iterator below.	HTTP response body.

Wrapped Iterator Interface

Lua-table **stateful_iterable** implements Wrapped Iterator Interface if it has the fields below.
Used for transfering HTTP Response Body incrementally.

- **stateful_iterable.gen**

  Iterator function.
  Function taking (state, param) and producing new param.

- **stateful_iterable.state**

  State of iterator.
  Variable of any type.

- **stateful_iterable.param**

  Initial value for Lua iterator.
  Variable of any type.

Middleware

TSGI middleware consists of TSGI-compliant functions.

Every TSGI-compliant middleware will work with any TSGI-compliant Web Server and other TSGI-compliant middleware (that is it doesn't depend on HTTP Router e.g.).

Examples of middleware may be: request preprocessors (session managing middleware, route dispatching), response postprocessors.

Artificial example of a session managing module:

tsgi_session_middleware.lua (implementation):

```
session_mt = {
   -- e.g. save() for saving the session,
   --      renew() for updating the expiration time,
}

local function session_from_space(cookie)
   local session_tuple = ...
   local session = ...
   return setmetatable(session, session_mt)
end

local function wrapper_call(self, env)
   local cookie = env.HEADER_COOKIE
   env.session = session_object_from_space(cookie)

   local res = self.handle(env)
   res.headers['Set-Cookie'] = res.headers['Set-Cookie'] .. '\n' ..
env.session:setcookie_header()
   return res
end

local wrapper_mt = {
   __index = {
      __call = wrapper_call
   },
}
```

```
local function wrap(handler)
    local new_handler = {handler = handler}
    return setmetatable(new_handler, wrapper_mt)
end

return {wrap = wrap}
```

application.lua (usage):

```
...
local session_middleware = require('tsgi_session_middleware')

local function handler(env)
    -- we will magically have a session object at our disposal if we chain the
handler with session middleware (like below)
    session = env.session
    -- ...
end

-- TSGI allows for chaining handlers
local main_handler = session_middleware.wrap(handler)
...
```

Closed Questions

Duplicate HTTP Headers

HTTP/1.1 allows sending multiple headers with same name.
We can support setting Lua table as a header value in response to facilitate the case.

It must be noted that this is not necessary by the standard, as one can provide multiple values for the header by constructing a comma-separated combination of them. This is equivalent by the standard https://www.w3.org/Protocols/rfc2616/rfc2616-sec4.html#sec4.2:

> *Multiple message-header fields with the same field-name MAY be present in a message if and only if the entire field-value for that header field is defined as a comma-separated list [i.e., #(values)]. It MUST be possible to combine the multiple header fields into one "field-name: field-value" pair, without changing the semantics of the message, by appending each subsequent field-value to the first, each separated by a comma. The order in which header fields with the same field-name are received is therefore significant to the interpretation of the combined field value, and thus a proxy MUST NOT change the order of these field values when a message is forwarded*

On the other hand from https://tools.ietf.org/html/rfc6265#section-3:

> *Origin servers SHOULD NOT fold multiple Set-Cookie header fields into a single header field. The usual mechanism for folding HTTP headers fields (i.e., as defined in [RFC2616]) might change the semantics of the Set-Cookie header field because the %x2C (",") character is used by Set-Cookie in a way that conflicts with such folding.*

Solution

To resolve the issue, we allow setting array of strings to specific key (header name) in **response.headers** table, in this case corresponding header values are duplicated in HTTP response.

TODO (not fixed by specification yet)

- Describe TCP Connection Interface returned by **env[`tsgi.hijack`]()**
- Describe Case Sensitivity and Hyphen conversion issues in Open Issues.