Valum Documentation

Release 0.3.13

Antono Vasiljev, Guillaume Poirier-Morency

May 12, 2017

Contents

Valum is a Web micro-framework written in Vala and licensed under the LGPLv3. Its source code and releases are available on GitHub: valum-framework/valum.

This user documentation aims to be as complete as possible and covers topics that are not directly related to the framework, but essential for Web development. If you think that this document could be improved, open a ticket on GitHub to let us know.

CHAPTER 1

Installation

This document describes the compilation and installation process. Most of that work is automated with Meson, a build tool written in Python.

Packages

Packages for RPM and Debian based Linux distributions will be provided for stable releases so that the framework can easily be installed in a container or production environment.

Fedora

RPM packages for Fedora (24, 25 and rawhide) and EPEL 7 (CentOS, RHEL) are available from the arteymix/valum-framework Copr repository.

```
dnf copr enable arteymix/valum-framework
```

The `valum-0.3` package contains the shared libraries, `valum-0.3-devel` contains all that is necessary to build an application and `valum-0.3-doc` deliver user and API documentation.

```
dnf install valum-0.3 valum-0.3-devel valum-0.3-doc
```

Nix

```
nix-shell -p valum
```

Solus

```
eopkg it valum
```

Arch Linux (AUR)

```
yaourt valum
```

Subproject

If your project uses the Meson build system, you may integrate the framework as a subproject. The project must be cloned in the `subprojects` folder, preferably using a git submodule. Be careful using a tag and not the `master` trunk.

The following variables can be used as dependencies:

- `vsgi` for the abstraction layer
- `valum` for the framework

Note that due to Meson design, dependencies must be explicitly provided.

```
project('app', 'c', 'vala')

glib = dependency('glib-2.0')
gobject = dependency('gobject-2.0')
gio = dependency('gio-2.0')
soup = dependency('libsoup-2.4')
vsgi = subproject('valum').get_variable('vsgi')
valum = subproject('valum').get_variable('valum')

executable('app', 'app.vala',
           dependencies: [glib, gobject, gio, soup, vsgi, valum])
```

Docker

To use Valum with Docker, use the provided valum/valum image. It is based on the latest stable Ubuntu.

```
FROM valum/valum:latest

WORKDIR /app
ADD . .

RUN valac --pkg=valum-0.3 app.vala

EXPOSE 3003

ENTRYPOINT /app/app
```

Vagrant

You can provision a Vagrant VM with Valum. There's no `Vagrantfile` provided because each project will likely have it's own setup and deployment constraints.

```
wget https://github.com/valum-framework/valum/archive/v0.3.0.zip

unzip v0.3.0.zip

cd valum-0.3.0
mkdir build
meson --prefix=/usr --buildtype=release build
ninja -C build
ninja -C build test
ninja -C build install
```

Dependencies

The following dependencies are minimal to build the framework under Ubuntu 12.04 LTS and should be satisfied by most recent Linux distributions.

Package	Version
vala	>=0.26
python	>=3.4
meson	>=0.36
ninja	>=1.6.0
glib-2.0	>=2.40
gio-2.0	>=2.40
gio-unix-2.0	>=2.40
libsoup-2.4	>=2.44

Recent dependencies will enable more advanced features:

Package	Version	Feature
gio-2.0	>=2.44	better support for asynchronous I/O
libsoup-2.4	>=2.48	new server API

You can also install additional dependencies to build the examples, you will have to specify the `-D enable_examples=true` flag during the configure step.

Package	Description
ctpl	C templating library
gee-0.8	data structures
json-glib-1.0	JSON library
libmemcached	client for memcached cache storage
libluajit	embed a Lua VM
libmarkdown	parser and generator for Markdown
template-glib	templating library

Download the sources

You may either clone the whole git repository or download one of our releases from GitHub:

```
git clone git://github.com/valum-framework/valum.git && cd valum
```

The `master` branch is a development trunk and is not guaranteed to be very stable. It is always a better idea to checkout the latest tagged release.

Build

```
mkdir build && cd build
meson ..
ninja # or 'ninja-build' on some distribution
```

Install

The framework can be installed for system-wide availability.

```
sudo ninja install
```

Once installed, VSGI implementations will be looked up into `${prefix}/${libdir}/vsgi-0.3/servers`. This path can be changed by setting the `VSGI_SERVER_PATH` environment variable.

Run the tests

```
ninja test
```

If any of them fail, please open an issue on GitHub so that we can tackle the bug. Include the test logs (e.g. `build/meson-private/mesonlogs.txt`) and any relevant details.

Run the sample application

You can run the sample application from the `build` folder if you called `meson` with the `-D enable_examples=true` flag. The following example uses the *HTTP* server.

```
./build/example/app/app
```

CHAPTER 2

Quickstart

Assuming that Valum is built and installed correctly (view *Installation* for more details), you are ready to create your first application!

Simple 'Hello world!' application

You can use this sample application and project structure as a basis. The full valum-framework/example is available on GitHub and is kept up-to-date with the latest changes in the framework.

```
using Valum;
using VSGI;

var app = new Router ();

app.get ("/", (req, res) => {
    return res.expand_utf8 ("Hello world!");
});

Server.new ("http", handler: app.handle).run ({"app", "--port", "3003"});
```

Typically, the `run` function contains CLI argument to make runtime the parametrizable.

It is suggested to use the following structure for your project, but you can do pretty much what you think is the best for your needs.

```
build/
src/
    app.vala
```

Building with valac

Simple applications can be built directly with `valac`:

```
valac --pkg=valum-0.3 -o build/app src/app.vala
```

The `vala` program will build and run the produced binary, which is convenient for testing:

```
vala --pkg=valum-0.3 src/app.vala
```

Building with Meson

Meson is highly-recommended for its simplicity and expressiveness. It's not as flexible as waf, but it will handle most projects very well.

```
project('example', 'c', 'vala')

glib = dependency('glib-2.0')
gobject = dependency('gobject-2.0')
gio = dependency('gio-2.0')
soup = dependency('libsoup-2.4')
vsgi = dependency('vsgi-0.3')    # or subproject('valum').get_variable('vsgi')
valum = dependency('valum-0.3') # or subproject('valum').get_variable('valum')

executable('app', 'src/app.vala', dependencies: [glib, gobject, gio, soup, vsgi,␣
↪valum])
valum = dependency('valum-0.3')
```

```
meson . build
ninja -C build
```

To include Valum as a subproject, it is sufficient to clone the repository into `subprojects/valum`.

Building with waf

It is preferable to use a build system like waf to automate all this process. Get a release of `waf` and copy this file under the name `wscript` at the root of your project.

```
def options(cfg):
    cfg.load('compiler_c')

def configure(cfg):
    cfg.load('compiler_c vala')
    cfg.check_cfg(package='valum-0.3', uselib_store='VALUM', args='--libs --cflags')

def build(bld):
    bld.load('compiler_c vala')
    bld.program(
        packages = 'valum-0.3',
        target   = 'app',
        source   = 'src/app.vala',
        use      = 'VALUM')
```

You should now be able to build by issuing the following commands:

```
./waf configure
./waf build
```

Running the example

VSGI produces process-based applications that are either self-hosted or able to communicate with a HTTP server according to a standardized protocol.

The *HTTP* implementation is self-hosting, so you just have to run it and point your browser at http://127.0.0.1:3003 to see the result.

```
./build/app
```

CHAPTER 3

Application

This document explains step-by-step the sample presented in the *Quickstart* document.

Many implementations are provided and documented in *Server*.

Creating an application

An application is defined by a function that respects the `VSGI.ApplicationCallback` delegate. The *Router* provides `handle` for that purpose along with powerful routing facilities for client requests.

```
var app = new Router ();
```

Binding a route

An application constitute of a list of routes matching and handling user requests. The router provides helpers to declare routes which internally use `Route` instances.

```
app.get ("/", (req, res, next, context) => {
    return res.expand_utf8 ("Hello world!");
});
```

Every route declaration has a callback associated that does the request processing. The callback, named handler, receives four arguments:

- a *Request* that describes a resource being requested
- a *Response* that correspond to that resource
- a `next` continuation to *keep routing*
- a routing `context` to retrieve and store states from previous and for following handlers

Note: For an alternative, more structured approach to route binding, see *Cleaning up route logic*

Serving the application

This part is pretty straightforward: you create a server that will serve your application at port `3003` and since `http` was specified, it will be served with *HTTP*.

```
Server.new ("http", handler: app).run ({"app", "--port", "3003"});
```

Server takes a server implementation and an `ApplicationCallback`, which is respected by the `handle` function.

Usually, you would only pass the CLI arguments to `run`, so that your runtime can be parametrized easily, but in this case we just want our application to run with fixed parameters. Options are documented per implementation.

```
public static void main (string[] args) {
    var app = new Router ();

    // assume some route declarations...

    Server.new ("http", handler: app).run (args);
}
```

CHAPTER 4

VSGI

VSGI is a middleware that interfaces different Web server technologies under a common and simple set of abstractions.

For the moment, it is developed along with Valum to target the needs of a Web framework, but it will eventually be extracted and distributed as a shared library.

HTTP authentication

VSGI provide implementations of both basic and digest authentication schemes respectively defined in RFC 7617 and RFC 7616.

Both `Authentication` and `Authorization` objects are provided to produce and interpret their corresponding HTTP headers. The typical authentication pattern is highlighted in the following example:

```
var authentication = BasicAuthentication ("realm");

var authorization_header = req.headers.get_one ("Authorization");

if (authorization_header != null) {
    if (authentication.parse_authorization_header (authorization_header,
                                                   out authorization)) {
        var user = User.from_username (authorization.username);
        if (authorization.challenge (user.password)) {
            return res.expand_utf8 ("Authentication successful!");
        }
    }
}

res.headers.replace ("WWW-Authenticate", authentication.to_authenticate_header ());
```

Basic

The `Basic` authentication scheme is the simplest one and expect the user agent to provide username and password in plain text. It should be used exclusively on a secured transport (e.g. HTTPS).

Connection

All resources necessary to process a *Request* and produce a *Response* are bound to the lifecycle of a connection instance.

> **Warning:** It is not recommended to use this directly as it will most likely result in corrupted operations with no regard to the transfer encoding or message format.

The connection can be accessed from the *Request* `connection` property. It is a simple gio-2.0/GLib.IOStream that provides native access to the input and output stream of the used technology.

The following example shows how to bypass processing with higher-level abstractions. It will only work on *HTTP*, as CGI-like protocols require the status to be part of the response headers.

```
var message = req.connection.output_stream;
message.write_all ("200 Success HTTP/1.1\r\n".data. null);
message.write_all ("Connection: close\r\n");
message.write_all ("Content-Type: text/plain\r\n");
message.write_all ("\r\n".data);
message.write_all ("Hello world!".data);
```

Request

Requests are representing incoming demands from user agents to resources served by an application.

Method

Deprecated since version 0.3: libsoup-2.4 provide an enumeration of valid HTTP methods and this will be removed once exposed in their Vala API.

The `Request` class provides constants for the following HTTP methods:

- `OPTIONS`
- `GET`
- `HEAD`
- `POST`
- `PUT`
- `DELETE`
- `TRACE`
- `CONNECT`
- `PATCH`

Additionally, an array of supported HTTP methods is provided by `Request.METHODS`.

```
if (req.method == Request.GET) {
    return res.expand_utf8 ("Hello world!");
}

if (req.method == Request.POST) {
    return res.body.splice (req.body, OutputStreamSpliceFlags.NONE);
}

if (req.method in Request.METHODS) {
    // handle a standard HTTP method...
}
```

Headers

Request headers are implemented with libsoup-2.4/Soup.MessageHeaders and can be accessed from the `headers` property.

```
var accept = req.headers.get_one ("Accept");
```

libsoup-2.4 provides a very extensive set of utilities to process the information contained in headers.

```
SList<string> unacceptable;
Soup.header_parse_quality_list (req.headers.get_list ("Accept"), out unacceptable);
```

Cookies

Cookies can also be retrieved from the request headers.

Query

The HTTP query is provided in various way:

- parsed as a `HashTable<string, string>?` through the `Request.query` property
- raw with `Request.uri.get_query`

If the query is not provided (e.g. no `?` in the URI), then the `Request.query` property will take the `null` value.

Note: If the query is not encoded according to `application/x-www-form-urlencoded`, it has to be parsed explicitly.

To safely obtain a value from the HTTP query, use `Request.lookup_query` with the null-coalescing operator `??`.

```
req.lookup_query ("key") ?? "default value";
```

Body

The body is provided as a gio-2.0/GLib.InputStream by the `body` property. The stream is transparently decoded from any applied transfer encodings.

Implementation will typically consume the status line, headers and newline that separates the headers from the body in the base stream at construct time. It also guarantee that the body has been decoded if any transfer encoding were applied for the transport.

If the content is encoded with the `Content-Encoding` header, it is the responsibility of your application to decode it properly. VSGI provides common *Converters* to simplify the task.

Flatten

New in version 0.2.4.

In some cases, it is practical to flatten the whole request body in a buffer in order to process it as a whole.

The `flatten`, `flatten_bytes` and `flatten_utf8` functions accumulate the request body into a buffer (a gio-2.0/GLib.MemoryOutputStream) and return the corresponding `uint8[]` data buffer.

The request body is always fixed-size since the HTTP specification requires any request to provide a `Content-Length` header. However, the environment should be configured with a hard limit on payload size.

When you are done, it is generally a good thing to close the request body and depending on the used implementation, this could have great benefits such as freeing a file resource.

```
var payload = req.flatten ();
```

Form

libsoup-2.4/Soup.Form can be used to parse `application/x-www-form-urlencoded` format, which is submitted by Web browsers.

```
var data = Soup.Form.decode (req.flatten_utf8 (out bytes_read));
```

Multipart body

Multipart body support is planned in a future minor release, more information on issue #81. The implementation will be similar to libsoup-2.4/Soup.MultipartInputStream and provide part access with a filter approach.

Response

Responses are representing resources requested by a user agent. They are actively streamed across the network, preferably using non-blocking asynchronous I/O.

Status

The response status can be set with the `status` property. libsoup-2.4 provides an enumeration in libsoup-2.4/Soup.Status for that purpose.

The `status` property will default to `200 OK`.

The status code will be written in the response with `write_head` or `write_head_async` if invoked manually. Otherwise, it is left to the implementation to call it at a proper moment.

```
res.status = Soup.Status.MALFORMED;
```

Reason phrase

New in version 0.3.

The reason phrase provide a textual description for the status code. If `null`, which is the default, it will be generated using libsoup-2.4/Soup.Status.get_phrase. It is written along with the status line if `write_head` or `write_head_async` is invoked.

```
res.status = Soup.Status.OK;
res.reason_phrase = "Everything Went Well"
```

To obtain final status line sent to the user agent, use the `wrote_status_line` signal.

```
res.wrote_status_line.connect ((http_version, status, reason_phrase) => {
    if (200 <= status < 300) {
        // assuming a success
    }
});
```

Headers

The response headers can be accessed as a libsoup-2.4/Soup.MessageHeaders from the `headers` property.

```
res.headers.set_content_type ("text/plain", null);
```

Headers can be written in the response synchronously by invoking `write_head` or asynchronously with `write_head_async`.

```
res.write_head_async.begin (Priority.DEFAULT, null, () => {
    // produce the body...
});
```

Warning: Once written, any modification to the `headers` object will be ignored.

The `head_written` property can be tested to see if it's already the case, even though a well written application should assume that already.

```
if (!res.head_written) {
    res.headers.set_content_type ("text/html", null);
}
```

Since headers can still be modified once written, the `wrote_headers` signal can be used to obtain definitive values.

```
res.wrote_headers (() => {
    foreach (var cookie in res.cookies) {
        message (cookie.to_set_cookie_header ());
    }
});
```

Body

The body of a response is accessed through the `body` property. It inherits from gio-2.0/GLib.OutputStream and provides synchronous and asynchronous streaming capabilities.

The response body is automatically closed following a RAII pattern whenever the `Response` object is disposed.

Note that a reference to the body is not sufficient to maintain the inner *Connection* alive: a reference to either the *Request* or response be maintained.

You can still close the body early as it can provide multiple advantages:

- avoid further and undesired read or write operation
- indicate to the user agent that the body has been fully sent

Expand

New in version 0.3.

To deal with fixed-size body, `expand`, `expand_bytes`, `expand_utf8` and `expand_file` utilities as well as their respective asynchronous versions are provided.

It will automatically set the `Content-Length` header to the size of the provided buffer, write the response head and pipe the buffer into the body stream and close it properly.

```
res.expand_utf8 ("Hello world!");
```

Filtering

One common operation related to stream is filtering. gio-2.0/GLib.FilterOutputStream and gio-2.0/GLib.ConverterOutputStream provide, by composition, many filters that can be used for:

- compression and decompression (gzip, deflate, compress, ...)
- charset conversion
- buffering
- writting data

VSGI also provides its own set of *Converters* which cover parts of the HTTP/1.1 specifications such as chunked encoding.

```
var body = new ConverterOutputStream (res.body,
                                      new CharsetConverter (res.body, "iso-8859-1",
↪"utf-8"));

return body.write_all ("Omelette du fromâge!", null);
```

Additionally, some filters are applied automatically if the `Transfer-Encoding` header is set. The obtained gio-2.0/GLib.OutputStream will be wrapped appropriately so that the application can transparently produce its output.

```
res.headers.append ("Transfer-Encoding", "chunked");
return res.body.write_all ("Hello world!".data, null);
```

Conversion

New in version 0.3.

The body may be converted, see *Converters* for more details.

Tee

New in version 0.3.

The response body can be splitted pretty much like how the `tee` UNIX utility works. All further write operations will be performed as well on the passed stream, making it possible to process the payload sent to the user agent.

The typical use case would be to implement a file-based cache that would tee the produced response body into a key-based storage.

```
var cache_key   = Checksum.compute_for_string (ChecksumType.SHA256, req.uri.to_string␣
↪());
var cache_entry = File.new_for_path ("cache/%s".printf (cache_key));

if (cache_entry.query_exists ()) {
    return res.body.splice (cache_entry.read ());
} else {
    res.tee (cache_entry.create (FileCreateFlags.PRIVATE));
}

res.expand_utf8 ("Hello world!");
```

End

New in version 0.3.

To properly close the response, writing headers if missing, `end` is provided:

```
res.status = Soup.Status.NO_CONTENT;
res.end ();
```

To produce a message before closing, favour `extend` utilities.

Cookies

Cookies are stored in *Request* and *Response* headers as part of the HTTP protocol.

Utilities are provided to perform basic operations based on libsoup-2.4/Soup.Cookie as those provided by libsoup-2.4 requires a libsoup-2.4/Soup.Message, which is not common to all implementations.

- extract cookies from request headers
- find a cookie by its name
- marshall cookies for request or response headers (provided by libsoup-2.4)

Extract cookies

Cookies can be extracted as a singly-linked list from a *Request* or *Response* their order of appearance (see libsoup-2.4/Soup.MessageHeaders.get_list for more details).

The `Request.cookies` property will extract cookies from the `Cookie` headers. Only the `name` and `value` fields will be filled as it is the sole information sent by the client.

```
var cookies = req.cookies;
```

The equivalent property exist for *Response* and will extract the `Set-Cookie` headers instead. The corresponding *Request* URI will be used for the cookies origin.

```
var cookies = res.cookies;
```

The extracted cookies can be manipulated with common glib-2.0/GLib.SList operations. However, they must be written back into the *Response* for the changes to be effective.

Warning: Cookies will be in their order of appearance and glib-2.0/SList.reverse should be used prior to perform a lookup that respects precedence.

```
cookies.reverse ();

for (var cookie in cookies)
    if (cookie.name == "session")
        return cookie;
```

Lookup a cookie

You can lookup a cookie by its name from a *Request* using `lookup_cookie`, `null` is returned if no such cookies can be found.

Warning: Although this is not formally specified, cookies name are considered as being case-sensitive by `CookieUtils` utilities.

If it's signed (recommended for sessions), the equivalent `lookup_signed_cookie` exists.

```
string? session_id;
var session = req.lookup_signed_cookie ("session", ChecksumType.SHA512, "secret".data,
↪ out session_id);
```

Marshall a cookie

libsoup-2.4 provides a complete implementation with the libsoup-2.4/Soup.Cookie class to represent and marshall cookies for both request and response headers.

The newly created cookie can be sent by adding a `Set-Cookie` header in the *Response*.

```
var cookie = new Cookie ("name", "value", "0.0.0.0", "/", 60);
res.headers.append ("Set-Cookie", cookie.to_set_cookie_header ());
```

Sign and verify

Considering that cookies are persisted by the user agent, it might be necessary to sign to prevent forgery. `CookieUtils.sign` and `CookieUtils.verify` functions are provided for the purposes of signing and verifying cookies.

> **Warning:** Be careful when you choose and store the secret key. Also, changing it will break any previously signed cookies, which may still be submitted by user agents.

It's up to you to choose what hashing algorithm and secret: `SHA512` is generally recommended.

The `CookieUtils.sign` utility will sign the cookie in-place. It can then be verified using `CookieUtils.verify`.

The value will be stored in the output parameter if the verification process is successful.

```
CookieUtils.sign (cookie, ChecksumType.SHA512, "secret".data);

string value;
if (CookieUtils.verify (cookie, ChecksumType.SHA512, "secret.data", out value)) {
    // cookie's okay and the original value is stored in value
}
```

The signature is computed in a way it guarantees that:

- we have produced the value
- we have produced the name and associated it to the value

The algorithm is the following:

```
HMAC (checksum_type, key, HMAC (checksum_type, key, value) + name) + value
```

The verification process does not handle special cases like values smaller than the hashing: cookies are either signed or not, even if their values are incorrectly formed.

If well-formed, cookies are verified in constant-time to prevent time-based attacks.

Converters

VSGI provide stream utilities named converters to convert data according to modern Web standards.

These are particularly useful to encode and recode request and response bodies in VSGI implementations.

GLib provide default converters for charset conversion and zlib compression. These can be used to compress the message bodies and convert the string encoding transparently.

- gio-2.0/GLib.CharsetConverter
- gio-2.0/GLib.ZLibCompressor
- gio-2.0/GLib.ZLibDecompressor

Converters can be applied on both the *Request* and *Response* object using the `convert` method.

```
res.headers.append ("Content-Encoding", "gzip");
res.convert (new ZlibCompressor (ZlibCompressorFormat.GZIP));
```

Warning: The `Content-Encoding` header must be adapted to reflect the current set of encodings applied (or unapplied) on the payload.

Since conversion typically affect the resulting size of the payload, the `Content-Length` header must be set appropriately. To ease that, the new value can be specified as second argument. Note that `-1` is used to describe an undetermined length.

```
res.convert (new CharsetConverter ("UTF-8", "ascii"), res.headers.get_content_length␣
↪());
```

The default, which apply in most cases, is to remove the `Content-Length` header and thus describe an undetermined length.

Server

Server provide HTTP technologies integrations under a common interface.

HTTP

libsoup-2.4 provides a libsoup-2.4/Soup.Server that you can use to test your application or spawn workers in production.

```
using Valum;

var https_server = Server.new ("http", https: true);
```

Parameters

The implementation provides most parameters provided by libsoup-2.4/Soup.Server.

Parameter	Default	Description
`interface`	3003	listening interface if using libsoup's old server API (<2.48)
`https`	disabled	listen for https connections rather than plain http
`tls-certificate`	none	path to a file containing a PEM-encoded certificate
`server-header`	disabled	value to use for the "Server" header on Messages processed by this server.
`raw-paths`	disabled	percent-encoding in the Request-URI path will not be automatically decoded

Notes

- if `--https` is specified, you must provide a TLS certificate along with a private key

CGI

CGI is a very simple process-based protocol that uses commonly available process resources:

- environment variables
- standard input stream for the *Request*
- standard output stream for the *Response*

Warning: The CGI protocol expects the response to be written in the standard output: writting there will most surely corrupt the response.

The `VSGI.CGI` namespace provides a basis for its derivatives protocols such as *FastCGI* and *SCGI* and can be used along with any HTTP server.

The interpretation of the environment prioritize the CGI/1.1 specification while providing fallbacks with values we typically found like `REQUEST_URI`.

Since a process is spawned per request and exits when the latter finishes, scheduled asynchronous tasks will not be processed.

If your task involve the *Request* or *Response* in its callback, the connection and thus the process will be kept alive as long as necessary.

```
public class App : Handler {

    public override bool handle (Request req, Response res) {
        Timeout.add (5000, () => {
            res.expand_utf8 ("Hello world!");
            return Source.REMOVE;
        });
        return true;
    }
}

Server.new ("cgi", handler: new App ()).run ();
```

lighttpd

There is an example in `examples/cgi` providing a sample lighttpd configuration file. Once launched, the application can be accessed at the following address: http://127.0.0.1:3003/cgi-bin/app/.

```
lighttpd -D -f examples/cgi/lighttpd.conf
```

FastCGI

FastCGI is a binary protocol that multiplexes requests over a single connection.

VSGI uses fcgi/FastCGI under the hood to provide a compliant implementation. See *Installation* for more information about the framework dependencies.

The whole request cycle is processed in a thread and dispatched in the main context, so it's absolutely safe to use shared states.

By default, the FastCGI implementation listens on the file descriptor `0`, which is conventionally the case when the process is spawned by an HTTP server.

The implementation only support file descriptors, UNIX socket paths and IPv4 addresses on the loopback interface.

Parameters

The only available parameter is `backlog` which set the depth of the listen queue when performing the `accept` system call.

```
var fastcgi_server = Server.new ("fastcgi", backlog: 1024);
```

Lighttpd

Lighttpd can be used to develop and potentially deploy your application. More details about the FastCGI module are provided in their wiki.

```
server.document-root = var.CWD + "/build/examples/fastcgi"
server.port = 3003

server.modules += ( "mod_fastcgi" )

fastcgi.server = (
    "" => (
        "valum" => (
            "socket"            => var.CWD + "/valum.sock",
            "bin-path"          => var.CWD + "/build/examples/fastcgi/fastcgi",
            "check-local"       => "disable",
            "allow-x-send-file" => "enable"
        )
    )
)
```

You can run the FastCGI example with Lighttpd:

```
./waf configure build --enable-examples
lighttpd -D -f examples/fastcgi/lighttpd.conf
```

Apache

Under Apache, there are two mods available: `mod_fcgid` is more likely to be available as it is part of Apache and `mod_fastcgi` is developed by those who did the FastCGI specifications.

- mod_fcgid
- mod_fastcgi

```
<Location />
    FcgidWrapper /usr/libexec/app
</Location>
```

Apache 2.5 provide a mod_proxy_fcgi, which can serve FastCGI instance like it currently does for *SCGI* using the `ProxyPass` directive.

```
ProxyPass fcgi://localhost:3003
```

Nginx

Nginx expects a process to be already spawned and will communicate with it on a TCP port or a UNIX socket path. Read more about ngx_http_fastcgi_module.

```
location / {
    fastcgi_pass 127.0.0.1:3003;
}
```

If possible, it's preferable to spawn processes locally and serve them through a UNIX sockets. It is safer and much more efficient considering that requests are not going through the whole network stack.

```
location / {
    fastcgi_pass unix:/var/run/app.sock;
}
```

To spawn and manage a process, it is recommended to use a systemd unit and socket. More details are available in Lighttpd wiki. Otherwise, it's possible to use the spawn-fcgi tool.

SCGI

SCGI (Simple Common Gateway Interface) is a stream-based protocol that is particularly simple to implement.

Note: SCGI is the recommended implementation and should be used when available as it takes the best out of GIO asynchronous API.

The implementation uses a gio-2.0/GLib.SocketService and processes multiple requests using non-blocking I/O.

Parameters

The only available parameter is `backlog` which set the depth of the listen queue when performing the `accept` system call.

```
var scgi_server = Server.new ("scgi", backlog: 1024);
```

Lighttpd

Similarly to *FastCGI*, Lighttpd can be used to spawn and serve SCGI processes.

```
server.document-root = var.CWD + "/build/examples/scgi"
server.port = 3003

server.modules += ( "mod_scgi" )

scgi.server = (
    "" => (
        "valum" => (
            "socket"      => var.CWD + "/valum.sock",
            "bin-path"    => var.CWD + "/build/examples/scgi/scgi",
            "check-local" => "disable",
        )
    )
)
```

Apache

Apache can serve SCGI instances with mod_proxy_scgi.

```
ProxyPass / scgi://[::]:3003
```

Nginx

Nginx support the SCGI protocol with ngx_http_scgi_module and can only pass requests over TCP/IP and UNIX domain sockets.

```
location / {
    scgi_pass [::]:3003;
}
```

Server implementations are dynamically loaded using gmodule-2.0/GLib.Module. It makes it possible to define its own implementation if necessary.

To load an implementation, use the `Server.new` factory, which can receive GObject-style arguments as well.

```
var cgi_server = Server.new ("cgi");

if (cgi_server == null) {
    assert_not_reached ();
}

cgi_server.set_application_callback ((req, res) => {
    return res.expand_utf8 ("Hello world!");
});
```

Custom implementation

For more flexibility, the `ServerModule` class allow a more fine-grained control for loading a server implementation. If non-null, the `directory` property will be used to retrieve the implementation from the given path instead of standard locations.

The computed path of the shared library is available from `path` property, which can be used for debugging purposes.

The shared library name must conform to `vsgi-<name>` with the appropriate prefix and extension. For instance, on GNU/Linux, the *CGI* module is stored in `${prefix}/${libdir}/vsgi-0.3/servers/libvsgi-cgi.so`.

```
var directory  = "/usr/lib64/vsgi-0.3/servers";
var cgi_module = new ServerModule (directory, "cgi");

if (!cgi_module.load ()) {
    error ("could not load 'cgi' from '%s'", cgi_module.path);
}

var server = Object.new (cgi_module.server_type);
```

Unloading a module is not necessary: once initially loaded, a use count is kept so that it can be loaded on need or unloaded if not used.

> **Warning:** Since a `ServerModule` cannot be disposed (see gobject-2.0/GLib.TypeModule), one must be careful of how its reference is being handled. For instance, `Server.new` keeps track of requested implementations and persist them forever.

Mixing direct usages of `ServerModule` and `Server.@new` (and the likes) is not recommended and will result in undefined behaviours if an implementation is loaded more than once.

Parameters

Each server implementation expose its own set of parameters via GObject properties which are passed using the provided static constructors:

```
var https_server = Server.new ("http", https: true);
```

More details on available parameters are presented in implementation-specific documents.

Listening

Once initialized, a server can be made ready to listen with `listen` and `listen_socket`. Implementations typically support listening from an arbitrary number of interfaces.

If the provided parameters are not supported, a gio-2.0/GLib.IOError.NOT_SUPPORTED will be raised.

The `listen` call is designed to make the server listen on a gio-2.0/GLib.SocketAddress such as gio-2.0/GLib.InetSocketAddress and gio-2.0/GLib.UnixSocketAddress.

```
server.listen (new InetSocketAddress (new InetAddress.loopback (SocketFamily.IPV4),␣
↪3003));
```

It's also possible to pass `null` such that the default interface for the implementation will be used.

```
server.listen (); // default is 'null'
```

The `listen_socket` call make the server listen on an existing socket or file descriptor if passed through *GLib.Socket.from_fd*.

```
server.listen_socket (new Socket.from_fd (0));
```

Serving

Once ready, either call `Server.run` or launch a glib-2.0/GLib.MainLoop to start serving incoming requests:

```
using GLib;
using VSGI;

var server = Server.new ("http");

server.listen (new InetSocketAddress (new InetAddress (SocketFamily.IPV4), 3003));

new MainLoop ().run (); // or server.run ();
```

Forking

To achieve optimal performances on a multi-core architecture, VSGI support forking at the server level.

Warning: Keep in mind that the `fork` system call will actually copy the whole process: no resources (e.g. lock, memory) can be shared unless inter-process communication is used.

The `Server.fork` call is used for that purpose:

```
using GLib;
using VSGI;

var server = Server.new ("http");

server.listen (new InetSocketAddress (new InetAddress.loopback (SocketFamily.IPV4),␣
↪3003));

server.fork ();

new MainLoop ().run ();
```

It is recommended to fork only through that call since implementations such as *CGI* are not guaranteed to support it and will gently fallback on doing nothing.

Application

The `VSGI.Application` class provide a nice cushion around `Server` that deals with pretty logging and CLI argument parsing. The `Server.run` function is a shorthand to create and run an application.

```
using VSGI;

public int main (string[] args) {
    var server = Server.new ("http");
    return new Application (server).run (args);
}
```

CLI

The following options are made available:

Option	Default	Description
`--forks`	none	number of forks to create
`--address`	none	listen on each addresses
`--port`	none	listen on each ports, '0' for random
`--socket`	none	listen on each UNIX socket paths
`--any`	disabled	listen on any address instead of only from the loopback interface
`--ipv4-only`	disabled	listen only to IPv4 interfaces
`--ipv6-only`	disabled	listen only on IPv6 interfaces
`--file-descriptor`	none	listen on each file descriptors

If none of `--address`, `--port`, `--socket` nor `--file-descriptor` flags are provided, it will fallback on the default listening interface for the implementation.

The `--address` flag uses gio-2.0/GLib.NetworkAddress.parse under the hood, which properly interpret IPv4 and IPv6 addresses. It will also resolve domains and parse ports. If no port is provided, a random one will be used.

The default when `--port` is provided is to listen on both IPv4 and IPv6 interfaces, or just IPv4 if IPv6 is not supported.

Use the `--help` flag to obtain more information about available options.

VSGI produces process-based applications that are able to communicate with various HTTP servers using standardized protocols.

Handler

The entry point of any VSGI application implement the vsgi-0.3/VSGI.Handler abstract class. It provides a function of two arguments: a *Request* and a *Response* that return a boolean indicating if the request has been or will be processed. It may also raise an error.

```
using VSGI;

public class App : Handler {

    public override handle (Request req, Response res) throws Error {
        // process the request and produce the response...
        return true;
    }
}

Server.new ("http", handler: new App ()).run ();
```

If a handler indicate that the request has not been processed, it's up to the server implementation to decide what will happen.

From now on, examples will consist of vsgi-0.3/VSGI.Handler.handle content to remain more concise.

Error handling

New in version 0.3.

At any moment, an error can be raised and handled by the server implementation which will in turn teardown the connection appropriately.

```
throw new IOError.FAILED ("some I/O failed");
```

Asynchronous processing

The asynchronous processing model follows the RAII pattern and wraps all resources in a connection that inherits from gio-2.0/GLib.IOStream. It is therefore important that the said connection is kept alive as long as the streams are being used.

The *Request* holds a reference to the said connection and the *Response* indirectly does as it holds a reference to the request. Generally speaking, holding a reference on any of these two instances is sufficient to keep the streams usable.

> **Warning:** As VSGI relies on reference counting to free the resources underlying a request, you must keep a reference to either the *Request* or *Response* during the processing, including in asynchronous callbacks.

It is important that the connection persist until all streams operations are done as the following example demonstrates:

```
res.body.write_async.begin ("Hello world!",
                            Priority.DEFAULT,
                            null,
                            (body, result) => {
    // the response reference will make the connection persist
    var written = res.body.write_async.end (result);
});
```

Dynamic loading

New in version 0.3.

It could be handy to dynamically load handlers the same way *Server* are.

Fortunately, this can be performed with the `HandlerModule` by providing a directory and name for the shared library containing a dynamically loadable application.

```
var module = var new HandlerModule ("<directory>", "<name>");

Server.new ("http", handler: Object.new (module.handler_type)).run ();
```

The only required definition is a `handler_init` symbol that return the type of some `Handler`. In this case, the library should be located in `<directory>/lib<name>.so`, although the actual name is system-dependant.

```
[ModuleInit]
public Type handler_init (TypeModule type_module) {
    return typeof (App);
}

public class App : Handler {

    public bool handle (Request req, Response res) {
        return res.expand_utf8 ("Hello world!");
    }
}
```

Eventually, this will be used to provide a utility to run arbitrary applications with support for live-reloading.

CHAPTER 5

Router

Router is the core component of Valum. It dispatches request to the right handler and processes certain error conditions described in *Redirection and Error*.

The router is constituted of a sequence of `Route` objects which may or may not match incoming requests and perform the process described in their handlers.

Route

The most basic and explicit way of attaching a handler is `Router.route`, which attach the provided `Route` object to the sequence.

```
app.route (new RuleRoute (Method.GET, "/", null, () => {}));
```

Route are simple objects which combine a matching and handling processes. The following sections implicitly treat of route objects such such as `RuleRoute` and `RegexRoute`.

Method

New in version 0.3.

The `Method` flag provide a list of HTTP methods and some useful masks used into route definitions.

Flag	Description
`Method.SAFE`	safe methods
`Method.IDEMPOTENT`	idempotent methods (e.g. `SAFE` and `PUT`)
`Method.CACHEABLE`	cacheable methods (e.g. `HEAD`, `GET` and `POST`)
`Method.ALL`	all standard HTTP methods
`Method.OTHER`	any non-standard HTTP methods
`Method.ANY`	anything, including non-standard methods
`Method.PROVIDED`	indicate that the route provide its methods
`Method.META`	mask for all meta flags like `Method.PROVIDED`

Note: Safe, idempotent and cacheable methods are defined in section 4.2 of RFC 7231.

Using a flag makes it really convenient to capture multiple methods with the `|` binary operator.

```
app.rule (Method.GET | Method.POST, "/", (req, res) => {
    // matches GET and POST
});
```

`Method.GET` is defined as `Method.ONLY_GET | Method.HEAD` such that defining the former will also provide a `HEAD` implementation. In general, it's recommended to check the method in order to skip a body that won't be considered by the user agent.

```
app.get ("/", () => {
    res.headers.set_content_type ("text/plain", null);
    if (req.method == Request.HEAD) {
        return res.end (); // skip unnecessary I/O
    }
    return res.expand_utf8 ("Hello world!");
});
```

To provide only the `GET` part, use `Method.ONLY_GET`.

```
app.rule (Method.ONLY_GET, "/", () => {
    res.headers.set_content_type ("text/plain", null);
    return res.expand_utf8 ("Hello world!");
});
```

Per definition, `POST` is considered cacheable, but if it's not desirable, it may be removed from the mask with the unary `~` operator.

```
app.rule (Method.CACHEABLE & ~Method.POST, "/", () => {
    res.headers.set_content_type ("text/plain", null);
    return res.expand_utf8 ("Hello world!");
});
```

Non-standard method

To handle non-standard HTTP method, use the `Method.OTHER` along with an explicit check.

```
app.method (Method.OTHER, "/rule", (req, res) => {
    if (req.method != "CUSTOM")
        return next ();
});
```

Reverse

New in version 0.3.

Some route implementations can be reversed into URLs by calling `Route.to_url` or the alternative `Route.to_urlv` and `Route.to_url_from_hash`. It may optionally take parameters which, in the case of the rule-based route, correspond to the named captures.

Introspection

The router introspect the route sequence to determine what methods are allowed for a given URI and thus produce a nice `Allow` header. To mark a method as *provided*, the `Method.PROVIDED` flag has to be used. This is automatically done for the helpers and the `Router.rule` function described below.

Additionally, the `OPTIONS` and `TRACE` are automatically handled if not specified for a path. The `OPTIONS` will produce a `Allow` header and `TRACE` will feedback the request into the response payload.

Named route

New in version 0.3.

Few of the helpers provided by the router also accept an additional parameter to name the created route object. This can then be used to generate reverse URLs with `Router.url_for`.

Note: This feature is only support for the rule-based and path-based route implementations.

```
var app = new Router ();

app.get ("/", (req, res) => {
    return res.expand_utf8 ("Hello world! %s".printf (app.url_for ("home")));
}, "home");
```

Likewise to `to_url`, it's possible to pass additional parameters as varidic arguments. The following example show how one can serve relocatable static resources and generate URLs in a Compose template.

```
using Compose.HTML5;
using Valum;
using Valum.Static;

var app = new Router ();

app.get ("/", () => {
    return res.expand_utf8 (
        html (
            head (
                title ("Hello world!"),
                link ("stylesheet",
                      app.url_for ("static",
                                   "path", "bootstrap/dist/css/bootstrap.min.css"))),
            body ()));
});

app.get ("/static/<path>", serve_from_path ("static"), "static");
```

Other helpers are provided to pass a `GLib.HashTable` via `Router.url_for_hash` or explicit varidic arguments via `Router.url_for_valist`.

Note: Vala also support the `:` syntax for passing varidic argument in a key-value style if the key is a glib-2.0/string which is the case for `Router.url_for` and `Route.to_url`.

```
var bootstrap_url = app.url_for ("static", path: "bootstrap/dist/css/bootstrap.min.css
↪");
```

Once

New in version 0.3.

To perform initialization or just call some middleware once, use `Router.once`.

```
Gda.Connection database;

app.once ((req, res, next) => {
    database = new Gda.Connection.from_string ("mysql", ...);
    return next ();
});

app.get ("/", (req, res) => {
    return res.expand_utf8 ("Hello world!");
});
```

Use

New in version 0.3.

The simplest way to attach a handler is `Router.use`, which unconditionally apply the route on the request.

```
app.use ((req, res, next) => {
    var params = new HashTable<string, string> (str_hash, str_equal);
    params["charset"] = "iso-8859-1";
    res.headers.set_content_type ("text/xhtml+xml", params);
    return next ();
});
```

It is typically used to mount a *Middlewares* on the router.

Asterisk

New in version 0.3.

The special `*` URI is handled by the `Router.asterisk` helper. It is typically used along with the `OPTIONS` method to provide a self-description of the Web service or application.

```
app.asterisk (Method.OPTIONS, () => {
    return true;
});
```

Rule

Changed in version 0.3: Rule helpers (e.g. `get`, `post`, `rule`) must explicitly be provided with a leading slash.

The rule syntax has been greatly improved to support groups, optionals and wildcards.

The *de facto* way of attaching handler callbacks is based on the rule system. The `Router.rule` as well as all HTTP method helpers use it.

```
app.rule (Method.ALL, "/rule" (req, res) => {
    return true;
});
```

The syntax for rules is given by the following EBNF grammar:

```
rule      = piece | parameter | group | optional | wildcard, [ rule ];
group     = '(', rule, ')';
optional  = (piece | parameter | group), '?';
wildcard  = '*';
parameter = '<', [ type, ':' ], name, '>'; (* considered as a terminal *)
type      = ? any sequence of word character ?;
name      = ? any sequence of word character ?;
piece     = ? any sequence of URL-encoded character ?;
```

Remarks

- a piece is a single character, so `/users/?` only indicates that the `/` is optional
- the wildcard `*` matches anything, just like the `.*` regular expression

The following table show valid rules and their corresponding regular expressions. Note that rules are matching the whole path as they are automatically anchored.

Rule	Regular expression
`/user`	`^/user$`
`/user/<id>`	`^/user/(?<id>\w+)$`
`/user/<int:id>`	`^/user/(?<id>\d+)$`
`/user(/<int:id>)?`	`^/user(?:/(?<id>\d+))?$`

Types

Valum provides built-in types initialized in the *Router* constructor. The following table details these types and what they match.

Type	Regex	Description
`int`	`\d+`	matches non-negative integers like a database primary key
`string`	`\w+`	matches any word character
`path`	`(?:\.?[\w/-\s/])+`	matches a piece of route including slashes, but not `..`

Undeclared types default to `string`, which matches any word characters.

It is possible to specify or overwrite types using the `types` map in *Router*. This example will define the `path` type matching words and slashes using a regular expression literal.

```
app.register_type ("path", new Regex ("[\w/]+", RegexCompileFlags.OPTIMIZE));
```

If you would like `int` to match negatives integer, you may just do:

```
app.register_type ("int", new Regex ("-?\d+", RegexCompileFlags.OPTIMIZE));
```

Rule parameters are available from the routing context by their name.

```
app.get ("/<controller>/<action>", (req, res, next, context) => {
    var controller = context["controller"].get_string ();
    var action     = context["action"].get_string ();
});
```

Helpers

Helpers for the methods defined in the HTTP/1.1 protocol and the extra `TRACE` methods are included. The path is matched according to the rule system defined previously.

```
app.get ("/", (req, res) => {
    return res.expand_utf8 ("Hello world!");
});
```

The following example deal with a `POST` request providing using libsoup-2.4/Soup.Form to decode the payload.

```
app.post ("/login", (req, res) => {
    var data = Soup.Form.decode (req.flatten_utf8 ());

    var username = data["username"];
    var password = data["password"];

    // assuming you have a session implementation in your app
    var session = new Session.authenticated_by (username, password);

    return true;
});
```

Regular expression

Changed in version 0.3: The regex helper must be provided with an explicit leading slash.

If the rule system does not suit your needs, it is always possible to use regular expression. Regular expression will be automatically scoped, anchored and optimized.

```
app.regex (Method.GET, new Regex ("/home/?", RegexCompileFlags.OPTIMIZE), (req, res)␣
↪=> {
    return res.body.write_all ("Matched using a regular expression.".data, true);
});
```

Named captures are registered on the routing context.

```
app.regex (new Regex ("/(?<word>\w+)", RegexCompileFlags.OPTIMIZE), (req, res, next,␣
↪ctx) => {
    var word = ctx["word"].get_string ();
});
```

Matcher callback

Request can be matched by a simple callback typed by the `MatcherCallback` delegate.

```
app.matcher (Method.GET, (req) => { return req.uri.get_path () == "/home"; }, (req,␣
↪res) => {
    // matches /home
});
```

Scoping

Changed in version 0.3: The scope feature does not include a slash, instead you should scope with a leading slash like shown in the following examples.

Scoping is a powerful prefixing mechanism for rules and regular expressions. Route declarations within a scope will be prefixed by `<scope>`.

The `Router` maintains a scope stack so that when the program flow enter a scope, it pushes the fragment on top of that stack and pops it when it exits.

```
app.scope ("/admin", (admin) => {
    // admin is a scoped Router
    app.get ("/users", (req, res) => {
        // matches /admin/users
    });
});

app.get ("/users", (req, res) => {
    // matches /users
});
```

To literally mount an application on a prefix, see the *Basepath* middleware.

Context

New in version 0.3.

During the routing, states can obtained from a previous handler or passed to the next one using the routing context.

Keys are resolved recursively in the tree of context by looking at the parent context if it's missing.

```
app.get ("/", (req, res, next, context) => {
    context["some key"] = "some value";
    return next ();
});

app.get ("/", (req, res, next, context) => {
    var some_value = context["some key"]; // or context.parent["some key"]
    return return res.body.write_all (some_value.data, null);
});
```

Next

Changed in version 0.3: The `next` continuation does not take the request and response objects as parameter. To perform transformation, see *Converters* and *Middlewares*.

The handler takes a callback as an optional third argument. This callback is a continuation that will continue the routing process to the next matching route.

```
app.get ("/", (req, res, next) => {
    return next (); // keep routing
});

app.get ("/", (req, res) => {
    // this is invoked!
});
```

Warning: The `next` continuation can only be called from within the handler callback. Since it is not maked as `owned`, the reference does not persist beyond the function return.

The `next` continuation can only be called synchronously. This is only temporary and an eventual release will revamp the whole routing when asynchronous delegates will be part of the Vala language (see bug 604827 for details).

Sequence

New in version 0.3.

The *Sequence* middleware should be used to chain handling callbacks.

```
app.get ("/", sequence ((req, res, next) => {
    return next ();
}, (req, res) => {
    return res.expand_utf8 ("Hello world!");
}));
```

Error handling

New in version 0.2.1: Prior to this release, any unhandled error would crash the main loop iteration.

Changed in version 0.3: Error and status codes are now handled with a `catch` block or using the *Status* middleware.

Changed in version 0.3: The default handling is not ensured by the *Basic* middleware.

Changed in version 0.3: Thrown errors are forwarded to VSGI, which process them essentially the same way. See *VSGI* for more details.

Similarly to status codes, errors are propagated in the `HandlerCallback` and `NextCallback` delegate signatures and can be handled in a `catch` block.

```
app.use (() => {
    try {
        return next ();
    } catch (IOError err) {
        res.status = 500;
        return res.expand_utf8 (err.message);
    }
});

app.get ("/", (req, res) => {
```

```
    throw new IOError.FAILED ("I/O failed some some reason.");
});
```

Thrown status code can also be caught this way, but it's much more convenient to use the *Status* middleware.

Subrouting

Since `VSGI.ApplicationCallback` is type compatible with `HandlerCallback`, it is possible to delegate request handling to another VSGI-compliant application.

In particular, it is possible to treat `Router.handle` like any handling callback.

Note: This feature is a key design of the router and is intended to be used for a maximum inter-operability with other frameworks based on VSGI.

The following example delegates all `GET` requests to another router which will process in isolation with its own routing context.

```
var app = new Router ();
var api = new Router ();

// delegate all GET requests to api router
app.get ("*", api.handle);
```

One common pattern with subrouting is to attempt another router and fallback on `next`.

```
var app = new Router ();
var api = new Router ();

app.get ("/some-resource", (req, res) => {
    return api.handle (req, res) || next ();
});
```

Cleaning up route logic

Performing a lot of route bindings can get messy, particularly if you want to split an application several reusable modules. Encapsulation can be achieved by subclassing `Router` and performing initialization in a `construct` block:

```
public class AdminRouter : Router {

    construct {
        rule (Method.GET,                "/admin/user",          view);
        rule (Method.GET | Method.POST, "/admin/user/<int:id>", edit);
    }

    public bool view (Request req, Response res) {
        return render_template ("users", Users.all ());
    }

    public bool edit (Request req, Response res) {
        var user = User.find (ctx["id"]);
```

```
        if (req.method == "POST") {
            user.values (Soup.Form.decode (req.flatten_utf8 ()));
            user.update ();
        }
        return render_template ("user", user);
    }
}
```

Using subrouting, it can be assembled to a parent router given a rule (or any matching process described in this document). This way, incoming request having the `/admin/` path prefix will be delegated to the `admin` router.

```
var app = new Router ();

app.rule (Method.ALL, "/admin/*", new AdminRouter ().handle);
```

The *Basepath* middleware provide very handy path isolation so that the router can be simply written upon the leading `/` and rebased on any basepath. In that case, we can strip the leading `/admin` in router's rules.

```
var app = new Router ();

// captures '/admin/users' and '/admin/user/<int:id>'
app.use (basepath ("/admin", new AdminRouter ().handle));
```

CHAPTER 6

Redirection and Error

Redirection, client and server errors are handled via a simple exception mechanism.

In a `HandlerCallback`, you may throw any of `Informational`, `Success`, `Redirection`, `ClientError` and `ServerError` predefined error domains rather than setting the status and returning from the function.

It is possible to register a handler on the *Router* to handle a specific status code.

```
app.use ((req, res, next) => {
    try {
        return next ();
    } catch (Redirection.PERMANENT red) {
        // handle a redirection...
    }
}));
```

Default handling

Changed in version 0.3: Default handling is not assured by the *Basic* middleware.

The *Router* can be configured to handle raised status by setting the response status code and headers appropriately.

```
app.use (basic ());

app.get ("/", () => {
    throw new ClientError.NOT_FOUND ("The request URI '/' was not found.");
});
```

To handle status more elegantly, see the *Status* middleware.

```
app.use (status (Status.NOT_FOUND, (req, res, next, ctx, err) => {
    // handle 'err' properly...
}));
```

The error message may be used to fill a specific *Response* headers or the response body. The following table describe how the router deal with these cases.

Status	Header	Description
Informational.SWITCHING_PROTOCOLS	Upgrade	Identifier of the protocol to use
Success.CREATED	Location	URL to the newly created resource
Success.PARTIAL_CONTENT	Range	Range of the delivered resource in bytes
Redirection.MOVED_PERMANENTLY	Location	URL to perform the redirection
Redirection.FOUND	Location	URL of the found resource
Redirection.SEE_OTHER	Location	URL of the alternative resource
Redirection.USE_PROXY	Location	URL of the proxy
Redirection.TEMPORARY_REDIRECT	Location	URL to perform the redirection
ClientError.UNAUTHORIZED	WWW-Authenticate	Challenge for authentication
ClientError.METHOD_NOT_ALLOWED	Allow	Comma-separated list of allowed methods
ClientError.UPGRADE_REQUIRED	Upgrade	Identifier of the protocol to use

The following errors does not produce any payload:

- `Information.SWITCHING_PROTOCOLS`
- `Success.NO_CONTENT`
- `Success.RESET_CONTENT`
- `Success.NOT_MODIFIED`

For all other domains, the message will be used as a `text/plain` payload encoded with `UTF-8`.

The approach taken by Valum is to support at least all status defined by libsoup-2.4 and those defined in RFC documents. If anything is missing, you can add it and submit us a pull request.

Informational (1xx)

Informational status are used to provide a in-between response for the requested resource. The *Response* body must remain empty.

Informational status are enumerated in `Informational` error domain.

Success (2xx)

Success status tells the client that the request went well and provide additional information about the resource. An example would be to throw a `Success.CREATED` error to provide the location of the newly created resource.

Successes are enumerated in `Success` error domain.

```
app.get ("/document/<int:id>", (req, res) => {
    // serve the document by its identifier...
});

app.put ("/document", (req, res) => {
    // create the document described by the request
    throw new Success.CREATED ("/document/%u".printf (id));
});
```

Redirection (3xx)

To perform a redirection, you have to throw a `Redirection` error and use the message as a redirect URL. The *Router* will automatically set the `Location` header accordingly.

Redirections are enumerated in `Redirection` error domain.

```
app.get ("/user/<id>/save", (req, res) => {
    var user = User (req.params["id"]);

    if (user.save ())
        throw new Redirection.MOVED_TEMPORAIRLY ("/user/%u".printf (user.id));
});
```

Client (4xx) and server (5xx) error

Like for redirections, client and server errors are thrown. Errors are predefined in `ClientError` and `ServerError` error domains.

```
app.get ("/not-found", (req, res) => {
    throw new ClientError.NOT_FOUND ("The requested URI was not found.");
});
```

Errors in next

The `next` continuation is designed to throw these specific errors so that the *Router* can handle them properly.

```
app.use ((req, res, next) => {
    try {
        return next ();
    } catch (ClientError.NOT_FOUND err) {
        // handle a 404...
    }
});

app.get ("/", (req, res, next) => {
    return next (); // will throw a 404
});

app.get ("/", (req, res) => {
    throw new ClientError.NOT_FOUND ("");
});
```

CHAPTER 7

Middlewares

Middlewares are reusable pieces of processing that can perform various work from authentication to the delivery of a static resource.

Authenticate

The valum-0.3/Valum.authenticate middleware allow one to perform HTTP basic authentications.

It takes three parameters:

- an vsgi-0.3/VSGI.Authentication object described in *HTTP authentication*
- a callback to challenge a user-provided vsgi-0.3/VSGI.Authorization header
- a forward callback invoked on success with the corresponding authorization object

If the authentication fails, a `401 Unauthorized` status is raised with a `WWW-Authenticate` header.

```
app.use (authenticate (new BasicAuthentication ("realm")), (authorization) => {
    return authorization.challenge ("some password");
}, (req, res, next, ctx, username) => {
    return res.expand_utf8 ("Hello %s".printf (username));
});
```

To perform custom password comparison, it is best to cast the `authorization` parameter and access the password directly.

```
public bool authenticate_user (string username, string password) {
    // authenticate the user against the database...
}

app.use (authenticate (new BasicAuthentication ("realm")), (authorization) => {
    var basic_authorization = authorization as BasicAuthorization;
```

```
    return authenticate_user (basic_authorization.username, basic_authorization.
↪password);
});
```

Basepath

The valum-0.3/Valum.basepath middleware allow a better isolation when composing routers by stripping a prefix on the *Request* URI.

The middleware strips and forwards requests which match the provided base path. If the resulting path is empty, it fallbacks to a root `/`.

Error which use their message as a `Location` header are automatically prefixed by the base path.

```
var user = new Router ();

user.get ("/<int:id>", (req, res) => {
    // ...
});

user.post ("/", (req, res) => {
    throw new Success.CREATED ("/5");
});

app.use (basepath ("/user", user.handle));

app.status (Soup.Status.CREATED, (req, res) => {
    assert ("/user/5" == context["message"]);
});
```

If `next` is called while forwarding or an error is thrown, the original path is restored.

```
user.get ("/<int:id>", (req, res, next) => {
    return next (); // path is '/5'
});

app.use (basepath ("/user", user.handle));

app.use ((req, res) => {
    // path is '/user/5'
});
```

One common pattern is to provide a path-based fallback when using the *Subdomain* middleware.

```
app.use (subdomain ("api", api.handle));
app.use (basepath ("/api", api.handle));
```

Basic

New in version 0.3.

Previously know under the name of *default handling*, the valum-0.3/Valum.basic middleware provide a conforming handling of raised status codes as described in the *Redirection and Error* document.

It aims at providing sane defaults for a top-level middleware.

```
app.use (basic ());

app.get ("/", () => {
    throw new Success.CREATED ("/resource/id");
});
```

If an error is caught, it will perform the following tasks:

1. assign an appropriate status code (500 for other errors)
2. setup required headers (eg. *Location* for a redirection)
3. produce a payload based on the message if required and not already used for a header

The payload will have the `text/plain` content type encoded with `UTF-8`.

For privacy and security reason, non-status errors (eg. gio-2.0/GLib.IOError) will not be used for the payload. To enable that for specific errors, it's possible to convert them into into a raised status, preferably a `500 Internal Server Error`.

```
app.use (() => {
    try {
        return next ();
    } catch (IOError err) {
        throw new ServerError.INTERNAL_SERVER_ERROR (err.message);
    }
})
```

Content Negotiation

Negotiating the resource representation is an essential part of the HTTP protocol.

The negotiation process is simple: expectations are provided for a specific header, if they are met, the processing is forwarded with the highest quality value, otherwise a `406 Not Acceptable` status is raised.

```
using Valum.ContentNegotiation;

app.get ("/", negotiate ("Accept", "text/html, text/html+xml",
                         (req, res, next, stack, content_type) => {
    // produce a response based on 'content_type'
}));
```

Or directly by using the default forward callback:

```
app.use (negotiate ("Accept", "text/html"));

// all route declaration may assume that the user agent accept 'text/html'
```

Preference and quality

Additionally, the server can state the quality of each expectation. The middleware will maximize the product of quality and user agent preference with respect to the order of declaration and user agent preferences if it happens to be equal.

If, for instance, you would serve a XML document that is just poorly converted from a JSON source, you could state it by giving it a low `q` value. If the user agent as a strong preference the former and a low preference for the latter

(eg. `Accept: text/xml; application/json; q=0.1`)), it will be served the version with the highest product (eg. `0.3 * 1 > 1 * 0.3`).

```
app.get ("/", negotiate ("Accept", "application/json;, text/xml; q=0.3",
                         (req, res, next, stack, content_type) => {
    // produce a response based on 'content_type'
}));
```

Error handling

The *Status* middleware may be used to handle the possible `406 Not Acceptable` error raised if no expectation can be satisfied.

```
app.use (status (Soup.Status.NOT_ACCEPTABLE, () => {
    // handle '406 Not Acceptable' here
}));

app.use (negotiate ("Accept", "text/xhtml; text/html", () => {
    // produce appropriate resource
}));
```

Custom comparison

A custom comparison function can be provided to valum-0.3/Valum.negotiate in order to handle wildcards and other edge cases. The user agent pattern is the first argument and the expectation is the second.

Warning: Most of the HTTP/1.1 specification about headers is case-insensitive, use libsoup-2.4/Soup.str_case_equal to perform comparisons.

```
app.use (negotiate ("Accept",
                    "text/xhtml",
                    () => { return true; },
                    (a, b) => {
    return a == "*" || Soup.str_case_equal (a, b);
});
```

Helpers

For convenience, helpers are provided to handle common headers:

Middleware	Header	Edge cases
`accept`	`Content-Type`	`*/*`, `type/*` and `type/subtype1+subtype2`
`accept_charset`	`Content-Type`	`*`
`accept_encoding`	`Content-Encoding`	`*`
`accept_language`	`Content-Language`	missing language type
`accept_ranges`	`Content-Ranges`	none

The valum-0.3/Valum.accept middleware will assign the media type and preserve all other parameters.

If multiple subtypes are specified (e.g. `application/vnd.api+json`), the middleware will check if the subtypes accepted by the user agent form a subset. This is useful if you serve a specified JSON document format to a client which only state to accept JSON and does not care about the specification itself.

```
accept ("text/html; text/xhtml", (req, res, next, ctx, content_type) => {
    switch (content_type) {
        case "text/html":
            return produce_html ();
        case "text/xhtml":
            return produce_xhtml ();
    }
});
```

The valum-0.3/Valum.accept_encoding middleware will convert the *Response* if it's either `gzip` or `deflate`.

```
accept ("gzip; deflate", (req, res, next, ctx, encoding) => {
    res.expand_utf8 ("Hello world! (compressed with %s)".printf (encoding));
});
```

The valum-0.3/Valum.accept_charset middleware will set the `charset` parameter of the `Content-Type` header, defaulting to `application/octet-stream` if undefined.

Decode

The valum-0.3/Valum.decode middleware is used to unapply various content codings.

```
app.use (decode ());

app.post ("/", (req, res) => {
    var posted_data = req.flatten_utf8 ();
});
```

It is typically put at the top of an application.

Encoding	Action
deflate	gio-2.0/GLib.ZlibDecompressor
gzip and x-gzip	gio-2.0/GLib.ZlibDecompressor
identity	nothing

If an encoding is not supported, a `501 Not Implemented` is raised and remaining encodings are *reapplied* on the request.

To prevent this behavior, the valum-0.3/Valum.DecodeFlags.FORWARD_REMAINING_ENCODINGS flag can be passed to forward unsupported content codings.

```
app.use (decode (DecodeFlags.FORWARD_REMAINING_ENCODINGS));

app.use (() => {
    if (req.headers.get_one ("Content-Encoding") == "br") {
        req.headers.remove ("Content-Encoding");
        req.convert (new BrotliDecompressor ());
    }
    return next ();
});

app.post ("/", (req, res) => {
    var posted_data = req.flatten_utf8 ();
});
```

Safely

Yet very simple, the valum-0.3/Valum.safely middleware provide a powerful way of discovering possible error conditions and handle them locally.

Only status defined in *Redirection and Error* are leaked: the compiler will warn for all other unhandled errors.

```
app.get ("/", safely ((req, res, next, ctx) => {
    try {
        res.expand_utf8 ("Hello world!");
    } catch (IOError err) {
        critical (err.message);
        return false;
    }
});
```

Sequence

New in version 0.3.

The valum-0.3/Valum.sequence middleware provide a handy way of chaining middlewares.

```
app.post ("/", sequence (decode (), (req, res) => {
    // handle decoded payload
}));
```

To chain more than two middlewares, one can chain a middleware with a sequence.

```
app.get ("/admin", sequence ((req, res, next) => {
    // authenticate user...
    return next ();
}, sequence ((req, res, next) => {
    // produce sensitive data...
    return next ();
}, (req, res) => {
    // produce the response
})));
```

Vala does not support varidic delegate arguments, which would be much more convenient to describe a sequence.

Server-Sent Events

Valum provides a middleware for the HTML5 Server-Sent Events protocol to stream notifications over a persistent connection.

The valum-0.3/Valum.ServerSentEvents.stream_events function creates a handling middleware and provide a valum-0.3/Valum.ServerSentEvents.SendEventCallback callback to transmit the actual events.

```
using Valum;
using Valum.ServerSentEvents;

app.get ("sse", stream_events ((req, send) => {
    send (null, "some data");
}));
```

```
var eventSource = new EventSource ("/sse");

eventSource.onmessage = function(message) {
    console.log (message.data); // displays 'some data'
};
```

Multi-line messages

Multi-line messages are handled correctly by splitting the data into into multiple `data:` chunks.

```
send (null, "some\ndata");
```

```
data: some
data: data
```

Static Resource Delivery

Middlewares in the valum-0.3/Valum.Static namespace ensure delivery of static resources.

```
using Valum.Static;
```

As of convention, all middleware use the `path` context key to resolve the resource to be served. This can easily be specified using a rule parameter with the `path` type.

For more flexibility, one can compute the `path` value and pass the control with `next`. The following example obtains the key from the HTTP query:

```
app.get ("/static", sequence ((req, res, next, ctx) => {
    ctx["path"] = req.lookup_query ("path") ?? "index.html";
    return next ();
}, serve_from_file (File.new_for_uri ("resource://"))));
```

If a `HEAD` request is performed, the payload will be omitted.

File backend

The valum-0.3/Valum.Static.serve_from_file middleware will serve resources relative to a gio-2.0/GLib.File instance.

```
app.get ("/static/<path:path>", serve_from_file (File.new_for_path ("static")));
```

To deliver from the global resources, use the `resource://` scheme.

```
app.get ("/static/<path:path>", serve_from_file (File.new_for_uri ("resource://static
↪")));
```

Before being served, each file is forwarded to make it possible to modify headers more specifically or raise a last-minute error.

Once done, invoke the `next` continuation to send over the content.

```
app.get ("/static/<path:path>", serve_from_file (File.new_for_path ("static"),
                                                 ServeFlags.NONE,
                                                 (req, res, next, ctx, file) => {
    var user = ctx["user"] as User;
    if (!user.can_access (file)) {
        throw new ClientError.FORBIDDEN ("You cannot access this file.")
    }
    return next ();
}));
```

Helpers

Two helpers are provided for File-based delivery: valum-0.3/Valum.Static.serve_from_path and valum-0.3/Valum.Static.serve_from_uri.

```
app.get ("/static/<path:path>", serve_from_path ("static/<path:path>"));

app.get ("/static/<path:path>", serve_from_uri ("static/<path:path>"));
```

Resource backend

The valum-0.3/Valum.Static.serve_from_resource middleware is provided to serve a resource bundle (see gio-2.0/GLib.Resource) from a given prefix. Note that the prefix must be a valid path, starting and ending with a slash / character.

```
app.get ("/static/<path:path>", serve_from_resource (Resource.load ("resource"),
                                                     "/static/"));
```

Compression

To compress static resources, it is best to negotiate a compression encoding with a *Content Negotiation* middleware: body stream and headers will be set properly if the encoding is supported.

Using the `identity` encoding provide a fallback in case the user agent does not want compression and prevent a `406 Not Acceptable` from being raised.

```
app.get ("/static/<path:path>", sequence (accept_encoding ("gzip, deflate, identity"),
                                          serve_from_path ("static")));
```

Content type detection

The middlewares will detect the content type based on the file name and a lookup on its content.

Content type detection, based on the file name and a small data lookup, is performed with GLib.ContentType.

Deal with missing resources

If a resource is not available (eg. the file does not exist), the control will be forwarded to the next route.

One can use that behaviour to implement a cascading failover with the *Sequence* middleware.

```
app.get ("/static/<path:path", sequence (serve_from_path ("~/.local/app/static"),
                                         serve_from_path ("/usr/share/app/static")));
```

To generate a `404 Not Found`, just raise a valum-0.3/Valum.ClientError.NOT_FOUND as described in *Redirection and Error*.

```
app.use (basic ());

app.get ("/static/<path:path>", sequence (serve_from_uri ("resource://"),
                                          (req, res, next, ctx) => {
    throw new ClientError.NOT_FOUND ("The static resource '%s' were not found.",
                                     ctx["path"]);
}));
```

Options

Options are provided as flags from the valum-0.3/Valum.Static.ServeFlags enumeration.

ETag

If the valum-0.3/Valum.Static.ServeFlags.ENABLE_ETAG is specified, a checksum of the resource will be generated in the `ETag` header.

If set and available, it will have precedence over valadoc:*valum-0.3/Valum.Static.ServeFlags.ENABLE_LAST_MODIFIED* described below.

Last-Modified

Unlike `ETag`, this caching feature is time-based and will indicate the last modification on the resource. This is only available for some File backend and will fallback to `ETag` if enabled as well.

Specify the valum-0.3/Valum.Static.ServeFlags.ENABLE_LAST_MODIFIED to enable this feature.

X-Sendfile

If the application run behind a HTTP server which have access to the resources, it might be preferable to let it serve them directly with valum-0.3/Valum.Static.ServeFlags.X_SENDFILE.

```
app.get ("/static/<path:path>", serve_from_path ("static", ServeFlags.X_SENDFILE));
```

If files are not locally available, they will be served directly.

Public caching

The valum-0.3/Valum.Static.ServeFlags.ENABLE_CACHE_CONTROL_PUBLIC let intermediate HTTP servers cache the payload by attaching a `Cache-Control: public` header to the response.

Expose missing permissions

The valum-0.3/Valum.Static.ServeFlags.FORBID_ON_MISSING_RIGHTS will trigger a `403 Forbidden` if rights are missing to read a file. This is not a default as it may expose information about the existence of certain files.

Status

Thrown status codes (see *Redirection and Error*) can be handled with the valum-0.3/Valum.status middleware.

The received *Request* and *Response* object are in the same state they were when the status was thrown. An additional parameter provide access to the actual glib-2.0/GLib.Error object.

```
app.use (status (Soup.Status.NOT_FOUND, (req, res, next, context, err) => {
    // produce a 404 page...
    var message = err.message;
});
```

To jump to the next status handler found upstream in the routing queue, just throw the error. If the error can be resolved, you might want to try `next` once more.

```
app.status (Soup.Status.NOT_FOUND, (req, res) => {
    res.status = 404;
    return res.expand_utf8 ("Not found!");
});

app.status (Soup.Status.NOT_FOUND, (req, res, next, ctx, err) => {
    return next (); // try to route again or jump upstream
});

app.use (() => {
    throw new ClientError.NOT_FOUND ("");
});
```

If an error is not handled, it will eventually be caught by the default status handler, which produce a minimal response.

```
// turns any 404 into a permanent redirection
app.status (Soup.Status.NOT_FOUND, (req, res) => {
    throw new Redirection.PERMANENT ("http://example.com");
});
```

Subdomain

The valum-0.3/Valum.subdomain middleware matches *Request* which subdomain is conform to expectations.

Note: Domains are interpreted in their semantical right-to-left order and matched as suffix.

The pattern is specified as the first argument. It may contain asterisk `*` which specify that any supplied label satisfy that position.

```
app.use (subdomain ("api", (req, res) => {
    // match domains like 'api.example.com' and 'v1.api.example.com'
}));

app.use (subdomain ("\*.user", (req, res) => {
    // match at least two labels: the first can be anything and the second
    // is exactly 'user'
}));
```

The matched subdomain labels are extracted and passed by parameter.

```
app.use (subdomain ("api", (req, res, next, ctx, subdomains) => {
    // 'subdomains' could be 'api' or 'v1.api'
}));
```

This middleware can be used along with subrouting to mount any *Router* on a specific domain pattern.

```
var app = new Router ();
var api = new Router ();

app.use (subdomain ("api", api.handle));
```

Strict

There is two matching mode: loose and strict. The loose mode only expect the request to be performed on a suffix-compatible hostname. For instance, `api` would match `api.example.com` and `v1.api.example.com` as well.

To prevent this and perform a _strict_ match, simply specify `true` the second argument. The domain of the request will have to supply exactly the same amount of labels matching the expectations.

```
// match every request exactly from 'api.*.*'
app.use (subdomain ("api", api.handle, true));
```

Skip labels

By default, the two first labels are ignored since Web applications are typically served under two domain levels (eg. example.com). If it's not the case, the number of skipped labels can be set to any desirable value.

```
// match exactly 'api.example.com'
app.use (subdomain ("api.example.com", api.handle, true, 0));
```

The typical way of declaring them involve closures. It is parametrized and returned to perform a specific task:

```
public HandlerCallback middleware (/* parameters here */) {
    return (req, res, next, ctx) => {
        var referer = req.headers.get_one ("Referer");
        ctx["referer"] = new Soup.URI (referer);
        return next ();
    };
}
```

The following example shows a middleware that provide a compressed stream over the *Response* body.

```
app.use ((req, res, next) => {
    res.headers.append ("Content-Encoding", "gzip");
    res.convert (new ZLibCompressor (ZlibCompressorFormat.GZIP));
    return next ();
});

app.get ("/home", (req, res) => {
    return res.expand_utf8 ("Hello world!"); // transparently compress the output
});
```

If this is wrapped in a function, which is typically the case, it can even be used directly from the handler.

```
HandlerCallback compress = (req, res, next) => {
    res.headers.append ("Content-Encoding", "gzip");
    res.convert (new ZLibCompressor (ZlibCompressorFormat.GZIP));
    return next ();
};

app.get ("/home", compress);

app.get ("/home", (req, res) => {
    return res.expand_utf8 ("Hello world!");
});
```

Alternatively, a middleware can be used directly instead of being attached to a valum-0.3/Valum.Route, the processing will happen in a valum-0.3/Valum.NextCallback.

```
app.get ("/home", (req, res, next, context) => {
    return compress (req, res, (req, res) => {
        return res.expand_utf8 ("Hello world!");
    }, new Context.with_parent (context));
});
```

Forward

New in version 0.3.

One typical middleware pattern is to take a continuation that is forwarded on success (or any other event) with a single value like it's the case for the *Content Negotiation* middlewares.

This can be easily done with valum-0.3/Valum.ForwardCallback. The generic parameter specify the type of the forwarded value.

```
public HandlerCallback accept (string content_types, ForwardCallback<string> forward)
↪{
    return (req, res, next, ctx) => {
        // perform content negotiation and determine 'chosen_content_type'...
        return forward (req, res, next, ctx, chosen_content_type);
    };
}

app.get ("/", accept ("text/xml; application/json", (req, res, next, ctx, content_
↪type) => {
    // produce a response according to 'content_type'...
}));
```

Often, one would simply call the `next` continuation, so a valum-0.3/Valum.forward definition is provided to do that. It is used as a default value for various middlewares such that all the following examples are equivalent:

```
app.use (accept ("text/html" () => {
    return next ();
}));

app.use (accept ("text/html", forward));

app.use (accept ("text/html"));
```

To pass multiple values, it is preferable to explicitly declare them using a delegate.

```
public delegate bool ComplexForwardCallback (Request      req,
                                             Response     res,
                                             NextCallback next,
                                             Context      ctx,
                                             int          a,
                                             int          b) throws Error;
```

CHAPTER 8

Recipes

Recipes are documents providing approaches to common Web development tasks and their potential integration with Valum.

Bump

Bump is a library providing high-level concurrency patterns.

Resource pooling

A resource pool is a structure that maintain and dispatch a set of shared resources.

There's various way of using the pool:

- execute with a callback
- acquire a claim that will release the resource automatically
- acquire a resource that has to be released explicitly

```
using Bump;
using Valum;

var app = new Router ();

var connection_pool = new ResourcePool<Gda.Connection> ();

connection_pool.construct_properties = {
    Property () {}
};

app.get ("/users", (req, res, next) => {
    return connection_pool.execute_async<bool> ((db) => {
        var users = db.execute_select_command ("select * from users");
```

```
        return next ();
    });
});
```

Configuration

There exist various way of providing a runtime configuration.

If you need to pass secrets, take a look at the Libsecret project. It allows one to securely store and retrieve secrets: just unlock the keyring and start your service.

Key file

GLib provide a very handy way of reading and parsing key files, which are widely used across freedesktop specifications.

It should be privileged if the configuration is mostly edited by humans.

```
[app]
public-dir=public

[database]
provider=mysql
connection=
auth=
```

```
using GLib;
using Valum;

var config = new KeyFile ();

config.parse_path ("app.conf");

var app = new Router ();

app.get ("/public/<path:path>",
         Static.serve_from_path (config.get_string ("app", "public-dir")));
```

JSON

The JSON-GLib project provide a really convenient JSON parser and generator.

```
{
    "app": {
        "publicDir": "public"
    },
    "database": {
        "provider": "mysql",
        "connection": "",
        "auth": ""
    }
}
```

```
using Json;
using Valum;

var parser = new Parser ();
parser.parse_from_file ("config.json");

var config = parser.get_root ();

var app = new Router ();

app.get ("/public/<path:path>",
         Static.serve_from_path (config.get_object ("app").get_string_member (
↪"publicDir")));
```

YAML

There is a GLib wrapper around libyaml that makes it more convenient to use. YAML in itself can be seen as a human-readable JSON format.

```
app:
    publicDir: public
database:
    provider: mysql
    connection:
    auth:
```

```
using Valum;
using Yaml;

var config = new Document.from_path ("config.yml").root as Node.Mapping;

var app = new Router ();

app.get ("/public/<path:path>",
         Static.serve_from_path (config.get_mapping ("app").get_scalar ("publicDir").
↪value));
```

Other approaches

The following approaches are a bit more complex to setup but can solve more specific use cases:

- GXml or libxml2
- GSettings for a remote (via DBus) and monitorable configuration
- environment variables via glib-2.0/GLib.Environment utilities
- CLI options (see `VSGI.Server.add_main_option` and `VSGI.Server.handle_local_options`)

JSON

JSON is a popular data format for Web services and json-glib-1.0/Json provide a complete implementation that integrates with the GObject type system.

The following features will be covered in this document with code examples:

- serialize a GObject
- unserialize a GObject
- parse an gio-2.0/GLib.InputStream of JSON like a *Request* body
- generate JSON in a gio-2.0/GLib.OutputStream like a *Response* body

Produce and stream JSON

Using a json-glib-1.0/Json.Generator, you can conveniently produce an JSON object and stream synchronously it in the *Response* body.

```
app.get ("/user/<username>", (req, res) => {
    var user      = new Json.Builder ();
    var generator = new Json.Generator ();

    user.set_member_name ("username");
    user.add_string_value (req.params["username"]);

    generator.root   = user.get_root ();
    generator.pretty = false;

    return generator.to_stream (res.body);
});
```

Serialize GObject

You project is likely to have a model abstraction and serialization of GObject with json-glib-1.0/Json.gobject_serialize is a handy feature. It will recursively build a JSON object from the encountered properties.

```
public class User : Object {
    public string username { construct; get; }

    public User.from_username (string username) {
        // populate the model from the data storage...
    }

    public void update () {
        // persist the model in data storage...
    }
}
```

```
app.get ("/user/<username>", (req, res) => {
    var user      = new User.from_username (req.params["username"]);
    var generator = new Json.Generator ();

    generator.root   = Json.gobject_serialize (user);
    generator.pretty = false;

    return generator.to_stream (res.body);
});
```

With middlewares, you can split the process in multiple reusable steps to avoid code duplication. They are described in the *Router* document.

- fetch a model from a data storage
- process the model with data obtained from a json-glib-1.0/Json.Parser
- produce a JSON response with json-glib-1.0/Json.gobject_serialize

```
app.scope ("/user", (user) => {
    // fetch the user
    app.rule (Method.GET | Method.POST, "/<username>", (req, res, next, context) => {
        var user = new User.from_username (context["username"].get_string ());

        if (!user.exists ()) {
            throw new ClientError.NOT_FOUND ("no such user '%s'", context["username
↪"]);
        }

        context["user"] = user;
        return next ();
    });

    // update model data
    app.post ("/<username>", (req, res, next, context) => {
        var username = context["username"].get_string ();
        var user     = context["user"] as User;
        var parser   = new Json.Parser ();

        // whitelist for allowed properties
        string[] allowed = {"username"};

        // update the model when members are read
        parser.object_member.connect ((obj, member) => {
            if (member in allowed)
                user.set_property (member,
                                   obj.get_member (member).get_value ());
        });

        if (!parser.load_from_stream (req.body))
            throw new ClientError.BAD_REQUEST ("unable to parse the request body");

        // persist the changes
        user.update ();

        if (user.username != username) {
            // model location has changed, so we throw a 201 CREATED status
            throw new Success.CREATED ("/user/%s".printf (user.username));
        }

        context["user"] = user;

        return next ();
    });

    // serialize to JSON any provided GObject
    app.rule (Method.GET, "*", (req, res, next, context) => {
        var generator = new Json.Generator ();

        generator.root   = Json.gobject_serialize (context["user"].get_object ());
        generator.pretty = false;
```

```
        res.headers.set_content_type ("application/json", null);

        return generator.to_stream (res.body);
    });
});
```

It is also possible to use json-glib-1.0/Json.Parser.load_from_stream_async and invoke *next* in the callback with *Router* `invoke` function if you are expecting a considerable user input.

```
parser.load_from_stream_async.begin (req.body, null, (obj, result) => {
    var success = parser.load_from_stream_async.end (result);

    user.update ();

    context["user"] = user;

    // execute 'next' in app context
    return app.invoke (req, res, next);
});
```

Persistence

Multiple persistence solutions have bindings in Vala and can be used by Valum.

- libgda for relational databases and more
- memcached
- redis-glib
- mongodb-glib
- couchdb-glib which is supported by the Ubuntu team

One good general approach is to use a per-process connection pool since handlers are executing in asynchronous context, your application will greatly benefit from multiple connections.

Memcached

You can use libmemcached.vapi to access a Memcached cache storage, it is maintained in nemequ/vala-extra-vapis GitHub repository.

```
using Valum;
using VSGI;

var app       = new Router ();
var memcached = new Memcached.Context ();

app.get ("/<key>", (req, res) => {
    var key = req.params["key"];

    int32 flags;
    Memcached.ReturnCode error;
    var value = memcached.get ("hello", out flags, out error);

    return res.expand (value, null);
```

```
});

app.post ("/<key>", (req, res) => {
    var key    = req.params["key"];
    var buffer = new MemoryOutputStream.resizable ();

    // fill the buffer with the request body
    buffer.splice (req);

    int32 flags;
    Memcached.ReturnCode error;
    var value = memcached.get ("hello", out flags, out error);

    return res.expand (value, null);
});

Server.new ("http", handler: app).run ();
```

Resources

GLib provides a powerful gio-2.0/GLib.Resource for bundling static resources and optionally link them in the executable.

It has a few advantages:

- resources can be compiled in the text segment of the executable, providing lightning fast loading time
- resource api is simpler than file api and avoids IOError handling
- application do not have to deal with its resource location or minimally if a separate bundle is used

This only applies to small and static resources as it will grow the size of the executable. Also, if the resources are compiled in your executable, changing them will require a recompilation.

Middlewares are provided for that purpose, see *../middlewares/static* for more details.

Integration

Let's say your project has a few resources:

- CTPL templates in a `templates` folder
- CSS, JavaScript files in `static` folder

Setup a `app.gresource.xml` file that defines what resources will to be bundled.

```
<?xml version="1.0" encoding="UTF-8"?>
<gresources>
  <gresource>
    <file>templates/home.html</file>
    <file>templates/404.html</file>
    <file>static/css/bootstrap.min.css</file>
  </gresource>
</gresources>
```

You can test your setup with:

```
glib-compile-resource app.gresource.xml
```

Latest version of `waf` automatically link `*.gresource.xml` if you load the `glib2` plugin and add the file to your sources.

```
bld.load('glib2')

bld.program(
   packages  = ['valum-0.1'],
   target    = 'app',
   source    = bld.path.ant_glob('**/*.vala') + ['app.gresource.xml'],
   uselib    = ['VALUM'])
```

The app example serves its static resources this way if you need a code reference.

Scripting

Through Vala VAPI bindings, application written with Valum can embed multiple interpreters and JIT to provide facilities for computation and templating.

Lua

luajit ships with a VAPI you can use to access a Lua VM, just add `--pkg lua` to `valac`.

```
valac --pkg valum-0.1 --pkg lua app.vala
```

```
require 'markdown'
return markdown('## Hello from lua.eval!')
```

```
using Valum;
using VSGI;
using Lua;

var app = new Router ();
var lua = new LuaVM ();

// GET /lua
app.get ("/lua", (req, res) => {
    // evaluate a string containing Lua code
    res.expand_utf8 (some_lua_code, null);

    // evaluate a file containing Lua code
    return res.expand_utf8 (lua.do_file ("scripts/hello.lua"));
});

Server.new ("http", handler: app.handle).run ();
```

The sample Lua script contains:

```
require 'markdown'
return markdown("# Hello from Lua!!!")
-- returned value will be appended to response body
```

Resulting response

```
<h1>Hello from Lua!!!</h1>
```

Scheme (TODO)

Scheme can be used to produce template or facilitate computation.

```
app.get ("/hello.scm", (req, res) => {
    return res.expand_utf8 (scm.run ("scripts/hello.scm"));
});
```

Scheme code:

```
;; VALUM_ROOT/scripts/hello.scm
(+ 1 2 3)
;; returned value will be casted to string
;; and appended to response body
```

Templating

Template engines are very important tools to craft Web applications and a few libraries exist to handle that tedious work.

Compose

For HTML5, Compose is quite appropriate.

```
app.get ("/", (req, res) => {
    return res.expand_utf8 (
        html ({},
            head ({},
                title ()),
            body ({},
                section (
                    h1 ({}, "Section Title")))));
});
```

It comes with two utilities: `take` and `when` to iterate and perform conditional evaluation.

```
var users = Users.all ();

take<User> (()     => { return users.next (); },
            (user) => { return user.username; });

when (User.current ().is_admin,
      () => { return p ({}, "admin") },
      () => { return p ({}, "user") });
```

Strings are not escaped by default due to the design of the library. Instead, all unsafe value must be escaped properly. For HTML, `e` is provided.

```
e (user.biography);
```

Templates and fragments can be store in Vala source files to separate concerns. In this case, arguments would be used to pass the environment.

```
using Compose.HTML5;

namespace Project.Templates
{
    public string page (string title, string content)
    {
        return
            div ({"id=%s".printf (title)},
                h2 ({}, e (title)),
                content);
    }
}
```

Template-GLib

Template-GLib provide a more traditional solution that integrates with GObject. It can render properties and perform method calls.

```
using Tmpl;

var home = new Template.from_resource ("home.tmpl");

app.get ("/", (req, res) => {
    var scope = new Scope ();
    scope.set_string ("title", "Home");
    home.expand (scope, res.body);
});
```

CHAPTER 9

Hacking

This document addresses hackers who wants to get involved in the framework development.

Code conventions

Valum uses the Vala compiler coding style and these rules are specifically highlighted:

- tabs for indentation
- spaces for alignment
- 80 characters for comment block and 120 for code
- always align blocks of assignation around = sign
- remember that little space between a function name and its arguments
- doclets should be aligned, grouped and ordered alphabetically

General strategies

Produce minimal headers, especially if the response has an empty body as every byte will count.

Since `GET` handle `HEAD` as well, verifying the request method to prevent spending time on producing a body that won't be considered is important.

```
res.headers.set_content_type ("text/html", null);

if (req.method == "HEAD") {
    size_t bytes_written;
    return res.write_head (out bytes_written);
}

return res.expand_utf8 ("<!DOCTYPE html><html></html>");
```

Use the `construct` block to perform post-initialization work. It will be called independently of how the object is constructed.

Tricky stuff

Most of HTTP/1.1 specification is case-insensitive, in these cases, libsoup-2.4/Soup.str_case_equal must be used to perform comparisons.

Try to stay by the book and read carefully the specification to ensure that the framework is semantically correct. In particular, the following points:

- choice of a status code
- method is case-sensitive
- URI and query are automatically decoded by libsoup-2.4/Soup.URI
- headers and their parameters are case-insensitive
- `\r\n` are used as newlines
- do not handle `Transfer-Encoding`, except for the libsoup-2.4 implementation with `steal_connection`: at this level, it's up to the HTTP server to perform the transformation

The framework should rely as much as possible upon libsoup-2.4 to ensure consistent and correct behaviours.

Coverage

gcov is used to measure coverage of the tests on the generated C code. The results are automatically uploaded to Codecov on a successful build.

You can build Valum with coverage by passing the `-D b_coverage` flag during the configuration step.

```
meson -D b_coverage=true
ninja test
ninja coverage-html
```

Once you have identified an uncovered region, you can supply a test that covers that particular case and submit us a pull request on GitHub.

Tests

Valum is thoroughly tested for regression with the glib-2.0/GLib.Test framework. Test cases are annotated with `@since` to track when a behaviour was introduced and guarantee its backward compatibility.

You can refer an issue from GitHub by calling `Test.bug` with the issue number.

```
Test.bug ("123");
```

Version bump

Most of the version substitutions is handled during the build, but some places in the code have to be updated manually:

- `version` and `api_version` variable in `meson.build`
- GIR version annotations for all declared namespaces
- `version` and `release` in `docs/conf.py`

CHAPTER 10

GNU Lesser General Public License

Version 3, 29 June 2007 Copyright © 2007 Free Software Foundation, Inc <http://fsf.org/>

Everyone is permitted to copy and distribute verbatim copies of this license document, but changing it is not allowed.

This version of the GNU Lesser General Public License incorporates the terms and conditions of version 3 of the GNU General Public License, supplemented by the additional permissions listed below.

0. Additional Definitions

As used herein, "this License" refers to version 3 of the GNU Lesser General Public License, and the "GNU GPL" refers to version 3 of the GNU General Public License.

"The Library" refers to a covered work governed by this License, other than an Application or a Combined Work as defined below.

An "Application" is any work that makes use of an interface provided by the Library, but which is not otherwise based on the Library. Defining a subclass of a class defined by the Library is deemed a mode of using an interface provided by the Library.

A "Combined Work" is a work produced by combining or linking an Application with the Library. The particular version of the Library with which the Combined Work was made is also called the "Linked Version".

The "Minimal Corresponding Source" for a Combined Work means the Corresponding Source for the Combined Work, excluding any source code for portions of the Combined Work that, considered in isolation, are based on the Application, and not on the Linked Version.

The "Corresponding Application Code" for a Combined Work means the object code and/or source code for the Application, including any data and utility programs needed for reproducing the Combined Work from the Application, but excluding the System Libraries of the Combined Work.

1. Exception to Section 3 of the GNU GPL

You may convey a covered work under sections 3 and 4 of this License without being bound by section 3 of the GNU GPL.

2. Conveying Modified Versions

If you modify a copy of the Library, and, in your modifications, a facility refers to a function or data to be supplied by an Application that uses the facility (other than as an argument passed when the facility is invoked), then you may convey a copy of the modified version:

- **a) under this License, provided that you make a good faith effort to** ensure that, in the event an Application does not supply the function or data, the facility still operates, and performs whatever part of its purpose remains meaningful, or
- **b) under the GNU GPL, with none of the additional permissions of** this License applicable to that copy.

3. Object Code Incorporating Material from Library Header Files

The object code form of an Application may incorporate material from a header file that is part of the Library. You may convey such object code under terms of your choice, provided that, if the incorporated material is not limited to numerical parameters, data structure layouts and accessors, or small macros, inline functions and templates (ten or fewer lines in length), you do both of the following:

- **a) Give prominent notice with each copy of the object code that the** Library is used in it and that the Library and its use are covered by this License.
- **b) Accompany the object code with a copy of the GNU GPL and this license** document.

4. Combined Works

You may convey a Combined Work under terms of your choice that, taken together, effectively do not restrict modification of the portions of the Library contained in the Combined Work and reverse engineering for debugging such modifications, if you also do each of the following:

- **a)** Give prominent notice with each copy of the Combined Work that the Library is used in it and that the Library and its use are covered by this License.
- **b)** Accompany the Combined Work with a copy of the GNU GPL and this license document.
- **c)** For a Combined Work that displays copyright notices during execution, include the copyright notice for the Library among these notices, as well as a reference directing the user to the copies of the GNU GPL and this license document.
- **d)** Do one of the following:
  - **0)** Convey the Minimal Corresponding Source under the terms of this License, and the Corresponding Application Code in a form suitable for, and under terms that permit, the user to recombine or relink the Application with a modified version of the Linked Version to produce a modified Combined Work, in the manner specified by section 6 of the GNU GPL for conveying Corresponding Source.
  - **1)** Use a suitable shared library mechanism for linking with the Library. A suitable mechanism is one that **(a)** uses at run time a copy of the Library already present on the user's computer system, and **(b)** will operate properly with a modified version of the Library that is interface-compatible with the Linked Version.

- **e)** Provide Installation Information, but only if you would otherwise be required to provide such information under section 6 of the GNU GPL, and only to the extent that such information is necessary to install and execute a modified version of the Combined Work produced by recombining or relinking the Application with a modified version of the Linked Version. (If you use option **4d0**, the Installation Information must accompany the Minimal Corresponding Source and Corresponding Application Code. If you use option **4d1**, you must provide the Installation Information in the manner specified by section 6 of the GNU GPL for conveying Corresponding Source.)

5. Combined Libraries

You may place library facilities that are a work based on the Library side by side in a single library together with other library facilities that are not Applications and are not covered by this License, and convey such a combined library under terms of your choice, if you do both of the following:

- **a)** Accompany the combined library with a copy of the same work based on the Library, uncombined with any other library facilities, conveyed under the terms of this License.
- **b)** Give prominent notice with the combined library that part of it is a work based on the Library, and explaining where to find the accompanying uncombined form of the same work.

6. Revised Versions of the GNU Lesser General Public License

The Free Software Foundation may publish revised and/or new versions of the GNU Lesser General Public License from time to time. Such new versions will be similar in spirit to the present version, but may differ in detail to address new problems or concerns.

Each version is given a distinguishing version number. If the Library as you received it specifies that a certain numbered version of the GNU Lesser General Public License "or any later version" applies to it, you have the option of following the terms and conditions either of that published version or of any later version published by the Free Software Foundation. If the Library as you received it does not specify a version number of the GNU Lesser General Public License, you may choose any version of the GNU Lesser General Public License ever published by the Free Software Foundation.

If the Library as you received it specifies that a proxy can decide whether future versions of the GNU Lesser General Public License shall apply, that proxy's public statement of acceptance of any version is permanent authorization for you to choose that version for the Library.