GNU Artanis

Mu Lei known as NalaGinrut

Table of Contents

1 Introduction

GNU Artanis is a web application framework(WAF) written in Guile Scheme.

A web application framework (WAF) is a software framework that is designed to support the development of dynamic websites, web applications, web services and web resources. The framework aims to alleviate the overhead associated with common activities performed in web development. GNU Artanis provides several tools for web development: database access, templating frameworks, session management, URL-remapping for RESTful, page caching, and so on.

This manual describes how to use GNU Artanis, and usage of APIs.

Guile is the GNU Ubiquitous Intelligent Language for Extensions, the official extension language for the GNU operating system. Guile is also an interpreter and compiler for other dynamic programming languages except Scheme programming language.

Scheme is a functional programming language and one of the two main dialects of the programming language Lisp. Scheme follows a minimalist design philosophy specifying a small standard core with powerful tools for language extension.

1.1 Conventions

In this manual, we'll use this kind of syntax to indicate the usage of API:

```
(api-name arg1 arg2 #:key0 val0 ... [optional-arg1 <- default-value1] ...)
```

If you're not comfortable with this syntax, maybe you're a newbie of Scheme, and I would recommend you read Chapter 5 [Basic in Scheme], page 8 chapter first.

1.2 No warranty

We distribute software in the hope that it will be useful, but without any warranty. No author or distributor of this software accepts responsibility to anyone for the consequences of using it or for whether it serves any particular purpose or works at all, unless they say so in writing. This is exactly the same warranty that proprietary software companies offer: none.

2 License

GNU Artanis is Free Software. GNU Artanis is under the terms of the GNU Lesser General Public License version 3 or later. See the files COPYING.LESSER and COPYING in toplevel of source code.

The manual you're now reading is published under the terms of the GNU Free Documentation License 1.3 or later.

You must be aware there is no warranty whatsoever for GNU Artanis. This is described in full in the licenses.

3 Installation

3.1 For users

Install GNU Guile-2.0.11 or higher version:

- Debian/Ubuntu users

```
sudo apt-get install guile-2.0-dev guile-2.0
```

- SUSE/openSUSE users

```
sudo zypper install guile guile-devel
```

- RedHat/Fedora

```
sudo dnf install guile guile-devel
```

- For LFS Guru

```
wget -c ftp://ftp.gnu.org/gnu/guile/guile-2.0.11.tar.gz
tar xvzf guile-2.0.11.tar.gz
cd guile-2.0.11 && ./configure && make #(NOTE: this may take very long time even looks
sudo make install
```

I would NOT recommend you trying to compile/install Guile from Git repo, since it'll take too much time of you.

Install dependencies:

- guile-dbi-2.1.5 [**Optional**]

```
wget -c http://download.gna.org/guile-dbi/guile-dbi-2.1.5.tar.gz
tar xvzf guile-dbi-2.1.5.tar.gz
cd guile-dbi-2.1.5 && ./configure && make
sudo make install
```

- guile-dbd [**Optional**], there're three dbd, mysql/postgresql/sqlite3

```
wget -c http://download.gna.org/guile-dbi/guile-dbd-mysql-2.1.4.tar.gz
tar xvzf guile-dbd-mysql-2.1.4.tar.gz
cd guile-dbd-mysql-2.1.4 && ./configure && make
sudo make install
```

You may find other dbd `ttp://download.gna.org/guile-dbi`. And the installation is similar.

Install the latest GNU Artanis:

```
wget -c http://alpha.gnu.org/gnu/artanis/artanis-latest.tar.bz2
tar xvjf artanis-latest.tar.bz2
cd artanis-latest && ./configure && make
sudo make install
```

3.2 For contributors

First, thanks for you contributions!

If you're comfortable with GitHub, then just follow the steps which you've already known.

Anyway, here's the git repo:

```
git clone git://git.savannah.gnu.org/artanis.git

# mirror on Github
git clone git@github.com:NalaGinrut/artanis.git
```

3.3 Configuration

Before the first time to run, GNU Artanis needs a config file whose name is `/etc/artanis/artanis.conf`.

The config items are listed below:

(To be continued . . .)

4 Hello World

4.1 Use Guile REPL and verify GNU Artanis installation

If you're NOT freshman of Guile, please skip this section.

Just type 'guile' in your console to enter Guile REPL, and you will see this screen:

```
GNU Guile 2.0.11
Copyright (C) 1995-2014 Free Software Foundation, Inc.

Guile comes with ABSOLUTELY NO WARRANTY; for details type ',show w'.
This program is free software, and you are welcome to redistribute it
under certain conditions; type ',show c' for details.

Enter ',help' for help.
scheme@(guile-user)>
```

Welcome to Guile world!

Now, we're going to play GNU Artanis. Before we start, please follow these instructions in the REPL to ensure that you installed GNU Artanis correctly:

(Just type them, you don't have to understand them at present)

```
,use (artanis artanis)
artanis-version
```

The expected output should be similar to this:

```
$1 = "GNU Artanis-x.x.x"
```

4.2 Simple HTTP server

Run this code in your console:

```
guile -c "(use-modules (artanis artanis))(init-server)(run)"
## You'll see this screen:
Anytime you want to Quit just try Ctrl+C, thanks!
http://0.0.0.0:3000
```

Assuming there's a file named "index.html" in the current path. Now you may try `http://localhost:3000/index.html` in your browser. It's simple to fetch static file with the path in URL: `http://localhost:3000/path/filename`

4.3 Try simple URL remapping

Type these code in Guile REPL:

```
(use-modules (artanis artanis))
(get "/hello" (lambda () "hello world"))
(run #:port 8080)
```

Now you can visit `http://localhost:8080/hello` with your browser, and see the result.

If you encounter "[EXCEPTION] /favicon.ico is abnormal request", please ignore it.

Let me explain these code.

- *line 1:* Load GNU Artanis module, (artanis artanis) is the name.
- *line 2:* The first argument *get* is GNU Artanis API corresponding to GET method in HTTP protocol. The second argument "/hello" is the URL rule to register. The Third argument is the handler which will be triggered if the registered URL rule is hit.
- *line 3:* Run GNU Artanis server, and listenning socket port 8080.

You may type Ctrl+C to quit the server according to the hint from your screen.

4.4 More complex URL remapping

Try this code:

```
(get "/hello/:who"
  (lambda (rc)
    (format #f "<p>hello ~a</p> " (params rc "who"))))
(run #:port 8080)
```

Now you can try `http://localhost:8080/hello/artanis` in your browser.

There're two differences:

- 1. The special rule, `"/hello/:who"`, *:who* means you can use *params* to reference the value of this section of URL with the key "who". `(params rc "who")` is the way for that.
- 2. You may notice that the handler defined as an anonymous function with *lambda* has one argument *rc*. It means *route context* which preserve all the related context information. Many GNU Artanis APIs need it, say, *params*.

And *format* is a Scheme lib function. It is similar to *sprintf* in C language, which outputs with a formatted pattern. The second argument #f (means FALSE) indicates that returning the result as string type rather than printing out.

4.5 Regex in URL remapping

You can use regex in the URL rule.

```
(get "/.+\\.(png|gif|jpeg)" static-page-emitter)
```

static-page-emitter is an GNU Artanis API which emits a static file like images to the client.

4.6 Database operating

GNU Artanis supports mysql/postgresql/sqlite3, we use mysql as a example here.

Please ensure that your DB service was started before you try.

If you encountered any problems, please check your config of DB first. You can use DB without running a server.

```
(use-module (artanis artanis))
(define conn (connect-db 'mysql #:db-username "your_db_username"
                         #:db-name "your_db_name" #:db-passwd "your_passwd"))
(define mtable (map-table-from-DB conn))
((mtable 'create 'Persons '((name varchar 10) (age integer) (email varchar 20))) 'vali
;; ==> #t
```

```
(mtable 'set 'Persons #:name "nala" #:age 99 #:email "nala@artanis.com")
(mtable 'get 'Persons #:columns '(name email))
;; ==> ((("name" . "nala") ("email" . "nala@artanis.com")))
```

- *map-table-from-DB* is GNU Artanis API handling tables in DB. Here, we define this mapping as the var *mtable*.
- And we can use *mtable* to handle tables, you can get values from table with 'get command.
- *mtable* is a functon which accepts the first argument as a command, say 'create is a command to create a new table; 'set command is used to insert/update the table; 'get command for fetch the values of specified columns.
- The second argument of *mtable* is the name of the table as you guess. Please note that it's case sensitive. But the columns name could be case insensitive.
- 'create command returns a function too, which also accepts an argument as a command. Here, we use 'valid? command to check if the table has been created successfully.

Here's just simple introduction. You may read the DB section in this manual for detail describing.

Of course, you can use DB in your web application.

```
(get "/dbtest" #:conn #t ; apply for a DB connection from pool
  (lambda (rc)
    (let ((mtable (map-table-from-DB (:conn rc))))
      (object->string
        (mtable 'get 'Persons #:columns '(name email))))))

(run #:use-db? #t #:dbd 'mysql #:db-username "your_db_username"
     #:db-name "your_db_name" #:db-passwd "your_passwd" #:port 8080)
```

Now, try `http://localhost:8080/dbtest` in your browser.

Here're some explains:

- The keyword-value pair `#:conn #t` means applying for a DB connection from connection-pool. Then you can use `(:conn rc)` to get the allocated connection for DB operations.
- Finally, the handler needs to return a string as the HTTP response body, so we have to use Guile API *object->string* to convert the query result to string, for this naive example case.

Exercise: Return a beautiful table in HTML rather than using object->string.

5 Basic in Scheme

This chapter introduces some useful documents to help you understand Scheme language well. Feel free to come back here if you have any problem with Scheme syntax.

If any possbile, read them again and again.

Scheme was introduced in 1975 by Gerald J. Sussman and Guy L. Steele Jr. and was the first dialect of Lisp to fully support lexical scoping, first-class procedures, and continuations. In its earliest form it was a small language intended primarily for research and teaching, supporting only a handful of predefined syntactic forms and procedures. Scheme is now a complete general-purpose programming language, though it still derives its power from a small set of key concepts. Early implementations of the language were interpreter-based and slow, but Guile Scheme is trying to implement sophisticated compiler that generate better optimized code, and even a plan for AOT compiler generated native code in the future.

5.1 For newbies

If you're not familiar with Guile Scheme, here's a simplest tutorial for you.

If you know basics of Scheme language, please skip this section.

I would recommend newbies to type/paste the code in Guile REPL following the guide in tutorial: Learn Scheme in 15 minutes

And here's a nice section in Guile manual for basics in Scheme: Hello Scheme

Please don't spend too much time on these tutorials, the purose is to let newbies get a little familiar with the grammar of Scheme.

5.2 For Pythoners

These are good articles for Pythoners:

1. Guile basics from the perspective of a Pythonista
2. Going from Python to Guile Scheme

Still, please don't spend too much time on them, the purose is to let newbies get a little familiar with the grammar of Scheme.

5.3 For Rubyist

Here's a great article for Rubyist to learn Scheme:

1. Scheme for ruby programmers

5.4 For deep learners

These two books are very good for learning Scheme seriously:

1. The Scheme Programming Language
2. Structure and Interpretation of Computer Programs(SICP)

Please don't read them if you just want to use GNU Artanis to build your webapp/site in few minutes.

If you really want to try these books seriously, please ignore GNU Artanis before you done them.

But once you've done them **carefully**, you may want to write a new GNU Artanis all by yourself.

Hold your horses. ;-)

6 Basic in GNU Artanis

6.1 How to run a site with GNU Artanis

This is the simplest case to run a site:

```
#!/bin/env guile
!#
(use-modules (artanis artanis))
(init-server)
(get "/hello" (lambda () "hello world"))
(run)
```

6.2 Initialization

It’s better to use (init-server) to init GNU Artanis.

```
(init-server #:statics '(png jpg jpeg ico html js css) #:cache-statics? #f #:exclude '
```

`#:statics` specifies the static files with the extension file. GNU Artanis is based on URL remapping, so this keyword let you avoid to handle each static file types. In default, it coveres the most static file types. So you may ignore it usually.

`#:cache-statics?` indicates if the static files should be cached.

`#:exclude` specifies the types should be excluded. This is useful when you want to generate image files dynamically. Even js/css could be generated dynamically, depends your design.

6.3 Registering handler of HTTP methods

Please read Section 9.2 [URL handling], page 15.

6.4 Emit Response

```
(response-emit body #:status 200 #:headers '() #:mtime (current-time))
```

body is the response body, it can be bytevector or literal string (in HTML).

`#:status` is HTTP status, 200 in default, which means OK.

`#:headers` let you specify customized HTTP headers. The headers must follow certain format, you have to read about the Response Headers.

`#:mtime` specifies the modify time in the response. GNU Artanis will generate it for you if you just ignore it.

```
(emit-response-with-file filename [headers <- '()])
```

filename is the filename to be sent as a response.

[headers] is the customized HTTP headers.

6.5 Running server

```
(run #:host #f #:port #f #:debug #f #:use-db? #f
     #:dbd #f #:db-username #f #:db-passwd #f #:db-name #f)
```

You may see all the keyword is #f in default, this means these items will be gotten from config file.

But you can specify them as will.

`#:host` specify the hostname.

`#:port` specify the socket port of the server.

`#:debug` set #t if you want to enable debug mode. Maybe verbose.

`#:use-db?` set #t if you want to use DB, and GNU Artanis will init DB config for you.

`#:dbd` choose dbd, there're three supported dbd: mysql, postgresql, and sqlite3.

`#:db-username` specify the username of your DB server.

`#:db-passwd` the DB password.

`#:db-name` specify DB name.

6.6 Working with Nginx

You may try GNU Artanis+Nginx with so-called reverse proxy.

I would recommend you use Nginx as the front server, since GNU Artanis hasn't done its own async server-core which will be based on delimited-continuations. The current server has some caveats, in spite of the performance, you may suffer from slow-header-ddos if you use GNU Artanis to serv you site directly. But it's fine when you use Nginx in front of GNU Artanis.

For example, you may add these lines to your /etc/nginx/nginx.conf:

```
location / {
        proxy_pass http://127.0.0.1:1234;
        proxy_set_header Host $host;
        proxy_set_header X-Real-IP $remote_addr;
        proxy_set_header X-Forwarded-For $proxy_add_x_forwarded_for;
}
```

Then restart you Nginx:

```
sudo service nginx restart
```

And run GNU Artanis:

```
(run #:port 1234)
```

7 The Art command line

GNU Artanis provides **art** command line tool to save users' time.

7.1 art create

If you want to set up your site/app within an application folder, and take advatage of MVC, you have to use this command to create the application folder first.

```
art create proj_path
```

7.2 art draw

This command will generate the specified component:

```
Usage:
  art draw <component> NAME [options]

component list:
  model
  controller
  migration

Options:
  -h, [--help]     # Print this screen
  -d, [--dry]      # Dry run but do not make any changes
  -f, [--force]    # Overwrite files that already exist
  -s, [--skip]     # Skip files that already exist
                   # If -s and -f are both provided, -f will be enabled
  -q, [--quiet]    # Suppress status output

Example:
  art draw model myblog
```

Please see Chapter 8 [MVC], page 14 to learn more about how to use these components.

7.3 art migrate

Migrate is used for Database migration.

```
Usage:
  art migrate operator name [OPTIONS]

Operators:
  up
  down

OPTIONS:
  VERSION=version
```

Please see Section 11.1 [Migration], page 18 for more detail.

7.4 art work

This command is used to start the server to run your site in the application folder:

```
Usage:
  art work [options]

Options:
  -c, [--config=CONFIG]          # Specify config file
                                   Default: conf/artanis.conf
                                            if no, /etc/artanis/artanis.conf
  -h, [--host=HOST]              # Specify the network host
                                   Default: 0.0.0.0
  -d, [--usedb]                  # Whether to use Database
                                   Default: false
  -b, [--dbd=DBD]                # Specify DBD, mysql/postgresql/sqlit3
                                   Default: mysql
  -n, [--name=DATABASE_NAME]     # Database name
                                   Default: artanis
  -w, [--passwd=PASSWD]          # Database password
                                   Default: none
  -u, [--user=USER]              # Database user name
                                   Default: root
  -p, [--port=PORT]              # Specify listenning port
                                   Default: 3000
  -g, [--debug]                  # Debug mode
                                   Default: disable
  -s, [--server=SERVER]          # Specify server core
                                   Default: inner (Guile built-in server)
  --help                         # Show this screen
```

8 MVC

MVC is Model-Views-Controller, the most classic architectural pattern for implementing user interfaces. It divides a given software application into three interconnected parts, so as to separate internal representations of information from the ways that information is presented to or accepted from the user.

8.1 Controllers/Views

When you run it to generate a controller named *article*:

```
art draw controller article show edit
```

show and *edit* are the name of methods for the controller named *article*.

And it'll generate both **controller** and **view** for *article*:

```
drawing    controller article
working    Controllers ‘article.scm’
create     app/controllers/article.scm
working    Views ‘article’
create     app/views/article/show.html.tpl
create     app/views/article/edit.html.tpl
```

As you may see, there're three files were generated:

```
app/controllers/article.scm
app/views/article/show.html.tpl
app/views/article/edit.html.tpl
```

This means the controller *article* has two methods mapped to URL rule named *show* and *edit*. And *view* component will generate HTML template for each method, say, **show.html.tpl**. For example, the controller *article* generate *show* method handler automatically:

```
(article-define show
  (lambda (rc)
  "<h1>This is article#show</h1><p>Find me in app/views/article/show.html.tpl</p>"
  ;; TODO: add controller method ‘show’
  ;; uncomment this line if you want to render view from template
  ;; (view-render "show")
  ))
```

Of course, it depends on you whether to use these template. If you want to use *view template*, just uncomment the last line `(view-render "show")`.

For more detail about template in Views, please see Chapter 10 [Layouts and Rendering in GNU Artanis], page 16.

8.2 Models

Models contains operations of database.

For modifying tables, you should read Section 11.1 [Migration], page 18.

For other DB operation, please read Section 11.4 [FPRM (experimental)], page 20.

(To be continue. . .)

9 URL remapping

9.1 Introduction to URL remapping

URL remapping is used to modify a web URL's appearance to provide short, pretty or fancy, search engine friendly URLs. It's largly used in modern WAF(web application framework) to provide RESTful web APIs.

9.2 URL handling

According to RFC2616, there're GET, POST, PUT, PATCH and DELETE methods. You may register handler for specified URL rule to these methods.

There'd be HEAD method, but in GNU Artanis, HEAD method is handled by the server, users can't use it.

The usage:

```
(method rule handler)
```

And the handler could be two types, depends on your need:

```
(lambda ()
  ...
  ret)

(lambda (rc)
  ...
  ret)
```

ret also has two types:

- 1. literal string as the returned response body
- 2. See Section 6.4 [Emit Response], page 10

```
(get "/hello" (lambda () "hello world"))
```

For POST method:

```
(post "/auth" (lambda (rc) ...))
```

9.3 Get params from URL

```
(params rc name)
;; e.g
(get "/hello/:who" (lambda (rc) (params rc "who")))
```

9.4 Redirect link

```
(redirect-to rc path #:status 301
                     #:scheme 'http)
;; e.g
(get "/aaa" (lambda (rc) (redirect-to rc "/bbb")))
(get "/bbb" (lambda () "ok bbb"))
```

10 Layouts and Rendering in GNU Artanis

10.1 Templating

Templating provides a way to mix programming code into HTML.

10.2 Templating for Pythoners

If you're familiar with Django, which implemented a DSL(Domain Specific Language) to express presentation rather than program logic. You may realize that the templating of GNU Artanis has different philosophy.

In templating of GNU Artanis, it's simply embedded Scheme code into HTML. Why? Because of the philosophy of FP(Functional Programming), everything could be a function. So obviously, `(filesizeformat size)` is enough for understanding, and it's just simple function calling in prefix-notation. There's no need to implement DSL like `size|filesizeformat` to increase the complexity of code. Let alone the syntax is very different from Python.

The syntax like `size|filesizeformat` is postfix-notation used in stack-based languages, say Forth. Such a language used to delegate another programming paradigm named concatenative programming. It's very different from the paradigm of Scheme(functional programming), and the paradigm of Python(imperative programming).

The philosophy of GNU Artanis templating is to bring it into correspondence with the paradigm of the language. And reduce the unnecessary complexities. KISS.

10.3 Templating for Rubyists

Templating in GNU Artanis looks very similar to Rails.

The Rails code:

```
<% if( @fullscreen == 1 ) %>
<%= "<div class='full'><p>...</p></div>" %>
<% end %>
```

And the same function in GNU Artanis code:

```
<% (if (= fullscreen 1) %>
<% "<div class='full'><p>...</p></div>" %>
<% ) %>
```

10.4 Templating APIs

```
(tpl->response filename/sxml [environment <- (the-environment)] [escape? <- #f])
(tpl->html filename/sxm [environment <- (the-environment)] [escape? <- #f])
```

The difference is that tpl->html returns a string, but tpl->response will return HTTP response.

[environment] is the environment you want to pass in. We often ignore it. But if you want to ref some vars defined outside your template string, you should pass (the-environment).

[escape?] If you want to HTML char-escaping with the returned string, set it to #t.

There're two kinds of different templating:

10.5 Embedded Templating

Example: Write a tpl file named "my.tpl":

```
<html>
  <p> <%= "This is tpl test!" %> </p>
  <p> <% (format #t "And this is ~a" (getcwd)) %> </p>
  <p> <%= external-var %> </p>
</html>
```

Of course, the ext filename ".tpl" is trivial, you may name it whatever you like.

```
(get "/test"
  (lambda (rc)
    (let ((external-var 123))
      (tpl->response "my.tpl" (the-environment)))))
(run #:port 8080)
```

In this case, make sure to put my.tpl to the same path with your GNU Artanis code.

Because **external-var** is defined outside the file "my.tpl", and it's bound in *let* with 123, you have to pass (the-environment). Or the template render will blame that it can't find variable named **external-var**.

If you don't have any external var needs to be referenced, just use `(tpl->response "file.tpl")` is fine.

Then see `http://localhost:3000/test` in your browser.

10.6 SXML Templating

SXML is an alternative syntax for writing XML data, using the form of S-expressions.

SXML is to Scheme as JSON is to ECMAScript(the so-called javascript). Maybe this explains clearer.

The benifit of SXML is to take advantage of quasiquote in Scheme. If you no little about it, then you may google "scheme quasiquote" for more details.

```
(tpl->response '(html (body (p (@ (id "content")) "hello world"))))
```

You would get a html string `"<html><body><p id=\"content\">hello world</p></body></html>"`.

Let's see an example of quasiquote:

```
(let ((content "hello world"))
  (tpl->response `(html (body (p (@ (id "content")) ,content)))))
```

11 Database

11.1 Migration

Migration provides a way do complicated modification of tables in database automatically. Here's an example.

First, draw a migration:

```
# art draw migration person
drawing    migration person
working    Migration '20151107040209_person.scm'
```

You'll see something similar like above.

In this case, you may edit file db/migration/20151107040209_person.scm:

```
(migrate-up
  (create-table
   'person
   '(id auto (#:primary-key))
   '(name char-field (#:not-null #:maxlen 10))
   '(age tiny-integer (#:not-null))
   '(email char-field (#:maxlen 20))))

(migrate-down
  (drop-table 'person))
```

Now you may run **up** command of migration:

```
art migrate up person
```

Then migrate-up function will be called, and this will create a table named *person*:

```
+-------+---------------------+------+-----+---------+----------------+
| Field | Type                | Null | Key | Default | Extra          |
+-------+---------------------+------+-----+---------+----------------+
| id    | bigint(20) unsigned | NO   | PRI | NULL    | auto_increment |
| name  | varchar(10)         | NO   |     | NULL    |                |
| age   | tinyint(4)          | NO   |     | NULL    |                |
| email | varchar(20)         | YES  |     | NULL    |                |
+-------+---------------------+------+-----+---------+----------------+
```

If you run **down** command of migration:

```
art migrate down person
```

Obviously, the table *person* will be dropped.

11.2 ORM problem

ORM stands for Object Relational Mapping, which is a popular approach to handle relational DB nowadays, in OOP.

Of course, Guile has it's own Object System named GOOPS. Users may use OOP with it. And it's possible to implement ORM in GNU Artanis as well.

However, FP fans realized that they don't have to use OOP if they can use FP features reasonably.

Besides, there're some criticism pointing to ORM:

- ORM Hate
- Vietnam of Computer Science
- Object-Relational Mapping is the Vietnam of Computer Science

And here're some known ways for trying to solve the problems of ORM:

- 1. *Give up ORM*.
- 2. *Give up relational storage model*. Don't use relational DB, pick up others, say, No-SQL. Well, this way is not cool when you have to use relational DB.
- 3. *Manual mapping*. Write SQL code directly. It's fine sometimes. But the code increases when things get complicated. Refactoring and reusing would be worth to consider.
- 4. *Limited ORM*. Limited the utility of ORM. And use ORM to solve part of your work rather than whole, depends on you. This may avoid some problems.
- 5. *SQL related DSL*. Design a new language. LINQ from Microsoft is one of the cases.
- 6. *Integration of relational concepts into frameworks*. Well, harder than 5, but worth to try.
- 7. *Stateless*. This is the critical hit to complexity and unreliability.

Basically, GNU Artanis has no ORM yet, and maybe never. GNU Artanis is trying to experiment new ways to solve the problems of ORM.

GNU Artanis provides three ways to complete this mission. All of them, are **experimental** at present.

- SSQL (1,3,5)
- FPRM (4,7)
- SQL Mapping (1,3,6)

11.3 SSQL (experimental)

The concept of SSQL is very easy. Write SQL in S-expr.

Usage:

```
(->sql sql-statement)
(where #:key val ... [literal string])
(having #:key val ... [literal string])
(/or conds ...)
(/and conds ...)
```

For example:

```
(->sql select * from 'Persons (where #:city "Shenzhen"))
(->sql select '(age name) from 'Persons (where "age < 30"))
```

11.4 FPRM (experimental)

FPRM stands for Functional Programming Relational Mapping. It's a new word I invented. But it's not new concept. FP here indicates **stateless**.

FPRM is still experimental and work-in-progress.

11.4.1 Connect to DB server

```
;; usage 1:
(connect-db dbd init-str)

;; usage 2:
(connect-db dbd #:db-name "artanis" #:db-username "root" #:db-passwd "" #:proto "tcp"
```

- **dbd** is a string, could be "mysql", "postgresql", and "sqlite3".
- **init-str** is a string for DB init, for example:

  ```
  (connect-db "mysql" "root:123:artanis:tcp:localhost:3306")
  ```
- `#:db-name` specifies the DB name.
- `#:db-username` specifis the DB username.
- `#:proto` specifies the socket protocol, which is related to DB server you choosen.
- `#:host` specifies the host name.
- `#:port` specifies the socket port.

11.4.2 Map DB table

This step will generate an new instance (as a closure) mapped to database table or view. In ORM, it is often called Active Record which maps the database view to an class object.

And there're two differences:

- FPRM doesn't create object for each table. It maps a whole database in concept, and generates SQL for each table as you choose. So it maybe lightweight compared to an ORM object.
- FPRM doesn't maintain any states at all, say, it keeps stateless in the object (Not in database).

These two points may decrease the power of FPRM, but our main philosophy in GNU Artanis is that

- *The best way to control DB is SQL, don't bother with other guile schemes.*

That means we're not going to develop a complicated ORM in GNU Artanis, but a promising way to interact with SQL easily. This is what Section 11.5 [SQL Mapping (experimental)], page 22 provided. FPRM aims to reduce states & complexity to privide reliabality, and SQL-Mapping will provide a convenient way to handle complex SQL for better performance and security (from SQL-Injection).

```
(define m (map-table-from-DB rc/conn))
```

rc/conn can be route-context or connection of DB.

map-table-from-DB returns a function, we named it **m** here for explaining.

11.4.3 Create table

```
(m 'create table-name defs #:if-exists? #f #:primary-keys '() #:engine #f)
```

- **table-name** specifies the name of the table in DB.
- **defs** is a list to define the columns' types. For example:
  ```
  '((name varchar 10) (age integer) (email varchar 20))
  ```
- `#:if-exists?` has two kinds of possible options:
  - '**overwrite** or '**drop** means overwriting the existed table if possible.
  - '**ignore** means ignore the table when there's an existed one.
- `#:primary-keys` specifies the primary keys in the created table.
- `#:engine` specifies the engine, depends on the dbd you chosen.

11.4.4 Get columns from table

```
(m 'get table-name #:columns '(*) #:functions '() #:ret 'all #:group-by #f #:order-by
```

- `#:column` is the columns list you wanted.
- `#:functions` is built-in functions calling, e.g:
  ```
  #:functions '((count Persons.Lastname))
  ```
- `#:ret` specifies how to return the result, there're three options:
  - 'all for returning all results
  - 'top for returning the first result
  - integer (larger than 0), you specify the number.
- `#:group-by` used in conjunction with the aggregate functions to group the result-set by one or more columns.
- `#:order-by` used to sort the result-set by one or more columns.

For example, to get Lastname and City column, and return the first result.

```
(m 'get 'Persons #:columns '(Lastname City) #:ret 'top)
```

11.4.5 Set values to table

```
(m 'set table-name . kargs)
```

kargs is a var-list to accept the key-value arguments.

For example:

```
(m 'set 'Persons #:name "nala" #:age 99 #:email "nala@artanis.com")
```

11.4.6 Drop a table

```
(m 'drop table-name)
```

11.4.7 Check existance of table

```
;; case sensitive
(m 'exists? table-name . columns)
;; or for case-insensitive
(m 'ci-exists? table-name . columns)
```

For example:

```
(m 'exists? 'Persons 'city 'lastname)
```

11.4.8 Get schema of a table

```
(m 'schema table-name)
```

NOTE: all the returned name of schema will be downcased.

11.5 SQL Mapping (experimental)

To be continued . . .

12 MIME

`#:mime` method is used to return the proper MIME type in the HTTP response.

```
#:mime type ; for registering type
(:mime rc body) ; for emit the reponse with the proper MIME
```

12.1 JSON

GNU Artanis intergrated the third-party module guile-json. You may use #:mime method to handle JSON:

```
(get "/json" #:mime 'json
  (lambda (rc)
    (let ((j (json (object ("name" "nala") ("age" 15)))))
      (:mime rc j))))
```

For example:

```
(define my-json
  (json (object ("name" "nala") ("age" 15)
               ("read_list" (object ("book1" "The interpreter and structure of Artani
                                    ("book2" "The art of Artanis programming"))))))
(scm->json my-json) ; scm->json will print json
;; ==> {"name" : "nala",
;;      "age" : 15,
;;      "read_list" : {"book2" : "The art of Artanis programming",
;;                     "book1" : "The interpreter and structure of Artanis"}}
```

`scm->json` will print the result directly.

If you need to format json as a string to return to clients, please use `scm->json-string`.

12.2 CSV

GNU Artanis intergrated the third-party module guile-csv. You may use #:mime method to handle CSV:

```
(get "/csv" #:mime 'csv
  (lambda (rc)
    (:mime rc '(("a" "1") ("b" "2")))))
```

12.3 XML

In Scheme, XML is handled with SXML. Another way is to use strings appending method.

```
(get "/xml" #:mime 'xml
  (lambda (rc)
    (:mime rc '(*TOP* (WEIGHT (@ (unit "pound")) (NET (@ (certified "certified")) "67"
```

12.4 SXML

You can use SXML to replace XML for exchanging data format. This way saves some bandwidth.

```
(get "/sxml" #:mime 'sxml
  (lambda (rc)
    (:mime rc '((a 1) (b 2)))))
```

13 Upload files

If you want to deal with uploading files, store-uploaded-files would be you friend.

13.1 Receive upload from client

```
(store-uploaded-files rc #:path (current-upload-path)
                         #:uid #f
                         #:gid #f
                         #:simple-ret? #t
                         #:mode #o664
                         #:path-mode #o775
                         #:sync #f)
```

rc is the route-context.

`#:path` is specified path to put uploaded files.

`#:uid` is new uid for uploaded files, #f means don't change the default uid.

`#:gid` specifies new gid.

`#:simple-ret?` specifies the mode of return:

- if #t, there're only two possible return value, 'sucess for sucess, 'none for nothing has been done.
- if #f, and while it's successful, it returns a list to show more detais: (success size-list filename-list).

`#:mode` chmod files to mode.

`#:path-mode` chmod upload path to mode.

`#:sync` sync while storing files.

13.2 Send upload to Server

Although GNU Artanis is often used in server-side, we provide this function for users to upload files from client.

```
(upload-files-to uri pattern)
```

uri is standard HTTP URL:

```
scheme://[user:password@]domain:port/path?query_string#fragment_id
```

pattern should be: ((file filelist . . .) (data datalist . . .)), for example:

```
(upload-files-to "ftp://nala:123@myupload.com/"
 '((data ("data1" "hello world"))
   (file ("file1" "filename") ("file2" "filename2"))))
```

14 Sessions

You have to use `#:session mode` while you defining URL rule handler.

```
(post "/auth" #:session mode
  (lambda (rc) ...))
```

mode could be:

- #t or 'spawn, to spawn a new session, the name of sid is `"sid"` in default.
- `(spawn ,sid) specify a name of sid to spawn.
- `(spawn ,sid ,proc) specify a name of sid and a proc to **define your own session spawner**.

And the APIs of session is :session

```
(:session rc cmd)
```

cmd could be:

- 'check to check session with name `"sid"`.
- `(check ,sid) to check session with a specified sid name.
- 'check-and-spawn check `"sid"` first, if no, then spawn it.
- `(check-and-spawn ,sid) the same with above, but specifed name of sid.
- `(check-and-spawn-and-keep ,sid) check then spawn then keep it, with the name of sid.
- 'spawn spawn a session with the name `"sid"`.
- 'spawn-and-keep spawn a session then keep with the name `"sid"`.

15 Cookies

You have to use `#:cookies mode` while you defining URL rule handler.

```
(get "/certain-rule" #:cookies mode
  (lambda (rc) ...))
```

mode could be:

- ('names names ...) specifies the name list of the cookies.
- ('custom (names ...) maker setter getter modifier) specify a more complicated customized cookie handers.

And the APIs:

```
(:cookies-set! rc cookie-name key val)

(:cookies-ref rc cookie-name key)

(:cookies-setattr! rc cookie-name #:expir #f #:domain #f #:path #f #:secure #f #:http-

(:cookies-remove! rc key) ; remove cookie from client

(:cookies-update! rc) ; cookies operations won't work unless you update it
```

NOTE: You don't have to call `:cookies-update!` yourself, since it'll be called automatically by the hook before response.

For example:

```
(get "/cookie" #:cookies '(names cc)
  (lambda (rc)
    (:cookies-set! rc 'cc "sid" "123321")
    "ok"))

(get "/cookie/:expires" #:cookies '(names cc)
  (lambda (rc)
    (:cookies-set! rc 'cc "sid" "123321")
    (:cookies-setattr! rc 'cc #:expir (string->number (params rc "expires")))
    "ok"))
```

Now you may use this command in the console to see the result:

```
curl --head localhost:3000/cookie
# and
curl --head localhost:3000/cookie/120
```

16 Authentication

16.1 Init Authentication

GNU Artanis provides flexible mechanism for authentication.

You have to use `#:auth mode` while you defining URL rule handler.

```
(get "/certain-rule" #:auth mode
  (lambda (rc) ...))
```

mode could be:

- SQL as Section 18.1 [String Template], page 31. You may write your own customized SQL for fetching & checking username and passwd.
- ('basic (lambda (rc user passwd) ...)) init a Basic Authentication mode. *user* is submitted username, *passwd* is submitted password value.
- ('table table-name username-field passwd-field) init a common Authentication mode. **The passwd will be encrypted by default algorithm**.
- ('table table-name username-field passwd-field crypto-proc) similar to the above item, but encrypt passwd with crypto-proc.
- (table-name crypto-proc), so passwd field will be "passwd" and username will be "username" in default, and you may encrypt passwd with crypto-proc.

Available crypto-proc helper functions listed here:

- (string->md5 str)
- (string->sha-1 str)

16.2 Basic Authentication

HTTP Basic authentication (BA) implementation is the simplest technique for enforcing access controls to web resources because it doesn't require cookies, session identifier and login pages. Rather, HTTP Basic authentication uses static, standard HTTP headers which means that no handshakes have to be done in anticipation.

The BA mechanism provides no confidentiality protection for the transmitted credentials. They are merely encoded with Base64 in transit, but not encrypted or hashed in any way. Basic Authentication is, therefore, typically used over HTTPS.

GNU Artanis doesn't support HTTPS at present, it's planned to support it in the future.

Let's see a simple example:

```
(get "/bauth" #:auth `(basic ,(lambda (rc u p) (and (string=? u "mmr") (string=? p "12
  (lambda (rc)
    (if (:auth rc)
        "auth ok"
        (throw-auth-needed))))
```

You have to define your own checker with the anonymous function `(lambda (rc u p) ...)`. #t for succeed, #f for failed.

APIs:

- `(:auth rc)` will check if Basic Authentication succeeded, #f for failed.
- `(throw-auth-needed)` is a useful helper function to ask for auth in client side.

16.3 Common Authentication

Actually, there're various authentication methods could be used by developers. Most of them are sort of tricky hacks. Here, we only introduce the most common way.

The most common and relative safe way for authentication is to use POST method. And check username and passwd from a table in DB.

Here is a simple example:

```
(post "/auth" #:auth '(table user "user" "passwd") #:session #t
  (lambda (rc)
    (cond
     ((:session rc 'check) "auth ok (session)")
     ((:auth rc)
      (:session rc 'spawn)
      "auth ok")
     (else (redirect-to rc "/login?login_failed=true")))))
```

NOTE: The passwd will be encrypted by default algorithm.

17 Cache

17.1 On web caching

Web caching is very important nowadays. This section raises a discussion on proper web caching. It couldn't be guide for product. But may help you to understand how to use cache in GNU Artanis.

(to be continued...)

17.2 Cache APIs

You have to use `#:cache mode` while you defining URL rule handler.

```
(get "/certain-rule" #:cache mode
  (lambda (rc) ...))
```

NOTE: the default value of maxage is defined by cache.maxage in `/etc/artanis/artanis.conf`. The default value is 3600 seconds.

mode could be:

- `#t` for enabling caching the page.
- `#f` for disabling caching the page explicitly. It's default to not cache.
- `('static [maxage <- 3600])` This mode must be used for static files, which means the URL rule must be a real path to a static file.
- `(filename [maxage <- 3600])` Specify a static file to cache. This is useful when you don't want to reveal actual path of the static file, but use a fake URL for it.
- `('public filename [maxage <- 3600])` Allow proxies cache the content of specified static file. If HTTP authentication is required, responses are automatically private.
- `('private filename [maxage <- 3600])` Not-Allow proxies cache the content of specified static file.

Let's see the simplest cache test (for dynamica content):

```
(get "/new" #:cache #t
  (lambda (rc)
    (:cache rc "hello world")))
```

If you want to cache a static file, and permit proxies cache the content:

```
(get "/hide" #:cache '(public "pub/some.html")
  (lambda (rc)
    (:cache rc)))
```

But, if your current URL rule is used for authentication (once you use `#:auth`), the cache will be changed to **private** even if you specify **public**.

```
(get "/pauth"
  #:auth `(basic ,(lambda (rc u p) (and (string=? u "nala") (string=? p "123"))))
  #:cache '(public "pub/some.html") ; will be changed to 'private' automatically.
  (lambda (rc) (:cache rc)))
```

18 Utils

The functions introduced here need to import (artanis utils) module.

18.1 String Template

GNU Artanis provides Python3-like template strings:

```
(make-string-template tpl . vals)
```

- **tpl** stands for template string.
- **vals** is varg-list specifying default value to certain key.

For an example:

```
(define st (make-string-template "hello ${name}"))
(st #:name "nala")
;; ==> "hello nala"

;; or you may specify a default value for ${name}
(define st (make-string-template "hello ${name}" #:name "unknown"))
(st)
;; ==> "hello unknown"
(st #:name "john")
;; ==> "hello john"
```

18.2 Random Number Generator

Get random number string from `/dev/urandom`.

```
(get-random-from-dev #:length 8 #:uppercase #f)
```

18.3 Encryption

```
;; hash a string with MD5
(string->md5 str)
;; hash a string with SHA-1
(string->sha-1 str)
```

18.4 Stack & Queue

GNU Artanis provides simple interfaces for stack & queue:

```
;; stack operations
(new-stack)
(stack-pop! stk)
(stack-push! stk elem)
(stack-top stk)
(stack-remove! stk key)
(stack-empty? stk)

;; queue operations
(new-queue)
```

```
(queue-out! q)
(queue-in! q elem)
(queue-head q)
(queue-tail q)
(queue-remove! q key)
(queue-empty? q)
```

19 Appendix A GNU Free Documentation License