rest-server

A library for providing REST APIs
Release 0.1

by Mariano Montone

Table of Contents

This manual is for rest-server version 0.1.

1 Introduction

rest-server is a Common Lisp library for implementing REST APIs providers

1.1 Summary

rest-server is a Common Lisp library for implementing REST APIs providers

1.2 Installation

1.3 Feedback

Mail marianomontone at gmail dot com with feedback

1.4 Conventions

Hear are some coding conventions we'd like to follow:

- We *do* believe in documentation. Document your dynamic variables, functions, macros and classes. Besides, provide a documentation from a wider perspective. Provide diagrams and arquitecture documentation; examples and tutorials, too.
- Use widely known Common Lisp coding guidelines: `http://web.archive.org/web/20050305123711/www`

2 Overview

REST-SERVER is a Common Lisp library for implementing REST APIs servers.

Purpose of the library:

- Method matching
  - Based on HTTP method (GET, PUT, POST, DELETE)
  - Based on Accept request header
  - URL parsing (argument types)
  - Matching based on "extension": i.e. /users.json or /users.xml, etc
  - Method combinations?
- Serialization
  - Different serialization types (JSON, XML, S-expressions)
- Materialization (unserialization)
  - Types
- Error handling
  - Condition serialization
  - Error codes configuration
- Validation
  - Types
  - Schemas (JSON, XML schemas)
- Versioning
  - Support for api versioning?
- Logging
- Cache handling
- Extensible
  - Backends (JSON, XML, etc)
  - Types
  - Validation
- Authentication
  - Different methods (token based, oauth)
  - Avoid changing the api interface spec because of this
- Modes
  - Debugging mode -> outputs full error serialization/backtrace
  - Production -> 500 internal server error
- Documentation
  - For the (lisp) developer
  - For the api consumer:
  - https://github.com/mashery/iodocs
  - http://swagger.wordnik.com/

- Resources
  - Good source of ideas:
  - http://django-rest-framework.org/
  - http://www.restlet.org/

3 API definition and implementation

This chapter is about APIs definitions and its implementation.

3.1 API definiton

APIs are defined using the define-api macro.

This is the syntax:

```
(define-api <api-name> <options-plist>
    &rest
    <api-function-definition>)
```

where some common options are:

- `:documentation` The api docstring
- `:content-types` Globally accepted content types. Valid content-types are :json, :xml, :html and :sexp

and

```
<api-function-definition> := (<api-function-name> <api-function-options> <api-function
```

with required api-function options:

- `:method`. The HTTP method. One of :get, :post, :put, :delete
- `:uri-prefix`. The api function uri prefix. Should start with "/", and encloses required arguments between `{` and `}`. Example: "/users/{id}"
- `:documentation`. The api function documentation string.

and `<api-function-arguments>` being a lambda-list like list with support for `&optional` arguments, but no keyword arguments. Besides, each argument declaration has the form `(<argument-name> <argument-type> <argument-docstring>)` for required arguments and `(<argument-name> <argument-type> <default-value> <argument-docstring>)` for optional values.

Example:

```
(get-user (:method :get
           :content-types (list :json)
           :uri-prefix "/users/{id}"
           :documentation "Retrive an user")
            ((id :string "The user id")
             &optional (expand-groups :boolean nil "Expand groups if true")))
```

3.2 API implementation

The API is supposed to be implemented in a separate package, using the `implement-api-function` macro. Arguments are injected with parsed values (using the argument type in the declaration). Besides, optional arguments in the declaration are assumed to be keyword arguments in the implementation. When the HTTP method is POST, or PUT, the variable name posted-content is filled with the HTTP request posted content.

For instance, when updating a user, we had the following api function declaration:

```
(update-user (:method :put
              :content-types (list :json)
              :uri-prefix "/users/{id}"
              :documentation "Update a user")
          ((id :string "The user id")))
```

To implement that api function, we do:

```
(implement-api-function update-user (posted-content id)
  (format nil "Update user: ~A ~A" id posted-content))
```

3.3 API logging

It is possible to look at what is happening behind the scenes enabling API logging. This is useful for debugging, track how many times and how the API is being accessed, etc.

To start API logging, simply evaluate: `(start-api-logging)`.

The api logging output is sent to `*api-logging-output*`.

Only api-function-implementations with :logging option enabled are logged.

Example:

```
(implement-api-function (get-users :logging t)
    (&key (expand-groups nil))
  (declare (ignore expand-groups))
  (with-output-to-string (s)
    (with-serializer-output s
      (with-serializer (rest-server::accept-serializer)
(with-elements-list ("users")
  (loop for user in (model-test:all-users)
     do
       (with-element ("user")
 (set-attribute "id" (cdr (assoc :id user)))
 (set-attribute "realname" (cdr (assoc :realname user)))))))))
```

And the log output:

```
3566032253 API: Handling GET /users by GET-USERS NIL
3566032253 Response: <!DOCTYPE html PUBLIC "-//W3C//DTD XHTML 1.0 Strict//EN" "http://
<ol class='elements'></ol> NIL
```

4 Serialization

Our purpose is to handle serialization/unserialization of data in a format independent way. That means, the developer shouldn't need change the code if the client is requesting or transferring data in JSON, XML or any other supported format.

There are two ways by which the library trys to achieve that: an intermediate representation, that can be later serialized/unserialized to the desired format; and a generic streaming serialization API: generic operations for serializing data in a format independent way directly to a stream.

4.1 General serialization interface

Serialization output is controlled by the `*serializer-output*` variable, via the `with-serializer-output` macro.

Serialization format is selected by the `*serializer*` variable, via the `with-serializer` macro. The built-in serializers are :json for JSON, :xml for XML, :sexp for Lisp SExpressions and :html for HTML.

Example:

```
(with-serializer-output s
   (with-serializer :json
       (serialize *element*)))
```

4.2 Serialization intermediate representation

Intermediate representation objects are created via the `element`, `attribute` and `elements` macros.

Example:

```
(defparameter *user*
  (element "user"
   (attribute "id" 22)
   (attribute "realname" "Mike")
   (attribute "groups"
      (elements "groups"
(element "group"
 (attribute "id" 33)
 (attribute "title" "My group"))))))
```

We can then serialize that:

```
(with-output-to-string (s)
   (with-serializer-output s
     (with-serializer :json
       (serialize *element*))))
=>

"{\"id\":22,\"realname\":\"Mike\",\"groups\":[{\"id\":33,\"title\":\"My group\"}]}"
(with-output-to-string (s)
  (with-serializer-output s
```

```
      (with-serializer :xml
        (serialize *element*))))
=>

"<user><id>22</id><realname>\"Mike\"</realname><groups><group><id>33</id><title>\"My g
```

4.3 Generic streaming serialization API

Insted of building an intermediate representation first, and then serializing it, we can serialize our data directly to a stream via an abstract generic interface.

The macros to be used in this case are `with-element`, `with-attribute` and `set-attribute`, `with-elements-list`, `with-list-member`.

Example:

```
(defparameter *streamed-element*
  (lambda ()
    (with-element ("user")
      (set-attribute "id" 22)
      (with-attribute ("realname")
(serialize "Mike"))
      (with-attribute ("groups")
(with-elements-list ("groups")
  (with-list-member ("group")
    (with-element ("group")
      (set-attribute "id" 33)
      (set-attribute "title" "My group")))))))))

(with-output-to-string (s)
   (with-serializer-output s
     (with-serializer :json
       (funcall *streamed-element*))))

=>

"{\"id\":22,\"realname\":\"Mike\",\"groups\":[{\"id\":33,\"title\":\"My group\"}]}"
(with-output-to-string (s)
   (with-serializer-output s
     (with-serializer :xml
       (funcall *streamed-element*))))

=>

"<user><id>22</id><realname>\"Mike\"</realname><groups><group><group><id>33</id><title
```

5 Schemas

6 Error handling

Error handling is controlled by the `%with-condition-handling` function, that is supposed to be private. The user api is the `with-condition-handling` macro.

The way condition handling is done is controlled by the `*development-mode*` variable. Its valid values are `:development`, `:production` and `:testing`.

When in `:development` program errors and conditions are not serialized or returned to the server in any way, but they are left unhandled.

In `:testing` program errors are serialized to the server.

And in `:production`, program errors are not serialized, but the request status header is modify according to the error. In general, on a program error a 500 internal server error is returned. There are special conditions, like http-not-found-error, http-internal-server-error, http-authorization-required-error, http-forbidden-error, http-service-unavailable-error, http-unsupported-media-type-error (all http-error subclasses), that should be signaled for the client to get the correct HTTP status code.

There is also special harmless-condition. Conditions from that family (subclasses) are never handled by the default error handler.

7 Authentication

8 Example

First, let's define a very simple model to work with. It is a CRUD on application users:

```
(defpackage :model-test
  (:use :cl)
  (:export :get-user
   :all-users
   :add-user
   :update-user))

(in-package :model-test)

(defvar *users* nil)

(defun make-user (id realname)
  (list (cons :id id)
(cons :realname realname)))

(defun add-user (user)
  (push (cons (cdr (assoc :id user))
      user)
*users*))

(defun update-user (user)
  (let ((user-id (cdr (assoc :id user))))
    (delete-user user-id)
    (add-user user)))

(defun get-user (id)
  (cdr (assoc id *users*)))

(defun delete-user (id)
  (setf *users* (delete id *users* :test #'equalp :key #'first)))

(defun all-users ()
  (mapcar #'cdr *users*))
```

Now we can define an api to do CRUD operations to that model via HTTP:

```
(defpackage :api-test
  (:use :rest-server :cl))

(in-package :api-test)

(define-api api-test
  (:documentation "This is an api test"
```

```
 :content-types (list :json :xml))
(get-users (:method :get
            :content-types (list :json)
            :uri-prefix "/users"
            :documentation "Retrive the users list")
           (&optional (expand-groups :boolean nil "Expand groups if true")))
(get-user (:method :get
           :content-types (list :json)
           :uri-prefix "/users/{id}"
           :documentation "Retrive an user")
          ((id :string "The user id")
           &optional (expand-groups :boolean nil "Expand groups if true")))
(create-user (:method :post
              :content-types (list :json)
              :uri-prefix "/users"
              :documentation "Create a user")
             ())
(update-user (:method :put
              :content-types (list :json)
              :uri-prefix "/users/{id}"
              :documentation "Update a user")
             ((id :string "The user id")))
(delete-user (:method :delete
              :content-types (list :json)
              :uri-prefix "/users/{id}"
              :documentation "Delete a user")
             ((id :string "The user id"))))
```

Now we have the API defined, but it is not actually implemented yet. To implement it, we define a new package.

```
(defpackage :api-test-implementation
  (:use :cl :rest-server))

(in-package :api-test-implementation)

(implement-api-function get-users (&key (expand-groups nil))
  (declare (ignore expand-groups))
  (with-output-to-string (s)
    (with-serializer-output s
      (with-serializer (rest-server::accept-serializer)
(with-elements-list ("users")
  (loop for user in (model-test:all-users)
     do
       (with-element ("user")
 (set-attribute "id" (cdr (assoc :id user)))
 (set-attribute "realname" (cdr (assoc :realname user)))))))))
```

```
(implement-api-function (get-user :serialization t)
    (id &key (expand-groups nil))
  (declare (ignore expand-groups))
  (let ((user (model-test:get-user id)))
    (if (not user)
(error 'http-not-found-error)
; else
(element "user"
 (attribute "id" (cdr (assoc :id user)))
 (attribute "realname" (cdr (assoc :realname user)))))))

(defun create-user (posted-content)
  (format nil "Create user: ~A" posted-content))

(defun update-user (posted-content id)
  (format nil "Update user: ~A ~A" id posted-content))

(defun delete-user (id)
  (format nil "Delete user: ~A" id))
```

9 System reference

`rest-server:attribute` [Class]
Class precedence list: `attribute, standard-object, t`
Serializer intermediate representation element attribute class

`rest-server:element` [Class]
Class precedence list: `element, standard-object, t`
Serializer intermediate representation element class

`rest-server:serializable-class` [Class]
Class precedence list: `serializable-class, standard-class, class, specializer, metaobject, standard-object, t`
Metaclass for serializable objects

`rest-server:add-list-member` *name value* **&key** *serializer stream* [Function]
Serializes a list member

`rest-server:attribute` *name value* [Function]
Build an element attribute to be serialized

`rest-server:element` *name* **&rest** *attributes* [Function]
Build an element to be serialized

`rest-server:elements` *name* **&rest** *elements* [Function]
Build a list of elements to be serialized

`rest-server:find-api` *name* [Function]
Find api by name

`rest-server:find-schema` *name* **&optional** *errorp* [Function]
Find a schema definition by name

`rest-server:make-api-function` *name method options args* [Function]
Make an api function.

`rest-server:serializable-class-schema` *serializable-class* [Function]
Generate a schema using the serializable class meta info

`rest-server:serialize-with-schema` *schema input* [Generic Function]
&optional *serializer stream*
Serialize input using schema

`rest-server:serialize` *element* **&optional** *serializer stream* [Generic Function]
Main serialization function. Takes the element to serialize, the serializer and the output stream

`rest-server:set-attribute` *name value* **&key** *serializer stream* [Function]
Serializes an element attribute and value

`rest-server:start-api-documentation` *api address port* [Function]
Start a web documentation application on the given api.

`rest-server:start-api` *api address port* **&optional** [Function]
api-implementation-package
Start an api at address and port. api-implementation-package: is the package where the api-functions are implemented.

`rest-server:define-api-function` *name method options args* [Macro]
Helper macro to define an api function

`rest-server:define-api` *name options* **&body** *functions* [Macro]
Define an api.

`rest-server:define-schema` *name schema* [Macro]
Define a schema

`rest-server:define-serializable-class` *name direct-superclasses* [Macro]
direct-slots **&rest** *options*
Helper macro to define serializable classes

`rest-server:implement-api-function` *name-and-options args* **&body** [Macro]
body
Define an api function implementation

`rest-server:with-api-backend` *backend* **&body** *body* [Macro]
Execute the client api function calling backend

`rest-server:with-api` *api* **&body** *body* [Macro]
Execute body under api scope. Example: (with-api test-api (define-api-function get-user :get (:url-prefix "users/{id}") '((:id :integer))))

`rest-server:with-attribute` *name* **&body** *body* [Macro]
Serializes an element attribute

`rest-server:with-element` *name* **&body** *body* [Macro]
Serializes a serializing element.

`rest-server:with-elements-list` *name* **&body** *body* [Macro]
Serializes an list of elements

`rest-server:with-list-member` *name* **&body** *body* [Macro]
Serializes a list member

`rest-server:with-serializer-output` *serializer-output* **&body** *body* [Macro]
Defines the serializer output when executing body.

Example:
(with-serializer-output s
(with-serializer :json (serialize user)))

`rest-server:with-serializer` *serializer* **&body** *body* [Macro]

Execute body in serializer scope. Binds *serializer* to serializer.

Example:
(with-serializer :json
(serialize user))

`rest-server:*development-mode*` [Variable]

Api development mode. One of :development, :testing, :production. Influences how errors are handled from the api

10 References

[Common Lisp Directory] [Common Lisp Wiki]

[Common Lisp Directory]: http://common-lisp.net [Common Lisp Wiki]: http://www.cliki.net

11 Index

11.1 Concept Index

11.2 Class Index

11.3 Function / Macro Index

11.4 Variable Index