Common Lisp REST Server Documentation
Release 0.2

Mariano Montone

November 19, 2014

Contents:

INTRODUCTION

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

1.1 Features

- Method matching - Based on HTTP method (GET, PUT, POST, DELETE) - Based on Accept request header - URL parsing (argument types)

- Serialization - Different serialization types (JSON, XML, S-expressions)

- Error handling - Development and production modes

- Validation via schemas

- Annotations for api logging, caching, permission checking, and more.

- Authentication - Different methods (token based, oauth)

- Documentation - Via Swagger: http://swagger.wordnik.com

INSTALL

Download the source code from https://github.com/mmontone/cl-rest-server and point *.asd* system definition files from
`./sbcl/system (ln -s <system definition file path>)` and then evaluate:

```
(require :rest-server)
```

from your lisp listener.

You will also need to satisfy these system dependencies:

- *alexandria*
- *cxml* and *cl-json* for the serialization module
- *cl-ppcre* for the validation module

The easiest way of installing those packages is via Quicklisp

This library is under the MIT licence.

API DEFINITION

APIs are defined using the `DEFINE-API` macro. APIs contain resources and resources contain api-functions.

macro (**define-api** *name options &body resources*)

> Define an api.

This is the syntax:

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

3.1 API options

- `:title`: The API title. This appears in the generated API documentation
- `:documentation`: A string with the API description. This appears in the generated API documentation.

3.2 Resources

Resources have the following syntax:

```
(<resource-name> <resource-options> <api-functions>)
```

Resources can be added to an already defined API via the :cl:function::*with-api* and `define-api-resource` macros

macro (**with-api** *api &body body*)

> Execute body under api scope.
>
> ```
> Example:
> (with-api test-api
>    (define-resource-operation get-user :get (:url-prefix "users/{id}")
>                               '((:id :integer))))
> ```

macro (**define-api-resource** *name options &body functions*)

> Define an api resource.

3.2.1 Resource options

- `:produces`: A list of content types produced by this resource. The content types can be `:json`, `:html`, `:xml`, `:lisp`

- `:consumes`: A list of content types consumed by this resource.

- `:documentation`: A string describing the resource. This appears in the generated API documentation.

- `:path`: The resource path. Should start with the `/` character. Ex: *"/users"*

- `:models`: A list of *models* used by the resource

3.3 Resource operations

Resources provide a set of operations to access them.

They have the following syntax:

```
(<resource-operation-name> <resource-operation-options> <resource-operation-arguments>)
```

New operations can be added to an already defined resource via the `with-api-resource`

macro (**with-api-resource** *resource &body body*)
Execute body under resource scope.

```
Example:
(with-api-resource users
   (define-resource-operation get-user :get (:url-prefix "users/{id}")
                              '((:id :integer))))
```

3.3.1 Resource operation options

- `:request-method`: The HTTP request method

- `:path`: The operation path. Arguments in the operation are enclosed between `{}`. For example: `"/users/{id}"`.

- `:produces`: A list of content types produced by the operation. The content types can be `:json`, `:html`, `:xml`, `:lisp`. This is matched with the HTTP "Accept" header.

- `:consumes`: A list of content types that the operation can consume.

- `:authorizations`: A list with the authorizations required for the operation. Can be one of `:token`, `:oauth`, `:oauth`, or a custom authorization type.

- `:documentation`: A string describing the operation. This appears in the generated API documentation.

3.3.2 Resource operation arguments

Arguments lists have the following syntax:

```
(*<required-arguments> &optional <optional-arguments>)
```

Required arguments are those appearing in the api function path between `{}`. They are specified like this:

```
(<argument-name> <argument-type> <documentation-string>)
```

Argument type can be one of: `string`, `integer`, `boolean`, `list`.

Optional arguments are those that can be passed after the `?` in the url. For instance, the `page` parameter in this url: `/users?page=1`. They are listed after the `&optional` symbol, and have the following syntax:

```
(<argument-name> <argument-type> <default-value> <documentation-string>)
```

Here is an example of an api function arguments list:

```
((id :integer "The user id")
 &optional (boolean :boolean nil "A boolean parameter")
           (integer :integer nil "An integer parameter")
           (string :string nil "A string parameter")
           (list :list nil "A list parameter"))
```

3.4 API example

Here is a complete example of an API interface:

```
(define-api api-test
    (:title "Api test"
            :documentation "This is an api test")
  (parameters (:produces (:json)
                          :consumes (:json)
                          :documentation "Parameters test"
                          :path "/parameters")
              (parameters (:produces (:json)
                                      :consumes (:json)
                                      :documentation "Parameters test"
                                      :path "/parameters")
                          (&optional (boolean :boolean nil "A boolean parameter")
                                     (integer :integer nil "An integer parameter")
                                     (string :string nil "A string parameter")
                                     (list :list nil "A list parameter"))))
  (users (:produces (:json :xml)
                    :consumes (:json)
                    :documentation "Users operations"
                    :models (user)
                    :path "/users")
         (get-users (:request-method :get
                                     :produces (:json)
                                     :path "/users"
                                     :documentation "Retrive the users list")
                    (&optional (page :integer 1 "The page")
                               (expand :list nil "Attributes to expand")))
         (get-user (:request-method :get
                                    :produces (:json)
                                    :path "/users/{id}"
                                    :documentation "Retrive an user")
                   ((id :integer "The user id")
                    &optional
                    (expand :list nil "Attributes to expand")))))
```

API IMPLEMENTATION

APIs need to implement its resources operations. This is done via the `implement-resource-operation` macro.

macro (**implement-resource-operation***api-name name-and-options args &body body*)
  Define an resource operation implementation

The required arguments of the resource operation appear as normal arguments in the function, in the order in which they were declared. The optional arguments of a resource operation appear as *&key* arguments of the function. In case the resource operation request method is either **PUT** or **POST**, then a ''posted-content' argument should be added to the implementation function as the first argument.

Some examples:

For this operation:

```
(get-user (:request-method :get
                     :produces (:json)
                     :path "/users/{id}"
                     :documentation "Retrieve an user")
                 ((id :integer "The user id")
                  &optional
                  (expand :list nil "Attributes to expand")))
```

The following resource implementation should be defined:

```
(implement-resource-operation get-user (id &key expand)
   (serialize (find-user id) :expand expand))
```

And for this POST operation:

```
(create-user (:request-method :post
                       :consumes (:json)
                       :path "/users"
                       :documentation "Create a user"
                       :body-type user)
             ())
```

The `posted-content` argument should be included:

```
(implement-resource-operation create-user (posted-content)
   (with-posted-content (name age) posted-content
       (serialize (model:create-user :name name :age age))))
```

4.1 Conditional dispatch

It is possible to dispatch to a particular resource operation implementation depending on the content type requested by the client in the HTTP Accept header via the `implement-resource-operation-case` macro.

function (**implement-resource-operation-case***name accept-content-type args &body body*)

Example:

```
(implement-resource-operation api-test::api-test
  api-test::conditional-dispatch ()
(error 'http-not-acceptable-error))

(implement-resource-operation-case
  api-test::conditional-dispatch "text/html"
  ()
"<p>Hello</p>")

(implement-resource-operation-case
  api-test::conditional-dispatch "application/json"
  ()
"\"hello\"")

(implement-resource-operation-case
  api-test::conditional-dispatch "application/xml"
  ()
"<p>Hello</p>")
```

STARTING THE API

APIs are started calling the function `start-api`

function (**start-api** *api address port &optional*) *development-mode *development-mode**
    configuration-args Start an api at address and port.

```
In production mode, we bind the api directly. In development mode, we only bind the ``API`` name
```

ACCESSING THE API

The `define-api` macro creates a function for accessing the api for each resource operation.

Before using the generated functions, the api backend needs to be selected via the `with-api-backend`.

macro (**with-api-backend** *backend &body body*)
　　Execute the client resource operation calling backend

For instance, for the api defined above, an `get-user` and a `get-users` functions are created, which can be used like this:

```
(with-api-backend "http://localhost/api"
   (get-user 22))
```

Assuming the api is running on http://localhost/api

ERROR HANDLING

APIs can be run with different error handling modes. This is controlled passing the desired mode to `start-api`. Can be one of `:development`, `:testing`, or `:production`. Default is `:production`.

variable `*development-mode*`

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

7.1 Production mode

In production mode, when an error occurs, 505 internal server error is returned.

7.2 Testing mode

In testing mode, when an error occurs the condition is serialized so it is possible to see what went wrong when accessing the API to some extent.

7.3 Development mode

In development mode, when an error occurs, the Lisp debugger is entered.

7.4 Global error mode

To setup a global error hanling mode, that has precedence to individual running apis error handling modes, assign one of `:development`, `:testing`, or `:production` to the `*SERVER-DEVELOPMENT-MODE*` variable.

variable `*server-development-mode*`

> Global server development mode. Takes precedence over *development-mode* when handling errors

API CONFIGURATION

Some aspects of the api can be configured either passing the configuration parameters to the `start-api` function, or via the `configure-api` function.

function (**configure-api***api-or-name &rest options*)
      Configure or reconfigure an already existent api

8.1 CORS configuration

APIs can be configured to append CORS headers to responses.

Syntax:

```
(configure-api api '(:cors &rest options))
```

8.1.1 Options:

- `:enabled`: Boolean. CORS enabled when `T`.

- `:allow-origin`: The "AllowOrigin" header. Default: `"*"`

- `:allow-headers`: A list. The "AllowHeaders" header.

- `:allow-methods`: A list. The "AllowMethods" header. Default: `(list :get :put :post :delete)`

8.2 Logging configuration

Log api requests and responses.

Syntax:

```
(configure-api '(:logging &rest options))
```

Then evaluate :cl:function::*start-api-logging*

function (**start-api-logging**)

API DOCUMENTATION

There's an (incomplete) implementation of a Swagger export.

First, configure the api for Swagger:

```
(define-swagger-resource api)
```

This will enable CORS on the API, as Swagger needs it to make requests.

After this you can download the Swagger documentation tool and point to the api HTTP address.

API

Rest Server external symbols documentation

function (**configure-api-resource***api-or-name resource-name &rest options*)

macro (**permission-checking***args resource-operation-implementation*)

function (**accept-serializer**)

macro (**serialization***args resource-operation-implementation*)

macro (**with-list-member**) *name &key*
*serializer 'rest-server::*serializer*stream 'rest-server::*serializer-output* &body body* Serializes a list member

macro (**implement-resource-operation-case***name accept-content-type args &body body*)
Implement an resource operation case

macro (**with-api***api &body body*)
Execute body under api scope.

```
Example:
(with-api test-api
   (define-resource-operation get-user :get (:url-prefix "users/{id}")
                                   '((:id :integer))))
```

macro (**with-api-backend***backend &body body*)
Execute the client resource operation calling backend

macro (**implement-resource-operation***api-name name-and-options args &body body*)
Define an resource operation implementation

function (**set-reply-content-type***content-type*)

macro (**with-serializer-output***serializer-output &body body*)
Defines the serializer output when executing body.

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

function (**http-error***status-code*)

macro (**define-schema***name schema*)
Define a schema

function (**disable-api-logging***api-name &optional*) *stop t*

function (**format-absolute-resource-operation-url***resource-operation &rest args*)

function (**boolean-value***boolean &optional*) *serializer *serializer**
    stream *serializer-output*

function (**start-api-documentation***api address port*)
    Start a web documentation application on the given api.

function (**list-value***list &optional*) *serializer *serializer**
    stream *serializer-output*

function (**find-schema***name &optional*) *errorp t*
    Find a schema definition by name

macro (**with-xml-reply***&body body*)

function (**self-reference***&rest args*)

macro (**unserialization***args resource-operation-implementation*)

function (**find-api***name &key*) *error-p t*
    Find api by name

macro (**fetch-content***args resource-operation-implementation*)

function (**serializable-class-schema***serializable-class*)
    Generate a schema using the serializable class meta info

function (**stop-api***api-acceptor*)

function (**make-resource-operation***name attributes args options*)
    Make an resource operation.

function (**configure-resource-operation-implementation***name &rest options*)
    Configure or reconfigure an already existent resource operation implementation

function (**configure-api***api-or-name &rest options*)
    Configure or reconfigure an already existent api

function (**validation-error***message &rest args*)

function (**stop-api-logging**)

function (**elements***name &rest elements*)
    Build a list of elements to be serialized

macro (**logging***args resource-operation-implementation*)

function (**start-api***api address port &optional*) *development-mode *development-mode**
    configuration-args Start an api at address and port.

    In production mode, we bind **the** api directly. In development mode, we only bind **the** ``API`` name

function (**set-attribute***name value &rest args &key*) *serializer *serializer**
    stream *serializer-output* &allow-other-keys Serializes an element attribute and value

function (**add-list-member***name value &key*) *serializer *serializer**
    stream *serializer-output* Serializes a list member

macro (**with-attribute**) *name &key*
    serializer 'rest-server::*serializer*stream 'rest-server::*serializer-output* &body body Serializes an element
    attribute

macro (**with-json-reply***&body body*)

macro (**with-list**) *name &key*
    serializer 'rest-server::*serializer*stream 'rest-server::*serializer-output* &body body Serializes an list of el-
    ements

---

macro (**define-resource-operation***name attributes args &rest options*)
> Helper macro to define an resource operation

macro (**schema***schema-def*)

function (**enable-api-logging***api-name &optional*) *start t*

macro (**define-serializable-class***name direct-superclasses direct-slots &rest options*)
> Helper macro to define serializable classes

macro (**validation***args resource-operation-implementation*)

macro (**error-handling***args resource-operation-implementation*)

macro (**with-permission-checking***check &body body*)

macro (**with-serializer***serializer &body body*)
> Execute body in serializer scope. Binds *serializer* to serializer.
>
> ```
> Example:
> (with-serializer :json
>   (serialize user))
> ```

macro (**define-api-resource***name options &body functions*)
> Define an api resource.

function (**start-api-logging**)

macro (**define-swagger-resource***api-name*)

macro (**with-element**) *name &key*
> *serializer 'rest-server::*serializer*stream 'rest-server::*serializer-output* &body body* Serializes a serializing
> element.

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

macro (**caching***args resource-operation-implementation*)

macro (**with-api-resource***resource &body body*)
> Execute body under resource scope.
>
> ```
> Example:
> (with-api-resource users
>     (define-resource-operation get-user :get (:url-prefix "users/{id}")
>                                '((:id :integer))))
> ```

macro (**with-content**) *var content*
> *&body body*

macro (**with-pagination**) *&rest args &key*
> *pageerror "Provide the page"element-name "pagination" &allow-other-keys &body body*

macro (**define-api***name options &body resources*)
> Define an api.

function (**element***name &rest attributes*)
> Build an element to be serialized

function (**attribute***name value &optional type formatter*)
> Build an element attribute to be serialized

variable *server-development-mode*
> Global server development mode. Takes precedence over *development-mode* when handling errors

macro (**with-reply-content-type**) *content-type*
    &body body

macro (**with-posted-content***args posted-content &body body*)

INDICES AND TABLES

- *genindex*
- *search*