Common Lisp REST Server Documentation

Common Lisp REST Server 0.2, Feb 15, 2022

Mariano Montone

Table of Contents

1 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 - HTTP error codes - 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`'

2 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[1]

This library is under the MIT licence.

[1] http://www.quicklisp.org

3 API definition

APIs are defined using the [DEFINE-API], page 3, macro. APIs contain resources and resources contain api-functions.

`(define-apiname` *superclasses options* **&body** *resources)* [Common Lisp Macro]
Define an api.

This is the syntax:

```
(define-api <api-name> (&rest <superclasses>) <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], page 3, macros

`(with-apiapi` **&body** *body)* [Common Lisp Macro]
Execute body under api scope.
Example: (with-api test-api

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

`(define-api-resourcename` *options* **&body** *functions)* [Common Lisp Macro]
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-argument
```

New operations can be added to an already defined resource via the [with-api-resource], page 4,

`(with-api-resourceresource` **&body** *body)* [Common Lisp Macro]

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")))))
```

4 API implementation

APIs need to implement its resources operations. This is done via the [implement-resource-operation], page 6, macro.

`(implement-resource-operationapi-name` [Common Lisp Macro]
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 "Retrive 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))))
```

5 Starting the API

APIs are started calling the function [start-api], page 7,

`(start-apiapi` **&rest** *args)* [Common Lisp Function]

Start an api at address and port.

In production mode, we bind the api directly. In debug mode, we only bind the API name in order to be able to make modifications to the api (definition) in development time

6 Accessing the API

The [define-api], page 3, 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], page 8.

`(with-api-backendbackend` **&body** *body)* [Common Lisp Macro]

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`'

7 Error handling

APIs can be run with different error handling modes. This is controlled via the argument *:catch-errors* in [start-api], page 7. Default is NIL.

`*catch-errors*` [Common Lisp Variable]
If T, then the error is serialize and the corresponding HTTP is returned. Otherwise, when an error occurs, the Lisp debugger is entered.

7.1 Global error mode

To setup a global error handling mode, that has precedence to individual running apis error handling modes, set [*SERVER-CATCH-ERRORS*], page 9, variable.

`*server-catch-errors*` [Common Lisp Variable]

8 API configuration

Some aspects of the api can be configured either passing the configuration parameters to the [start-api], page 7, function, or via the [configure-api], page 10, function.

`(configure-apiapi-or-name &rest` *options)* [Common Lisp Function]

Configure or reconfigure an already existent api

8.1 CORS configuration

APIs can be configured to append CORS[1] 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*

`(start-api-logging)` [Common Lisp Function]

[1] https://developer.mozilla.org/en-US/docs/Web/HTTP/Access_control_CORS

9 API documentation

There's an (incomplete) implementation of a Swagger[1] export.

First, configure the api for Swagger:

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

This will enable CORS[2] 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.

[1] https://helloreverb.com/developers/swagger

[2] https://developer.mozilla.org/en-US/docs/Web/HTTP/Access_control_CORS

10 API

Rest Server external symbols documentation

`(configure-api-resourceapi-or-name` *resource-name* **&rest** *options)* [Common Lisp Function]

`(permission-checkingargs` *resource-operation-implementation*`)` [Common Lisp Macro]

`(accept-serializer)` [Common Lisp Function]

`serialization` [Common Lisp Macro]

`with-list-member` [Common Lisp Macro]

`(implement-resource-operation-casename` *accept-content-type args* **&body** *body)* [Common Lisp Macro]

Implement an resource operation case

`(with-apiapi` **&body** *body)* [Common Lisp Macro]

Execute body under api scope.
Example: (with-api test-api

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

`(with-api-backendbackend` **&body** *body)* [Common Lisp Macro]

Execute the client resource operation calling backend

`(implement-resource-operationapi-name` *name-and-options args* **&body** *body)* [Common Lisp Macro]

Define an resource operation implementation

`(set-reply-content-typecontent-type)` [Common Lisp Function]

`with-serializer-output` [Common Lisp Macro]

`(http-error)` [Common Lisp Function]

`define-schema` [Common Lisp Macro]

`(disable-api-logging)` [Common Lisp Function]

`(format-absolute-resource-operation-urlresource-operation` **&rest** *args)* [Common Lisp Function]

`(boolean-value)` [Common Lisp Function]

`(start-api-documentationapi` *address port*`)` [Common Lisp Function]

Start a web documentation application on the given api.

`(list-value)` [Common Lisp Function]

`(find-schema)` [Common Lisp Function]

`(with-xml-reply&body` *body*`)` [Common Lisp Macro]

`(self-reference&rest` *args*) [Common Lisp Function]

`unserialization` [Common Lisp Macro]

`(find-apiname` **&key** (*error-p* **t**)*)* [Common Lisp Function]
Find api by name

`fetch-content` [Common Lisp Macro]

`(serializable-class-schema)` [Common Lisp Function]

`(stop-apiapi-acceptor)` [Common Lisp Function]

`(make-resource-operationname` *attributes args options*) [Common Lisp Function]
Make an resource operation.

`(configure-resource-operation-implementationname` **&rest** *options)* [Common Lisp Function]
Configure or reconfigure an already existent resource operation implementation

`(configure-apiapi-or-name` **&rest** *options)* [Common Lisp Function]
Configure or reconfigure an already existent api

`(validation-error)` [Common Lisp Function]

`(stop-api-logging)` [Common Lisp Function]

`(elements)` [Common Lisp Function]

`logging` [Common Lisp Macro]

`(start-apiapi` **&rest** *args)* [Common Lisp Function]
Start an api at address and port.

> In production mode, we bind the api directly. In debug mode, we only bind the API name in order to be able to make modifications to the api (definition) in development time

`(set-attribute)` [Common Lisp Function]

`(add-list-member)` [Common Lisp Function]

`with-attribute` [Common Lisp Macro]

`(with-json-reply&body` *body*) [Common Lisp Macro]

`with-list` [Common Lisp Macro]

`(define-resource-operationname` *attributes args* **&rest** *options)* [Common Lisp Macro]
Helper macro to define an resource operation

`schema` [Common Lisp Macro]

`(enable-api-logging)` [Common Lisp Function]

`define-serializable-class` [Common Lisp Macro]

`validation` [Common Lisp Macro]

`error-handling` [Common Lisp Macro]

`(with-permission-checkingcheck` **&body** *body)* [Common Lisp Macro]

`with-serializer` [Common Lisp Macro]

`(define-api-resourcename` *options* **&body** *functions)* [Common Lisp Macro]
Define an api resource.

`(start-api-logging)` [Common Lisp Function]

`define-swagger-resource` [Common Lisp Macro]

`with-element` [Common Lisp Macro]

`*catch-errors*` [Common Lisp Variable]

`(cachingargs` *resource-operation-implementation*`)` [Common Lisp Macro]

`(with-api-resourceresource` **&body** *body)* [Common Lisp Macro]
Execute body under resource scope.
Example: (with-api-resource users
(define-resource-operation get-user :get
(:url-prefix "users/{id}")
'((:id :integer))))

`(with-content` (**&key** *(setter)*) **&body** *body)* [Common Lisp Macro]
Macro to build HTTP content to pass in client functions.
Example:
(with-api-backend *api-backend*
(let ((content (with-content ()
(:= :name "name") (when some-condition
(:= :attr 22)))))
(app.api-client:my-client-function :content content)))

`(with-pagination` (**&rest** *args* **&key** *(page)* *(object-name)* **&allow-other-keys**) **&body** *body)* [Common Lisp Macro]

`(define-apiname` *superclasses options* **&body** *resources)* [Common Lisp Macro]
Define an api.

`(element)` [Common Lisp Function]

`(attribute)` [Common Lisp Function]

`*server-catch-errors*` [Common Lisp Variable]

`(with-reply-content-type` (*content-type*) **&body** *body)* [Common Lisp Macro]

`(with-posted-contentargs` *posted-content* **&body** *body)* [Common Lisp Macro]
Bind `ARGS` to `POSTED-CONTENT`. `POSTED-CONTENT` is supposed to be an alist. Also, argx-P is T[1] iff argx is present in `POSTED-CONTENT`

[1] http://www.lispworks.com/reference/HyperSpec/Body/a_t.htm

Indices and tables

* genindex
* search

Index

W