Common Lisp ODATA Client

Mariano Montone (marianomontone@gmail.com)

Table of Contents

1 Introduction

CL-FORMS is a web forms handling library for Common Lisp.

Although it is potentially framework agnostic, it runs on top of Hunchentoot at the moment.

It features:

- Several form field types: String, boolean, integer, email, password fields. And more.
- Custom fields. CL-FORMS is extensible and it is possible to define new field types.
- Server and client side validation
- Rendering backends. Forms can be rendered via CL-WHO, or Djula, or something else; the backend is pluggable. The default renderer is CL-WHO.
- Themes (like Bootstrap)
- Control on rendering and layout.
- Handling of form errors.
- CSRF protection

2 Installation

3 Usage

3.1 Basics

Use [DEFFORM], page 4 to define a form. Example:

```
(defform fields-form (:action "/fields-post")
  ((name :string :value "")
   (ready :boolean :value t)
   (sex :choice :choices (list "Male" "Female") :value "Male")
   (submit :submit :label "Create")))
```

On your web handler, grab the form via ‘get-form‘, select a renderer with ‘with-form-renderer‘and then render the form with ‘render-form‘:

```
(let ((form (forms::get-form 'fields-form)))
   (forms:with-form-renderer :who
      (forms:render-form form))
```

To handle the form, grab it via ‘get-form‘ and then call ‘handle-request‘ (you should probably also call ‘validate-form‘ after). Then bind form fields via either ‘with-form-field-values‘, that binds the form field values; or ‘with-form-fields‘ that binds the form fields.

```
(let ((form (forms:get-form 'fields-form)))
    (forms::handle-request form)
    (forms::with-form-field-values (name ready sex) form
       (who:with-html-output (forms.who::*html*)
          (:ul
            (:li (who:fmt "Name: ~A" name))
            (:li (who:fmt "Ready: ~A" ready))
            (:li (who:fmt "Sex: ~A" sex))))))
```

Plase have a look at the demo sources for more examples of how to use the library

3.2 Demo

There's a demo included. To run:

```
(require :cl-forms.demo)
(forms.test:run-demo)
```

4 API

4.1 CL-FORMS package

`CL-FORMS` [PACKAGE]

External definitions

Variables

`*BASE64-ENCODE*` [CL-FORMS]
If , encode form parameters in base64

Macros

`CL-FORMS:DEFFORM-BUILDER` (*form-name args* **&body** *body*) [Macro]

`CL-FORMS:WITH-FORM-RENDERER` (*renderer* **&body** *body*) [Macro]
Bind ⟨undefined⟩ [*FORM-RENDERER*], page ⟨undefined⟩ to *RENDERER* and evaluate *BODY* in that context.

`CL-FORMS:WITH-FORM-THEME` (*form-theme* **&body** *body*) [Macro]
Bind ⟨undefined⟩ [*FORM-THEME*], page ⟨undefined⟩ to *FORM-THEME* and evaluate *BODY* in that context.

`CL-FORMS:WITH-FORM` (*form* **&body** *body*) [Macro]
Bind ⟨undefined⟩ [*FORM*], page ⟨undefined⟩ to *FORM* and evaluate *BODY* in that context.

`CL-FORMS:DEFFORM` (*form-name args fields*) [Macro]
Define a form at top-level

`CL-FORMS:WITH-FORM-FIELDS` (*fields form* **&body** *body*) [Macro]

`CL-FORMS:WITH-FORM-TEMPLATE` ((**&optional** *form-var) form-name args* **&body** *body*) [Macro]

`CL-FORMS:WITH-FORM-FIELD-VALUES` (*fields form* **&body** *body*) [Macro]

Generic functions

`CL-FORMS:FIELD-FORMATTER` (*sb-pcl::object*) [Generic-Function]

`CL-FORMS:FIELD-PARSER` (*sb-pcl::object*) [Generic-Function]

`CL-FORMS:FIELD-VALID-P` (*form-field* **&optional** *(form)*) [Generic-Function]

`CL-FORMS:FIELD-READER` (*field*) [Generic-Function]

`CL-FORMS:FIELD-WRITER` (*field*) [Generic-Function]

`CL-FORMS:FORMAT-FIELD-VALUE` (*form-field field-value* **&optional** *stream*) [Generic-Function]

`CL-FORMS:FIELD-VALUE` (*field*) [Generic-Function]

`CL-FORMS:FIELD-ACCESSOR` (*sb-pcl::object*) [Generic-Function]

`CL-FORMS:FORM-ERRORS` (*sb-pcl::object*) [Generic-Function]

Functions

`CL-FORMS:GET-FORM` (*name* **&rest** *args*) [Function]

`CL-FORMS:GET-FIELD` (*form field-name* **&optional** *(error-p t)*) [Function]

`CL-FORMS:HANDLE-REQUEST` (**&optional** *(form *form*) (request hunchentoot:*request*)*) [Function]

Populates *FORM* from parameters in HTTP *request*. After this, the *form* field contains values, but they are not validated. To validate call [VALIDATE-FORM], page 5 after.

`CL-FORMS:RENDER-FIELD-ERRORS` (*field* **&optional** *(form *form*)* **&rest** *args*) [Function]

`CL-FORMS:RENDER-FORM` (**&optional** *(form *form*)* **&rest** *args*) [Function]

`CL-FORMS:RENDER-FORM-START` (**&optional** *(form *form*)* **&rest** *args*) [Function]

`CL-FORMS:ADD-FORM-ERROR` (*field error-msg* **&optional** *(form *form*)*) [Function]

Add an error on *FIELD*

`CL-FORMS:RENDER-FORM-ERRORS` (**&optional** *(form *form*)* **&rest** *args*) [Function]

`CL-FORMS:RENDER-FIELD-WIDGET` (*field* **&optional** *(form *form*)* **&rest** *args*) [Function]

`CL-FORMS:FILL-FORM-FROM-MODEL` (*form model*) [Function]

Fill a *FORM* from a *MODEL*

`CL-FORMS:VALIDATE-FORM` (**&optional** *(form *form*)*) [Function]

Validates a *form*. Usually called after [HANDLE-REQUEST], page 5. Returns multiple values; first value is true if the *form* is valid; second value a list of errors.
The list of errors is an association list with elements (<field> . <field errors strings list>).

`CL-FORMS:MAKE-FORMATTER` (*symbol*) [Function]

Create a field formatter. *SYMBOL* is the function to call.

`CL-FORMS:RENDER-FIELD-LABEL` (*field* **&optional** *(form *form*)* **&rest** *args*) [Function]

`CL-FORMS:ADD-FIELD` (*form field*) [Function]

`CL-FORMS:REMOVE-FIELD` (*form field*) [Function]

`CL-FORMS:GET-FIELD-VALUE` (*form field-name* **&optional** *(error-p t)*) [Function]

`CL-FORMS:FORM-VALID-P` (*form*) [Function]

`CL-FORMS:RENDER-FIELD` (*field* **&optional** *(form *form*)* **&rest** *args*) [Function]

`CL-FORMS:FILL-MODEL-FROM-FORM` (*form model*) [Function]
Set a *MODEL*'s values from *FORM* field values

`CL-FORMS:SET-FIELD-VALUE` (*form field-name value*) [Function]

`CL-FORMS:FORMAT-FIELD-VALUE-TO-STRING` (*form-field* **&optional** *(field-value (field-value form-field))*) [Function]

`CL-FORMS:RENDER-FORM-END` (**&optional** *(form *form*)*) [Function]

Classes

`CL-FORMS:FORM` [Class]
A form

Class precedence list: `form, standard-object, t`

Slots:

- `id` — initarg: `:id`; reader: `cl-forms::form-id`; writer: `(setf cl-forms::form-id)`
  The form id
- `name` — initarg: `:name`; reader: `cl-forms::form-name`; writer: `(setf cl-forms::form-name)`
  The form name
- `action` — initarg: `:action`; reader: `cl-forms::form-action`; writer: `(setf cl-forms::form-action)`
  The form action
- `method` — initarg: `:method`; reader: `cl-forms::form-method`; writer: `(setf cl-forms::form-method)`
  The form method
- `enctype` — initarg: `:enctype`; reader: `cl-forms::form-enctype`; writer: `(setf cl-forms::form-enctype)`
  Form encoding type. i.e. Use multipart/form-data for file uploads
- `fields` — initarg: `:fields`; reader: `cl-forms::form-fields`; writer: `(setf cl-forms::form-fields)`
  Form fields
- `model` — initarg: `:model`; reader: `cl-forms::form-model`; writer: `(setf cl-forms::form-model)`
  The form model object
- `csrf-protection` — initarg: `:csrf-protection`; reader: `cl-forms::form-csrf-protection-p`; writer: `(setf cl-forms::form-csrf-protection-p)`
  T when csrf protection is enabled

- `csrf-field-name` — initarg: `:csrf-field-name`; reader: `cl-forms::form-csrf-field-name`; writer: `(setf cl-forms::form-csrf-field-name)`

  csrf field name
- `errors` — reader: `cl-forms:form-errors`; writer: `(setf cl-forms:form-errors)`

  Form errors after validation. An association list with elements (<field> . <field errors strings list>).
- `display-errors` — initarg: `:display-errors`; reader: `cl-forms::display-errors`; writer: `(setf cl-forms::display-errors)`

  A list containing the places where to display errors. Valid options are :list and :inline
- `client-validation` — initarg: `:client-validation`; reader: `cl-forms::client-validation`; writer: `(setf cl-forms::client-validation)`

  When T, form client validation is enabled

`CL-FORMS:FORM-FIELD` [Class]

A form field

Class precedence list: `form-field, standard-object, t`

Slots:

- `name` — initarg: `:name`; reader: `cl-forms::field-name`; writer: `(setf cl-forms::field-name)`

  The field name
- `label` — initarg: `:label`; reader: `cl-forms::field-label`; writer: `(setf cl-forms::field-label)`

  The field label
- `value` — initarg: `:value`

  Field value
- `default-value` — initarg: `:default-value`; reader: `cl-forms::field-default-value`; writer: `(setf cl-forms::field-default-value)`

  Value to use when the field value is nil
- `placeholder` — initarg: `:placeholder`; reader: `cl-forms::field-placeholder`; writer: `(setf cl-forms::field-placeholder)`

  Field placeholder (text that appears when the field is empty)
- `help-text` — initarg: `:help-text`; reader: `cl-forms::field-help-text`; writer: `(setf cl-forms::field-help-text)`

  Field help text
- `parser` — initarg: `:parser`; reader: `cl-forms:field-parser`; writer: `(setf cl-forms:field-parser)`

  Custom field value parser
- `formatter` — initarg: `:formatter`; reader: `cl-forms:field-formatter`; writer: `(setf cl-forms:field-formatter)`

  The field formatter. The function takes two arguments, a VALUE and STREAM to format it into.

- `constraints` — initarg: `:constraints`; reader: `cl-forms::field-constraints`; writer: `(setf cl-forms::field-constraints)`

  The field constraints
- `required` — initarg: `:required-p`; reader: `cl-forms::field-required-p`; writer: `(setf cl-forms::field-required-p)`

  Whether the field is required
- `required-message` — initarg: `:required-message`; reader: `cl-forms::field-required-message`; writer: `(setf cl-forms::field-required-message)`

  Message to display when field is required
- `invalid-message` — initarg: `:invalid-message`; reader: `cl-forms::field-invalid-message`; writer: `(setf cl-forms::field-invalid-message)`

  Message to display when field is invalid
- `read-only` — initarg: `:read-only-p`; reader: `cl-forms::field-read-only-p`; writer: `(setf cl-forms::field-read-only-p)`

  Whether the field is read only
- `disabled` — initarg: `:disabled-p`; reader: `cl-forms::field-disabled-p`; writer: `(setf cl-forms::field-disabled-p)`

  Whether the field is disabled
- `accessor` — initarg: `:accessor`; reader: `cl-forms:field-accessor`; writer: `(setf cl-forms:field-accessor)`

  The field accessor to the underlying model
- `reader` — initarg: `:reader`

  The function to use to read from the underlying model
- `writer` — initarg: `:writer`

  The function to use to write to underlying model
- `trim` — initarg: `:trim-p`; reader: `cl-forms::field-trim-p`; writer: `(setf cl-forms::field-trim-p)`

  Trim the input
- `validation-triggers` — initarg: `:validation-triggers`; reader: `cl-forms::field-validation-triggers`; writer: `(setf cl-forms::field-validation-triggers`

  Client side validation triggers. A list of :change, :focus, :focusout, :focusin, etc
- `form` — initarg: `:form`; reader: `cl-forms::field-form`; writer: `(setf cl-forms::field-form)`

  The form the field belongs to

5 Index

(Index is nonexistent)

*

C