# CL-FORMS

**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 18 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], page 19, select a renderer with ‘with-form-renderer‘and then render the form with [RENDER-FORM], page 18:

```
(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], page 19 and then call [HANDLE-REQUEST], page 19 (you should probably also call [VALIDATE-FORM], page 19 after). Then bind form fields via either [WITH-FORM-FIELD-VALUES], page 17, that binds the form field values; or [WITH-FORM-FIELDS], page 17 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))))))
```

Please 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)
```

### 3.2.1 Basic example

Define a form. Render the form via CL-WHO backend, doing:

```
(forms:with-form-renderer :who
   (forms:render-form form))
```

Then handle and validate the form.

Source code:

```
(in-package :forms.test)

(forms:defform fields-form (:action "/fields-post"
                                    :enctype "multipart/form-data")
  ((name :string :value "")
   (ready :boolean :value t)
   (sex :choice :choices (list "Male" "Female") :value "Male")
   (avatar :file :upload-handler 'handle-file-upload)
   (disabled :string :disabled-p t :required-p nil)
   (readonly :string :read-only-p t :required-p nil)
   (readonly-checkbox :boolean :read-only-p t :required-p nil)
   (disabled-checkbox :boolean :disabled-p t :required-p nil)
   (submit :submit :label "Create")))

(defun fields-demo ()
  (who:with-html-output (forms.who:*html*)
    (:h1 (who:str "Fields example"))
    (:div :class :container
          (:div :class :row
                (:div :class :heading
                      (:h3 (who:str "Simple form")))
                (let ((form (forms::get-form 'fields-form)))
                  (forms:with-form-renderer :who
                    (forms:render-form form))))
          (:div :class :row
                (:div :class :heading
                      (:h3 (who:str "Choices")))
                (let ((form (forms::get-form 'choices-form)))
                  (forms:with-form-renderer :who
                    (forms:render-form form)))))))

(hunchentoot:define-easy-handler (fields-demo-handler :uri "/fields") ()
  (render-demo-page :demo #'fields-demo
                    :source (asdf:system-relative-pathname :cl-forms.demo
                                                           "test/demo/fields.lisp")
                    :active-menu :fields))

(hunchentoot:define-easy-handler (fields-form-post
                                  :uri "/fields-post"
                                  :default-request-type :post)
    ()
  (flet ((fields-post ()
           (let ((form (forms:get-form 'fields-form)))
             (forms::handle-request form)
             (forms::with-form-fields (name ready sex avatar) form
               (who:with-html-output (forms.who:*html*)
                 (:ul
                  (:li (who:fmt "Name: ~A" (forms::field-value name)))
                  (:li (who:fmt "Ready: ~A" (forms::field-value ready)))
                  (:li (who:fmt "Sex: ~A" (forms::field-value sex)))
```

```
                   (:li (who:fmt "Avatar: ~A" (forms::file-name avatar))
                        (when (forms::file-name avatar)
                          (who:htm
                           (:img :width 200 :height 200
                                 :src (format nil "/files?f=~A" (forms::file-▮
name avatar))))))))))))
    (render-demo-page :demo #'fields-post
                      :source (asdf:system-relative-pathname :cl-forms.demo▮
                                                             "test/demo/fields.lisp")▮
                      :active-menu :fields)))

;; Choices widget test

(forms:defform choices-form (:action "/choices-post")
  ((sex :choice
        :choices (list "Male" "Female")
        :value "Male")
   (sex2 :choice
         :choices (list "Male" "Female")
         :value "Female"
         :expanded t)
   (choices :choice
            :choices (list "Foo" "Bar")
            :value (list "Foo")
            :multiple t)
   (choices2 :choice
             :choices (list "Foo" "Bar")
             :value (list "Bar")
             :multiple t
             :expanded  t)
   (submit :submit :label "Ok")))

(hunchentoot:define-easy-handler (choices-form-post :uri "/choices-post"
                                                    :default-request-type :post) ()▮
  (flet ((choices-post ()
           (let ((form (forms:get-form 'choices-form)))
             (forms::handle-request form)
             (forms::validate-form form)
             (forms::with-form-field-values (sex sex2 choices choices2) form▮
               (who:with-html-output (forms.who:*html*)
                 (:ul
                  (:li (who:fmt "Sex: ~A" sex))
                  (:li (who:fmt "Sex2: ~A" sex2))
                  (:li (who:fmt "Choices: ~A" choices))
                  (:li (who:fmt "Choices2: ~A" choices2))))))))
    (render-demo-page :demo #'choices-post
                      :source (asdf:system-relative-pathname :cl-forms.demo▮
                                                                          "test/demo/fields.lisp")
                           :active-menu :fields)))

;; File handling

(defvar *files* nil)
(defvar *files-path* (pathname "/tmp/cl-forms/"))

(defun handle-file-upload (file-field)
  ;; Store the file
  (let ((new-path (merge-pathnames
                    (forms::file-name file-field)
                    *files-path*)))
    (rename-file (forms::file-path file-field)
                 (ensure-directories-exist new-path))
    ;; Save for handler
    (push (cons (forms::file-name file-field)
                (list new-path (forms::file-content-type file-field)))
          *files*)))

(defun handle-uploaded-file ()
  (let ((finfo (cdr (assoc (hunchentoot:parameter "f") *files* :test #'equalp))))
    (hunchentoot:handle-static-file (first finfo) (second finfo))))

(push
 (hunchentoot:create-prefix-dispatcher "/files" 'handle-uploaded-file)
 hunchentoot:*dispatch-table*)
```

### 3.2.2 Validation

Example of forms validation.

Add Clavier constraints to the form. Then call [VALIDATE-FORM], page 19 after [HANDLE-REQUEST], page 19.

```
(in-package :forms.test)

(forms:defform validated-form (:action "/validation-post"
                                       :client-validation nil)
  ((name :string :value "" :constraints (list (clavier:is-a-string)
                                              (clavier:not-blank)
                                              (clavier:len :max 5)))
   (single :boolean :value t)
   (sex :choice :choices (list "Male" "Female") :value "Male")
   (age :integer :constraints (list (clavier:is-an-integer)
                                    (clavier:greater-than -1)
                                    (clavier:less-than 200)))
   (email :email)
   (birth-date :date :required-p nil)
```

```
    (submit :submit :label "Create")))

(defun validation-demo (&optional form)
  (forms:with-form-renderer :who
    (who:with-html-output (forms.who::*html*)
      (:h1 (who:str "Server side validation"))
      (:p (who:str "This is a demo of server side validation. Submit the form and play
ues to see how it works. Also look at field constraints in source code tab."))
      (let ((form (or form (forms::get-form 'validated-form))))
        (forms:render-form form)))))

(hunchentoot:define-easy-handler (validated-form-post :uri "/validation-
post"
                                                        :default-request-
type :post) ()

  (flet ((validation-post ()
           (let ((form (forms:get-form 'validated-form)))
             (forms::handle-request form)
             (if (forms::validate-form form)
                 ;; The form is valid
                 (forms::with-form-field-values (name single sex age email birth-
date) form
                   (who:with-html-output (forms.who::*html*)
                     (:ul
                      (:li (who:fmt "Name: ~A" name))
                      (:li (who:fmt "Single: ~A" single))
                      (:li (who:fmt "Sex: ~A" sex))
                      (:li (who:fmt "Age: ~A" age))
                      (:li (who:fmt "Email: ~A" email))
                      (:li (who:fmt "Birth date: ~A" birth-date)))))
                 ;; The form is not valid
                 (validation-demo form)))))
    (render-demo-page :demo #'validation-post
                      :source (asdf:system-relative-pathname :cl-forms.demo
                                                             "test/demo/validation.lis
                      :active-menu :validation)))

(hunchentoot:define-easy-handler (validation-demo-handler :uri "/valida-
tion") ()
  (render-demo-page :demo #'validation-demo
                    :source (asdf:system-relative-pathname :cl-forms.demo
                                                           "test/demo/validation.lisp"
                    :active-menu :validation))
```

### 3.2.3 Client validation

To validate in the client, just set `:client-validation` to `T`.

```
(in-package :forms.test)

(forms:defform client-validated-form (:action "/client-validation-post"
                                              :client-validation t)
  ((name :string :value "" :constraints (list (clavier:is-a-string)
                                              (clavier:not-blank)
                                              (clavier:len :max 5))
         :validation-triggers '(:focusin))
   (single :boolean :value t)
   (sex :choice :choices (list "Male" "Female") :value "Male")
   (age :integer :constraints (list (clavier:is-an-integer)
                                    (clavier:greater-than -1)
                                    (clavier:less-than 200)))
   (email :email)
   (submit :submit :label "Create")))

(defun client-validation (&optional form)
  (let ((form (or form (forms::get-form 'client-validated-form))))
    (forms:with-form-renderer :who
      (who:with-html-output (forms.who::*html*)
        (:h1 (who:str "Client side validation"))
        (:p (who:str "This is an example of how client side validation works. Client s
        (:p (who:str "The interesting thing about the implementation is that validatio
        (forms:render-form form)))))

(hunchentoot:define-easy-handler (client-validation-handler
                                  :uri "/client-validation") ()
  (render-demo-page :demo #'client-validation
                    :source (asdf:system-relative-pathname :cl-forms.demo
                                                           "test/demo/client-validatio
                    :active-menu :client-validation))

(hunchentoot:define-easy-handler (client-validation-post :uri "/client-validation/post
  (flet ((client-validation-post ()
           (let ((form (forms:get-form 'client-validated-form)))
             (forms::handle-request form)
             (if (forms::validate-form form)
                 ;; The form is valid
                 (forms::with-form-field-values (name single sex age email) form
                   (who:with-html-output (forms.who::*html*)
                     (:ul
                      (:li (who:fmt "Name: ~A" name))
                      (:li (who:fmt "Single: ~A" single))
                      (:li (who:fmt "Sex: ~A" sex))
                      (:li (who:fmt "Age: ~A" age))
                      (:li (who:fmt "Email: ~A" email)))))
                 ;; The form is not valid
                 (client-validation form)))))
    (render-demo-page :demo #'client-validation-post
                      :source (asdf:system-relative-pathname :cl-forms.demo
                                                             "test/demo/client-validat
                      :active-menu :client-validation)))
```

### 3.2.4 Models

Forms can be attached to model objects. Model objects are CLOS instances from where form values are read and written to.

To work with models, forms are defined via defform-builder instead of defform. A form-builder is a function that takes the model objects and attaches it to the form. The form needs to define the accessors to access the model for each form field.

```
(in-package :forms.test)

(defclass person ()
  ((name :initarg :name
         :accessor person-name
         :initform nil)
   (single :initarg :single
           :accessor person-single
           :initform t)
   (sex :initarg :sex
        :accessor person-sex
        :initform :male)))

(forms:defform-builder model-form (person)
  (make-instance 'forms::form
                 :name 'model-form
                 :model person
                 :action "/models-post"
                 :fields (forms::make-form-fields
                          `((name :string :label "Name"
                                          :accessor person-name)
                            (single :boolean :label "Single"
                                             :accessor person-single)
                            (sex :choice :label "Sex"
                                         :choices (:male :female)
                                         :accessor person-sex
                                         :formatter format-sex)
                            (submit :submit :label "Update")))))

(defun format-sex (sex stream)
  (write-string
   (if (equalp sex :male) "Male" "Female")
   stream))

(defun models-demo ()
  (who:with-html-output (forms.who::*html*)
    (:h1 (who:str "Form models"))
    (:p "Forms can be attached to model objects. Model objects are CLOS instances from
    (:p "To work with models, forms are defined via defform-builder instead of defform
    (:p "This is an example of a form attached to a person object. Please have a look
    (render-model-form)))

(defun render-model-form (&optional form)
  (let ((form (or form
                  (let ((person (make-instance 'person
                                               :name "Foo"
                                               :single t
                                               :sex :male)))
                    (forms::get-form 'model-form person)))))
    (forms:with-form-renderer :who
      (forms:render-form form))))

(hunchentoot:define-easy-handler (model-form :uri "/models") ()
  (render-demo-page :demo #'models-demo
                    :source (asdf:system-relative-pathname :cl-forms.demo
                                                           "test/demo/models.lisp")
                    :active-menu :models))

(hunchentoot:define-easy-handler (model-form-post :uri "/models-post"
                                                  :default-request-type :post) ()
  (flet ((model-post ()
           (let ((person (make-instance 'person)))
             (let ((form (forms:get-form 'model-form person)))
               (forms::handle-request form)
               (forms::validate-form form)
               (who:with-html-output (forms.who::*html*)
                 (:ul
                  (:li (who:fmt "Name: ~A" (person-name person)))
                  (:li (who:fmt "Single: ~A" (person-single person)))
                  (:li (who:fmt "Sex: ~A" (person-sex person)))))))))
    (render-demo-page :demo #'model-post
                      :source (asdf:system-relative-pathname :cl-forms.demo
                                                             "test/demo/models.lisp")
                      :active-menu :models)))
```

### 3.2.5 Composition

It is possible to compose forms using the `subform` field type:

```
(in-package :forms.test)

(forms:defform member-form ()
  ((name :string :value "" :required-p nil)
   (ready :boolean :value t :required-p nil)
   (sex :choice :choices (list "Male" "Female") :value "Male")))

(forms:defform composition-form (:action "/composition-post")
  (
   ;; Subforms
   (main-member :subform :subform 'member-form)
   (secondary-member :subform :subform 'member-form)
      ;; Simple list
   (todo :list :type '(:string :required-p nil)
         :empty-item-predicate (lambda (field)
                                 (let ((val (forms:field-value field)))
                                   (or (null val)
                                       (string= val "")))))
  ;; Subform list
   (members :list :type '(:subform :subform member-form)
            :empty-item-predicate (lambda (field)
                                    (let* ((subform (forms:field-value field))
                                           (val (forms:get-field-value subform 'name))
                                      (or (null val)
                                          (string= val "")))))
   (save :submit :label "Save")))

(defun form-composition-demo (&optional form)
  (let ((form (or form (get-form 'composition-form))))
    (forms:with-form-renderer :who
      (who:with-html-output (forms.who::*html*)
        (:h1 (who:str "Forms composition"))
        (:p (who:str "These are examples of subforms and the list field type"))
        (forms::render-form-start form)
        (:h2 (who:str "Subforms"))
        (:p (who:str "This is an example of subform composition. main-member and secon
        (forms::render-field 'main-member form)
        (forms::render-field 'secondary-member form)
        (forms::render-field 'save form)
        (:h2 (who:str "List field"))
        (:p (who:str "This is an example of the list field. In this case, the list has
        (forms::render-field 'todo form)
        (forms::render-field 'save form)
        (:h2 (who:str "List of subforms"))
        (:p (who:str "This is the most complex example. This shows a list of subforms.
        (forms::render-field 'members form)
        (forms::render-field 'save form)
```

```
            (forms::render-form-end form)))))

    (hunchentoot:define-easy-handler (composition-demo :uri "/composition") ()
      (render-demo-page :demo #'form-composition-demo
                        :source (asdf:system-relative-pathname :cl-forms.demo
                                                               "test/demo/composition.lisp
                        :active-menu :composition))

    (hunchentoot:define-easy-handler (composition-demo-post :uri "/composition-post") ()
      (let ((form (forms:get-form 'composition-form)))
        (forms:handle-request form)
        (render-demo-page :demo (lambda ()
                                  (form-composition-demo form))
                          :source (asdf:system-relative-pathname :cl-forms.demo
                                                                 "test/demo/composition.li
                          :active-menu :composition)))
```

### 3.2.6 Form templates

Form templates is an alternative way of defining and rendering forms. Instead of defining a form with defform and then specifiying a template and render it, forms templates allow to do all that at the same time.

```
    (in-package :forms.test)

    (defun form-template-demo ()
      (macrolet ((row (&body body)
                   `(who:htm
                     (:div :class "row"
                           (who:htm
                            ,@body))))
                 (col (&body body)
                   `(who:htm
                     (:div :class "col-md-2"
                           (who:htm
                            ,@body)))))
        (forms:with-form-renderer :who
          (who:with-html-output (forms.who::*html*)
            (:div :class :container
                  (:div :class :row
                        (:div :class :heading
                              (:h1 (who:str "Form templates")))
                        (:p (who:str "Form templates is an alternative way of defining and
                        (:p (who:str "Form definition is embedded in rendering spec via wi
                        (:div :class :container
                              (forms:with-form-template () template-form (:action "/templa
                                (row
                                 (:h3 (who:str "General"))
```

```
                             (col (form-field firstname :string :value ""))
                             (col (form-field lastname :string :value "")))
                            (form-field active :boolean :value t)
                            (row
                             (:h3 (who:str "Address"))
                             (form-field address :string :value ""))
                            (row
                             (:h3 (who:str "Other"))
                             (form-field choices :choice
                                         :choices (list "Foo" "Bar")
                                         :value (list "Foo")
                                         :multiple t)
                             (form-field choices2 :choice
                                         :choices (list "Foo" "Bar")
                                         :value (list "Bar")
                                         :multiple t
                                         :expanded  t))
                            (row
                             (form-field submit :submit :label "Create"))))))))))

(hunchentoot:define-easy-handler (template-demo-handler :uri "/template") ()
  (render-demo-page :demo #'form-template-demo
                    :source (asdf:system-relative-pathname :cl-forms.demo
                                                           "test/demo/form-templates.l
                    :active-menu :template))

(hunchentoot:define-easy-handler (template-form-post
                                  :uri "/template-post"
                                  :default-request-type :post) ()
  (flet ((fields-post ()
           (let ((form (forms:get-form 'template-form)))
             (forms::handle-request form)
             (if (forms::validate-form form)
                 (forms::with-form-field-values (firstname lastname active address
                                                           choices choices2) form
                   (who:with-html-output (forms.who::*html*)
                     (:ul
                      (:li (who:fmt "Firstname: ~A" firstname))
                      (:li (who:fmt "Lastname: ~A" lastname))
                      (:li (who:fmt "Active: ~A" active))
                      (:li (who:fmt "Address: ~A" address))
                      (:li (who:fmt "Choices: ~A" choices))
                      (:li (who:fmt "Choices2: ~A" choices2)))))
                 "Form is not valid"))))
    (render-demo-page :demo #'fields-post
                      :source (asdf:system-relative-pathname :cl-forms.demo
                                                             "test/demo/form-templates
```

```
                         :active-menu :template)))
```

### 3.2.7 Renderers

```
(in-package :forms.test)

(hunchentoot:define-easy-handler (demo-renderers :uri "/renderers") ()
  (flet ((render ()
           (forms:with-form-renderer :who
     (who:with-html-output (forms.who:*html*)
       (:h2 "CL-WHO")
       (:p (who:str "Render via CL-WHO and whole form with RENDER-FORM."))
       (forms:render-form (forms:get-form 'fields-form))
       (:h2 "CL-WHO render by part")
       (:p (who:str "Render via CL-WHO and the individual rendering functions RENDER-F
       (forms:with-form (forms:get-form 'fields-form)
 (forms:render-form-start)
 (forms:render-field 'name)
 (forms:render-field-label 'ready)
 (forms:render-field-widget 'ready)
 (forms:render-field 'sex)
 (forms:render-field 'avatar)
 (forms:render-field 'disabled)
 (forms:render-field 'readonly)
 (forms:render-field 'readonly-checkbox)
 (forms:render-field 'disabled-checkbox)
 (forms:render-field 'submit)
 (forms:render-form-end))
       (:h2 "Djula")
       (:p (who:str "Render a form with a Djula template."))
       (who:str (djula:render-template* (asdf:system-relative-pathname :cl-forms.demo
       (:h2 "Djula by part")
       (:p (who:str "Render a form with a Djula template, by parts."))
       (who:str (djula:render-template* (asdf:system-relative-pathname :cl-forms.demo
               ))))
    (render-demo-page :demo #'render
                      :source (asdf:system-relative-pathname :cl-forms.demo
                                                             "test/demo/renderers.lisp
                      :active-menu :renderers)))
```

## 3.3 Form rendering

A form can be rendered via different renderers and themes. There are implemented renderers for CL-WHO and Djula. The only theme at the moment is a Bootstrap theme that runs under CL-WHO.

To be able to render a form a form renderer needs to be bound first. Renderers are bound using [WITH-FORM-RENDERER], page 17 macro.

Similarly, to use a theme other than the default one, it needs to be bound using [WITH-FORM-THEME], page 17.

### 3.3.1 Form rendering functions

Forms are renderer using [RENDER-FORM], page 18 to render the whole form all at once, or via [RENDER-FORM-START], page 19,[RENDER-FORM-END], page 19,[RENDER-FIELD], page 18,[RENDER-FIELD-LABEL], page 19,[RENDER-FIELD-WIDGET], page 19, to only render specific parts of a form and have more control.

### 3.3.2 CL-WHO renderer

The CL-WHO renderer uses CL-WHO library for rendering forms.

Needs `cl-forms.who` ASDF system loaded.

To render a form using CL-WHO bind the renderer via [WITH-FORM-RENDERER], page 17, bind `FORMS.WHO:*HTML*` variable, and then render the form:

```
(let ((form (forms::get-form 'fields-form)))
   (who:with-html-output (forms.who:*html*)
      (forms:with-form-renderer :who
         (forms:render-form form))))
```

### 3.3.3 Bootstrap theme

There's a Bootstrap theme implemented for CL-WHO renderer.

Needs `cl-forms.who.bootstrap` ASDF system loaded.

Select the theme via [WITH-FORM-THEME], page 17:

```
(let ((form (forms::get-form 'bs-fields-form)))
    (forms:with-form-theme 'forms.who::bootstrap-form-theme
      (forms:with-form-renderer :who
        (who:with-html-output (forms.who::*html*)
           (forms:render-form form)))))
```

### 3.3.4 Djula

CL-FORMS integrates with Djula template system.

Needs `cl-forms.djula` ASDF system loaded.

Djula tags:

- `{% form form %}`. Renders a whole form.
- `{% form-start form %}`. Renders the form start part.
- `{% form-end form %}`. Renders the form end part.
- `{% form-row form field-name %}`. Renders the row with label and widget for the form field.
- `{% form-field-label form field-name %}`. Renders the form field label.
- `{% form-field-widget form field-name %}`. Renders the form field widget.

Make sure to `{% set-package %}` at the beggining of your Djula template to the package where the form lives. Otherwise, Djula wont' be able to find form fields by name.

Examples:

A Djula template that renders a whole form:

```
{% form form %}
```

A Djula template that renders a form by parts:

```
{% set-package forms.test %}

{% form-start form %}
{% form-row form name %}
{% form-row form ready %}
<div>
  {% form-field-label form sex %}
  {% form-field-widget form sex %}
</div>
{% form-row form avatar %}
{% form-row form disabled %}
{% form-row form disabled-checkbox %}
{% form-row form readonly-checkbox %}
{% form-row form submit %}
{% form-end form %}
```

# 4 API

## 4.1 CL-FORMS package

`CL-FORMS` [PACKAGE]

### External definitions

#### Variables

`*BASE64-ENCODE*` [CL-FORMS]

Whether to encode form parameters in base64 or not.

#### Macros

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

Bind the value of FIELDS in FORM.

Example:

```
(with-form-field-values (name) form
   (print name))
```

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

Bind *FORM* to FORM and evaluate BODY in that context.

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

Registers a function with arguments ARGS and body BODY as a form builder.

BODY is expected to instantiate a FORM object using ARGS in some way.

FORM-NAME is the symbol under which the FORM is registered.

Use GET-FORM with FORM-NAME and expected arguments to obtain the registered form.

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

Bind *FORM-THEME* to FORM-THEME and evaluate BODY in that context.

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

Bind FIELDS to the form fields in FORM.

Example:

```
(with-form-fields (name) form
   (print (field-value name)))
```

Also see: WITH-FORM-FIELD-VALUES

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

Define a FORM named FORM-NAME and render it at the same time.

`CL-FORMS:WITH-FORM-RENDERER` (*renderer* **&body** *body*) [Macro]

Bind *FORM-RENDERER* to RENDERER and evaluate BODY in that context.

`CL-FORMS:DEFFORM` (*form-name args fields*) [Macro]

Define a form at top-level.

ARGS are the arguments passed to FORM class via MAKE-INSTANCE. FIELDS are the form field specs.

```
(forms:defform client-validated-form (:action "/client-validation-post"
                                       :client-validation t)
  ((name :string :value "" :constraints (list (clavier:is-a-string)
                                               (clavier:not-blank)
                                               (clavier:len :max 5))
         :validation-triggers '(:focusin))
   (single :boolean :value t)
   (sex :choice :choices (list "Male" "Female") :value "Male")
   (age :integer :constraints (list (clavier:is-an-integer)
                                    (clavier:greater-than -1)
                                    (clavier:less-than 200)))
   (email :email)
   (submit :submit :label "Create")))
```

## Generic functions

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

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

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

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

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

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

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

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

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

## Functions

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

Top level function to render the web form FORM. *FORM-RENDERER* and *FORM-THEME* need to be bound. See: WITH-FORM-RENDERER, WITH-FORM-THEME

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

Fill a FORM from a MODEL. Read MODEL using FORM accessors and set the FORM field values.

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

Render form FIELD, both label and widget.

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

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

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

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

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

Render the end of the web form FORM.

`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 after.

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

Set a MODEL's values from FORM field values.

`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]

Render the label of FIELD.

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

Get the form named NAME.

ARGS is the list of arguments to pass to a possible form builder function.

See: DEFFORM-BUILDER macro.

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

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

Render the validation errors associated with FIELD.

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

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

Validates a form. Usually called after HANDLE-REQUEST. 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:RENDER-FORM-START` (**&optional** *(form *form*)* **&rest** *args*) [Function]

Render only the beggining of the web form FORM. Use RENDER-FIELD, RENDER-FIELD-LABEL, etc manually, after.

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

Render FIELD widget.

`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]

Render a section for displaying form validation errors.

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

## Classes

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

An integer input field

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

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

A field that contains a form (subform)

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

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

An url input field

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

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

A date input field

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

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

A password input field

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

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

A string input field

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

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

A multi-purpose field used to allow the user to "choose" one or more options. It can be rendered as a select tag, radio buttons, or checkboxes. NOTE: the defaults of this field type are too complicated for just working with string choices. STRING-CHOICE-FIELD is more convenient for that.

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

Slots:

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

  An alist with the choices. Or a function with which to obtain the choices.
- `preferred-choices` — initarg: `:preferred-choices`; reader: `cl-forms::field-preferred-choices`; writer: `(setf cl-forms::field-preferred-choices)`

  If this option is specified, then a sub-set of all of the options will be moved to the top of the select menu.

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

  If set to true, radio buttons or checkboxes will be rendered (depending on the multiple value). If false, a select element will be rendered.
- `multiple` — initarg: `:multiple`; reader: `cl-forms::field-multiple`; writer: `(setf cl-forms::field-multiple)`

  If true, the user will be able to select multiple options (as opposed to choosing just one option). Depending on the value of the expanded option, this will render either a select tag or checkboxes if true and a select tag or radio buttons if false.
- `key-reader` — initarg: `:key-reader`; reader: `cl-forms::field-key-reader`; writer: `(setf cl-forms::field-key-reader)`

  Function to read the option key from the request
- `hash-function` — initarg: `:hash-function`; reader: `cl-forms::field-hash-function`; writer: `(setf cl-forms::field-hash-function)`

  The function to use for choices key
- `test` — initarg: `:test`; reader: `cl-forms::field-test`; writer: `(setf cl-forms::field-test)`

  Function to test equality between choices
- `use-key-as-value` — initarg: `:use-key-as-value`; reader: `cl-forms::use-key-as-value`; writer: `(setf cl-forms::use-key-as-value)`

  When T, use the key/s of the field as value of the field when it is read from request

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

A file input field

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

Slots:

- `multiple` — initarg: `:multiple-p`; reader: `cl-forms::multiple-p`; writer: `(setf cl-forms::multiple-p)`

  If this fields handles multiple file uploads
- `upload-handler` — initarg: `:upload-handler`; reader: `cl-forms::upload-handler`; writer: `(setf cl-forms::upload-handler)`

  Function that handles the file upload
- `accept` — initarg: `:accept`; reader: `cl-forms::file-accept`; writer: `(setf cl-forms::file-accept)`

  Files accepted. See https://www.w3schools.com/tags/att_input_accept.asp

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

A boolean input

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

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

A date input field

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

`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)`

  A list of CLAVIER validators.
- `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

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

A submit input button

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

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

A hidden form field

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

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

A string input field

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

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

A field that contains a list of elements (either other fields or subforms)

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

Slots:

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

  The list elements type.

- `empty-item-predicate` — initarg: `:empty-item-predicate`; reader: `cl-forms::empty-item-predicate`; writer: `(setf cl-forms::empty-item-predicate)`

  A predicate that tells when a list item is considered empty, and so it is removed from the list
- `add-button` — initarg: `:add-button`; reader: `cl-forms::add-button-p`; writer: `(setf cl-forms::add-button-p)`

  Whether have a list 'ADD' button or not
- `remove-button` — initarg: `:remove-button`; reader: `cl-forms::remove-button-p`; writer: `(setf cl-forms::remove-button-p)`

  Whether add an item removal button or not

`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:TEXT-FORM-FIELD` [Class]

A text field. Renders as a text area

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

# 5 Index

(Index is nonexistent)