cl-config

A configuration library for Common Lisp
Release 0.1

by Mariano Montone

Table of Contents

This manual is for cl-config version 0.1.

1 Introduction

cl-config is a configuration library for Common Lisp

You can get a copy and this manual at http://common-lisp.net/project/cl-config

1.1 Summary

cl-config is a configuration library for Common Lisp

1.2 Installation

To install cl-config, start a Common Lisp session and type the following:

```
CL-USER> (require :asdf-install)
CL-USER> (asdf-install:asdf-install 'cl-config)
```

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. Consider using an automatic documentation generator (see the bitacora package in the dependencies).
- We don't want functions to be shorter than the should nor longer than they should. There is no "every function should have at most ten lines of code" rule. We think that coding is like literature to a great extent. So you should strive for beauty and clarity. Again, that your code is object oriented doesn't imply that your methods will ideally have two lines of code and dispatch to somewhere else; that is not always good at all. It may be good from an object oriented point of view, but it is too low level. We want to think in terms of languages, that is higher level, instead of objects sending messages.
- Use destructuring-bind or let or a pattern-matching library instead of car, cdr, cadr, and the like to obtain more readable code.
- Use widely known Common Lisp coding guidelines: http://web.archive.org/web/20050305123711/www

2 Overview

CL-CONFIG is a configuration library for Common Lisp.

The idea is to define configuration-schemas and get a proper way of:

- Sharing and versioning your project's configuration schemas, but not your configurations. That way, you avoid overwriting configurations from different coders. Each coder has his own configurations that need to match the configuration schemas in the project. Whenever a project's configuration schema changes, each coder is reponsible of updating his configurations to match the new schemas.
- Being able to define configuration schemas declaratively.
- Provide configurations documentation and validation.
- Edit configurations from a GUI.
- Define your own option configurations types and provide validation for them.

3 Configuration schemas

Configuration schemas define the structure of a configuration.

The syntax to define configuration schemas is the following:

```
(define-configuration-schema configuration-schema-name
        ({parent-configuration-schema}*)
        (:title configuration-schema-title)
        [(:documentation configuration-schema-documentation)]
        {configuration-schema-section}*)
```

Where:

- *configuration-schema-name* is the name of the configuration-schema and the configuration-schema is globally identified by it. See *find-configuration-schema*
- *parent-configuration-schema* is the configuration schema we inherit from. Inheriting from a configuration schema means adding its sections to the child schema. Configuration schemas can inherit from several parents
- *configuration-schema-title* is a string describing very shortly the configuration schema. It is used to display configuration schemas from the editing GUI. It is a required argument.
- *configuration-schema-documentation* is the configuration schema documentation. This is not a required argument. It is also used from the editing GUI and is very useful for the configuration schema user.

Each configuration schema section follows this syntax:

```
(:section section-identifier section-title
   [(:documentation section-documentation)]
   {option-schema}*)
```

Where:

- *section-identifier* is a keyword that uniquely identifies the section
- *section-title* is a string describing very shortly the section. It is used to display sections from the editing GUI.

And option schemas are specified like this:

```
(option-identifier option-title option-type {option-parameter*})
```

where:

- *option-identifier* is a keyword that uniquely identifies the option
- *option-title* is a string describing very shortly the option. It is used to display sections from the editing GUI.
- *option-type* is the option type. There are different ways of specifying an option type, depending on the type.
- *option-parameters* may be:
  - *:documentation* followed by the documentation string. To document the option.
  - *default* followed by the default option value. If the configuration leaves the option unspecified, then it has the default value.

- *optional*, followed by true (t) of false (nil). Determines if the option value can be left unspecified. Default is false.
- *avanced*, followed by true (t) or false (nil). Determines if the option category is "advanced" (default is false)

Here is a simple example:

```
(define-configuration-schema database-configuration ()
   (:title "Database configuration")
   (:documentation "Database configuration")
   (:section :database-configuration "Database configuration"
      (:documentation "Section for configuring the database")
      (:connection-type "Connection type"
          (:one-of (:socket "Socket"
    :configuration 'db-socket-configuration)
   (:tcp "TCP"
 :configuration 'db-tcp-configuration)))
      (:username "Username" :text :documentation "The database engine username")
      (:password "Password" :text :documentation "The database engine password")
      (:database-name "Database name" :text)
      (:database-parameters "Database parameters" :text :default "" :advanced t)))
```

That is a typical configuration schema needed to connect to a database.

It has only one section *database-configuration* where the user is supposed to specify the connection type, the database name, the username, password, and extra parameters needed to connect to a database. In this case, most of the options are of type *:text*.

3.1 Built-in option types

3.1.1 Text

The text option type is specified with *:text*. It ensures that the the option value is of type string.

Example:

```
(:username "Username" :text :documentation "The database engine username")
(:password "Password" :text :documentation "The database engine password")
```

3.1.2 Integer

The integer option type is specified with *:integer*. It ensures that the the option value is of type integer.

Example:

```
(:port "Port" :integer :documentation "Web application port")
```

3.1.3 Boolean

The boolean option type is specified with *:boolean*. It ensures that the the option value is of type boolean (t or nil).

Example:

```
(:catch-errors-p "Catch errors?" :boolean :documentation "Whether to handle applicatio
```

3.1.4 Email

The email option type is specified with *:email*. It ensures that the the option value is a valid email string.

Example:

```
(:port "Email" :email :documentation "User email")
```

3.1.5 Url

The url option type is specified with *:url*. It ensures that the the option value is a valid url. The option value is converted to a url (cl-url) if it is a string, or left unmodified if already a url.

Example:

```
(:host "Host" :url :documentation "The web application host")
```

3.1.6 Pathname

The pathaname option type is specified with *:path*. It ensures that the the option value is a valid pathname and the file or directory exists. The option value is converted to a pathname if it is a string, or left unmodified if already a pathname.

Example:

```
(:stylesheet "Stylesheet" :pathname :documentation "The stylesheet file")
```

3.1.7 One of

The *one of* option type is specified with *:one-of* followed by the list of options, all between parenthesis. It is ensured that the option value is one of the options listed. Options are specified as a list with the option-identifier as a keyword, and the option title with a string.

Example:

```
(:connection-type "Connection type"
      (:one-of (:socket "Socket")
       (:tcp "TCP")))
```

3.1.8 List

The *list* option type is specified with *:list* followed by the list of options, all between parenthesis. It is ensured that the option value is a subset of the options listed. Options are specified as a list with the option-identifier as a keyword, and the option title with a string.

Example:

```
(:debugging-levels "Debugging levels" (:list (:info "Info")
     (:warning "Warning")
     (:profile "Profile")))
```

4 Configurations

How to define configurations

4.1 Working with configurations

The API for working with configurations

4.2 Configurations serialization

There are two output backends: an sexp-backend and a xml-backend

5 Installer

CL-Config provides machinery for building your application installers.

In CL-Config an application installer is an instance of the class *installer*. Installers are funcallable, and the idea is to think of installers as a function returning other function, that is, they follow a Continuation Passing Style design.

Some installers features:

- They are funcallable
- They are indenpendent of the UI. You can think of an installer as some kind of Controller in MVC. There's a web frontend available for installers; we plan to build some lighter-weight frontends, such as installing from the REPL or from an ncurses interface.
- They follow a continuation passing design.

There's a Domain Specific Language for defining installers and in particular, wizard installers. Wizard installers are those that asks for data in different sections or "pages" to complete the installation.

To define installers, there are three macros: *define-installer*. *define-wizard-installer* and *define-standard-installer*.

An example of an installer session:

```
CFG> *i*
#<WIZARD-INSTALLER MY-INSTALLER "My installer" 1006C2AB99>
```

We use funcall to start interacting with the installer:

```
CFG> (funcall *i*)
(:SECTION :WEBAPP-CONFIGURATION "Web configuration")
```

The web configuration section starts

```
CFG> (funcall *i*)
(:INPUT (HTTP-SERVER HOST PORT))
```

We are asked to input the :http-server, the :host and the :port

```
CFG> (funcall *i* :http-server :apache :host "localhost" :port 8080)
(:SECTION :DATABASE-CONFIGURATION "Database configuration")
```

Success. Now we enter the database configuration section.

```
CFG> (funcall *i*)
(:INPUT (NAME HOST USERNAME PASSWORD CONNECTION-TYPE))
```

We are asked to enter database parameters:

```
CFG> (funcall *i* :name "my-database" :host "localhost" :username "mariano" :password
(:ERRORS
 (#<VALIDATION-ERROR
    FOO value should be one of (SOCKET TCP) on #<ONE-OF-CONFIGURATION-OPTION
                                                 #<CONFIGURATION-SCHEMA-OPTION
                                                   CONNECTION-TYPE "Connection type"
                                                   1005BDF081> FOO
                                                 10085A57F1>
    10085A6701>))
```

We entered the parameters, but the :connection-type is not valid. We have to try again.

```
CFG> (funcall *i*)
(:INPUT (NAME HOST USERNAME PASSWORD CONNECTION-TYPE))
CFG> (funcall *i* :name "my-database" :host "localhost" :username "mariano" :password
(:SECTION :LOGGING-CONFIGURATION "Logging configuration")
```

We succeed this time, and we enter the logging configuration section

```
CFG> (funcall *i*)
(:INPUT (BACKEND DEBUGGING-LEVELS OUTPUT-LOCATION ACTIVE-LAYERS))
```

We are asked for logging section parameters

```
CFG> (funcall *i* :backend :log5 :debugging-levels '(:info) :output-location :file :ac
#<CONFIGURATION MY-CONFIG "My config" 1006D3DB91>
```

We finish the installation and we get the configured configuration as result.

6 Examples

```
(define-configuration-schema database-configuration ()
   (:title "Database configuration")
   (:documentation "Database configuration")
   (:section :database-configuration "Database configuration"
      (:documentation "Section for configuring the database")
      (:connection-type "Connection type"
          (:one-of (:socket "Socket"
    :configuration 'db-socket-configuration)
   (:tcp "TCP"
 :configuration 'db-tcp-configuration)))
      (:username "Username" :text :documentation "The database engine username")
      (:password "Password" :text :documentation "The database engine password")
      (:database-name "Database name" :text)
      (:database-parameters "Database parameters" :text :default "" :advanced t)))

(define-configuration-schema cl-config-application-configuration ()
  (:title "CL-CONFIG Application Configuration")
  (:documentation "CL-CONFIG Application Configuration")
  (:section :configuration-settings "Configuration settings"
    (:load-configs-from-file "Load configurations from file"
     :boolean :default t)
    (:load-configs-file "Configurations file" :pathname :optional t)
    (:select-config-from-file "Select configuration from file"
      :boolean :default t)
    (:select-config-file "Select configuration file" :pathname :optional t)))

(define-configuration-schema db-socket-configuration ()
    (:title "Socket configuration")
    (:section :db-socket-configuration "Socket configuration"
        (:path "Socket" :pathname
               :default "/tmp/socket.soc")))

(define-configuration-schema db-tcp-configuration ()
    (:title "TCP configuration")
    (:section "TCP configuration"
        (:url "URL" :url
              :default "localhost")))

(define-configuration-schema logging-configuration ()
    (:title "Logging configuration")
    (:documentation "Logging configuration")
    (:section :logging-configuration "Logging configuration"
        (:documentation "Logging configuration")
        (:backend "Backend"
```

```
            (:one-of (:log5 "Log5")))
(:debugging-levels "Debugging levels" (:list (:info "Info")
     (:warning "Warning")
     (:profile "Profile")))
(:output-location "Output location"
                    (:one-of (:standard-output "Standard output"
       :default *standard-output*)
     (:file "File" :default "/tmp/log.log"))
    :default '*standard-output)
        (:active-layers "Active layers"
(:list
 (:debugging "Debugging"
     :configuration 'debugging-layer)
 (:database "Database"
    :configuration database-layer)
 (:control-flow "Control flow")
 (:system "System")))))

(define-configuration-schema webapp-configuration (logging-configuration)
    (:title "Web application configuration")
    (:documentation "Web application configuration")
    (:section :webapp-configuration "Web application configuration"
        (:documentation "Web application configuration")
        (:http-server "HTTP server"
            (:one-of (:apache "Apache"
      :configuration 'apache-configuration)
                    (:hunchentoot "Hunchentoot"
  :configuration 'hunchentoot-configuration)))
        (:host "Host" :text :default "localhost")
(:port "Port" :integer :default 8080)
(:catch-errors "Catch errors" :boolean :default t)))

(define-configuration-schema standard-configuration
    (cl-config-application-configuration
     webapp-configuration
     database-configuration)
      (:title "Standard configuration")
      (:documentation "Standard configuration for a Gestalt application")
      (:page-title "Page title" :text :default "Gestalt application"))

(define-configuration standard-configuration ()
  (:title "Standard configuration")
  (:configuration-schema standard-configuration)
  (:section :database-configuration
    (:connection-type :socket
      :value2
      '(:db-socket-configuration
```

```
(:path "/tmp/my-socket.soc")))
    (:username "root")
    (:password "root")
    (:database-name "standard-database"))
  (:section :webapp-configuration
    (:http-server :hunchentoot))
  (:section :logging-configuration
    (:active-layers (:debugging))
    (:output-location :standard-output)
    (:debugging-levels (:info))
    (:backend :log5)))

(define-configuration debug-configuration (standard-configuration)
    (:configuration-schema standard-configuration)
    (:title "Debug configuration")
    (:section :database-configuration
        (:database-name "debug-database"))
    (:section :logging-configuration
       (:output-location :standard-output)
       (:active-layers (:debugging :database))
       (:debugging-levels (:info :warning :error)))
    (:section :webapp-configuration
      (:catch-errors nil))
    (:documentation "Debugging configuration scheme"))

(define-configuration test-configuration (standard-configuration)
    (:configuration-schema standard-configuration)
    (:title "Test configuration")
    (:section :database-configuration
        (:database-name "test-database"))
    (:section :logging-configuration
       (:output-location :file :value2 "/tmp/test.log")
       (:active-layers (:debugging :database) :inherit t)
       (:debugging-levels (:warning :error)))
    (:documentation "Testing configuration scheme"))
```

An installer example:

```
(defmacro configure-section (section configuration options)
  (let ((fname (gensym "CONFIGURE-SECTION-")))
  `(labels ((,fname ()
    (with-input ,(mapcar (lambda (option)
  (intern (symbol-name option)))
options)
      (collecting-validation-errors (errors found-p)
  (progn
    ,@(loop for option in options
 collect
```

```
    `(setf (get-option-value (list ,section ,option)
     ,configuration)
  ,(intern (symbol-name option)))))
(when found-p
  (install-errors errors)
  (,fname))))))
     (,fname))))

(define-wizard-installer my-installer
    (:title "My installer"
    :documentation "This is my installer")
  (let ((configuration (cfg:with-schema-validation (nil)
 (make-configuration my-config ()
     (:title "My config")
     (:configuration-schema standard-configuration)))))

    ;; Configure web section
    (start-section :webapp-configuration "Web configuration")
    (configure-section :webapp-configuration configuration
       (:http-server :host :port))

    ;; Configure the database section
    (start-section :database-configuration "Database configuration")
    (configure-section :database-configuration configuration
       (:name :host :username :password :connection-type))

    ;; Configure logging section
    (start-section :logging-configuration "Logging configuration")
    (configure-section :logging-configuration configuration
       (:backend :debugging-levels :output-location :active-layers))
    (validate-configuration configuration)
    configuration))
```

6.1 Use cases

6.1.1 Debugging

6.1.2 Logging

6.1.3 Testing

6.1.4 Deployment

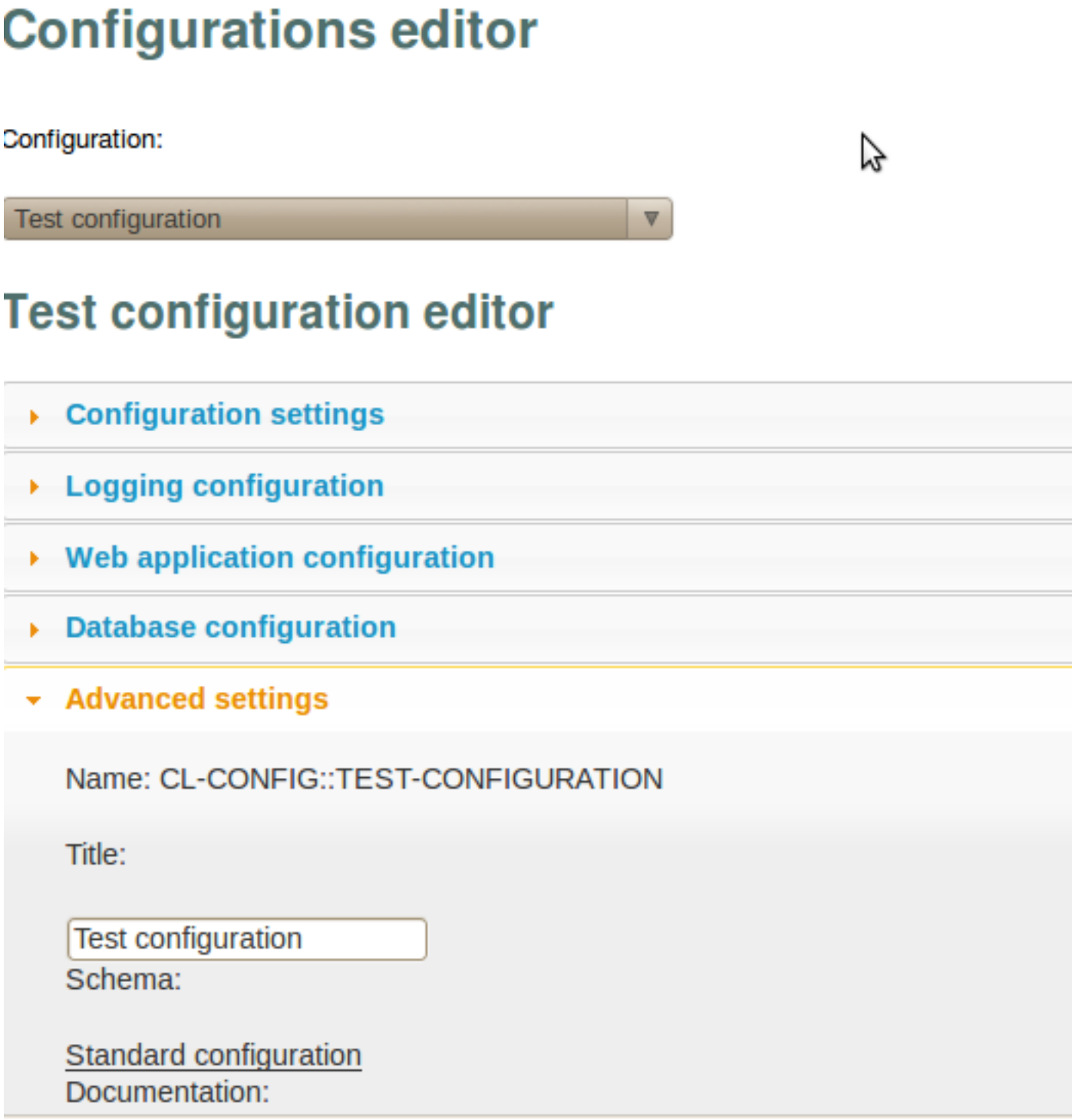
7 Configuration editing

Configurations can be edited from a web interface.

To start the web configuration editor, evaluate:

```
(require :cl-config-web)
(cfg.web:start-cl-config-web)
```

and then point your browser to http://localhost:4242



8 Custom option types

How to define custom option types

9 System reference

`cl-config:configuration-installer` [Class]

Class precedence list: `configuration-installer`, `installer`, `funcallable-standard-object`, `function`, `standard-object`, `t`

The class for configuration installers

`cl-config:configuration-schema` [Class]

Class precedence list: `configuration-schema`, `standard-object`, `t`

Slots:

- `name` — initargs: `:name`

  The configuration-schema name
- `parents` — initargs: `:parents`

  Configuration-Schema mixins
- `title` — initargs: `:title`

  Configuration-Schema title
- `direct-sections` — initargs: `:direct-sections`

  Configuration-Schema direct-sections
- `documentation` — initargs: `:documentation`

  Configuration-Schema documentation

A configuration-schema

`cl-config:configuration` [Class]

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

Slots:

- `name` — initargs: `:name`

  The configuration name
- `title` — initargs: `:title`

  Configuration title

The configuration class

`cl-config:installer` [Class]

Class precedence list: `installer`, `funcallable-standard-object`, `function`, `standard-object`, `t`

The main installer class. Installer instances are funcallable

`cl-config:standard-installer` [Class]

Class precedence list: `standard-installer`, `wizard-installer`, `configuration-installer`, `installer`, `funcallable-standard-object`, `function`, `standard-object`, `t`

standard-intallers are installers used for installing configurations in a wizard fashion

`cl-config:wizard-installer` [Class]
Class precedence list: `wizard-installer, installer, funcallable-standard-object, function, standard-object, t`

The class for wizard installers

`cl-config:cfg*` *path* **&optional** *configuration* [Function]
Function for getting a configuration value (the functional version of the cfg macro) path can be one of:

- A list with the form (<section> <option>). Example:
  (cfg* '(:database-configuration :username))
- A symbol with the form <section>.<option> Example:
  (cfg* :database-configuration.username)

The default configuration used is *configuration* (the current configuration)

`cl-config:find-configuration-schema` *name* [Function]
Get a configuration-schema by its name

`cl-config:find-configuration` *name* [Function]
Get a configuration by its name

`cl-config:find-installer` *name* [Function]
Gets an installer by name

`cl-config:register-installer` *installer* [Function]
Registers an installer. The installer can then be obtained via find-installer

`cl-config.web:start-cl-config-web` **&optional** *configuration* [Function]
Starts the web configuration editor

Default arguments are in standard-cl-config-web-configuration

Evaluate (cfg.web:start-cl-config-web) and point your browser to http://localhost:4242

`cl-config.web:stop-cl-config-web` [Function]
Stops the web configuration editor

`cl-config:cfg` *path* **&optional** *configuration* [Macro]
Macro for getting a configuration value. path can be one of:

- A list with the form (<section> <option>). Example:
  (cfg (:database-configuration :username))
- A symbol with the form <section>.<option> Example:
  (cfg :database-configuration.username)

  The default configuration used is *configuration* (the current configuration)

`cl-config:define-configurable-function` *name args* **&body** *body* [Macro]
Defines a configurable function. See macroexpansion to understand what it does

Example:

```
(cfg:define-configurable-function connect (&configuration (conf 'postgres-data
    (cfg:with-configuration-values (database-name username password host)
configuration
    (connect database-name username password host)))
```

And then:

```
(connect :database-name "My database"
         :host "localhost"
         :username "foo"
         :password "bar")
```

`cl-config:define-configuration-schema-option-type` *type args* **&body** *body* [Macro]

Define a custom configuration-schema option type. Example:

```
(define-configuration-schema-option-type :email (&rest args)
    (apply #'make-instance 'email-configuration-schema-option-type
args))
```

`cl-config:define-configuration-schema` *name parents* **&rest** *args* [Macro]

Syntax for defining a configuration-schema.

Parameters:

- name - The name of the schema
- parents - A list of schema parents

Comments:

- A configuration schema can inherit from several parents.
- A title parameter is required to define the schema (see example below)

Example:

```
(cfg::define-configuration-schema postgres-database-configuration ()
  (:title "Postgres database configuration")
  (:documentation "Postgres database configuration")
  (:section :database-configuration "Database configuration"
     (:documentation "Section for configuring the database")
     (:connection-type "Connection type"
         (:one-of (:socket "Socket"
      :configuration 'db-socket-configuration)
     (:tcp "TCP"
    :configuration 'db-tcp-configuration)))
         (:username "Username" :text :documentation "The database engine username")
         (:password "Password" :text :documentation "The database engine password")
         (:database-name "Database name" :text)
         (:host "Host" :text :documentation "The database host")
         (:database-parameters "Database parameters" :text :default "" :advanced t))
```

`cl-config:define-configuration-validator` *configuration-schema configuration* **&body** *body* [Macro]

Defines a validator on a configuration.

Example:

```
(cfg::define-configuration-validator postgres-database-configuration (configur
  (cfg:with-configuration-section :database-configuration
    (cfg:with-configuration-values
         (database-name username password host) configuration
        (handler-bind
  (postmodern:connect database-name username password host)
     (postmodern:database-error (error)
(cfg::validation-error
 (cl-postgres::message error)))))))
```

`cl-config:define-configuration` *name parents* **&rest** *args* [Macro]

Create and register a configuration Example:

```
(define-configuration debug-configuration (standard-configuration)
 (:configuration-schema standard-configuration)
 (:title "Debug configuration")
 (:section :database-configuration
     (:database-name "debug-database"))
 (:section :logging-configuration
    (:output-location :standard-output)
    (:active-layers (:debugging :database))
    (:debugging-levels (:info :warning :error)))
 (:section :webapp-configuration
   (:catch-errors nil))
 (:documentation "Debugging configuration scheme"))
```

`cl-config:define-installer` *name* **&key &body** *body* [Macro]

Define a vanilla installer

`cl-config:define-option-processor` *type value* **&body** *body* [Macro]

Define a processor for a custom type

`cl-config:define-option-validator` *type value condition error-msg* **&rest** *args* [Macro]

Define a validator for a custom type

Example:

```
(define-option-validator email-configuration-schema-option-type
   (value option)
     (valid-mail-address-p value)
           "~A is not a valid email address in ~A" value option)
```

`cl-config:define-standard-installer` *name* **&key &body** *body* [Macro]

Defines a standard-installer

`cl-config:define-wizard-installer` *name* **&key &body** *body* [Macro]

Define a wizard installer

`cl-config:idefun` *name args* **&body** *body* [Macro]

Defines a installer function

`cl-config:make-configuration` *name parents* **&rest** *args* [Macro]
Create a configuration without registering it globally

`cl-config:with-configuration-section` *section-name* **&body** *body* [Macro]
Executes body in the context of the given configuration section

Example:

```
(with-configuration test-configuration
    (with-configuration-section :database-configuration
(cfg :username)))
```

`cl-config:with-configuration-values` *values configuration* **&body** *body* [Macro]
Macro for binding a configuration option values

Example:

```
(with-configuration test-configuration
    (with-configuration-section :database-configuration
(with-configuration-values (username) *configuration*
  username)))
```

`cl-config:with-configuration` *configuration-name* **&body** *body* [Macro]
Executes body in the context of the given configuration Example:

```
(with-configuration test-configuration
    (cfg (:database-configuration :username)))
```

`cl-config:with-current-configuration-values` *values* **&body** *body* [Macro]
The same as with-configuration-values but using the current configuration *configuration*

Example:

```
(with-configuration test-configuration
    (with-configuration-section :database-configuration
(with-current-configuration-values (username)
  username)))
```

`cl-config:with-input` *bindings* **&body** *body* [Macro]
Wizard installer operation. Asks for input.

`cl-config:with-schema-validation` **&optional &body** *body* [Macro]
Executes body validating or or not the configurations created in body context (depending the value of value). The default when using this macro is to not validate. This macro is more commonly used for internal implementation options.

Example:

```
(with-schema-validation (nil)
    (setf (cfg :database-configuration.username) 2323))
```

`cl-config:*configuration-schemas*` [Variable]
The defined configuration-schemas. Access the confiuration-schemas through the find-configuration-schema function

`cl-config:*configuration*` [Variable]
The current configuration. Use with-configuration macro to set this

`cl-config:*configurations*` [Variable]
The defined configurations. Use find-configuration to access configurations by name

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