

# cl-config

---

A configuration library for Common Lisp  
Release 0.1

by Mariano Montone

# Table of Contents

<b>1</b>	<b>Introduction.....</b>	<b>1</b>
1.1	Summary .....	1
1.2	Installation .....	1
1.3	Feedback.....	1
1.4	Conventions.....	1
<b>2</b>	<b>Output backends.....</b>	<b>2</b>
<b>3</b>	<b>Examples .....</b>	<b>3</b>
<b>4</b>	<b>Configuration editing.....</b>	<b>6</b>
<b>5</b>	<b>System reference.....</b>	<b>7</b>
<b>6</b>	<b>References .....</b>	<b>11</b>
<b>7</b>	<b>Index .....</b>	<b>12</b>
7.1	Concept Index .....	12
7.2	Class Index .....	12
7.3	Function / Macro Index.....	12
7.4	Variable Index .....	12

---

This manual is for cl-config version 0.1.

Copyright © 2011 Mariano Montone

Permission is granted to copy, distribute and/or modify this document under the terms of the GNU Free Documentation License, Version 1.3 or any later version published by the Free Software Foundation; with no Invariant Sections, with the Front-Cover texts being “A GNU Manual,” and with the Back-Cover Texts as in (a) below. A copy of the license is included in the section entitled “GNU Free Documentation License.”

(a) The FSF’s Back-Cover Text is: “You have the freedom to copy and modify this GNU manual. Buying copies from the FSF supports it in developing GNU and promoting software freedom.”

This document is part of a collection distributed under the GNU Free Documentation License. If you want to distribute this document separately from the collection, you can do so by adding a copy of the license to the document, as described in section 6 of the license.

# 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](mailto:marianomontone@gmail.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 architecture 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 they 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 Output backends

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

### 3 Examples

Example:

```
(define-configuration '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 :text :default "" :advanced)))

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

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

(define-configuration 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")))

(define-configuration logging-configuration ()
  (:title "Logging configuration")
  (:documentation "Logging configuration")
  (let-configuration*
    ((output-logging ()
      (output-location "Output location"
        (one-of (:standard-output "Standard output" :value '*standard-outp
```

```

        (:file "File" :text :default "/tmp/log.log"))))■
(debugging-layer (output-logging)
  (debugging-levels "Debuggin levels" (list (:info "Info")
                                             (:warning "Warning")■
                                             (:profile "Profile"))))■

(database-layer (output-logging)
  ...)))
(:section :logging-configuration "Logging configuration"
  (:documentation "Logging configuration")
  (:backend "Backend"
    (:one-of (:log5 "Log5"))))
  (:active-layers "Active layers" (list (:debugging "Debugging" (:configuration (
                                             (:database "Database" (:configuration da
                                             (:control-flow "Control flow")■
                                             (:system "System")))))■

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

```

The typical attributes types are, `:text`, where the user fill text in; `:one-of options*`, where the user chooses one of the options in `options*`; `:list list*`, where the user selects one or more of the items of the list `*list`; `:bool`, a boolean, `:maybe option`, where the user can disable or enable option, etc.

Configurations can inherit from several configurations (that act like mixins). The same as with classes or models or templates. So, for example, `web-app-configuration` inherits from `logging-configuration`. That means the `web-app-configuration` will have the sections defined in `logging-configuration` too.

Documentation is used as a section or configuration help from the UI. From the UI, each section is shown collapsable and there's and option for showing/hiding advanced fields, and the help button.

The user can define several configuration schemes for an application and switch between the configurations. For example, there will probably be a “development configuration”, a “deployment configuration”, a “testing configuration”, and so on.

There's no need for a GUI, although it is desirable. We can define configurations with files, for example:

```

(define-configuration-scheme standard-configuration-scheme ()
  (:configuration standard-configuration)
  (:database-configuration
    (:connection-type :socket
      (:db-socket-configuration
        (:path "/tmp/my-socket.soc"))))
  (:username "root")
  (:password "root"))

```



```

        (:database-name "standard-database"))
    (:webapp-configuration
      (:host "localhost")
      (:http-server :hunchentoot)))

(define-configuration-scheme debug-configuration-scheme (standard-configuration-scheme)
  (:configuration standard-configuration)
  (:database-configuration
    (:database-name "debug-database"))
  (:logging-configuration
    (:output-location :file "/tmp/debug.log")
    (:active-layers :debugging :database
      (:debugging-levels :info :warning :error)))
  (:documentation "Debugging configuration scheme"))

(define-configuration-scheme test-configuration-scheme (standard-configuration-scheme)
  (:configuration standard-configuration)
  (:database-configuration
    (:database-name "test-database"))
  (:logging-configuration
    (:output-location :file "/tmp/test.log")
    (:active-layers :debugging :database
      (:debugging-levels :warning :error)))
  (:documentation "Testing configuration scheme"))

And then we attach the desired configuration to the application:
(defapplication my-application (standard-application)
  ...
  (:configuration 'debug-configuration-scheme))

```

## 4 Configuration editing

The web configuration editing tool

## 5 System reference

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

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* [Macro]  
*configuration &body body*

Defines a validator on a configuration.

Example:

```
(cfg::define-configuration-validator postgres-database-configuration (configuration
  (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-option-processor** *type value &body body* [Macro]  
 Define a processor for a custom type

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

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:make-configuration** *name parents &rest args* [Macro]  
 Create a configuration without registering 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* [Macro]  
*body*

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-schema-validation** **&optional** **&body** *body* [Macro]  
 Executes body validating 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 configuration-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

## 6 References

[Common Lisp Directory] [Common Lisp Wiki]

[Common Lisp Directory]: <http://common-lisp.net> [Common Lisp Wiki]:  
<http://www.cliki.net>

## 7 Index

### 7.1 Concept Index

#### B

backends ..... 2

#### C

conventions ..... 1

#### E

examples ..... 3

#### F

feedback ..... 1

#### I

installation ..... 1

introduction ..... 1

#### R

reference ..... 11

#### S

summary ..... 1

### 7.2 Class Index

(Index is nonexistent)

### 7.3 Function / Macro Index

cl-config:cfg ..... 7

cl-config:cfg\* ..... 7

cl-config:define-configuration ..... 8

cl-config:define-configuration-schema ..... 7

cl-config:define-configuration-schema-  
option-type ..... 7

cl-config:define-configuration-validator .. 8

cl-config:define-option-processor ..... 8

cl-config:define-option-validator ..... 9

cl-config:find-configuration ..... 7

cl-config:find-configuration-schema ..... 7

cl-config:make-configuration ..... 9

cl-config:with-configuration ..... 9

cl-config:with-configuration-section ..... 9

cl-config:with-configuration-values ..... 9

cl-config:with-current-configuration-values  
..... 9

cl-config:with-schema-validation ..... 9

### 7.4 Variable Index

cl-config:\*configuration\* ..... 10

cl-config:\*configuration-schemas\* ..... 10

cl-config:\*configurations\* ..... 10