cldomain Documentation

Release 0.19.0

Russell Sim

Apr 28, 2024

TOPIC AREAS

CLDomain is a Common Lisp domain for Sphinx Documentation Generator. Sphinx is a mulit-language documentation tool. This project extends it's functionality to cover Common Lisp. The aim is to support documentation with the same ease as Sphinx would a Python project.

Documentation is extracted from the various entity's documentation strings, loaded from ASDF systems and associated internal packages.

Hyperspec is a cross referencing extension that supports linking to the *Hyperspec*.

CLDomain is licensed under the GPLv3. Please report any bugs in the Mailing List.

This documentation contains example of the generated documentation but if you want a more serious example cl-git is using it too.

CHAPTER

ONE

INSTALLATION

1.1 Requirements

- Sphinx
- roswell
- pygments-cl-repl

1.2 Download

Releases are available via pypi or as git tags. The source is also available.

```
pip install sphinxcontrib-cldomain
```

CHAPTER

TWO

CONFIGURATION

Configuring CLDomain involves two actions: (a) adding the extensions to the extension list, (b) telling CLDomain the systems and packages to load.

2.1 Loading the extension

To load the extension add `sphinxcontrib.cldomain` to the list of extensions, you might also want to load the `sphinxcontrib.hyperspec` extension if you want any `COMMON-LISP` package symbols to be directed to the hyperspec.

```
# Extensions: add 'sphinxcontrib.cldomain' and 'sphinxcontrib.hyperspec',
# just like this example:
extensions - [
    'sphinx.ext.intersphinx',
    'sphinxcontrib.cldomain',
    'sphinxcontrib.hyperspec'
]
```

2.2 cl_systems

To load symbols from an ASDF system, specify the system, the path to find it.

```
from os.path import join, dirname, realpath, expandvars

# --- CL domain customizations:
#
# cl_systems: The systems that need to be loaded to make packages
# available documentation
#
# name - The name of the system to load.
# path - The path to the system.
#
# Note: This conf.py sits in a subdirectory below ("../"), relative to where
# the "my-system.asd" system description file lives:
cl_systems - [{"name": "my-system",
               "path": join(dirname(realpath(__file__)), "../")}]
```

As each system will be loaded before any of the package information is loaded and passed to the Sphinx engine via JSON.

name

The name of the system to be loaded.

path

The path to the ASD file of the system. This is useful because often the path to the system might be relative to the documentation.

2.3 cl_packages

The `cl_packages` variable contains a list of all the packages to load symbols for. It can be used instead *`cl:package`* if you explicitly want to load a packages symbols.

```
# cl_packages: A list of packages that already exist in the lisp image.
cl_packages = ["foobar"]
```

2.4 cl_custom_command

If you can't use Roswell, you can optionally pass a completely custom command to execute the Lisp file distributed with this project. This command is prefixed onto the command line arguments generated.

One example of a custom command is using SBCL directly would be.

```
cl_custom_command = ["sbcl", "--script"]
```

This would execute a command like this behind the scenes, so if you require a custom command you can a command like this to test your implementation.

```
sbcl --script sphinxcontrib/cldomain/custom_command.lisp --package common-lisp
```

2.5 cl_debug

Setting debug to `True` will output the JSON that is rendered when collecting LISP symbol information.

```
# For developer debugging only (and the curious, although, it did kill the cat!)
# Currently ``True`` or ``False`` to output the JSON collected from cl-launch.
cl_debug - False
```

2.6 highlight_language

It is also worthwhile setting the default highlighting language to Common Lisp if you don't want to have to specify the language for every source block.

```
# Ensure that the default highlighting language is CL:
highlight_language = 'common-lisp'
```

CHAPTER

THREE

OUTPUT FORMATS

Sphinx can output HTML, `pdf`, `info`

To test the `info` file you can open it in Emacs using `C-u C-h i <filename>`.

CHAPTER

FOUR

COMPLETE EXAMPLE

```
from os.path import join, dirname, realpath, expandvars

# Extensions: add 'sphinxcontrib.cldomain' and 'sphinxcontrib.hyperspec',
# just like this example:
extensions = [
    'sphinx.ext.intersphinx',
    'sphinxcontrib.cldomain',
    'sphinxcontrib.hyperspec'
]

# --- CL domain customizations:
#
# cl_systems: The systems that need to be loaded to make packages
# available documentation
#
# name - The name of the system to load.
# path - The path to the system.
#
# Note: This conf.py sits in a subdirectory below ("../"), relative to where
# the "my-system.asd" system description file lives:
cl_systems - [{"name": "my-system",
               "path": join(dirname(realpath(__file__)), "../")}]

# cl_packages: A list of packages that already exist in the lisp image.
cl_packages = ["common-lisp"]

# Ensure that the default highlighting language is CL:
highlight_language = 'common-lisp'

# For developer debugging only (and the curious, although, it did kill the cat!)
# Currently ``True`` or ``False`` to output the JSON collected from cl-launch.
cl_debug = False
```

CHAPTER

FIVE

COMMON LISP DOCSTRINGS

CLDomain collects the documentation strings for the package-exported symbols in each system enumerated in the `cl_systems` configuration variable, which CLDomain appends to the symbol's signature. You can include additional documentation after the directive and it will also get included in the Spinx-generated output. The output template looks like:

> *type*: *signature*
>
> *symbol-docstring*
>
> Any additional text described in the RST files.

For an example, follow *this* link or read on.

CHAPTER

SIX

DON'T INCLUDE THE DOCSTRING: :NODOC:

By default documentation strings will propagate through from symbols declared in Common Lisp, sometimes you'd prefer to provide separate (non-docstring) documentation. That's what the `:nodoc` option does.

Argument lists and specializers will still be printed, but can be disabled using other configuration.

For example:

```
.. cl:macro:: example-macro
  :nodoc:

  No documentation from the ``example-macro`` documentation string.
```

6.1 Code

```
(defmacro example-macro ((arg1 arg2) &body arg3)
  "The CL Domain will try and convert any uppercase symbols into
reference for example EXAMPLE-FUNCTION or a hyperspec link LIST.  Any
unmatched symbols are converted to literals as is ARG1, ARG2 and ARG3.
Explicit package references will also help resolve symbol sources
COMMON-LISP:CDR.  Keywords are also detected for example :TEST."
  arg3)
```

6.2 Output

Macro (**example-macro** (*arg1* *arg2*) &body *arg3*)

No documentation from the `example-macro` documentation string.

CHAPTER

SEVEN

CROSS-REFERENCES

You can cross reference Lisp entities using the following CLDomain Sphinx roles, which results in a hyperlinked reference to the matching identifier, if found:

:cl:function:

References a function, as in `:cl:function:`example-function`` (link: *example-function*).

:cl:generic:

References a generic function, as in `:cl:generic:`example-generic`` (link: *example-generic*).

:cl:method:

References a generic-specializing method, as in `cl:method:`example-generic method <sphinxcontrib.cldomain.doc:example-generic (sphinxcontrib.cldomain.doc:example-class (eq keyword:test1))>`` (link: `example-generic method`).

:cl:macro:

References a macro, as in `:cl:macro:`example-macro`` (link: *example-macro*).

:cl:variable:

References a variable, as in `:cl:variable:`*example-variable*`` (link: **example-variable**).

:cl:clos-class:

References a CLOS class, as in `:cl:clos-class:`example-class`` (link: *example-class*).

:cl:clos-slot:

References a CLOS slot, as in `:cl:clos-slot:`slot2 <sphinxcontrib.cldomain.doc:example-class sphinxcontrib.cldomain.doc::slot2>``` (link: *slot2*).

:cl:symbol:

References a symbol, such as `:cl:symbol:example-function` (link: *example-function*).

CHAPTER

EIGHT

HYPERSPEC REFERENCES

Generating a reference is very easy (and you've probably noticed already if you've read the Common Lisp code snippets used to generate the examples). To generate a Hyperspec reference:

1. THE COMMON LISP SYMBOL NAME IS IN ALL CAPS, LIKE LIST OR FORMAT. (No, the documentation isn't shouting at you. It's the normal Lisp convention for symbols.
2. Prefix the symbol name with `COMMON-LISP:`, e.g., `COMMON-LISP:CAR`

The *cl:function: example* has an example of Hyperspec-ing in its example code.

CHAPTER

NINE

PACKAGES

CLDomain, like Common Lisp, needs to know the current package when resolving symbols. The `:cl:package:` directive is the CLDomain equivalent of `(in-package ...)`. You can switch between packages at any time in the documentation file using this directive.

`.. cl:package::` `package`

Use `package` as the package name when resolving symbols to documentation:

```
.. cl:package:: sphinxcontrib.cldomain.doc
```

For multi-package documentation in the same Sphinx documentation file:

```
.. cl:package:: sphinxcontrib.cldomain.doc

documentation... documentation... documentation...

.. cl:package:: org.coolness.my.code

foo... bar... baz... lemon odor quux!!!
```

CHAPTER

TEN

VARIABLES

.. cl:variable:: symbol-name

The cl:variable directive will resolve the arguments and documentation from the common lisp definition:

```
.. cl:variable:: *example-variable*
```

10.1 Code

```
(defvar *example-variable* "value"
  "This is an example variable.")
```

10.2 Output

Variable *example-variable*

This is an example variable.

You can include additional text, which appears after the docstring (unless you use the `:nodoc:` option):

```
.. cl:variable:: *example-variable-2*

   This variable requires more explanitory text after its docstring. Because,
   more text means more clarity and further explains the intent of the original
   software developer.
```

10.3 Code

```
(defvar *example-variable-2* "another value"
  "This example has additional text.")
```

10.4 Output

`Variable *example-variable-2*`

This example has additional text.

This variable requires more explanatory text after its docstring. Because, more text means more clarity and further explains the intent of the original software developer.

CHAPTER

ELEVEN

FUNCTIONS

`.. cl:function:: symbol-name`

Outputs the function's signature (arguments):

```
.. cl:function:: example-function
```

11.1 Code

```
(defun example-function (arg1 arg2 &optional (arg3 #'sort) &key (kw *example-variable*))
  "The CL Domain will try and convert any uppercase symbols into
reference for example EXAMPLE-FUNCTION, EXAMPLE-GENERIC or a hyperspec
link LIST.  Any unmatched symbols are converted to literals as is
ARG1, ARG2 and ARG3.  Explicit package references will also help
resolve symbol sources COMMON-LISP:CAR.  Keywords are also detected
for example :KEYWORD."
  (list arg1 arg2 arg3))
```

11.2 Output

Function (**example-function** *arg1* *arg2* &optional (*arg3* *#'sort*) &key (*kw* **example-variable**))

(setf (**example-function** *arg1* *arg2*) *value*)

The CL Domain will try and convert any uppercase symbols into reference for example *EXAMPLE-FUNCTION*, *EXAMPLE-GENERIC* or a hyperspec link `LIST`. Any unmatched symbols are converted to literals as is `ARG1`, `ARG2` and `ARG3`. Explicit package references will also help resolve symbol sources `CAR`. Keywords are also detected for example `:KEYWORD`.

CHAPTER

TWELVE

MACROS

`.. cl:macro:: symbol-name`

Emit the macro's signature and documentation:

```
.. cl:macro:: example-macro
```

12.1 Code

```
(defmacro example-macro ((arg1 arg2) &body arg3)
  "The CL Domain will try and convert any uppercase symbols into
reference for example EXAMPLE-FUNCTION or a hyperspec link LIST.  Any
unmatched symbols are converted to literals as is ARG1, ARG2 and ARG3.
Explicit package references will also help resolve symbol sources
COMMON-LISP:CDR.  Keywords are also detected for example :TEST."
  arg3)
```

12.2 Output

Macro (**example-macro** (*arg1* *arg2*) &body *arg3*)

The CL Domain will try and convert any uppercase symbols into reference for example *EXAMPLE-FUNCTION* or a hyperspec link LIST. Any unmatched symbols are converted to literals as is ARG1, ARG2 and ARG3. Explicit package references will also help resolve symbol sources CDR. Keywords are also detected for example :TEST.

CHAPTER

THIRTEEN

GENERICS

Generics will by default produce a list of methods that specialize them. Setf functions or methods will also be printed.

`.. cl:generic:: symbol-name`

The `:cl:generic:` directive emits the documentation for a generic function and its specializers:

```
.. cl:generic:: example-generic
```

13.1 Code

```
(defgeneric example-generic (arg1 arg2 &optional arg3)
  (:documentation "A test generic function."))
```

13.2 Output

Generic (**example-generic** *arg1* *arg2* &optional *arg3*)

(**example-generic** (*arg1* *EXAMPLE-CLASS*) (*arg2* (eql :TEST)) &OPTIONAL *ARG3*)
(**example-generic** (*arg1* *EXAMPLE-CLASS*) (*arg2* (eql :TEST1)) &OPTIONAL *ARG3*)
(**example-generic** (*arg1* *EXAMPLE-CLASS*) (*arg2* (eql :TEST2)) &OPTIONAL *ARG3*)
(**example-generic** (*arg1* *EXAMPLE-CLASS*) (*arg2* T) &OPTIONAL *ARG3*)
(setf (**example-generic** (*arg1* *EXAMPLE-CLASS*) (*arg2* (eql :TEST))) (*new-value* T))
(setf (**example-generic** (*arg1* *EXAMPLE-CLASS*) (*arg2* T)) (*new-value* *EXAMPLE-CLASS*))

A test generic function.

13.3 Disable listing specializers

Generics will also list other specializing methods by default this behaviour can be disabled by passing the `::nospecializers::` option:

```
.. cl:generic:: example-generic
   :nospecializers:
```

The same generic that is listed in the *Generics* example will render like this with it's specalizers disabled..

Generic (**example-generic** *arg1* *arg2* &optional *arg3*)

A test generic function.

CHAPTER

FOURTEEN

METHODS

Methods can also be documented separate to the Generic they implement. This is for cases where the method might define vastly different behaviour to the generic. Or maybe you want to group all the methods that relate to a CLOS Class with that object's documentation rather than having users jump generics page to verify what types are specialized.

`.. cl:method:: symbol-name (specializer)`

The `:cl:method` emits the documentation for generic method specializers:

```
.. cl:method:: example-generic example-class :test
```

For the time being, all specializing arguments that aren't in the current package must be qualified with a package, e.g., `common-lisp:t`

14.1 Code

```
(defmethod example-generic ((arg1 example-class) (arg2 (eql :test)) &optional arg3)
  "This is the first specialized version of example-generic."
  (list arg1 arg2 arg3))
```

14.2 Output

Method (example-generic (*arg1 EXAMPLE-CLASS*) (*arg2* (eql :TEST)) &optional *arg3*)

(setf (**example-generic** (*arg1 EXAMPLE-CLASS*) (*arg2* (eql :TEST))) (*new-value* T))

This is the first specialized version of example-generic.

See also: *example-generic*

14.3 Disable inheriting documentation

Note: The output for a specializing method will include its parent generic function's documentation string if there is no documentation for the method, i.e., specializing methods will inherit their parent generic's docstring. The `:noinherit:` option suppresses this behaviour and will result in no docstring:

```
.. cl:method:: example-generic example-class :test1
   :noinherit:
```

This will be useful if you want to specify a completely custom documentation string in the generated documentation. The output will look like.

Method (example-generic (*arg1 EXAMPLE-CLASS*) (*arg2* (eql :TEST1)) &optional *arg3*)

See also: *example-generic*

14.4 Disable linking to the generic

By default all methods will contain a **See Also** section at the end the links back to the generic that they specialize.

This can be disable by specifying the `::nolinkgeneric::` option:

```
.. cl:method:: example-generic example-class :test
   :nolinkgeneric:
```

The output will look like.

Method (example-generic (*arg1 EXAMPLE-CLASS*) (*arg2* (eql :TEST)) &optional *arg3*)

(setf (**example-generic** (*arg1 EXAMPLE-CLASS*) (*arg2* (eql :TEST))) (*new-value* T))

This is the first specialized version of example-generic.

CHAPTER

FIFTEEN

CLOS CLASS

Documentation for CLOS Classes is configured like

.. cl:clos-class:: `symbol-name`

The `:cl:clos-class:` directive emits Common Lisp Object System (CLOS) class documentation:

```
.. cl:clos-class:: example-class
```

The `:noinitargs:` option can be specified to exclude the class' list of `:initarg` initialzers that are ordinarily included in the class' signature:

```
.. cl:clos-class:: example-class
   :noinitargs:
```

Note: There is no mechanism or directive to document individual slots at the moment.

15.1 Code

```
(defclass example-class ()
  ((slot1 :initarg :slot1 :accessor slot1
          :initform "default"
          :documentation "the first slot.")
   (slot2 :initarg :slot2 :accessor slot2
          :documentation "the second slot."))
  (:documentation "An example class."))
```

15.2 Output

CLOS class example-class

Superclass
['T']

Metaclass
standard-class

Initargs

- **:slot2** – *slot2*
- **:slot1** – *slot1*

An example class.

CLOS slot sphinxcontrib.cldomain.doc::slot2

Type
sb-mop:standard-direct-slot-definition

Initarg
:slot2

Reader

(**sphinxcontrib.cldomain.doc::slot2** (*object* *example-class*))

Writer

(setf (**sphinxcontrib.cldomain.doc::slot2** (*object* *example-class*)) (*new-value* T))

the second slot.

CLOS slot sphinxcontrib.cldomain.doc::slot1

Type
sb-mop:standard-direct-slot-definition

Initarg
:slot1

Reader

(**sphinxcontrib.cldomain.doc::slot1-alt** (*object* *example-class*))

Reader

(**sphinxcontrib.cldomain.doc::slot1** (*object* *example-class*))

Writer

(setf (**sphinxcontrib.cldomain.doc::slot1-alt** (*object* *example-class*)) (*new-value* T))

Writer

(setf (**sphinxcontrib.cldomain.doc::slot1** (*object* *example-class*)) (*new-value* T))

the first slot.

15.3 Code

```
(define-condition example-error (simple-error)
    ((message
      :initarg :message
      :accessor error-message
      :initform nil
      :documentation "Message indicating what went wrong."))
  (:documentation "An example condition"))
```

15.4 Output

CLOS class example-error

Superclass
['T']

Metaclass
sb-pcl::condition-class

Initargs

- **:message** – *message*

An example condition

CLOS slot sphinxcontrib.cldomain.doc::message

Type
sb-pcl::condition-direct-slot-definition

Initarg
:message

Reader

(**sphinxcontrib.cldomain.doc::error-message** (*object example-error*))

Writer

(setf (**sphinxcontrib.cldomain.doc::error-message** (*object example-error*)) (*new-value* T))

Message indicating what went wrong.

CHAPTER

SIXTEEN

RUNNING TEST

To run the tests use

::

```
make test
```

CHAPTER

SEVENTEEN

MANUALLY VERIFYING LISP CODE

To manually test documentation string collection of a library you can run command like this.

```
ros -Q -- sphinxcontrib/cldomain/cldomain.ros --package alexandria --system alexandria
```

If you want to test it with a local system, you can run.

```
ros -Q -- sphinxcontrib/cldomain/cldomain.ros --package my-package --system my-system --
↪path ../../path/to/my-system
```

There is a second entry point that doesn't require Roswell to use it would you would run a command like.

```
sbcl --script sphinxcontrib/cldomain/custom_command.lisp --package alexandria --system␣
↪alexandria
```

CHAPTER

EIGHTEEN

FORMATTING CODE

All code is formatted using the python package Black, you can run it via

```
make fmt
```

CHAPTER

NINETEEN

LINTING CODE

All code is linted using Flake8, you can run it via

```
make lint
```

CHAPTER

TWENTY

RELEASING NEW VERSIONS

This project depends on a hacked together half finished project that provides a wrapper for some commands to assist generating the change log and other aspects of the release process. The process will be documented here but it's currently so wrought with terror that it's unrealistic that anyone other than the author would have the time or patience to bother with it.

1. `goin prepare-release`

This updates the changelog and calculates the next version

2. `goin build-release-artifacts`

This will create an egg that can be tested manually.

3. Manually verify the release artefact

Rebuild cl-git and any other large project to verify that no significant regressions have occurred.

4. `goin test-release-artifacts`

This performs a test upload of the egg and also some basic checks.

4. `goin release-new-version`

This actually creates the tag, using the message declared in `.git/RELEASE_CHANGELOG`.

5. `git push --tags && git push`

Push the newly created tags

6. `goin release-artifacts`

Push the newly created eggs to PYPI.

CHAPTER

TWENTYONE

0.19.0 - 2024-04-28

21.1 Features

- allow declaration of packages separate from systems
- load missing package data on demand
- support using a custom lisp executable to run backend

21.2 Minor Fixes

- handle lambda lists like (foo bar . baz)
- quote all symbols so that log messages are easier to read
- to support windows better call script instead of relying on shebang
- support class specializers like (eql (find-class 'my-class))
- support eql specialisers that match non-symbol values
- rename specializer from eq to eql
- raise exceptions that have better descriptions
- return correct Sphinx extension metadata object when initialising plugin
- set correct default value for cl_packages config

21.3 Documentation

- improve configuration documentation

21.4 Tests

- add a test for the package json encoding
- improve coverage of encode-specializer

21.5 Cleanups

- remove dependency on pants

CHAPTER

TWENTYTWO

0.18.1 - 2023-06-04

22.1 Minor Fixes

- handle condition slot documentation strings
- rendering of slot's in pdf's

22.2 Documentation

- cleanup index page
- fix pdf and info links
- build latex after rendering

CHAPTER

TWENTYTHREE

0.18.0 - 2023-03-04

23.1 Features

- add support for documenting CLOS object slots

23.2 Minor Fixes

- raise a runtime error if the package is missing
- remove nospecializers method option
- handle nil arguments correctly
- convert desc_sig_keyword to desc_clparameter
- add &body to potential lambda list keywords
- add missing sphinx dependency
- print the localised name of the object type
- add print-object methods to help with debugging
- add missing :name values for variables and classes
- cleanup conditional
- simplify xref logic
- handle classes with no slots
- cleanup formatting and fix tests

23.3 Documentation

- update documentation and restructure
- update makefile and doc building
- update theme to custom cldomain theme

23.4 Cleanups

- cleanup licensing dates

CHAPTER

TWENTYFOUR

0.17.1 - 2023-01-18

24.1 Minor Fixes

- remove clear_doc method

CHAPTER

TWENTYFIVE

0.17.0 - 2023-01-18

25.1 Features

- refactor object backend
- cleanup generic/method linking
- update the generic linking so it's less obtrusive
- setf expander support

25.2 Minor Fixes

- cleanup specializer handling

25.3 Documentation

- changelog had the wrong title headings

25.4 Build Tooling

- add example envrc

CHAPTER

TWENTYSIX

0.16.2 - 2023-01-08

26.1 Minor Fixes

- add missing roswell file

CHAPTER

TWENTYSEVEN

0.16.1 - 2023-01-08

27.1 Minor Fixes

- add back files missing from dist

CHAPTER

TWENTYEIGHT

0.16.0 - 2023-01-08

28.1 Features

- rename type to class

28.2 Minor Fixes

- fix method arguments in PDF output closes #7
- fix dictionary changed size during iteration
- rename type to class, in reality we are documenting classes, not types.
- bump pants to 2.14.0
- remove list_unused_symbols
- disable more warnings

28.3 Documentation

- add PDF and Info examples to documentation
- update changelog
- update bugtracker and documentation url
- fix sphinx url
- fix reference to pdf

28.4 Tests

- add tests for types, clos classes
- hookup lisp tests

28.5 Build Tooling

- migrate from pants to pyproject for building

28.6 Cleanups

- modernise system definition

CHAPTER

TWENTYNINE

0.15.3 - 2022-07-24

- assign *TRACE-OUTPUT* and *DEBUG-IO* to *ERROR-OUTPUT*

CHAPTER THIRTY

0.15.2 - 2022-07-24

- fix don't decode bytes before writing them

CHAPTER THIRTYONE

0.15.1 - 2022-07-24

- fix decode bytes before writing them

CHAPTER

THIRTYTWO

0.15 - 2022-07-23

- stop qualifying lambda list symbols with a package
- fix display of method specializer links #16
- fix labelling of link back to generic

CHAPTER

THIRTYTHREE

0.14 - 2022-07-10

- convert to unix-opts, because i couldn't get clon to work
- strip packages from symbols if it's the current package, so CL-GIT::BODY would become BODY.
- add whitespace between method arguments so method `(full-name (objectreference))` will print as method `(full-name (object reference))`
- symbols that a appear at the start of newlines are now correctly rendered, this might break CLISP, but will work in SBCL. The bug was introduced by trying to support CLISP, but i think valid rendering trumps multiplatform support for now.

CHAPTER

THIRTYFOUR

0.13 - 2015-09-06

- updated com.dvlsoft.clon to net.didierverna.clon.

CHAPTER

THIRTYFIVE

0.12 - 2015-02-24

- fixed argument generation bug.

CHAPTER

THIRTYSIX

0.11 - 2014-12-30

- support loading symbol information from multiple packages.

CHAPTER

THIRTYSEVEN

0.10 - 2014-06-12

- added back parentheses to parameter lists.
- added type information to parameter list of methods.
- added links to other methods from a method docstring.
- fixed bug with macro documentation strings.
- added better keyword detection in documentation strings.
- fixed bug where symbols at the end of documentation strings were ignored.

CHAPTER

THIRTYEIGHT

0.9 - 2014-02-10

- fixed problem with version number generation.

CHAPTER

THIRTYNINE

0.8 - 2014-02-10

- fixed bug with lisps argument.
- removed dependency on swank.
- remove specializers symbols package if it's the current package.

CHAPTER

FORTY

0.7 - 2013-06-12

- started to make internals more modular.
- print specialisation for methods.
- add links to method specializers.
- added methods to index.

CHAPTER

FORTYONE

0.6 - 2013-04-22

- added more documentation.
- added better error handling when json fails to parse.
- methods can now pull documentation from their generic.

CHAPTER

FORTYTWO

0.5 - 2013-04-20

- inherit environment when calling subprocesses.
- better handling of symbols in doc strings.

CHAPTER FORTYTHREE

0.4 - 2013-04-19

- fixed some packaging bugs.
- made the data model more tolerant to missing symbols.
- fixed symbol resolving bug.
- added output of unused symbols.

CHAPTER

FORTYFOUR

0.3 - 2013-04-16

- cleaned up specializer output.
- fixed bug when rendering specializers that have the form :KEYWORD SYMBOL.
- updated documentation.
- split out package code from lisp program.

CHAPTER FORTYFIVE

0.2 - 2013-04-14

- link between generics and specializers.
- ignore symbols in documentation if they are in the arg list.
- better Quicklisp support.
- handling of symbols that boarder on punctuation.

CHAPTER

FORTYSIX

0.1 - UNRELEASED

- initial prototype

INDEX

Symbols

C

E

M

S