Clojure Documentation

Clojure

Clojure Users Guide

Notice

Notice

Topics:

This information was developed for products and services offered in the U.S.A.

This product is meant for educational purposes only. Some of the trademarks mentioned in this product appear for identification purposes only. Not responsible for direct, indirect, incidental or consequential damages resulting from any defect, error or failure to perform. Any resemblance to real persons, living or dead is purely coincidental. Void where prohibited. Some assembly required. Batteries not included. Use only as directed. Do not use while operating a motor vehicle or heavy equipment. Do not fold, spindle or mutilate. Do not stamp. No user-serviceable parts inside. Subject to change without notice. Drop in any mailbox. No postage necessary if mailed in the United States. Postage will be paid by addressee. Post office will not deliver without postage. Some equipment shown is optional. Objects in mirror may be closer than they appear. Not recommended for children. Your mileage may vary.

No other warranty expressed or implied. This supersedes all previous notices.

COPYRIGHT LICENSE:

Trademarks

The following terms are trademarks of the Foobar Corp. in the United States, other countries, or both:

Foobarista ®

The following terms are trademarks of other companies:

Red, Orange, Yellow, Green, Blue, Indigo, and Violot are registered trademarks of Rainbow Corporation and/or its affiliates.

Linux is a trademark of Linus Torvalds in the United States, other countries, or both.

Other company, product, or service names may be trademarks or service marks of others.

Contents

Abstract

User Guide for Clojure. See also the Technical Manual.

Preface

About Clojure

Plan of the Work

The basic approach: describe the problem, then describe general strategies for addressing the problem, then describe Clojure-specific solutions, then give an example.

Key distinctions: Clojure “task” (i.e. command) v. developer's task (e.g. test, document); project map (i.e. configuration) v. command

Note: Clojure's terminology should be changed to use “command” where it currently uses “task”. This is because the tasks are not in fact tasks; e.g. `lein compile` is a command from the user to Clojure rather than a task to be accomplished. We need to reserve the word “task” to refer to “user tasks”, the tasks confronting the developer, which appear as problems to be solved. So for example the first task is merely to configure the project. The way to solve this problem is to write the project.clj and set up the directory structure.

Plan of the work:

Overview	installation, projects, infrastructure
Project Maps	How the projmap is constructed at runtime; how tasks use the projmap to drive configuration of the
Tasks	Two kinds of task: • Dev tasks, e.g coding, testing, QA, etc • Clojure “tasks”, i.e. commands General description of tasks; commonly used tasks with examples
Infrastructure: Respositories	role of repos; how to specify, etc. and how Clojure searches, etc
etc	etc

Chapter 1

Introduction: Configuration, Customization, Coordination

Topics:

"Clojure is for automating Clojure projects without setting your hair on fire."

Clojuret is a tool for:

Configuration	Use it to "configure" tools, i.e. set options, for various tasks.
Customization	Groups and individuals can “inherit” general configurations and then customize them on a per-task and/or per-project basis.
Coordination	Use it to integrate and coordinate related activities. For example, quality assurance, code check-in, and issue tracking are related activities. Clojure can help you automate standard procedures such as: first QA check code, then check it in with a reference to the bug it fixes, and then close the bug in the Issues Tracker, annotating it with the commit ID.

The Problem

Clojuret is a general “CCC” (configuration, customization, and coordination) tool.

This section describes the sort of problems Clojure is designed to solve: configuration, coordination, etc.

See this thread *https://groups.google.com/forum/#!topic/clojure/OiAijeJVr6k*

The Solution

This section describes in general how Clojure solves the problems described in the previous topic. I.e. it's more about the strategies Clojure adopts than the specific implementation techniques; they are the subject of the remainder of this book..

etc

Chapter 2

What is Clojure?

Topics:

Clojure is a collection of commands that address developer tasks and needs. It depends on a kernel of core functionality, a set of plugins that implement the commands available to users, and a set of external resources - directories, files, maven repositories, etc.

Clojure can be compared to Emacs. Emacs consists of a kernel of core functionality (implemented in C) made available to the user by means of commands or functions defined in Elisp, just like user-defined functions and commands. Clojure also has a kernel of core functionality (implemented in Clojure) that is made available to users by means of *plugins*, which define Clojure *commands*; user-defined commands are implemented in the same way.

Structure & Dynamics of Clojure Projects

Structure of project: project config map (in project.clj) determines dir structure.

Dynamics: when a lein command is invoked, lein dynamically constructs the effective project map and invokes the command, passing the EPM as arg. The command inspects the map and uses its content to control its processing.

Project Maps

The central concept of Clojure is the *project map*.

What does a project map look like? An ordinary Clojure map: set of key-val pairs. In the project map, keys are always Clojure :-prefixed keys. L defines a default set of keys; users are free to extend this set.

The Project Configuration Map

The *project configuration map* is defined by `defproject` macro in the `project.clj` file in the project *root directory*.

Terminology: the *project configuration map* is specified as part of `defproject` in `project.clj`. The *effective project map* is dynamically constructed by combining the project config map and various other maps - see X for details.

Profiles

Profiles provide a means of customizing the effective project map.

A profile is a named map. Clojure predefines several *system profiles*: :base, :system, :user, :dev, and :provided. The system profiles are always in effect; user-defined can override and/or extend them. User-defined profiles can be specified in several places:

At runtime, Clojure integrates profiles with the project configuration map of `project.clj` to produce the effective project map.

The Effective Project Map

The *effective project map* is dynamically constructed by combining the project config map and various other maps

Clojure command implementations receive the effective project map as an argument when they are invoked.

General description of how L constructs the *effective project map* from project.clj/defproject in combination with various other maps. This is one of the basic jobs of the kernel. For details, refer to the Project Map node.

Project Map Semantics

Project map semantics are determined by command implementations.

What are the semantics of the (effective) project map? Determined entirely by the command implementations. All Clojure commands use the effective project map, which is delivered by Clojure to the command as its sole argument. Different commands pick out different parameters from the map for use in controlling processes.

Commands

Clojure *commands* (formerly “tasks”) expose functionality to the user.

Commands are what you would expect: something you type to tell lein what to do.

L comes with a set of predefined commands. Users can extend this set by writing a *plugin*, which is the implementation of a command. There is no technical difference between L's predefined commands and user-defined commands.

The Clojure Infrastructure

Clojure's infrastructure config files, the local repo, remote repos, etc. We should also include default profiles such as :base, since they are just as fundamental. This topic provides a brief overview of these parts and how they fit together.

External

Directories, files, env vars used by Clojure

Directories: ~/.lein, ~/lein/profiles.d, etc; ~/.m2/repository;

Files: ~/.lein/profiles.clj

Environment variables

Command line options

Anything else Clojure can take from the env?

Do this

Internal infrastructure: profiles, etc.

Profiles and other internally defined (but overridable) stuff on which Clojure's functionality depends.

Getting Started

We get started by first exploring a minimal project, and then exploring the Clojure infrastructure. This should give us a good general idea of what pieces are involved and how the work together.

Before delving into the details we take a brief tour of:

- A minimal project
- The Clojure infrastructure

Installation

Unix-like systems

If your preferred *package manager* has a relatively recent version of Clojure, try that first. Otherwise you can install by hand:

Clojure bootstraps itself using the `lein` shell script; there is no separate install script. It handles installing its own dependencies, which means the first run will take longer.

1. Make sure you have a Java JDK version 6 or later.
2. *Download the script*
3. Place it on your *$PATH*. (`~/bin` is a good choice if it is on your path.)
4. Set it to be executable. (`chmod 755 ~/bin/lein`)

Windows

There is an *installer* which will handle downloading and placing Clojure and its dependencies.

The manual method of putting the *batch file*. on your *PATH* and running `lein self-install` should still work for most users. If you have Cygwin you should be able to use the shell script above rather than the batch file.

A Quick Tour

We create a minimal project and explore its structure and configuration.

A minimal Clojure project involves: a directory structure, a project configuration file (project.clj), and source code files. In addition to these static resources, Clojure dynamically determines a project map, a dependency tree, and several other structures. This section takes you through some simple steps to exlore these elements.

1. Create a new project by executing: `lein new app my-app`

   where `lein` is the Clojure command, and the args are:
   new

a Clojure *task*. To see the syntax and semantics of this task, run `lein help new`. To see a list of the tasks that come preinstalled, run `lein help`.

app

is the name of a project template, in this case the default application template. To see a list of preinstalled templates, run `lein help new`. Clojure's concept of *task* is discussed in detail under the X topic of this guide.

my-app

is the name to be used for the new project

This will create a hierarchy of directories and populate it with some files generated from templates:

```
.
./.gitignore
./doc
./doc/intro.md
./project.clj
./README.md
./src
./src/my_app
./src/my_app/core.clj
./test
./test/my_app
./test/my_app/core_test.clj
```

2. Take a look at the Clojure project file buy running `less my-app/project.clj`
You should see something like:

```
(defproject my-app "0.1.0-SNAPSHOT"
  :description "FIXME: write description"
  :url "http://example.com/FIXME"
  :license {:name "Eclipse Public License"
            :url "http://www.eclipse.org/legal/epl-v10.html"}
  :dependencies [[org.clojure/clojure "1.5.1"]])
```

Note: This has the form of a function application, where `defproject` is the function, and the args come as a list of pairs. In fact, `defproject` is a macro that expands into the definition of a Clojure map named `project`. Each task - both those preinstalled and those defined by plugins - takes this `project` map as input. The key point is that the key-value pairs of the `project` map are thus available for use by task implementations to control setting of options etc. for the processes they manage.

Note: The final `project` map is determined by a combination of several sources in addition to project.clj, such as *~/.lein/profiles.clj*. See X for details.

In this minimal example, the only really functional parameter is `:dependencies`. This is used to specify which libraries/jars the project uses (and thus depends on). One of the best things about Clojure is its powerful management of dependencies. Using the [?] library under the hood, Clojure is able to construct the entire dependency graph for the project and arrange for everything needed to be installed. Clojure's dependency management capabilities are described in X.

Important: Clojure dependency specification uses the naming conventions established by Maven: artifact-group/artifact-id. Etc. See X for details.

3. Examine the project map: `lein pprint`
You should see something like:

```
{:compile-path "/Users/gar/tmp/my-app/target/classes",
 :group "my-app",
 :license
```

```
 {:name "Eclipse Public License",
  :url "http://www.eclipse.org/legal/epl-v10.html"},
 :global-vars {},
 :checkout-deps-shares
 [:source-paths
  :test-paths
  :resource-paths
  :compile-path
  #&Var@3e25e2b8:
    #amp;classpath$checkout_deps_paths leiningen.core.classpath
$checkout_deps_paths@89bbe8c>>],
 :dependencies
 ([org.clojure/clojure "1.5.1"]
  [org.clojure/tools.nrepl "0.2.3" :exclusions ([org.clojure/clojure])]
  [clojure-complete/clojure-complete
   "0.2.3"
   :exclusions
   ([org.clojure/clojure])]),
 :plugin-repositories
 [["central" {:snapshots false, :url "http://repo1.maven.org/maven2/"}]
  ["clojars" {:url "https://clojars.org/repo/"}]],
 :test-selectors {:default (constantly true)},
 :target-path "/Users/gar/tmp/my-app/target",
 :name "my-app",
 :deploy-repositories
 [["clojars"
   {:username :gpg,
    :url "https://clojars.org/repo/",
    :password :gpg}]],
 :root "/Users/gar/tmp/my-app",
 :offline? false,
 :source-paths ("/Users/gar/tmp/my-app/src"),
 :certificates ["clojars.pem"],
 :version "0.1.0-SNAPSHOT",
 :jar-exclusions [#"^\."],
 :profiles {:uberjar {:aot :all}},
 :prep-tasks ["javac" "compile"],
 :url "http://example.com/FIXME",
 :repositories
 [["central" {:snapshots false, :url "http://repo1.maven.org/maven2/"}]
  ["clojars" {:url "https://clojars.org/repo/"}]],
 :resource-paths
 ("/Users/gar/tmp/my-app/dev-resources"
  "/Users/gar/tmp/my-app/resources"),
 :uberjar-exclusions [#"(?i)^META-INF/[^/]*\.(SF|RSA|DSA)$"],
 :main my-app.core,
 :jvm-opts ["-XX:+TieredCompilation" "-XX:TieredStopAtLevel=1"],
 :eval-in :subprocess,
 :plugins
 ([lein-localrepo/lein-localrepo "0.4.1"]
  [lein-difftest/lein-difftest "1.3.8"]
  [org.clojure/java.classpath "0.2.0"]
  [lein-marginalia/lein-marginalia "0.7.1"]
  [lein-mustache/lein-mustache "0.2"]
  [lein-pprint/lein-pprint "1.1.1"]),
 :native-path "/Users/gar/tmp/my-app/target/native",
 :description "FIXME: write description",
 :test-paths ("/Users/gar/tmp/my-app/test"),
 :clean-targets [:target-path],
 :aliases nil}
```

Take some time to look this over. Most of these parameters you will never have to deal with, but it's good to have an idea of what sorts of things Clojure is interested in.

Note: The predefined paramters are documented in the Technical Reference Manual. Other chapters of this User's Guide explain how to use them.

Note: We should distinguish between the project map that results from Clojure's work and the defproject map in project.clj that forms the starting point for project map construction.

4. Tell Clojure to install all dependencies by running `lein deps`

   Since you already have Clojure installed, you probably won't see any output.

5. Check your dependency tree: `lein deps :tree`
   You should see something like the following:

```
[clojure-complete "0.2.3" :exclusions [[org.clojure/clojure]]]
[org.clojure/clojure "1.5.1"]
[org.clojure/tools.nrepl "0.2.3" :exclusions [[org.clojure/clojure]]]
```

   This is a complete listing of the jars your project depends on, derived from your project.clj :dependencies parameter plus a set of default maps to be described later. Since this is a tree representation, you can infer that these three libraries are independently specified; none of them has required any of the others as a dependency. In fact, `clojure-complete` and `org.clojure/tools.nrepl` are both specified as dependencies by the default `:base` *profile*, which means that they will always be in the dependency tree for every project (unless overriden). Profiles are named maps that can be used to customize the `project` map in various ways; they are fully described in X.

6. Now let's take a look at the application code installed by the `app` template. Use your editor, or run `cat src/my_app/core.clj`

   The contents of `my-app/src/my_app/core.clj` should look something like:

```
(ns my-app.core
  (:gen-class))

(defn -main
  "I don't do a whole lot ... yet."
  [& args]
  (println "Hello, World!"))
```

Notice: The `:gen-class` option and the definition of `-main`. Remember this was produced by the application template, rather than the library template. So it assumes you want to execute the result, which means you need to generate Java byte code. That's what :gen-class does. You also need a main entry point; thats what `-main` does.

Warning: In case it isn't obvious: to effectively use Clojure, you need to know Clojure. That is, the better your mastery of Clojure, the more you can do with Clojure. Clojure is a Clojure application, after all.

The Clojure Infrastructure

Clojure's infrastructure config files, the local repo, remote repos, etc. We should also include default profiles such as :base, since they are just as fundamental. This topic provides a brief overview of these parts and how they fit together.

External

Directories, files, env vars used by Clojure

Directories: ~/.lein, ~/lein/profiles.d, etc; ~/.m2/repository;

Files: ~/.lein/profiles.clj

Environment variables

Command line options

Anything else Clojure can take from the env?

Do this

Internal infrastructure: profiles, etc.

Profiles and other internally defined (but overridable) stuff on which Clojure's functionality depends.

Chapter 3

Bindings

Topics:

Bindings are Clojure's way of doing osmosis.

Bindings are:

- A Trojan horse that spills hordes of data into the streets of Illium
- A Robin Hood, that breaks open the treasure chests of traveling data and distributes the contents to the locals
- Che! a liberator that rips open stodgy ol' data structures and liberates the data
- A transport mechanism for shuttling information from the outside world into closed computations like `let` and function bodies
- Actually, just a function from names to values

Overview

A binding is a function from names to values.

Clojure says: `binding => binding-form init-expr`

Z says: a *binding* is a "finite function from names to values". A set of bindings is a *schema*

Z uses a special symbol for bindings: (\u2989 Z NOTATION LEFT BINDING BRACKET) and (\u298A Z NOTATION RIGHT BINDING BRACKET). So a *binding construction* expression looks like <| foo bar |>.

I find Clojure's terminology ("binding-form") a little confusing so I'm going to stick with Z, and say a binding is a pair of a name and a value expression, with the result being a bound name or variable. (Plus, the Clojure doc is slightly inconsistent between "binding-form" and "binding symbol".) Ah, but the reason for "binding-form" is that you can use structures instead of names, to get indirect binding.

blah blah

Binding Construction

A binding is a function from names to values, i.e. a set of ordered pairs. Bindings are *expressed* via *binding construction expressions*, or binding constructors for short. So we need to make a distinction between the binding and the expression. Z notation is useful since it provides a rigorously defined, minimal notation for semantics: to say that 2 is bound to x you write `<|a=2|>`, no matter how that was expressed syntactically.

The simplest binding constructor just pairs a name (technically, a symbol) and a (simple) value, just like the Z notation: `(let [a 1]...) == <|a=1|>`. Slighly more complex constructors can use complex expressions on the value side, as in `(let [foo (+ 1 2)] ...) == <|foo=3|>`. More complex yet, binding constructors can also put complex expressions on both sides, thus pairing name and value *structures*; examples below.

Binding Contexts

- `let`
- function parameter definitions
- macros that expand into `let` or function expressions

Simple Bindings

Simple bindings ...

For example:

```
user=> (let [a 1 b 2 c 3] [a b c])
[1 2 3]
```

Structural Bindings

Clojure supports abstract structural binding, often called destructuring, in let binding lists, fn parameter lists, and any macro that expands into a let or fn. The basic idea is that a binding-form can be a data structure literal containing symbols that get bound to the respective parts of the init-expr. The binding is abstract in that a vector literal can bind to anything that is sequential, while a map literal can bind to anything that is associative.

Vector Destructuring

Vector binding-exprs allow you to bind names to parts of sequential things (not just vectors), like vectors, lists, seqs, strings, arrays, and anything that supports nth. The basic sequential form is a vector of binding-forms, which will be bound to successive elements from the init-expr, looked up via `nth`. In addition, and optionally, & followed by a binding-form will cause that binding-form to be bound to the remainder of the sequence, i.e. that part not yet bound, looked up via nthnext .

Finally, also optional, `:as` followed by a symbol will cause that symbol to be bound to the entire init-expr:

```
(let [[a b c & d :as e] [1 2 3 4 5 6 7]]
[a b c d e])

->[1 2 3 (4 5 6 7) [1 2 3 4 5 6 7]]
```

These forms can be nested:

```
(let [[[x1 y1][x2 y2]] [[1 2] [3 4]]]
[x1 y1 x2 y2])

->[1 2 3 4]
```

Strings work too:

```
(let [[a b & c :as str] "asdjhhfdas"]
[a b c str])

->[\a \s (\d \j \h \h \f \d \a \s) "asdjhhfdas"]
```

Examples

Suppose you had a vector `v == [1 2]` and you wanted to work with its elements in a binding context (`let` or function body). Without structural binding, you would have to destructure `v` by hand, as it were, and explicitly project its elements into the local environment. For example:

```
(def v [1 2])
...
(let [a (nth v 0)
      b (nth v 1)]
  ;; do something with a and b
  )
```

This works, but it would be cumbersome with larger or more complex structures. Structural binding constructors make such "destructuring" a snap:

```
(let [[a b] v]
  ;; do something with a and b
  )
```

Notice the structural parallel between the input datum - a vector of ints `[1 2]` - and the structure with which it is paired by the binding - another vector of the same length (`[a b]`), but whose elements are symbols (i.e. names). What makes this work is that the binding logic can use `nth` behind the scenes to iterate over the contents of `v` and project them to the corresponding elements of `[a b]`, much like we did by hand in the previous example.

A critical point is that even though `a` and `b` are embedded *syntactically* in a vector in the binding constructor, they can be directly used in the local environment. You do not have to use `nth` to get at `a` and `b`. So not only does the binding mechanism "transport" data from an incoming structure to a local structure (namely `[a b]`), it also, in turn, projects the components of that local structure into the local environment. So it "transports" (i.e. binds) 1 to a, and then "transports" a from its vectorial starting point to the local environment.

Map Destructuring

Map binding-forms allow you to bind names to parts of associative things (not just maps), like maps, vectors, string and arrays (the latter three have integer keys). It consists of a map of binding-form-key pairs, each symbol being bound to the value in the init-expr at the key. In addition, and optionally, an :as key in the binding form followed by a symbol will cause that symbol to be bound to the entire init-expr. Also optionally, an :or key in the binding form followed by another map may be used to supply default values for some or all of the keys if they are not found in the init-expr:

```
(let [{a :a, b :b, c :c, :as m :or {a 2 b 3}} {:a 5
:c 6}] [a b c m])
->[5 3 6 {:c 6, :a 5}]
```

It is often the case that you will want to bind same-named symbols to the map keys. The :keys directive allows you to avoid the redundancy: `(let [{fred :fred ethel :ethel lucy :lucy} m] ...` can be written: `(let [{:keys [fred ethel lucy]} m] ...`

As of Clojure 1.6, you can also use prefixed map keys in the map destructuring form:

```
(let [m {:x/a 1, :y/b 2}
{:keys [x/a y/b]} m]
(+ a b))

-> 3
```

As shown above, in the case of using prefixed keys, the bound symbol name will be the same as the right-hand side of the prefixed key. You can also use auto-resolved keyword forms in the :keys directive:

```
(let [m {::x 42}
{:keys [::x]} m]
x)

-> 42
```

There are similar :strs and :syms directives for matching string and symbol keys, the latter also allowing prefixed symbol keys since Clojure 1.6.

Nested Destructuring

Since binding forms can be nested within one another arbitrarily, you can pull apart just about anything:

```
(let [{j :j, k :k, i :i, [r s & t :as v] :ivec, :or {i 12 j 13}}
```

```
      {:j 15 :k 16 :ivec [22 23 24 25]}]
      [i j k r s t v])

      -> [12 15 16 22 23 (24 25) [22 23 24 25]]
```

Part I

Extending Clojure: Macros

Topics:

Chapter 1

Template Development

How to implement custom templates.

Should this go in a separate “Developer's Manual”?

Appendix

Appendices

Topics:

Appendix
A

Contributing to this document

WARNING: Document design can be hazardous to your other interests. (Apologies to Knuth)

This is a DITA document. The original is available from the *github project*.

Please use the *Issues Tracker* to register bugs, corrections, enhancement requests, etc. Alternatively, you can clone the repository and edit the text directly. If you do please credit yourself in the metadata.

Intellectual property

DITA

DITA (Darwin Information Typing Architecture) is an *OASIS Standard* for writing, managing, and publishing information. It supports "topic-based" authoring.

The *DITA Wiki Knowledgebase* contains lots of information on DITA, from introductions and overviews to detailed documentation.

Another good overview is *DITA for the Impatient* from *XMLMind*.

The *DITA Language Specification* documents the elements and atributes of the DITA Schema.

Tools

There are two open source DITA transformation tools available:

- *DITA Open Toolkit*
- *XMLMind DITA Converter*