



Nico

Clojure for pleasure

Cool Clojure and how to get started quickly

Content

| | | | |
|----|--------------------|----|--------------------------------|
| 4 | Welcome to Clojure | 21 | Vaadin |
| 4 | Clojure | 21 | clj-opennlp |
| 4 | Leiningen | 22 | clj-http |
| 8 | Jark | 22 | Cascalog |
| 9 | Pomegrenate | 22 | Aleph |
| 10 | Libraries | 22 | Lamina |
| 10 | Noir | 22 | Conduit |
| 12 | Incanter | 23 | Processing, OpenCV and imaging |
| 13 | Marginalia | 23 | Quil |
| 15 | Lacij | 25 | OpenCV, Arduino |
| 16 | Seesaw | 25 | Physics with Clojure |
| 17 | clj-xpath | 25 | Cloduino |
| 18 | clojure-soup | 27 | Live Music |
| 19 | Enlive | 27 | Overtone |
| 19 | Docjure | 28 | In the Clouds |
| 20 | Postal | 28 | Pallet |
| 21 | Korma | 28 | Heroku |

Chapter 1

Welcome to Clojure

This book is about sharing my love for Clojure, why I use it, and why I will keep using it for a fair bit of time. You don't need to be a great IT geek, you just have to start using Clojure. Here we go.

Clojure

http://learn-clojure.com/clojure_tutorials.html

I will not go too much into the first steps details, since this is not the aim of this book. There is a completely new online book written by [John](#) that should get all the attention it deserves.

Do not forget to do a few [clojure koans](#) to make sure the basic and the roots are ok.

Leiningen

<https://github.com/technomancy/leiningen>

To start using clojure, you actually do not install it. You start by installing Leiningen. Once you have it installed, you should be able to see the following:

```
[Niko@Modrzyks-MacBook-Pro][13:44][~/projects/] % lein  
Leiningen is a tool for working with Clojure projects.
```

Several tasks are available:

classpath Print the classpath of the current project.
 clean Remove compiled class files and jars from project.
 compile Compile Clojure source into .class files.
 deploy Build jar and deploy to remote repository.
 deps Download :dependencies and put them in :library-path.
 help Display a list of tasks or help for a given task.
 install Install current project or download specified project.
 interactive Enter an interactive task shell. Aliased to "int".
 jar Package up all the project's files into a jar file.
 javac Compile Java source files.
 new Create a new project skeleton.
 plugin Manage user-level plugins.
 pom Write a pom.xml file to disk for Maven interop.
 repl Start a repl session either with the current project or standalone.
 retest Run only the test namespaces which failed last time around.
 run Run the project's -main function.
 search Search remote maven repositories for matching jars.
 test Run the project's tests.
 test! Run a project's tests after cleaning and fetching dependencies.
 trampoline Run a task without nesting the project's JVM inside Leiningen's.
 uberjar Package up the project files and all dependencies into a jar file.
 upgrade
 version Print version for Leiningen and the current JVM.

Run `lein help $TASK` for details.

See also: [readme](#), [tutorial](#), [copying](#), [sample](#), [deploying](#) and [news](#).

Now if you just want to start using some live clojure, you just go with:

```
[Niko@Modrzyks-MacBook-Pro][13:50][~/] % lein repl  
REPL started; server listening on localhost port 27991  
user=>
```

And there you are. Was simple no ?

And here is your mother of them all hello world

```
; yes indeed  
(println "hello clojure!")  
  
; include a new namespace in the current namespace  
; and use a custom name  
(require '[clojure.string :as string])  
  
; use the newly imported namespace  
(def new_string (string/replace "Hello World" "World" "Nico"))  
  
; print  
(println new_string)
```

Now you have to realize that all the libs available to the Java World through Maven repositories, and all the libs hosted on Clojars can be integrated in the project in no time by using the command:

lein deps

The libraries will come to your local project folder. Let's hack in no time.

Search clojars

To search a library online, use the following syntax:

```
lein search <libraryname>
```

For example

```
lein search jsoup
```

will return

```
== Results from clojars - Showing page 1 / 1 total [org.clojars.clizzin/jsoup "1.5.1"] jsoup HTML parser
```

Upload to clojars

There is a plugin called [lein-clojars](#)

You install it somehow like this:

```
lein plugin install lein-clojars 0.9.1
```

Then in a clojure project, just go and upload it with

```
lein push
```

Nothing else.

Then a lein search will (should) show it.

Examples

All the examples in this book can be run with: `(use 'clojure.string) (load-file "code/18_clojure_soup.clj")`

If you can not run them, let me know ;)

Jark

<http://icylisper.in/jark/started.html>

Next you probably going to get tired of the Java VM so very slow startup time. Here comes `nrepl` and `Jark`

Create a new leiningen project, and follow the jark install steps:

```
; project.clj
(comment
  ; include this in project.clj
  (defproject book01 "1.0.0-SNAPSHOT"
    :dependencies [
      [org.clojure/clojure "1.3.0"]
      [jark "0.4.2" :exclusions [org.clojure/clojure]]
    ])

; now let's start the embedded repl
(require '[clojure.tools.nrepl :as nrepl])
(nrepl/start-server 9000)
```

and then start the repl with the new code from `core.clj`


```
[Niko@Modrzyks-MacBook-Pro][14:11][~/projects/book01/] % lein repl  
REPL started; server listening on localhost port 42326  
user=> (load-file "src/book01/core.clj")
```

And now straight from the shell you can use and run remote clojure code

```
[Niko@Modrzyks-MacBook-Pro][14:16][~/Downloads/jark-0.4.2-Darwin-x86_64/] % jark -e "(+ 2 2)"  
=> 4
```

Now that is some sweet fast starting time, so stop buzzing around.

Pomegranate

<https://github.com/cemerick/pomegranate>

This is the best way to add libraries at runtime. You still need the library itself to be available to the repl, by adding this to your project.clj file:

```
[com.cemerick/pomegranate "0.0.13"]
```

Then, here is how to import incanter in a running REPL session:

```
[missing 'code/11_pomegranata.clj' file]
```

Chapter 2

Libraries

Now that we are ready for some hacking, here are sites, that refer interesting Clojure libraries:

- [clojure-libraries on appspot.com/](http://clojure-libraries-on-appspot.com/)
- [clojurewerkz](http://clojurewerkz.com/)
- [Clojure on github](https://github.com/)
- <http://programmers.stackexchange.com/questions/125107/what-are-the-essential-clojure-libraries-to-learn-beyond-the-basics-of-core>
- <http://www.clojure-toolbox.com/>

Noir

<https://github.com/ibdknox/noir> <http://www.webnoir.org/>

It's dark in here ! Noir is probably the simplest way to write a functional web application in clojure.

This is how it looks like in Clojure code

```
(ns my-app
  (:use noir.core)
  (:require [noir.server :as server]))

(defpage "/welcome" []
```

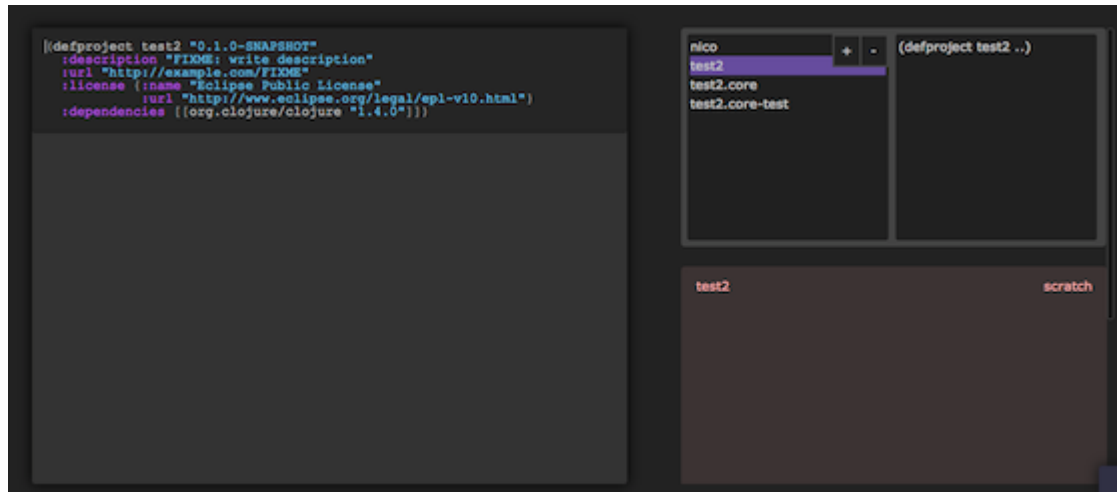
"Welcome to Noir!")

(server/start 8080)

and/or this is how you would have it started in a few seconds with the leiningen command we installed earlier on

```
lein plugin install lein-noir 1.2.1
lein noir new my-website
cd my-website
lein run
```

Now the guy from Noir has also started a very cute project named [Playground](#) where you can do live execution of your code with a nice UI.



Incanter

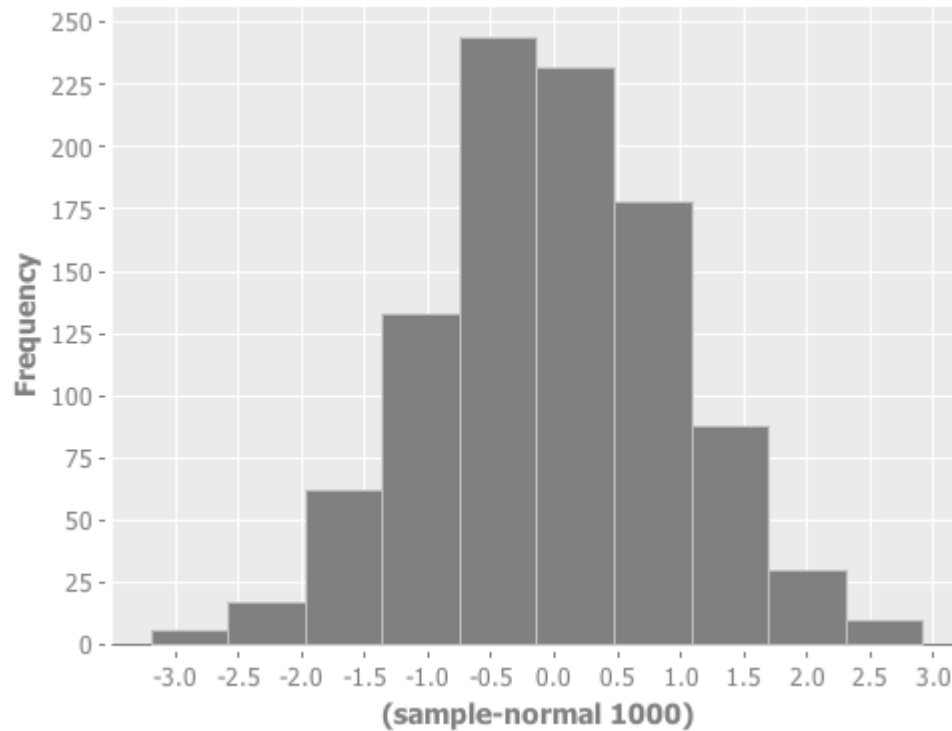
[Download](#) and [Get started](#)

From the Clojure REPL, load the Incanter libraries:

```
user=> (use '(incanter core stats charts))
```

Try an example: sample 1,000 values from a standard-normal distribution and view a histogram:

```
user=> (view (histogram (sample-normal 1000)))
```



Marginalia

<https://github.com/fogus/marginalia>

Marginalia is your best literate programming tool for clojure.

Install it as a dependency in your project.clj file:

```
:dev-dependencies [lein-marginalia "0.7.1"]
```

Then, just use it:

```
lein marg
```

And depending on the amount of comments you wrote in the code, you will get something similar to this:

quilme 1.0.0-SNAPSHOT

FIXME: write description

dependencies

| | |
|-----------------------|-------|
| org.clojure / clojure | 1.4.0 |
| overtone | 0.7.1 |
| quil | 1.6.0 |

dev dependencies

| | |
|-----------------|-------|
| lein-marginalia | 0.7.1 |
|-----------------|-------|

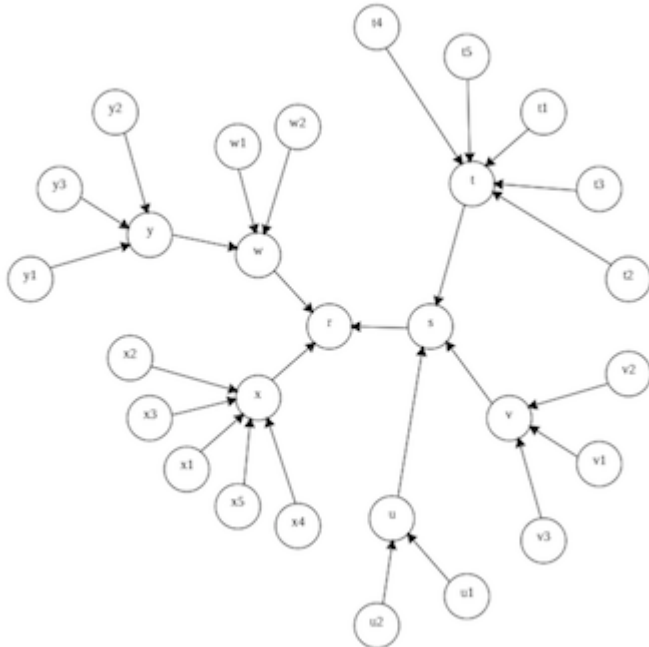
namespaces

quilme.core
quilme.example2

Lacij

<https://github.com/pallix/lacij>

A library that can quickly create SVG diagram with automatic layout, like this one:



(use `'lacij.graph.svg.graph'`)

Then, a graph can be drawn like this:

```

(-> (create-graph :width 800 :height 400)
      (add-default-node-style :fill "lightgreen")
      (add-default-edge-style :stroke "royalblue")
      (add-node :hermes "Hermes" :x 10 :y 30 :style {:fill "lightblue"})
      (add-node :zeus "Zeus" :x 300 :y 150 :rx 15 :ry 15)
      (add-node :ares "Ares" :x 300 :y 250 :style {:fill "lavender" :stroke "red"})
      (add-edge :father1 :hermes :zeus "son of"
                :style {:stroke "darkcyan" :stroke-dasharray "9, 5"})
      (add-edge :father2 :ares :zeus)
      (add-default-node-attrs :rx 5 :ry 5)
      (add-node :epaphus "Epaphus" :x 450 :y 250)
      (add-edge :epaphus-zeus :epaphus :zeus)
      (add-node :perseus "Perseus" :x 600 :y 150)
      (add-edge :perseus-zeus :perseus :zeus)
      (add-label :father2 "son of" :style {:stroke "crimson"
                                           :font-size "20px"
                                           :font-style "italic"})

      (build)
      (export "/tmp/styles.svg"))

```

The advantage is that you can add and remove nodes dynamically, thus giving a dynamic view about live typologies.

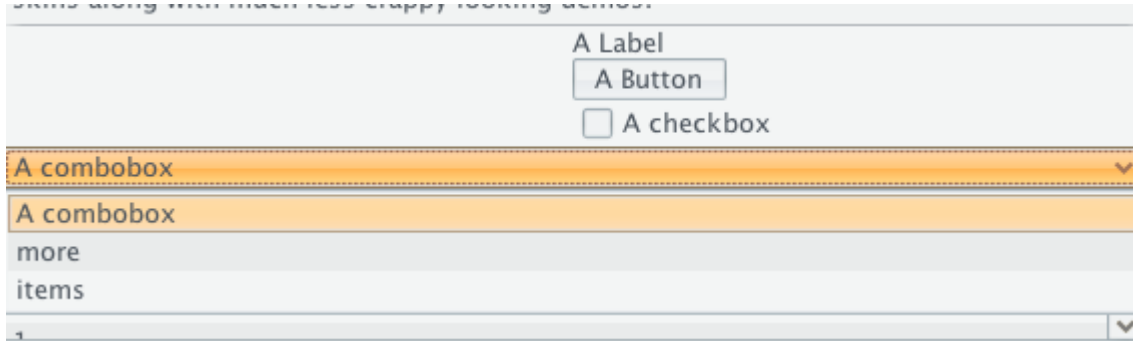
Seesaw

<https://github.com/daveray/seesaw>

Make cool UIs in no time.

The best tutorial I have found so far is [here](#)

And by trying the example, here is a style you can get after a few lines:



clj-xpath

<https://github.com/kyleburton/clj-xpath>

Now you been looking on how to process those xml files as fast as possible, here is a super way to do it, clj-xpath.

```
; import clj
(use 'clj-xpath.core)

; slurp the remote document
(def xdoc (slurp "http://google-web-toolkit.googlecode.com/svn/trunk/samples/expenses/pom.xml"))

; show the top element
($x:tag "/" "*" xdoc)
```

```

; show the developer name
($x:text* "/project/developers/developer/name" xdoc)

; describe all the dependencies
(doseq [node ($x "/project/dependencies/dependency" xdoc)]
  (prn (format "%s %s %s"
    ($x:text "./groupId" node)
    ($x:text "./artifactId" node)
    ($x:text "./version" node))))

```

closure-soup

<https://github.com/mfornos/closure-soup>

This is a wrapper to provide awesome parsing of html files, whether local or remote. Once you have added the library, check out the following example:

```

; start jsoup
(use 'jsoup.soup)

; Get all Emoji names concatenated by single bars from 'emoji-cheat-sheet.com':
($ (get! "http://www.emoji-cheat-sheet.com/")
  "li div:has(span.emoji)" (text)
  (map #(closure.string/replace % ":" ""))
  (closure.string/join "|"))

```

Enlive

<https://github.com/cgrand/enlive>

HTML parser, and Templating framework at the same time, Enlive does a super job of making HTML fun. (yes you read me.)

This is an example taken from a [slick tutorial](#) by David Nolen.

```
(html/deftemplate index "tutorial/template1.html"
  [ctxt]
  [:p#message] (maybe-content (:message ctxt) "Nothing to see here!"))
```

You declare templates in a regular html files, thus your designer can do his Dreamweaver work the way he/she usually does it. Then as a developer you just come and stick content at the location that has been decided. Slick uh ?

Docjure

<https://github.com/ative/docjure>

The best way to have fun with spreadsheet in clojure. Relies on the [Apache POI](#) library but with some clojure sauce so it can actually be eaten.

This is how you write a simple spreadsheet.

```
(use 'dk.ative.docjure.spreadsheet)

;; Create a spreadsheet and save it
```

```

(let [wb (create-workbook "Price List"
      [["Name" "Price"]
       ["Foo Widget" 100]
       ["Bar Widget" 200]])
      sheet (select-sheet "Price List" wb)
      header-row (first (row-seq sheet)))
  (do
    (set-row-style! header-row (create-cell-style! wb {:background :yellow,
                                                         :font {:bold true}}))
    (save-workbook! "spreadsheet.xlsx" wb)))

```

For more examples, I suggest you look at the ["horrible" documentation](#) :)

Postal

<https://github.com/drewr/postal> I am pretty sure you knew how to send a mail before, but look at the awesome way of doing this in Clojure.

```

(use 'postal.core)

(send-message ^{ :host "smtp.gmail.com"
                 :user "hellonico"
                 :pass "tiabtfda28!"
                 :ssl :yes!!!11
               }
  { :from "hellonico@gmail.com"
    :to "hellonico@gmail.com"
    :subject "Hi!"
    :body [

```

```
{:content "<html>新橋</html>" :type "text/html; charset=utf-8"}
{:type :attachment :content (java.io.File. "code/mail.txt")}
{  :type :inline
  :content (java.io.File. "code/mail.txt")
  :content-type "application/text"}
})
```

This is including attachment and Japanese encoding. yey

Korma

<http://sqlkorma.com/docs>

Korma makes actually enjoyable to write SQL queries.

Vaadin

<http://dev.vaadin.com/wiki/Articles/ClojureScripting>

This is not so much a library than a way to develop slick web application using Vaadin, but without the java boiler plate code.

clj-opennlp

<https://github.com/dakrone/clojure-opennlp> Natural Language Processing in Clojure.

[Tokenizer](#) for text analysis.

clj-http

<https://github.com/dakrone/clj-http>

Cascalog

<https://github.com/nathanmarz/cascalog/wiki>

Aleph

<https://github.com/ztellman/aleph>

Lamina

<https://github.com/ztellman/lamina> Event workflow for clojure

Conduit

http://www.intensivesystems.net/tutorials/stream_proc.html Stream processing in Clojure

Chapter 3

Processing, OpenCV and imaging

Quil

<https://github.com/quil/quil>

Quil is to [Processing](#) what Clojure is to Java, some fresh air.

This is how your quil-ed processing sketch now looks like:

```
(ns quilme.core
  (:use quil.core))

(defn setup []
  ;(smooth)                ;;Turn on anti-aliasing
  (frame-rate 5)          ;;Set framerate to 1 FPS
  (background 0))         ;;Set the background colour to
                          ;; a nice shade of grey.

(defn draw []
  (stroke (random 255))    ;;Set the stroke colour to a random grey
  (stroke-weight (random 10)) ;;Set the stroke thickness randomly
  (fill (random 255))      ;;Set the fill colour to a random grey

  (let [diam (random 100)  ;;Set the diameter to a value between 0 and 100
        x    (random (width)) ;;Set the x coord randomly within the sketch
        y    (random (height))] ;;Set the y coord randomly within the sketch
```

```

    (ellipse x y diam diam)))      ;;Draw a circle at x y with the correct diameter

(defsketch example                ;;Define a new sketch named example
  :title "Oh so many grey circles" ;;Set the title of the sketch
  :setup setup                    ;;Specify the setup fn
  :draw draw                      ;;Specify the draw fn
  :render :opengl
  :decor false
  :size [800 600])                ;;You struggle to beat the golden ratio

```

Note the decor set to false, that hides most of the ugliness of the Window borders.

And all the [examples](#) you have ever dreamed from the Generative Art book have been implemented in Clojure/Quil.

- <https://github.com/baguicode/clj-native>
- https://groups.google.com/forum/?fromgroups=#!topic/clojure/WHKlys_SmcU
- <http://im4java.sourceforge.net/>
- <https://github.com/baguicode/clj-native>

Chapter 4

OpenCV, Arduino

Physics with Clojure

- <http://nakkaya.com/2010/07/21/physics-with-clojure/>
- <http://antoniogarrote.wordpress.com/2011/01/30/ocr-with-clojure-tesseract-and-opencv/>

Clodiuno

<https://github.com/nakkaya/clodiuno>

Once you have uploaded the Firmata protocol on your board `File -> Examples -> Firmata -> StandartFirmata`

You can send a Clojure SOS.

```
(ns sos
  (:use :reload-all clodiuno.core)
  (:use :reload-all clodiuno.firmata))

(def short-pulse 250)
(def long-pulse 500)
(def letter-delay 1000)

(def letter-s [0 0 0])
```

```

(def letter-o [1 1 1])

(defn blink [board time]
  (digital-write board 13 HIGH)
  (Thread/sleep time)
  (digital-write board 13 LOW)
  (Thread/sleep time))

(defn blink-letter [board letter]
  (doseq [i letter]
    (if (= i 0)
      (blink board short-pulse)
      (blink board long-pulse)))
  (Thread/sleep letter-delay))

(defn sos []
  (let [board (arduino :firmata "/dev/tty.usbserial-A900adPT")]
    ;;allow arduino to boot
    (Thread/sleep 5000)
    (pin-mode board 13 OUTPUT)

    (doseq [_ (range 3)]
      (blink-letter board letter-s)
      (blink-letter board letter-o)
      (blink-letter board letter-s))

    (close board)))

```

There are so many more great [Samples](#) that Nakkaya has been writing. Make sure you go through them.

Chapter 5

Live Music

Overtone

<https://github.com/overtone/overtone/wiki/Getting-Started>

Because you really need to live audio programming to be a real VJ these days. This is how you install it:

```
(defproject tutorial "1.0"
  :dependencies [ [org.clojure/clojure "1.3.0"]
                  [overtone "0.7.1"] ])
```

Once the library is in your project, type

```
(use 'overtone.live)
```

And now you can define an instrument

```
(definst foo [] (saw 220))
```

And make some sound !

(foo) ; Call the function returned by our synth 4 ; returns a synth ID number (kill 4) ; kill the synth with ID 4 (kill foo) ; or kill all instances of synth foo

Chapter 6

In the Clouds

Pallet

<http://palletops.com/>

Pallet is the mother of them all of cloud infrastructure tool. See the [list of providers](#) it supports! They are actually doing this through [jclouds](#)

Heroku

[Debugging clojure on Heroku](#)

VMFest

<https://github.com/tbatchelli/vmfest>

Easy VirtualBox wrapper for easy cloud management.

```
; you can define your machine this way:
```

```
(def my-machine
```

```
  (instance my-server "my-vmfest-vm"
```

```
    {:uuid "/Users/tbatchelli/imgs/vmfest-Debian-6.0.2.1-64bit-v0.3.vdi"})
```

```
{:memory-size 512
:cpu-count 1
:network [{:attachment-type :host-only
           :host-only-interface "vboxnet0"}
          {:attachment-type :nat}]
:storage [{:name "IDE Controller"
            :bus :ide
            :devices [nil nil {:device-type :dvd} nil]]}
:boot-mount-point ["IDE Controller" 0]]))
```

; then this is the way to start/stop.. any of the virtual machines you have
(start my-machine)
(pause my-machine)
(resume my-machine)
(stop my-machine)
(destroy my-machine)

Make sure you also look at the [playground](#) and have a look at the [tutorial](#)

- http://www.intensivesystems.net/tutorials/stream_proc.html
- <https://github.com/sattvik/Clojure-Android-Examples>

