Learn basics of Clojure/script and Reagent

About me

@matystl – twitter, github

currently working in web development

worked in C++, C#, Java, Dart, JavaScript, Ruby

love learning new things

Clojure

dynamic, LISP-like programing language hosted on JVM

started in 2007 by Rich Hickey

current version 1.7 (June 30 2015)

ClojureScript

compiler for Clojure that targets JavaScript (use Google Closure compiler for production builds)

started in 2012

same syntax and features as Clojure with few unsupported ones (STM, threads)

Why it’s worth to know it

Clojure has great community

different programming paradigm than mainstream OOP

excellent presentations from Rich Hickey and David Nolen

it’s fun

Defining features

LISP syntax
- Macros

Immutability

Functional programing

Concurrency

Native host (Java/Javascript) interoperability

REPL-based

Leiningen

the easiest way to use Clojure

project automation and declarative configuration

dependency management

project generation

Read Eval Print Loop

Clojure repl

- **`lein repl`**

Clojurescript repl

- **`lein new re-frame`** `<project-name>`
  - creates folder with clojurescript project with figwheel, reagent, re-frame prepared
- **`cd <project-name> && lein figwheel dev`** – download dependencies and run webserver
- http://localhost:3449 – open in browser and in console appear appropriate clojurescript repl
- write `(in-ns 'folder.core)` to get to correct namespace
- on linux use `rlwrap` and put it before lein to have proper console like history and cursor movement

For Light Table users

- Light table is build in clojure so you can evaluate forms inside of it in it's clojure environment
- create folder with some file in it

```
(ns folder.file)
(+ 1 2)
```

- place cursor inside (+ 1| 2) hit `Ctrl + Enter` to evaluate it and see result instantly (on first try wait to connect to repl)
- place cursor on some symbol (|+ 1 2) and hit `Ctrl + D` to see documentation
- `Ctrl + space` open quick search

Syntax

numbers - `47`

ratios - `22/7` (only in Clojure not in ClojureScript)

strings – `"hello“, "world"`

characters - `\a \b \c` (as strings in JS)

symbols - `foo, bar`

keywords - `:a, :bar`

Booleans – `true/false`

null - `nil`

Syntax - collections

Lists – `(1 2 3 4)`

Vectors – `[1 2 3 4]`

Maps – `{:a 1 "b" 2}` or `{:a 1, "b" 2}`

Sets - `#{1 2 :a "hello"}`

Syntax

That’s it. No other syntax is needed(Few shortcuts exists).

Data structures are code.

Homoiconic

Data literals stand for themselves except:

- Lists
- Symbols

Semantics

Usually list are evaluated(in REPL, your code)

First element is function to be called and rest is arguments passed to that function. Arguments are evaluated before passing to function (except for macros).

```
(+ 1 2) ; => 3
```

```
(add 1 (add 2 3)); add(1, add(2,3))
```

for suppressing evaluation add `'` - `'(+ 2 3)` this will be list also in repl and code

Documentation

```
(doc +)
-------------------------
cljs.core/+
[x]
  Returns the sum of nums. (+) returns 0.
(find-doc "trim")
-------------------------
subvec
[v start end]
  Returns ....
-------------------------
trim-v
([handler])
  Middleware ...
```

Hello world

```
(ns folder.filename)

(defn hello
  "An example function – documentation string"
  [argument]
  (println "Hello" argument))

(hello "World!")
```

Basics

```
(def simple-variable 5)
(fn [arg1 arg2] (+ arg1 arg2))
(def one (fn [] 1))
(defn add [arg1 arg2] (+ arg1 arg2))
(defn add-multiple-arity
  ([] 0)
  ([a] (add a 0))
  ([a b] (+ a b)))
#(+ 10 %) ;=> (fn [%] (+ 10 %))
```

Variables inside function - let

```
(defn message [a b]
  (let [c (+ a b)
        d (* c 15)]
    (/ d 12)))

; function message(a, b) {
;  const c = a + b;
;  const d = c * 15;
;  return (d/12);
;}
```

Collection operation – vector, list, map, set

Collections are immutable, operation return new instances

```
get, count, vec, conj, first, rest, peek, pop,
into, nth, assoc, dissoc, contains?, disj
```

http://clojure.org/cheatsheet

Some example

```
(conj [1 2 3] 5) ; => [1 2 3 5]

(conj '(1 2 3) 5) ; => (5 1 2 3)

(assoc {} :a 5) ; => {:a 5}

(dissoc {:a 5} :a) ; => {}

(conj #{} :a) ; => #{:a}

(disj #{:a} :a) ; => #{}
```

Map access

```
(def m {:a 5 "b" 7})

(get m :a) ; => 5
(m :a) ; map is function from key to value
(:a m) ; keyword can retrieve itself from map and set

(get m "b") ; => 7
(m "b") ; => 7
("b" m) ; error
```

Map manipulation

```
(update m key func a2 a3 ...)
```

call (func (get key m) a2 a3 ...) and update m under key with new value and return it

```
(update {:counter 0} :counter + 2) ; => {:counter 2}
```

for nested maps/vectors you can use update-in which take key path

```
(update-in m [key1 key2 ...] func a2 a3)
(update-in {:a {:b 5}} [:a :b] inc) ; => {:a {:b 6}}
```

Sequences

Not a data structure – source of values in some order

can be lazy(infinite)

most sequence functions return sequence

```
(range 3) ; => (0 1 2)

(seq {:a 5 :b 10}) ; => ([:a 5] [:b 10])

(seq []) ; => nil
```

Sequences - operations

```
(map inc [1 2 3]) ; => (2 3 4)

(map + [1 2 3] [20 40 60]) ; => (21 42 63)

(take 2 [1 2 3 4]) ; => (1 2)
```

```
take drop take-while drop-while filter remove partition
group-by sort shuffle reverse mapcat flatten concat
```

```
repeat repeatedly cycle interpose interleave iterate
```

apply

```
(apply func arg-seq)

(+ 1 2 3) ; => 6

(+ [1 2 3]) ; => [1 2 3]

(apply + [1 2 3]); => 6
```

Sequence - results

```
(vec (filter even? (range 10)))

(set (map inc (range 10)))

(apply hash-map (range 10))

(apply str (interpose \, (range 4)))


(into {} [[:x 1] [:y 2]])
```

Sequence – results 2

```
(reduce func init coll)
(some func coll)
(every? func coll)
```

remember laziness is hell for side-effects so put them at end

```
(take 4 (map println (range 100)))
```

`(doseq [i (range 5)] (println i))` – iteration for side effects

Flow control

everything is expression and return value

expression for side-effects return nil

```
(if test then else?)
(do exp1 exp2 ...)
(when test exp1 exp2 ...)
(cond test1 exp1 test2 exp2 ...)
(case test-val val1 exp1 val2 exp2 ...)
```

Flow control – imperative loop

```
(loop [i 0 j 0]
  (println i i)
  (if (< i 10)
    (recur (inc i) (+ 2 b))))
```

Destructuring

can be used in function declaration, let binding and other bindings

```
(defn f [a b & rest] res)
(f 1 2 3 4) ; => (3 4)
[a b & rest :as whole-list]

(defn f [{the-a :a the-b :b}] '(:result the-a the-b))
(f {:a 5 :b 7}) ; => '(:result 5 7)
```

Destructuring - 2

```
{a :a :as whole} ; same as in vector
```

works recursively

```
(let [[[x1 y1] [x2 y2]] [[1 2] [4 5]]]
  [x1 x2 y1 y2]) ; => [1 4 2 5]
{{c :c :as inner} :a} ; => {:a {:c 10}}
```

if name will be same as keyword you can use this helper

```
(let [{:keys [x y]} {:x 1 :y 2}]
```

Identity, state and values

clojure has multiple explicit representation for identity

identity - a stable logical entity associated with a series of different values over time

state is value of identity in particular time

Identity, state and values - atom

atom is identy which can hold changing value in time

```
(def a (atom 0))
```

read value with

```
(deref a) or @a
```

set value with swap!

```
(swap! atom f a2? a3?)
```

can be shared between threads and swap is atomic operation

can be observed for changes

Identity, state and values - other

other primitives are refs, vars, agent

refs are interesting because they implement transactions in memory(STM) between multiple threads

not supported in ClojureScript

What now?

Continue with reagent and re-frame or

you are eagerly waiting to code?

Namespaces

every file has own namespace matching structure folder.file defined by macro `ns`

require import other functionality into current namespace

usage of items from required namespace with

- name-of-imported-namespace/what-i-want-to-use

```
(ns reagent-tutorial.core
  (:require [clojure.string :as string]
            [reagent.core :as r]))
(def s (r/atom 0))
```

Javascript interoperability

use js/ prefix to access js global namespace

- `js/document js/window js/console js/date`

for data structure conversion there are helpers

- `js->clj clj->js`

functions from clojure are javascript functions

Javascript interoperability - 2

invoking js function from clojure

```
(.hello someObject a1 a2) ; someObject.hello(a1, a2);
```

accessing property

```
(.-name someObject) ; someObject.name
```

setting property

```
(set! (.-name someObject) value)
; someObject.name = value;
```

Reagent

simple ClojureScript interface to React

building blogs are functions, data, atoms

uses Hiccup-like markup no jsx

Reagent – simple component

```
(defn some-component []
  [:div
   [:h3 "I am a component!"]
   [:p.someclass
    [:span {:style {:color "red"}} " Red color "]
    " text."]])
```

Reagent – render into dom

```
(ns example
  (:require [reagent.core :as r]))

(r/render-component [some-component]
  (.-body js/document))
```

Reagent – use of other component

```
(defn child [name]
  [:p "Hi, I am " name])

(defn childcaller []
  [child "Foo Bar"])

(defn childcaller []
  (child "Foo Bar"))
```

Reagent – usage of atoms

```
(def counter (r/atom 0))


(defn comp []
  [:div
   "Value of counter is:" @counter
   [:div {:on-click #(swap! counter inc)} "inc"])
```

Reagent – local state

local let to create atom with state and return rendering function

```
(defn test3 []
  (let [c (reagent/atom 0)]
    (fn []
      [:div
        "Value of counter " @c
        [:button {:on-click #(swap! c inc)} "inc"]
      ])))
```

Re-frame

Reagent is only V so for building application you need more

Re-frame is library and more importantly pattern how to develop complicated SPA with Reagent

implementation is 200LOC description 800LOC

https://github.com/Day8/re-frame

if you want to try it read description and try it

Thanks!

QUESTIONS?