Full Stack Clojure

Michiel Borkent
@borkdude
HAN University of Applied Sciences
October 1st 2015

Agenda

- Part 1: Why Clojure
- Part 2: Some basic Clojure
- Part 3: Full stack Clojure

Part 1: Why Clojure

Clojure

- Designed by Rich Hickey in 2007
- Frustration with Java, C++
- Deliver the same functionality faster
- Without giving up operational requirements

Non-goals

- Easy to learn by Java etc. developers
- Experiment in language design

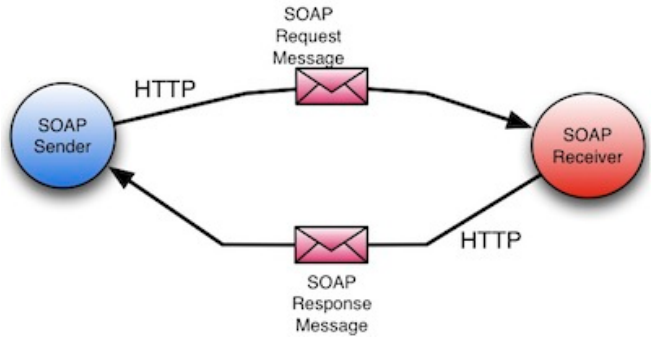
Programs are (mostly) about data

- Language should not get in the way of data
- Good support for data literals
- Data transformations
- Data is immutable
- OO is not great for working with data
- Big part of program can be built using plain data

Systems and immutability

- Each system receives a message and/or sends a message
- Mutating a message does not affect other system
- In Java, references lead to uncontrolled mutation
- You can protect yourself by using Value Objects or DTOs, but takes work

- Clojure adopts system model in the small by using immutable data structures
- Mutation only happens in controlled and explicit ways



Clojure

- dynamic language
- lisp
- functional programming
- immutable data structures
- strong concurrency support
- embraces host platform (JVM, js)
- EDN

Choosing a language is all about trade offs!

Clojure in industry

https://www.youtube.com/watch?v=av9Xi6CNqq4

http://dev.clojure.org/display/community/Clojure+Success+Stories
http://clojure.org/Companies

Part 2: Basic Clojure

Data literals

```
Symbol:       :a
Vector:       [1 2 3 4]
Hash map:     {:a 1, :b 2}
Set:          #{1 2 3 4}
List:         '(1 2 3 4)
```

Extensible Data Notation

```
{:key1"Bar"
 :key2 [1 2 3]
 "key3", #{1.0 2.0 \c}
 :key4,{:foo {:bar {:baz 'hello}}}}

(pr-str {:foo "bar"})
(read-string "{:foo \"bar\"}")
```

Syntax

```
f(x) -> (f x)
```

Syntax

```
if (...) {
  ...
} else {
  ...
}
```

->

```
(if ...
    ...
    ...)
```

Syntax

```
var foo = "bar";
```

```
(def foo "bar")
```

JavaScript - ClojureScript

```
if (bugs.length > 0) {
  return 'Not ready for release';
} else {
  return 'Ready for release';
}
```

```
(if (pos? (count bugs))
  "Not ready for release"
  "Ready for release")
```

source: http://himera.herokuapp.com/synonym.html

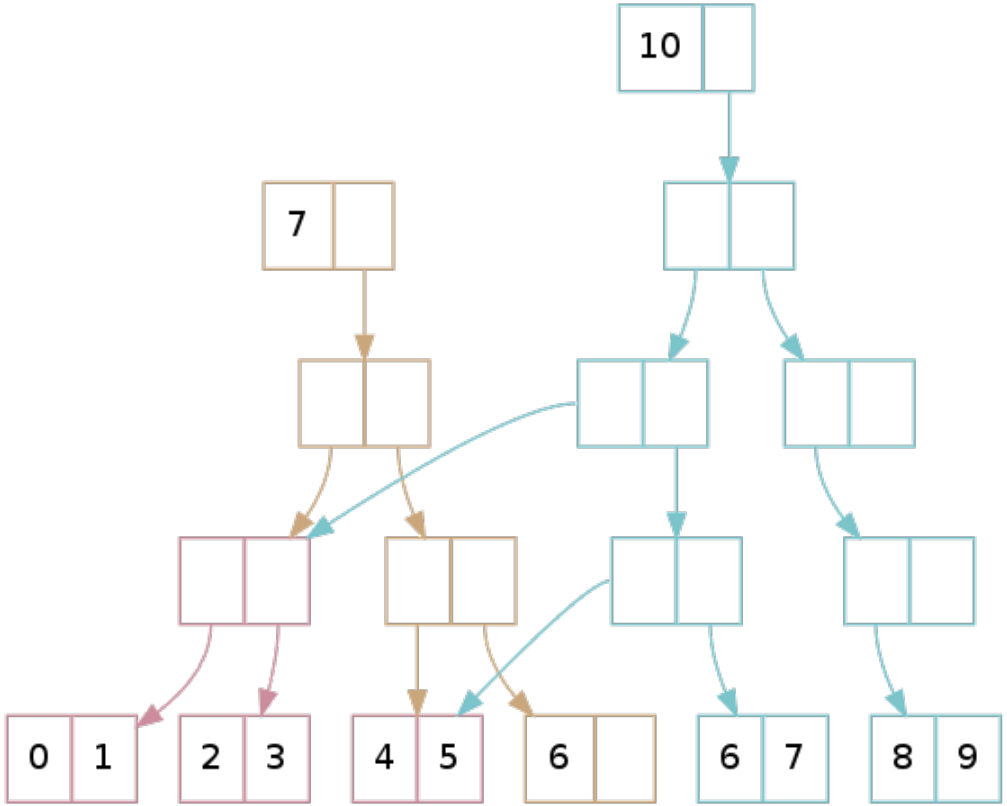
JavaScript - ClojureScript

`var foo = {bar: "baz"};`
`foo.bar = "baz";`
`foo["abc"] = 17;`
 	
`alert('foo')`
`new Date().getTime()`
`new`
`Date().getTime().toString()`
	`(def foo (js-obj "bar" "baz"))`
	`(set! (.-bar foo) "baz")`
	`(aset foo "abc" 17)`
	
	`(js/alert "foo")`
	`(.getTime (js/Date.))`
	`(.. (js/Date.) (getTime)`
	`(toString))`

source: http://himera.herokuapp.com/synonym.html

Persistent data structures

```
(def v [1 2 3])
(conj v 4) ;; => [1 2 3 4]
(get v 0) ;; => 1
(v 0) ;; => 1
```



source: http://hypirion.com/musings/understanding-persistent-vector-pt-1

Persistent data structures

```
(def m {:foo 1 :bar 2})
(assoc m :foo 2) ;; => {:foo 2 :bar 2}
(get m :foo) ;;=> 1
(m :foo) ;;=> 1
(:foo m) ;;=> 1
(dissoc m :foo) ;;=> {:bar 2}
```

Functional programming

```
(def r (->>
         (range 10)     ;; (0 1 2 .. 9)
         (filter odd?) ;; (1 3 5 7 9)
         (map inc)))    ;; (2 4 6 8 10)
;; r is (2 4 6 8 10)
```

Functional programming

```
;; r is (2 4 6 8 10)
(reduce + r)
;; => 30
(reductions + r)
;; => (2 6 12 20 30)
```

```
var sum = _.reduce(r, function(memo, num){ return memo + num; });
```

Sequence abstraction

Data structures as seqs

```
(first [1 2 3]) ;;=> 1
(rest [1 2 3]) ;;=> (2 3)
```

General seq functions: `map`, `reduce`, `filter`, …

```
(distinct [1 1 2 3]) ;;=> (1 2 3)
(take 2 (range 10)) ;;=> (0 1)
```

See http://clojure.org/cheatsheet for more

Sequence abstraction

Most seq functions return lazy sequences:

```
(take 2 (map
          (fn [n] (js/alert n) n)
            (range)))
```

side effect

infinite lazy sequence of numbers

Mutable state: atoms

```
(def my-atom (atom 0))
@my-atom ;; 0
(reset! my-atom 1)
(reset! my-atom (inc @my-atom)) ;; bad idiom
(swap! my-atom (fn [old-value]
                 (inc old-value)))
(swap! my-atom inc) ;; same
@my-atom ;; 4
```

Lisp: macros

```
(map inc
  (filter odd?
    (range 10)))

(->>
  (range 10)
  (filter odd?)
  (map inc))
```

thread last macro

Lisp: macros

```
(macroexpand
  '(->> (range 10) (filter odd?)))

;; => (filter odd? (range 10))

(macroexpand
  '(->> (range 10) (filter odd?) (map inc)))

;; => (map inc (filter odd? (range 10)))
```

Lisp: macros

JVM Clojure:

```
(defmacro defonce [x init]
 `(when-not (exists? ~x)
    (def ~x ~init)))
```

ClojureScript:

```
(defonce foo 1)
(defonce foo 2) ;; no effect
```

notes:

- macros must be written in JVM Clojure
- are expanded at compile time
- generated code gets executes in ClojureScript

Part 3: Full Stack Clojure

Full Stack Clojure

<table>
<tr><td>Front-end (js)
ClojureScript</td><td>reagent + react
secretary
cljs-http
figwheel</td><td rowspan="2">REPL
leiningen
core.async
Prismatic/Schema
timbre</td></tr>
<tr><td>Server side (JVM)
Clojure</td><td>ring
compojure
liberator
environ
hiccup

Framework: Pedestal</td></tr>
<tr><td>Persistence</td><td colspan="2">RDBMS (via JDBC, e.g. PostgresQL, MySQL, H2, etc)
clojure.java.jdbc
YesQL

Datomic</td></tr>
</table>

Show me the code!

https://github.com/borkdude/full-stack-clojure-han-okt-2015

Get started with Clojure(Script)

- Read a Clojure(Script) book
- Do the 4clojure exercises
- Start hacking on your own project
- Pick an online Clojure course
- Join the AMSCLJ meetup
- Join the Slack community