ClojureScript

as a compilation target to JS

Michiel Borkent @borkdude
Vijay Kiran @vijaykiran
FP AMS October 16th 2014

This work is licensed under a Creative Commons Attribution 4.0 International License.

Agenda

- History and Rationale of ClojureScript
- ClojureScript: advantages over JS
- Syntax compared
- React + ClojureScript
- Om
- Reagent

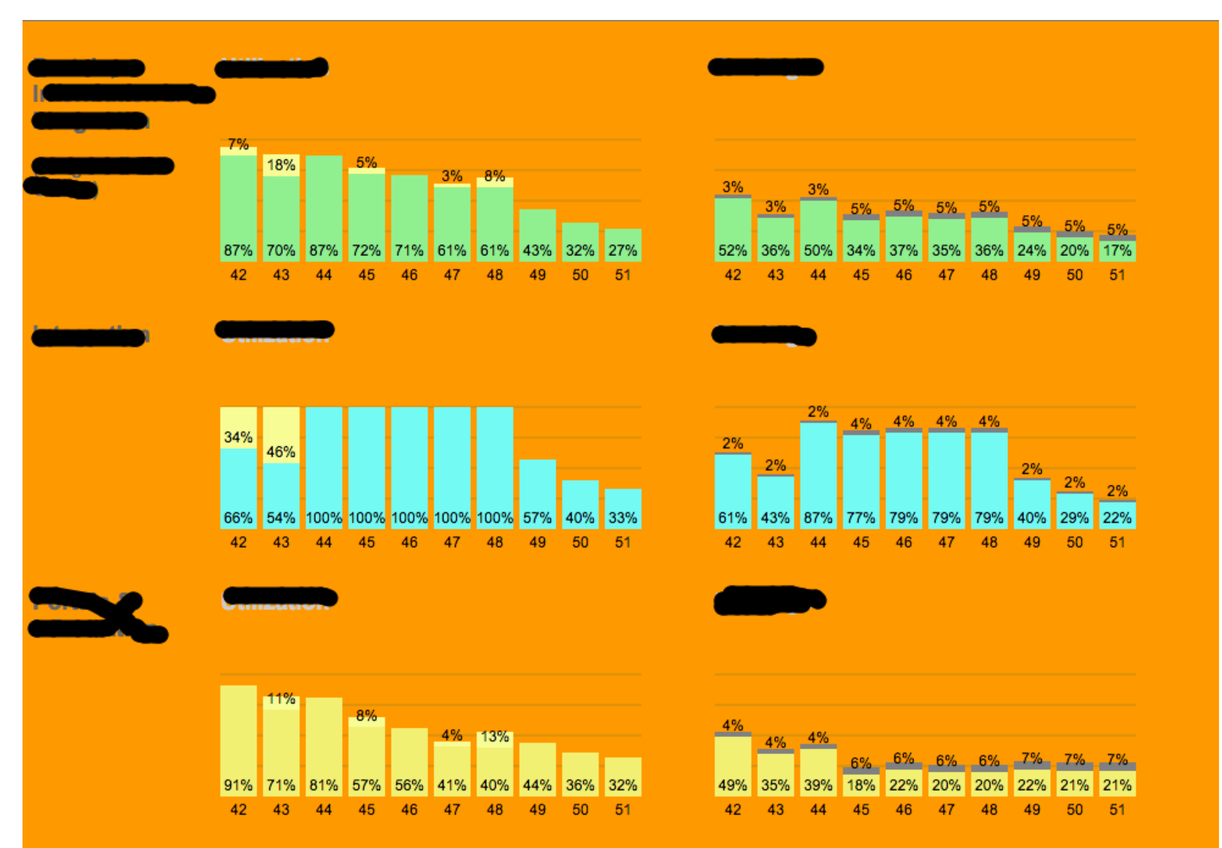
Introduction

Michiel

Vijay

Full Clojure stack examples @ Finalist

- Clojure + Liberator + Datomic backend
- ClojureScript + Om frontend
- Plain SVG graphs, home made, no JS libs used
- Integrates multiple systems (resource planner, Salesforce, billing system, etc)
- Runs on Immutant. Uses Immutant job scheduling for refreshing results



Typical in-house "ugly" app. Very light weight, quickly programmed, quick results. Useful information during meetings.

Full Clojure stack examples @ Finalist

Same stack. Real commercial app.

Fairly complex UI

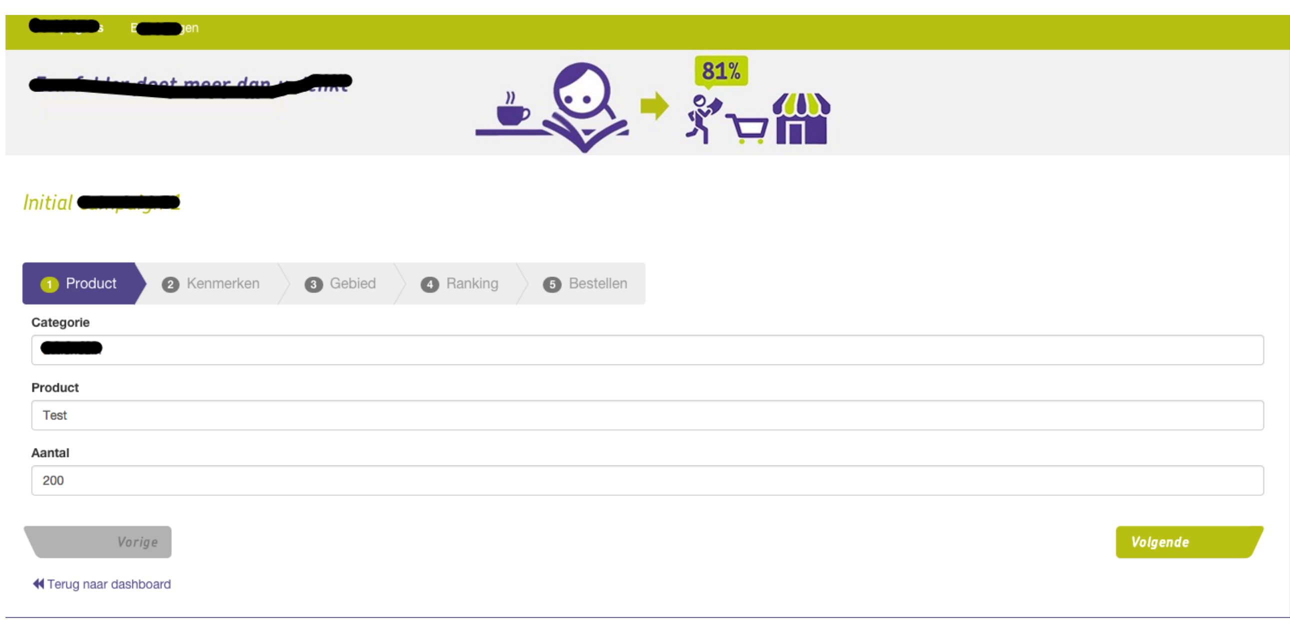
- Menu: 2 "pages"

Page 1:

Dashboard. Create new or select existing entity to work on.

Then:

- Wizard 1
  - Step 1..5
  - Each step has component
- Wizard 1 - Step2
  - Wizard 2
    - Step 1'
    - Step 2'

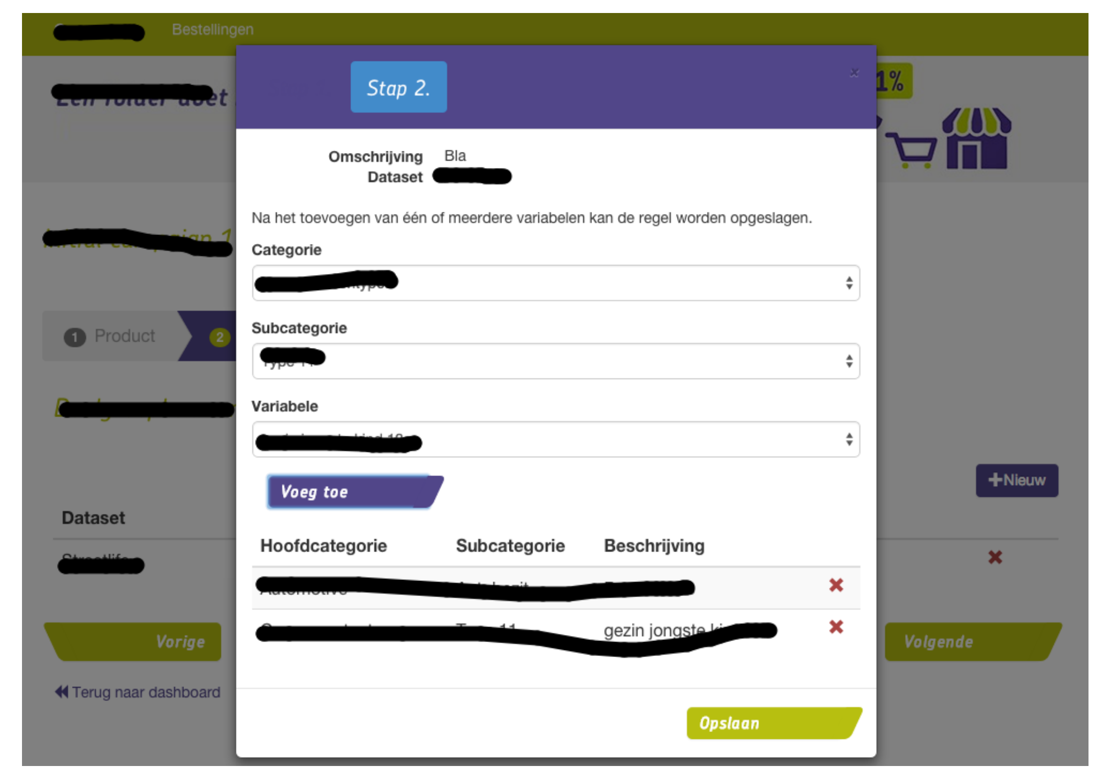


Full Clojure stack examples @ Finalist

Step 2 of inner wizard:

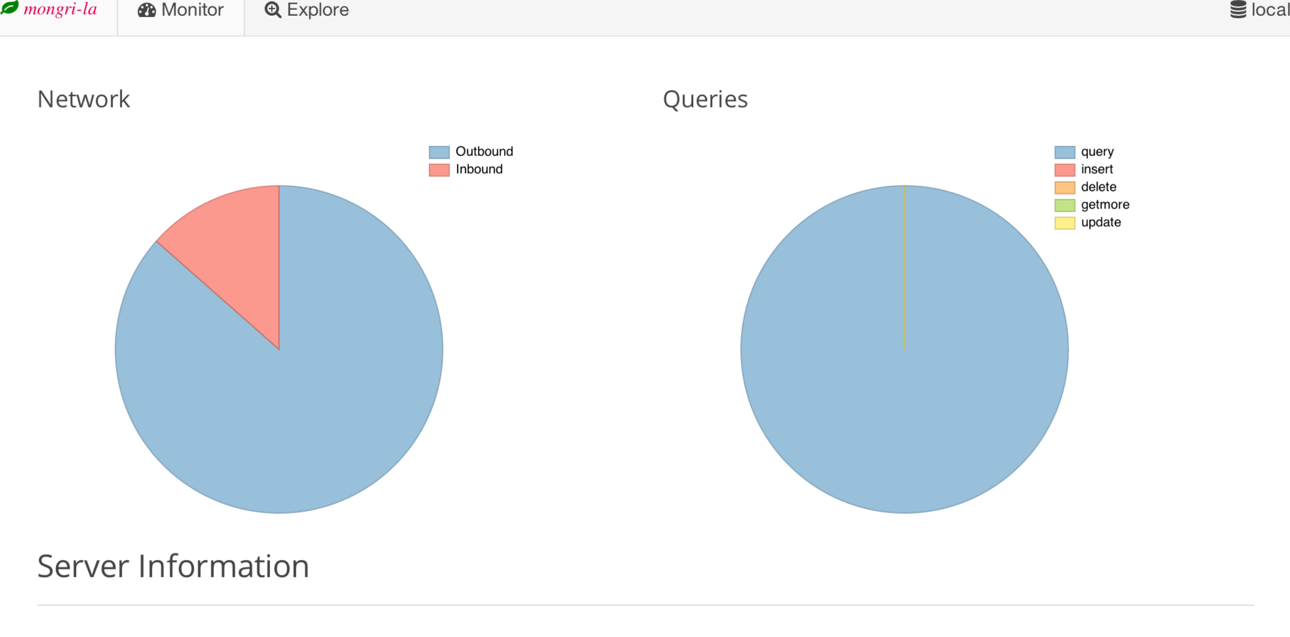
- Three dependent dropdowns + backing ajax calls
- Crud table of added items + option to remove
- When done: create something based on all of this on server and reload entire "model" based on what server says

Because of React + Om we didn't have to think about updating DOM performantly or keeping "model" up to date.



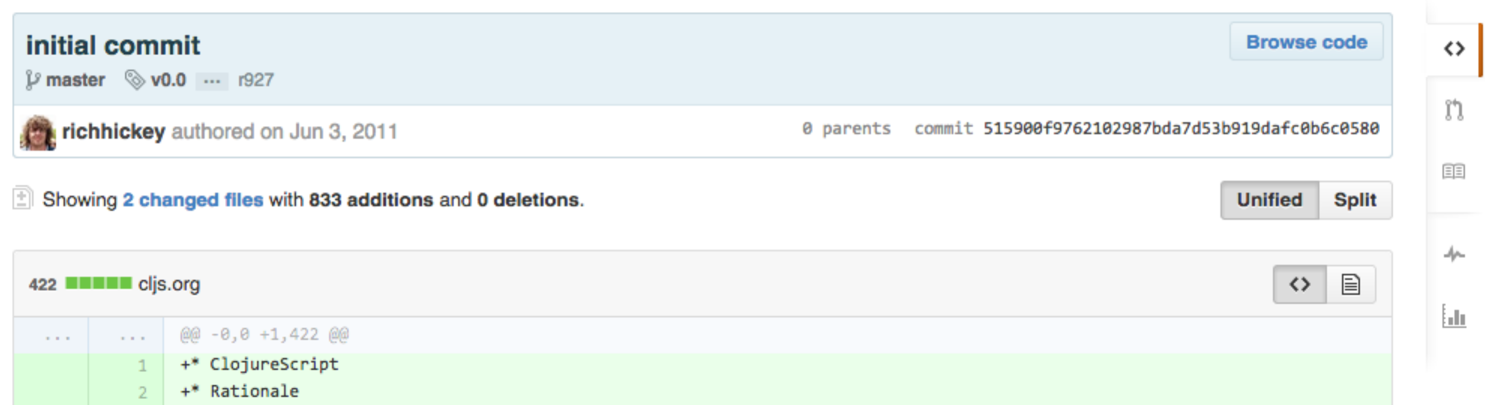
ClojureCup Entry

- Clojure Backend
- Om Front-End



Brief history of ClojureScript

June 20th 2011: first release of ClojureScript



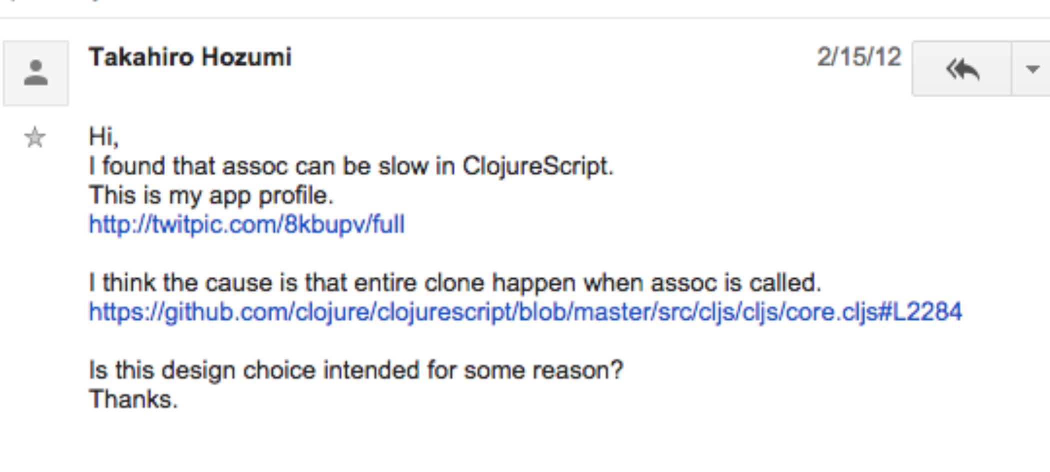
Brief history of ClojureScript

Early 2012: first release of lein cljsbuild

Leiningen plugin to make ClojureScript development easy

```
:cljsbuild {:builds [{:id "dev"
                      :source-paths ["src"]
                      :compiler {:output-to "resources/public/main.js"
                                 :output-dir "resources/public/out"
                                 :optimizations :none
                                 :source-map true}}]}
```

Brief history of ClojureScript



Takahiro Hozumi 2/15/12

Hi,
I found that assoc can be slow in ClojureScript.
This is my app profile.
http://twitpic.com/8kbupv/full

I think the cause is that entire clone happen when assoc is called.
https://github.com/clojure/clojurescript/blob/master/src/cljs/cljs/core.cljs#L2284

Is this design choice intended for some reason?
Thanks.

David Nolen 2/15/12

It is intended, copy-on-write. No one has yet attempted persistent data structures for ClojureScript.

Until then I think transient versions of the current data structures might be useful if someone is willing to contribute them.

David

Brief history of ClojureScript

April 2012:

persistent data structures were ported

Light Table

June 2012

Funded as Kickstarter Project

Interactive, experimental IDE written in ClojureScript, running on Node Webkit

Became open source early 2014

```
(defn move [me]
  (let [speed 5
        vx (cond
             (input? :left) (- speed)
             (input? :right) speed
             :else 0)
        moved (update-in me [:x] + vx)]
    (if (zero? vx)
      me
      (if-let [block (colliding? moved)]
        (let [block-edge (if (< vx 0)
                           (+ (:x block) (:w block) (:r me))
                           (- (:x block) (:r me)))]

          (assoc me :x block-edge))
        moved))))
```

```
(defn gravity [{:keys [vy y] :as me}]
  (let [g 0.5
        vy (or vy 0)
        neue-vy (+ vy g)
        dir (if (< neue-vy 0) :up :down)
        moved (update-in me [:y] + neue-vy)
    (if-let [block (colliding? moved)]
      (let [block-edge (if (= dir :up)
                         (+ (:y block) (:h
                         (- (:y block) (:r
        (assoc me :y block-edge
               :jumping (= dir :up)
               :vy 0))
      (-> moved
        (assoc :vy neue-vy)))))
```

7,317
backers

$316,720
pledged of $200,000 goal

0
seconds to go

Project by

Chris Granger
San Francisco, CA

First created · **4 backed**

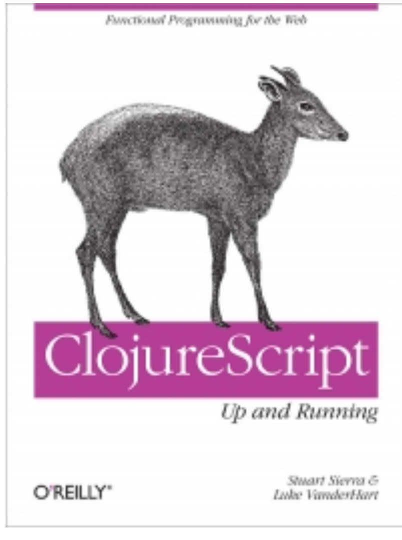
Chris Granger 189 friends

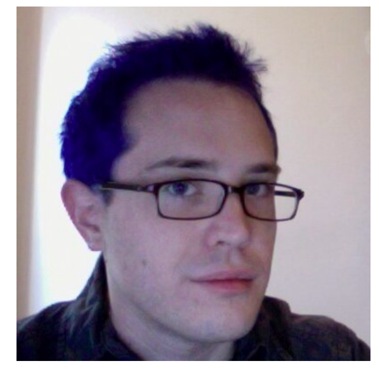
chris-granger.com

See full bio | Contact me

Brief history of ClojureScript

October 2012: ClojureScript Up and Running - O'Reilly





Brief history of ClojureScript

June 2013: core.async was announced

Clojure core.async Channels

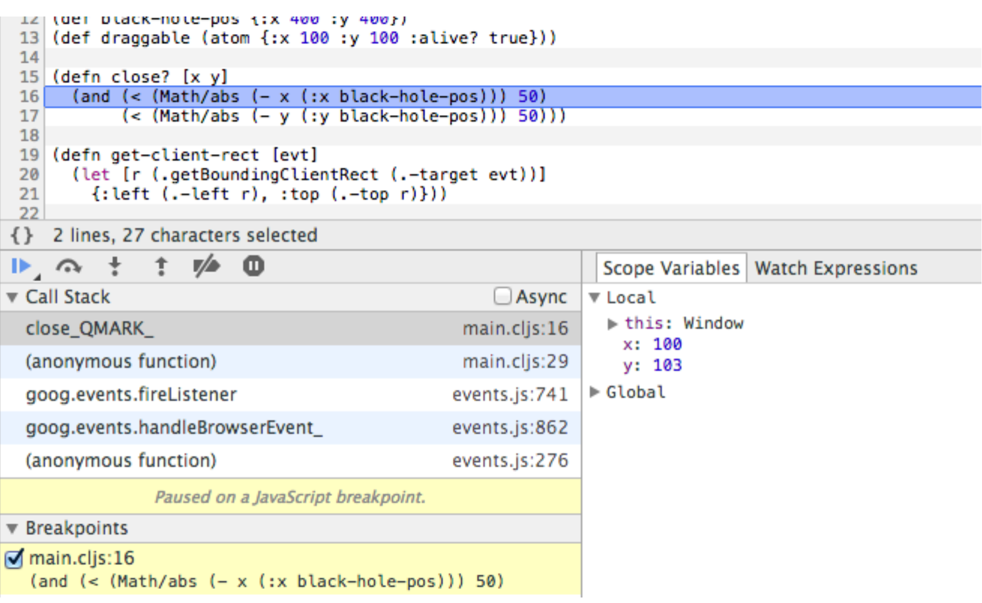
Posted by Rich Hickey on Jun 28, 2013

It is our hope that async channels will greatly simplify efficient server-side Clojure programs, and offer simpler and more robust techniques for front-end programming in ClojureScript.

Brief history of ClojureScript

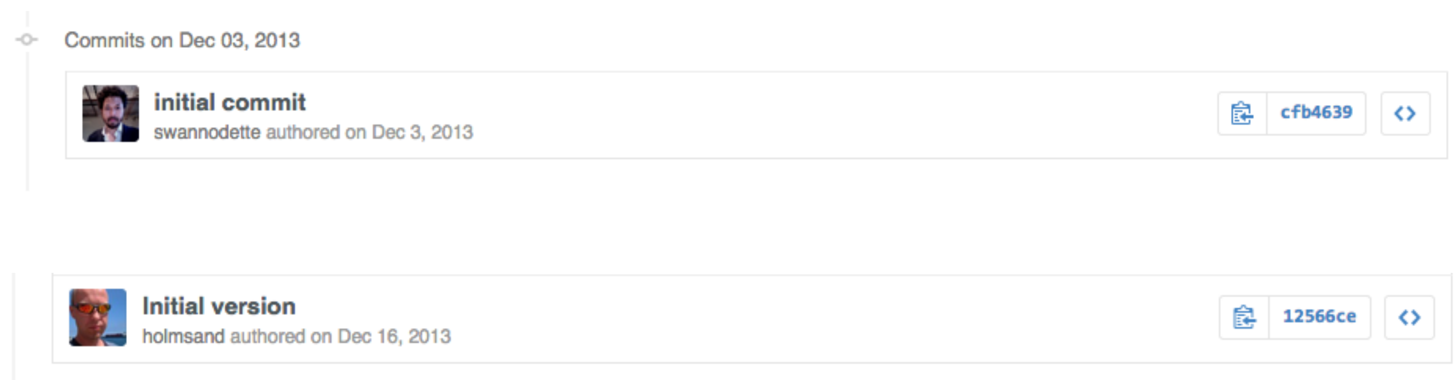
September 2013: source maps

Lets you debug ClojureScript directly from the browser.



Brief history of ClojureScript

December 2013: ClojureScript interfaces to React

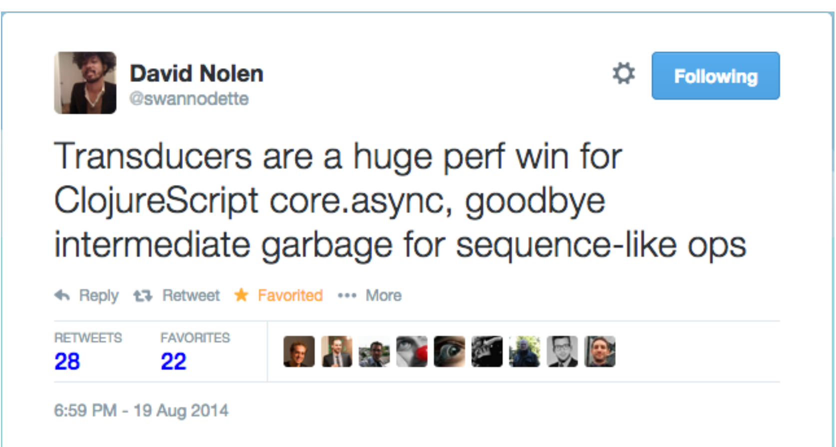


Brief history of ClojureScript

August 2014

TRANSDUCERS ARE COMING

Posted by Rich Hickey on August 6, 2014



ClojureScript: rationale

- JavaScript is ubiquitous, but not a robust and concise language

  Requires a lot of discipline to only use "the good parts"
- JavaScript is taking over in the browser: UI logic from server to client
- JavaScript is not going away in the near future
- Advanced libraries and technologies exist to optimize JavaScript:

  Google Closure
- Clojure is a robust and concise language
- ClojureScript targets JavaScript by adopting Google Closure's strategy
- Brings Clojure goodness to JavaScript environments
- Clojure is designed to play well with host (does not aim to be cross platform compatible)

Advantages over JavaScript

- less cognitive load for Clojure programmers
- less wat
- functional programming
- immutable/persistent data structures
- namespaces
- destructuring
- macros - code as data

Advantages over JavaScript

- EDN vs JSON
- core.async - solves callback hell
- sequence abstraction: many composable functions on whatever data structure that implements ISeq
- transducers: algorithm decoupled from concrete sequential data structures and/or channels
- core.typed
- able to share code across client/server (cljx)

JavaScript - ClojureScript

JavaScript	ClojureScript
console.log("Hello, world!");	(.log js/console "Hello, world!") or (println "Hello, world!")
no implementation	(ns my.library (:require [other.library :as other]))
var foo = "bar";	(def foo "bar")

function foo() {
  var bar = 1;
}
	(defn foo []
	  (let [bar 1]))

// In JavaScript locals are mutable
 	
function foo(x) {
  x = "bar";
}
	;; this will issue an error
	
	(defn foo [x]
	  (set! x "bar"))

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

JavaScript - ClojureScript

`No implementation`	`(def v (vector))`
	`(def v [])`
	`(def v [1 2 3])`
	`(conj v 4) ;; => [1 2 3 4]`
	`(get v 0) ;; => 1`
	`(v 0) ;; => 1`

`No implementation`	`(def s (set))`
	`(def s #{})`
	`(def s #{"cat" "bird" "dog"})`
	`(conj s "cat") ;; => #{"cat" "bird" "dog"}`
	`(contains? s "cat") ;; true`
	`(s "cat") ;; "cat"`
	`(s "fish") ;; nil`

`No implementation`	`(def m (hash-map))`
	`(def m {})`
	`(def m {:foo 1 :bar 2})`
	`(conj m [:baz 3]) ;; => {:foo 1 :bar 2 :baz 3}`
	`(assoc m :foo 2) ;; => {:foo 2 :bar 2}`
	`(get m :foo) ;;= > 2`
	`(m :foo) ;;= > 2`

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

JavaScript - ClojureScript

<table>
<tr>
<td>

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

</td>
<td>

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

</td>
</tr>
<tr>
<td>

```
function foo() {
  var bar = 1;
  var baz = 2;
  return bar + baz;
}
foo(); // 3
```

</td>
<td>

```
(defn foo []
  (let [bar 1
        baz 2]
    (+ bar baz))
(foo) ;; => 3
```

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

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

core.async + transducer

without transducer: creates intermediate hash-map of response

```
(go (let [body (:body (<! (http/get "/is-dev")))]
      (when (= body true) ;; has to match exactly true and not some string
```

with transducer: no need for intermediate hash-map

```
;; reload
(go (let [body (<! (http/get "/is-dev"
                             {:channel
                              (chan 1 (map :body))}))]
     (when (= body true) ;; has to match exactly true and not some string
```

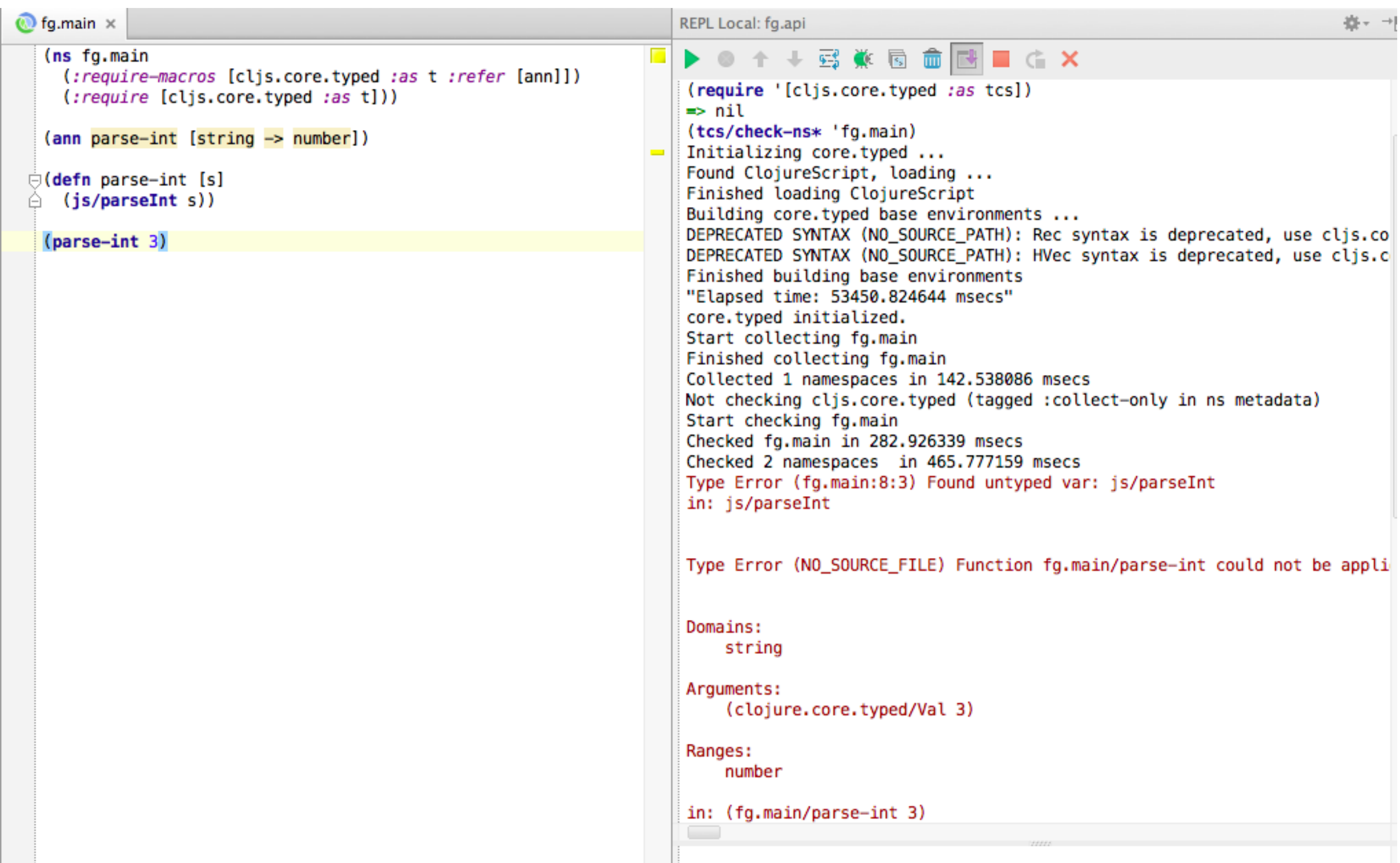
core.typed (JVM)

```
(ann f [String -> int])
(defn f [s]
  (.length ^String s))
```

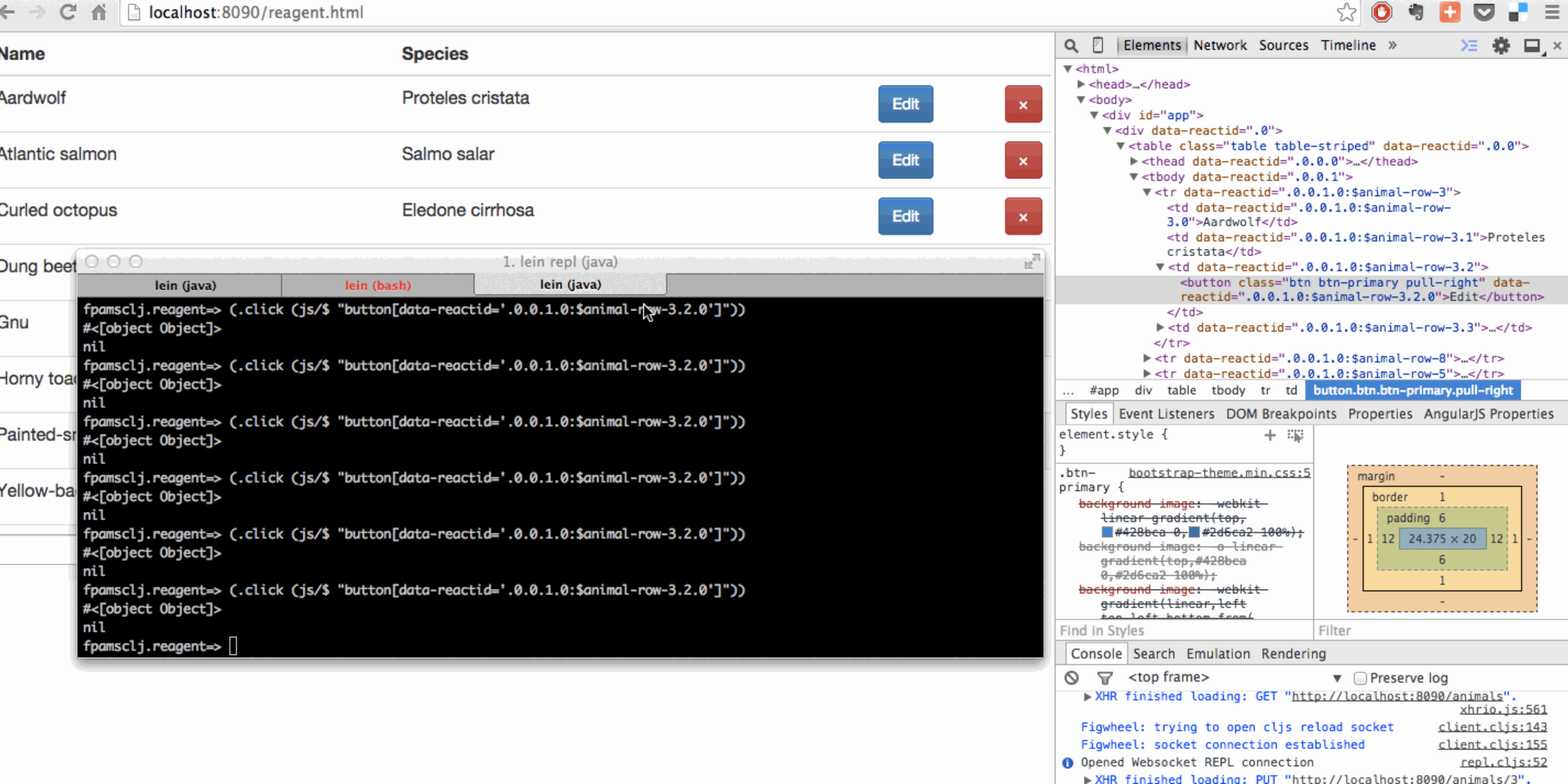
```
Function f could not be applied to arguments:


Domains:
        java.lang.String

Arguments:
        (t/U String nil)

Ranges:
        int

with expected type:
        t/Any

in: (f (java.lang.System/getProperty "foo"))
```

```
  (f (System/getProperty "foo")))
```

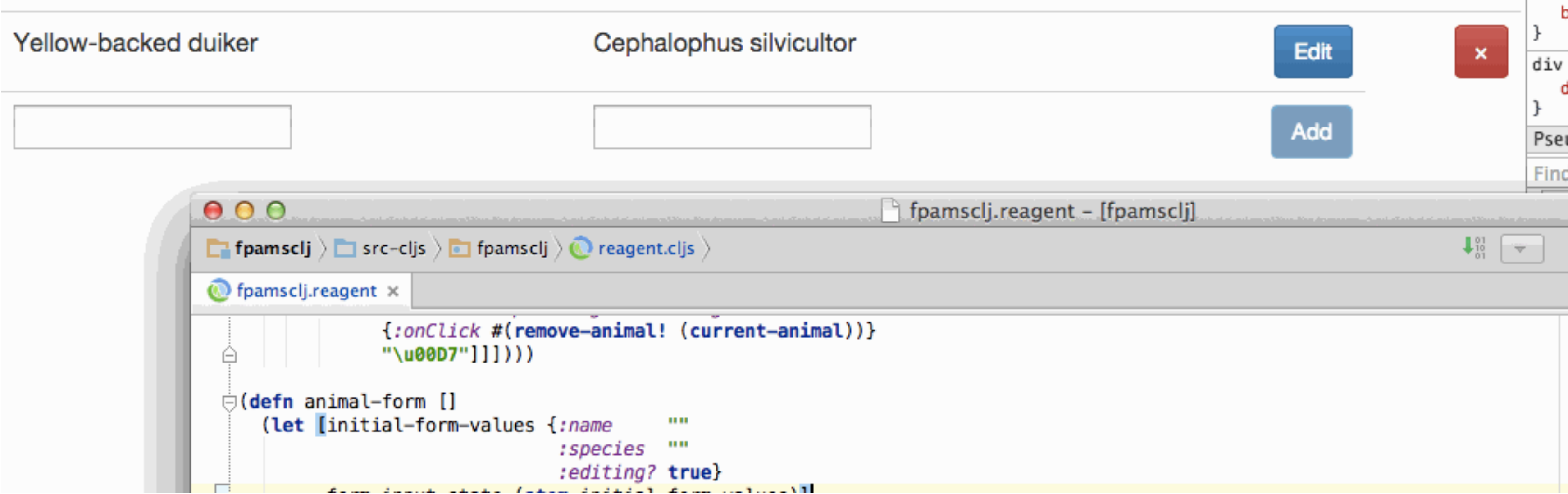
cljs.core.typed

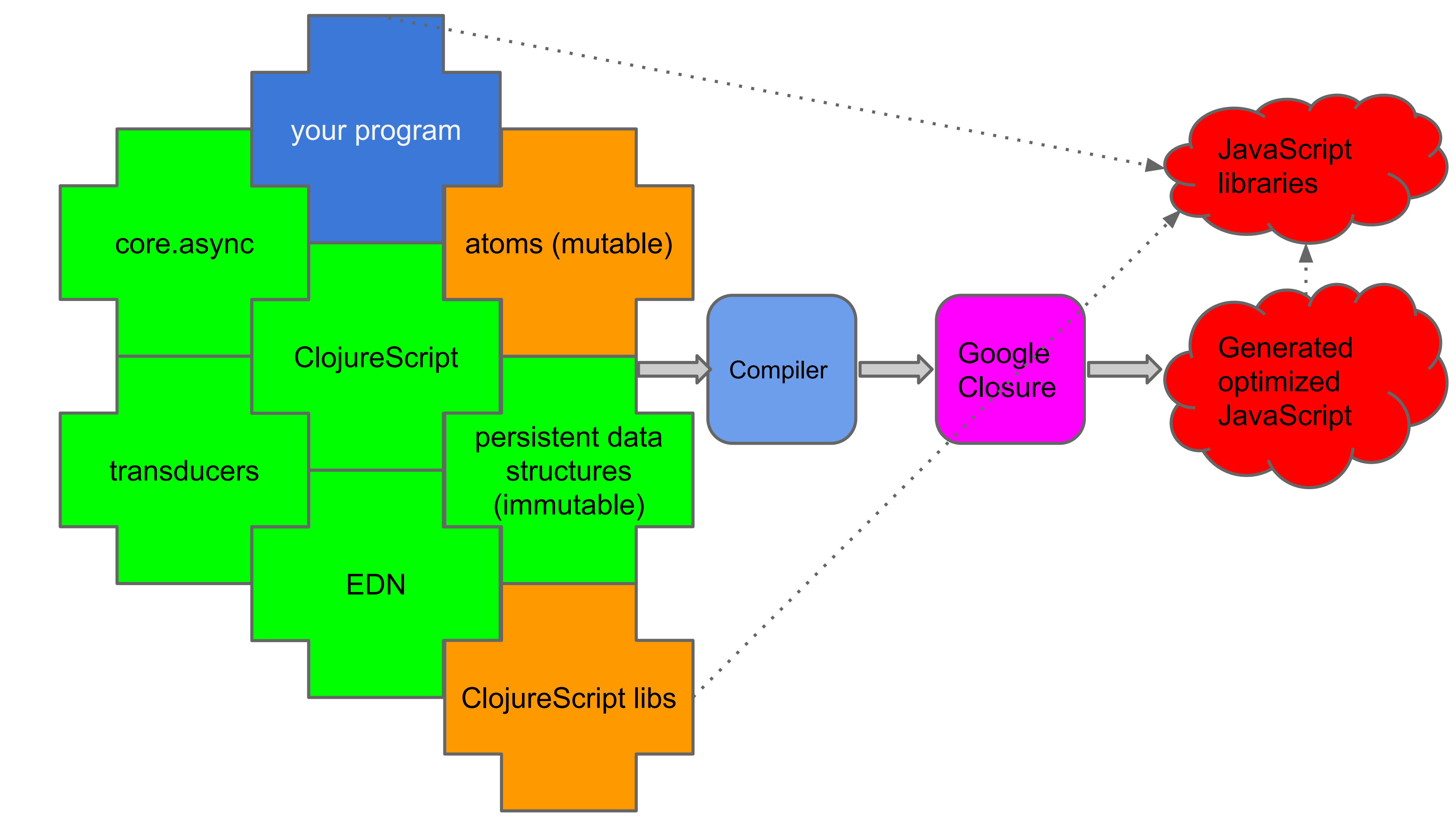


Weasel (browser connected REPL)



figwheel: live code reloading




your program
core.async
atoms (mutable)
ClojureScript
transducers
persistent data structures (immutable)
EDN
ClojureScript libs
Compiler
Google Closure
JavaScript libraries
Generated optimized JavaScript

Mutable state

Atoms are mutable references to immutable values.
Isolation of mutation.
One of 4 kinds of mutable references in Clojure.
(the others: vars, refs and agents)

In JVM Clojure:

```
1 (def my-atom (atom 1)) ;; atom with long in it
2 (deref my-atom) ;; 1
3 @my-atom ;; same, 1
4 (reset! my-atom 2)
5 @my-atom ;; now atom contains 2
6 (doseq [i (range 100)]
7   (future (reset! my-atom (inc @my-atom))))
8 @my-atom ;; 95, OMG, WHY!!!
```

Mutable state

Atoms are atomically updated only via `swap!`

- swap! takes a function of one or more arguments
- the function receives the old value of the atom as the first argument
- in ClojureScript you don't have this concurrency problem, but you still want to use the correct semantics (e.g. for Reagent atoms)

```
1 (def my-atom (atom 1))
2 (swap! my-atom (fn [old-value]
3                  (inc old-value)))
4 (swap! my-atom inc) ;; same
5 @my-atom ;; 3, inc-ed two times so far
6 (doseq [i (range 100)]
7   (future (swap! my-atom inc)))
8 @my-atom ;; 103, that's better
```

Web Applications

- Application State
  - Undo!
- User Interface & Interaction
  - Responding to changes in state & user actions
- Back-End integration
  - REST
  - WebSockets

Web App Dev in ClojureScript

An incomplete history

- Google Closure Libraries (goog.*)
- ClojureScriptOne (now defunct)
- WebFUI
- Pedestal.io - app library
- Hoplon

Web App Dev in ClojureScript

The Age of React

- Om
- Reagent

React

- Developed by Facebook
- Helps building *reusable*/*composable* UI components
  - *V* in MVC
- Leverages virtual DOM for performance
  - “dirty checking”
- Unidirectional Data Flow
  - vs. Data-binding
- Can render on “server-side”
  - To make apps crawler-friendly

React LifeCycle Methods

Mounting	**Updating**	**Unmounting**
• willMount • didMount	• willReceiveProps • shouldComponentUpdate • willUpdate • didUpdate	• willUnmount

- *Vaguely* resembles Cocoa/UIKit

Om

ClojureScript Interface to React.js

React + ClojureScript

Both Reagent and Om leverage

- immutability for faster comparison in `shouldComponentUpdate`
- Fewer redraws by batching updates with `requestAnimationFrame`

Om - Core Concepts

- Protocols to represent the *React's Life Cycle*
  - IWillMount, IDidUpdate, IWillUnmount etc.
- Om Component
  - a function that returns reified instances
- Component State
  - Cursor into App State

Om - Application Architecture

- Application State
  - Global app-state
  - components with cursors into app-state
  - state-transition
    - using transact! update! functions
- Local State
  - transient state for a component (e.g. form values)
- Shared State
  - globally shared via app root component

Om - State - Undo!

http://jackschaedler.github.io/goya/

Om - Component communication

- Inter-component communication
  - via mutating cursor (not good!)
  - Using core.async channels
  - callbacks

Om - UI

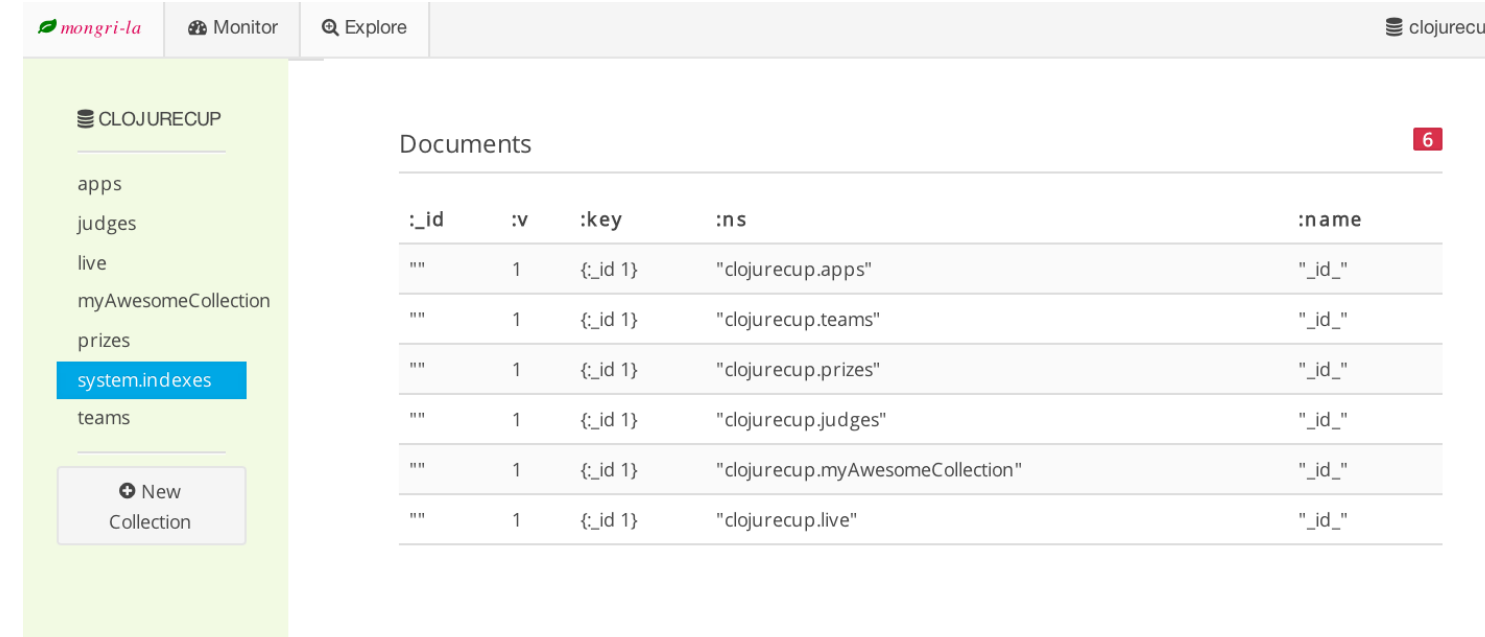
- Pluggable Templating
  - clojure DSL
    - library: Sablono
  - HTML selector style
    - library: kioo

Om Root Component

```
(om.core/root
  (fn [app-state owner]
    (reify
      om.core/IRender
      (render [_]
        (dom/h1 nil (:text app-state)))))
  {:text "Hello world!"}
  {:target (. js/document getElementById "my-app")})
```

Om Component Tree

- Navbar
  - Monitor
  - Explore
- Collections Sidebar
  - Collections
  - New Button
- Documents List
  - Documents
  - Count Badge



Alternatives to Om

- Reagent

Reagent

- ClojureScript interface to React
- Uses implementation of atom, called RAtom, for state management
- RAtoms can be shared at will: globally or locally (closure), no matter structure of component tree
- Components are "just" functions that
  - accept props
  - can deref atom(s)
  - return something renderable by React
  - may return a closure, useful for setting up local state
- Components are only re-rendered when
  - props change
  - watched atoms change

    (you're automatically watching when dereffing one)

Example

```
(defonce app-state (atom (rand-int 100)))

(defn number-component []
  [:h2 @app-state])

(defn main []
  [:div
   [:h1 "Welcome to my Reagent app!"]
   [number-component]
   [:button.btn.btn-primary
    {:on-click #(reset! app-state (rand-int 100))}
    "Click to change number"]])

(reagent/render-component [main]
                          (js/document.getElementById "app"))
```

Welcome to my Reagent app!

87

Click to change number

More complicated example

`fmamsclj.reagent.cljs`

- animals-state contains set with animals retrieved from server
- crud operations: add, delete, change are done asynchronously in go blocks and state is updated using response from server
- each table row has a local atom shared with its fields for update
- editable component: renders itself as text or input depending on click on button "Edit"
- buttons are disabled if relevant input is not valid
- table is sorted automatically by name of animal

Let's see the code and the running app

My experience with Om and Reagent

- Both awesome
- Added value on top of React (which is awesome in itself)
- Reagent is simple, flexible, straightforward
  May be a bit overlooked by newcomers
  More clojure-ish and less verbose than Om

Questions?