

# 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

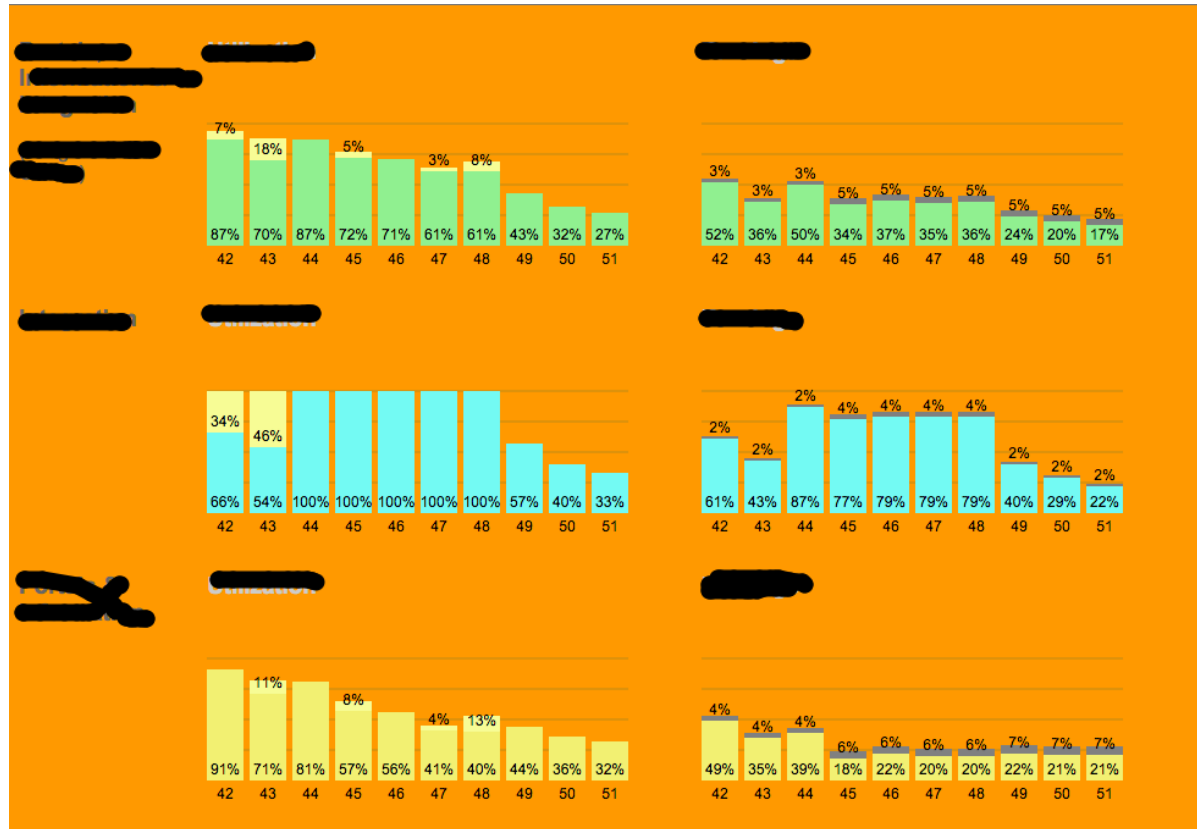


Vijay

# Full Clojure stack examples @ Finalist

- Clojure + Liberator + Atomic 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'

The screenshot displays a web application interface. At the top, there is a green header bar with a navigation menu. Below the header, a navigation bar contains icons for a coffee cup, a person reading, a shopping cart, and a storefront, with a '81%' progress indicator. The main content area features a breadcrumb trail: 'Initial > Product > Kenmerken > Gebied > Ranking > Bestellen'. The 'Product' step is active. Below the breadcrumb, there are three input fields: 'Categorie' (with a dropdown menu), 'Product' (containing 'Test'), and 'Aantal' (containing '200'). At the bottom, there are two buttons: 'Vorige' (Previous) and 'Volgende' (Next), and a link 'Terug naar dashboard' (Back to dashboard).

# 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.

Bestellingen

Step 2.

Omschrijving Bla  
Dataset

Na het toevoegen van één of meerdere variabelen kan de regel worden opgeslagen.

Categorie

Subcategorie

Variabele

Voeg toe

Hoofdcategorie	Subcategorie	Beschrijving	
Automotive			✕
		gezin jongste ki	✕

Opslaan

Terug naar dashboard

Vorige

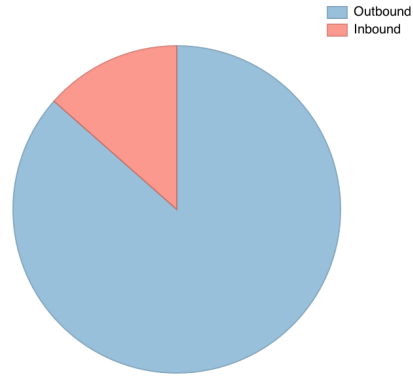
Volgende

+Nieuw

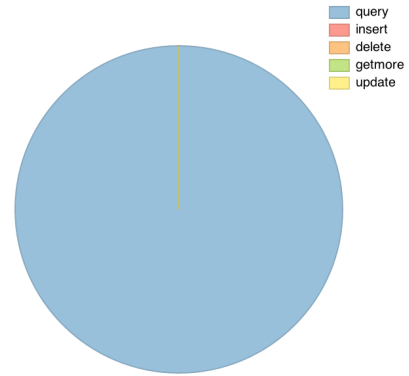
# ClojureCup Entry

- Clojure Backend
- Om Front-End

Network



Queries



## Server Information

### Server Info

<i>Version</i>	2.4.9
<i>Hostname</i>	ubuntu-14
<i>Host</i>	127.0.0.1
<i>Port</i>	27017
<i>PID</i>	860

### Operation Counters

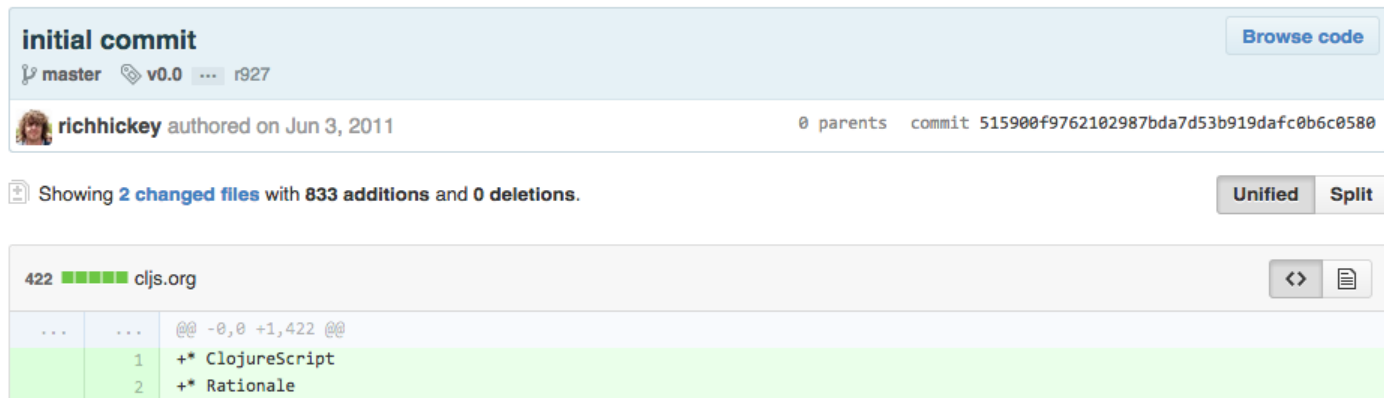
<i>insert</i>	1
<i>query</i>	46065
<i>getmore</i>	0
<i>update</i>	0
<i>delete</i>	0

### Network

<i>Active connections</i>	4
<i>Available connections</i>	19996
<i>Data In</i>	29 MB
<i>Data Out</i>	190 MB
<i>Request count</i>	532766

# Brief history of ClojureScript


June 20th 2011: first release of ClojureScript



The screenshot shows a GitHub commit page for the initial commit of ClojureScript. The commit is titled "initial commit" and is associated with the master branch, version v0.0, and commit hash r927. It was authored by richhickey on June 3, 2011. The commit message is "422 cljs.org" and it shows 2 changed files with 833 additions and 0 deletions. The files listed are ClojureScript and Rationale.


**initial commit** [Browse code](#)

master v0.0 r927

 richhickey authored on Jun 3, 2011

0 parents commit 515900f9762102987bda7d53b919dafc0b6c0580

Showing 2 changed files with 833 additions and 0 deletions. [Unified](#) [Split](#)

422  cljs.org [<>](#) [📄](#)

...	...	@@ -0,0 +1,422 @@
	1	+* ClojureScript
	2	+* Rationale



# 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



**PersistentHashMap ported from Clojure** ...

michalmarczyk authored on Apr 11, 2012 → David Nolen committed on Apr 20, 2012



# 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)))
      (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 block) (:r me))
                          (- (:y block) (:h block) (:r me)))]
        (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

**K** First created · 4 backed

**f** **Chris Granger** 189 friends

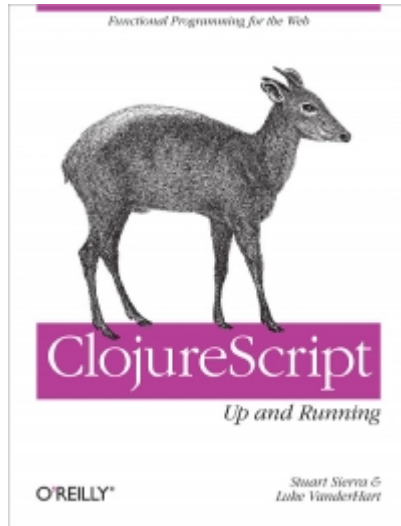
**globe** [chris-granger.com](http://chris-granger.com)

[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.

Clojure >

### ANN: ClojureScript 0.0-1885 (Source maps!)

8 posts by 5 authors 



David Nolen

```
12 (def black-hole-pos {:x 400 :y 400})
13 (def draggable (atom {:x 100 :y 100 :alive? true}))
14
15 (defn close? [x y]
16   (and (< (Math/abs (- x (:x black-hole-pos))) 50)
17        (< (Math/abs (- y (:y black-hole-pos))) 50)))
18
19 (defn get-client-rect [evt]
20   (let [r (.getBoundingClientRect (.-target evt))]
21     {:left (.-left r), :top (.-top r)}))
22
```

{ } 2 lines, 27 characters selected

	Scope Variables	Watch Expressions
<b>Call Stack</b> <input type="checkbox"/> Async		
close_QMARK_	main.cljs:16	
(anonymous function)	main.cljs:29	
goog.events.fireListener	events.js:741	
goog.events.handleBrowserEvent_	events.js:862	
(anonymous function)	events.js:276	
Paused on a JavaScript breakpoint.		
<b>Breakpoints</b>		
<input checked="" type="checkbox"/> main.cljs:16		
(and (< (Math/abs (- x (:x black-hole-pos))) 50)		

Local
▶ this: Window
x: 100
y: 103
▶ Global

# Brief history of ClojureScript

December 2013: ClojureScript interfaces to React

Commits on Dec 03, 2013



**initial commit**

swannodette authored on Dec 3, 2013



cfb4639



**Initial version**

holmsand authored on Dec 16, 2013



12566ce





# Brief history of ClojureScript

## TRANSDUCERS ARE COMING

August 2014



Posted by Rich Hickey on August 6, 2014



**David Nolen**  
@swannodette



Following

Transducers are a huge perf win for ClojureScript core.async, goodbye intermediate garbage for sequence-like ops

↩ Reply ↻ Retweet ★ Favorited ⋮ More

RETWEETS

28

FAVORITES

22



6:59 PM - 19 Aug 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 (`cljs`)

# JavaScript - ClojureScript

<pre>console.log("Hello, world!");</pre>	<pre>(.log js/console "Hello, world!") or (println "Hello, world!")</pre>
<pre>no implementation</pre>	<pre>(ns my.library   (:require [other.library :as other]))</pre>
<pre>var foo = "bar";</pre>	<pre>(def foo "bar")</pre>
<pre>function foo() {   var bar = 1; }</pre>	<pre>(defn foo []   (let [bar 1]))</pre>
<pre>// In JavaScript locals are mutable  function foo(x) {   x = "bar"; }</pre>	<pre>;; this will issue an error  (defn foo [x]   (set! x "bar"))</pre>

# JavaScript - ClojureScript

No implementation	<pre>(def v (vector)) (def v []) (def v [1 2 3]) (conj v 4) ;; =&gt; [1 2 3 4] (get v 0) ;; =&gt; 1 (v 0) ;; =&gt; 1</pre>
No implementation	<pre>(def s (set)) (def s #{}) (def s #{"cat" "bird" "dog"}) (conj s "cat") ;; =&gt; #{"cat" "bird" "dog"} (contains? s "cat") ;; true (s "cat") ;; "cat" (s "fish") ;; nil</pre>
No implementation	<pre>(def m (hash-map)) (def m {}) (def m {:foo 1 :bar 2}) (conj m [:baz 3]) ;; =&gt; {:foo 1 :bar 2 :baz 3} (assoc m :foo 2) ;; =&gt; {:foo 2 :bar 2} (get m :foo) ;; =&gt; 2 (m :foo) ;; =&gt; 2</pre>

# JavaScript - ClojureScript

<pre>if (bugs.length &gt; 0) {   return 'Not ready for release'; } else {   return 'Ready for release'; }</pre>	<pre>(if (pos? (count bugs))   "Not ready for release"   "Ready for release")</pre>
<pre>function foo() {   var bar = 1;   var baz = 2;   return bar + baz; } foo(); // 3</pre>	<pre>(defn foo []   (let [bar 1         baz 2]     (+ bar baz))) (foo) ;; =&gt; 3</pre>

# 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"))

```
(defn g []  
  (f (System/getProperty "foo")))
```

# cljs.core.typed

The image shows a Clojure REPL window with two panes. The left pane shows the source code for a namespace named `fg.main`. The right pane shows the REPL's output, including the initialization of `cljs.core.typed` and a type error.

```
(ns fg.main
  (:require-macros [cljs.core.typed :as t :refer [ann]])
  (:require [cljs.core.typed :as t]))

(ann parse-int [string -> number])

(defn parse-int [s]
  (js/parseInt s))

(parse-int 3)
```

REPL Local: fg.api

```
(require '[cljs.core.typed :as tcs])
=> nil
(tcs/check-ns* 'fg.main)
Initializing core.typed ...
Found ClojureScript, loading ...
Finished loading ClojureScript
Building core.typed base environments ...
DEPRECATED SYNTAX (NO_SOURCE_PATH): Rec syntax is deprecated, use cljs.co
DEPRECATED SYNTAX (NO_SOURCE_PATH): HVec syntax is deprecated, use cljs.c
Finished building base environments
"Elapsed time: 53450.824644 msec"
core.typed initialized.
Start collecting fg.main
Finished collecting fg.main
Collected 1 namespaces in 142.538086 msec
Not checking cljs.core.typed (tagged :collect-only in ns metadata)
Start checking fg.main
Checked fg.main in 282.926339 msec
Checked 2 namespaces in 465.777159 msec
Type Error (fg.main:8:3) Found untyped var: js/parseInt
in: js/parseInt

Type Error (NO_SOURCE_FILE) Function fg.main/parse-int could not be appli

Domains:
  string

Arguments:
  (closure.core.typed/Val 3)

Ranges:
  number

in: (fg.main/parse-int 3)
```

# Weasel (browser connected REPL)

localhost:8090/reagent.html

Name	Species		
Aardwolf	Proteles cristata	Edit	x
Atlantic salmon	Salmo salar	Edit	x
Curled octopus	Eledone cirrhosa	Edit	x
Dung beetle			
Gnu			
Horny toad			
Painted-snail			
Yellow-billed cuckoo			

```
1. lein repl (java)
lein (java) | lein (bash) | lein (java)
fpmsclj.reagent=> (.click (js/$ "button[data-reactid='.0.0.1.0:$animal-row-3.2.0']"))
#<[object Object]>
nil
fpmsclj.reagent=> (.click (js/$ "button[data-reactid='.0.0.1.0:$animal-row-3.2.0']"))
#<[object Object]>
nil
fpmsclj.reagent=> (.click (js/$ "button[data-reactid='.0.0.1.0:$animal-row-3.2.0']"))
#<[object Object]>
nil
fpmsclj.reagent=> (.click (js/$ "button[data-reactid='.0.0.1.0:$animal-row-3.2.0']"))
#<[object Object]>
nil
fpmsclj.reagent=> (.click (js/$ "button[data-reactid='.0.0.1.0:$animal-row-3.2.0']"))
#<[object Object]>
nil
fpmsclj.reagent=> (.click (js/$ "button[data-reactid='.0.0.1.0:$animal-row-3.2.0']"))
#<[object Object]>
nil
fpmsclj.reagent=> []
```

Elements Network Sources Timeline

```
<html>
  <head>...</head>
  <body>
    <div id="app">
      <div data-reactid=".0">
        <table class="table table-striped" data-reactid=".0.0">
          <thead data-reactid=".0.0.0">...</thead>
          <tbody data-reactid=".0.0.1">
            <tr data-reactid=".0.0.1.0:$animal-row-3">
              <td data-reactid=".0.0.1.0:$animal-row-3.0">Aardwolf</td>
              <td data-reactid=".0.0.1.0:$animal-row-3.1">Proteles cristata</td>
              <td data-reactid=".0.0.1.0:$animal-row-3.2">
                <button class="btn btn-primary pull-right" data-reactid=".0.0.1.0:$animal-row-3.2.0">Edit</button>
              </td>
            <tr data-reactid=".0.0.1.0:$animal-row-3.3">...</tr>
            <tr data-reactid=".0.0.1.0:$animal-row-8">...</tr>
            <tr data-reactid=".0.0.1.0:$animal-row-5">...</tr>
          </tbody>
        </table>
      </div>
    </div>
  </body>
</html>
```

... #app div table tbody tr td button.btn.btn-primary.pull-right

Styles Event Listeners DOM Breakpoints Properties AngularJS Properties

```
element.style {
}
.btn- bootstrap-theme.min.css:5
primary {
  background-image: -webkit-linear-gradient(to top, #428bca 0%, #2d6ca2 100%);
  background-image: -o-linear-gradient(to top, #428bca 0%, #2d6ca2 100%);
  background-image: -webkit-gradient(linear, left top, left bottom, from(
```

margin -

border 1

padding 6

1 12 24.375 x 20 12 1 -

6 1

1

Find in Styles Filter

Console Search Emulation Rendering

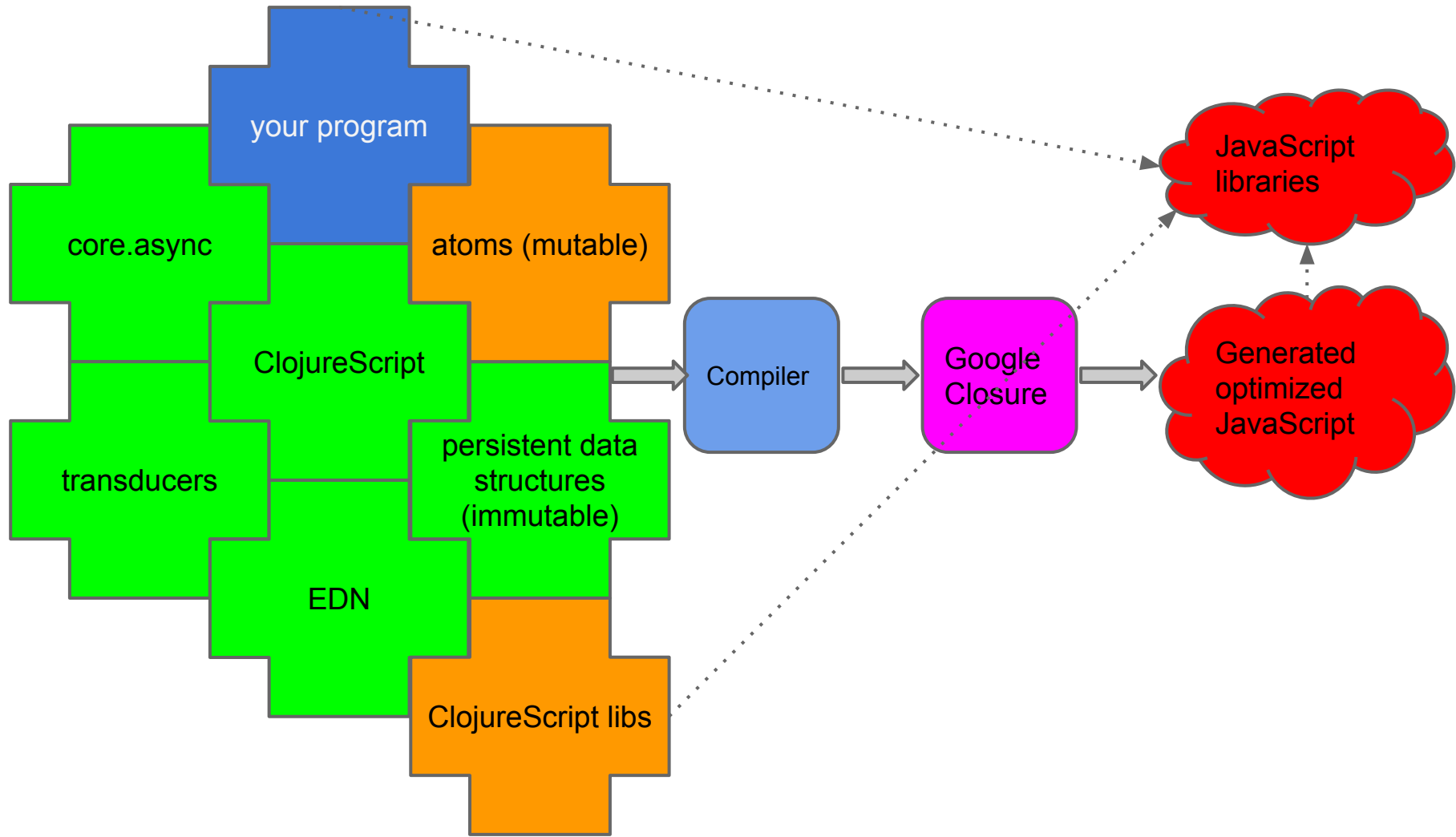
```
<top frame>
  XHR finished loading: GET "http://localhost:8090/animals".
    xhr.io.js:561
  Figwheel: trying to open cljs reload socket client.cljs:143
  Figwheel: socket connection established client.cljs:155
  Opened Websocket REPL connection repl.cljs:52
  XHR finished loading: PUT "http://localhost:8090/animals/3".
```

# figwheel: live code reloading

The image shows a web browser interface with two columns of animal names: "Yellow-backed duiker" and "Cephalophus silvicultor". Each name has an empty input field below it. To the right of the second name are "Edit" and "Add" buttons. A red "X" button is also visible. Below the browser, a code editor window titled "fpamsclj.reagent - [fpamsclj]" is open, showing Clojure code for a reagent component. The code includes a function definition for "animal-form" and a click handler for "remove-animal!".

```
    {:onClick #(remove-animal! (current-animal))}
    "\u00D7"]]))))

(defn animal-form []
  (let [initial-form-values {:name      ""
                            :species   ""
                            :editing? true}
        form-state (atom initial-form-values)]
    ...))
```



# 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
<ul style="list-style-type: none"><li>• willMount</li><li>• didMount</li></ul>	<ul style="list-style-type: none"><li>• willReceiveProps</li><li>• shouldComponentUpdate</li><li>• willUpdate</li><li>• didUpdate</li></ul>	<ul style="list-style-type: none"><li>• willUnmount</li></ul>

- *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

The screenshot shows the Mongri-la application interface. At the top, there is a navigation bar with the Mongri-la logo, a 'Monitor' button, an 'Explore' button, and the 'clojurecup' logo. Below the navigation bar is a sidebar with a light green background. The sidebar contains the following items: 'CLOJURECUP' (with a hamburger icon), 'apps', 'judges', 'live', 'myAwesomeCollection', 'prizes', 'system.indexes' (highlighted in blue), and 'teams'. At the bottom of the sidebar is a 'New Collection' button. To the right of the sidebar is a 'Documents' section with a red badge containing the number '6'. Below the 'Documents' section is a table with the following columns: ':\_id', ':v', ':key', ':ns', and ':name'. The table contains six rows of data.

:_id	:v	:key	:ns	:name
""	1	{:_id 1}	"clojurecup.apps"	"_id_"
""	1	{:_id 1}	"clojurecup.teams"	"_id_"
""	1	{:_id 1}	"clojurecup.prizes"	"_id_"
""	1	{:_id 1}	"clojurecup.judges"	"_id_"
""	1	{:_id 1}	"clojurecup.myAwesomeCollection"	"_id_"
""	1	{:_id 1}	"clojurecup.live"	"_id_"

# 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?**