

ClojureScript ReactJS


Michiel Borkent

[@borkdude](https://twitter.com/borkdude)


DomCode, May 26th 2015

FINALIST
open *IT* oplossingen


Michiel Borkent ([@borkdude](https://twitter.com/borkdude))

- Clojure(Script) developer at  **FINALIST**
open IT oplossingen
- Clojure since 2009
- Former lecturer, taught Clojure



Agenda

- Part 1: ClojureScript
- Part 2: ClojureScript  ReactJS





Warning




William Morgan
@wm




i love functional programming. it takes smart people who would otherwise be competing with me and turns them into unemployable crazies

 Reply  Retweeted  Favorite  More

RETWEETS	FAVORITES	
923	875	

8:53 PM - 30 Dec 2009



Part 1: ClojureScript



Current status

- JavaScript is everywhere, but not a robust and concise language - [wat](#)
Requires discipline to only use "the good parts"
- JavaScript is taking over: UI logic from server to client
- JavaScript is not going away in the near future
- Advanced libraries and technologies exist to optimize JavaScript: (example: Google Closure)

ClojureScript

- Released June 20th 2011
- Client side story of Clojure ecosystem
- Serves Clojure community:
 - 50%* of Clojure users also use ClojureScript
 - 93%** of ClojureScript users also use Clojure
- ClojureScript targets JavaScript by adopting Google Closure
 - libraries: `goog.provide/require` etc.
 - optimization: dead code removal

* <http://cemerick.com/2013/11/18/results-of-the-2013-state-of-clojure-clojurescript-survey/>

** <http://blog.cognitect.com/blog/2014/10/24/analysis-of-the-state-of-clojure-and-clojurescript-survey-2014>

Syntax

$f(x) \rightarrow (f\ x)$

Syntax

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

->

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

Syntax

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

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

JavaScript - ClojureScript

```
// In JavaScript  
// locals are mutable
```

```
function foo(x) {  
  x = "bar";  
}
```

```
;; this will issue an  
;; error
```

```
(defn foo [x]  
  (set! x "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")
```

JavaScript - ClojureScript

```
var foo = {bar: "baz"};  
foo.bar = "baz";  
foo["abc"] = 17;
```

```
alert('foo')  
new Date().getTime()  
new Date().getTime().toString()
```

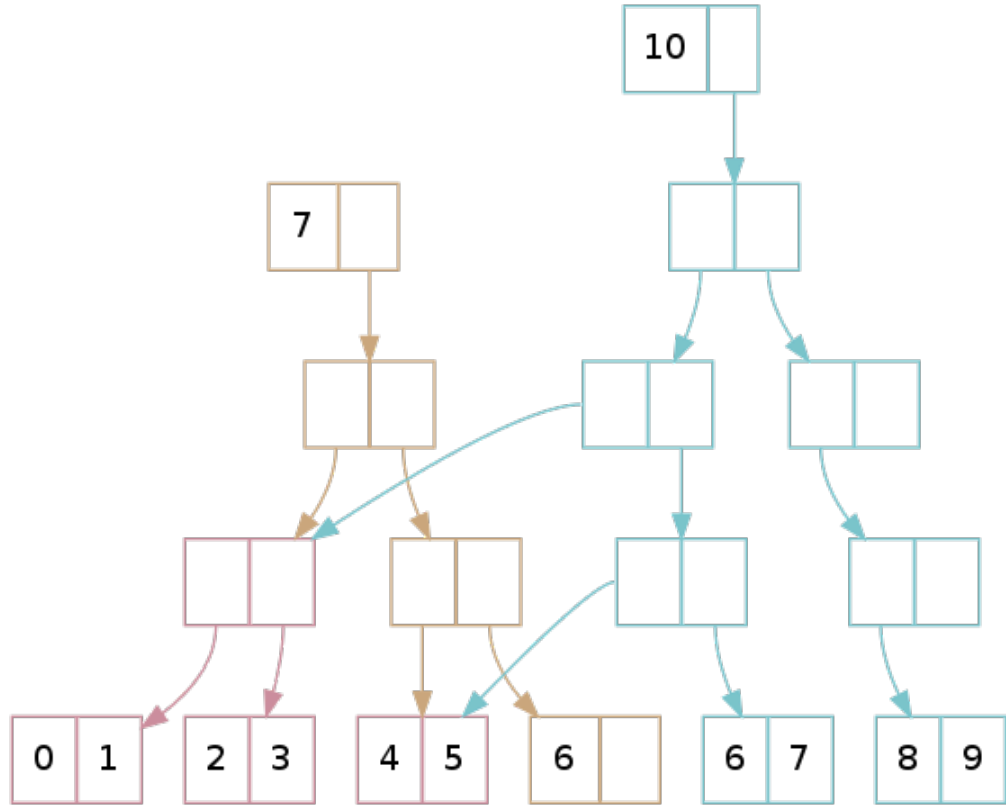
```
(def foo (js-obj "bar" "baz"))  
(set! (.bar foo) "baz")  
(aset foo "abc" 17)  
  
(js/alert "foo")  
(.getTime (js/Date.))  
(.. (js/Date.) (getTime) (toString))
```

Core language features

- persistent immutable data structures
- functional programming
- sequence abstraction
- isolation of mutable state (atoms)
- Lisp: macros, REPL
- `core.async`

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
```

Isolation of state

one of possible
pre-React patterns

```
(def app-state (atom []))

(declare rerender)

(add-watch app-state ::rerender
  (fn [k a o n]
    (rerender o n)))

(defn add-todo [text]
  (let [tt (.trim text)]
    (if (seq tt)
      (swap! app-state conj
        {:id (get-uuid)
         :title tt
         :completed false}))))
```

function called
from event
handler

new todo

Lisp: macros

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

thread last macro



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


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
```

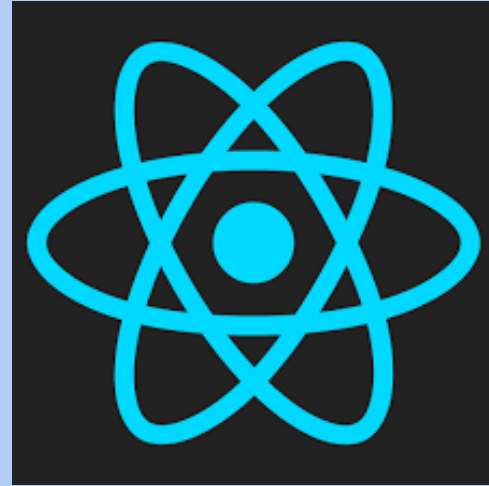
notes:

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

core.async

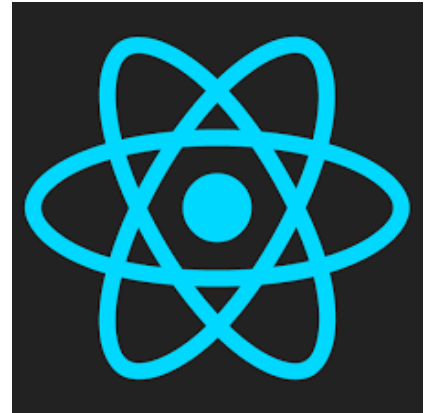
```
(go (let [email (:body
                (<! (http/get
                     (str "/api/users/"
                          "123"
                          "/email"))))
          orders (:body
                 (<! (http/get
                      (str
                       "/api/orders-by-email/"
                       email)))]
    (count orders)))
```

Part 2:



React

- Developed by Facebook
- Helps building reusable and composable UI components
- Leverages virtual DOM for performance
- Can render on server to make apps crawlable
- JSX templating language



```
var Counter = React.createClass({
  getInitialState: function() {
    return {counter: this.props.initialCount};
  },
  inc: function() {
    this.setState({counter: this.state.counter + 1});
  },
  render: function() {
    return <div>
      {this.state.counter}
      <button onClick={this.inc}>x</button>
    </div>;
  }
});
```

10



```
React.renderComponent(<Counter initialCount={10}/>, document.body);
```

Reagent

ClojureScript interface to React

- Uses special atoms for state
- Data literals for templating
- Uses batching + more efficient `shouldComponentUpdate`



Components are **functions** that

- **must** return something renderable by React
- **can** deref atom(s)
- **can** accept props as args
- **may** return a closure, useful for setting up initial state

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)

Hiccup syntax

```
[ :a { :href "/logout" }  
  "Logout" ]
```

```
<a href="/logout">Logout</a>
```

```
[ :div#app.container  
  [ :h2 "Welcome" ] ]
```

```
<div id="app" class="container">  
  <h2>Welcome</h2>  
</div>
```

RAtom

```
(def count-state (atom 10))
```

```
(defn counter []  
  [:div  
    @count-state  
    [:button {:on-click (fn [] (swap! count-state inc))}  
      "x"]])
```

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

10



```
(defn local-counter [start-value]
  (let [count-state (atom start-value)]
    (fn []
      [:div
       @count-state
       [:button {:on-click #(swap! count-state inc)}
        "x"]]))))
```

local
RAtom

10



```
(reagent/render-component [local-counter 10]
  (js/document.getElementById "app"))
```

CRUD!

Name	Species		
Aardwolf	Proteles cristata	Edit	×
Atlantic salmon	Salmo salar	Edit	×
Curled octopus	Eledone cirrhosa	Edit	×
Dung beetle	Scarabaeus sacer	Edit	×
Gnu	Connochaetes gnou	Edit	×
Horny toad	Phrynosoma cornutum	Edit	×
Painted-snipe	Rostratulidae	Edit	×
Yellow-backed duiker	Cephalophus silvicultor	Edit	×
<input type="text"/>	<input type="text"/>	Add	

```
(def Animals
  "A schema for animals state"
  #{{:id      s/Int
      :type    s/Keyword
      :name    s/Str
      :species s/Str}}})
```

RAtom with set containing
animal hash-maps

```
(defonce animals-state
  (atom #{}
        :validator
        (fn [n]
          (s/validate Animals n))))
```

```
;; initial call to get animals from server
(go (let [response
          (<! (http/get "/animals"))
          data (:body response)]
      (reset! animals-state (set data))))
```

```
(...
{:id 2,
 :type :animal,
 :name "Yellow-backed duiker",
 :species "Cephalophus silvicultor"}
{:id 1,
 :type :animal,
 :name "Painted-snipe",
 :species "Rostratulidae"})
```

Render all animals from state

```
(defn animals []
```

```
  [:div  
    [:table.table.table-striped  
      [:thead  
        [:tr  
          [:th "Name"] [:th "Species"] [:th ""] [:th ""]]]]
```

```
    [:tbody  
      (map (fn [a]  
            ^{:key (str "animal-row-" (:id a))}  
            [animal-row a])  
            (sort-by :name @animals-state))
```

```
      [animal-form]]]]])
```

animal-row component

```
{:editing? false, :name "Yellow-backed duiker", :species "Cephalophus silvicultor"}
```

Yellow-backed duiker	Cephalophus silvicultor	Edit	×
----------------------	-------------------------	------	---

```
{:editing? true, :name "Yellow-backed pony", :species "Cephalophus silvicultor"}
```

Yellow-backed pony	Cephalophus silvicultor	Save	×
--------------------	-------------------------	------	---

```
(defn animal-row [a]
  (let [row-state (atom {:editing? false
                        :name      (:name a)
                        :species   (:species a)})]
    current-animal (fn []
                     (assoc a
                            :name (:name @row-state)
                            :species (:species @row-state))))]
  (fn []
    [:tr
     [:td [editable-input row-state :name]]
     [:td [editable-input row-state :species]]
     [:td [:button.btn.btn-primary.pull-right
           {:disabled (not (input-valid? row-state))
            :on-click (fn []
                        (when (:editing? @row-state)
                          (update-anim! (current-animal)))
                          (swap! row-state update-in [:editing?] not))
                        (if (:editing? @row-state) "Save" "Edit"))]}]
     [:td [:button.btn.pull-right.btn-danger
           {:on-click #(remove-anim! (current-animal))}
           "\u00D7"]]]]))
```



```
(defn editable-input [atom key]
  (if (:editing? @atom)
    [:input {:type "text"
             :value (get @atom key)
             :on-change (fn [e] (swap! atom
                                       assoc key
                                       .. e -target -value)))]
    [:p (get @atom key)]))
```

```
{:editing? true, :name "Yellow-backed pony", :species "Cephalophus silvicultor"}
```

<input type="text" value="Yellow-backed pony"/>	<input type="text" value="Cephalophus silvicultor"/>	<input type="button" value="Save"/>	<input type="button" value="x"/>
---	--	-------------------------------------	----------------------------------

```
{:editing? false, :name "Yellow-backed duiker", :species "Cephalophus silvicultor"}
```

Yellow-backed duiker	Cephalophus silvicultor	<input type="button" value="Edit"/>	<input type="button" value="x"/>
----------------------	-------------------------	-------------------------------------	----------------------------------

```
(defn animal-row [a]
  (let [row-state (atom {:editing? false
                        :name      (:name a)
                        :species   (:species a)})]
    current-animal (fn []
                     (assoc a
                           :name (:name @row-state)
                           :species (:species @row-state))))]
  (fn []
    [:tr
     [:td [editable-input row-state :name]]
     [:td [editable-input row-state :species]]
     [:td [:button.btn.btn-primary.pull-right
           {:disabled (not (input-valid? row-state))
            :on-click (fn []
                       (when (:editing? @row-state)
                         (update-animal! (current-animal)))
                         (swap! row-state update-in [:editing?] not))}
           (if (:editing? @row-state) "Save" "Edit")]]]
     [:td [:button.btn.pull-right.btn-danger
           {:on-click #(remove-animal! (current-animal))}
           "\u00D7"]]]))
```

```
(defn input-valid? [atom]
  (and (seq (-> @atom :name))
       (seq (-> @atom :species))))
```

```
(defn animal-row [a]
  (let [row-state (atom {:editing? false
                        :name      (:name a)
                        :species   (:species a)})]
    current-animal (fn []
                     (assoc a
                            :name (:name @row-state)
                            :species (:species @row-state))))]
  (fn []
    [:tr
     [:td [editable-input row-state :name]]
     [:td [editable-input row-state :species]]
     [:td [:button.btn.btn-primary.pull-right
           {:disabled (not (input-valid? row-state))
            :on-click (fn []
                        (when (:editing? @row-state)
                          (update-anim! (current-animal)))
                          (swap! row-state update-in [:editing?] not))}
           (if (:editing? @row-state) "Save" "Edit")]]
     [:td [:button.btn.pull-right.btn-danger
           {:on-click #(remove-anim! (current-animal))}
           "\u00D7"]]]))
```

```
(defn update-animal! [a]
  (go (let [response
            (<! (http/put (str "/animals/" (:id a))
                          {:edn-params a}))
            updated-animal (:body response)]
      (swap! animals-state
              (fn [old-state]
                (conj
                 (remove-by-id old-state (:id a))
                 updated-animal)))))))
```

```
(defn animal-row [a]
  (let [row-state (atom {:editing? false
                        :name      (:name a)
                        :species   (:species a)})]
    current-animal (fn []
                     (assoc a
                            :name (:name @row-state)
                            :species (:species @row-state))))]
  (fn []
    [:tr
     [:td [editable-input row-state :name]]
     [:td [editable-input row-state :species]]
     [:td [:button.btn.btn-primary.pull-right
           {:disabled (not (input-valid? row-state))
            :on-click (fn []
                       (when (:editing? @row-state)
                         (update-animal! (current-animal)))
                         (swap! row-state update-in [:editing?] not))
                       (if (:editing? @row-state) "Save" "Edit"))]}]
     [:td [:button.btn.pull-right.btn-danger
           {:on-click #(remove-animal! (current-animal))}
           "\u00D7"]]]))])
```

```
(defn remove-animal! [a]
  (go (let [response
            (<! (http/delete (str "/animals/"
                               (:id a))))]
        (if (= 200 (:status response))
            (swap! animals-state remove-by-id (:id a)))))))
```

if server says:
"OK!", remove
animal from
CRUD table

Exercises

- Sort table by clicking on name or species
- Optimistic updates

Code and slides at:

<https://github.com/borkdude/domcode-cljs-react>

How to run at home?

- Install JDK 7+
- Install [leiningen](#) (build tool)
- `git clone https://github.com/borkdude/domcode-cljs-react.git`
- `cd domcode-cljs-react/code/animals-crud`
- See README.md for further instructions

Probably Cursive IDE (IntelliJ) is most beginner friendly

Leiningen

- Used by 98% of Clojure users
- Clojure's Maven
- Managing dependencies
- Running a REPL
- Packaging and deploying
- Plugins:
 - `lein cljsbuild` – building ClojureScript
 - `lein figwheel` – live code reloading in browser



Debugging

Source maps let you debug ClojureScript directly from the browser

The image shows a browser's developer tools interface. On the left is a file explorer showing a directory structure with folders like 'clojure', 'cognitect', 'com/cognitect', 'drag', 'main.cljs', 'main.js', 'goog', 'no/en', 'reagent', and 'webinars'. The 'main.cljs' file is selected. The main area displays ClojureScript source code with line numbers 12 through 24. Line 16 is highlighted in blue, containing the code: `(and (< (Math/abs (- x (:x black-hole-pos))) 50`. Below the code editor is a status bar showing '{ } Line 16, Column 1'. At the bottom, the 'Call Stack' panel is open, showing a list of function calls: 'close_QMARK_' (main.cljs:16), '(anonymous function)' (main.cljs:29), 'goog.events.fireListener' (events.js:741), 'goog.events.handleBrowserEvent_' (events.js:862), and '(anonymous function)' (events.js:276). To the right of the call stack are panels for 'Scope Variables' and 'Watch Expressions'. The 'Local' scope shows 'this: Window', 'x: 197', and 'y: 116'. The 'Global' scope shows 'Window'.

```
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
23 (defn draggable-button []
24   (let [mouse-move-handler
```

{ } Line 16, Column 1

Scope Variables Watch Expressions

Call Stack 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

Local

- this: Window
- x: 197
- y: 116

Global Window

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 [AMSC LJ](#) meetup
- Join the [Slack](#) community

Thanks!