

# ClojureScript and core.async

Michiel Borkent

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

DevJam, July 29th 2015



# Agenda

- Part 1: ClojureScript
- Part 2: core.async

# 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>

The following slides are just for reference.  
Let's see some live ClojureScript!

## 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)

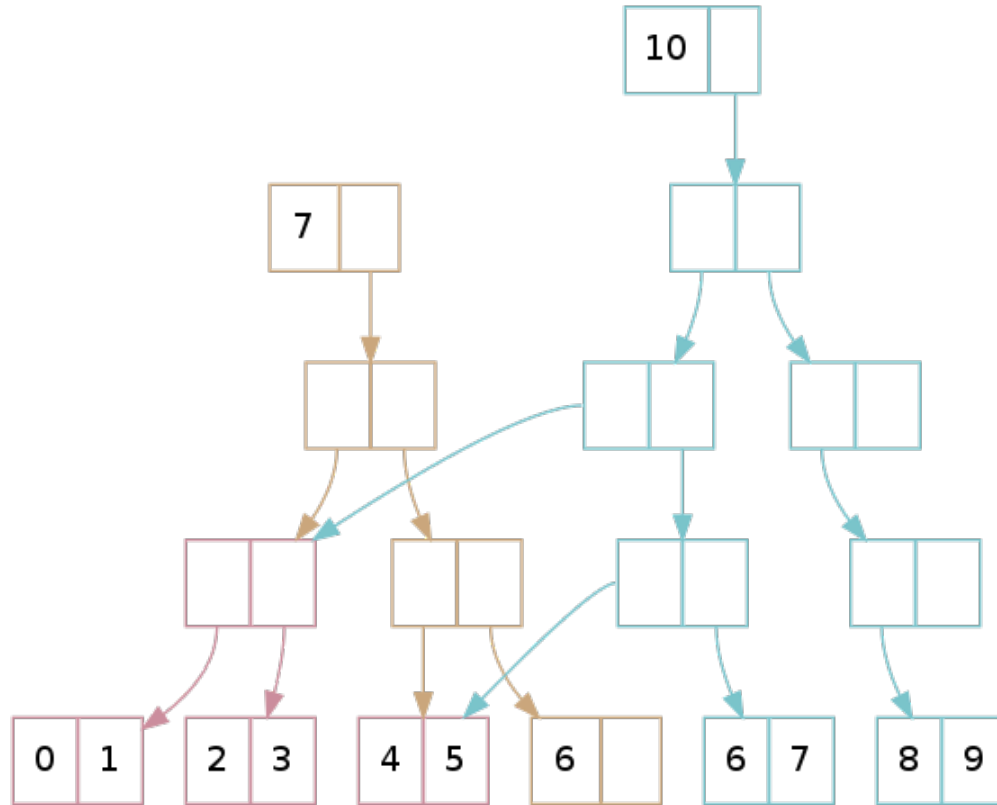
(js/alert "foo")
(.getTime (js/Date.))
(.. (js/Date.) (getTime)
(toString))
```

# 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
```



# 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

adapted from: <https://github.com/dfuenzalida/todo-cljs>

# 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

## 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 screenshot displays a web browser's developer tools interface. On the left, a file explorer shows a directory structure with folders like 'clojure', 'cognitect', 'com/cognitect', 'drag', 'goog', 'no/en', 'reagent', and 'webiars'. The file 'main.cljs' is selected under the 'drag' folder. The main panel shows the source code of 'main.cljs' with line 16 highlighted: `(and (< (Math/abs (- x (:x black-hole-pos))) 50)`. Below the code, a status bar indicates 'Line 16, Column 1'. At the bottom, the 'Call Stack' panel is expanded, showing a sequence 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, the 'Scope Variables' panel shows the current scope is 'Local', with variables 'this' (Window), 'x' (197), and 'y' (116). The 'Global' scope also 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

Call Stack	
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

Scope Variables	Watch Expressions
Local	
▶ this: Window	
x: 197	
y: 116	
Global	Window

# How to run at home?

- Install JDK 7+
- Install [leiningen](#) (build tool)
- `git clone https://github.com/borkdude/sytac-core-async.git`
- `cd sytec-core-async/code/cljs-demo`
- See README.md for further instructions

Probably Cursive IDE (IntelliJ) is most beginner friendly

## Part 2: core.async



# Excellent webinar by David Nolen

[http://go.cognitect.com/core\\_async\\_webinar\\_recording](http://go.cognitect.com/core_async_webinar_recording)

<https://github.com/cognitect/async-webinar>

# core.async ajax with cljs-http

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

# Get started with Clojure(Script)

- Read a Clojure(Script) [book](#)
- Do the [4closure](#) exercises
- Start hacking on your own project
- Pick an online Clojure [course](#)
- Join the [AMSClJ](#) meetup
- Join the [Slack](#) community