Electric Clojure v3 teaser: improved transfer semantics (2024)

2024 Aug 6 by Dustin Getz - https://twitter.com/dustingetz

The biggest remaining barrier to adopting Electric Clojure today is that it requires users to write **too many macros**. Some users have been willing to accept and set aside this issue while we deal with it but many cannot, they see writing macros as an insurmountable problem and that the technology is brittle or a hack.

So, today we are thrilled to finally say: **Electric v3 is coming, and it fixes this!** The root of the problem (that v3 solves) is what we call **undesired transfer**: when an Electric program implies a network topology that is different than the one that the programmer wants.

The easiest way to understand the new semantics is to understand what's wrong with the old, so let's start there.

Example of undesired transfer in Electric v2

Consider this (broken) Electric v2 code:

- There's server state - a database connection
- There's client state - a search box, bound to an atom
- We render a view, and the view queries the database with the search state

Clojure

```
#?(:clj (def !conn (d/create-conn ...))) ; database on server (e/defn
RenderView [search db] (e/client (js/console.log "querying with filter:
" search) (dom/div (dom/text (e/server (query-database search db))))))
(e/client (let [db (e/server (e/watch !conn)) !search (atom "") search
(e/watch !search)] (dom/input (dom/on! "keydown" (fn [e] (reset! !search
(-> e .-target .-value))))) (RenderView. search db) #_(RenderView.
search (e/server (e/watch !conn)) )))
```

Aside for newcomers: Electric can be quickly understood as "streaming lexical and dynamic scope" – so here, as `search` updates, the idea is to stream it to the server, update the server query, stream the result back into the client to render to the dom, in basically a big streamy loop as data flows from client to server and back. The dataflow topology is no different than that of any web app, Electric simply expresses the same thing more fluently.

Undesired transfer in Electric v2 happens in two ways:

1. **`let` is too eager, transferring all values to the site of the let.** This let is client-sited, so Electric attempts to transfer the result of `(e/server (e/watch !conn))` to the client to be bound to the name `db`, and fails to serialize `db` in the process because it is an unserializable reference type.
2. **Electric v2 functions transfer all of their arguments to the call site before calling the function.** For example, this call `(RenderView. search (e/server (e/watch !conn)))` is client-sited, therefore Electric attempts to first transfer the result of `(e/watch !conn)` (a database reference) to the client before booting `RenderView` with all client side args, and again the expr (being a database reference) will fail to serialize.

We can work around the issues by refactoring the program with these rules in mind:

Clojure

```
(e/server (let [db (e/watch !conn)] (e/client (let [!search (atom "")
search (e/watch !search)] (dom/input (dom/on! "keydown" (fn [e] (reset!
!search (-> e .-target .-value))))) (e/server (RenderView. search
db))))))
```

- split the `let` into two forms, one for each site
- move the `RenderView` call to server authority, so `db` is not transferred by function boot

But watch out, this refactor is not just inelegant, it has also introduced a **spurious transfer**: it transmits `search` to the server **too soon**, because the `(js/console.log` transfer: it transmits `search` to the server too soon, because the `(js/console.log "querying with filter: " search)` also needs `search` on the client, but now all `RenderView`'s args have been forced to the server, so Electric v2 will now send `search` back to the client! (So, `search` went client → server → client for no reason.)

At this point you end up needing to write macros to control placement by very carefully **unquoting arguments in the correct site** (and note the painful Hungarian notation to keep track of argument color):

Clojure

```clojure
(defmacro render-view [search-C db-S] ; notational bookkeeping :( `(e/
server (let [db# ~db-S] ; unquote db on server (e/client (let [search#
~search-C] ; unquote search on client (js/console.log search#) (dom/div
(dom/text (e/server (query search# db#)))))))))
```

Obviously not great. Especially given our belief that functional composition is the fundamental basis for creating non-leaky abstractions, Electric v2 lambdas are clearly not living up to our mission.

And this problem is not just theoretical. Imagine a reusable datagrid component, which is parameterized by various higher order fns for querying, rendering, sorting, filtering, pagination. **Macros are not values**, we cannot pass them as arguments like we do with closures. So once we start needing to use macros instead of functions, it's pretty much game over for abstraction: we've lost the very primitive—lambda—that we set out to build in the first place.

We spent about a year dealing with this, and are happy to now demonstrate that Electric v3 fixes it!

Electric v3: "Interop is sited, edges are not"

In Electric v3, network transfer (i.e. of a value from server to client) is driven by **Clojure/Script interop only**, because that—platform interop—is the essential reason why a computation must occur in one place versus another.

For example, a platform call like `(datomic.api/q db)` (JVM platform) or `(.createTextNode js/document "")` (browser platform) is inherently **sited**, i.e. those

[illegible]
vars are only available on the server or client classpaths respectively.

Siting is essential complexity (arguably *the* essence of a distributed system), and consequently we as programmers are hyper-aware of where (at which site) our effects must run, and we require perfect control over their placement.

And as far as placement goes, other than the location of the interop call, nothing else matters. If we write:

Clojure

```clojure
(e/client (let [db (e/server (e/watch !conn))] (e/server (datomic.api/q
db ...))))
```

the `let` conceptually is abstract symbolic plumbing; we do NOT want the fact that we assigned a *symbolic name* to an expression's result, to infect the *result itself* by changing it's site!

Therefore, in Electric v3, `e/client` and `e/server` forms only influence the site of actual Clojure/Script interop calls. Plumbing and abstraction forms such as `let` and `e/fn` no longer have any significance in determining network transfer. `e/client` and `e/server` have no impact on these expressions.

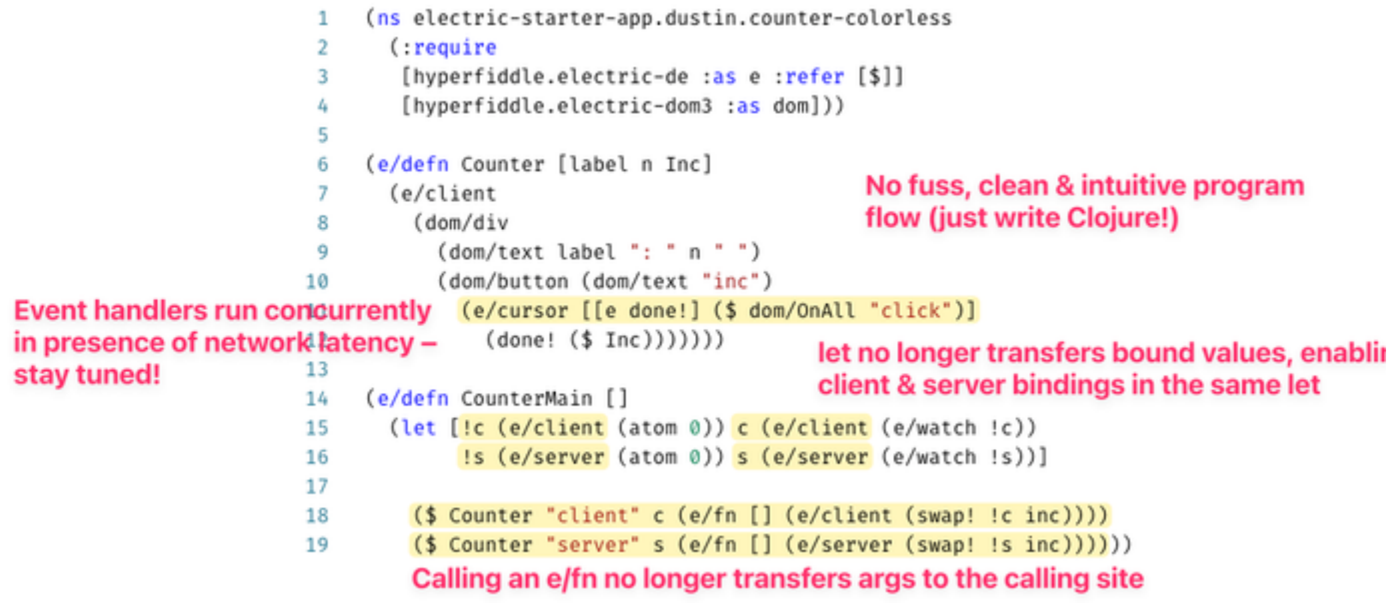
Summary: "Interop is sited, edges are not"

- **interop** = clojure/script interop, e.g. `atom` , `swap!` , `println` , `dom/text` , `vector` , `inc` , `clojure.core/+` , `e/watch` , `clojure.core/fn`
  - Any and all clojure/script calls, not just side effecting calls (`println` , `dom/text`) but also pure calls (`inc` , `vector` , `mapv`)
  - Another word for clojure/script interop is **"platform effects"** = any computation that ultimately happens on the host platform (java or js). This includes computations over collections, strings, dates, numerics!
- **edges** = points of sharing in the reactivity graph, e.g. `let` , `e/fn` arguments, `binding`
  - the purpose of `let` , `binding` etc in Electric is to bind a symbolic name to an

expression so that its result can be efficiently shared and reused across multiple downstream consumers.

- The ultimate consumer that forces resolution of these bindings is a platform effect, and that effect is either on the same site (no transfer) or on a different site (needing transfer). And as the programmer, the new Electric v3 semantics put you *entirely* in control of this.

Electric v3 example – simple counter



```clojure
(ns electric-starter-app.dustin.counter-colorless
  (:require
   [hyperfiddle.electric-de :as e :refer [$]]
   [hyperfiddle.electric-dom3 :as dom]))

(e/defn Counter [label n Inc]
  (e/client
    (dom/div
      (dom/text label ": " n " ")
      (dom/button (dom/text "inc")
        (e/cursor [[e done!] ($ dom/OnAll "click")]
          (done! ($ Inc)))))))

(e/defn CounterMain []
  (let [!c (e/client (atom 0)) c (e/client (e/watch !c))
        !s (e/server (atom 0)) s (e/server (e/watch !s))]

    ($ Counter "client" c (e/fn [] (e/client (swap! !c inc))))
    ($ Counter "server" s (e/fn [] (e/server (swap! !s inc))))))
```

No fuss, clean & intuitive program flow (just write Clojure!)

Event handlers run concurrently in presence of network latency – stay tuned!

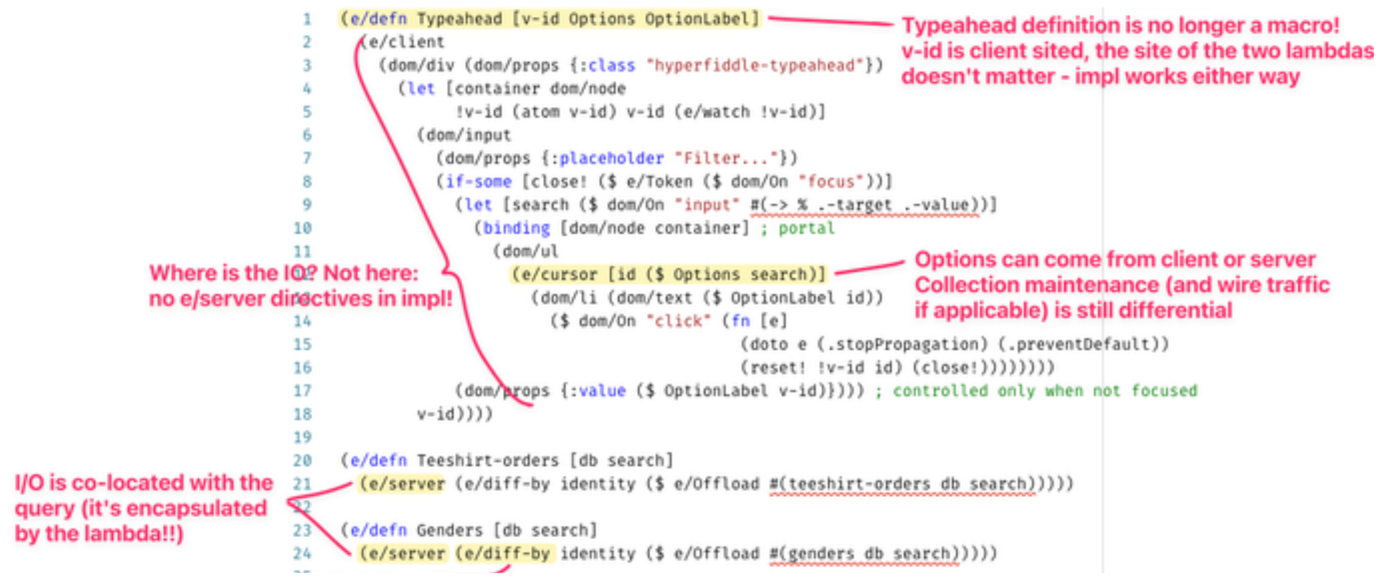
let no longer transfers bound values, enabli client & server bindings in the same let

Calling an e/fn no longer transfers args to the calling site

As of this writing, Electric v3 fns are called with `$` not `new`. Yes, the old syntax was very cute and highlighted an illuminating symmetry with OOP. But on the other hand, the overloaded syntax made it difficult to give a good error message when illegally attempting to call an Electric function from Clojure function. We'll see where we end up.

Example: Typeahead picker backed by server query

0:00

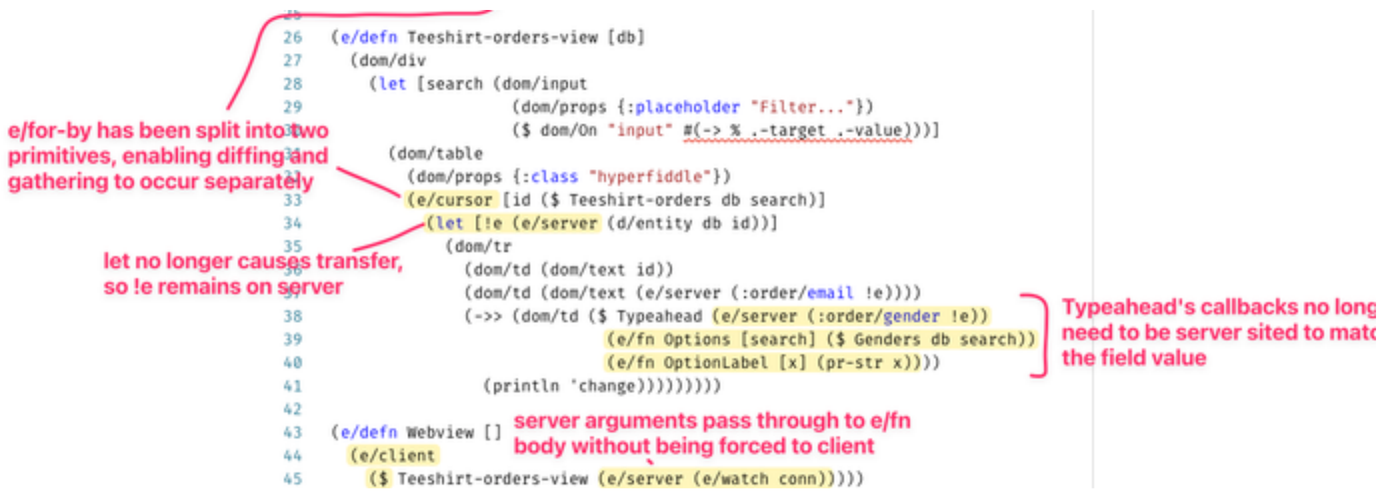


```clojure
(e/defn Typeahead [v-id Options OptionLabel]
  (e/client
    (dom/div (dom/props {:class "hyperfiddle-typeahead"})
      (let [container dom/node
            !v-id (atom v-id) v-id (e/watch !v-id)]
        (dom/input
          (dom/props {:placeholder "Filter..."})
          (if-some [close! ($ e/Token ($ dom/On "focus"))]
            (let [search ($ dom/On "input" #(-> % .-target .-value))]
              (binding [dom/node container] ; portal
                (dom/ul
                  (e/cursor [id ($ Options search)]
                    (dom/li (dom/text ($ OptionLabel id))
                      ($ dom/On "click" (fn [e]
                                          (doto e (.stopPropagation) (.preventDefault))
                                          (reset! !v-id id) (close!))))))))
            (dom/props {:value ($ OptionLabel v-id)}))) ; controlled only when not focused
        v-id))))

(e/defn Teeshirt-orders [db search]
  (e/server (e/diff-by identity ($ e/Offload #(teeshirt-orders db search)))))

(e/defn Genders [db search]
  (e/server (e/diff-by identity ($ e/Offload #(genders db search)))))
```

Typeahead definition is no longer a macro!
v-id is client sited, the site of the two lambdas doesn't matter - impl works either way
Where is the IO? Not here: no e/server directives in impl!
Options can come from client or server Collection maintenance (and wire traffic if applicable) is still differential
I/O is co-located with the query (it's encapsulated by the lambda!!)



```
(e/defn Teeshirt-orders-view [db]
  (dom/div
    (let [search (dom/input
                   (dom/props {:placeholder "Filter..."})
                   ($ dom/On "input" #(-> % .-target .-value)))]
      (dom/table
        (dom/props {:class "hyperfiddle"})
        (e/cursor [id ($ Teeshirt-orders db search)]
          (let [!e (e/server (d/entity db id))]
            (dom/tr
              (dom/td (dom/text id))
              (dom/td (dom/text (e/server (:order/email !e))))
              (->> (dom/td ($ Typeahead (e/server (:order/gender !e))
                                (e/fn Options [search] ($ Genders db search))
                                (e/fn OptionLabel [x] (pr-str x))))
                (println 'change)))))))))

(e/defn Webview []
  (e/client
    ($ Teeshirt-orders-view (e/server (e/watch conn)))))
```

e/for-by has been split into two primitives, enabling diffing and gathering to occur separately

let no longer causes transfer, so !e remains on server

Typeahead's callbacks no long
need to be server sited to matc
the field value

server arguments pass through to e/fn body without being forced to client

Gist with full typeahead example here

FAQ: When launch? No hard date yet, we're currently upgrading all our apps and fixing regressions as we find them. Soon!