Clojure SPAs with re-frame

Artem Chernyak

2020

Groundwork

- ClojureScript
- Hiccup
- React
- Reagent

ClojureScript

- Mostly Clojure
- Runs on node
- Runs in browser
- Reagent

Hiccup for HTML

```
[:html
  [:h1 "Hello World"]]
```

- Supports basic html tags
- Auto closing tags
- Lisp syntax

Properties in Hiccup

```
[:html
  [:h1
    {:class "title"}
    "Hello World"]]

[:html
  [:h1.title "Hello World"]]
```

- Supports basic html tags
- Auto closing tags
- Lisp syntax

Clojure in Hiccup

```
[:ul
  (for [i (range 1 4)]
    [:li i])]
```

- Supports basic html tags
- Auto closing tags
- Lisp syntax

React

- JavaScript framework
- Component based
- Highly reusable abstractions
- Great performance

Clojure in Hiccup

```
(defn hello [name]
  [:h1  "Hello " name])

[:html
  [hello "World" ]]
```

- Use Hiccup
- Allows modularizing with components
- Work just like function

State management

```
(defn click-count (r/atom 0))

(defn counting-component []
  [:div
   "The atom " [:code "click-count"] " has value: "
   @click-count ". "
   [:input {:type "button" :value "Click me!"
            :on-click #(swap! click-count inc)}]])
```

- Uses clojure atom syntax
- Allows for dynamic components
- Automatically triggers re-render

Local state

```
(defn timer-component []
  (let [seconds-elapsed (r/atom 0)]
    (fn []
      (js/setTimeout #(swap! seconds-elapsed inc)
       1000)
      [:div
       "Seconds Elapsed: " @seconds-elapsed])))
```

- Only initializes when neede
- Isolated to the component
- Can share as needed
- Tight re-draw loop

Sharing state

```
(defn atom-input [value]
  [:input {:type "text"
           :value @value
           :on-change #(reset! value
                        (-> % .-target
                            .-value))}])

(defn shared-state []
  (let [val (r/atom "foo")]
    (fn []
      [:div
       [:p "The value is now: " @val]
       [:p "Change it here: " [atom-input val]]])))
```

Leaky React

```
(defn list [items]
  [:ul
    (for [item items]
      ^{:key item}
      [:li "Item " name])])
```

- Use Hiccup
- Allows modularizing with components
- Work just like function

Passing arguments gotcha

```
(defn timer-component [event]
  (let [seconds-elapsed (r/atom 0)]
    (fn [event]
      (js/setTimeout #(swap! seconds-elapsed inc)
                     1000)
      [:div
       "Seconds Elapsed since "
       event ": "
       @seconds-elapsed])))
```

- Required in both functions
- Breaks initial render if missing at top level
- Breaks subsequent renders if missing in return

Lazy isn’t always good

```
(defn list []
  [:ul
    (doall
      (for [i (range 1 4)]
        [:li i]))])
```

- Forces update
- Can cause intermittent bugs

Why re-frame?

- Data oriented
- State sharing
- Less decisions

The six domains

- Event dispatch
- Event handling
- Effect handling
- Query
- View
- DOM

Mental Model

- Events trigger `dispatchers`
- Dispatch handlers produce effect maps
- Effects modify db and external resources
- `subscribers` listen to changes on the db

Events

- User interactions
- Other events
- Time

Dispatching

```
(defn delete-button [item-id]
 [:button
  {:on-click #(re-frame.core/dispatch
               [:delete-item item-id])}])
```

- Just a function
- Can take extra parameters
- Every dispatch type must have a handler

Handling a Dispatch

```
(defn delete-item
  [{:keys [db]} [_ item_id]]
  {:db (dissoc-in db [:items item-id])})

(rf/reg-event-fx
 :delete-item
 delete-item)
```

- Always return effect maps
- Effects get automatically triggered
- Only effects can modify state

Effect handlers

```
(rf/reg-fx
 :db
 (fn [val]
  (reset! app-db val)))
```

- Performs some side effect
- Libraries can provide effects

Why?

- Easy testing
- Single point of control
- Force separation of concerns
- Your own DSL

Creating subscribers

```
(defn query-fn
  [db v]
  (:items db))

(rf/reg-sub
  :query-items
  query-fn)
```

- Works like a view over the db
- Can combine or modify data
- Allows the subscriber to get called from anywhere

Using subscribers

```
(defn items-view
  []
  (let [items @(rf/subscribe [:query-items])]
    [:div (map item-render items)]))
```

- Component automatically updates with subscriber
- Subscribers can take arguments

Subscribers that subscribe

```
(defn get-item-fn
  [items [_ id]]
  (get items id))

(rf/reg-sub
  :get-item
  :<- [:items]
  query-fn)
```

- Automatically update when underlying subscribers change
- Allows incrementally computing computations

Coeffects

```
(rf/reg-cofx
  :now
  (fn [coeffects _]
     (assoc coeffects :now (js.Date.))))

(rf/reg-event-fx
  :dont-care
  [(inject-cofx :now)]
  (fn [{:keys [db now]}]
    ...))
```

- Allows getting things from the outside re-frame
- Keeps re-frame functions pure
- easy testing

Make testing easier

```
(defmacro reg-sub [id & args]
  `(do
     (def ~(-> id name symbol) ~(last args))
     (rf/reg-sub
      ~id
      ~@args)))
```

- Automatically creates a function for testing
- Keeps function and registration together

Always namespace

- Create with `::event`
- Reference through named import `:my/event`
- Clean organization, easy navigation

Database recommendations

- Keep things in maps by unique id
- Transform as needed in subscribers
- Do not copy whole objects, use the ids