ClojureScript

interfaces to React

Michiel Borkent
@borkdude
Øredev, November 6th 2014

Michiel Borkent (@borkdude)

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

Full Clojure stack example @ Finalist

Commercial app.

Fairly complex UI

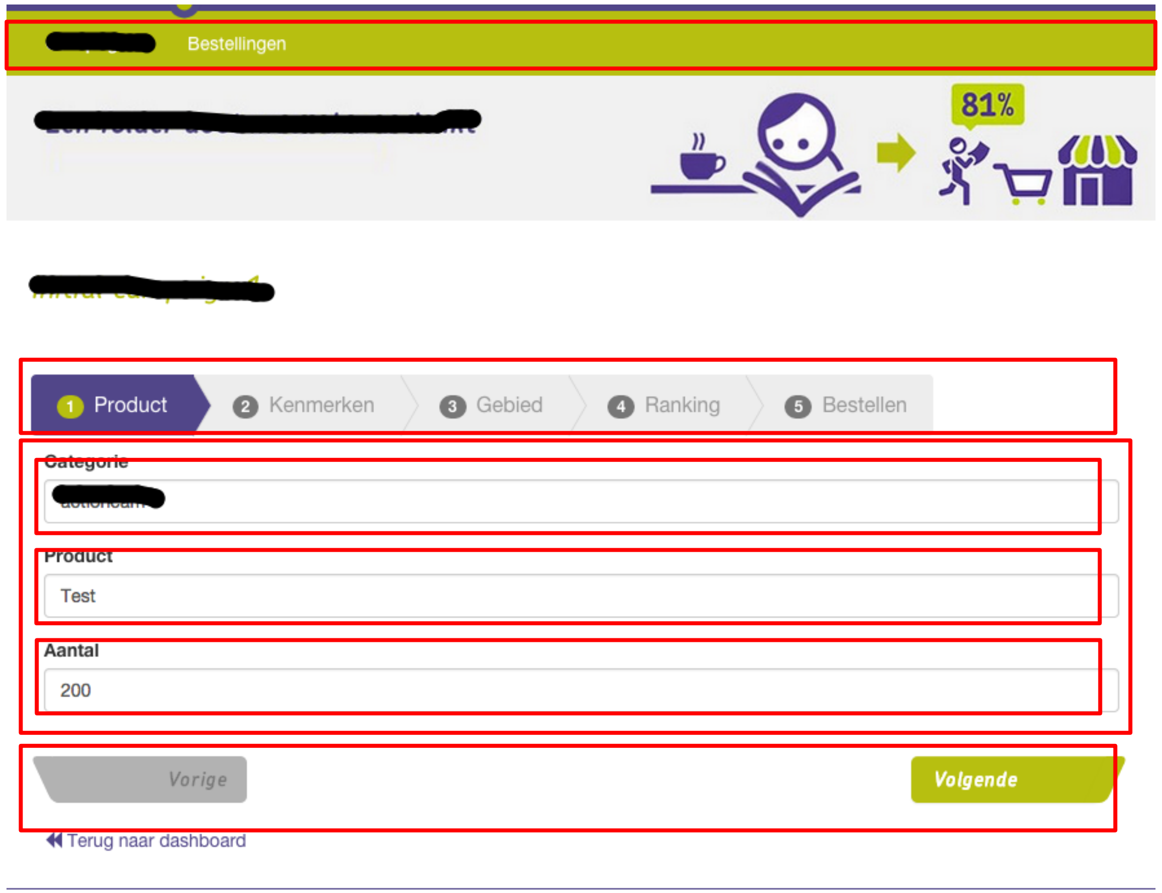
- Menu: 2 "pages"

Page 1:

Dashboard. Create new or select existing entity to work on.

Then:

- Wizard 1
  - Step 1..5
  - Each step has a component
- Wizard 1 - Step2
  - Wizard 2
    - Step 1'
    - Step 2'



Categorie
[redacted]

Product
Test

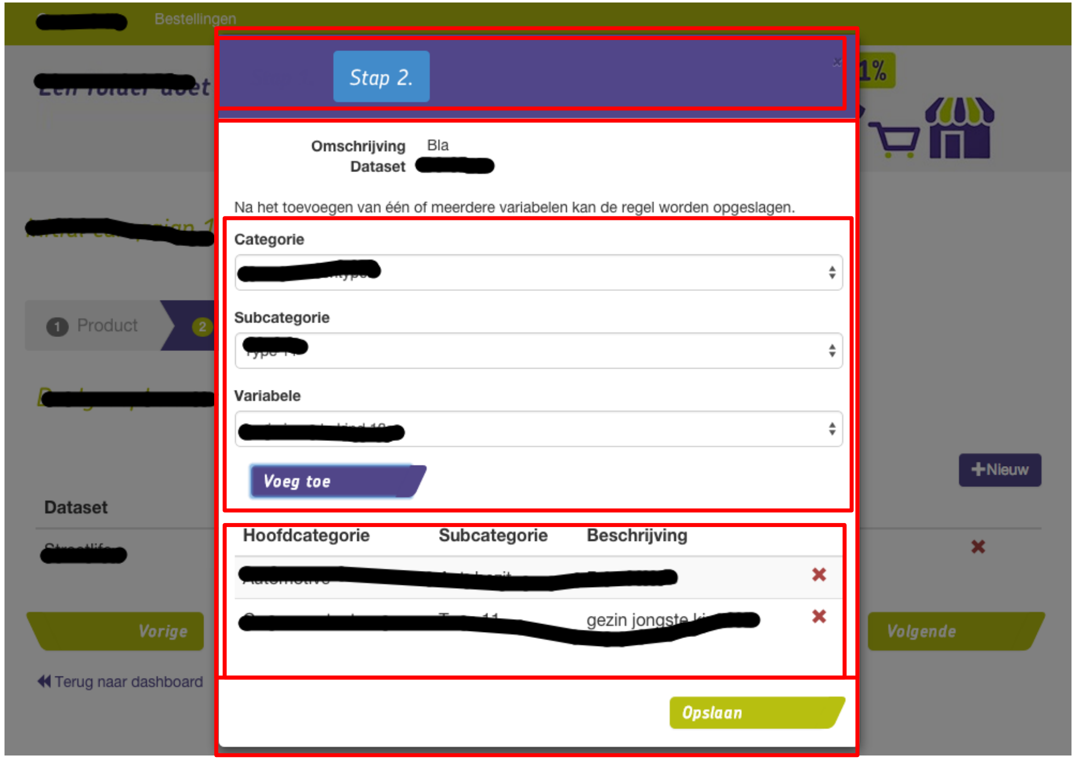
Aantal
200

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.



[redacted] Bestellingen

[redacted]

Stap 2.

1%

[redacted]

1 Product 2

[redacted]

Omschrijving Bla

Dataset [redacted]

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

Categorie

[redacted]

Subcategorie

[redacted]

Variabele

[redacted]

Voeg toe

Hoofdcategorie	Subcategorie	Beschrijving
[redacted]	[redacted]	[redacted]
[redacted]	[redacted]	gezin jongste [redacted]

+Nieuw

Dataset

[redacted]

Vorige

Volgende

Terug naar dashboard

Opslaan

Agenda

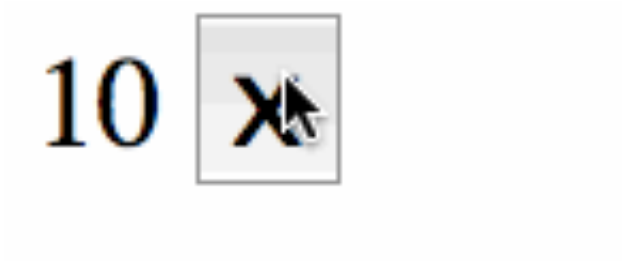
- What is React?
- Om
- Reagent

What is React?

React

- Developed by Facebook
- Helps building reusable and composable UI components
- Unidirectional Data Flow
- Less need for re-rendering logic
- Leverages virtual DOM for performance
- Can render on server to make apps crawlable

```
/** @jsx React.DOM */

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>;
    }
});

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



ClojureScript interfaces

Prior knowledge

```
(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
```

Before React: manual DOM edits

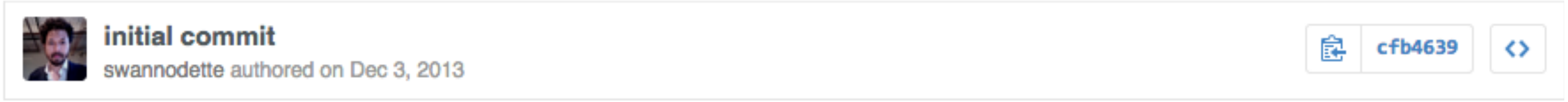
```
(add-watch greeting-form :form-change-key
           (fn [k r o n]
             (dispatch/fire :form-change {:old o :new n})))

(dispatch/react-to #{:form-change}
                   (fn [_ m]
                     (doseq [s (form-fields-status m)]
                       (render-form-field s))
                     (render-button [(-> m :old :status)
                                     (-> m :new :status)] )))
```

source: http://clojurescriptone.com/documentation.html

ClojureScript interfaces

Om - David Nolen

★ Star 2,917



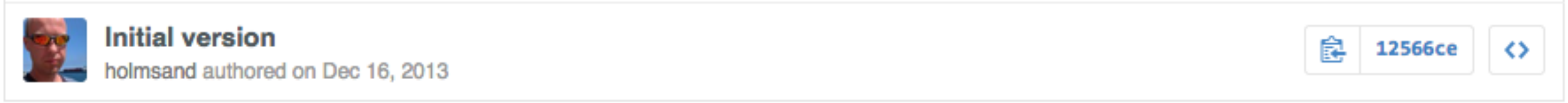
initial commit

swannodette authored on Dec 3, 2013

cfb4639

Reagent (was: Cloact) - Dan Holmsand

★ Star 651



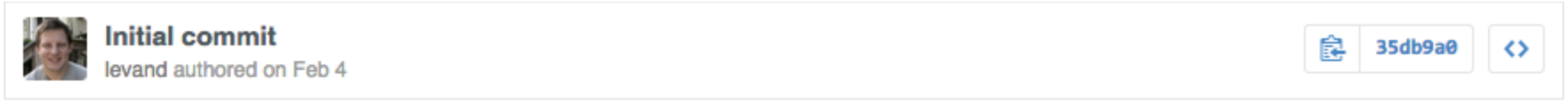
Initial version

holmsand authored on Dec 16, 2013

12566ce

Quiescent - Luke vanderHart

★ Star 262



Initial commit

levand authored on Feb 4

35db9a0

React + ClojureScript

Both Om and Reagent leverage:

- immutability for faster comparison in `shouldComponentUpdate`
- Fewer redraws by batching updates with `requestAnimationFrame`

Om

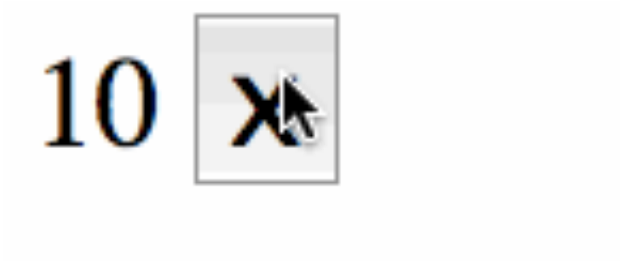
- Opinionated library by David Nolen
- One atom for app state
- Props: narrowed scope of app state (cursor)

```
(def app-state (atom {:counter1 {:count 10}
                      :counter2 {:count 11}}))

(defn main [app owner]
  (om/component
   (dom/div nil
            (om/build counter (:counter1 app))
            (om/build counter (:counter2 app)))))
```

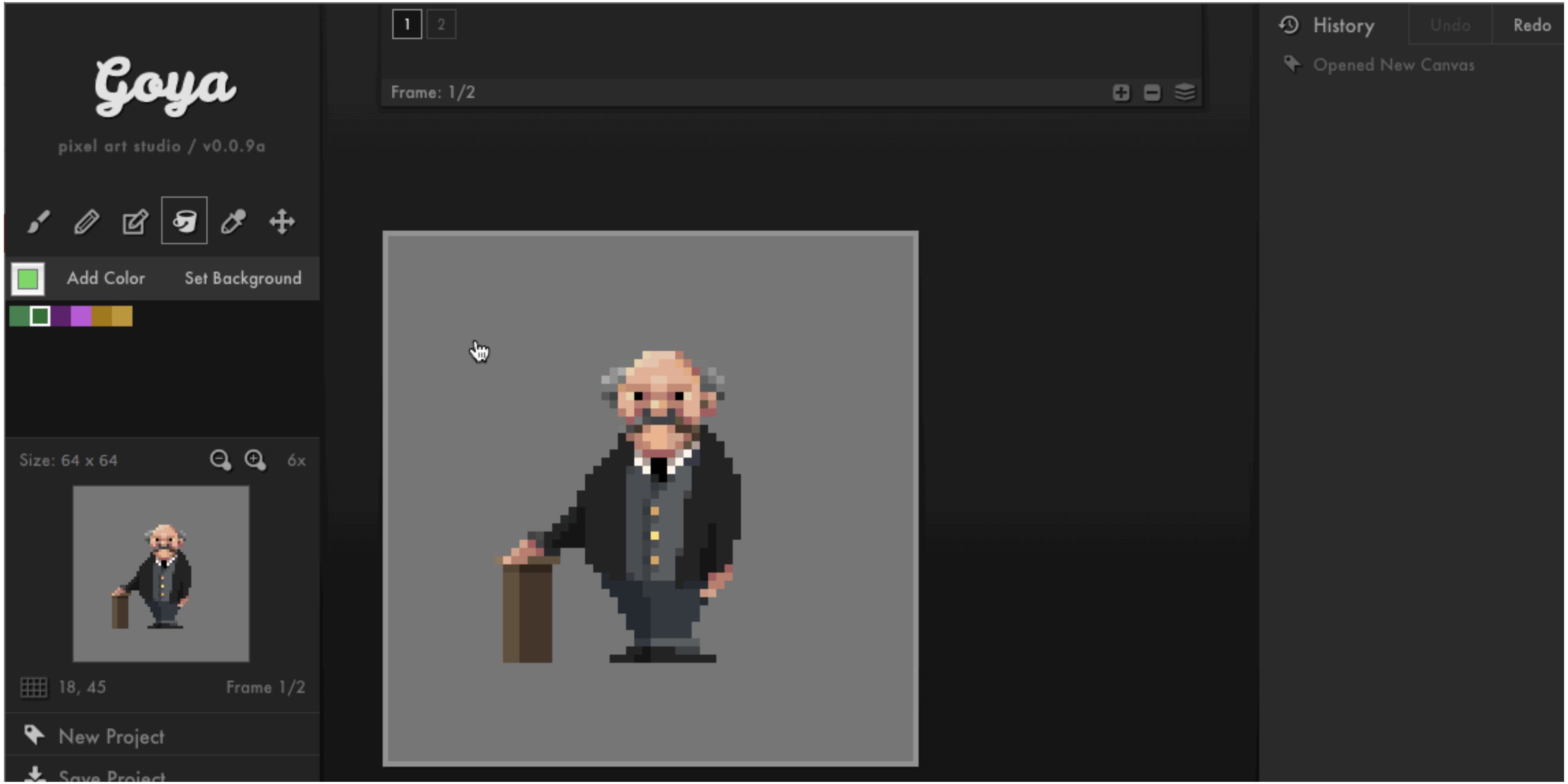
Om

- Communication between components via
  - setting init-state / state (parent -> child)
  - callbacks (child -> parent)
  - app-state
  - core.async
- Explicit hooks into React lifecycle via ClojureScript protocols
- Follows React semantics closely (e.g. local state changes cause re-rendering)

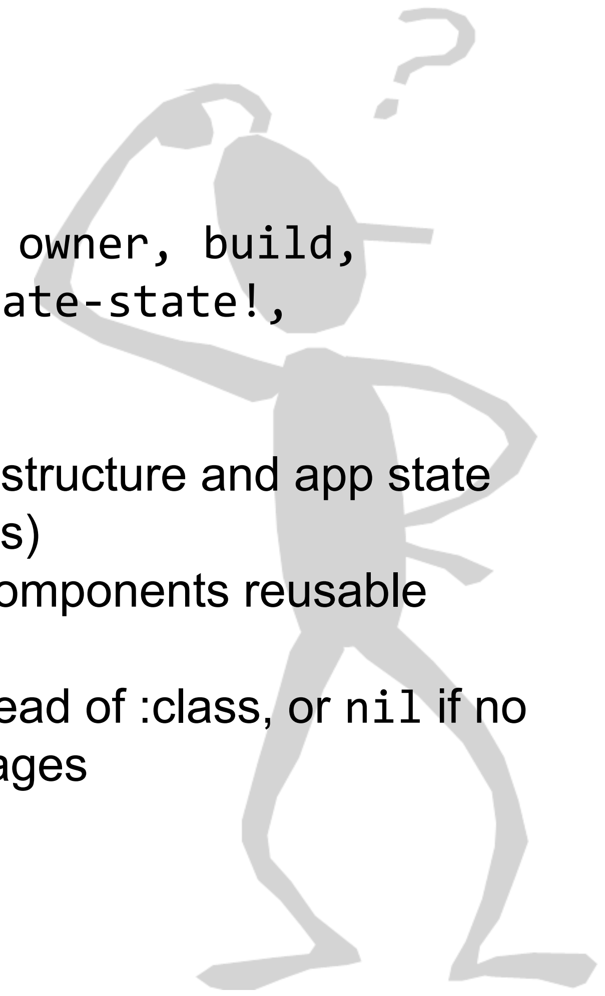
```
(def app-state (atom {:counter 10}))

(defn app-state-counter [app owner]
  (reify
    om/IRender
    (render [_]
      (dom/div nil
               (:counter app)
               (dom/button
                #js {:onClick
                      #(om/transact! app :counter inc)}
                "x")))))
(om/root
 app-state-counter
 app-state
 {:target (. js/document (getElementById "app"))})
```



Goya pixel editor



Some catches



- Large vocabulary around cursors: `app(-state)`, `owner`, `build`, `cursors`, `ref-cursors`, `root`, `update!`, `update-state!`, `transact!`, `opts`
- Cursor behaves differently depending on lifecycle
- Strong correspondence between component tree structure and app state structure (`ref-cursors` are supposed to solve this)
- Heavy use of callbacks or `core.async` to make components reusable (should not rely on `app-state`)
- Ommission of `#js` reader literal, `:className` instead of :class, or `nil` if no attributes used, fails silently or cryptic error messages

Reagent

Reagent

Uses `RAtoms` for state management

Components are 'just functions'™ that

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

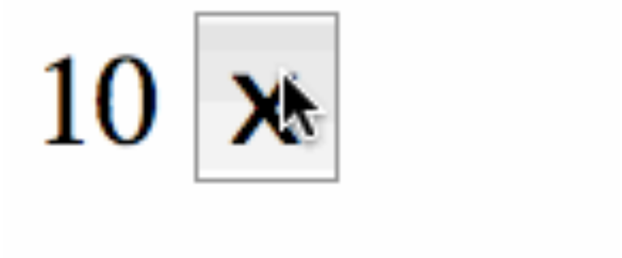
Reagent

- Components should be called like `[component args]` instead of `(component args)`
- Components are re-rendered when
  - props (args) change
  - referred `RAtoms` change
- Hook into React lifecycle via metadata on component functions

```
(def component
  (with-meta
    (fn [x]
      [:p "Hello " x ", it is " (:day @time-state)])
    {:component-will-mount #(println "called before mounting")
     :component-did-update #(js/alert "called after updating")} ))
```

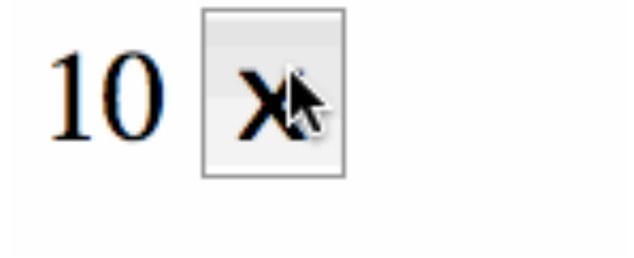
RAtom

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

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

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

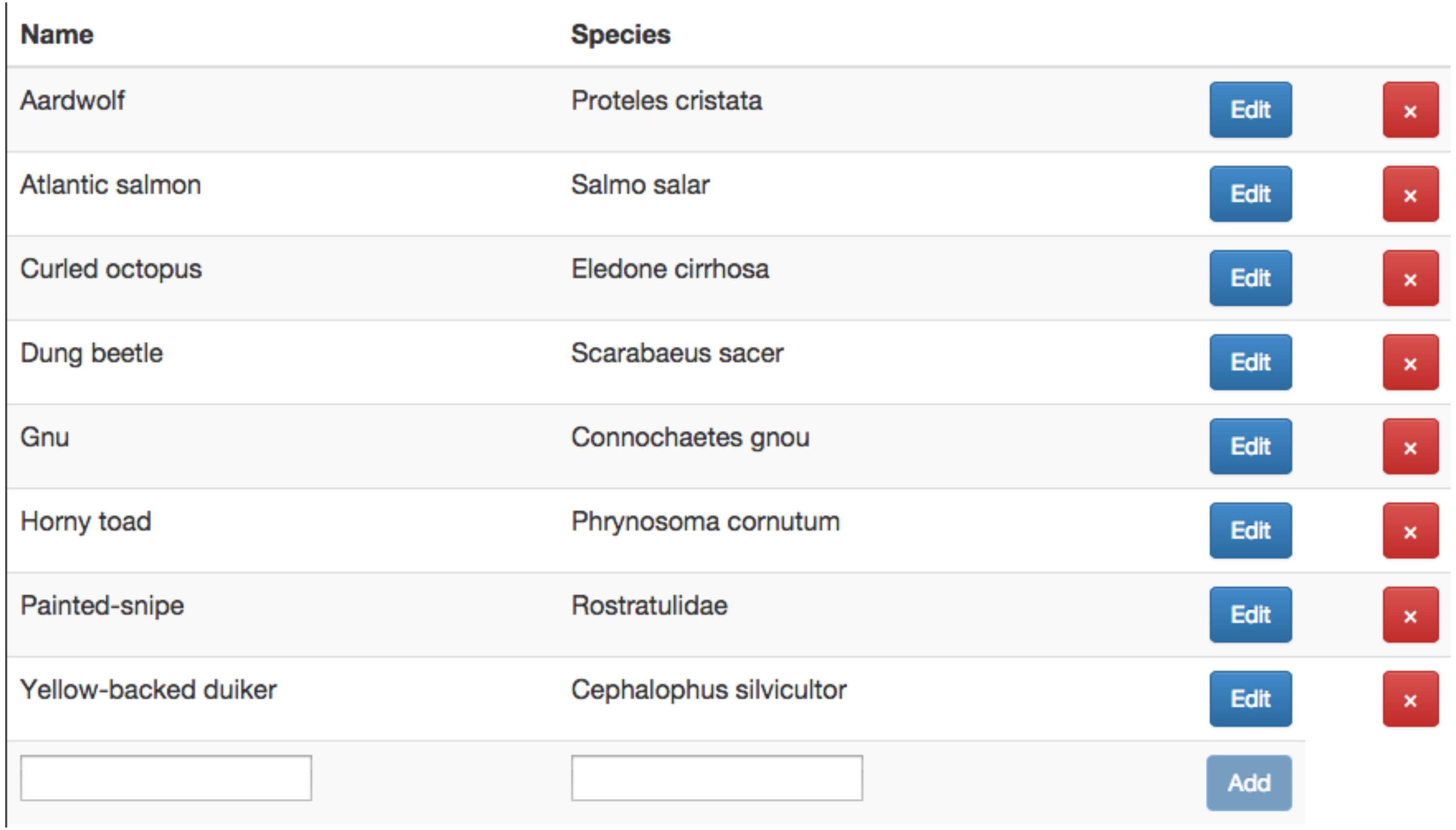


```
(defn local-counter [start-value]
  (let [count-state (atom start-value)]
    (fn []
      [:div
       @count-state
       [:button {:on-click #(swap! count-state inc)}
        "x"]])))

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

local
RAtom



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	×
		Add	

RAtom with set containing animal hash-maps

```
(def animals-state (atom #{}))

(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"}
```

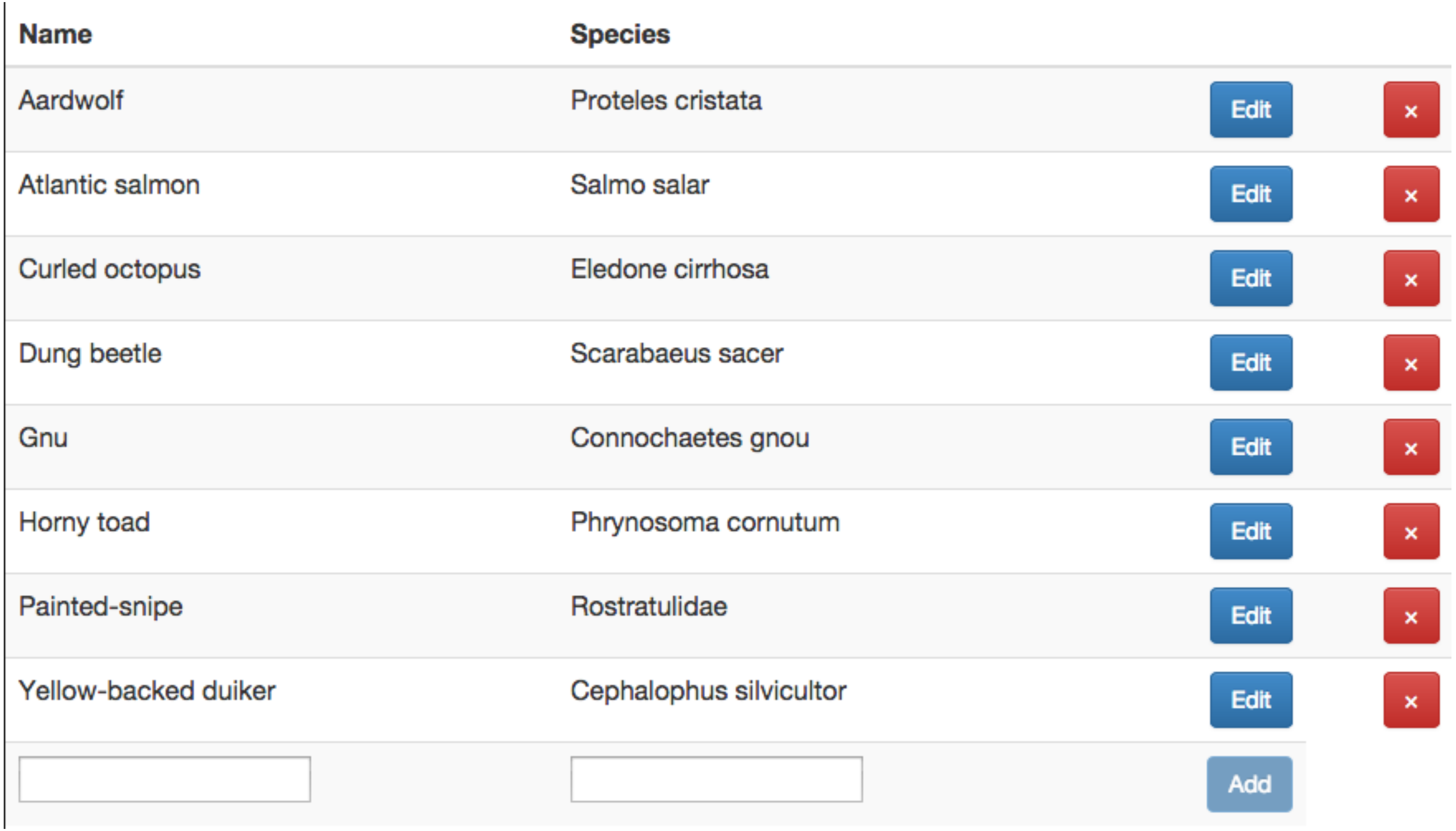
Render all animals from state

```
(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]]]])
```

key needed for React to keep track of rows

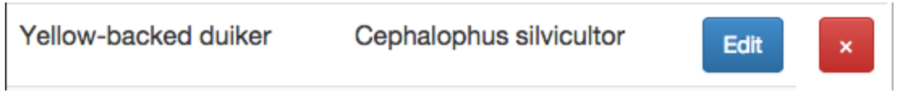
a row component for each animal

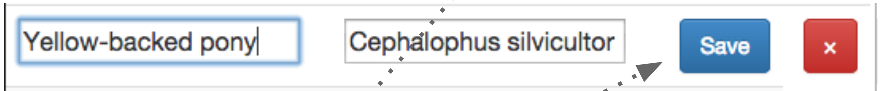
form to create new animal



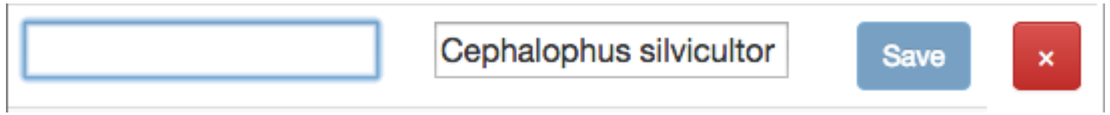
Name	Species		
Aardwolf	Proteles cristata	Edit	×
Atlantic salmon	Salmo salar	Edit	×
Curled octopus	Eledone cirrhosa	Edit	×
Dung beetle	Scarabaeus sacer	Edit	×
Gnu	Connochaetes gnou	Edit	×
Horny toad	Phrynosoma cornutum	Edit	×
Painted-snipe	Rostratulidae	Edit	×
Yellow-backed duiker	Cephalophus silvicultor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

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


Yellow-backed duiker | Cephalophus silvicultor | Edit | ×


Yellow-backed pony | Cephalophus silvicultor | Save | ×

```clojure
(defn field-input-handler
  "Returns a handler that updates value in atom map,
  under key, with value from onChange event"
  [atom key]
  (fn [e]
    (swap! atom
           assoc key
           (.. e -target -value))))

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

(defn editable-input [atom key]
  (if (:editing? @atom)
    [:input {:type     "text"
             :value    (get @atom key)
             :onChange (field-input-handler atom key)}]
    [:p (get @atom key)]))
```



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

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

```clojure
(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
                   (set (filter (fn [other]
                                  (not= (:id other)
                                        (:id a)))
                                old-state))
                   updated-animal))))))
```

replace updated animal retrieved from server

Live demo

If you want to try yourself. Code and slides at:

https://github.com/borkdude/oredev2014

My experience with Om and Reagent

- Both awesome
- Added value to React
- Om encourages snapshot-able apps but:
  - surprises
  - large vocabulary
- Reagent
  - easy to learn and use
  - readable