Web-Applikationen und UIs funktional programmieren mit Reacl

Michael Sperber
@sperbsen

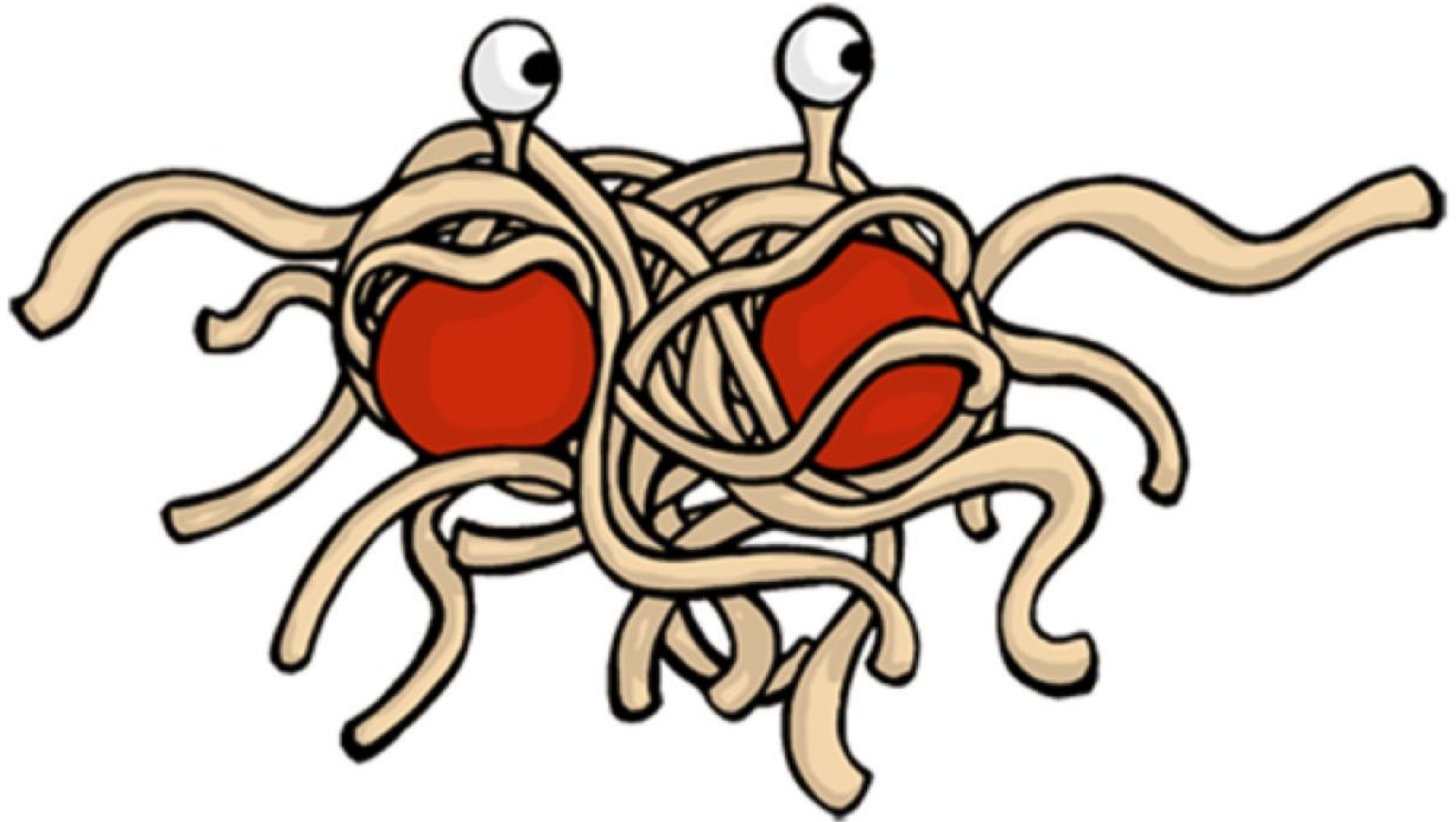
- software project development
- in many fields
- Scala, Clojure, Erlang, Haskell, F#, OCaml
- training, coaching
- co-organize BOB conference

www.active-group.de

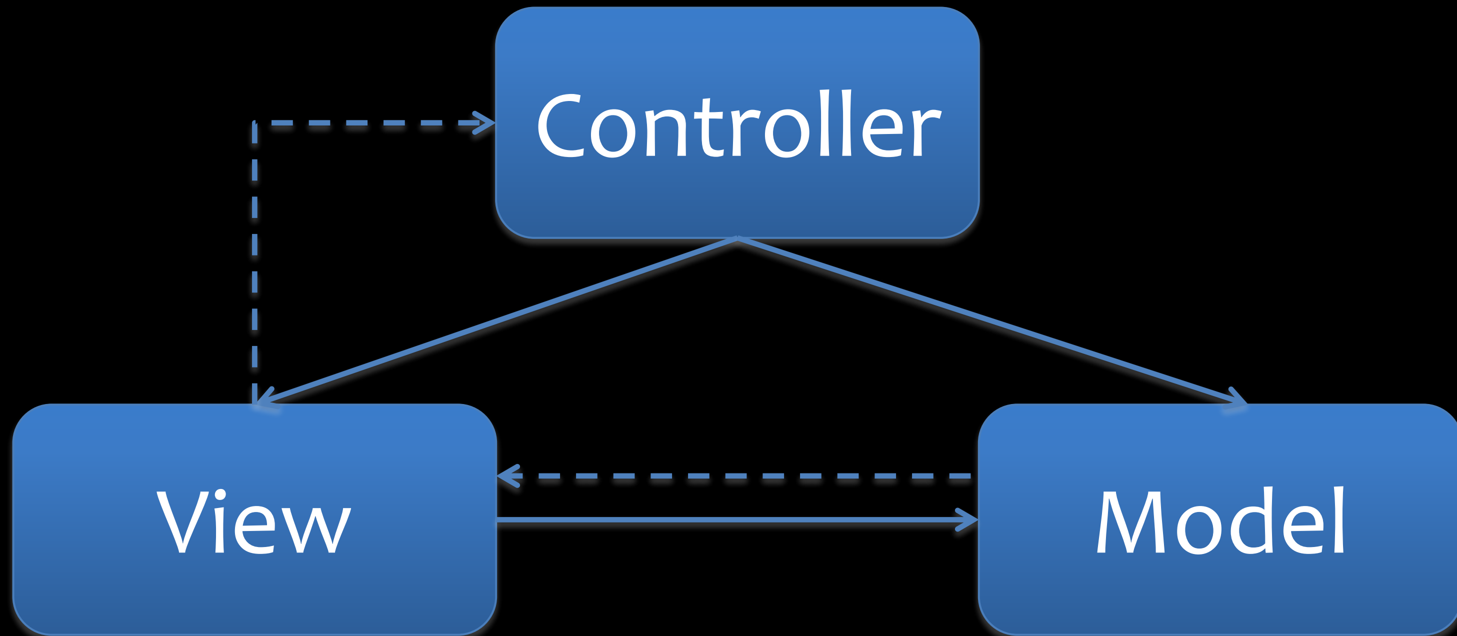
funktionale-programmierung.de

Everybody's Next GUI

MVC

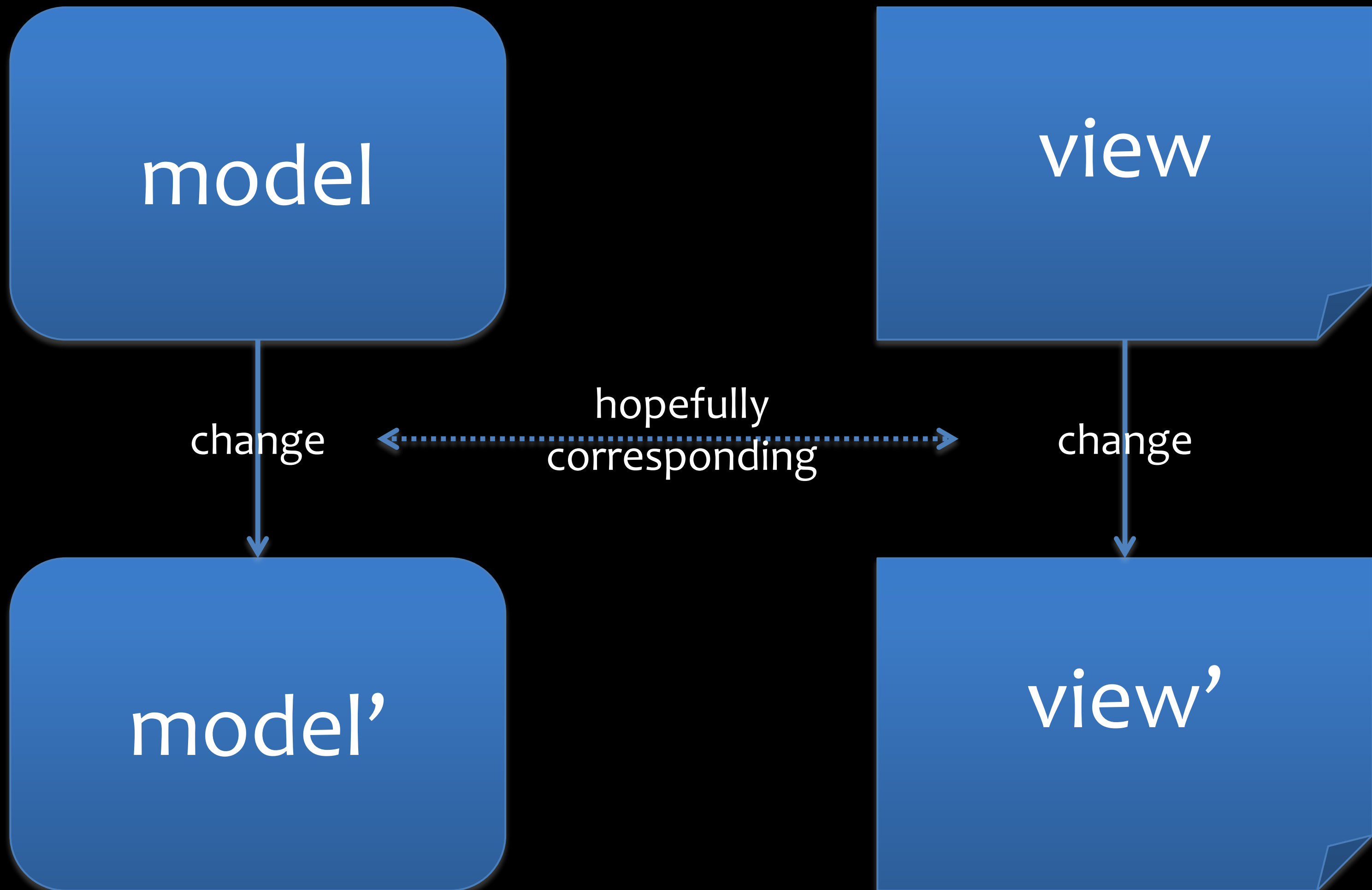


MVC



@rtive group

Problem



@rtive group

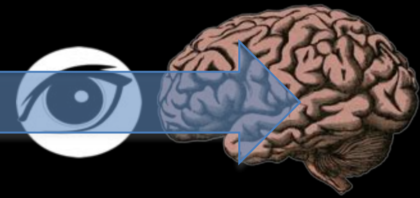
A World of Objects

@rlive group

Imperative Programming

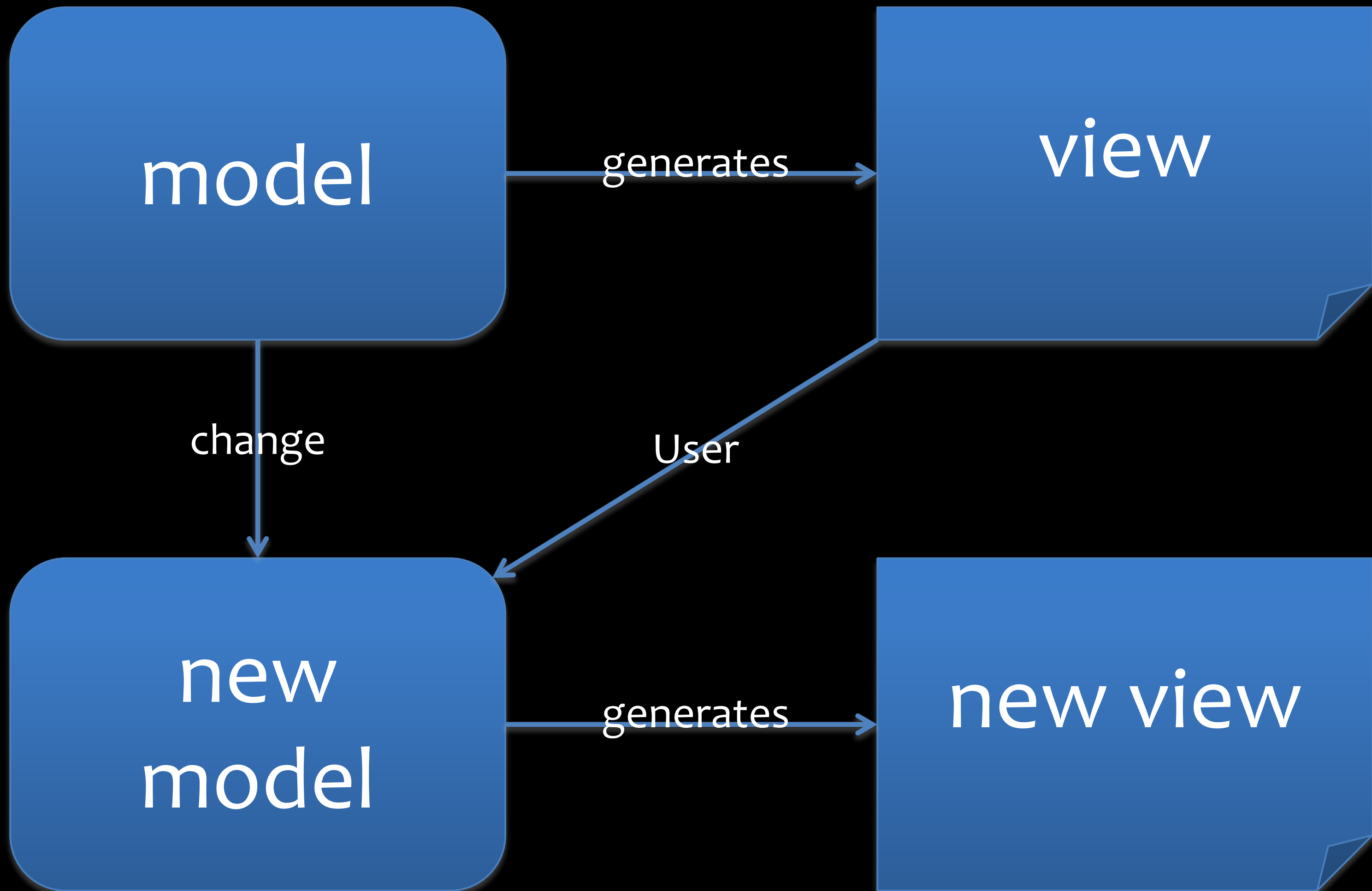
```
room1.exit(elephant)
hallway.enter(elephant)
hallway.exit(elephant)
room2.enter(elephant)
```

Reality and Snapshots



@rtive group

React



@ctive group

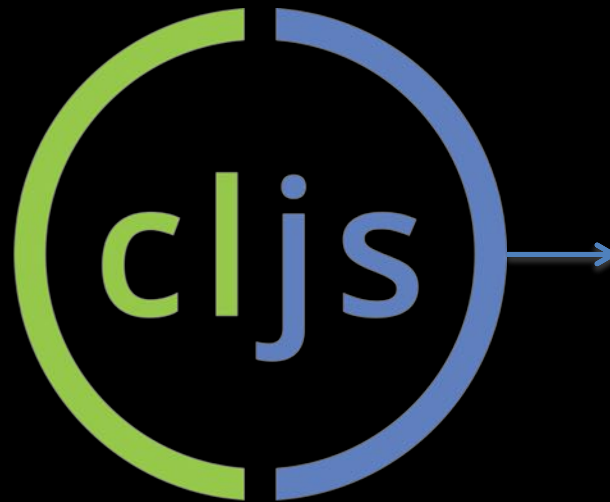
React Gripes

- .setState
- conflates app state with transient GUI state
- props object instead of separate arguments
- props and state must be JS hashmaps
- `#js`
- implicit binding of `this`
- refs

@rtive group

Clojure

- Lisp
- funktional
- JVM



JS

@rtive group

Clojure

```
15
true
false
"foo"
(+ 1 2)
(+ 1 (* 2
        (+ 17 21)))
```

@ctive group

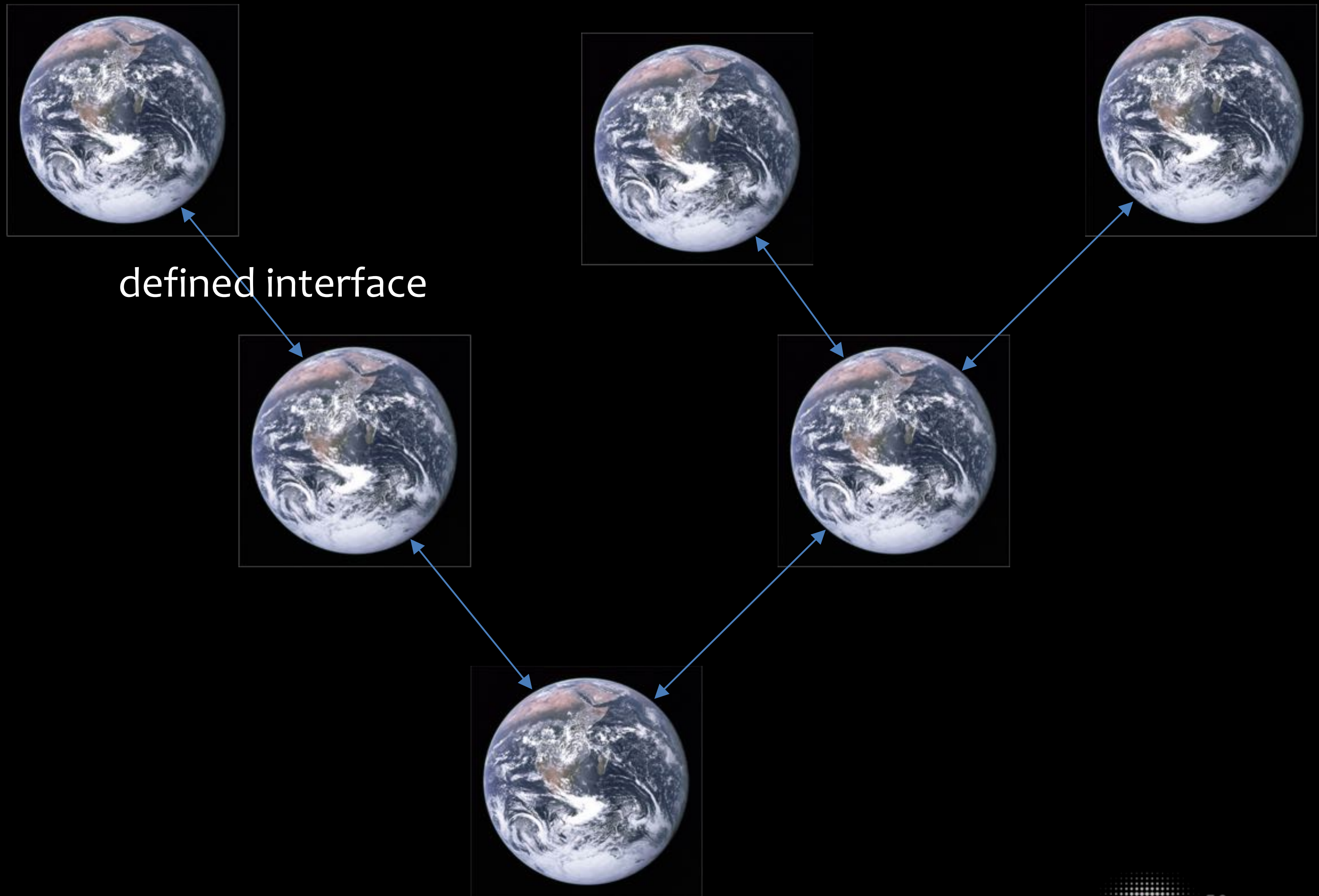
Clojure

```
(def pi 3.14159265)

(defn circumference
  [r]
  (* 2 pi r))
```

@ctive group

Reacl Component Tree



@rtive group

TODO

- [ ] Zap Make money
- [ ] Zap retire

die Add #2

@rtive group

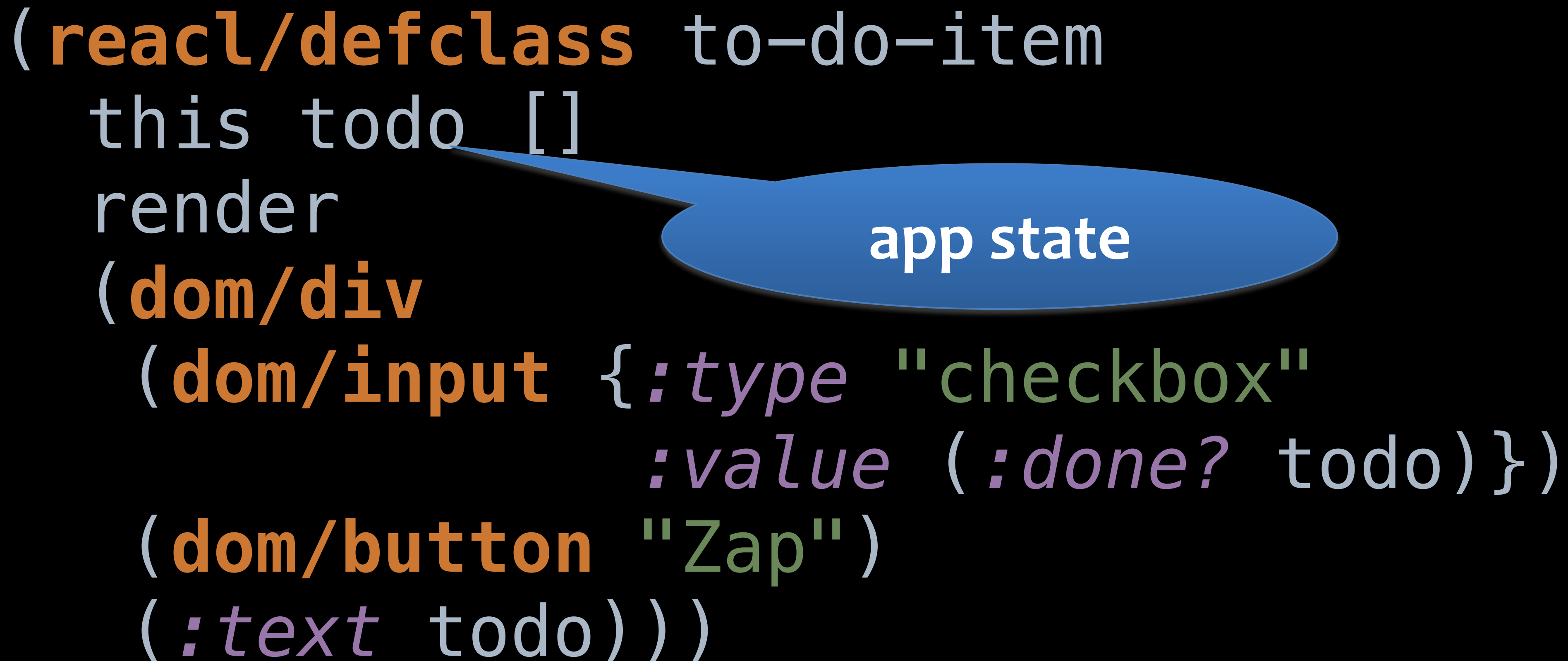
Application State

```
(defrecord Todo
  [id text done?])
(def t1 (->Todo 0 “Make money” false))
(def t2 (->Todo 1 “retire” false))

(defrecord TodosApp
  [next-id todos])
(def ts (->TodosApp 2 [t1 t2]))
```

@ctive group

Single Todo



@ctive group

Check Todo

```
(reacl/defclass to-do-item
  this todo []
  render
  (dom/div
   (dom/input {:type "checkbox"
               :value (:done? todo)
               :onchange
               (fn [e] ... )})
   (dom/button "Zap")
   (:text todo)))
```

@ctive group

Check Todo

```
(reacl/defclass to-do-item
  this todo []
  render
  (dom/div
   (dom/input {:type "checkbox"
               :value (:done? todo)
               :onchange
               (fn [e]
                 (reacl/send-message!
                   this
                   ...))})
   (dom/button "Zap")
   (:text todo)))
```

@ctive group

Check Todo

```
(reacl/defclass to-do-item
  this todo [parent]
  render
  (dom/div (dom/input
            {:type "checkbox"
             :value (:done? todo)
             :onchange
             (fn [e]
              (reacl/send-message! this
               (.. e -target -checked)))})
           (dom/button "Zap")
           (:text todo))
```

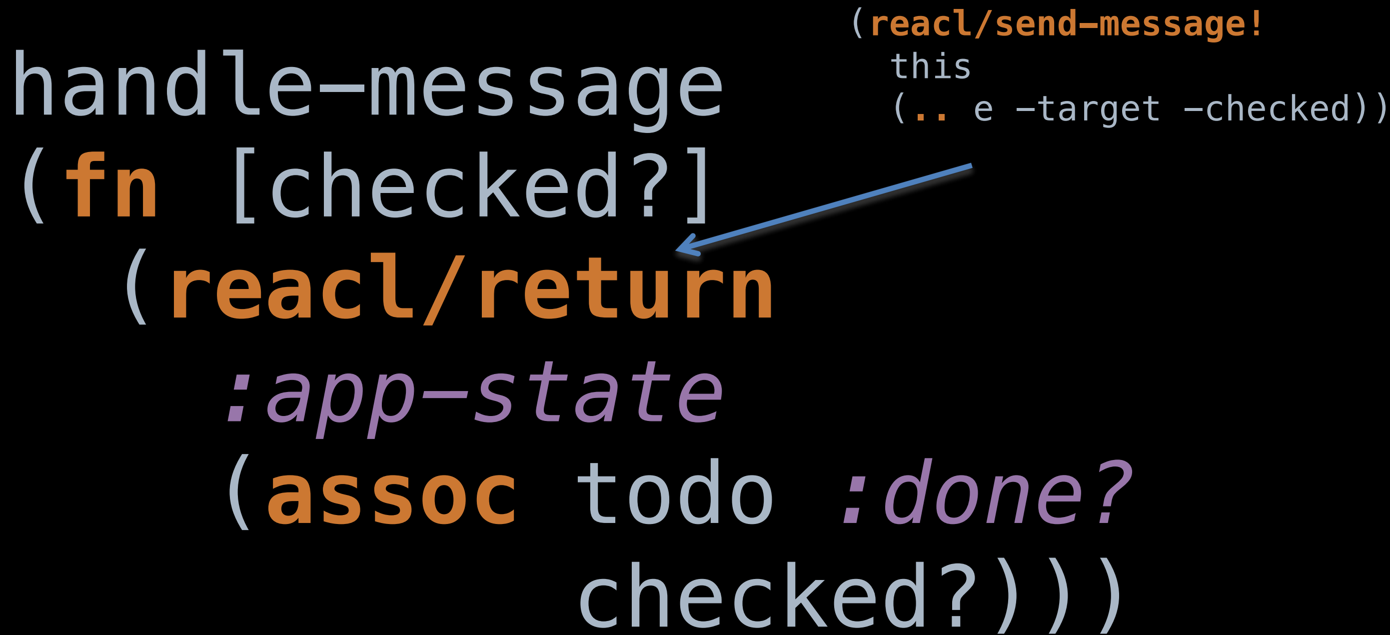
@active group

Handle Checked Message

```
(reacl/defclass to-do-item
  this todo [parent]
  mixins [(mix parent)]
  render
  (dom/div (dom/input
            {:type "checkbox"
             :value (:done? todo)
             :onchange
             (fn [e]
               (reacl/send-message! this
                  (.. e -target -checked)))})
           (dom/button "Zap")
           (:text todo))
  handle-message
  (fn [checked?]
    (reacl/return :app-state
                  (assoc todo :done? checked?))))
```

@ctive group

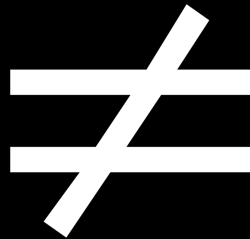
Handle Checked Message



Pure Functions

id	5
text	retire
done?	false

`(assoc :done? true)`

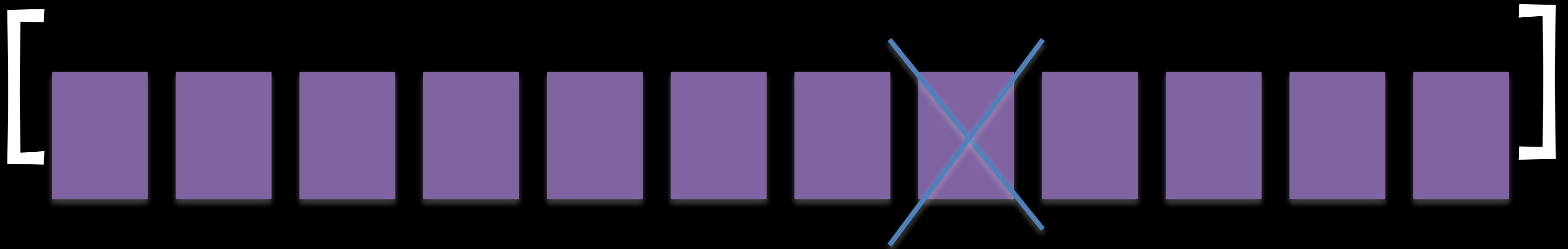


id	5
text	retire
done?	true

@ctive group

Zap Todo

```
(dom/button "Zap")
```



@ctive group

Zap Todo

```
(defrecord Delete [todo])
(dom/button
 {:onclick
  (fn [_]
    (reacl/send-message!
      parent (->Delete todo)))}
 "Zap")
```

@ctive group

Parent Parameter

```
(reacl/defclass to-do-item
  this todo [parent]
  …)
```

@ctive group

Local State

TODO

Zap Make money
Zap retire
die
Add #2

local state

@active group

Application State vs. Local State

```
(reacl/defclass to-do-app
  this app-state []
  local-state [local-state “”]
  render
  (dom/div
   (dom/h3 "
   ...))
```

app state

local state

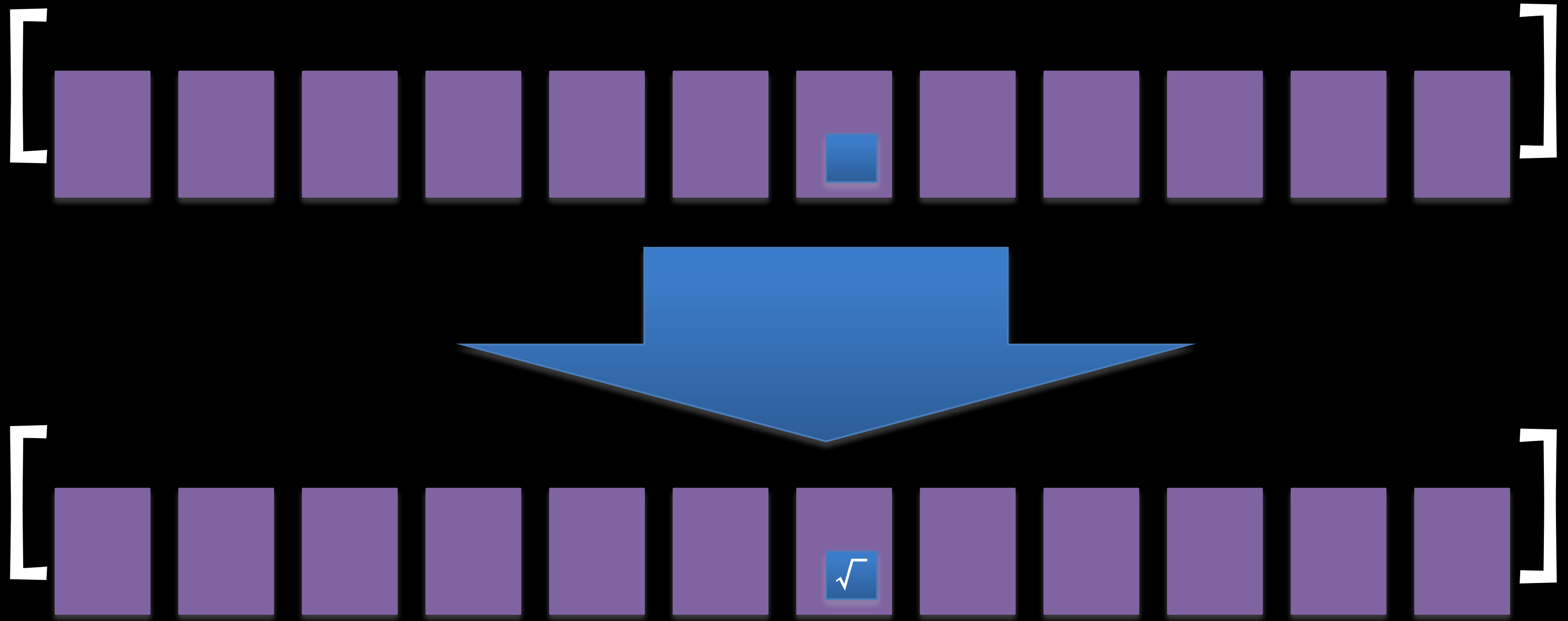
Todo App

```
(defrecord TodosApp
  [next-id todos])


(defrecord NewText [text])
(defrecord Submit [])
(defrecord Change [todo])
(defrecord Delete [todo])
```

active group

Immutable Data



@active group

Todo App

```
(reacl/defclass to-do-app
  this app-state []
  local-state [local-state “”]
  render
  (dom/div
   (dom/h3 "TODO")
   (dom/div
    (map (fn [todo]
           (dom/keyed (str (:id todo))
                      (to-do-item
                       (reacl/opt :embed-app-state
                                  embed-changed-todo)
                      todo
                       this)))
         (:todos app-state)))
   ...)))
```

@active group

Instantiating a Reacl Element

```
(to-do-item
 (reacl/opt ...)
 todo this))
```

@rtive group

Reactions

```
(reacl/opt :embed-app-state
           embed-changed-todo)

(defn embed-changed-todo
  [app-state changed-todo]
  (let [changed-id (:id changed-todo)]
    (assoc app-state
           :todos (map (fn [todo]
                         (if (= changed-id (:id todo))
                           changed-todo
                           todo))
                       (:todos app-state)))))
```

@active group

TODO

- ☐ Zap Make money
- ☐ Zap retire

die | Add #2

@rtlive group

New Todos

```
(reacl/defclass to-do-app
  this app-state []
  local-state [local-state “”]
  render
  (dom/div

    ...
    (dom/form
     {:onSubmit (fn [e]
                  (.preventDefault e)
                  (reacl/send-message! this
                    (->Submit)))}
     ...
     (dom/button
      (str "Add #" (:next-id app-state))))
```

@ctive group

New Todos

```
handle-message
(fn [msg]
  (cond
   ...
   (instance? Submit msg)
   (let [next-id (:next-id app-state)]
     (reacl/return :local-state ""
                   :app-state
                   (assoc app-state
                     :todos
                     (concat (:todos app-state)
                             [(->Todo next-id
                                      local-state
                                      false)])
                     :next-id (+ 1 next-id))))
```

@ctive group

Text for New Todo

```
(reacl/defclass to-do-app
  this app-state []
  local-state [local-state “”]
  render
  (dom/div
    ...
    (dom/form
      ...
      (dom/input {:onchange
                  (fn [e]
                    (reacl/send-message!
                     this
                     (->NewText
                      (.. e -target -value))))
                  :value local-state})))))
```

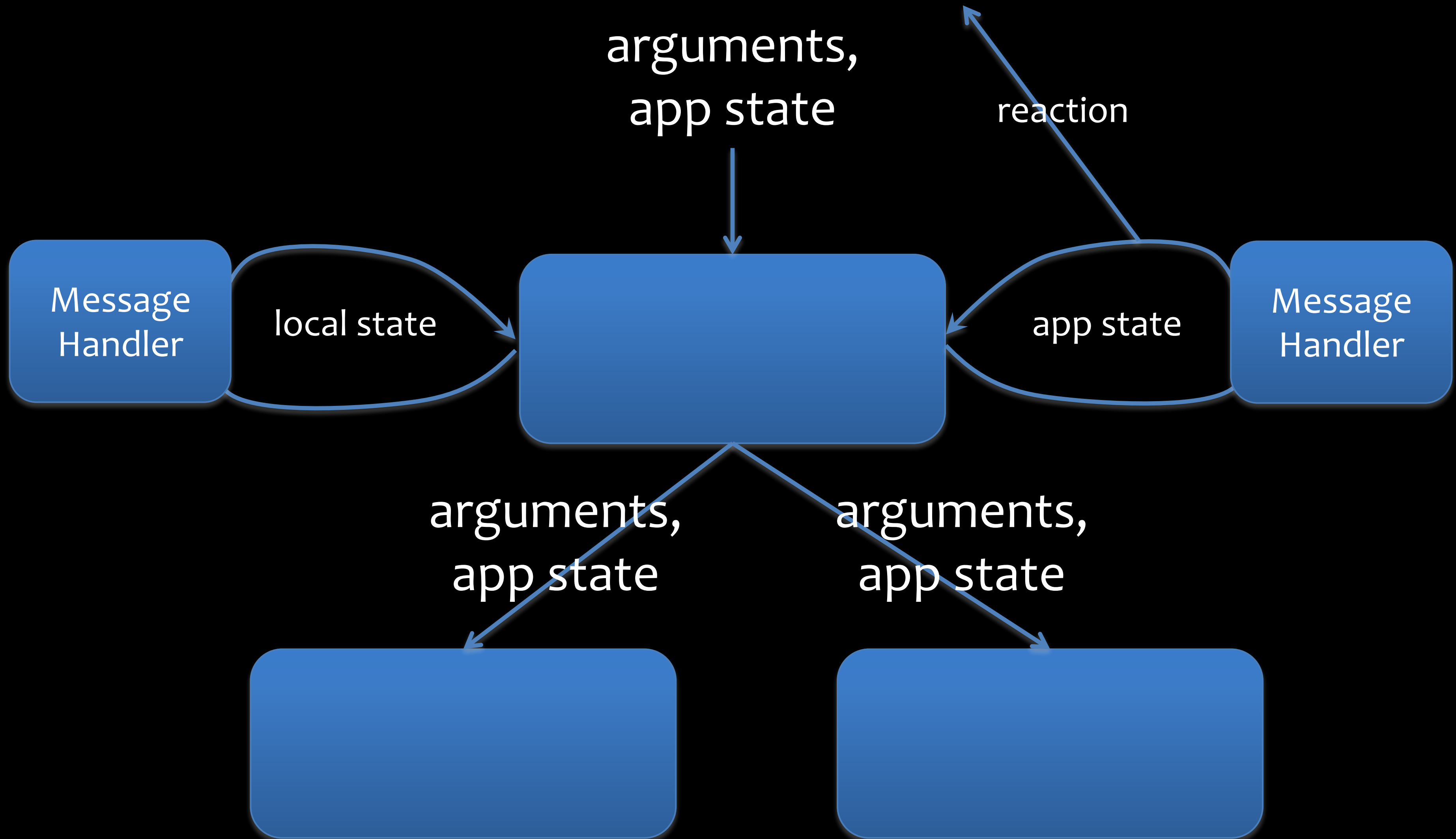
@ctive group

Text for New Todo

```
handle-message
(fn [msg]
  (cond
    ...
    (instance? NewText msg)
    (reacl/return :local-state
                  (:text msg))))
```

@ctive group

Reacl Component Tree



@rtive group

Tests

```
(reacl/defclass blam
  this app-state []
  local [braf (+ app-state 7)]
  render
  (dom/div (str braf))
  handle-message
  (fn [new]
    (reacl/return :app-state new)))

(deftest local-app-state-change
  (let [item (test-util/instantiate&mount blam 5)]
    (reacl/send-message! item 6)
    (is (= ["13"]
           (map dom-content (doms-with-tag item "div"))))))
```

@ctive group

Tests without DOM

```
(deftest string-display-test
  (let [e (string-display "Hello, Mike")
        renderer (reacl-test/create-renderer)]
    (reacl-test/render! renderer e)
    (let [t (reacl-test/render-output renderer)]
      (is (reacl-test/dom=? (dom/h1 "Hello, Mike")
                            t))
      (is (reacl-test/element-has-type? t :h1))
      (is (= ["Hello, Mike"]
             (reacl-test/element-children t))))))
```

Tests without components

```
(deftest contacts-display-handle-message-test
  (let [[_ st] (reacl-test/handle-message
                 contacts-display
                 [{:first "David" :last "Frese"}]
                 [] "Foo"
                 (->Add {:first "Mike”
                         :last "Sperber"}))]
    (is (= [{:first "David", :last "Frese"}
            {:first "Mike", :last "Sperber"}]
           (:app-state st))))
  (let [[_ st] (reacl-test/handle-message
                 contacts-display
                 [{:first "David" :last "Frese"}]
                 [] "Foo"
                 (->NewText "David Frese"))]
    (is (= "David Frese"
           (:local-state st)))))
```

@ctive group

Actions

```
(defn edn-xhr
 [{:keys [method url data on-complete]}]
  (let [xhr (XhrIo.)]
    (events/listen xhr EventType.COMPLETE
      (fn [e]
        (on-complete
          (reader/read-string
            (.getResponseText xhr)))))
    (. xhr
      (send url (meths method)
            (when data (pr-str data))
        #js {"Content-Type”
             "application/edn"}))))
```

@ctive group

Comments example

```
(reacl/defclass comment-box
  this comments []
  render
  (dom/div {:class "commentBox"}
           (dom/h1 "Comments")
           (comment-list
               comments))
```

@ctive group

Comments example

```
handle-message
(fn [msg]
  (cond
    (instance? NewComments msg)
    (reacl/return :app-state
                  (map (fn [e]
                         (->Comment
                           (:author e)
                           (:text e)))
                       (:comments msg))))
```

@ctive group

Comments example

```
handle-message
(fn [msg]
  (cond
    ....
    (instance? Refresh msg)
    (reacl/return :action
      (->EdnXhr this
                "comments.edn”
                ->NewComments))))
```

@active group

Actions

```
(defrecord EdnXhr
  [component url make-message])
```

@ctive group

Comments example

```
component-did-mount
(fn []
  (reacl/return :action
        (->RefreshMeEvery this
                          2000))))
```

@ctive group

Handling actions

```
(defn handle-action
  [app-state action]
  (cond
    (instance? RefreshMeEvery action)
    (let [refresh (fn []
                    (reacl/send-message! (:component action)
                                         (->Refresh)))]
      (refresh)
      (js/setInterval refresh 2000))

    (instance? EdnXhr action)
    (edn-xhr {:method :get
              :url (str (:url action) "?") ; prevent caching
              :on-complete (fn [edn]
                             (reacl/send-message!
                               (:component action)
                               (->NewComments. edn)))})))
```

@ctive group

Dependency Injection

```
(reacl/render-component
 (.getElementById js/document
                  "content")
 comment-box
 (reacl/opt :reduce-action
            handle-action)
 [])
```

active group

Reacl

- lexically scoped
- no destructive state manipulation
  in user code
- separates app state from transient GUI state
- no-DOM testing
- in production

React in ClojureScript

```
(def Comment
  (js/React.createClass #js
   {:render
    (fn []
      (this-as this
                (let [props (.-props this)]
                  (js/React.DOM.div
                   nil
                   (js/React.DOM.h2 nil
                     (.-author props))
                   (js/React.DOM.span nil
                     (.-text props)))))))}))
```

@ctive group

Component Classes

```
(def CommentList
  (js/React.createClass #js
   {:render
    (fn []
      (this-as this
               (js/React.DOM.div
                nil
                (into-array
                 (map (fn [c]
                        (Comment #js
                         {:author (:author c)
                          :text (:text c)}))
                      (.-comments (.-props this)))))))}))
```

@ctive group

Interactive Components

```
(def NewComment
  (js/React.createClass #js
   {:render
    (fn []
      (this-as this
               (js/React.DOM.form
                #js {:onSubmit (.-handleSubmit this)}
                (js/React.DOM.input
                 #js {:type "text" :ref "author"})
                (js/React.DOM.input
                 #js {:type "text" :ref "text"})
                (js/React.DOM.button nil "Submit"))))
```

@ctive group

React Sublety

```
(def NewComment
  (js/React.createClass #js
   {:render
    (fn []
      (this-as this
               (js/React.DOM.form
                #js {:onSubmit (.-handleSubmit this)}
                 (js/React.DOM.input
                  #js {:type "text" :ref "author"})
                 (js/React.DOM.input
                  #js {:type "text" :ref "text"})
                 (js/React.DOM.button nil "Submit"))))
```

Why not
(fn [...] ...)?

@rlive group

Event Handlers

```
:handleSubmit
(fn [e]
  (this-as
   this
   (let [props (.-props this)
         refs (.-refs this)
         author-dom (.getDOMNode (.-author refs))
         text-dom (.getDOMNode (.-text refs))]
     (.newComment props
                  {:author (.-value author-dom)
                   :text (.-value text-dom)})))
  false)}))
```

@ctive group

Interactive Components

```
(def NewComment
  (js/React.createClass #js
   {:render
    (fn []
      (this-as this
                (js/React.DOM.form
                 #js {:onSubmit (.-handleSubmit this)}
                  (js/React.DOM.input
                   #js {:type "text" :ref "author"})
                  (js/React.DOM.input
                   #js {:type "text" :ref "text"})
                  (js/React.DOM.button nil "Submit"))))
```

@active group

Component State

```
(def CommentBox
  (js/React.createClass #js
   {:getInitialState
    (fn []
      (this-as this
               #js {:comments (.-comments (.-props this))}))
    :render
    (fn []
      (this-as this
               (js/React.DOM.div
                nil
                (js/React.DOM.h1 nil "Comments")
                (CommentList
                 #js {:comments (.-comments (.-state this))})
                (js/React.DOM.h2 nil "New Comment")
                (NewComment
                 #js {:newComment (.-newComment this)})))))
```

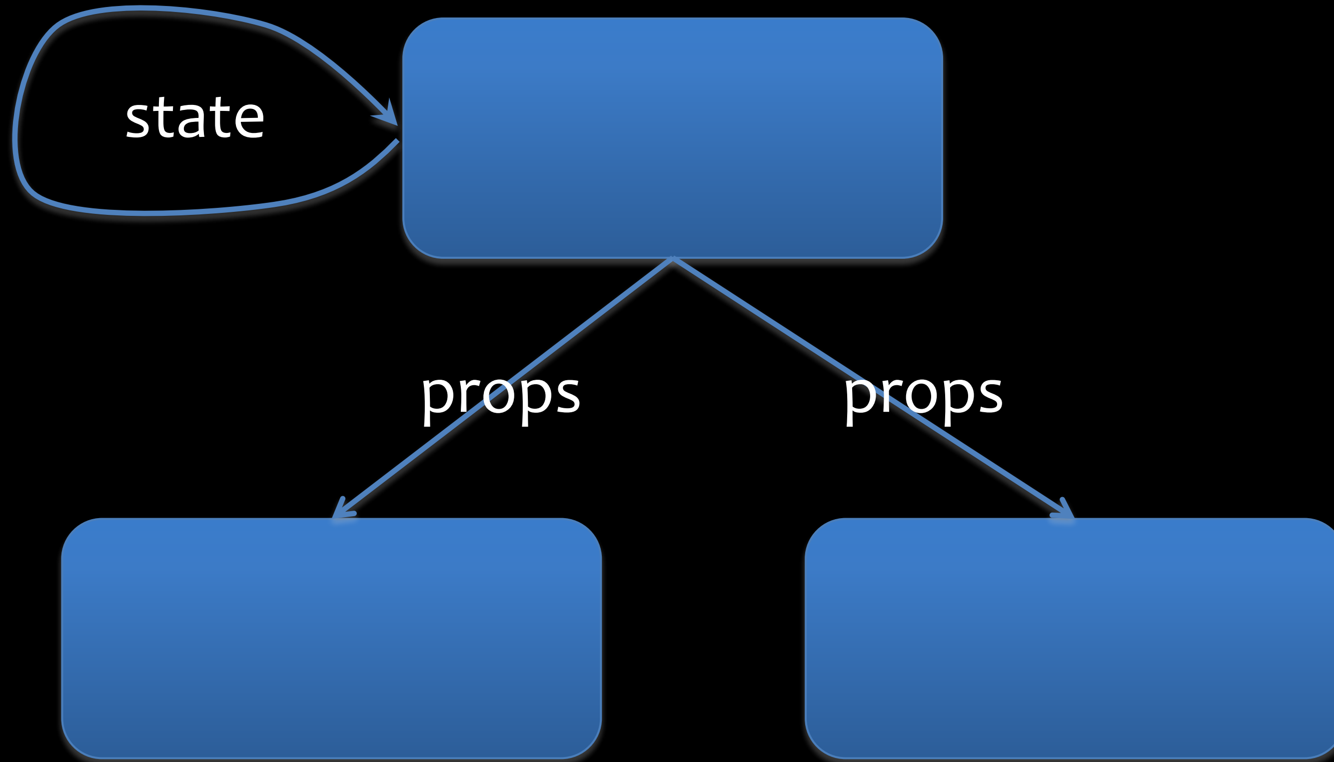
@rtive group

Event Handlers & Component State

```
:newComment
(fn [c]
  (this-as
  this
    (.setState
     this
     #js {:comments
          (conj (.-comments (.-props this))
                c)})))))
```

@rtive group

React Component Tree



@rtive group