Web-Applikationen und UIs funktional programmieren mit React

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

Controller

View

Model
Problem

model

change

model'

view

hopefully corresponding

change

view'

model

change

model'

view

view'
A World of Objects

© Trustees of the British Museum
Imperative Programming

room1.exit(elephant)
hallway.enter(elephant)
hallway.exit(elephant)
room2.enter(elephant)
Reality and Snapshots
React

model

new model

change

generates

view

new view

User

generates
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
• Lisp
• funktional
• JVM

Clojure

cljs

JS
15
truetruefalse"foo"
(+ 1 2)
(+ 1 (* 2
    (+ 17 21))))
(def pi 3.14159265)

(defn circumference [r] (* 2 pi r))
Reacl Component Tree

defined interface
TODO

- Zap
- Zap
- Make money
- retire
- die
- Add #2
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]))
(react/defclass to-do-item
todo [ ]
render
(dom/div
  (dom/input {:type "checkbox"
              :value (:done? todo)}
  (dom/button "Zap")
  (:text todo)))

app state
Check Todo

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

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

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

```clojure
handle-message
(fn [checked?]
  (reacl/return
    :app-state
    (assoc todo :done? checked?)))
```

(reacl/send-message!
  this
  (.. e -target -checked))
Pure Functions

<table>
<thead>
<tr>
<th>id</th>
<th>5</th>
</tr>
</thead>
<tbody>
<tr>
<td>text</td>
<td>retire</td>
</tr>
<tr>
<td>done?</td>
<td>false</td>
</tr>
</tbody>
</table>

(assoc :done? true)

<table>
<thead>
<tr>
<th>id</th>
<th>5</th>
</tr>
</thead>
<tbody>
<tr>
<td>text</td>
<td>retire</td>
</tr>
<tr>
<td>done?</td>
<td>true</td>
</tr>
</tbody>
</table>
Zap Todo

(dom/button "Zap")
Zap Todo

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

(reactl/defclass to-do-item
  this todo [parent]
  ...)

...
Local State
Application State vs. Local State

```(reac1/defclass to-do-app
   this app-state []
   local-state [local-state ""]
   render
   (dom/div
    (dom/h3 "TODO"
     ...
     )))```
Todo App

(defrecord TodosApp [next-id todos])

(defrecord NewText [text])
(defrecord Submit [])
(defrecord Change [todo])
(defrecord Delete [todo])
Todo App

(react/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
              (react/opt :embed-app-state embed-changed-todo)
              todo
              this)))
          (:todos app-state)))
    ...
))
Instantiating a Reacl Element

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

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

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

(reactl/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)))))
TODO

- Zap
- Make money
- Zap
- retire

- die

Add #2
New Todos

(react/defclass to-do-app
  this app-state []
  local-state [local-state ""]
  render (dom/div
    ...
    (dom/form
     {:onSubmit (fn [e]
        (.preventDefault e)
        (react/send-message! this
         ->Submit)))}
    ...
    (dom/button
     (str "Add #" (:next-id app-state)))
  )))
handle-message
(fn [msg]
  (cond
    ...
    (instance? Submit msg)
    (let [next-id (:next-id app-state)]
      (reac/return :local-state ""
        :app-state
        (assoc app-state
          :todos (concat (:todos app-state)
                   [[(->Todo next-id local-state false)]
                    :next-id (+ 1 next-id)])
        :next-id (+ 1 next-id))))
)
(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}))))
  )
handle-message
(fn [msg]
  (cond
    ...
    (instance? NewText msg)
    (reacl/return :local-state (:text msg))))
Reacl Component Tree

Message Handler

local state

arguments, app state

app state

Message Handler

arguments, app state

arguments, app state
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"))))))
```
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] (reactl-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] (reactl-test/handle-message
                 contacts-display
                 [{:first "David" :last "Frese"}]
                 [] "Foo"
                 (->NewText "David Frese"))]
    (is (= "David Frese"
          (:local-state st))))))
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
            (.getResponseBody xhr)))))
    (.xhr
      (send url (meths method)
        (when data (pr-str data))
        #js {
          "Content-Type"
          "application/edn"}))))
Comments example

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

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

handle-message

(fn [msg]
  (cond
    ....
    ....
    (instance? Refresh msg)
    (reac1/return :action
      (->EdnXhr this
        "comments.edn"
        (->NewComments)))))
Actions

(defrecord EdnXhr
    [component url make-message])
component-did-mount

(fn []
  (reactl/return :action
    (->RefreshMeEvery this 2000))))
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))))))))
Dependency Injection

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

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

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

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

:newComment

(fn [c]
  (this-as this
    (.setState this
      #js { :comments
        (conj (. comments (. props this))
          c)}))))
React Component Tree

state

props

props