CS 696 Functional Programming and Design
Fall Semester, 2015
Doc 21 MVC, Re-frame
Nov 17, 2015

Norris Number

Average amount of code an untrained programmer can write before they hit a wall

~1,500 lines

Beyond that the code becomes so tangled they cannot debug or modify it without herculean effort

http://www.teamten.com/lawrence/writings/norris-numbers.html

Scale Changes Everything

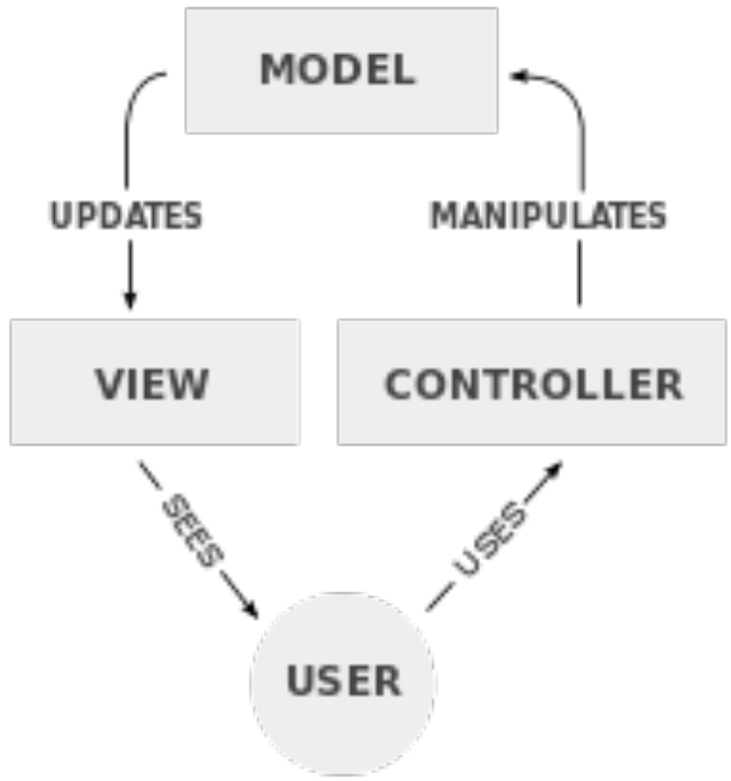
Architecture

What are the major parts of the program

What are the resposiblities of each part

How to the parts interact

Model-View-Controller (MVC)



Started in Smalltalk

Model - data for the app

View - Displays model in the UI

Controller

Takes user input

Manipulates model

Cause view to update appropriately

Talks to both model & view

Image from https://en.wikipedia.org/wiki/Model%E2%80%93view%E2%80%93controller, Viewed 11/16/15

Model-View-Controller

- Separation of presentation from Model
  - Model and View are different concerns
  - View changes at different rate
  - Multiple ways of presenting same data
  - Easier to test model logic

- Separation of view & controller
  - Smalltalk had little separation between
  - In desktop frameworks each view usually has one controller
  - Martin Fowler
    - This seperation not as important

Web & MVC

Web frameworks commonly use MVC

Each framework as slightly different definition of MVC

Django

Controller
- Handling requests & responses
- Setting up database connections
- URL config file

Model
- Database + code that uses the database

View
- HTML page & code that renders templates

Reagent & re-frame

Reagent - View

re-frame

Architecture for app using Reagent

re-frame

Big ratom

Immutable data

Pure functions

One-way data flow

Big Ratom

Place all state in one ratom

```
(def app-db  (reagent/atom {}))
```

Benefits of Big Ratom

Single source of truth

Now synchronization issues between widgets

Save & undo

Issues with Big Ratom

What is the structure of the ratom?

Widget only needs small part of ratom

Structure verse Freedom

Structure vs Freedom

Structure

Types
- Java
- Swift

Data
- Classes

Process
- Waterfall Model

Freedom

Types
- Clojure
- Ruby

Data
- Maps

Process
- Test-Drive Design
- Agile methods

Structure builds in discipline for you

Freedom requires self discipline

Clojure & Types

Informal documentation

Naming convention

(defn foo [s xs line-map] ...)

(defn foo
“line-map {:start {:x 12 :y 0} :end {:x 18 :y 202}}”
[s xs line-map] ...)

Clojure & Types

Records

```
(defrecord Point [x y])

(defrecord Line [^Point start ^Point end])
```

```
(def a (Line. (Point. 12 0) (Point. 18 202)))

(:start a)
(:end a)
```

```
(defn foo
  [^Line line]
  ....)
```

Clojure & Types

Schema

Prismatic https://github.com/Prismatic/schema

Define schema for your data

Validate data

Annotate function arguments & return values

Prismatic Schema Use Cases

Documentation

Validate data usage in tests

Check data that from/to external sources

- Files
- Database
- Network

Prismatic Schema

```
(ns schema-examples
  (:require [schema.core :as s
             :include-macros true ;; cljs only
             ]))
```

Basic Types

s/Any, s/Bool, s/Num, s/Keyword, s/Symbol, s/Int, and s/Str
String long double java.lang.Long etc

Compound Types

Vectors

[s/Str] -> [“a” “2”]
[s/Int]   -> [1 2 3]

Maps

{s/Str s/Num} -> {“a” 4 “b” 0}

{long {String double}} -> {1 {"2" 3.0 "4" 5.0}}

Validate & check

(s/validate s/Num 42)	42
(s/validate s/Num "42")	Exception Value does not match schema: (not (instance? java.lang.Number "42"))
(s/check s/Num 4)	nil
(s/check s/Num "4")	(not (instance? java.lang.Number "42"))

Documentation

```
(def point-schema
     {:x s/Num :y s/Num})

(def line-schema
     {:start point-schema
      :end point-schema})


(defn foo
     “line is of type line-schema”
      [line]
      (-> line :start :x))


(foo {:start {:x 1 :y 10} :end {:x 20 }})
```

Checking at Runtime

```
(def point-schema
     {:x s/Num :y s/Num})

(def line-schema
     {:start point-schema
      :end point-schema})

(defn foo
     [line]
     {:pre [(s/validate line-schema line)]
      :post [(s/validate s/Num %)] }
     (-> line :start :x))

(foo {:start {:x 1 :y 10} :end {:x 20 }})
```

Selective Checks with with-fn-validation

```
(def point-schema
     {:x s/Num :y s/Num})

(def line-schema
     {:start point-schema
      :end point-schema})

(s/defn foo :- s/Num
     [line :- line-schema]
     (-> line :start :x))


(foo {:start {:x 1 :y 10} :end {:x 20 }})          ;; runs fine


(s/with-fn-validation
     (foo {:start {:x 1 :y 10} :end {:x 20 }}))    ;; Throws an error
```

Always-validate

```
(def point-schema
  {:x s/Num :y s/Num})

(def line-schema
  {:start point-schema
   :end point-schema})

(s/defn ^:always-validate foo :- s/Num
        [line :- line-schema]
      (-> line :start :x))


(foo {:start {:x 1 :y 10} :end {:x 20 }})              ;; Exception
```

Back to reframe & Reagent

Streams or Flows

Database

- Stream of requests
- Prevayler (http://prevayler.org)

Files

- Mirror Worlds 1992, David Gelernter
- Intellij
- Smalltalk

Refactoring

How Flow Happens In Reagent

ratom

reaction

Wraps a computation

returns a ratom holding the result of the computation

computation redone when input changes

```
(ns firstreagent.reframe
  (:require-macros [reagent.ratom :refer [reaction]])  ;; reaction is a macro
  (:require        [reagent.core  :as    reagent]))

(def app-db  (reagent/atom {:a 1}))

(def ratom2  (reaction {:b (:a @app-db)}))

(def ratom3  (reaction (condp = (:b @ratom2)
                                 0 "World"
                                 1 "Hello")))
```

```
(println @ratom2)    ;; ==>  {:b 1}
(println @ratom3)    ;; ==> "Hello"


(reset!  app-db  {:a 0})

(println @ratom2)    ;; ==>  {:b 0}
(println @ratom3)    ;; ==> “World”
```

How does reaction work

reaction is a macro

```
(def ratom2  (reaction {:b (:a @app-db)}))
```

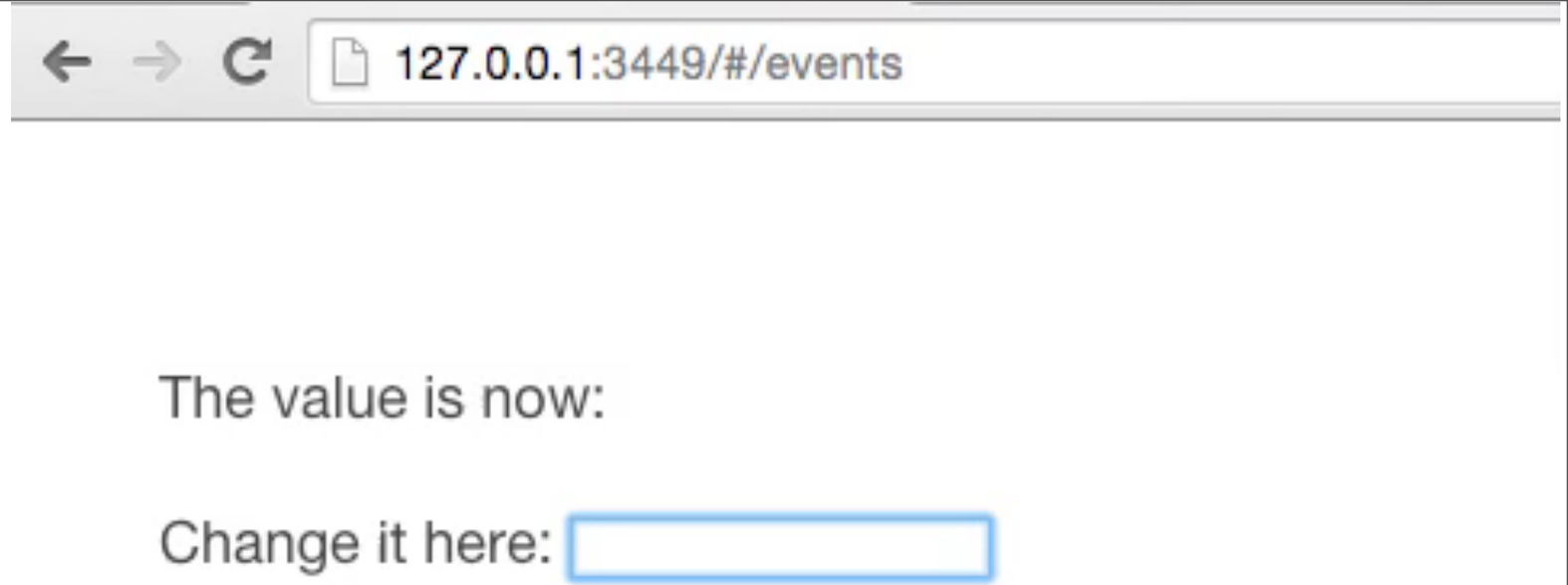
So it know about the atom

Can register a watcher on the atom

Bit more complex than that

How React Works

```
(defn greet
  [name]
  [:div "Hello "  @name])

(def n (reagent/atom "re-frame"))

(def hiccup-ratom  (reaction (greet n)))

(println @hiccup-ratom)          ;; ==>  [:div "Hello " "re-frame"]


(reset! n "blah")            ;;    n changes

(println @hiccup-ratom)      ;; ==>   [:div "Hello " "blah"]
```



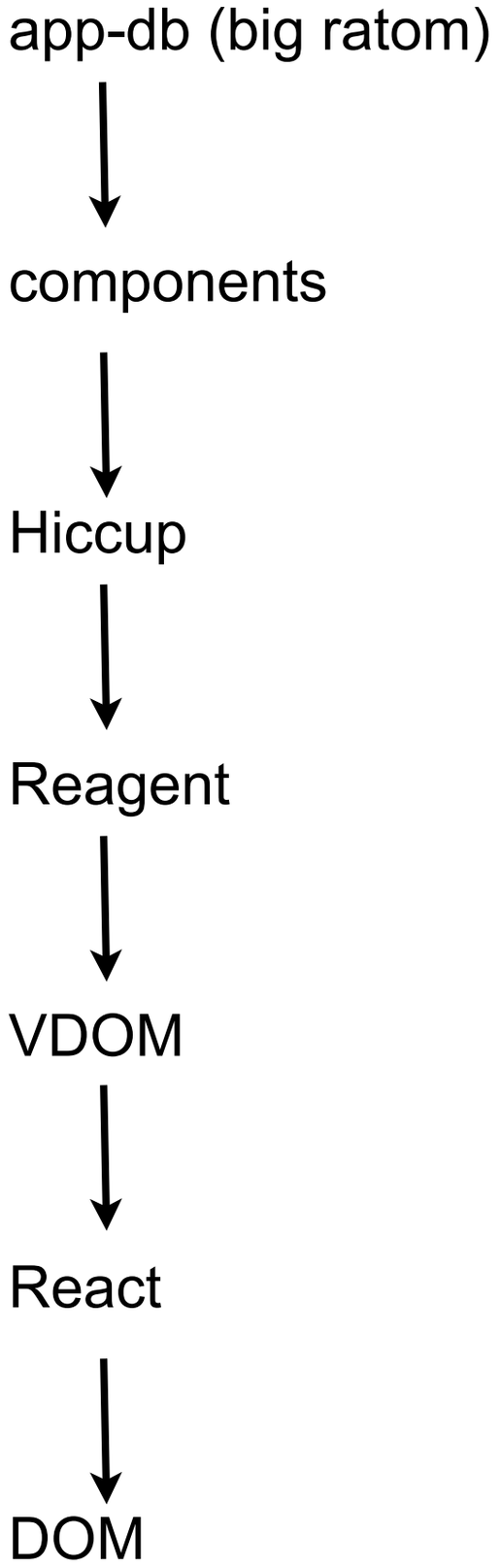

```
(ns firstreagent.events
  (:require  [reagent.core :as r] ))

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

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

How does this Work?

Your Hiccup vectors are wrapped in a reaction

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

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

Data Flow



Issues of Big Ratom

What is the structure of the ratom?

Widget only needs small part of ratom

Reagent Cursors
reframe Subscriptions

Reagent Cursor

(cursor ratom [path])

Returns cursor on part of ratom

Acts like a ratom

Example - Changing Cursor changes ratom

```
(ns firstreagent.reframe
  (:require-macros [reagent.ratom :refer [reaction]])  ;; reaction is a macro
  (:require        [reagent.core  :as    reagent]))

(def app-db  (reagent/atom {:a 1 :b [1 2 3]}))

(print @app-db)                                        ;==>      {:a 1, :b [1 2 3]}


(def sample (reagent/cursor app-db [:b 0]))
(print @sample)                                        ;==> 1



(reset! sample 9)

(print @sample)                                        ;==> 9
(print @app-db)                                        ;==> {:a 1, :b [9 2 3]}
```

Example - Changing ratom changes cursor

```
(def app-db  (reagent/atom {:a 1 :b [1 2 3]}))

(print @app-db)                                  ;==> {:a 1, :b [1 2 3]}

(def sample (reagent/cursor app-db [:b 0]))

(print @sample)                                  ;==> 1

(swap! app-db update-in [:b 0] inc)

(print @app-db)                                  ;==> {:a 1, :b [2 2 3]}
(print @sample)                                  ;==> 2
```

Example


127.0.0.1:3449/#/graphics
Current state: {:name {:first-name "John", :last-name "Smith"}}
I'm editing John Smith.
First name: John
Last name: Smith

Example

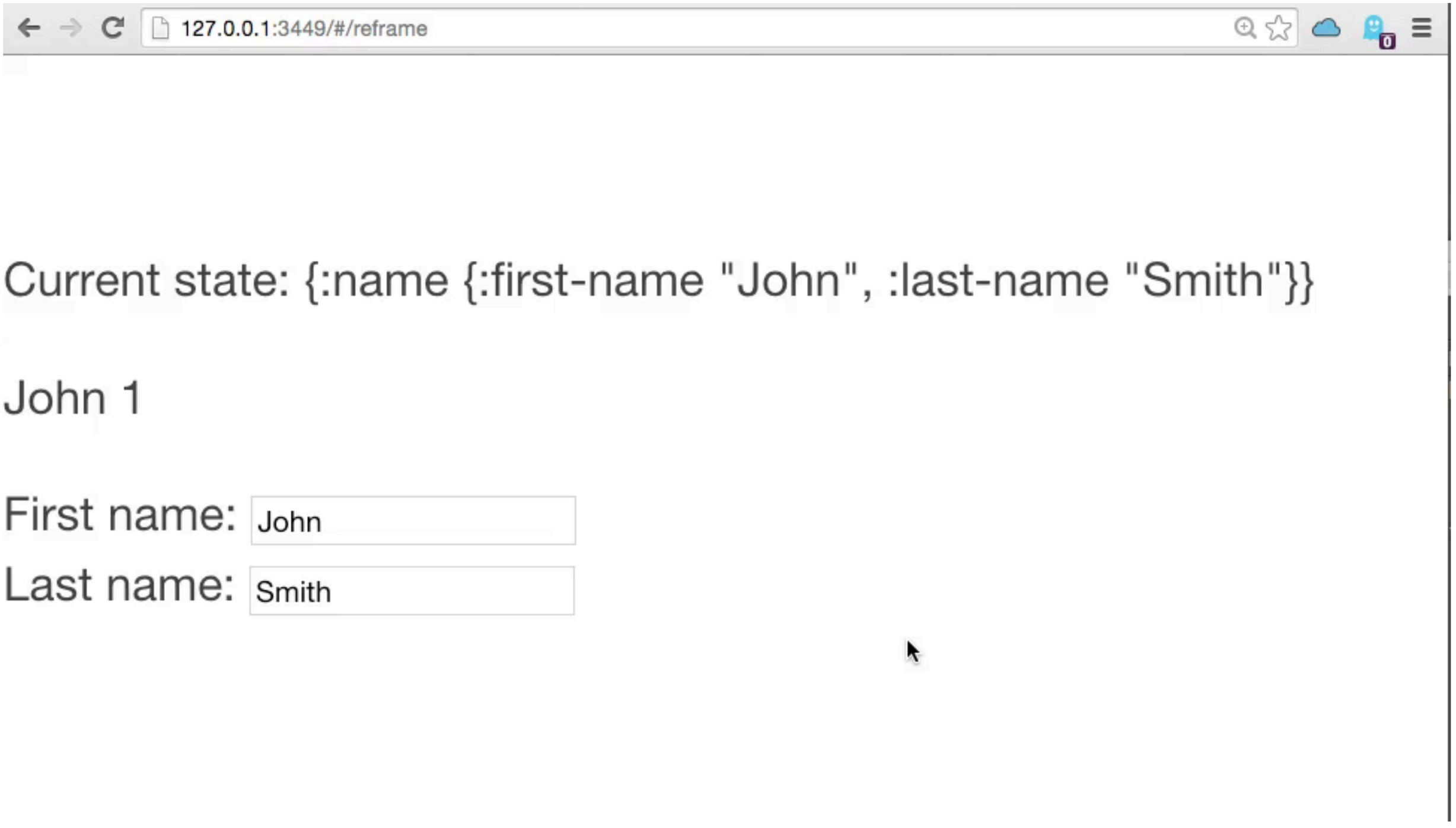
```
(def app-db  (reagent/atom {:name
                            {:first-name "John" :last-name "Smith"}}))

(defn input [prompt val]
  [:div
   prompt
   [:input {:value @val
            :on-change #(reset! val (.-target.value %))}]])

  (defn cursor-name-edit [n]
    (let [{:keys [first-name last-name]} @n]
      [:div
       [:p "I'm editing " first-name " " last-name "."]

       [input "First name: " (reagent/cursor n [:first-name])]
       [input "Last name:  " (reagent/cursor n [:last-name])]]))

(defn cursor-parent []
  [:div
   [:p "Current state: " (pr-str @app-db)]
   [cursor-name-edit (reagent/cursor app-db [:name])]])
```

Cursor and Big Ratom

Cursors represent small part of the data in big ratom

Cursors only update when their part of big ratom change

Changes to other parts of big ratom do not affect a cursor


127.0.0.1:3449/#/reframe
Current state: {:name {:first-name "John", :last-name "Smith"}}
John 1
First name:
John
Last name:
Smith

```
(def app-db  (reagent/atom {:name
                        {:first-name "John" :last-name "Smith"}}))

(def first-name (reagent/cursor app-db [:name :first-name]))

(defn display-count
  [value]
  (let [counter (atom 0)]
    (fn []
      (swap! counter inc)
      [:p value " " @counter])))

(defn input [prompt val]
  [:div
   prompt
   [:input {:value @val
            :on-change #(reset! val (.-target.value %))}]])
```

```
  (defn cursor-name-edit [n]
    (let [{:keys [first-name last-name]} @n]
      [:div
       [input "First name: " (reagent/cursor n [:first-name])]
       [input "Last name:  " (reagent/cursor n [:last-name])]]))


(defn cursor-parent []
  [:div
   [:p "Current state: " (pr-str @app-db)]
   [display-count @first-name]
   [cursor-name-edit (reagent/cursor app-db [:name])]])
```

Back to MVC

Model

- Data
- Reading & writing of data
- Logic on the data

Big ratom & cursors

- Model
- Like database for app

```
(def app-db  (reagent/atom {:name
                            {:first-name "John" :last-name "Smith"}}))

(def first-name (reagent/cursor app-db [:name :first-name]))
(def last-name (reagent/cursor app-db [:name :last-name]))
```

View

View - Displays model in the UI

Hiccup part of view

```
[:p "Current state: " (pr-str @app-db)]
```

```
(defn display-count
  [value]
  (let [counter (atom 0)]
    (fn []
      (swap! counter inc)
      [:p value " " @counter])))

(defn cursor-parent []
  [:div
   [:p "Current state: " (pr-str @app-db)]
   [display-count @first-name]
   [cursor-name-edit (reagent/cursor app-db [:name])]])
```

Controller

Controller

- Takes user input
- Manipulates model
- Cause view to update appropriately
- Talks to both model & view

```
(defn input [prompt val]
  [:div
   prompt
   [:input {:value @val
            :on-change #(reset! val (.-target.value %))}]])
```

MVC, Big Ratom & Cursors

View & Controller are mixed together

Separation of view & controller

- Smalltalk had little separation between
- In desktop frameworks each view usually has one controller
- Martin Fowler
  - This seperation not as important

reframe Dislikes Cursor

Two way flow

Mixes view & controller

Can not create different views on data

Scale Changes Everything