Scalable Vector Graphics (SVG)

w3schools SVG Tutorial

http://www.w3schools.com/svg/

A gentle introduction

http://cloudfour.github.io/slides-svg-101/#/

What does this imply?
Scalable Vector Graphics (SVG)

**PNG8**
331 bytes (optimized)

**SVG**
103 bytes (optimized)

```xml
<img width="240" height="240"
    src="data:image/png;base64,iVBORw0KGgoAAAANSUhEUgAAABgAAAAIAAABAAECAwMAAAABGddefault.jpg"

<svg width="240" height="240" viewBox="0 0 24 24">
  <circle fill="#F26941" cx="12" cy="12" r="12"/>
</svg>
```
Scalable Vector Graphics (SVG)

Shapes

Rectangles  Filters  Coordinate systems
Circle       Gradients  Transformations
Ellipse      Line       Viewport
Line         Polygon    View box
Polygon      Polyline   
Polyline     Path       
Path         Text       
Text         Stroking   

4
(defn main []
  [:div
    [:svg {:width 600 :height 600 :stroke "black"
        :style {:position :fixed :top 0 :left 0 :border "red solid 1px"}}
    [:line {:x1 0 :y1 0
            :x2 300 :y2 300}]
    [:circle {:cx 350 :cy 350 :r 50 :fill "red"}]
    [:rect {:x 300 :y 450 :width 100 :height 50}]
  ])

[:rect {x 300 :y 450 :width 100 :height 50}]

<rect x="300" y="450" width="100" height="50" />
Mouse position is given with respect to the window coordinates

Need to know location of svg canvas in window

Easiest way is to use fixed position for svg canvas
Don’t nest shapes in a div or other tags
(defn main []
  [:div
    [:svg {:width 600 :height 600 :stroke "black"
        :style {:position :fixed :top 0 :left 0 :border "red solid 1px"}}
     (list [:line {:x1 0 :y1 0
                     :x2 300 :y2 300}]
           [:circle {:cx 350 :cy 350 :r 50 :fill "red" }]
           [:rect {:x 300 :y 450 :width 100 :height 50}]])
  ])

In a list works
(defn draw []
[:line {:x1 0 :y1 0
         :x2 300 :y2 300}]
[:circle {:cx 350 :cy 350 :r 50 :fill "red" }]
[:rect {:x 300 :y 450 :width 100 :height 50}]])

(defn main [] [:div [:svg {:width 600 :height 600 :stroke "black"
             :style {:position :fixed :top 0 :left 0 :border "red solid 1px"}} (draw) ]])

Needs to be in a list
(defn draw []
  (list [:line {:x1 0 :y1 0 :x2 300 :y2 300}]
    [:circle {:cx 350 :cy 350 :r 50 :fill "red"}]
    [:rect {:x 300 :y 450 :width 100 :height 50}])))

(defn main []
  [:div
   [:svg {:width 600 :height 600 :stroke "black"
         :style {:position :fixed :top 0 :left 0 :border "red solid 1px"}}
    (draw)]]

See list works
(defn draw-lines []
  (for [x [0 50 100]]
    [:line {:x1 x :y1 0 :x2 (+ x 300) :y2 300}])))

(defn draw-circles []
  (for [x [200 300 400]]
    [:circle {:cx x :cy 350 :r 50 :fill "red"}])))

(defn draw []
  (list
   (draw-lines)
   (draw-circles)
   [:rect {:x 300 :y 450 :width 100 :height 50}])))

(defn main []
  [:div
   [:svg {:width 600 :height 600 :stroke "black"
         :style {:position :fixed :top 0 :left 0 :border "red solid 1px"}}
    (draw)
   ]])
Data Flow

app-db (big ratom) → components → Hiccup → Reagent → VDOM → React → DOM
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

Current state: {:name {:first-name "John", :last-name "Smith"}}

I'm editing John Smith.

First name: John
Last name: Smith
Example

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

(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 cursor app-db [:name])]]
)

(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 cursor app-db [:name])]]
)

(def app-db (reagent/atom {:name
  {:first-name "John" :last-name "Smith"}})))
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
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 separation not as important
reframe Dislikes Cursor

Two way flow

Mixes view & controller

Can not create different views on data
reframe Subscribe & Events

Subscriptions
- Used in views to get data from big ratom
- Only way for views to get data
- Only used in views

Events
- When things happen that need change in data
- Used to trigger changes in big ratom

Handlers
- Subscription & Events have handlers that do the work
Goal

Keep Model & Controller logic

Separate
Out of Views

reframe manages big ratom
So view has to use subscription to get data
(ns firstreagent.reframefull
 (:require-macros [reagent.ratom :refer [reaction]])
 (:require [re-frame.core :refer [register-handler path register-sub dispatch dispatch-sync subscribe]]))

(def initial-state
 {:name "Roger" :age 21})

(register-handler
 :initialize
 (fn [db _] (merge db initial-state)))

(dispatch-sync [:initialize])
(register-sub
 :name-sub
 (fn
   [db _]
   (reaction (:name @db))))

(defn display-name
 []
 (let [name (subscribe [:name-sub])]
   (fn name-render
     []
     [:p "Hello " @name]]))
(register-sub
 :name-sub
 (fn
  [db [label a b]]
  (reaction (str label " a: " b " b: " b " (:name @db))))))

(defn display-name
 []
  (let [name (subscribe [:name-sub "cat" "dog"])]
    (fn name-render
     []
     [:p "Hello " @name]]))
Updating Big Ratom

Need event handler

dispatch an event with data
(register-handler
   :name
   (fn
      [db [value]]
      (assoc db :name value)))

(defn name-input
 []
 (let [name (subscribe [:name])]
  (fn name-input-render
     []
     [:div
      "Name: "
      [:input {:type "text"
               :value @name
               :on-change #(dispatch [:name (-> % .-target .-value)])}])))

(defn main
 []
 [:div
  [:div
   [name-input]
   [display-name]])
(defonce time-updater (js/setInterval
   #(dispatch [:timer (js/Date.)]) 1000))

(defn clock []
  (let [timer (subscribe [:timer])]
    (fn clock-render []
      (let [time-str (-> @timer .toTimeString
                          (clojure.string/split " ")
                          first)]
        [:div time-str]))))

(defn main [] [clock]
  (dispatch-sync [:initialize])
The imports

(ns firstreagent.timer
 (:require-macros [reagent.ratom :refer [reaction]])
 (:require [reagent.core :as reagent]
   [re-frame.core :refer [register-handler
                              path
                              register-sub
                              dispatch
                              dispatch-sync
                              subscribe]]))
Gave smallest possible examples

Logic will grow in handlers

Views remain clear of
  Model
  Controller
Single Page App with multiple views
First Project

lein new reagent projectname
(def mount-target
  [:div#app
   [:h3 "ClojureScript has not been compiled!"
   [:p "please run 
    [:b "lein figwheel"] " in order to start the compiler"]])

(dfroutes routes
  (GET "/" [] loading-page)
  (GET "/about" [] loading-page)
defroutes

(defroutes routes
  (GET "/" []) loading-page)
(defroutes routes
  (GET "/about" []) loading-page)

Method
GET
POST

URL

URL parameters

Function to call
When URL is requested

Thursday, November 19, 15
(def loading-page
(html [:html [:head [:meta {:charset "utf-8"}] [:meta {:name "viewport" :content "width=device-width, initial-scale=1"]}]
(include-css (if (env :dev) "css/site.css" "css/site.min.css"))]
[:body mount-target
(include-js "js/app.js")]))
(def mount-target
  [:div#app
   [:h3 "ClojureScript has not been compiled!"
   [:p "please run 
     [:b "lein figwheel"
     " in order to start the compiler"]]]
  )

Client will replace this if working correctly
Client-Side Libraries

accountant.core

ClojureScript library to make navigation in single-page applications simple

secretary.core

Defines client side routes
  URLs & function to call

reagent.session

Just an atom
  Used to store state
(defn home-page []
  [:div [:h2 "Welcome to foobar"]
   [:div [:a {:href "/about"} "go to about page"]]]
)

(defn about-page []
  [:div [:h2 "About foobar"]
   [:div [:a {:href "/"} "go to the home page"]]]
)

(defn current-page [] [:div [(session/get :current-page)]]

(secretary/defroute "/" []
  (session/put! :current-page #"home-page")

(secretary/defroute "/about" []
  (session/put! :current-page #"about-page")

(defn mount-root []
  (reagent/render [current-page] (.getElementById js/document "app"))

(defn init! []
  (accountant/configure-navigation!)
  (accountant/dispatch-current!)
  (mount-root))
(defn home-page []
   [:div [:h2 "Welcome to foobar"]
    [:div [:a {:href "/about"} "go to about page"]]]
)

(defn about-page []
   [:div [:h2 "About foobar"]
    [:div [:a {:href "/"} "go to the home page"]]])
Routes

(secretary/defroute "/" []
  (session/put! :current-page #'home-page))

(secretary/defroute "/about" []
  (session/put! :current-page #'about-page))

For each URL
  Change atom to hold reference to which function to call
(defn current-page [] [:div [(session/get :current-page)]]]

Lists are expanded in Hiccup
So expands to the current page

(defn mount-root []
  (reagent/render [current-page] (.getElementById js/document "app")))

Magic function
Render the client page each time current-page changes