Flow

Learnings from writing a ClojureScript DSL

James Henderson

github: james-henderson (https://github.com/james-henderson)

twitter: @jarohen (https://twitter.com/jarohen)

james@jarohen.me.uk (mailto:james@jarohen.me.uk)

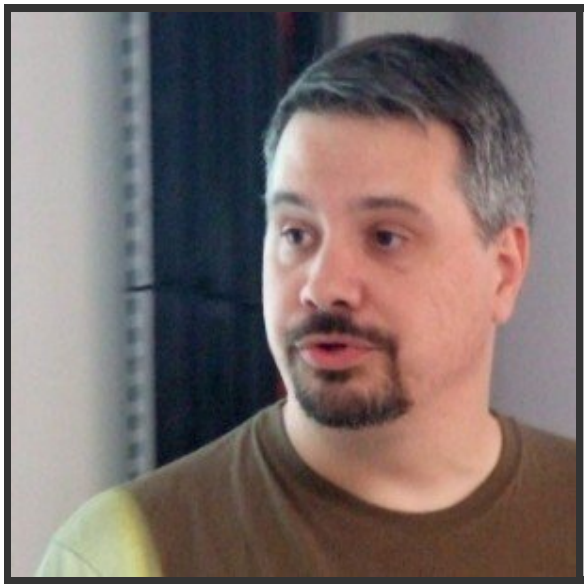
What's coming up?

- Looking back to ClojureX 2013
- Introducing Flow
- Write yourself a Flow

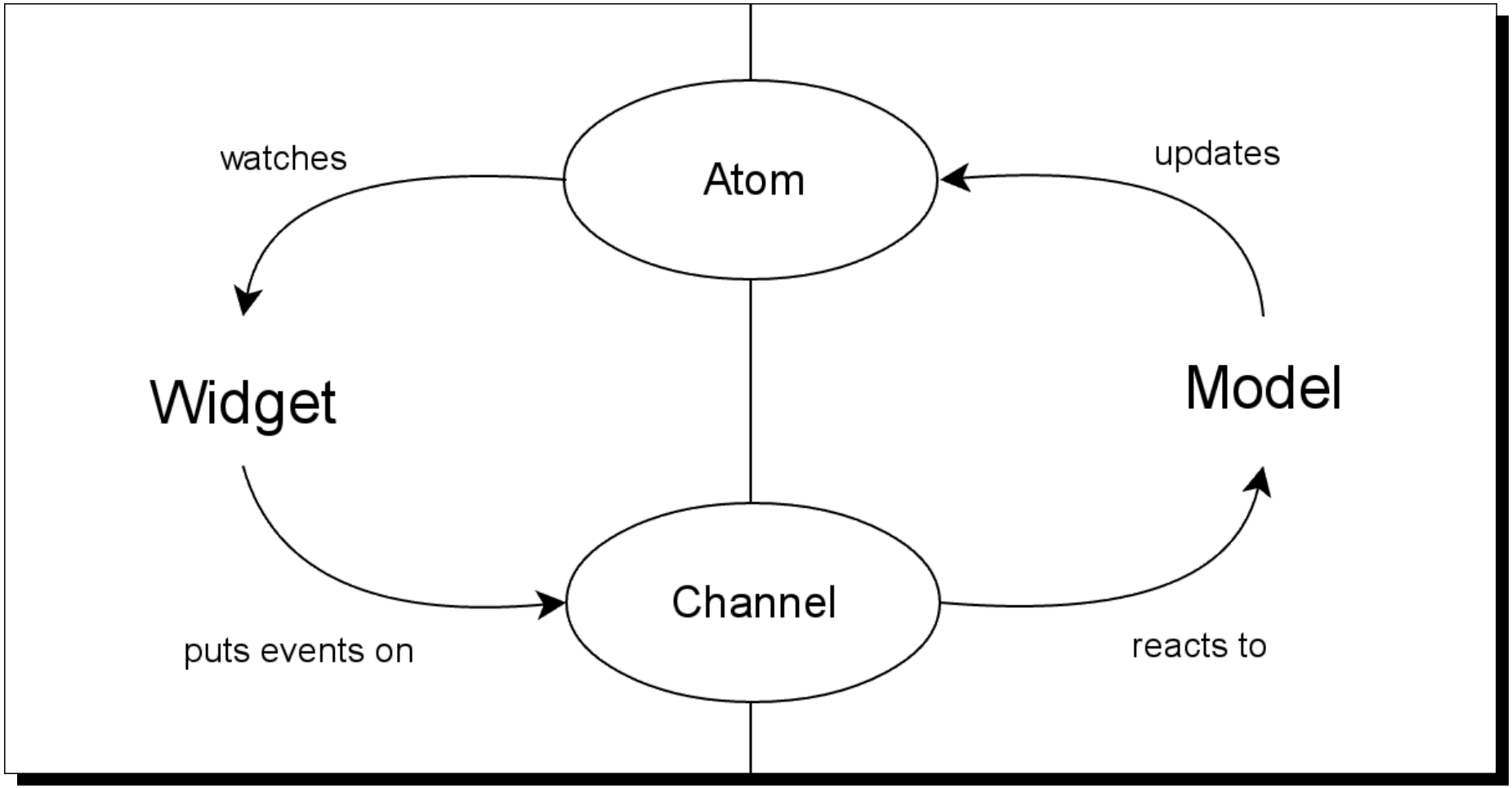
Looking back to ClojureX 2013

Photo: Ben Rogan, Oxford Knight

Looking back to ClojureX 2013



Of Models and Widgets



A typical Widget

```
(defprotocol CounterWidget
  (->counter-element [_])
  (set-counter-value! [_ new-value])
  (event-ch [_]))

(defn create-counter-widget []
  (reify CounterWidget
    ...))

(defn watch-counter! [!counter widget]
  (add-watch !counter ::watch-key
             (fn [_ _ _ new-value]
               (set-counter-value! widget new-value))))

(defn counter-widget [!counter command-ch]
  (let [widget (create-counter-widget)]
    (watch-counter! !counter widget)
    (a/pipe (event-ch widget) command-ch)

    (->counter-element widget)))
```

A typical Model

```
(defn wire-up-counter-model! [!counter command-ch]
  (go-loop []
    (let [msg (a/<! command-ch)]
      (swap! !counter
             (fn [old-value]
               (case msg
                 :increment (inc old-value)
                 ...)))
      (recur))))
```

Wiring it up

```
(defn counter-component []
  (let [!counter (atom 0)
        command-ch (a/chan)]
    (wire-up-counter-model! !counter command-ch)
    (counter-widget !counter command-ch)))
```

After ClojureX

- Feedback from the talk
- The Advent of Om/Reagent/React
- Why would you carry on with Flow?!


CHALLENGE ACCEPTED

Flow - Aims

- 100% declarative - no *how* or *when*, just *what*
- Minimise the number of new concepts introduced
- Perform *'well enough'* to be useful

Introducing Flow

https://github.com/james-henderson/flow
(https://github.com/james-henderson/flow)

```
[jarohen/flow "0.3.0-alpha1"]
```

```
lein new splat flow-hacking
```

Hello world!

```
(:require [flow.core :as f :include-macros true])

(defn hello-world []
  (f/root js/document.body
    (f/el
      [:p "Hello world!"])))
```

Hello world!

```
(:require [flow.core :as f :include-macros true])

(defn hello-world []
  (f/root js/document.body
    (f/el
      [:p
       [:b "Hello world!"]])))
```

Adding CSS

```
(:require [flow.core :as f :include-macros true])

(defn hello-world []
  (f/root js/document.body
    (f/el
      [:p.message {::f/style {:font-weight :bold
                              :color "#a00"}}
       "Hello world!"])))
```

Adding event listeners

```
(:require [flow.core :as f :include-macros true])

(defn hello-world []
  (f/root js/document.body
    (f/el
      [:p {::f/on {:click (fn [e]
                            (js/alert "Hi!"))}}
       "Hello world!"])))
```

Making it dynamic

```
;; ideal case, pseudo-code
(defn counter-component [counter]
  (f/el
    [:p "The value of the counter is " counter]))
```

Making it dynamic

```
;; this example is actual Flow code
(defn counter-component [!counter]
  (f/el
    [:p "The value of the counter is " (<< !counter)]))
```

Basic Buttons

```
(defn effect-counter-events! [event-ch !counter]
  (go-loop []
    (case (a/<! event-ch)
      :increment! (swap! !counter inc)
      :reset! (reset! !counter 0))

    (recur)))


(defn counter-component []
  (let [!counter (atom 0)
        event-ch (doto (a/chan)
                   (effect-counter-events! !counter))]
    (f/el
      [:div
       [:p "The value of the counter is " (<< !counter)]

       [:p [:button {::f/on {:click #(a/put! event-ch :increment!)}}
            "Increment"]]

       [:p [:button {::f/on {:click #(a/put! event-ch :reset!)}}
            "Reset"]]])))
```

Sub-components

```
(defn todo-component [!todo event-ch]
  (f/el
    (let [{:keys [caption deadline]} (<< !todo)]
      [:li caption
       (when deadline
         [:span " (due " deadline ")"])])))

(defn todo-list-component [!todos event-ch]
  (f/el
    [:ul
     (for [todo (<< !todos)]
       [todo-component (!<< todo) event-ch])]))

(f/root js/document.body
  (todo-list-component
    (atom [{:caption "Write talk", :deadline "Today"}
           {:caption "Christmas shopping", :deadline "2014-12-24"}
           {:caption "Clean house"}])))
```

Sub-components

```
(defn todo-component [!todo event-ch]
  (f/el
    (let [{:keys [caption deadline]} (<< !todo)]
      [:li caption
       (when deadline
         [:span " (due " deadline ")"])])))

(defn todo-list-component [!todos event-ch]
  (f/el
    [:ul
     (for [todo (->> (<< !todos)
                     (sort-by :todo-order))]
       [todo-component (!<< todo) event-ch])]))

(f/root js/document.body
  (todo-list-component
    (atom [{:caption "Write talk", :deadline "Today", :todo-order 0}
           {:caption "Christmas shopping", :deadline "2014-12-24",
            :todo-order 2}
           {:caption "Clean house", :todo-order 1}])))
```

Sub-components

```
(defn todo-component [!todo event-ch]
  (f/el
    (let [{:keys [caption deadline]} (<< !todo)]
      [:li caption
       (when deadline
         [:span " (due " deadline ")"])])))

(defn todo-list-component [!todos event-ch]
  (f/el
    [:ul
     (for [todo (->> (<< !todos)
                     (f/keyed-by :todo-id)
                     (sort-by :todo-order))]

       [todo-component (!<< todo) event-ch])]))

(f/root js/document.body
  (todo-list-component
    (atom [{:todo-id 49, :caption "Write talk",
            :deadline "Today", :todo-order 0}
           {:todo-id 58, :caption "Christmas shopping",
            :deadline "2014-12-24", :todo-order 2}
           {:todo-id 93, :caption "Clean house", :todo-order 1}])))
```

Write yourself a Flow

Steps for writing a Compiler:

- Input
- Lexical Analysis (tokenising)
- Syntax Analysis (parsing)
- Code Synthesis
- Output

Lexer / Parser

Synthesis - Translating the AST

```
(defmulti compile-el-form
  (fn [form opts]
    (el-type form)))

(defmethod compile-el-form :if [...]
  ...)

(defmethod compile-el-form :let [...]
  ...)

(defmethod compile-el-form :for [...]
  ...)

(defmethod compile-el-form :case [...]
  ...)

(defmethod compile-el-form :watch [...]
  ...)

(defmethod compile-el-form :dom-node [...]
  ...)

...
```

Our compilation target

```
DynamicElement :: AppState -> (Element, DynamicElement)


(letfn [(update-el! [el-state]
          (let [$new-el ...
                new-el-state ...]
            [$new-el #(update-el! new-el-state)]))]

  (update-el! initial-el-state))
```

A slight optimisation...

```
DynamicValue :: AppState -> Value
```

```
(defmulti compile-value-form
  (fn [form opts]
    (value-type form)))

(defmethod compile-value-form ... [...]
  ...)

...
```

Looking at ‘if’

```
(if <DynamicValue>
  <DynamicElement>
  <DynamicElement>)
```

```
(defmethod fc/compile-el-form :if [[_ test then else] opts]
  `(build-if (fn [] ~(fc/compile-value-form test opts))
             (fn [] ~(fc/compile-el-form then opts))
             (fn [] ~(fc/compile-el-form else opts))))
```

(Aside) Rules of Macro Club

2. Don't write macros.
3. (Unless you know you have to)

4. Get out of macro-land as soon as you can
    - Changing the execution order?
    - Analysing a form?

> *Macros are harder to write than ordinary LISP functions, and it's considered to be bad style to use them when they're not necessary.*
>
> *Paul Graham (via Stuart Sierra)*

Looking at ‘if’

```
(if <DynamicValue>
  <DynamicElement>
  <DynamicElement>)
```

```
(defmethod fc/compile-el-form :if [[_ test then else] opts]
  `(build-if (fn [] ~(fc/compile-value-form test opts))
             (fn [] ~(fc/compile-el-form then opts))
             (fn [] ~(fc/compile-el-form else opts))))
```

The Flow runtime

```
(defn build-if [test-fn build-then build-else]
  (fn []
    (letfn [(update-if [old-test-value update-current-branch!]
              (let [new-test-value (boolean (test-fn))

                    new-branch (if (and update-current-branch!
                                        (= old-test-value new-test-value))

                                 update-current-branch!

                                 (if new-test-value
                                   (build-then)
                                   (build-else)))

                    [$branch-el update-branch!] (new-branch)]

                [$branch-el #(update-if new-test-value update-branch!)]))]

      (update-if nil nil))))
```

What's missing from this ClojureScript function?

```
(defn build-if [test-fn build-then build-else]
  (fn []
    (letfn [(update-if [old-test-value update-current-branch!]
              (let [new-test-value (boolean (test-fn))

                    new-branch (if (and update-current-branch!
                                        (= old-test-value new-test-value))

                                 update-current-branch!

                                 (if new-test-value
                                   (build-then)
                                   (build-else)))

                    [$branch-el update-branch!] (new-branch)]

                [$branch-el #(update-if new-test-value update-branch!)]))]

      (update-if nil nil))))
```

Progress

```
(defmulti compile-el-form
  (fn [form opts]
    (el-type form)))

(defmethod compile-el-form :if [...]
  ;; Done :)
  ...)

(defmethod compile-el-form :let [...]
  ...)

(defmethod compile-el-form :for [...]
  ...)

(defmethod compile-el-form :case [...]
  ...)

(defmethod compile-el-form :watch [...]
  ...)

(defmethod compile-el-form :dom-node [...]
  ...)

...
```

What's next for Flow?

- A stable release :)
- More tutorials, example applications and docs
- Next version(s) ??

What can you do now?

- Feedback on this talk?
- Give Flow a try!

```
lein new splat flow-hacking
```

- Get involved!

Thank you!

https://github.com/james-henderson/flow (https://github.com/james-henderson/flow)

James Henderson

github: james-henderson (https://github.com/james-henderson)

twitter: @jarohen (https://twitter.com/jarohen)

james@jarohen.me.uk (mailto:james@jarohen.me.uk)