



pedestal

 **relevance**

Introducing Pedestal

- Who: Relevance
- What: alpha release, open source libs
- Where: Clojure/West
- When: Now
- Why, How...

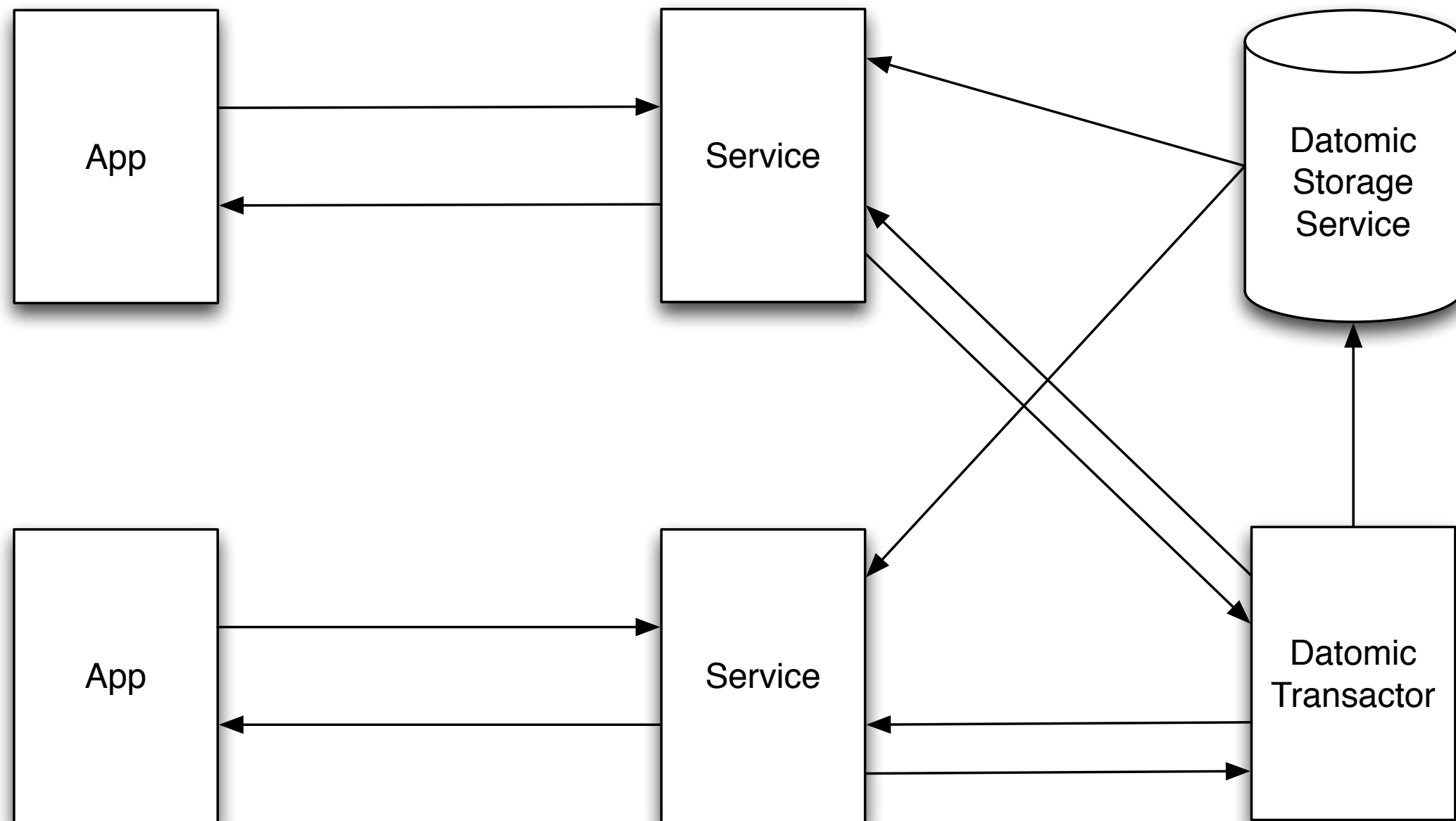


Goal

- Use clojure end-to-end to build rich interactive collaborative Web applications and services that scale



Archetype



Problems

- Services notifying apps
- Building complex UIs in browser



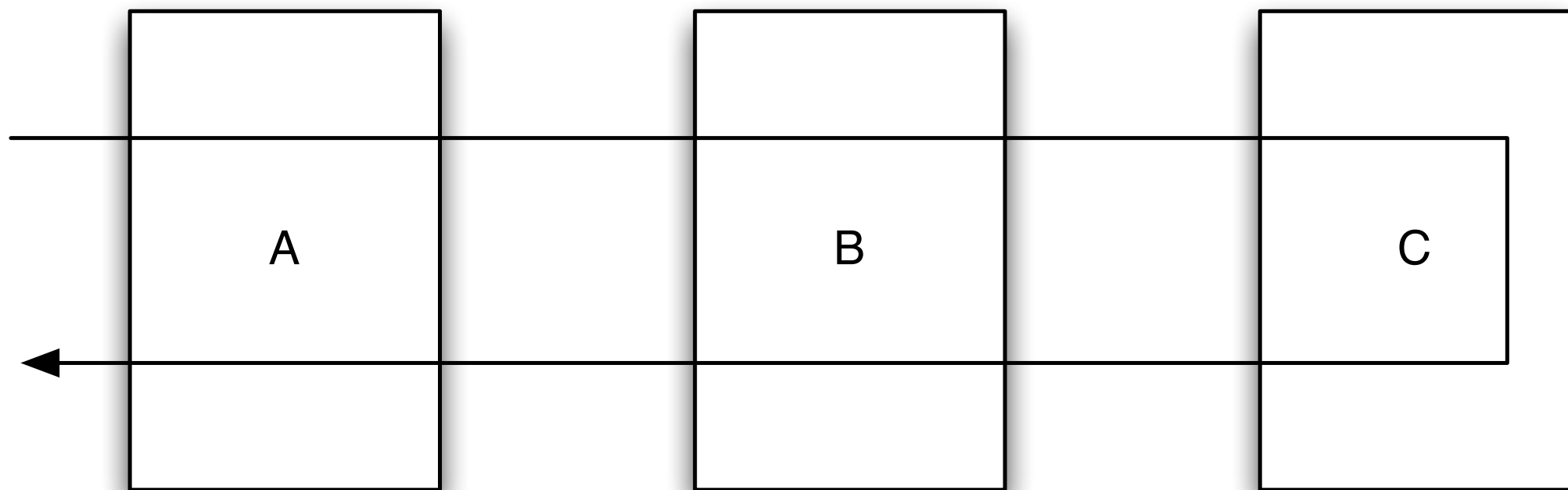
Service Plumbing

- Interceptor mechanism
- Long polling, server-sent events
- Routing, url generation



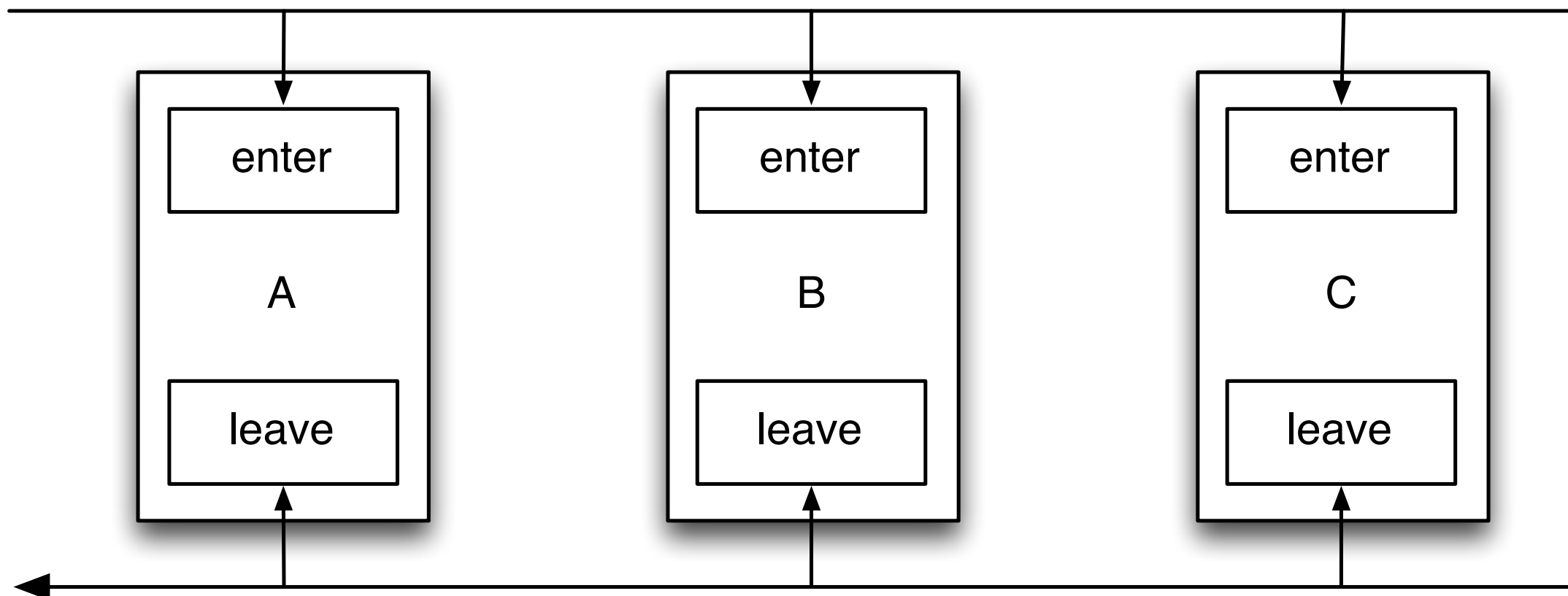
Ring Middlewares

- Chained fns bound to thread's stack



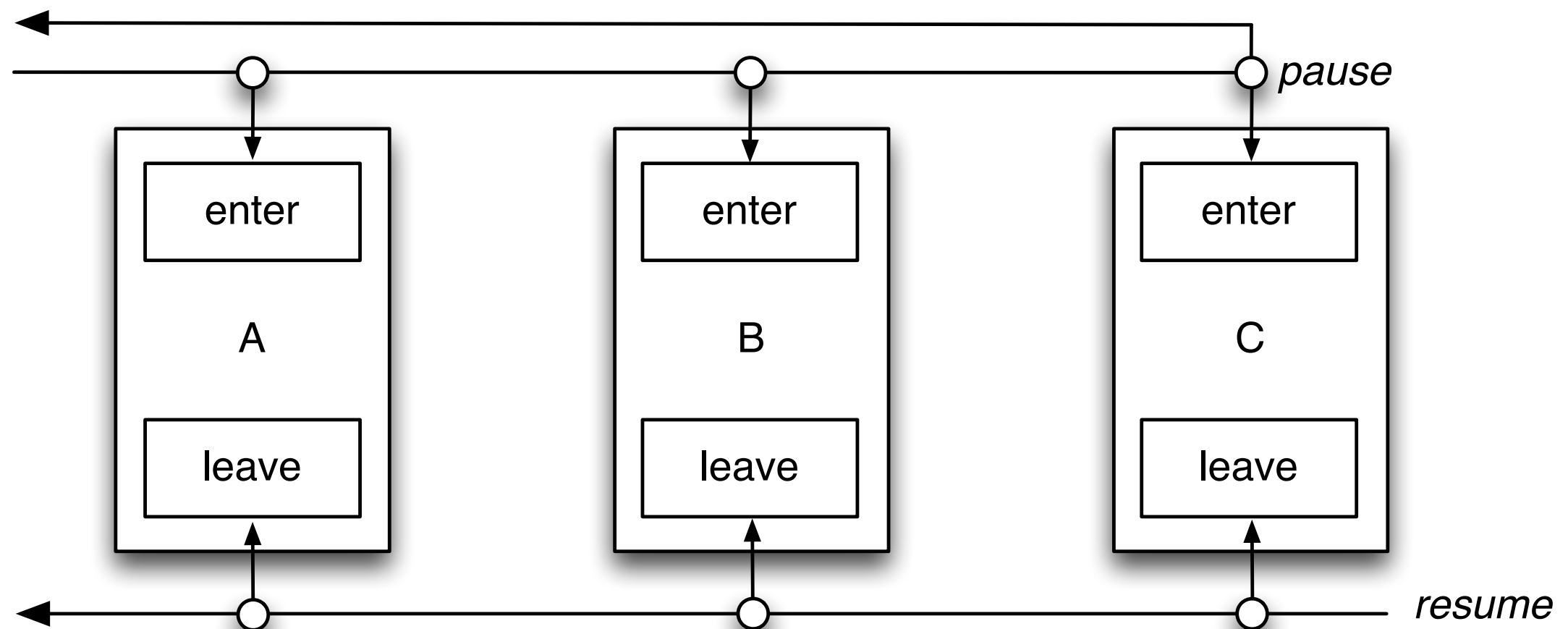
Interceptors

- Maps of fns not bound to thread's stack



Pause/Resume

- Can pause/resume across threads
- Supports bindings and error propagation



Ring Compatibility

- As compatible as possible
- Same request/response maps
- Core middlewares refactored and accepted
 - Interceptor versions provided
- Easy to port existing code



Notifications

- Thread management enables long polling
 - Park request as needed
- Also, server-sent-events
 - Built on low-level streaming API



Routes and URLs

ring handler fn



```
(defn hello-world [req]
  (ring/response (map inc [1 2 3])))
```

native edn serialization



```
(defroutes routes
  [[["/hello-world" {:get hello-world}]]])
```

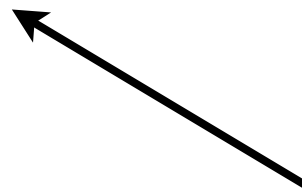
routes as data



```
(def url-for (routes/url-for-routes routes))
```

```
(url-for ::hello-world)
;;=> "/hello-world"
```

make urls
from routes



Problems

- ~~Services notifying apps~~
- Building complex UIs in browser



3 Simple Steps

- Event handler affects state
- *Figure out what changed*
- Update DOM

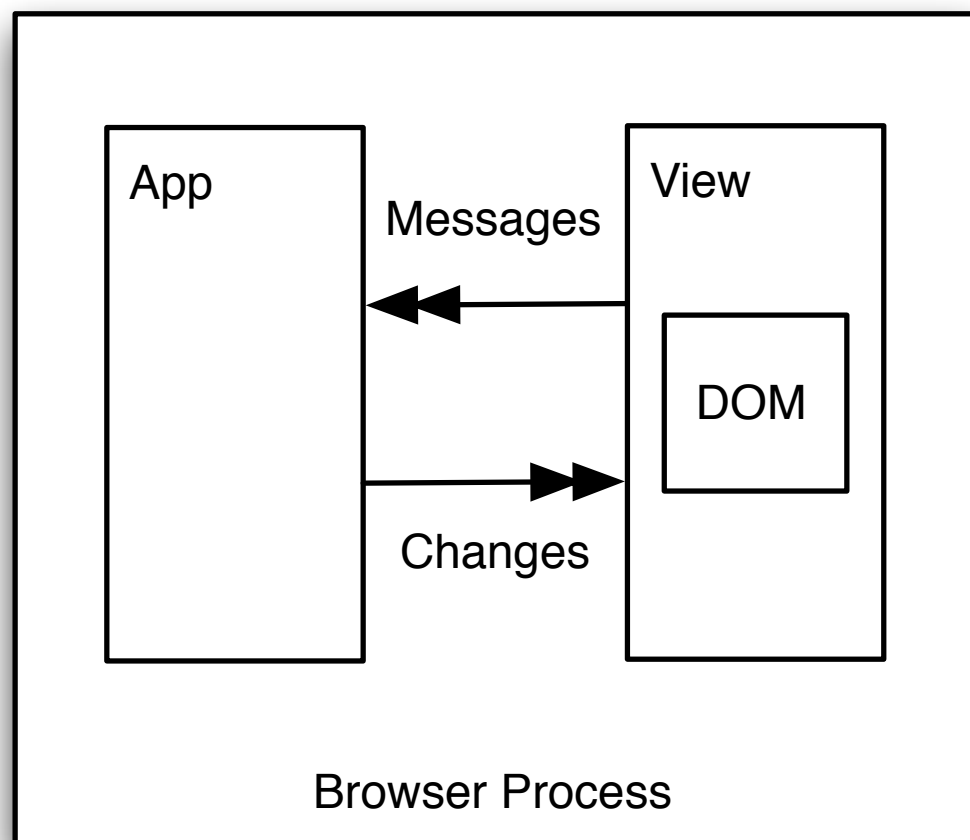


What to compare?

- JS: event, old OR new state, DOM
- CLJS: event, old AND new state, DOM
- Can remove DOM from equation!



App vs. View



- App: behavior
- View: presentation



App Model

- Encapsulate behavior and state
- Input: messages
- Output: app tree deltas
- Implemented as pure functions
- Fns wired up declaratively



Messages

- Map with topic and type
- Other keys as needed
- Used for input to app
- Used to control aspects of engine

```
{msg/topic :count-transform  
  msg/type :inc  
  :key :a}
```



App Tree Deltas

op path args

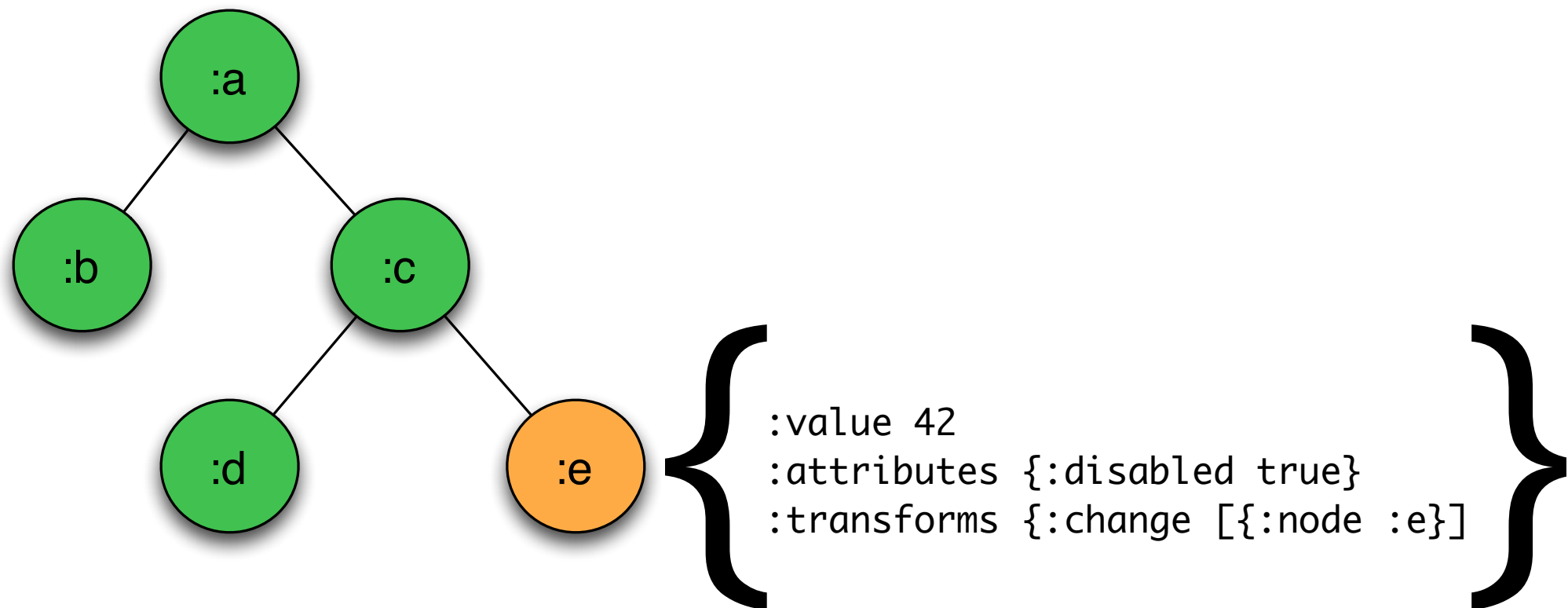
↓ ↓ ↓

```
[ :node-create [ :a :b :c ] :map ]
[ :node-destroy [ :a :b :c ] ]
[ :value [ :a :b :c ] { :count 2 } ]
[ :attr [ :a :b :c ] :active true ]
[ :transform-enable [ :a :b :c ] :send-info
                                [ { msg/topic :some-model
                                  msg/type :send-name
                                  (msg/param :name) {} } ] ]
[ :transform-disable [ :a :b :c ] :send-info ]
```

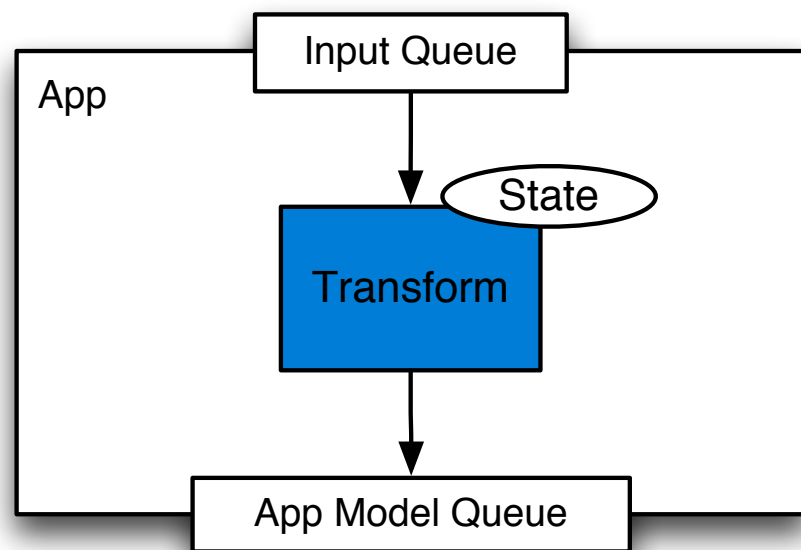


App Tree is Logical

- Consumer *may or may not* realize (portions of) tree as real structure



Transform

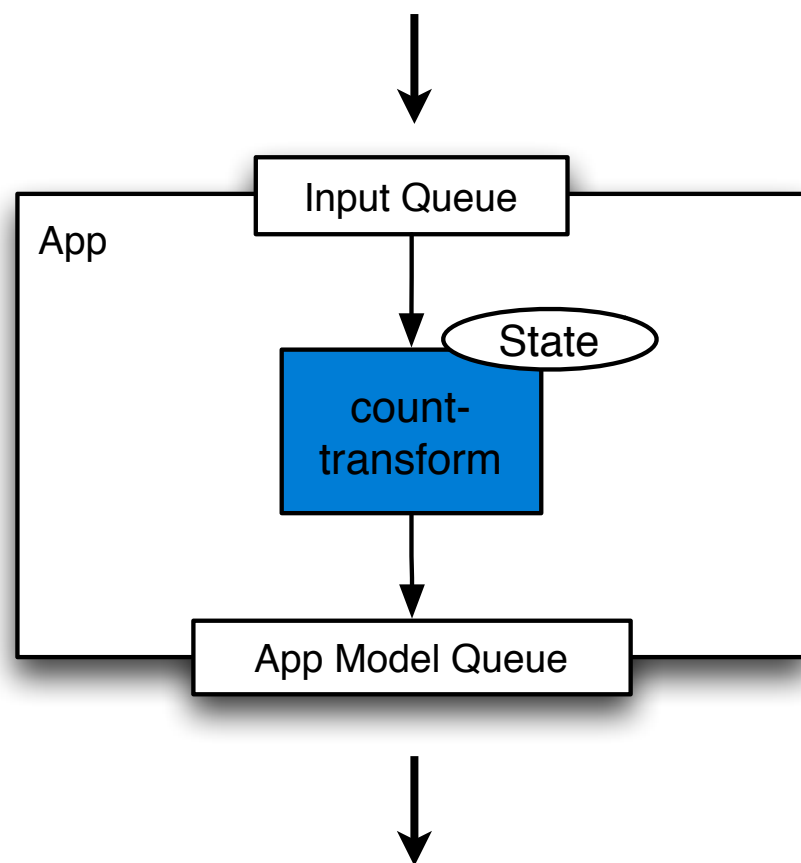


- Fn that modifies state
- Message, state input
- Last output state kept
- Only changes flow



Transform

```
(put-message (:input-queue app)
  {msg/topic :count-transform msg/type :inc})
```



```
(defn count-transform [t-state message]
  (condp = (msg/type message)
    msg/init (:value message)
    :inc (inc (or t-state 0))
    t-state))
```

```
([:value [:io.pedestal.app/view-count-transform] 10 11])
```



Transform output

```
;; message input...
{msg/topic :count-transform msg/type :inc}

;; deltas output...
([:value [:io.pedestal.app/view-count-transform] 10 11])

;; message input...
{msg/topic :count-transform msg/type :inc}

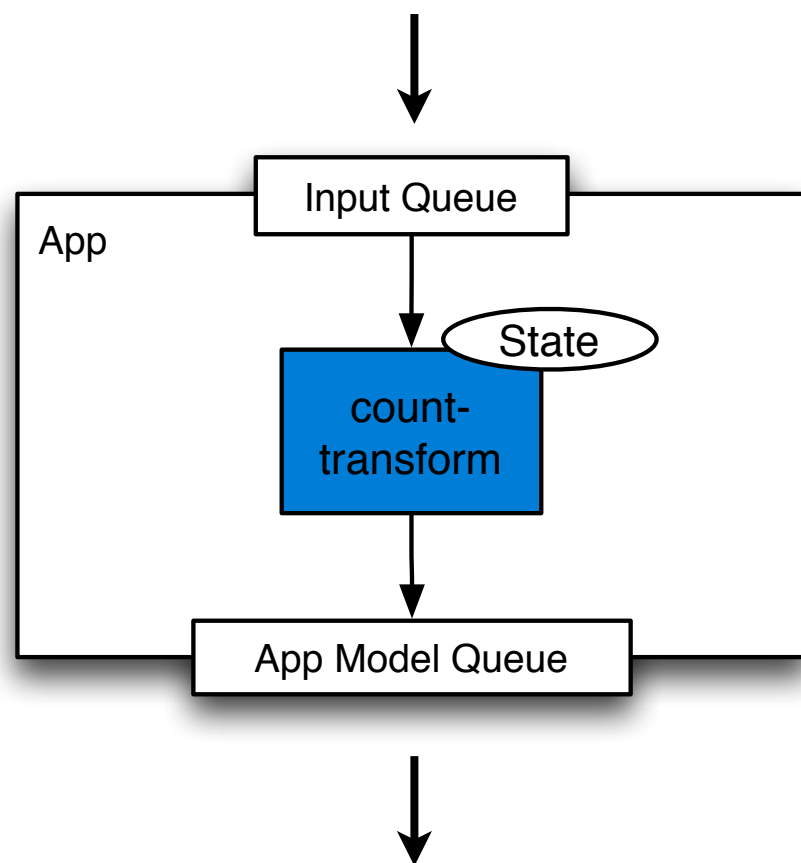
;; deltas output...
([:value [:io.pedestal.app/view-count-transform] 11 12])

...
```



More State

```
(put-message (:input-queue app)
  {msg/topic :count-transform msg/type :inc :key :a})
```



```
(defn count-transform [t-state message]
  (condp = (msg/type message)
    msg/init (:value message)
    :inc (update-in (or t-state {})
      (:key message)
      inc)
    t-state))
```

```
([ :value [:io.pedestal.app/view-count-transform]
  { :a 10 :b 9 } { :a 11 :b 9 } ])
```



Affecting Parts of State

```
;; put a message in...  
{msg/topic :count-transform msg/type :inc :key :a}
```

```
;; get deltas out...  
([:value [:io.pedestal.app/view-count-transform]  
  {:a 10 :b 9} {:a 11 :b 9}])
```

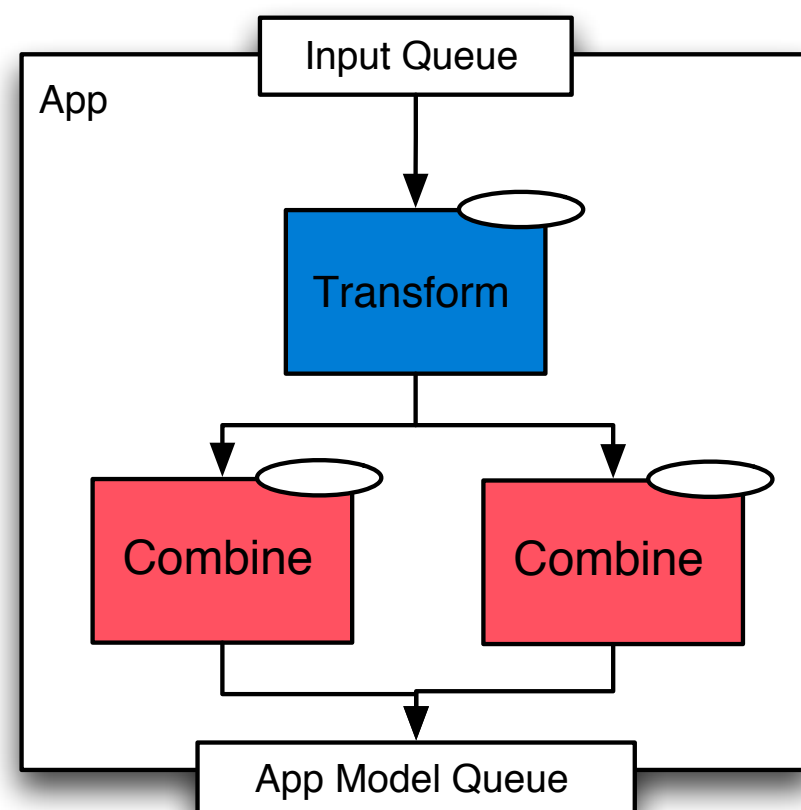
```
;; put a message in...  
{msg/topic :count-transform msg/type :inc :key :b}
```

```
;; get deltas out...  
([:value [:io.pedestal.app/view-count-transform]  
  {:a 11 :b 9} {:a 11 :b 10}])
```

...



Combine

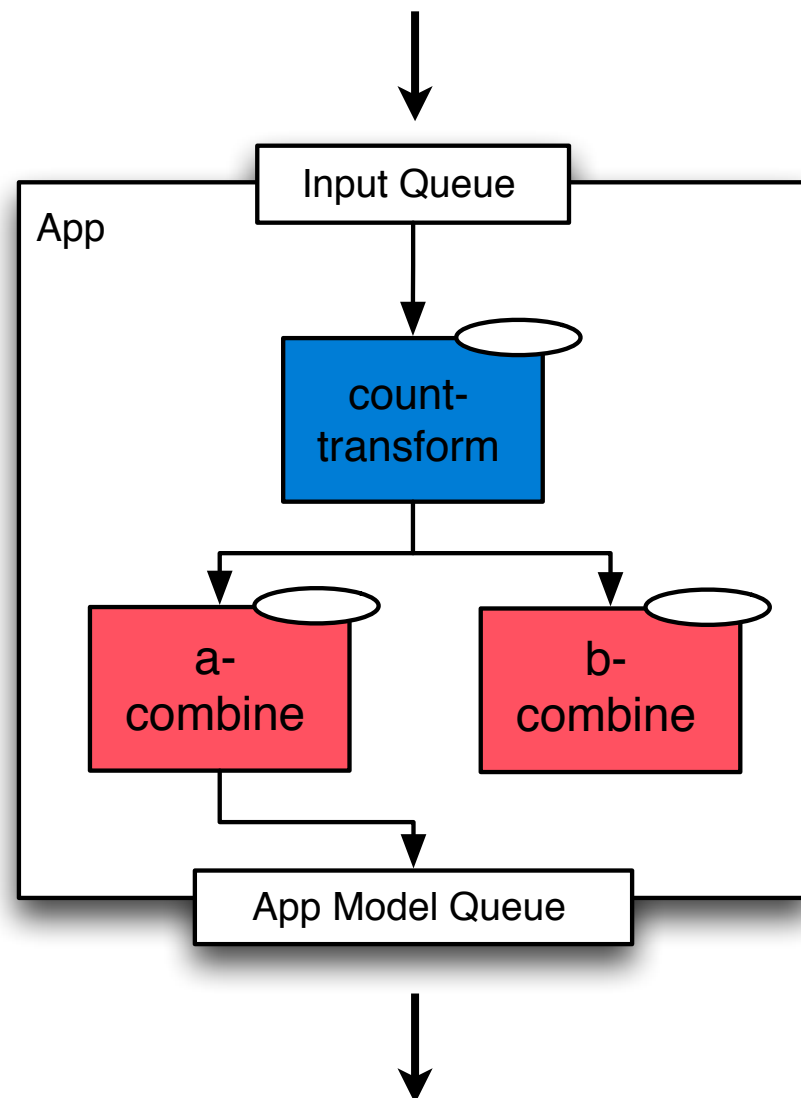


- Fn that merges or splits state(s)
- Transform and/or combine state(s) input
- Engine keeps last output
- Only changes flow



Combine

```
{msg/topic :count-transform msg/type :inc :key :a}
```



```
(defn a-combine [c-state t-name  
                 t-old-val t-new-val]  
  (:a t-new-val))
```

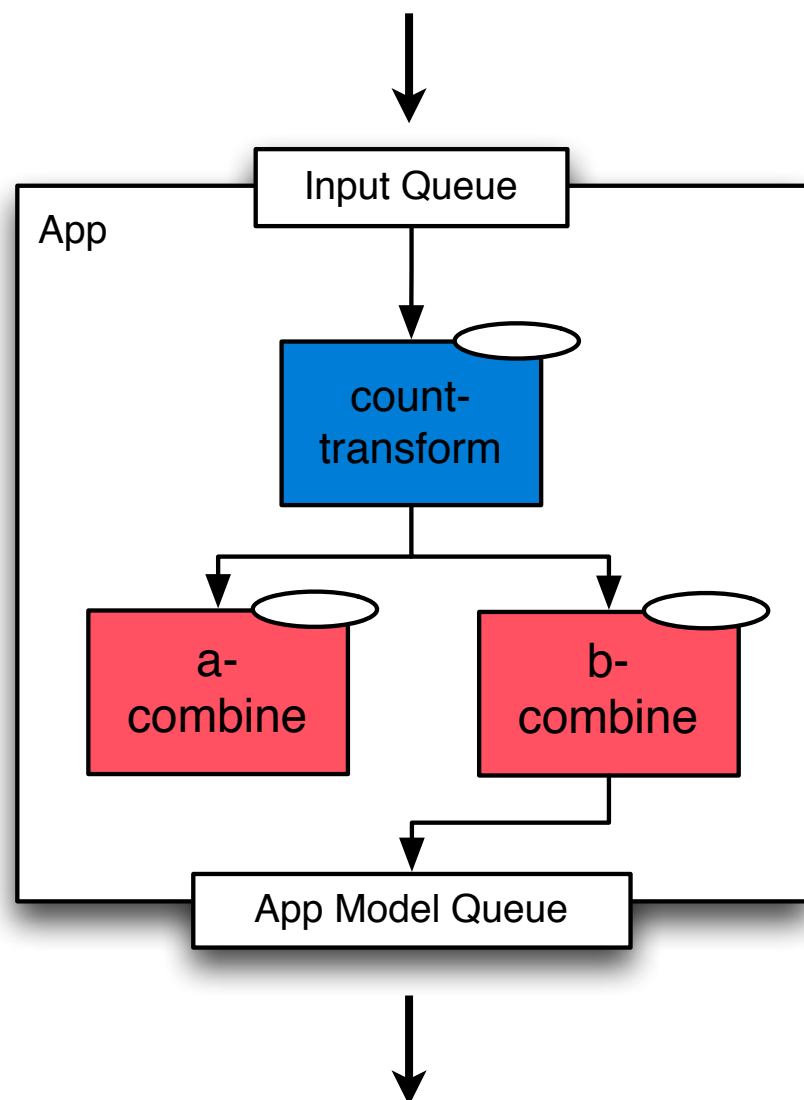
```
(defn b-combine [c-state t-name  
                 t-old-val t-new-val]  
  (:b t-new-val))
```

```
([:value [:a-combine] 10 11])
```



Combine

```
{msg/topic :count-transform msg/type :inc :key :b}
```



```
(defn a-combine [c-state t-name  
                 t-old-val t-new-val]  
  (:a t-new-val))
```

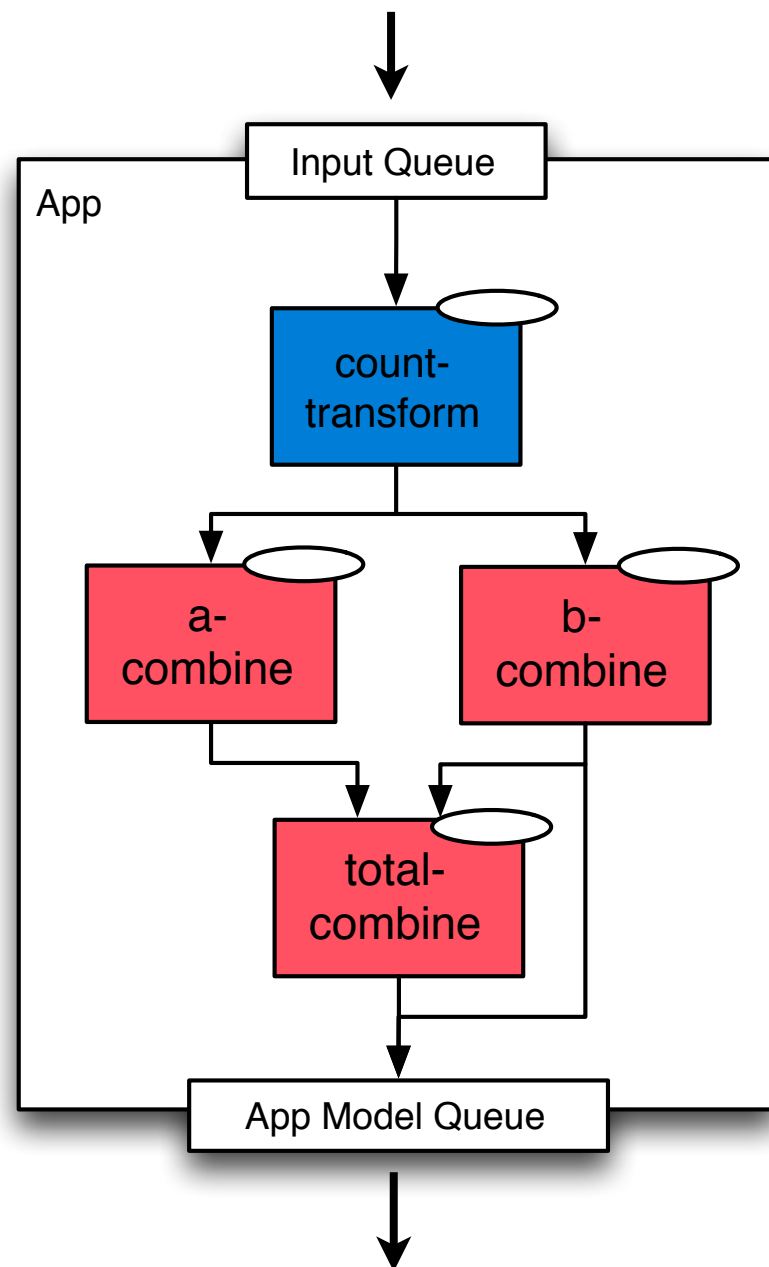
```
(defn b-combine [c-state t-name  
                 t-old-val t-new-val]  
  (:b t-new-val))
```

```
([:value [:b-combine] 9 10])
```



Combine

```
{msg/topic :count-transform msg/type :inc :key :b}
```



```
(defn a-combine [c-state t-name  
                 t-old-val t-new-val]  
  (:a t-new-val))
```

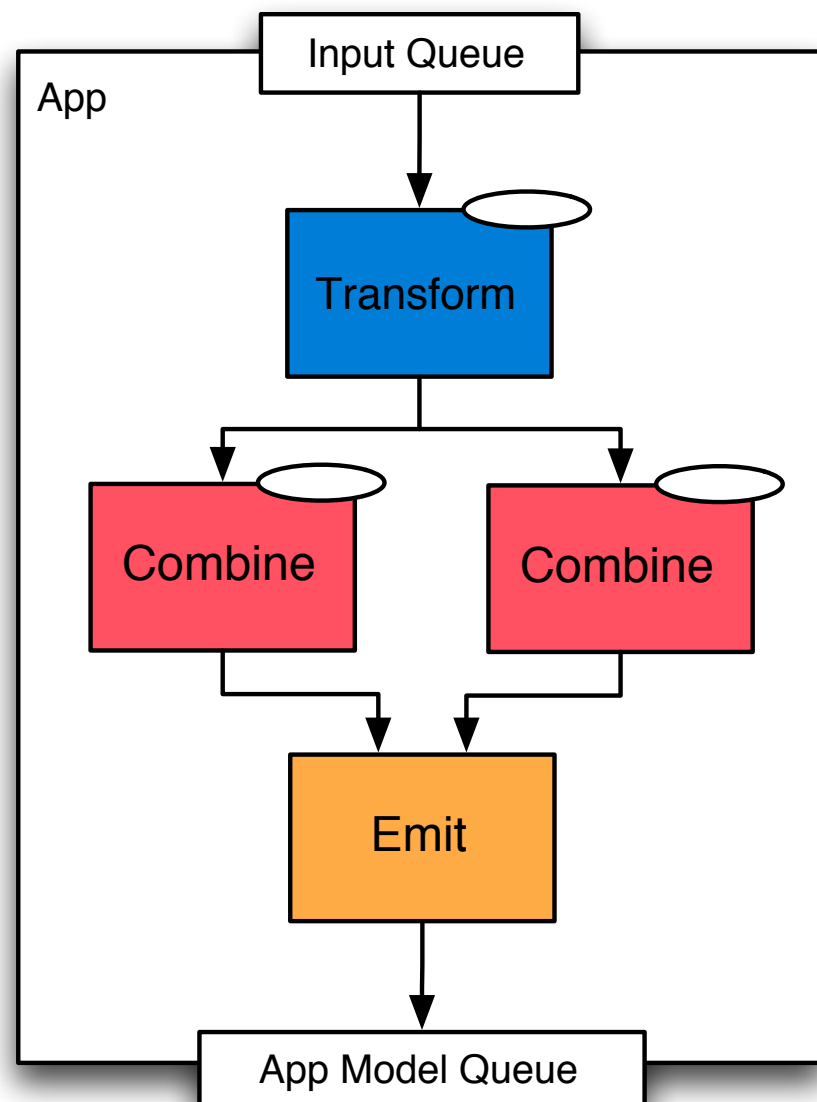
```
(defn b-combine [c-state t-name  
                 t-old-val t-new-val]  
  (:b t-new-val))
```

```
(defn total-combine [c-state inputs]  
  (apply + (map :new (vals inputs))))
```

```
([:value [:b-combine] 10 11] [:value [:total-combine] 21 22])
```



Emit

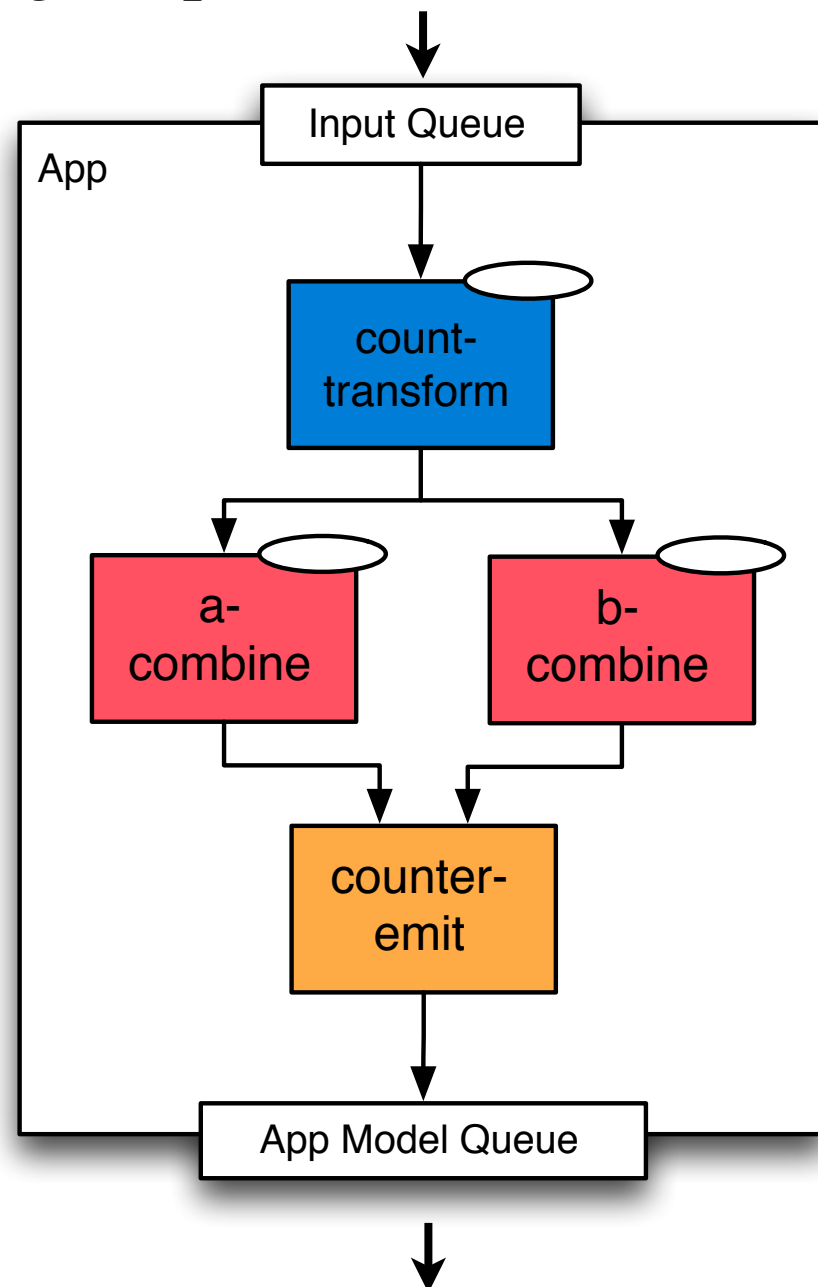


- Fn that converts state(s) to tree deltas
- Overrides default tree mapping



Emit

```
{msg/topic :count-transform msg/type :inc :key :b}
```



```
(defn counter-emit
  ([inputs] [{:counter {:a {:value 0}
                          :b {:value 0}}}]
  ([inputs changed-inputs]
    (concat []
      (when (changed-inputs :a-combine)
        [[:value [:counter :a]
          (-> inputs :a-view :new)]]))
      (when (changed-inputs :b-combine)
        [[:value [:counter :b]
          (-> inputs :b-view :new)]]))))))
```

```
([:value [:counter :b] 11 12])
```

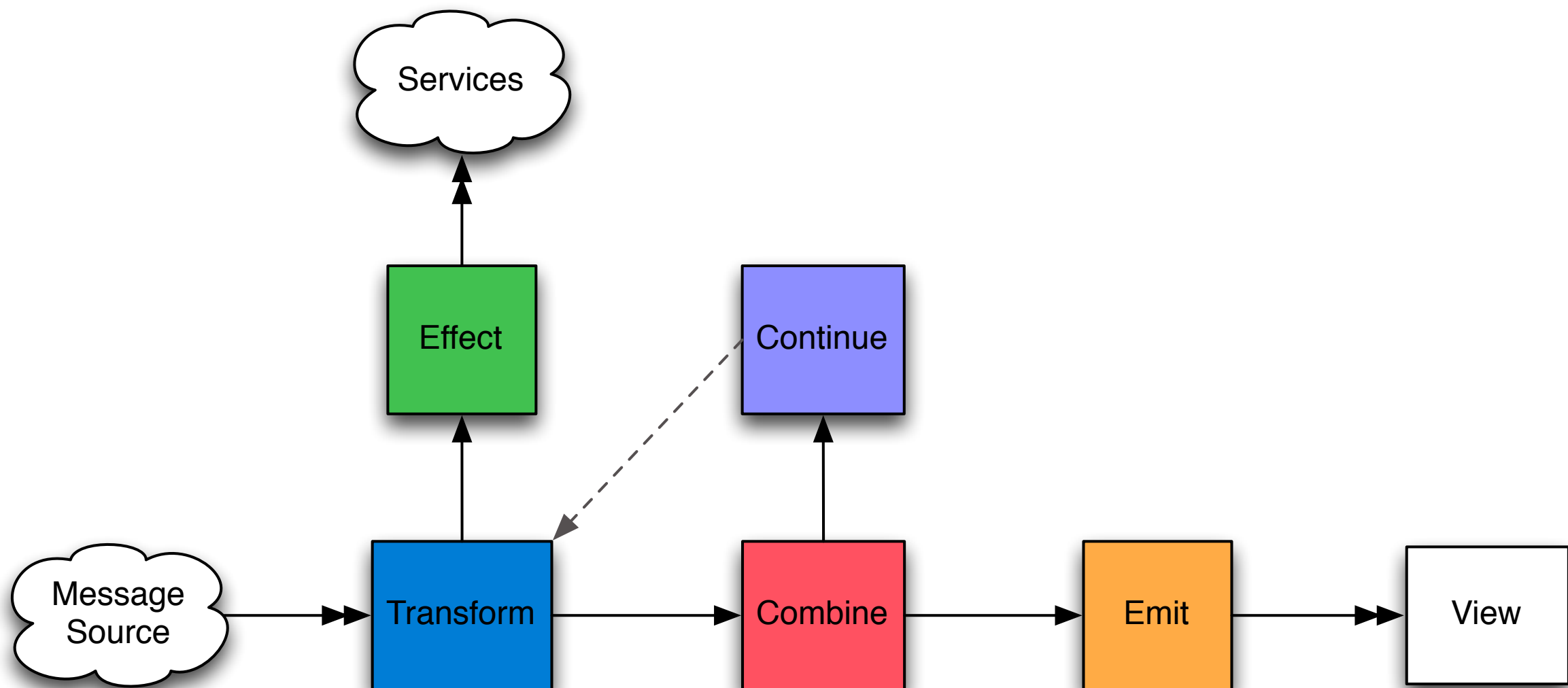


Messages in a flow

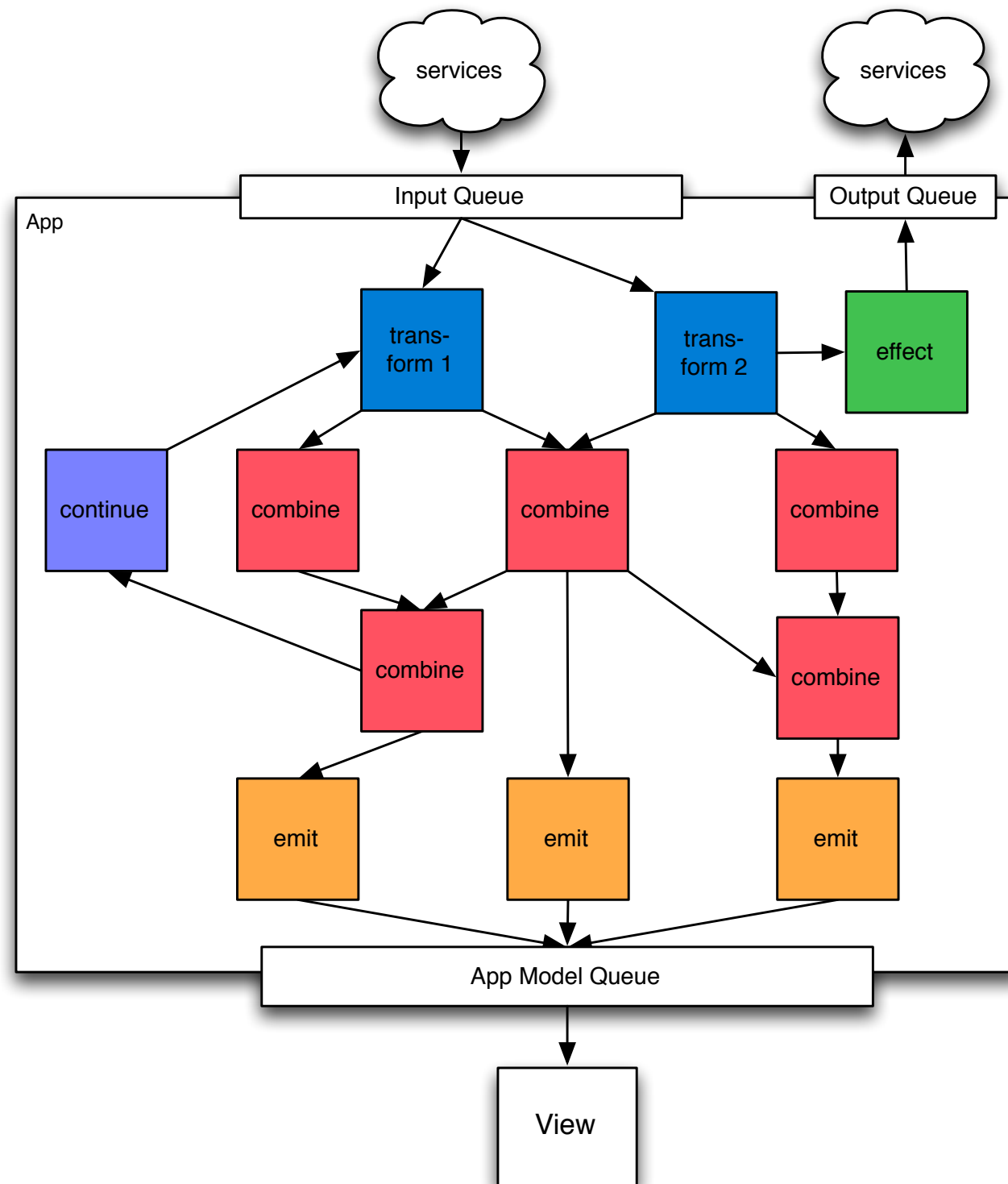
- Effect fn
 - transform or combine state input
 - msgs for services output
 - queued *after* flow
- Continue fn
 - combine state input
 - msgs for transforms output
 - sent *during* flow



All the pieces...

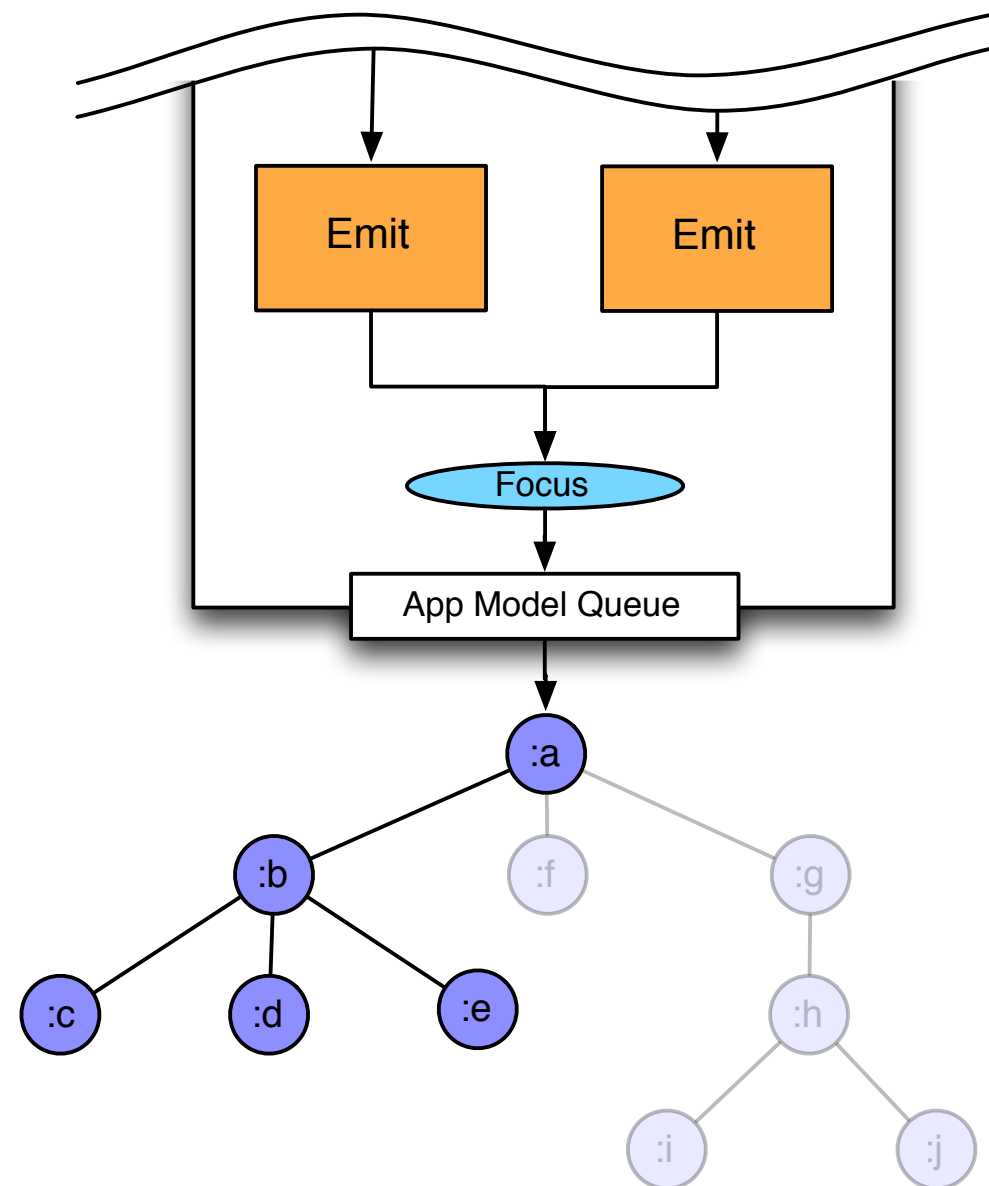


...put together



Focus

- Focus filters deltas
 - by name
 - by path
- Set by consumer
 - defaults to all
- Helpful “navigation”



Benefits of data flow

- Write small pure fns
- No big comparator
- Let engine track state changes
- Only the necessary fns get called
- Projects all the way out to consumer

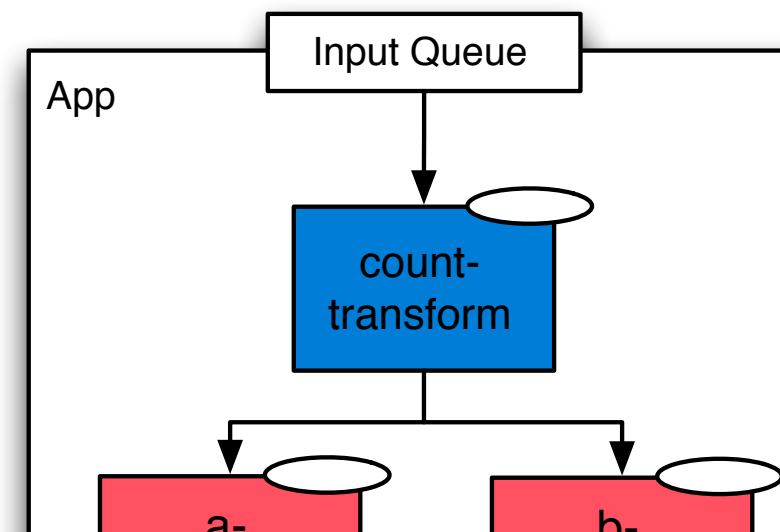


Making an App

```
(def counter-dataflow
  {:transform {:count-transform {:init {:a 0 :b 0}
                                   :fn transform-count}}
   :combine {:a-combine {:fn a-combine
                           :inputs #{:count-transform}}
             :b-combine {:fn b-combine
                           :inputs #{:count-transform}}}}
  :emit {:counter-emit {:fn counter-emit
                        :inputs #{:a-combine :b-combine}}}}))
```



```
(def app (app/build counter-dataflow))
```



Consuming App Output

- App produces logical tree deltas
- Provide a fn to consume them

```
(defn console-renderer [out]
  (fn [deltas input-queue]
    (binding [*out* out]
      (doseq [d deltas] (println d))))))
```

```
(def app-model
  (render/consume-app-model app (console-renderer *out*)))
(app/begin app)
```



View Model

- Encapsulate presentation logic and state
- Input: deltas from logical app tree
- Output: messages
- Implemented as fn(s) that
 - update UI
 - handle events



Push Renderers

- Map tree deltas to fns
- Maintain structure for portion(s) of tree in focus

```
(def render-config
  [[:node-create []
    render-page]
   [:value [:counter :a]
    render-a-view]
   [:value [:counter :b]
    render-b-view]])
```

Diagram annotations:

- op**: points to `render-config`
- path**: points to `[:node-create []`
- fn**: points to `render-page]`

```
(def app-model
  (render/consume-app-model
   app
   (push/renderer
    render-config)))
```



Simple render fns

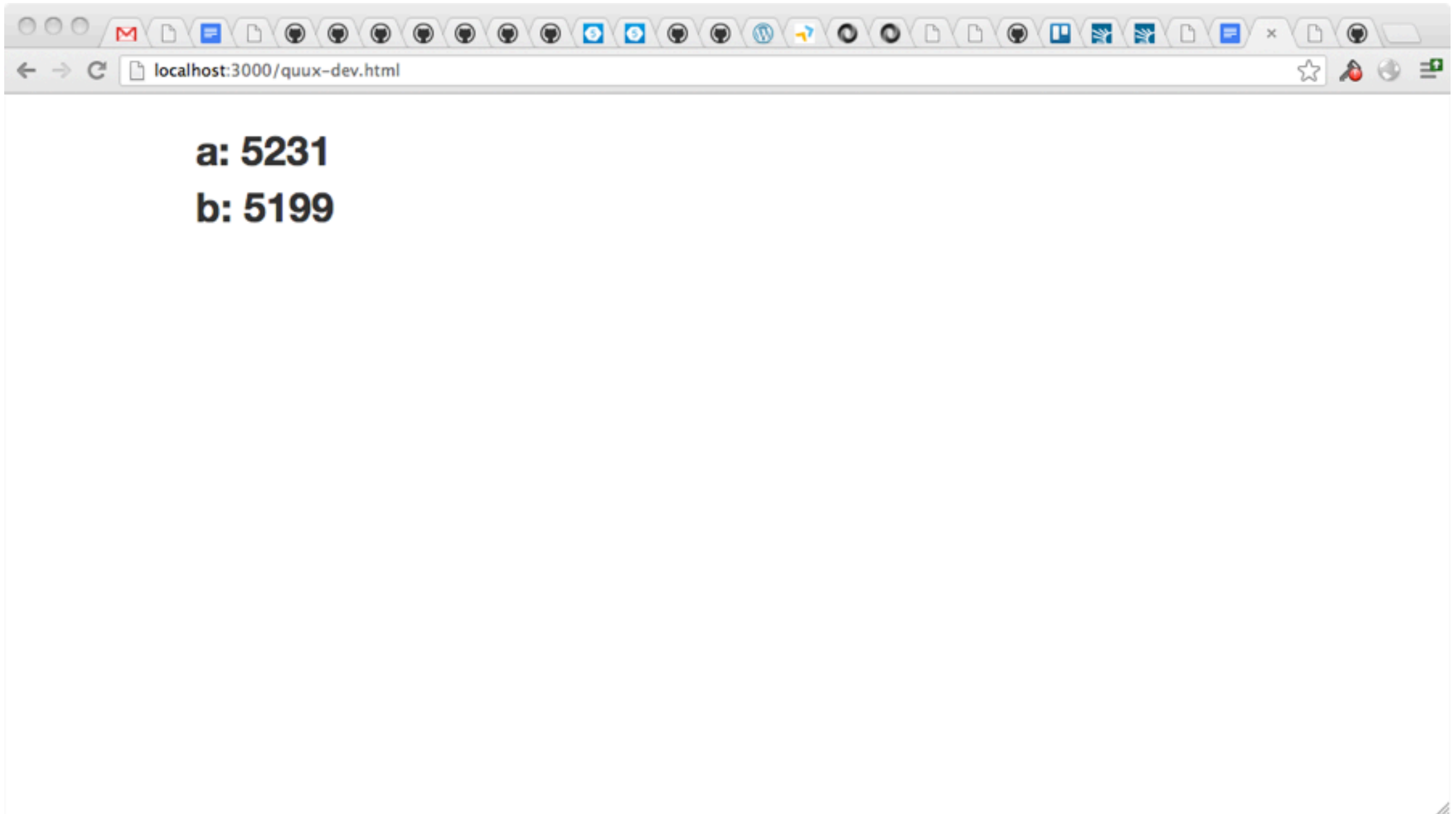
```
(defn render-page
  [renderer [op path old-value new-value] input-queue]
  (dom/append! (dom/by-id "content") "<h1 id=\"a\">a</h1>")
  (dom/append! (dom/by-id "content") "<h1 id=\"b\">b</h1>"))
```

```
(defn render-a-view
  [renderer [op path old-value new-value] input-queue]
  (dom/set-text! (dom/by-id "a") (str "a: " new-value)))
```

```
(defn render-b-view
  [renderer [op path old-value new-value] input-queue]
  (dom/set-text! (dom/by-id "b") (str "b: " new-value)))
```



Result!



Problems

- ~~Services notifying apps~~
- ~~Building complex UIs in browser~~



App / View Benefits

- Clean separation of concerns
- Build, test app outside browser
- Generic data renderer can drive app before UI is ready
- Record/playback changes to build, test, debug rendering code



Getting Started

- Run chat sample, look at app and service code
- `lein new pedestal-service my-service`
- `lein new pedestal-app my-app`



Thanks!

- <http://pedestal.io>
- <http://thinkrelevance.com>

