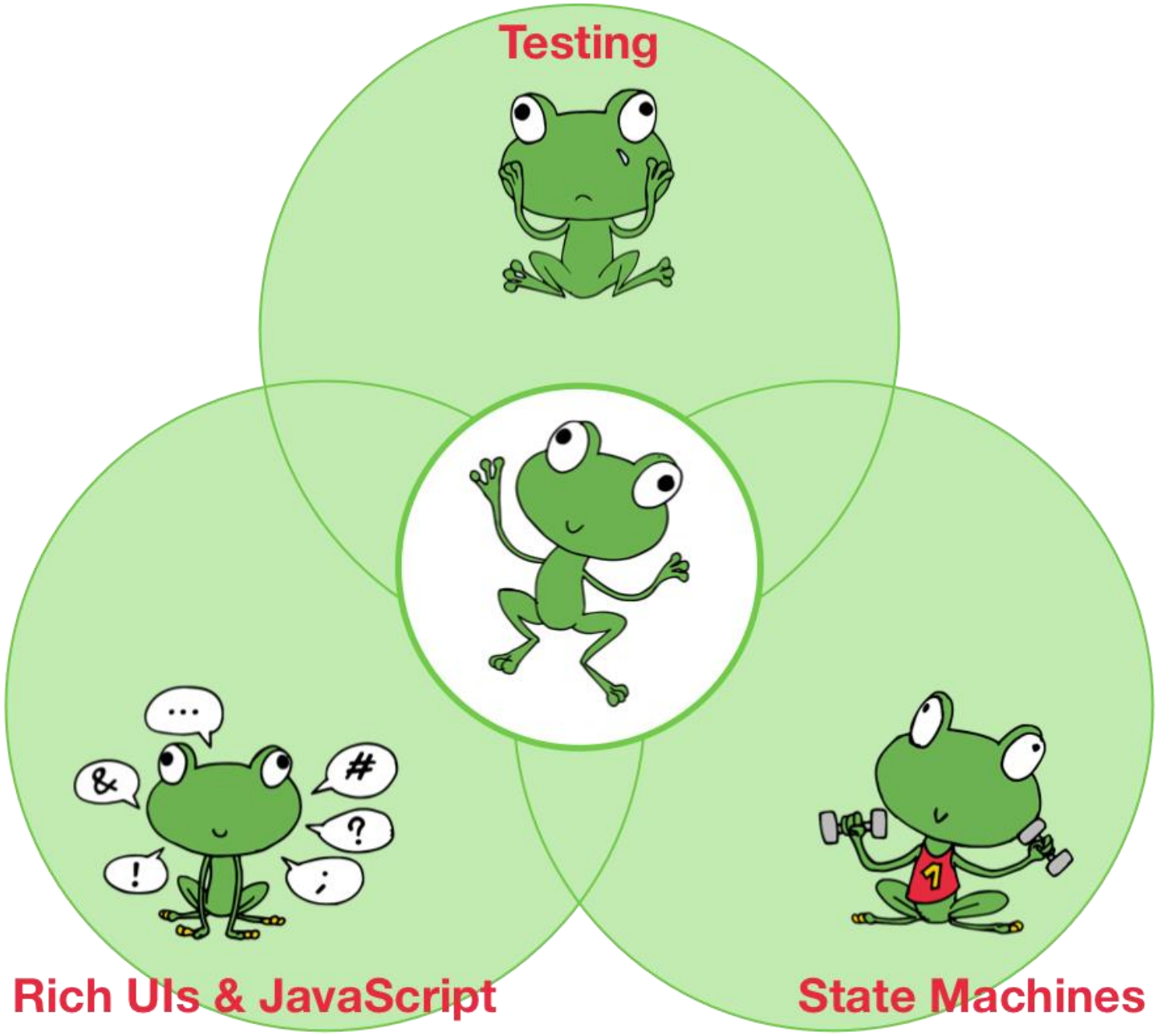
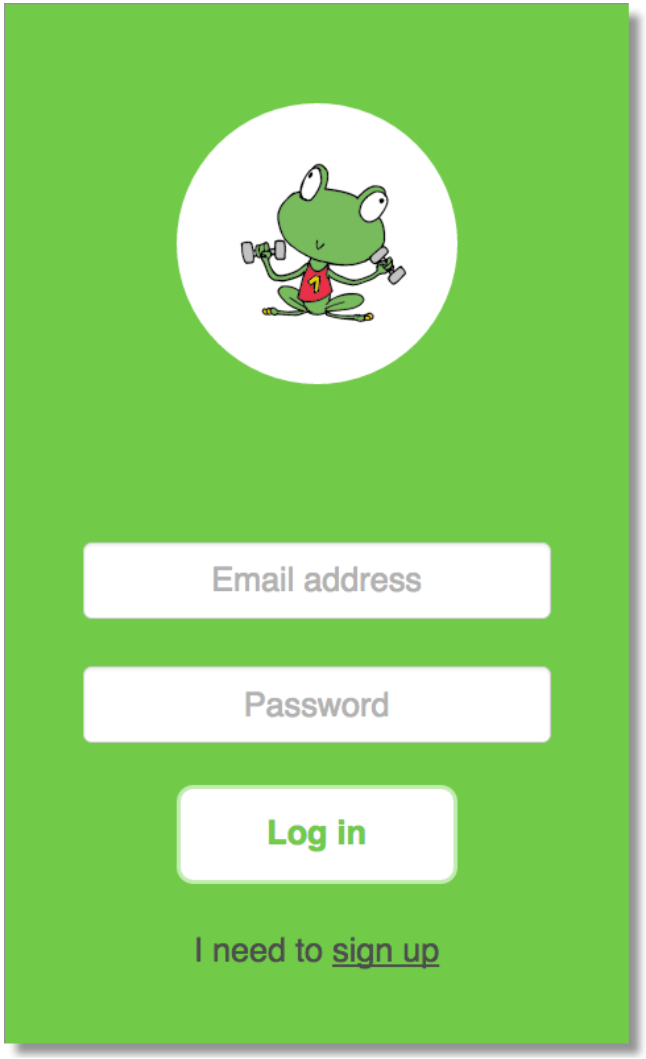
Functional CRUD

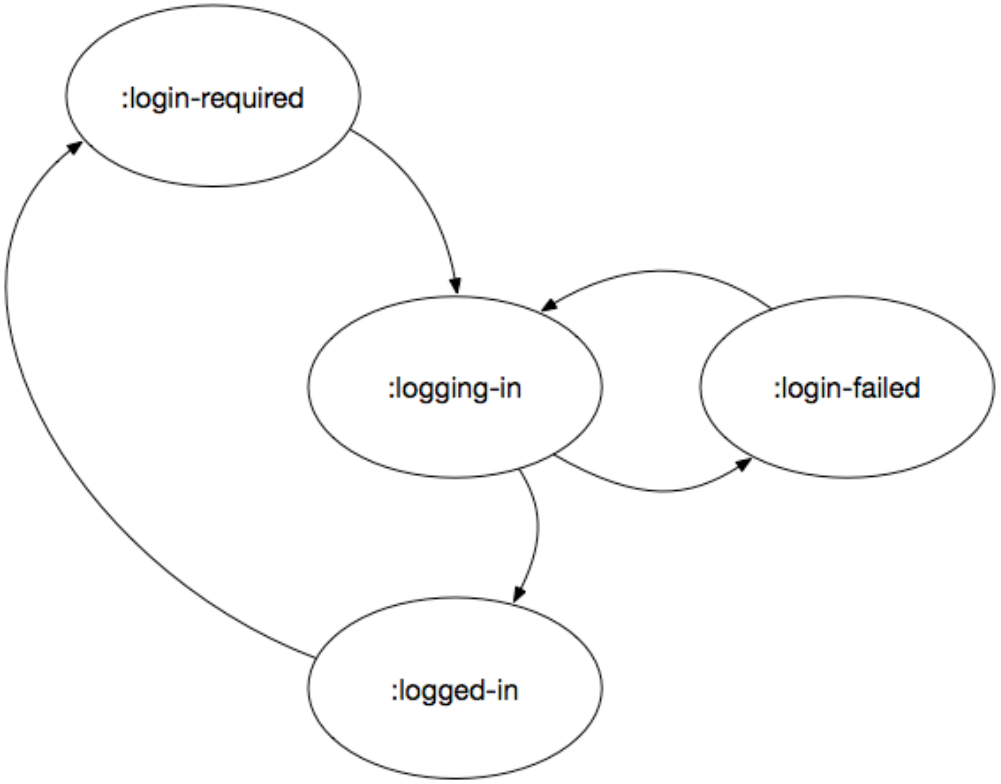
Using ‘bureaucracy’ to
tame a full-stack
Clojure/ClojureScript app

Sam Roberton – @sroberton
github.com/samroberton/bureaucracy

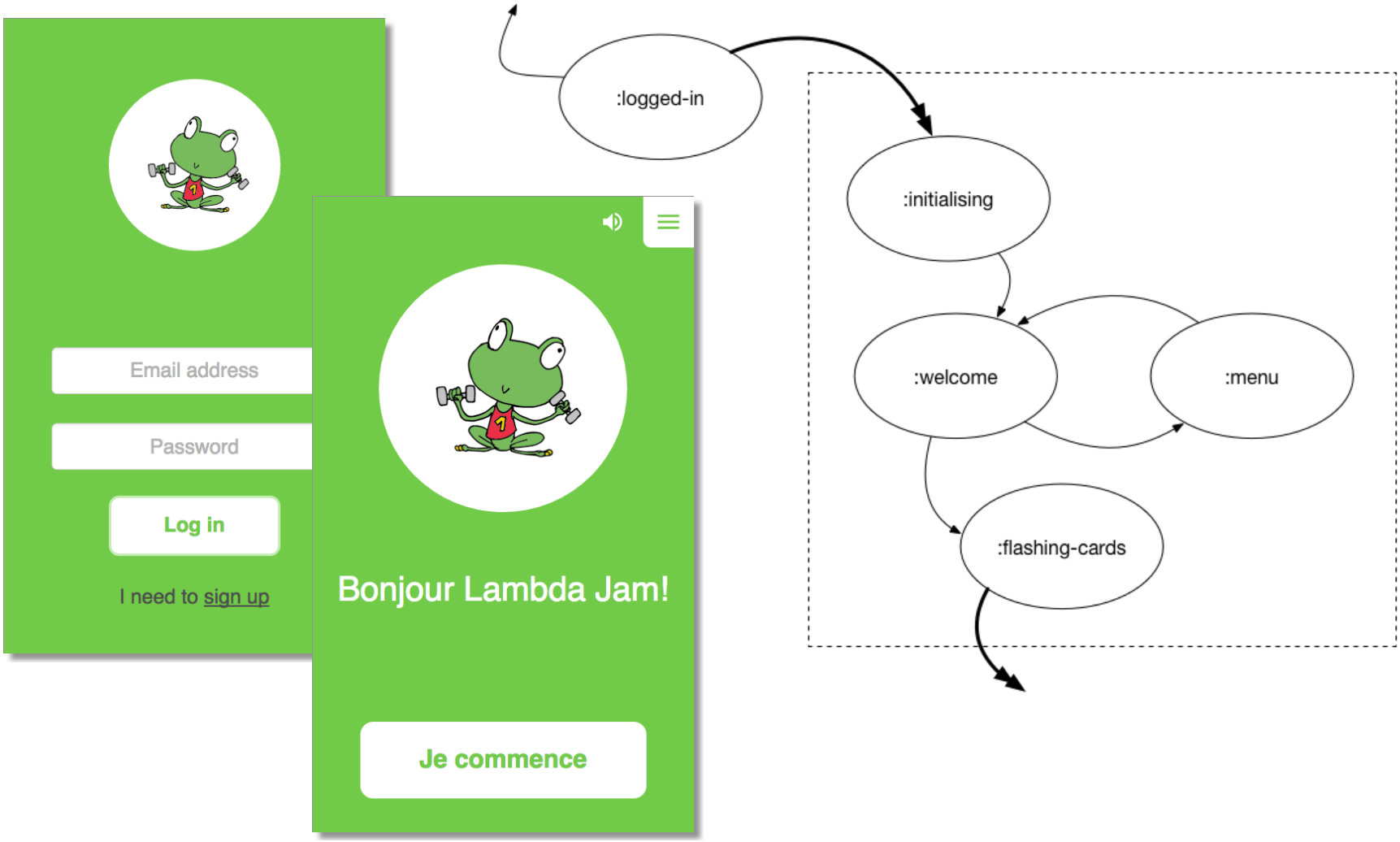

Testing
Rich UIs & JavaScript
State Machines

State machines

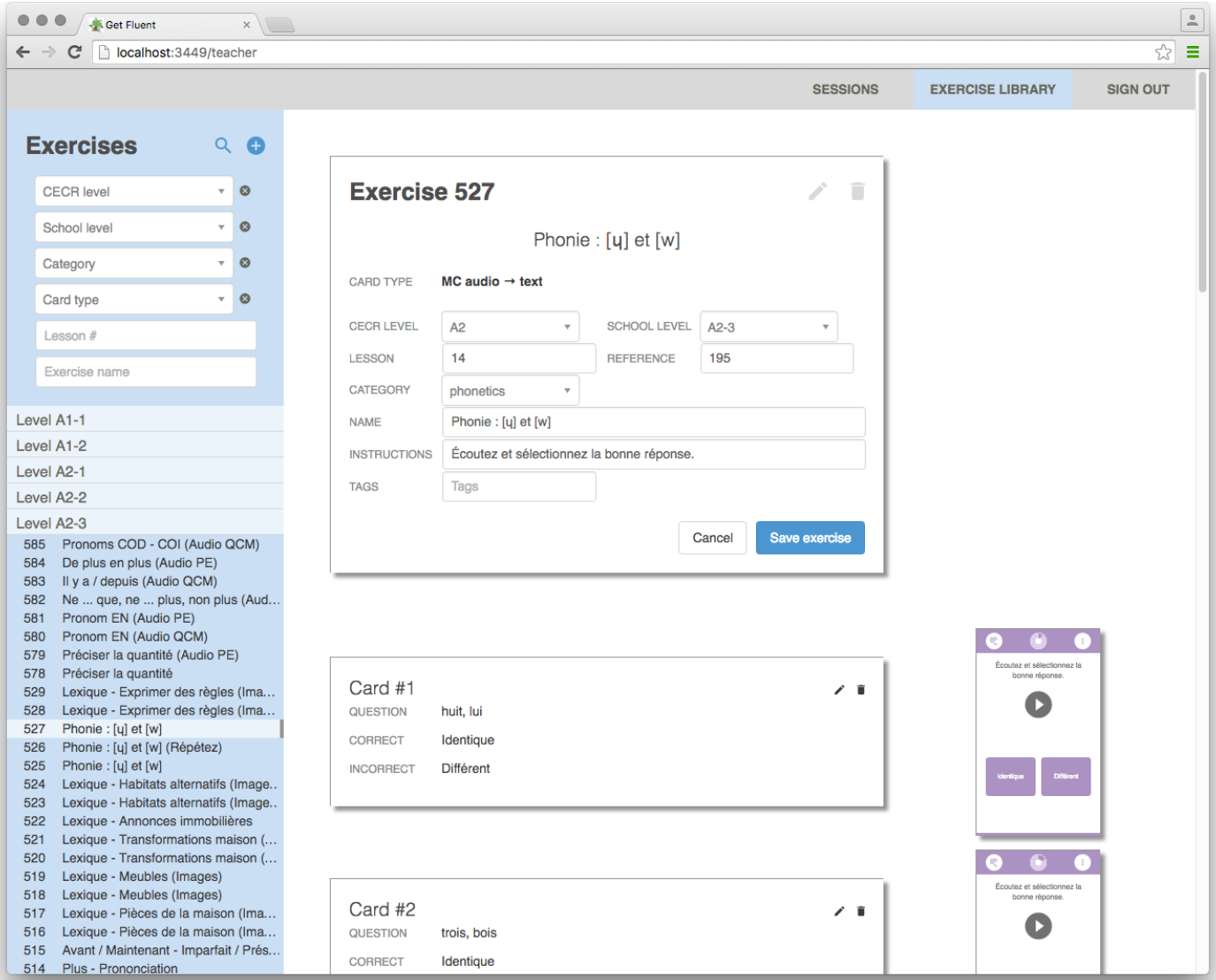




Composeable state machines



Composeable state machines



How do we win?

- portable Clojure
  - test code which targets the browser alongside code which targets the server
- *complete* separation of view from state and behaviour
  - view is a *very simple* pure function of state
  - view effects change by dispatching events with minimal information:

    (dispatch :update :username “sam”)

    (dispatch :submit)
  - behavioural tests don't need the view

Sélectionnez la bonne réponse.

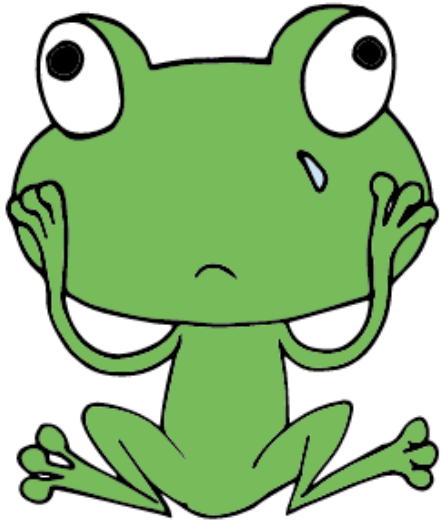
Bonjour ! Comment ça va ?

Ça va bien, merci !

Je m'appelle Bob

J'adore le chocolat !

Je déteste la végémite !



Réessayer

Voir la réponse

Model

```
{:state    :flashing-cards
 :username “sam”
 :nickname “Sam”
 :card {:question      “Bonjour, comment ça va?”
        :right-answer  “Ça va bien, merci!”
        :wrong-answers [“Je m'appelle Bob”
                       “Je suis australien”
                       ...]}
 :incorrect-attempts   [“Je m'appelle Bob”]
 :remaining-cards      [{...}
                        {...}]}
```

Controller

```
(defmachine state-machine
  {:start       :question
   :transitions   {:question    {::right-answer   :correct
                                 ::wrong-answer   :incorrect
                                 ::submit-answer  [#{:correct :incorrect}
                                                   submit-answer-next-state]
                                 ::skip-card      :skipping
                                 ::finish-session :done}
                   :correct     {::next-card      [#{:question :done}
                                                   next-card-next-state]}
                   :incorrect   {::show-answer    :show-answers
                                 ::try-again      :question
                                 ::skip-card      :skipping
                                 ::finish-session :done}
                   :skipping    {::next-card      [#{:question :done}
                                                   next-card-next-state]}
                   :show-answers {::right-answer   :correct
                                 ::submit-answer  [#{:correct :incorrect}
                                                   submit-answer-next-state]
                                 ::skip-card      :skipping
                                 ::finish-session :done}
                   :done        {}}
   :transition-fn transition})
```

Controller (cont)

```
(defmulti transition (fn [db event] (:id event)))

(defmethod transition ::right-answer
  [{:keys [state-db] :as db} _]
  (-> db
      (update-in [:state-db :stats
                  (-> (:card-and-exercise state-db)
                      :exercise
                      :category)
                  (if (:incorrect-attempts state-db)
                    :incorrect
                    :correct)]
                 (fnil inc 0))
      (update :state-db
              dissoc
              :student-answer
              :incorrect-attempts
              :has-shown-answer?)))
```

View

```
(defn [{:keys [dispatcher]} _ model]
  [:div
   [:span (:instructions model)]
   [:span (:question-text model)]
   [:button {:on-click (dispatcher :right-answer)
     (:right-answer-text model)]
   [:button {:on-click (dispatcher :wrong-answer)
     (first (:wrong-answer-text model))]
   [:button {:on-click (dispatcher :wrong-answer)
     (second (:wrong-answer-text model))]])
```

How do we win?

- test behaviour

```
(let [system (...)]
  (input! system :update :username “sam”)
  (input! system :update :password “pass”)
  (input! system :submit)
  (is (= :logged-in
         (current-state system []))))
```

- “test” views
  - call render code, but without viewing result
  - Devcards

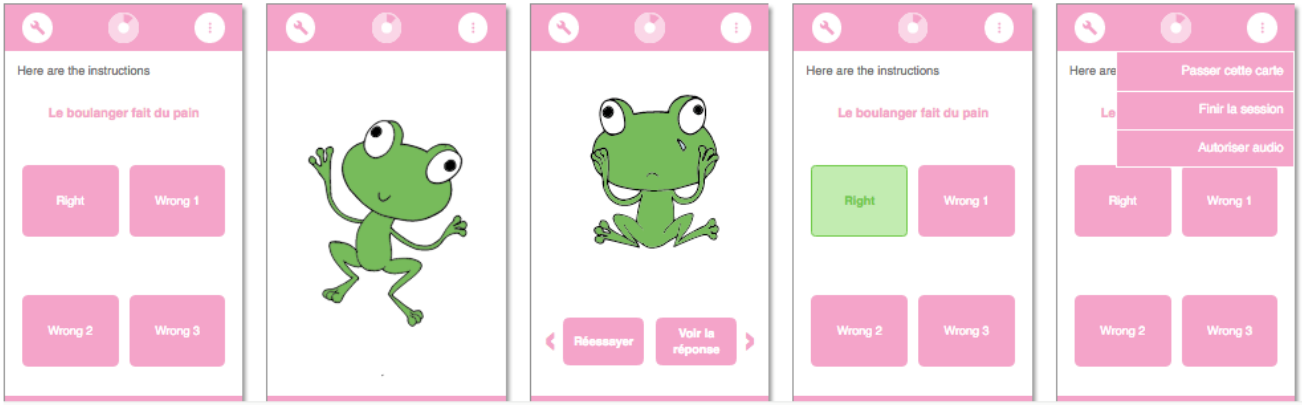
Devcards

localhost:3449/devcards#!/nanny.web_client.devcards.cards

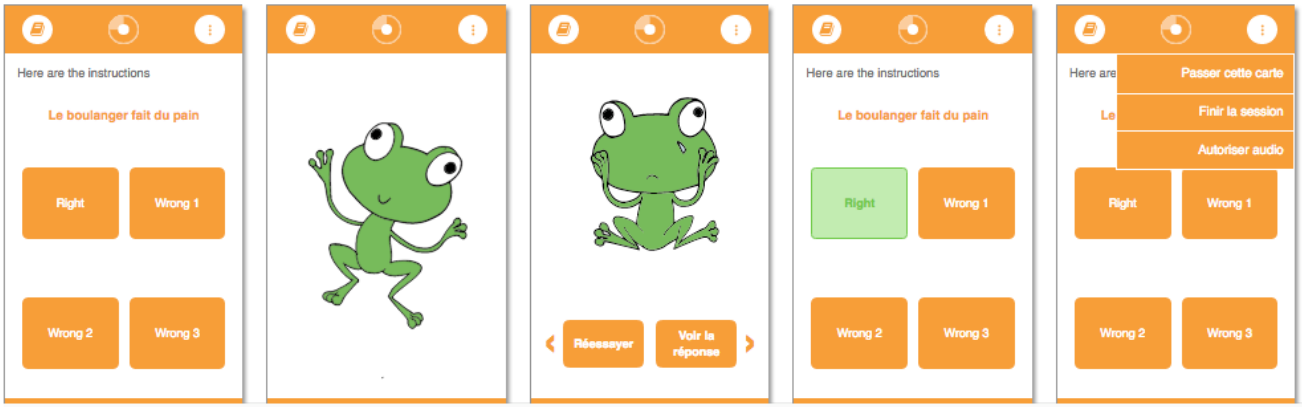
devcards / nanny.web_client.devcards.cards

cljs

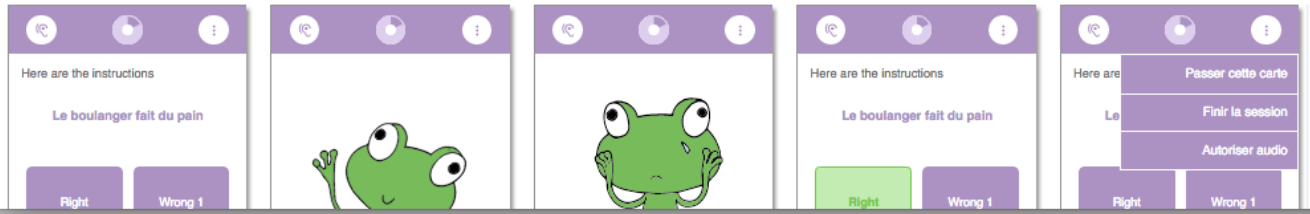
mc-fill-the-blank-card-grammar



mc-fill-the-blank-card-vocabulary



mc-fill-the-blank-card-phonetics



What are we testing?

- the whole app

```
(def db (atom {…}))
(def state-machine …)
(def view-tree …)

(add-watch db (view-renderer view-tree))

(defn init! []
  (swap! db #(start state-machine % nil)))
```

- only ‘**db**’ changes, in response to discrete inputs
  - it is a simple succession of values over time

Yay, FP is so easy! …

- … but real apps aren't pure functions of an initial DB + user inputs
  - real apps have AJAX calls
  - or local databases
  - or audio to play, or js/setTimeout, or …
- we want to test our app's interactions with the real world, too
  - we need a way to model and reason about effects

Testing the real world?

- keep our state machines pure
  - state machine transitions are side-effect free
  - but they can produce an “output” data structure

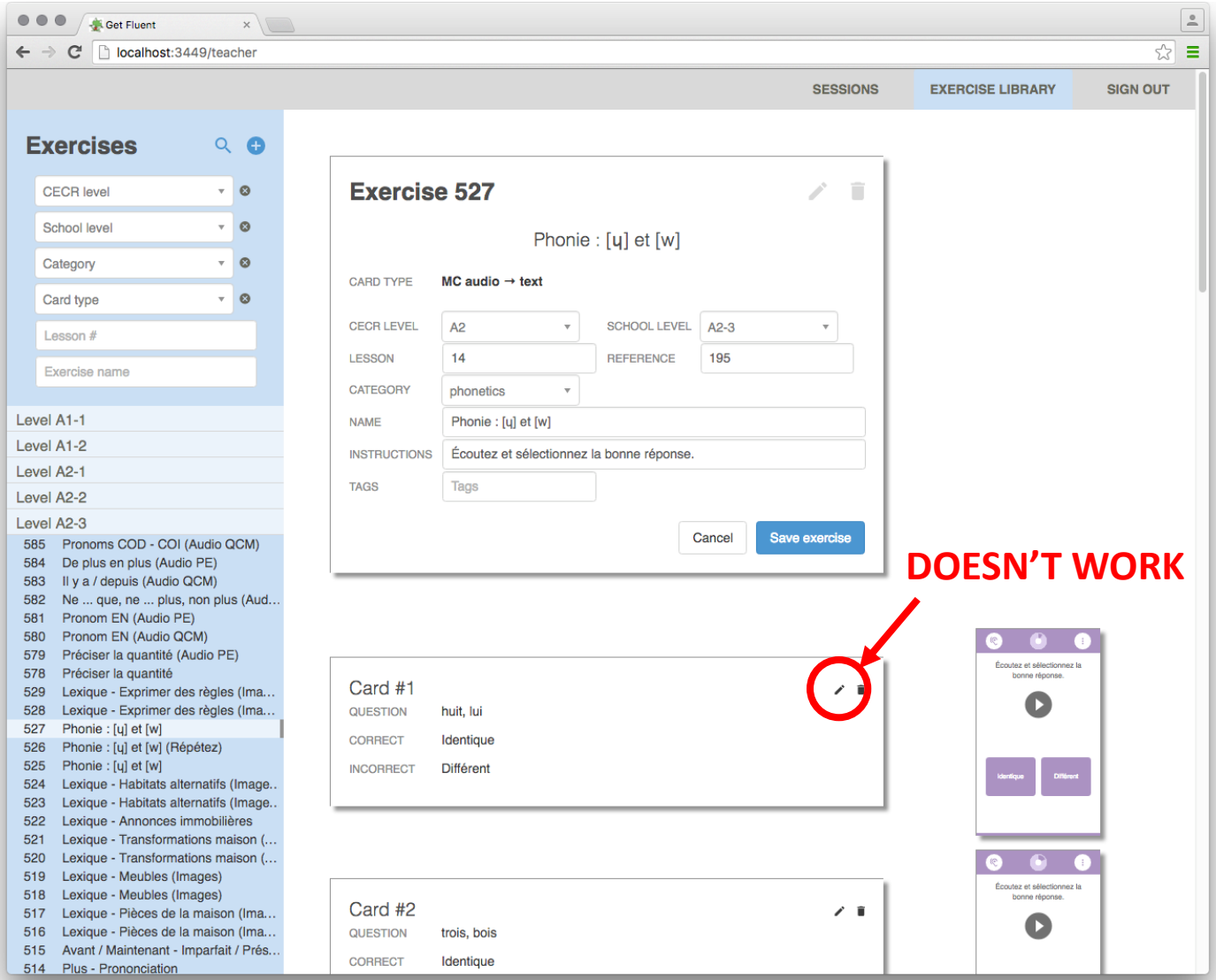
```
:outputs [{:id :submit-login
           :payload {:username “sam”
                     :password “password}}]
```

- in the app, this is an AJAX call
- in tests, we have options
  - maybe we assert its contents
  - maybe we give it to the server, but in-process

Why bother?

- why go to all this effort to make state machines so pure and theoretical?
  - easier to reason about
  - ***put the stuff we might get wrong (without noticing) where we can test it most easily***

Why? It's all good already, no?



Test client + server in-process

```
(with-rolled-back-db-tx [tx db-spec]
  (let [server (create-server tx)
        client (create-client
                {:output-handler
                 (mock-ajax server)})]
    (input! client :update
            :username “sam”)
    ...))
```

What do we get?

- ability to test as much or as little as we want
  - comprehensive system-level tests invoking a real server system with a live database
  - or unit-level tests of a single component with test-supplied (mocked) responses from server
- fast tests
  - no more extended light-saber fights while Selenium pokes at a browser
- succinct tests

What do we get? (bonus)

- a model of our system that matches the way we think and talk about it
  - "user enters username"

    (input! system :update :username "sam")
  - "user clicks submit"

    (input! system :submit)
  - "user is logged in"

    (is (= :logged-in
    (current-state system [])))

- loose coupling of tests to implementation

What else? (more bonus?)

- dispatcher tracking: know what inputs your view is capable of producing
  - report test coverage
  - random input generation / fuzzing
  - property-based testing?

- re-use “behaviour” for different views
  - React for the web
  - React Native for a native phone app

Merci !

Questions?

Sam Roberton – @sroberton
github.com/samroberton/bureaucracy