Fm Synth

```clojure
(defn fm-synth [{:keys [out] :as props} & children]
  "FM synth"
  ;; The new synth is instanciated using js interop
  (let [synth (js/Tone.FMSynth. (clj->js (dissoc props :out)))]
    (reagent/create-class
     {:component-did-mount
      (fn []
        ;; The synth is connected to it's output (which is passed
        ;; in as a property to the component)
        (.. synth (connect out)))
      :reagent-render
      (fn [props & children]
        ;; The render function renders a dummy span dom element and
        ;; renders it's children and passing it's synth as the out
        ;; for these components.
        (into [:span]
              (map (fn [child]
                     (assoc-in child [1 :out] synth))
                   children)))
      :component-will-unmount
      (fn []
        ;; Here we dispose of the synth
        ;; This will happen when the parent note is removed or when
        ;; a live code reload happens
        (.. synth dispose))})))
```

Audio component tree

```clojure
;; Parent component is the project and
;; has settings such as tempo
[project {:project :settings}
 [master-bus {}
  ;; Master volume is set here
  [volume {:volume :settings}
   ;; Adds a simple reverb effect
   [reverb-effect {}
    [
     ;; First note
     ;; Each note has a vibrato effect
     [vibrato-effect {}
      ;; Envelope to control vibrato depth
      [timeline-evt evt
```

```
       [envelope {:param "depth"
                  :env   vib-env}]]
      ;; The fm synth that generates the signal
      [fm-synth (get-in evt [:preset])
       ;; Timeline event that takes care of queuing
       ;; it's child components
       [timeline-evt evt
        ;; In this case a note
        [note {:note :settings}]
        ;; And a frequency envelope
        [envelope {:param "frequency"
                   :state state
                   :env   freq-env}]]]]
     ;; Second note
     [vibrato-effect {}
      ;; ...
      ]
     ;; ...
     ]]]]]
```

Graphics notes

```
(defn graphics-notes []
  (let [notes         (subscribe [:notes])
        width         (subscribe [:graphics-stage-width])
        height        (subscribe [:graphics-stage-height])
        ....]
    ;; playback-time @(subscribe [:playback-time])
    (into [:Group]
          (map (fn [note]
                 ^{:key (:id note)} [graphics-note* ...]) @notes))))
```

Editing notes

```
(let [enabled (-> (Math.random)
                  (<= ,, probability))]
  ;; Only trigger the note if enabled is true.
  ;; If probability is 1.0 will always be true.
  (if enabled
    (do
      (rf/dispatch [:note-enable id])
      (some-> out
              (.triggerAttackRelease ,,, freq (prep-time dur) t velocity)))
    (rf/dispatch [:note-disable id])))
```

Undoable middleware

```
;; Code here
```

Note view animation

```
(rf/reg-sub
 :note
 :<- [:notes-raw]
 :<- [:playback-beat]
 (fn [[notes playback-time] [_ id]]
   (let [{:keys [onset duration] :as note} (get notes id)
         end-time                          (+ onset duration)]
     (let [updated-note (if (and
                              (> playback-time onset)
                              (< playback-time end-time))
                          (-> note
                              (assoc ,,, :playing true)
                              (assoc ,,, :playback-time (- playback-time onset)))
                          (if (true? (-> note :playing))
                            (assoc note :playing false)
                            note))]
       updated-note))))
```