UNC COMP 590-145
Single Effect Architecture pt. 2

Wednesday, April 15, 2020
Review from part 1

- state is a reference to a value that can change in time (value + time)
- variation in application state drives changes in the UI
- consolidating state into a single place enables the single-effect architecture
- screen designs, state value examples, and list of parameterized effects is a great first step in designing an app
- reducers take a state value and (parameterized) effect and return a new state
Example from part 1

```lisp
{:x-next? true
 :squares [[:x nil :o ]
            [nil :o nil]
            [nil nil :x ]]

  [:make-move <row> <column>]
  [:reset-game]
```

New game
Reducer from part 1

(defn make-move [cur-state effect]
  (let [[_ row col] effect
        cur-player (if (:x-next? cur-state) :x :o)]
    (-> cur-state
        (update :x-next? not)
        (assoc-in [:squares row col] cur-player))))
Plan for today

- Seeing tic-tac-toe app in Redux.js
- Benefits of the single-effect architecture (incl. time travel)
- Redux.js overview (a SEA library for Javascript)
- The view layer and what React.js provides
- Asynchrony and external interactions with Javascript promises
- Code commenting best practices (if time)
Tic-tac-toe in Redux.js

- code is here
- to run it, you'll need Node.js installed
- after cloning, run `npm install` then `npm run watch` to start the compilation process
- run `npm install -g http-server` then `http-server dist -p 3002` then open `http://localhost:3002` in your browser
- find React dev tool and Redux dev tool extensions for your browser; close & reopen the tab
Benefits of the single-effect architecture

- IMO most difficulties debugging have to do with implicit state, and there is no implicit state in the SEA.
- With a little bit of tooling, you can time-travel.
- In fact, you can even directly manipulate history to explore what might have happened.
- Imagine doing tech support when you can access the user's state and effect history.
- (This is not difficult to attach to your exception logging infrastructure if you use the SEA.)
Redux.js overview

- Redux.js (a Facebook library) uses the single-effect architecture; in fact, it popularized it.
- It stores the state in a "store", and *dispatch* (the polymorphic single effect) is a method of the store.
- We give the store our initial state value and our top reducer.
The view layer

- We're using React.js (another Facebook library) for the view layer.
- Essentially, React ensures that the HTML tags in our page match the current state.
- React observes the state, and if it changes, React will efficiently re-render what needs to be re-rendered.
- We're responsible for telling React how to display a given state value in terms of HTML tags.
- The react-redux library connects React to Redux and enables our React components to call `dispatch`.

UNC COMP 590-145
Single Effect Architecture pt. 2
Asynchronous actions - motivation

- Most apps will want to send and receive data over the network to the backend
- We don't want to hang the app while we wait on the network request; rather, it should happen "in the background" while the user still uses the app
- So we need *asynchronous actions* to do this background work in parallel to the foreground work
- In JS, a popular mechanism for asynchrony is Promise objects; I won't dive in too much to these
- But I do want to show how to do asynchrony in the SEA
Asynchronous interactions in the SEA

- We use a kind of middleware to power-up our Redux store: redux-thunk
- The idea is that, instead of dispatching a polymorphic effect as a simple data structure, we can also dispatch a "thunk", i.e. function.
- When the redux-thunk middleware sees a thunk, it calls it and passes the `dispatch` function (and a `getState` function)
- The function can call dispatch whenever and however it wants
Thunk example

```javascript
function fetchUser(dispatch) {
  dispatch(fetchUserStart());
  const url = 'https://reqres.in/api/users/2'
  return fetch(url)
    .then(response => response.text())
    .then(json => dispatch(fetchUserSuccess(json)))
    .catch(error => dispatch(fetchUserFailure(error)))
}
```

- Calling dispatch once initially (always), once if success, and once if error
- The lower-level `fetchUser*` actions add state about the progress of the network request, so that loading/success/error indicators and data can be displayed.
Code commenting best practices

- Aim for legible code rather than explanatory comments
  - You don't want to have to maintain a comment in parallel to changes you make to the code (a.k.a. "comments lie")
  - Clear names of variables and functions really help
    - Add names if necessary to make things clear, e.g. with more names in your let block
- Use git commit messages for comments about your motivations, your intent, your design approach, and any alternatives explored
- `git blame` connects lines of code to commit messages
- Use a comment if it's important for devs to see it in connection with some code, e.g. you might think `<another approach> should work here, but it doesn't because <reason>`. 