Unit 7 – State Management – Redux

1. It is the first tech we’ll find that is **opinionated**
   1. It’s super particular. It’s redux’s way or the highway.
   2. The boilerplate is a major challenge – getting set up, getting the system running, etc, is difficult. Once it’s set up, it is incredibly beneficial.
   3. Mike hated it, now he can’t imagine building an app without a state management tech involved.
2. Managing state.
   1. What does all of this do??
3. In our tic-tac-toe board, the Board was our “stateful” component.
   1. The row didn’t do much other than help drill down our props into Box.
4. But there’s another option. We can put something into an even higher level area called the “Redux store” and call data as we need to from any component.
   1. This is uhhhhh going to be tough to get set up, but once we do, it can be amazing! No more prop drilling.
5. Benefits of state management library (like Redux)
   1. Simplified access to state across complex apps
   2. Better debugging – including time travel
   3. Simplified Testing
   4. And acts as a ‘single source of truth,’ like Fox News and Rupert Murdoch’s Butthole.
6. When states are stored all across the application, it is hard to keep track of what’s happening in the app.
7. Redux is based on the **flux** architecture, used by Facebook.
   1. Flux is a modified version of the observer pattern, which is one of the 23 design patterns outlined within a book called “Design patterns: elements of reusable object oriented software.”
   2. **Observer patterns…**?
8. Observer design pattern
   1. All observers are viewing a single subject
   2. And a single subject, for example might notify observers about new tweets.
   3. The **subject** is an object which is subscribed to by observers.
      1. When the state of the subject changes, we update the data seen by observers.
9. Flux Data Flow
   1. Keywords to know:
      1. **Actions -** Objects
      2. **Action Creator –** Generator Functions
      3. **Dispatcher** – Other functions
      4. **Stores –** Objects (in flux, there are multiple stores)… basically, our states.
      5. **Views –** what the observer receives.
   2. An **action** occurs, which is created and passed by an **action creator** to the **global** **dispatcher**, which then passes it to every registered callback.
   3. If a store has a callback registered with the **dispatcher**, it will receive the dispatched action.
   4. Logic within the **store** runs functionality using the action, and then emits a change event.
   5. **Views** get the change event and updates itself.
10. Redux
    1. This is a stripped down version of flux, which still follows unidirectional data flow.
    2. **The skill we’re building with redux is reading the flow of data.**
    3. Data Flow
       1. The Store is one big object with all the states.
          1. A state is changed
          2. A re-render is triggered
       2. React Component
          1. Connects to the store via these Hooks
             1. useSelector()

use this on the react component to pull the information we need from the store to populate local props.

* + - * 1. useDispatch()

This connects us to the reducer.

* + - 1. And these are the things that connect to the store.
      2. And when a user interacts with a React component, it will invoke the Action Creator.
    1. Action Creator
       1. Creates an object that contains that action data
          1. This is the “Action” object
          2. Contains the properties

type: Some\_action

Payload: “some new value”

* + - 1. Dispatches an action to a Reducer
         1. The action creator does not change the value; it produces the object containing the value that the reducer needs to update the store with.
    1. Reducer
       1. IS THE ONLY THING THAT CAN UPDATE VALUES IN THE STORE
       2. Reducer is a **function**, maybe with a bunch of switch cases
          1. It’ll go through those cases and see what we need to do.
       3. The object that it returns should contain all the states that were previously stored in the state.
       4. Returns the new state object and passes this new state to the Store.
       5. The reducer
  1. React loves Redux, but Redux only looks at React as a friend.

1. React likes Redux because:
   1. Efficient state management
   2. Hot module reloading that doesn’t lose track of state
   3. Time travel debugging with redux dev tools
2. CSX uses react/redux; check it out for some deployment examples.
   1. Especially the redux dev tools
   2. As actions are taken, it will keep a time log of initialization and of actions as having happened a certain time after initialization.
      1. And each action that is dispatched
      2. Contains the type and payload.
      3. The type is something like an identifier
      4. And the payload contains
3. Golden Rules of Redux
   1. **Redux Store** contain all of our state
      1. An object representing the state of our application
      2. Store is **STATE PLUS REDUCERS!!!**
   2. **Actions** describes updates to the state
      1. **Simple obejcts** that contain data for our reducers
      2. **It must contain a “type” that indicates the type of update**
      3. **And the payload which is an object containing the state data.**
   3. **Action Creators** creates the action object dynamically
   4. **Dispatchers** send action to all reducers
      1. When the dipatch method is invoked, all reducers are fired.
   5. **Reducers** apply updates to the state.
      1. **Ususally there is one reducer**
         1. That contains a bunch of other functions.
         2. But there can be multiple reducers, depending on the architecture of our app.
      2. **Take two arguments**
      3. The current state object
      4. and an action (which is an object containing type and payload).
      5. The reducers often contains **switch statements** that switch on the type property passed in.
      6. Reducers should treat state as immutable. It should make a change on a copy of the initial state.

Data flow:

* A user clicks on a tic-tac-toe box. This will probably try to see what the current state is by accessing data from the Store.
* It invokes an action creator which will talk about the type of action: a click; and the payload: the new state (an x or o).
* This is passed to the reducer, which uses switch cases to determine what action was taken place.
* And it uses this to create a new state object to pass to the store.
* Which will run the React diffing algorithm and re-render the update state.