React+Redux

Code 401 (Class 31)

State Management with Redux

Redux gives us a higher level “Store” to store application state.

Gone is “setState” …

To update the store:

We “Dispatch” an “Action” to a “Reducer” that will update the “Store”

Dispatch will be a function that we call, sending it an action

Based on the action, our reducer will determine which operation should occur.

Actions

Actions describe the fact that ***something happened***, but don't specify how application state changes in response.

They are "payloads" of information that you send from your application to your store. They are the only source of information for the store. They are sent to the store using:

```
store.dispatch()
```

Actions carry a type (required) and a payload that typically carries data or instruction. I.e.

```
{
   type: ADD_TODO,
   text: `do this thing`
}
```

A common pattern is to use an action creator (a function that returns an action) which makes them portable and easier to test.

Reducers

A reducer is a pure function that takes the previous state and an action and returns the next state.

(previousState, action) => newState

Things you should never do in a reducer:

- Mutate its arguments
- Perform side effects (like API calls and routing transitions)
- Call non-pure functions like Math.random()

Given the same arguments, a reducer should calculate the next state and return it. No surprises. No side effects. No API calls. No mutations. Just a calculation.

Reducers

```
function todoApp(state = initialState, action) {
  switch (action.type) {
    case SET_VISIBILITY_FILTER:
      return Object.assign({}, state, {
        visibilityFilter: action.filter
      })
    default:
      return state
  }
}
```

Note:

- State was not mutated ({} was the first argument to Object.assign!)
- The default case is to simply return the previous state

The Store

Actions that represent the facts about “what happened”

Reducers update the state according to those actions.

The Store is the object that brings them together. The store has the following responsibilities:

- Holds application state;
- Allows access to state via **getState()**
- Allows state to be updated via **dispatch(action)**
- Registers listeners via **subscribe(listener)**
- Handles unregistering of listeners via the function returned by **subscribe(listener)**

It's important to note that you'll only have a single store in a Redux application. When you want to split your data handling logic, you'll use **reducer composition** instead of many stores. (Class 32)

Implementing Redux

Directory Structure

```
/actions
   category-action.js
/reducers
   category-reducer.js
/lib
   Store.js

__or__

/app
   store.js
   reducer.js
   actions.js
```

Implementing Redux

Create reducer (reducer.js)

Create actions (actions.js)

Create a store

- import the reducer
- return createStore(reducer)

Root Component (typically main.js)

- Import Provider from react-redux
- import the store
- Create an instance of createStore
- Wrap root with a <Provider>

Container Component:

- Import {connect}
- Import {action1,action2} from your actions file
- mapStateToProps
- mapDispatchToProps
- connect(mapStateToProps,mapDispatchToProps)(Component)

Data path

Main Component
- Creates a store and shares it

Container Component
- Connects to the store
- Maps state and dispatch to props
- These are then available as this.props.xxx in the container component
- Might send state and dispatch methods to child components as props

Child Components
- Can either receive the mapped store items as props
- … or can connect() to the store themselves.

When something happens (like a form submit), rather than handle the state in the components, simply call the dispatched method (from this.props)

That will get the action from the actions file and dispatch it to the reducer

The reducer will then “run” the right action and operate on state