When and why is redux the right choice?

1. Most commonly used with React
2. Useful for applications that maintain a large amount of info in state and is a highly reactive app with a high volume of data flow
3. Basically – whenever you need to streamline state management.
4. Flux and Redux both have unidirectional data flow.
   1. Actions passed to Dispatcher
   2. Dispatcher passes to store
   3. Store sends data to view.
   4. Action is triggered…
5. To change the state of an app, we invoke an Action Creator
   1. Which creates an action Object.
   2. This is fed to a dispatch function via a user action
      1. This forwards the data to a reducer
         1. Which takes the action, performs the functionality, and then returns the state.
6. When we pass down states, we **only** pass down the states that are necessary.
   1. useSelector is what subscribes a component to the store.
   2. If we put useSelector all over the place, all of those components will re-render each time the store is updated.

**The roadmap -**

1. Store.js
   1. We connect redux dev tools and createStore.
2. Index.js
   1. Here, we will be combining our reducers
   2. We first import “combineReducers” from redux
   3. And use this to combine all the reducers into one.
   4. And we initialize Redux by wrapping the App component in the Provider component, giving the store as a property of the Provider.
      1. This will allow us to pass down the store information to other components.
3. MarketsReducer.js
   1. This always accepts an object as state and return an object as new state.
4. Action.js
   1. These are bare-bones actionCreators.
   2. We modularize and export our actionCreator functions.
   3. They should only create an action object. It **must** have a type.
      1. It might not have a payload, depending on the action.
5. MainContainer.js
   1. Sky does most of the business logic in the containers.
   2. The other components are mostly “presentation components.”
   3. In the main container, Sky adds totalCards and totalMarkets data.
      1. This is passed to totalsDisplay.
      2. TotalsDisplay is rendered as a component.
      3. TotalsDisplay.jsx, therefore, only has the labels, divs, and spans with the appropriate props that were drilled down.
      4. But no business logic is implemented.
   4. Something to keep in mind:
      1. The purpose of this exercise – in part – was to emphasize the distinction between presentational components and… the other non-presentational and functional components.
      2. Containers are good places to have functionality.
6. MarketsContainer.jsx
   1. We subscribe here to marketList, newLocation, and totalCards.
   2. Then we implement a dispatch and assign different action creators for each of addMarket, updateLocation, addCard, and deleteCard.
   3. All it renders is the MarketCreator component and the MarketsDisplay component. We pass into these the addCard and deleteCard actions as well to pass into those children.
7. MarketCreator.jsx
   1. Here, we have our presentation component
   2. And here, in the input field, onChange, we update the “Set\_New\_Location” action.
8. In MarketsDisplay:
   1. Const MarketsDisplay = ({marketList, totalCards, deleteCard, addCard}) => {…}
   2. This essentially extracts these properties as individual vars that can then be called.
   3. This kind of destructuring is quite common.
      1. It is called “implicit deconstruction” because… maybe because we are not explicitly assigning them to individual variables? Check MDN.
9. In Markets.js
   1. Another destructuring syntax:
   2. Const Market = props => {
   3. }