Composing reducers

Eventually our web application will have a component that will show the user all the seats in a theater and allow them to reserve one or more. They should be able to tap on a seat to reserve it and then tap it again to un-reserve it should they change their mind. It sounds like we will need an action to ADD_SEAT_TO_CART and another to REMOVE_SEAT_FROM_CART. Let's see what that entails.

1. Edit actions.js. Add actionTypes for each of these. Then add action creators to the actions enumeration. Both action creators will receive *seat* as an argument.

Here comes the tricky part. We want state to have a cart. Cart should have an array of seats and an array of food for when the user pre-orders their meals. When the user taps on a seat to reserve it, we need to add a seat to state.cart.seats. Here is how that reducer case would look:

```
case actionTypes.ADD_SEAT_TO_CART:
  return {...state, cart: {...state.cart, seats: [...state.cart.seats, action.seat]}}
```

Wow, that's a mouthful! Note that because the levels of nesting, this reducer case is overly complex. And REMOVE_SEAT_FROM_CART is even more complex! Let's see if we can make it less complex. We'll start by extracting a sub-reducer for the cart.

Adding a sub-reducer

2. To prepare, change the name of your existing reducer to rootReducer and delete the *export* in front of it.
3. At the bottom of the file, add this:

```
export const reducer = (state, action={}) => (
  {
    ...rootReducer(state, action),
    cart: cartReducer(state.cart, action),
  }
);
```

This exports reducer as before but this time, it spreads the results of rootReducer and overrides its cart property with the results of the cartReducer(). So obviously we need to create a cartReducer.

4. Add a new method to reducer.js. Call it cartReducer and give it the typical shape of a reducer function. You know, the *(state, action)* => *state* shape.

Focus on state for a moment, notice how cartReducer is being called. We're passing state.cart into it. Thus we're not giving this sub-reducer the entire state. Instead, we're giving it only the portion of state that has to do with the cart. Therefore, inside cartReducer, the "state" is really a portion (or slice) of state and is much, much simpler!

Here is the case for ADD_SEAT_TO_CART in cartReducer:

```
case actionTypes.ADD_SEAT_TO_CART:
  return { ...state, seats: [...state.seats, action.seat] }
```

See? Much simpler. Easier to understand, easier to get right, easier to debug.

5. Add that case to cartReducer and remove it from the rootReducer if you had in there.
6. Run and test by adding this to App.js's useEffect:

```
store.dispatch(actions.addSeatToCart({id:1,seat_number:1,price:1.00 }));
```

7. Run and test. Look in the console for your state. If you've implemented all of this carefully you should see a seat in the store's cart property.

Removing a seat from the cart

Now that you have the idea, you're going to create a reducer action that will remove a seat from the cart. But this time we're supplying you with fewer hints.

8. Add this to App.js's useEffect immediately after you've added the seat:

```
const theSeat = store.getState().cart.seats[0];
store.dispatch(actions.removeSeatFromCart(theSeat));
```

9. If you run and test, it won't work; the seat will still be in the cart.
10. Your mission is to add a reducer case to remove that seat from the cart. Use any JavaScript you'd like to remove it. (Hint: slice might work, brute-force looping and adding to another array could work, but filter would be my favorite).
11. Once you've got it working, you can remove the three testing lines from useEffect and be finished with the lab.

12. Bonus!! It may be better to have each reducer in its own file. Extract rootReducer and cartReducer into their own files and import them into reducer.js. This'll just clean things up a bit.