Ajax Lab

In this lab we're going to make a major transition from simulated data to real data fetched from RESTful endpoints, processed through Redux, and displayed in the React Native components we've been building up. Don't worry, we've written most of the Redux code for you but if you can handle writing it yourself without relying on our code suggestions, please try to do it on your own.

A whole lot of prep we've been doing in prior labs will come to fruition in this lab. Are you excited? Let's get started!

Showtime!

At a couple of places we are going to want to show the user a list of showing times that they can select. Fortunately we've extracted that functionality into the ShowingTimes component so we only have to change it in one place. We'll start by fetching the showings.

Remember that as of right now, you have a component called DatePicker.js. When the user chooses a date, we're dispatching a "SET_SELECTED_DATE" action. What would be cool is if we had something that would also fetch showing times when SET_SELECTED_DATE is dispatched. But you can't do that in the reducer since reducers must be pure functions. Sounds like a job for middleware!

Before you start this lab, run the `npm run reload-daam` command from the /server folder. It's been many days since you last seeded the database, and by now we might not have fresh data for upcoming showings and reservations!

1. Create a middleware to handle the fetch. It should send a fetch to the API and in the callback, dispatch the showings it has retrieved. Something like this might work for you:

```
const fetchShowingsForDateMiddleware = ({ dispatch, getState }) => next =>
async (action) => {
  // in this case, we want the reducer to respond to and update state *before*
running our network calls
  next(action)
  if (action.type === 'SET_SELECTED_DATE' || action.type ===
'SET_SELECTED_FILM') {
    try {
      // because of the next(action) invocation happening already, we can
trust these values will be updated to whatever
      // was set by the action.
      const selectedDate = getState().selectedDate.toISOString().split('T')[0]
      const filmId = getState().selectedFilm.id
      const { data: showings } = await
axios.get(`${host}/api/showings/${filmId}/${selectedDate}`)
      dispatch({ type: 'SET_SHOWINGS', showings })
    } catch (e) {
      console.error('Could not fetch showings', e)
    }
  }
}
```

2. Don't forget to register your middleware with the store.

3. Then, create a reducer handler for "SET_SHOWINGS". It should do something like ...

```
case "SET_SHOWINGS":
  return { ...state, showings: action.showings };
```

4. Run and test. Once you have a selected film and a selected date, your showings times should change on both the Landing scene and in Film Details.

5. Bonus! You should no longer have a need for showings.json. Feel free to delete that file and change your code to no longer read from it.
6. You'll find that the ShowingTimes component may look garbled now that you're displaying more available times. Fiddle with flexbox until you can get it right.

Fetching the tables and chairs

If you tap on a showing, you're navigating to the PickSeats scene. Currently this scene is using a hardcoded list of tables and seats. Let's grab real tables and seats from the API.

7. It should be receiving a *showing* object as one of its props.navigation.state params. Make sure that's the case. If not, find where you're navigating to it, and pass the showing object in the navigation parameters.
8. When the component mounts for the first time only, dispatch an action:

```
dispatch({type:"FETCH_TABLES_AND_SEATS", theaterId });
```

Clearly this middleware doesn't exist and the reducer action to set the tables and seats doesn't exist so we should create them.

9. Create a new middleware that will fetch the tables and seats in a theater. The middleware might look like this:

```
const fetchTablesAndSeatsMiddleware = ({ dispatch }) => next => async (action)
=> {
  next(action)

  const { type, theaterId } = action
  if (type === 'FETCH_TABLES_AND_SEATS') {
    try {
      const { data: tables } = await axios.get(
        `${host}/api/theaters/${theaterId}/tables`
      )
      dispatch({ type: 'SET_TABLES', tables })
    } catch (e) {
      console.error('Could not fetch tables', e)
    }
  }
}
```

10. The reducer case might look like this:

```
case "SET_TABLES":
  return { ...state, tables: action.tables };
```

11. Run and test. If we've put everything together right, you should now be able to tap on any showing time and see the actual tables and actual seats in the theater. Remember that you'll need to stop reading from the hardcoded list of tables and instead pull that data from Redux.

12. Bonus! Delete the tables.json file. We don't need it now that we're reading real data.

Showing taken seats

Some of these seats have already been reserved. We should read those reservations from our API.

13. Create another middleware function for fetching reservations:

```
const fetchReservationsMiddleware = ({ dispatch }) => next => async (action)
=> {
  const { showingId, type } = action
  if (type === 'FETCH_RESERVATIONS') {
    try {
      const { data: reservations } = await
axios.get(`${host}/api/showings/${showingId}/reservations`)
      dispatch({ type: 'SET_RESERVATIONS', reservations })
    } catch (e) {
      console.error('Could not fetch reservations, e)
    }
  }

  return next(action)
}
```

14. And the reducer case:

```
case "SET_RESERVATIONS":
 return { ...state, reservations: action.reservations };
```

15. Edit PickSeats.js. In componentDidMount, add another dispatch:

```
dispatch({ type: "FETCH_RESERVATIONS", showingId });
```

16. Run and test. You should be able to see all the reservations for this showing.

Now all that's left is to set the "status" property on each seat. Here's what we want... If there is a reservation for this seat in this showing, the background color should be different. You should have already prepared for this in the styling lab

17. You now have an array of reservations for individual seats, and you have seats that belong to a table. Find a way to determine whether a particular seat has a reservation; if it does, give it a special styling. You can accomplish this task in a wide variety of ways, try your hand at one (or several!).
18. Run and test. You should be seeing all of the reserved seats as a different color.
19. You should also make the reserved seats not respond to button presses.

Allow user to actually pick a seat

20. Originally in `<Checkout />`, you used mock data from `carts.json`. Update code in PickSeats.js to allow the user to choose a seat and pass that seat information along to the Checkout component.
21. For now, we're not going to worry about picking multiple seats. You can tackle that later if you're feeling adventurous.
22. You should then use that seat information to display the subtotal.

Allow user to make reservation

23. Little did they know that if they made their reservation and paid for it, then went back to look at that movie, they'd find their seats unclaimed! Oh dear.

24. It turns out we never POSTed the information to the server, telling it that a new reservation has been made. If you were creating a single reservation, you'd POST to /api/reservations with a payload looking something like this:

```
{
    "showing_id": 1,
    "seat_id": 42,
    "user_id": 100,
    "payment_key": "pk_123456"
}
```

Some of those values don't really matter, e.g. we won't be using user_id or payment_key in the app, so feel free to put in bogus values there.

25. Unfortunately, there's not an easy way with our simple server to POST a *collection* of reservations, so you'll need to make individual POST requests for each reservation the user is making. You can use logic like Promise.all to wait for them all to finish in parallel, or just fire them off sequentially.
26. Once you've implemented that logic, test to make sure it works! You should see seats be reserved the next time you try to book seats for that movie at that time.

Deal with slow networks

27. Right now we're doing our development locally, so all the network calls are blazing fast. In the real world, you have to deal with latency. A well-known research paper indicates your app has 100ms to show response to a user's interaction before the app starts to feel "sluggish" or unresponsive.
28. Let's make it more dramatic for our testing purposes by creating a server that slows all requests by 3 seconds. Instead of running `npm run server`, do `npm run server-slow`. Now try going through you app, and notice all the dull moments where the app doesn't seem to care about your input, because it's silently making a network request.
29. Use the ActivityIndicator from React Native or one of the many other loading/spinner libraries out there to show when contents are loading.
    a. How do you want to manage your loading state logic? Local within the React component, and move your network calls out of middleware? Or you could dispatch updates to redux state from the middleware. Or you could hop on to a system like redux-saga or redux-thunk to help manage the transitions. Each approach has its advantages and disadvantages, and there's no "right answer".
30. Also notice the times (e.g. w/ PickSeats) where you pick a *different movie* but the old seats are still showing! Oops… Think about a way to fix that issue, and do so! Dealing with stale data is a common challenge with React Native.

Deal with network failures

31. Now it's time to deal with intermittent network failures! Run the server with the command

```
npm run server-flaky
```

32. Notice all the fun ways in which your app doesn't gracefully handle errors. Trust me, you'll get errors a lot more than you'd want. It turns out networks can be astonishingly flaky for mobile devices, and you need to build a robust app.
33. Some key principles to remember:
    a. The user doesn't know how to easily "reload" the app. (Maybe they know how to close and reopen an app, maybe not.) There's no "Reload" button like the web.

b. If an action fails, you should make it easy to try again, without the user having to do a bunch of extra work (e.g. if a form submission fails, they shouldn't have to re-enter all their information).
c. The error should appear close to where the action occurred, or otherwise require their attention. It's amazing how good people are at not noticing a little red error box that pops up in a corner of a mobile device.
d. You should tell the user *what* failed (e.g. "Could not load movies") and explain what they can do to fix the problem (e.g. "Retry")

Try extracting a Loader component

34. You may be noticing how there's potentially repeatable logic about loading data from the server. You need to handle a call, manage retry logic in case of errors, otherwise display the data. Create a <Loader> component that encapsulates this logic, and try using it on the Landing screen.

Try extracting a useLoader custom hook

35. Another common pattern is to pull business logic out into React custom hooks. Create a useLoader hook that takes a URL and manages some pieces of state. Share that state with consumers of the hook so they can act accordingly. Make use of this custom hook on the FilmDetail page to show the available showings.
36. Compare and contrast the two approaches. What does each do well? What does each lack?