Single-value Components Lab

We talked about a few React Native components that hold a single value. Let's work with a few of them. But before we do, we have to create some components to host them in. We're going to create five or six in fact.

For now, please don't worry about making any of these things "work" yet and don't worry about laying out the views nicely or styling them. We'll do that in future chapters. Just get them created and briefly checked out on a device.

The Landing component

Our main view will hold a header and a list of films we're currently showing.

1. Create a new component called Landing.js.
2. At the top,

```
import React from 'react';
import { Text, View } from 'react-native';
```

3. In the JSX, return a <View></View>.
4. Inside the <View>, add a <Text> with the name of our business, Dinner And A Movie.
5. Add a <Text> to tell the user to tap on a film to see its details and pick a date to see showtimes.
6. Edit App.js. Leave the <View> tag but remove the tag(s) that expo init added for us in there. Replace them with one <Landing> tag.
7. Run your app. Do you see your component? Great! Let's do some more.

Setting up the store

Since we'll be displaying data and maintaining that data, our components could get messy. We'll clean it up by using Redux.

8. First, install Redux in the root of your application.

```
npm install redux react-redux
```

9. Next, create a folder called "store". This is where you'll put all of your data-related JavaScript files.
10. Create a reducers.js file. It can start with this:

```
export const reducer = (state, action) => state;
```

11. Create a middleware.js file. Here's a simple start:

```
const fetchFilmsMiddleware = ({dispatch, getState}) => next => action =>
  next(action);
export default [ fetchFilmsMiddleware ];
```

12. Create your redux store in a file called store.js. It can begin like so:

```
import { createStore, applyMiddleware } from 'redux';
import { reducer } from './reducers.js';
import middlewares from './middleware.js';
const initialState = {};
export const store = createStore(reducer, initialState,
                                 applyMiddleware(...middlewares));
```

Connecting through react-redux

We want to use react-redux to hook our React components into the redux state. For that to happen, we need to wrap our React <App> in another component, the <Provider> which we'll put in a level above App.js.

13. To start, open package.json and change the "main" entry to point to a new file: "index.js"
14. Create a new file index.js at your root with the following contents:

```
import React from 'react'
import { registerRootComponent } from 'expo'
import { App } from './App'
import { Provider } from 'react-redux'
import { store } from './store/store'
registerRootComponent(() => (
  <Provider store={store}>
    <App />
  </Provider>
))
```

   Note that we changed the expo's default export to a named export pattern. Using named exports makes renaming much easier. We'll use it for this project, but ultimately it's a matter of style and the choice is up to you.
15. Edit App.js. At the top, import the useEffect hook from React, the useDispatch and useSelector hooks from react-redux and your store from store/store.js.
16. Use the useSelector hook (https://react-redux.js.org/api/hooks#useselector) to access the redux store. Also wire up a dispatch function.

```
const state = useSelector(state => state)
const dispatch = useDispatch()
```

17. Open store.js. Give initial state these properties:

```
const initialState = {
  films: [],
  selected_date: new Date(),
  selected_film: {},
  show_film_details: false,
  showings: [],
  tables: [],
}
```

18. Run and test. Make sure that state is being read properly. (Hint: You could console.log({state}).)

Reading film data

Our film data is in the database. To get it, we need to make an HTTP request from our RESTful API on port 3007 of the database server machine just like we did in the first lab. And to make an HTTP request with Redux we'll need to use middleware whether it is in the form of a libraries like redux-thunk or redux-saga, or it is written by hand. We're going to write by hand but if you want to use one of the other methods, feel free.

But there's a problem; different OSs and different emulators have requirements on how the machine is accessed. We've created a JavaScript library to help you with that.

19. Find api_host_maker.js in the starters/helpers folder. Copy api_host_maker.js to your project's store folder alongside the other Redux-related things. Edit this small file and see what it does. Make adjustments to the URL based on how you plan to view your app. (iOS vs Android, tethered vs. emulated, and so forth).
20. Edit your middleware.js file and import host at the top:

```
import { host } from './api_host_maker';
```

21. Note that we've already created a middleware function called fetchFilmsMiddleware. Let's change it to check if the action.type is "FETCH_FILMS" and if so, make an Ajax call to `${host}/api/films`. In the Promise's .then() method, loop through the films coming back and do something similar to this ...

```
dispatch({type:"ADD_FILM", film: theFilm});
```

22. In your App.js's useEffect(), dispatch({type:"FETCH_FILMS"}); Note that it should only fetch films on the first render or you may throw yourself into an endless loop.

If you debug at this point, you should see that the device is making the HTTP request just fine and even getting a response. But when our middleware dispatches "ADD_FILM", nothing happens yet. Let's fix that.

23. Make your reducer handle an action type called "ADD_FILM". The action should have a payload of a single film object. It should add that film to the films array. Something like this will do:
```
return {...state, films:[...state.films, action.film]};
```
24. Run and test. If you've done all that right, you should be seeing a bunch of films in state.films. Work with your pair programming partner until you've got that solved.

Displaying the film data

We have the films in App but we need them in Landing, so how do we get them there? Props, right? But soon, Landing will need much more than just films, so let's use the object spread trick to pass multiple props down.

25. In App.js, send the entire state object down into <Landing> as props:
```
<Landing {...state} />
```

Hey, now that they can be seen inside Landing.js, let's display them.

26. In the JSX of Landing.js, Array.prototype.map() through your films and put a <View> with two <Text>s. For each film, display the title of the film and the tagline. The key should be film.id
27. Run and test. Do you see all of the titles and taglines?

Adding the poster image

Movie posters can be downloaded from our data server at '/img/posters/<film_id>.jpg'. Let's show a poster for each film!

28. Remember that the different devices/emulators will honor different URLs so at the top of Landing.js,
```
import { host } from './store/api_host_helper.js'
```
29. Add to all of that an image, which is the movie poster. The posters can be served via http if you request them at film.poster_path. So make that the source. Here, something like this should work for you.
```
<Image source={{uri:`${host}/${film.poster_path}`}} style={{height: 100,
width: 100}} />
```
30. Run and test. You should be seeing the poster now.
31. But the poster may be cut off on the top or bottom. Try adding a resizeMode to the Image's style prop. "contain" would be a good choice here.
32. Run and test again. You should see the poster fully.

Extracting a FilmBrief component

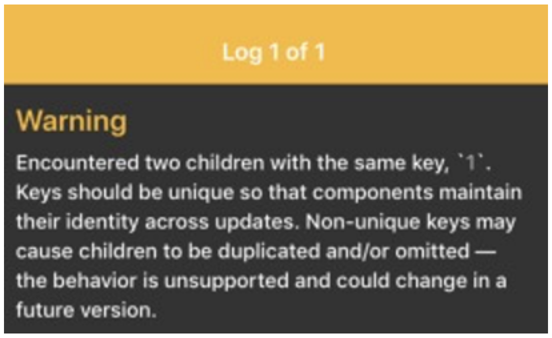
Right now we're listing films and displaying a film object in the Landing component. That's too many things according to SRP[1]. Let's give the film its own component.

33. Create a new component called FilmBrief.js. This should receive a film object in its props.
34. In the JSX, display one film. (Hint: you can cut some of your JSX from the Landing component. And don't forget the imports!).
35. Back in Landing.js, display a <FilmBrief film={film} key={film.id} /> instead of the <View>, <Text>s and the <Image>.
36. Run and test. You should still see your film data but now it is better designed.

[1] https://en.wikipedia.org/wiki/Single-responsibility_principle

Improving the FETCH_FILM logic

There's another complication, darn it. Currently, when you make any change to your <App />, Fast Refresh will preserve state and trigger a remount. It dispatches the FETCH_FILMS action and loads the same films again resulting in this warning message:



```
Log 1 of 1

Warning
Encountered two children with the same key, `1`.
Keys should be unique so that components maintain
their identity across updates. Non-unique keys may
cause children to be duplicated and/or omitted —
the behavior is unsupported and could change in a
future version.
```

37. Improve the business logic of FETCH_FILM so duplicate films aren't added. There are many ways to achieve this goal, pursue a path that seems fun to you.