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 --save 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));
```

13. We want to use react-redux to make it possible for React components to hook into the redux state. For that to happen, we need to wrap our React <App> in another component, the <Provider>. For this to happen, we're going to create a different entrypoint for expo other than App.js. 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'

registerRootComponent(App)
```

Note that I changed the expo's "default component export" pattern 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. Lastly, follow the instructions (https://react-redux.js.org/introduction/quick-start#provider) to wrap your App component in a provider, and have that be registered with expo.
16. Edit App.js. At the top, use the useSelector hook (https://react-redux.js.org/api/hooks#useselector) to access the redux store. You can use a very simple useSelector function that simply returns the entire redux state.
17. Run and test. Make sure that state is being read properly. (Hint: You could console.log({state}).)

At this point we have Redux ready to go but it doesn't do anything.

18. In initial state, give it these properties:

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

19. Run and test again. This time the state should have films (an array) and selected_film (an object).

Reading film data

20. 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]};
```

21. There are many ways to perform network requests within Redux, including redux-thunk, redux-saga, or writing plain ol' middleware. We're going to write middleware, but feel free to take a different approach if you would prefer. 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 http://localhost:3007/api/films. Await the result, then 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 the action to FETCH_FILMS. You'll need to get access to the dispatch function: https://react-redux.js.org/api/hooks#usedispatch
23. 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.

24. In App.js, create a <Landing /> component, which will be responsible for displaying films.

25. **Make a design choice**: this component should read `films` off the redux state and display them. Think about each of these options and the pros/cons of each. Would you…
    a. Pass the whole state object as a prop?
    b. Pass just the films array as a prop?
    c. Having <Landing /> manage its own reading of Redux state?

26. In the JSX of Landing.js, 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?
28. 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:`http://localhost:3007/${film.poster_path}`}}
style={{height: 100, width: 100}} />
```

29. Run and test. You should be seeing the poster now.
30. 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.
31. Run and test again. You should see the poster fully.

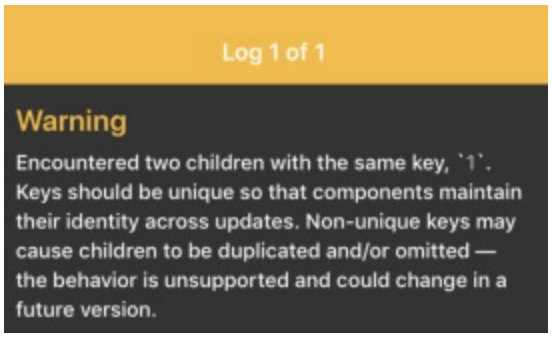
Extracting a FilmBrief component

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.

32. Create a new component called FilmBrief.js. This should receive a film object in its prop.
33. In the JSX, display one film. (Hint: you can cut some of your JSX from the Landing component).
34. Back in Landing.js, display a <FilmDetails film={film} key={film.id} /> instead of the <View>, <Text>s and the <Image>.
35. Run and test. You should still see your film data but now it is better designed.

Improving the FETCH_FILM logic

Currently, any changes to <App /> with Fast Refresh enabled will trigger a remount, and thus the FETCH_FILMS action is dispatched again with the same store. That middleware isn't currently designed to ignore films that were already added; you probably will get warning messages like this:



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.

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