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 and test. 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
```
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. Edit App.js. Convert it to a class-based component so we can maintain state in it.
14. import your store at the top and in the constructor, do this:
```
constructor() {
  super();
  this.state = store.getState();
  store.subscribe( () => this.setState(store.getState()) );
}
```

15. Run and test. Make sure that state is being read properly. (Hint: You could console.log(this.state) in your render method).

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

16. In initial state, give it these properties:

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

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

Reading film data

18. 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]};
```

19. To read film data from our RESTful API, you'll need to use Redux middleware. Note that there's already a middleware function called fetchFilmsMiddleware. It should check if the action.type is "FETCH_FILMS" and if so, it should make an Ajax call to http://localhost:5000/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});
```

20. In your App.js, create a componentDidMount() method and dispatch({type:"FETCH_FILMS"});
21. If you've done all that right, you should be seeing a bunch of films in this.state.films. Work with your pair programming partner until you've got that solved.

Displaying the film data

22. In App.js, send the array of this.state.films down into <Landing> via a prop called films.

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

23. 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
24. Run and test. Do you see all of the titles and taglines?
25. 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:5000/${props.film.poster_path}`}}
style={{height: 100, width: 100}} />
```

26. Run and test. You should be seeing the poster now.
27. 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.
28. 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. Let's give the film its own component.

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