Layout Components Lab

By now you've surely seen that not all of your movies fit on the landing scene. We're going to fix that first by using a ScrollView.

1. Open Landing.js and wrap your <View> in a <ScrollView>
2. Run and test. You can scroll now! Well, that was easy, wasn't it?

Now you might notice that some of the content is way too high on the screen, especially on an iPhone.

3. Add a <SafeAreaView> that wraps your <ScrollVew>.
4. Run and test. If you're on Android, it works but just doesn't do much for us. But if you're on iOS, this should look great.

Notice that the status bar is covering the top of our app. Let's see what it looks like if we hide the status bar.

5. Open App.js. Add the <StatusBar> control. First change it's barStyle property to 'dark-content' and then 'light-content'. See which you like better.
6. Then set the hidden property to true.
7. Run and test. The status bar should now be hidden. Note that you can still get to the status bar if you pull down from the top.

Selecting a film

When we have real customers on this app eventually, we'll show them the list of film briefs so they can select one by tapping it. Unfortunately, <View>s are not clickable. So let's wrap it in something that is.

8. In FilmBrief.js, surround your main <View> with a Pressable. Put something like this before your View.

```
<Pressable onPress={()=>selectThisFilm()}>
```

   (Hint: Don't forget to get a dispatch method from react-redux.)

9. And close off the Pressable after the <View>:

```
</Pressable>
```

10. Create a selectThisFilm method. Make sure it does this dispatch:

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

11. Of course do all the necessary work in reducers.js to make state.selected_film the one just tapped.
12. App's JSX should pass selected_film={state.selected_film} into <Landing> as a prop and Landing.js should pass isSelected (a boolean) down into <FilmBrief>. (Hint: isSelected={film===selected_film})
13. At this point, you should be able to tap on a film and see it populated in state as the *selected_film*. Debug your app or just put a console.log(isSelected) into FilmBrief to prove that a click is making the right film the selected_film.

Working with a Modal

FilmBrief was just a little bit of info about the movie. But when the user wants more details, let's show them those details in a Modal.

14. Inside Landing.js, add a <Modal> view. It should be just inside the top-level view (You probably wrote that as a <SafeAreaView> above).
15. Put a little <Text> in that Modal. Just something to see. And a <Button> titled "Done".
16. Run and test. You should be seeing your modal and there is no way to dismiss it yet.
17. Add an onPress event to the Button. It should

```
dispatch({type:"HIDE_FILM_DETAILS"})
```

18. Edit FilmBrief ... briefly. :-) In the selectThisFilm function, also

```
dispatch({type:"SHOW_FILM_DETAILS"})
```

19. Go back to Landing.js. Add a visible prop to the Modal. Set it equal to show_film_details which comes in as a prop.
20. In App.js, make sure you're passing state.show_film_details as a prop to Landing.js
21. Edit reducer.js. Add a case for HIDE_FILM_DETAILS and another for SHOW_FILM_DETAILS. All they need to do is set show_film_details to false and to true respectively.
22. Run and test again. The modal should start out hidden but you can show it by choosing any film. Then dismiss it with a click of the "Done" button.

Showing the film details

23. Create a new component called FilmDetails.js. It should receive a film object, selected_date, and an array called *showings* as props.
24. Make this component show all of the details of the film. Put them in <Text>s inside a root <View> or fragment. Again, don't worry about layout or styling until later.
25. Let's tell the user when they can see the movie:

```
<View>
  <Text>Showing times for {selected_date.toDateString()}</Text>
  {props.showings.map(showing => <Text key={showing.id}>
                                   {showing.showing_time}
                                 </Text> )}
  (All your other film <Texts> go here)
</View>
```

26. Finally nest this new <FilmDetails /> in the Modal instead of the placeholder text you added earlier. Pass the selected_film and selected_date in as props.
27. Run and test by hardcoding some values. Or if you'd prefer, there's a file of showings in starters called showings.json you can use.
28. Bonus!! If you have extra time, put the details inside a ScrollView to make it layout just a bit better.
29. One last thing. It would be cleaner to have those showing times in their own component. Besides, we may want to reuse that component in multiple places. Go ahead and extract it into its own component called ShowingTimes.js which should receive showings and selected_date as props.

Use a keyboardHidingView

After the user has selected their movie and date and has selected their seats we'd like for them to actually pay us. That'd be nice, wouldn't it? So let's create a Checkout component.

30. Write a new component called Checkout.js.
31. We're going to be needing state in this component so add some hooks:

```
const [firstName, setFirstName] = useState(props.firstName);
const [lastName, setLastName] = useState(props.lastName);
const [creditCard, setCreditCard] = useState(props.creditCard);
const [email, setEmail] = useState(props.email);
const [phone, setPhone] = useState(props.phone);
```

32. Add a <SafeAreaView> as the root component. Put a <ScrollView> inside of that.
33. Put a <Text> at the top to tell the user we're checking out.
34. Add <TextInput>s for first and last name, credit card, email, phone. Go ahead and give them <Text>s above each so the user knows what each is for. Here's an example for just the firstName field:

```
<Text>First name</Text>
<TextInput value={firstName} onChangeText={setFirstName} />
```

35. Make the credit card <TextInput> show a number-pad keyboard.
36. Make the email <TextInput> show an email-address keyboard.
37. Make the phone <TextInput> show a phone-pad keyboard.
38. Add a <Button> titled "Purchase". It'll force you to add an onPress event. Make that run a function called purchase.
39. We'll learn how to properly navigate to Checkout later. But for now, edit App.js and just comment out <Landing> and add <Checkout>.

40. Run and test. Try to enter some text in each <TextInput>. Depending on the emulator you're using, the soft keyboard may slide up. If not, force it to come up through the emulator settings.

You may notice a problem. When the keyboard slides in the view it covers (or occults) some content behind it. This could be a problem someday. The solution is a KeyboardAvoidingView.

41. Edit Checkout.js. Add a <KeyboardAvoidingView> around the root <ScrollView> of the component.
42. Run and test again. Do you like this better? (Note: You may need to put a bunch of Lorem Ipsum text above your form to see the behavior. Try that if you don't see a difference).
43. Add a behavior property to the KeyboardAvoidingView. Switch between position, padding, and height to see the differences. Set it to the one you like best.
44. Bonus!! Add a <ScrollView> just inside the <KeyboardAvoidingView> so whether the keyboard is up or out, we can scroll through the content.
45. Before finishing, don't forget to uncomment <Landing> and comment out <Checkout> in App.js. Also delete all your Lorem Ipsum test. Then, you can be finished.