

React Native One Day Course

Paul Barriere, Cazton consultant

The plan



- Expo online demo
- Get Expo running locally
- React's Composition Model and JSX
- Hooks
- State Management with Redux
- Sharing Behavior with components
- Routing
- Data Loading and Caching
- Realtime Updates
- Client side Authentication

Snack time!

<https://snack.expo.io>

—

Get Expo running locally

Git clone <https://github.com/webpaul/react-native-example>

Run install.bat from git repo

Type npm install -g expo-cli into git bash window opened

Close git bash and all other windows, run expo-cli --version from command line -> 3.11.1

Go to hello-world folder and run expo start

Install Expo client from Play Store, scan QR code from expo start

What is Expo?

Get up and running quickly, without installing full Android or iOS development environment

Interact with the phone through the Expo SDK for camera, contacts, notifications, motion, etc.

Not everything accessible through Expo such as Bluetooth and other more complex things

<https://docs.expo.io/versions/latest/introduction/why-not-expo/>

So why use it? Get up and running quickly, save the complications for when you really need to worry about them (if ever).

Once you hit “the wall” execute “npm run eject” which will convert to a pure Native React app, then setup full development environment and all its complications

Eject!

Once you eject a project, there is no going back unless you revert in source control

You'll need to use Android Studio and/or XCode to continue to develop

But you'll have access to all the lower level features, libraries, etc. for each development environment

BUT you'll have to be very familiar with the tooling and details of each of those environments

So for today we will stick with Expo and keep it simple.

Try running `npm run eject`, look at the changed files and then revert them using git!

React Native vs. React

React Native does not run as a web page, it is converted to run as native code in a managed environment, similar to .NET, PhoneGap and others.

No className or CSS imports, use React Native Stylesheet.

Any libraries that use HTML or CSS you will not be able to use. Pure JavaScript libraries will work though!

All tags in React Native must be React Native component, not HTML. So you will see `<Text>` instead of `<div>`, `<form>`, similar to WPF.

Example:

Instead of `` -> `<Image source="myPic.gif" />`

React's Composition Model and JSX

```
3
4 function FancyBorder(props) {
5   return (
6     <View style={{backgroundColor: props.color}}>
7       <Text>Fancy border!</Text>
8       {props.children}
9     </View>
10  );
11 }
12
13 export default function App() {
14   return (
15     <View style={styles.container}>
16       <Text>Open up App.js to start working on your app!</Text>
17
18       <FancyBorder color='green'>
19         <Text style={{color: 'white'}}>Composition Text!</Text>
20       </FancyBorder>
21     </View>
22   );
23 }
24
```

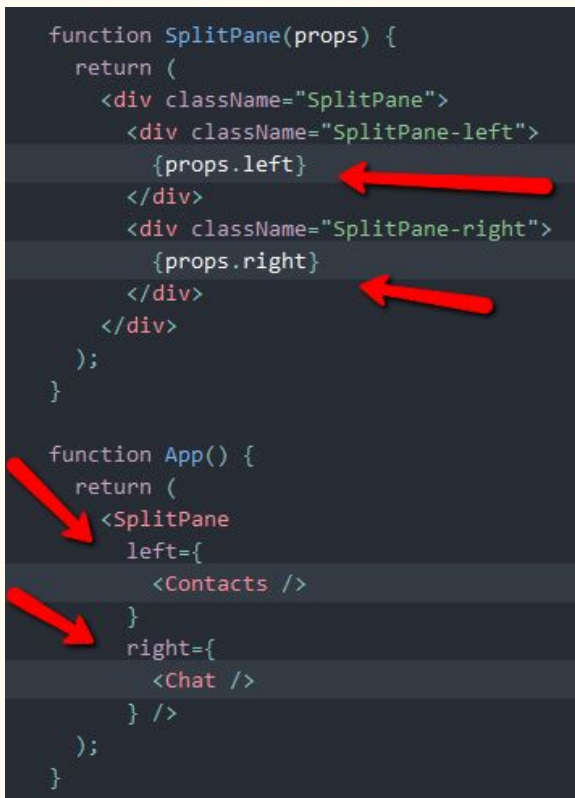
“Children” property gives you all inner JSX content passed.

Other properties can be used as attributes such as “color.”

This example also has inline styles

React's Composition Model and JSX

```
function SplitPane(props) {  
  return (  
    <div className="SplitPane">  
      <div className="SplitPane-left">  
        {props.left}  
      </div>  
      <div className="SplitPane-right">  
        {props.right}  
      </div>  
    </div>  
  );  
}  
  
function App() {  
  return (  
    <SplitPane  
      left={  
        <Contacts />  
      }  
      right={  
        <Chat />  
      } />  
  );  
}
```

The diagram illustrates the composition model in React. It shows two functions: SplitPane and App. SplitPane is a component that takes props and returns a JSX element. App is a component that returns a JSX element. Red arrows indicate the flow of data and component composition. One arrow points from the {props.left} prop in SplitPane to the <Contacts /> element in App. Another arrow points from the {props.right} prop in SplitPane to the <Chat /> element in App. A third arrow points from the <SplitPane> element in App to the SplitPane function definition. A fourth arrow points from the <SplitPane> element in App to the {props.left} prop in SplitPane.

You can also create your own “children” properties if you need more than one collection in your control.

React's Composition Model and JSX

Modify the composition example to take two parameters “left” and “right” with arbitrary content and display them next to each other, one on the left and one on the right.

React's Composition Model and JSX

```
function Dialog(props) {  
  return (  
    <FancyBorder color="blue">  
      <h1 className="Dialog-title">  
        {props.title}  
      </h1>  
      <p className="Dialog-message">  
        {props.message}  
      </p>  
    </FancyBorder>  
  );  
}  
  
function WelcomeDialog() {  
  return (  
    <Dialog  
      title="Welcome"  
      message="Thank you for visiting our spacecraft!" />  
  );  
}
```

Composition is usually used instead of inheritance

React's Composition Model and JSX

```
function Dialog(props) {  
  return (  
    <FancyBorder color="blue">  
      <h1 className="Dialog-title">  
        {props.title}  
      </h1>  
      <p className="Dialog-message">  
        {props.message}  
      </p>  
      {props.children}  
    </FancyBorder>  
  );  
}
```

You can also use
composition for classes
and return JSX in the
“render” method.

```
class SignUpDialog extends React.Component {  
  constructor(props) {  
    super(props);  
    this.handleChange = this.handleChange.bind(this);  
    this.handleSignUp = this.handleSignUp.bind(this);  
    this.state = {login: ''};  
  }  
  
  render() {  
    return (  
      <Dialog title="Mars Exploration Program"  
        message="How should we refer to you?">  
        <input value={this.state.login}  
          onChange={this.handleChange} />  
  
        <button onClick={this.handleSignUp}>  
          Sign Me Up!  
        </button>  
      </Dialog>  
    );  
  }  
  
  handleChange(e) {  
    this.setState({login: e.target.value});  
  }  
  
  handleSignUp() {  
    alert(`Welcome aboard, ${this.state.login}!`);  
  }  
}
```

React's Composition Model and JSX

Modify the composition example to use a class instead of a method to return values

Hooks

```
3
4 export default function App() {
5   const [count, setCount] = useState(0);
6
7   return (
8     <View style={styles.container}>
9       <Text>Hooks {count}!</Text>
10      <Button title="One more" onPress={() => setCount(count + 1)}></Button>
11    </View>
12  );
13 }
14
```

Think of hooks like mini state management for controls, a page or some UI element

Hooks - Rules

Only call Hooks **at the top level**. Don't call Hooks inside loops, conditions, or nested functions.

Only call Hooks **from React function components**. Don't call Hooks from regular JavaScript functions. (There is just one other valid place to call Hooks — your own custom Hooks. We'll learn about them in a moment.)

Component state for component state, Redux for application state.

Hooks - Use When:

Doesn't use the network.

Doesn't save or load state.

Doesn't share state with other non-child components.

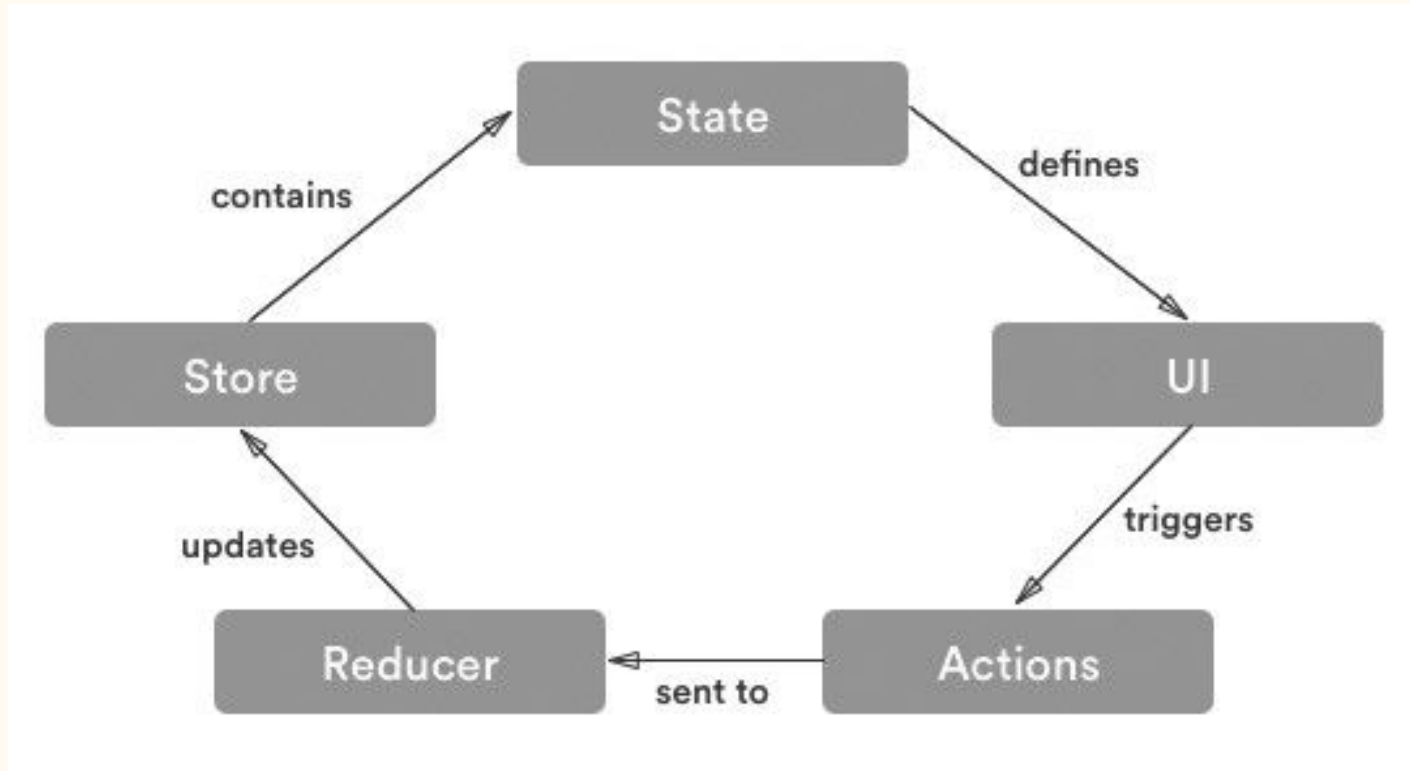
Does need some ephemeral local component state.

Hooks

Modify the hooks example so that every third time the button is clicked the color changes in the “Hooks” text

Use AsyncStorage to save the count so that if you refresh or kill the app the count persists once the app is reopened.

State Management with Redux



State Management with Redux

Hooks maintain state within a component, use Redux for larger state management concerns such as the state of all data on the page, results from an API call, UI options chosen that will be used on multiple pages, etc.

Install redux and react-redux which is an official layer for managing redux in React

npm install redux react-redux --save

```
1  import { combineReducers, createStore, } from 'redux';
2  import { connect } from 'react-redux';
3
4  const testAction = () => ({
5    type: 'TEST_ACTION',
6  });
7
8  var testReducer = (state = { count: 0 }, action) => {
9    switch (action.type) {
10     case 'TEST_ACTION':
11       return {
12         content: "Hello Redux",
13         count: state.count+1
14       };
15     default:
16       return state;
17   }
18 };
19
20 const mapStateToProps = (state) => ({
21   testReducer: state.testReducer,
22 });
23
24 const mapDispatchToProps = (dispatch) => {
25   return {
26     testAction: () => {
27       dispatch(testAction())
28     }
29   }
30 };
31
32 export const redux = {
33   connect: connect(mapStateToProps, mapDispatchToProps),
34   store: createStore(combineReducers({
35     testReducer,
36   })))
37 }
```

State Management with Redux

After setting up the actions, reducers and stores, the Redux Provider must be setup and connected with your UI.

All redux functionality goes through the standard “props” variable used for extending components as seen previously.

Actions will also be triggered from the “props” variable (See Button)

```
4 import { redux } from './redux';
5 import { Provider } from 'react-redux';
6
7 class AppInner extends React.Component {
8   render() {
9     return (
10       <View style={styles.container}>
11         <Button onPress={this.props.testAction} title="Test!" />
12         <Text>
13           {this.props.testReducer.content || "No current state to display"}
14           {this.props.testReducer.count} times
15         </Text>
16       </View>
17     );
18   }
19 }
20
21 const AppWithRedux = redux.connect(AppInner);
22
23 export default class App extends React.Component {
24   render() {
25     return (
26       <Provider store={redux.store}>
27         <AppWithRedux />
28       </Provider>
29     );
30   }
31 }
```

State Management with Redux

Change the names of some of the variables in `redux.js` and `App.js` and try to keep the code working. When you change the variable name in one page and it breaks in the other you will start to understand the example and how everything interacts more. Change them one at a time and keep checking if the app still works!

Components

We have some copy and pasted code in both screens of the redux example. Look at the components folder for how to create a reusable component and make one called reduxCounter to use in both screens without copy and paste (the root of all evil)

```
components > 📄 TabBarIcon.js > ...
1  import React from 'react';
2  import { Ionicons } from '@expo/vector-icons';
3
4  import Colors from '../constants/Colors';
5
6  export default function TabBarIcon(props) {
7    return (
8      <Ionicons
9        name={props.name}
10        size={26}
11        style={{ marginBottom: -3 }}
12        color={props.focused ? Colors.tabIconSelected : Colors.tabIconDefault}
13      />
14    );
15  }
16
```

Routing

npm install --save react-navigation

React Navigation is a JavaScript-based library for routing. It's officially promoted by both Facebook and the React Native documentation as the primary solution for routing. But it is certainly not the only available option!

A StackNavigator works exactly like a call stack or a stack of dishes. Each screen we navigate to is pushed to the top of the stack, and each time we hit the back button, this screen pops off the top of the stack.

```
import { createAppContainer } from 'react-navigation';
import { createStackNavigator } from 'react-navigation-stack';
import HomeScreen from './HomeScreen';
import OtherScreen from './OtherScreen';

const AppNavigator = createStackNavigator({
  HomeScreen: { screen: HomeScreen },
  OtherScreen: { screen: OtherScreen },
});
const AppContainer = createAppContainer(AppNavigator);

export default AppContainer;
```

```
import React from 'react';
import AppNavigator from './AppNavigator';

export default class App extends React.Component {
  render() {
    return (
      <AppNavigator/>
    );
  }
}
```

```
export default class HomeScreen extends React.Component {
  render() {
    return (
      <View style={styles.container}>
        <Text>Home screen</Text>
        <Button
          title="Show me the other"
          onPress={() =>
            this.props.navigation.navigate('OtherScreen')
          }
        />
      </View>
    );
  }
}
```

Routing

Create a third page for the routing example. Add a second button on every page that goes to the other two pages.

Data Loading and Caching

Use fetch (built-in with React) to retrieve API and other data from the web.

Cache with state, hooks or Redux!

```
fetch('https://mywebsite.com/endpoint/', {  
  method: 'POST',  
  headers: {  
    Accept: 'application/json',  
    'Content-Type': 'application/json',  
  },  
  body: JSON.stringify({  
    firstParam: 'yourValue',  
    secondParam: 'yourOtherValue',  
  }),  
});
```

Data Loading and Caching

Using the data-fetching project as an example, modify the UI to display two weather boxes instead of just one. Arbitrarily modify the device's position information (or hard code lat/long) and display both weather conditions on two different pages with buttons to get back and forth between them.

Realtime updates

If you're using Redux and polling is ok:
update the Redux cache on a schedule
and the UI will be updated
automagically!

Another option is doing it manually
using socket.io or through a cloud
service such as Pusher, shown here
and in sample “realtime”.

```
12
13
14   onSendMessage(e) {
15     this.props.onSendMessage(e.nativeEvent.text);
16     this.refs.input.clear();
17   }
18
19   render() {
20     return (
21       <KeyboardAvoidingView style={styles.container} behavior="padding">
22         <FlatList data={ this.props.messages }
23                   renderItem={ this.renderItem }
24                   style={ styles.messages }
25                   ref="messages" />
26
27         <TextInput autoFocus
28                   keyboardType="default"
29                   returnKeyType="done"
30                   enablesReturnKeyAutomatically
31                   style={ styles.input }
32                   blurOnSubmit={ false }
33                   onSubmitEditing={ this.handleSendMessage }
34                   ref="input"
35                   />
36       </KeyboardAvoidingView>
37     );
38   }
39 }
```

Pseudo realtime update

Modify your data fetching example so that the weather is fetched every 5 seconds for the weather conditions that you are arbitrarily changing the lat/long for. Store the forecast information in redux using the redux example to guide you. Add another page that displays the information from the redux cache without fetching it.

Enforce login for some pages

If not logged in, show login page. If logged in show protected page. Use the SwitchNavigator which disables the back button, pages can only be navigated to explicitly.

We will nest the StackNavigator into a SwitchNavigator so that the public and private pages are kept separate.

```
AppNavigator.js > ...
import { createAppContainer, createSwitchNavigator } from 'react-navigation';
import { createStackNavigator } from 'react-navigation-stack';
import HomeScreen from './HomeScreen';
import OtherScreen from './OtherScreen';
import LoginScreen from './LoginScreen';

const AppStack = createStackNavigator({ Home: HomeScreen, Other: OtherScreen });
const LoginStack = createStackNavigator({ SignIn: LoginScreen });

export default createAppContainer(createSwitchNavigator(
  {
    Login: LoginStack,
    App: AppStack,
  },
  {
    initialRouteName: 'Login',
  }
));
```

Client side authentication

Create your own express API that contains a login endpoint that checks for a hard coded login and password, returning HTTP code 201 if matching, 400 if not.

(Bonus) Show an activity indicator while the API request is running.

Create a login screen that is the default screen for your app. Submit the info to the API and if it is successful add an indicator to AsyncStorage and send users to a series of 3 pages with buttons that navigate to the other two pages.

Add a header control that contains a logout button which removes the AsyncStorage setting. In that control also check if the redux indicator is valid, if not navigate to the login page.

(Bonus) Use Redux instead of AsyncStorage