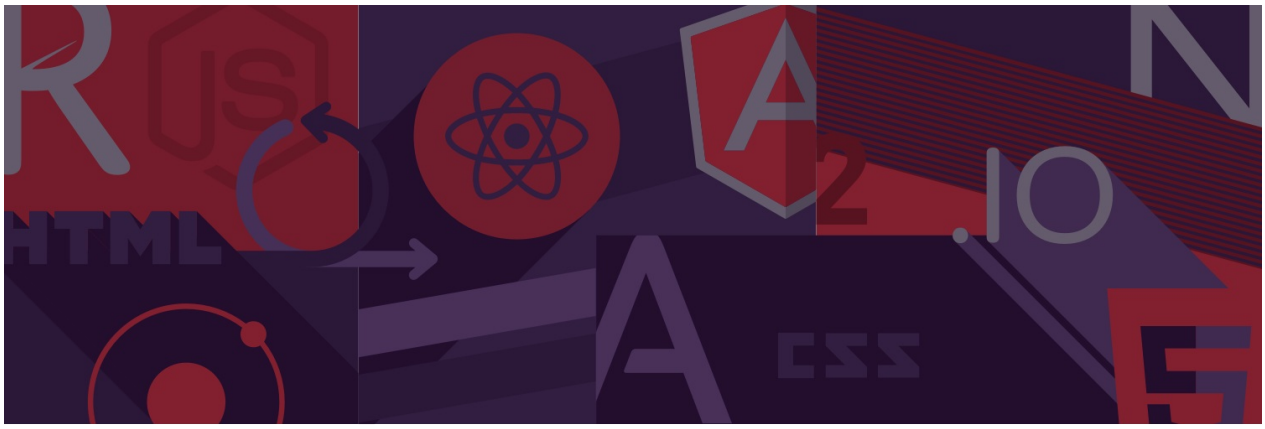
# Table of Contents


- Introduction 1.1
- Setup 1.2
  - Hello World 1.2.1
  - Debugging 1.2.2
- Core Concepts 2.1
  - Components 2.1.1
  - Styles 2.1.2
  - Flexbox 2.1.3
  - APIs 2.1.4
- Navigation 3.1
  - Navigation Experimental 3.1.1
  - Creating Some Helpers 3.1.2
  - The Navigator Component 3.1.3
- ListView (Pokédex) 4.1
  - List View 4.1.1
  - List View: Render Row 4.1.2
  - Text Input 4.1.3
  - Keyboard Spacer 4.1.4
- Selectors (Filtering) 5.1
- Animation 6.1
  - LayoutAnimation 6.1.1
  - Animated 6.1.2
    - Animated.Value 6.1.2.1
    - Timing, Spring & Decay 6.1.2.2
    - Animated Components 6.1.2.3
    - More Animated 6.1.2.4
  - Resources 6.1.3
  - Exercise 6.1.4
- ScrollView (Pokemon Details) 7.1
- MapView 8.1

Testing 8.2
Gesture Responder System 8.3
PanResponder 8.3.1

# React Native Workshop




## Who is this material for?

This material is for those who are familiar with ReactJS and are willing to dive into developing mobile apps with React Native. For this course we assume that our audience has solid knowledge of JavaScript, ES6 syntax, Redux, CSS, and Flexbox for layouts.

## What is React Native?

React Native is a framework for building mobile applications with JavaScript and ReactJS by leveraging native UI components.

In ReactJS we have a virtual DOM which reflects the real DOM.

Each element corresponds to a node in the Virtual DOM and when an element changes, that change is reflected onto the real DOM. In React Native we are not using the DOM but Native Components which are provided by specific platforms. Instead of dealing with WebViews, we use actual platform specific native components.

For example, instead of using HTML elements such as `<div>` & `<span>` we use the native components such as `<View>` & `<Text>`. This course we will introduce other, more complex, native components and some platform specific components which look and behave differently on each platform.

React Native embraces the *Learn Once And Apply Everywhere* paradigm, which is quite different from *Write Once Use Everywhere*. With React Native we can use the concepts learned from React to build separate apps for Android and iOS reusing most of the business logic code for both platforms.

# How does it work?

React Native has an embedded instance of JavaScriptCore. When your app starts, the JavaScript code is loaded and executed in this engine.

Using the `RCTBridgeModule` it bridges native code to JavaScript. This allows the JSX components to have bindings to native UI components.

# Setup

You can find the most up to date information about how to get started here.

# Install Dependencies

## Update Brew for OS X

Since `brew` will be used to install all the needed tools, you should update it to ensure that you will get the most recent versions of all the required programs:

```
$ brew update && brew upgrade
```

## Install Node and NPM

```
$ brew install node
```

Alternatively you can download an installer from: nodejs.org/en/download. Recommend to have `Node >=4.0` and `NPM >=3.0`

## Install Watchman

This tool will be used by React Native to detect changes of your code and auto reload your application. **Install watchman via brew, and not npm.**

```
$ brew install watchman
```

## Install the React Native CLI

```
$ npm install -g react-native-cli
```

# Setup Native SDKs

- For iOS install Xcode from the OS X App Store.
- For Android follow these instructions here.

# Hello World

To initiate a new React Native project you need to run: `react-native init <ProjectName>`. `react-native init` generates the following:

- `index.ios.js`
- `index.android.js`
- iOS (Xcode) project
- Android projects

For this workshop, **we have already done `react-native-init` for you and setup a skeleton project**. To get started:

```
$ git clone https://github.com/rangle/react-native-workshop.git
$ cd react-native-workshop
$ npm install
$ git checkout 1-hello-world
```

**Each section of the workshop is available in a separate git branch, so `1-hello-world` is the first one.** Let's open the project in a text editor to go through the generated code.

# Bootstrapping

In order to bootstrap a React Native app we use `AppRegistry` instead of `ReactDOM`, for example:

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

class Root extends Component {
  ...
}

AppRegistry.registerComponent('ApplicationName', () => Root);
```

# Run the App

- iOS

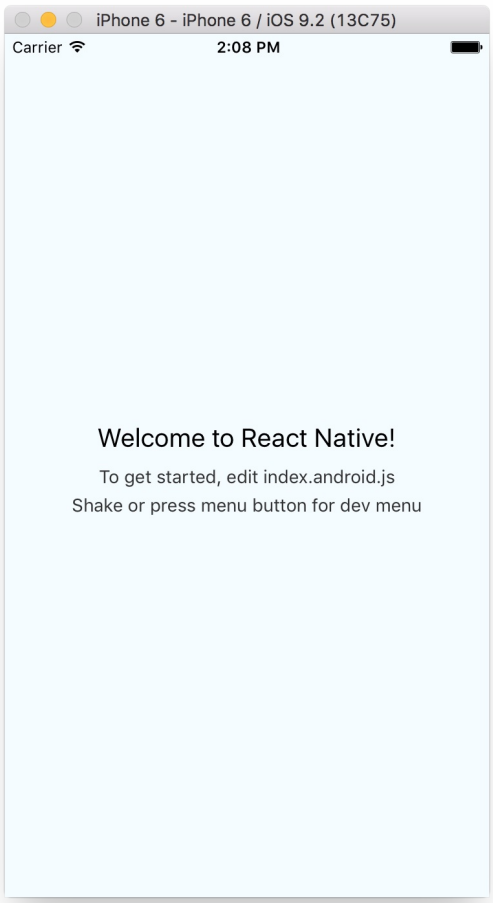
```
$ react-native run-ios
```

or Open `/Users/<userName>/reactNativeWorkshop/ios/reactNativeWorkshop.xcodeproj` in Xcode

- Android

```
$ react-native run-android
```

You should see something like this:

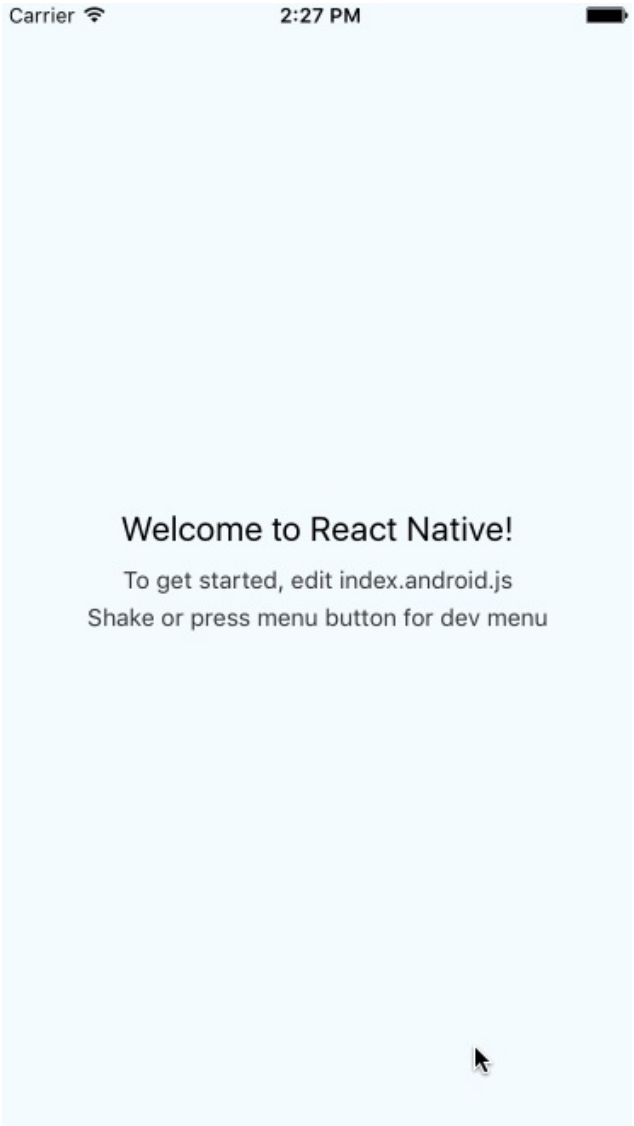


# Debugging

React Native provides several tools to make debugging easier. To access the in-app developer menu:

- Press `⌘ + d` in the iOS simulator
- `⌘ + m` or `F2` in the Android emulator
- Alternatively use the shake gesture:
  - `control + ⌘ + z` in the iOS simulator
  - Clicking on the menu button in the Genymotion Android simulator

You can use this menu to enable/disable live reloading, hot reloading, component inspector, etc.

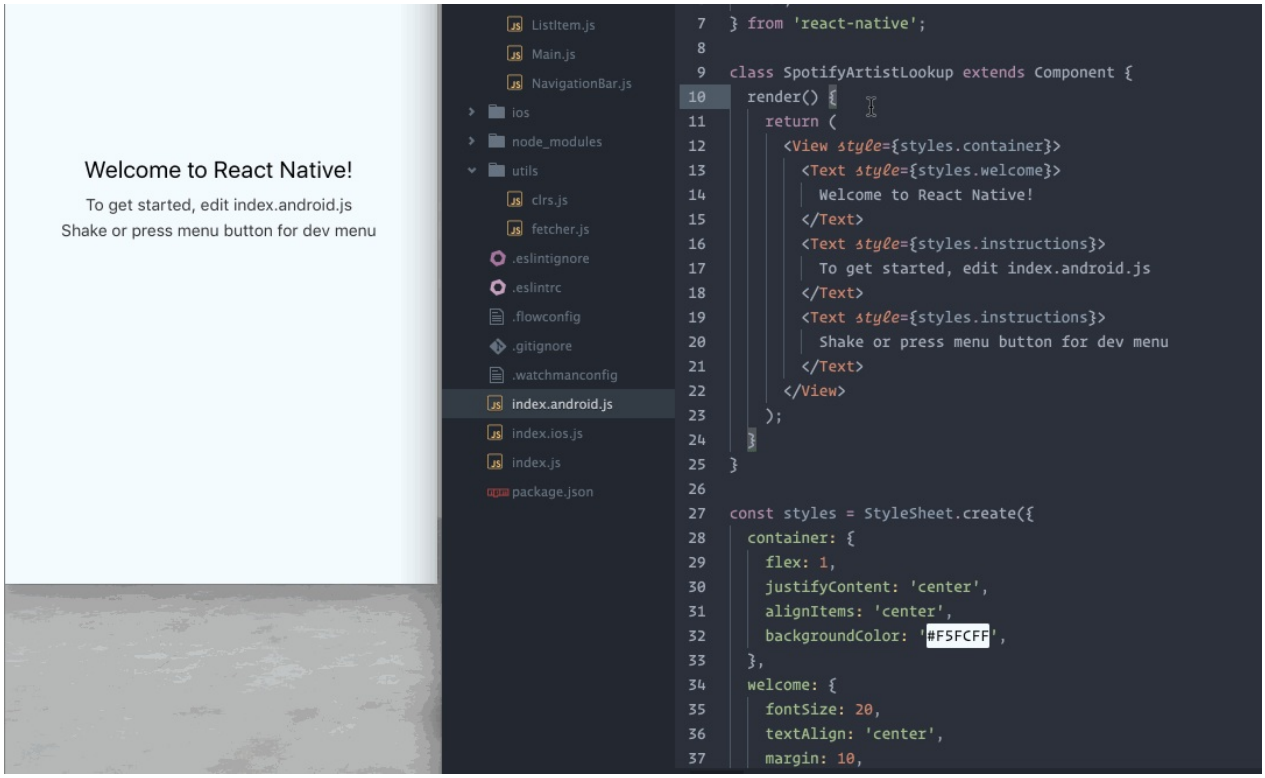


# Debug in Chrome

This option allows you to debug your JavaScript code in Google Chrome. The code is executed in a Chrome tab you have access to all the usual devtools such as: `debugger` statements, break-points, console logging, etc.

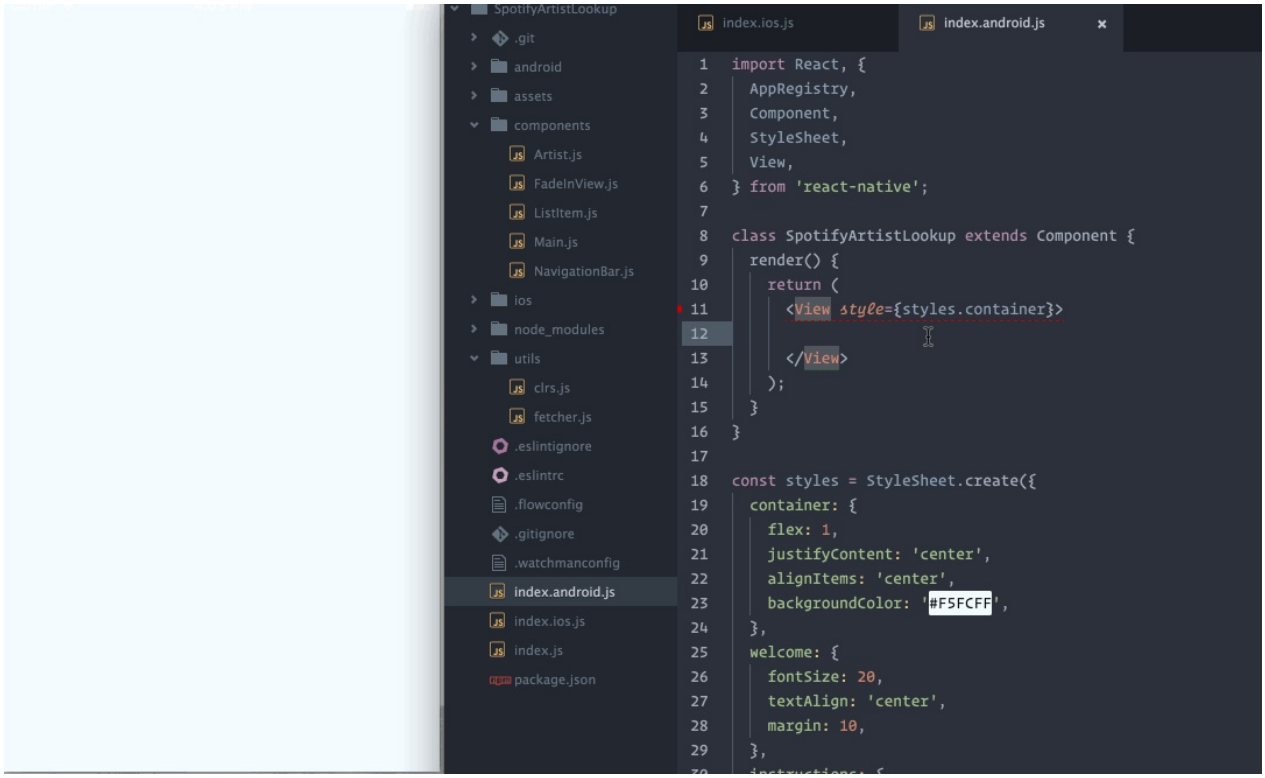
## YellowBox

> Using console.warn will display an on-screen log on a yellow background. Click on this warning to show more information about it full screen and/or dismiss the warning.




## RedBox

> You can use console.error to display a full screen error on a red background.




More info on Yellow/Red Box available here: rn-docs/debugging.html#yellowbox-redbox

# Core Concepts

React Native communicates with the native UI components through a bridge. It exposes the native layer to JavaScript as both React components and APIs.

On the other hand it also provides polyfills for certain APIs that we have available on the web. This makes it easier for web developers to transition over to the React Native platform using technologies they know and love.

# Components

React Native provides JSX wrappers for several native UI components, like `View`, `ScrollView`, `Text`, `TextInput`, etc. Most components work on both iOS and Android. If a component is limited to one platform then it is indicated in the name, for example: `ActivityIndicatorIOS` or `ProgressBarAndroid`.

The 3 basic building blocks for layouts are:

## 1. View

The most fundamental component for building UI in React Native. Equivalent to `<div>` in HTML. It maps to `UIView` and `android.view`.

## 2. ScrollView

A scrolling container that allows you to place content larger than the container within it. Similar to `overflow: scroll` on the web. It requires a bounded height in order to work – either set directly on the component or by setting it on a parent view.

## 3. ListView

Allows you to efficient display vertically scrolling lists of changing data. It has several performance optimizations and works well for creating infinite scrolls. Additionally, it supports sticky headers and grouping of data. The data needs to be passed in as an instance of `ListView.DataSource`.

# Platform Specific Behaviour

To support platform specific functionality, React Native determines the component to use based on the platform and a simple naming convention:

```
// MyComponent.ios.js
// MyComponent.android.js

import MyComponent from './components/MyComponent';
```

# Styles

React Native allows you to style components using a subset of CSS properties as inline styles. For layout only the flexbox module and absolute positioning is available. The style properties are split into five categories:

1. View Properties
2. Image Properties
3. Text Properties
4. Flex Properties
5. Transform Properties

# Usage for Static Styles

Use `StyleSheet.create` to construct styles and define them at the end of the file. This ensures that the values are immutable and they are only created once for the application and not on every render.

```
var RedBox = ({ children }) => {
  return (
    <View style={ styles.base }>
      { children }
    </View>
  );
};

const styles = StyleSheet.create({
  base: {
    backgroundColor: '#222222',
    color: '#fff',
  },
  active: {
    backgroundColor: '#85144b',
    color: '#B10DC9'
  },
});
```

You can also compose styles

```
// As an array
<View style={[styles.base, styles.background]} />
// or conditionally
<View style={[styles.base, this.state.active && styles.active]} />
```

# Usage for Dynamic Styles

Dynamic styles can be created as objects in the render. However, the official documentation recommends that you avoid this:

> Finally, if you really have to, you can also create style objects in render, but they are highly discouraged. Put them last in the array definition.

```
var RedBox = ({ children, width }) => {
  return (
    <View style={[styles.base, {
      width: this.state.width,
    }]}>
      { children }
    </View>
  );
};

const styles = StyleSheet.create({
  base: {
    backgroundColor: '#222222',
    color: '#fff',
  },
  active: {
    backgroundColor: '#85144b',
    color: '#B10DC9'
  },
});
```

# Flexbox

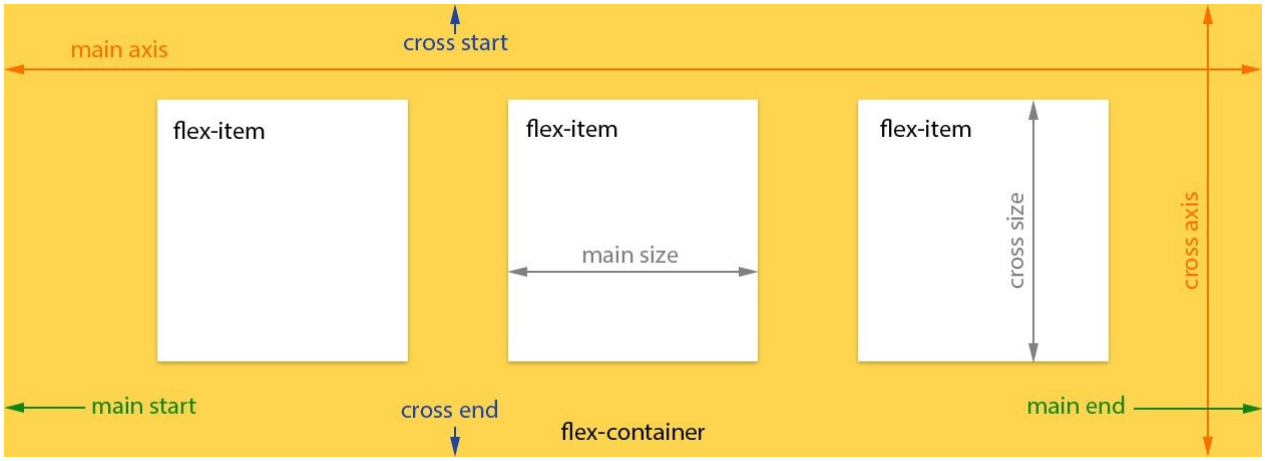



**Image from scotch.io/tutorials/a-visual-guide-to-css3-flexbox-properties**

## Single Line

## Flex Wrap

## Align Content

# APIs

Certain native functionalities are exposed as APIs. For example, datepicker is a component on iOS: `DatePickerIOS`. However, it's an API on Android: `DatePickerAndroid`.

## Commonly Used APIs

**Dimensions** provides you the screen size.

**AsyncStorage** provides a `LocalStorage` style API for storage on the device.

**InteractionManager** allows you to register long-running work to be scheduled after any interactions/animations have completed.

**PixelRatio** class gives access to the device pixel density.

## Polyfills

React Native provides native polyfills for the following APIs:

- Flexbox layout module
- `ShadowPropTypesIOS` so that you can define shadows in CSS
- Geolocation which follows the web spec: https://developer.mozilla.org/en-US/docs/Web/API/Geolocation
- Network fetch, XHR & WebSockets
- Timers: `setTimeout`, `requestAnimationFrame`, `setInterval`, etc.
- Named colors in CSS

# Navigation

You have a few options for building out the navigation of your app with React Native. For example there is a `NavigatorIOS` component which you can use to build out a simple navigation for your iOS specific apps. There is also the generic `Navigator` component which can be used for either iOS or Android Applications.

See the navigator comparison documentation for more information.

In this workshop we will be working with a third option: `NavigationExperimental`. `NavigationExperimental` differs from the `Navigator` component in that it attempts to be more like Redux using a single-direction flow of data and reducers to manage its state.

*At the time of this writing, `NavigationExperimental` is replacing the `Navigator`. The documentation will be updated to reflect this.*

# Some Setup

We're going to need to add a few things to our project that will be used by our Navigator. Firstly, we will need to create some navigation specific actions, as well as the reducers our redux store will need to use to properly manage our state.

Let's get started by adding a few new constants to our `src/constants/index.js` file:

```
// src/constants/index.js /

export const GOTO_ROUTE = '@@navigator/GOTO_ROUTE';
export const ROUTE_IDS = {
  MAIN_NAVIGATOR: 'MAIN_NAVIGATOR',
  POKEDEX: 'POKEDEX',
  POKEMON_DETAIL: 'POKEMON_DETAIL',
};

export const ROUTES = {
  POKEDEX: {
    key: ROUTE_IDS.MAIN_NAVIGATOR,
    index: 0,
    children: [{key: ROUTE_IDS.POKEDEX, title: 'Pokedex' }],
  },
  POKEMON_DETAIL: (title, url) => ({
    key: ROUTE_IDS.POKEMON_DETAIL,
    index: 1,
    title,
    url
  }),
};
```

Now that we've setup these constants, we'll add our navigation specific actions:

```
// src/actions/index.js

import {GOTO_ROUTE, ROUTES} from '../constants';

// ...

export function goToPokemonDetail({ title, url }) {
  return {
    type: GOTO_ROUTE,
    payload: ROUTES.POKEMON_DETAIL(title, url),
  };
}

export function gotoPokedex() {
  return {
    type: GOTO_ROUTE,
    payload: ROUTES.POKEDEX,
  };
}

export function onNavigate(payload) {
  return {
    type: GOTO_ROUTE,
    ...payload,
  };
}

// ...
```

The only other thing left to do now is setup our navigation's reducer!

```
// src/reducers/navigator.js

import {ROUTES, GOTO_ROUTE} from '../constants';
import {NavigationExperimental} from 'react-native';

const { Reducer: NavigationReducer } = NavigationExperimental;

const navigatorReducer = NavigationReducer.StackReducer({
  getPushedReducerForAction: action => {
    if (action.type === GOTO_ROUTE) {
      /* The below line is confusing as other NavigationReducer's
      may consist of a state, however for StackReducer, it is always null
      so the action.payload is taken as the next route */
      return state => state || action.payload;
    }

    return null;
  },
  getReducerForState: initialState => state => state || initialState,
  initialState: ROUTES.POKEDEX,
});

export default navigatorReducer;
```

# NavigationRootContainer

The `NavigationRootContainer` is the component that handles your application's navigation state. Normally you would pass a reducer to this component and it will handle returning a new navigation state when it's onNavigate method has been called.

There is also the `NavigationContainer.create` method, which acts similar to react-redux's `connect` method. It does not pull in any state however, it passes an `onNavigate` function to the component it wraps.

When using Redux, this component isn't necessary, because we are already using Redux to handle our state!

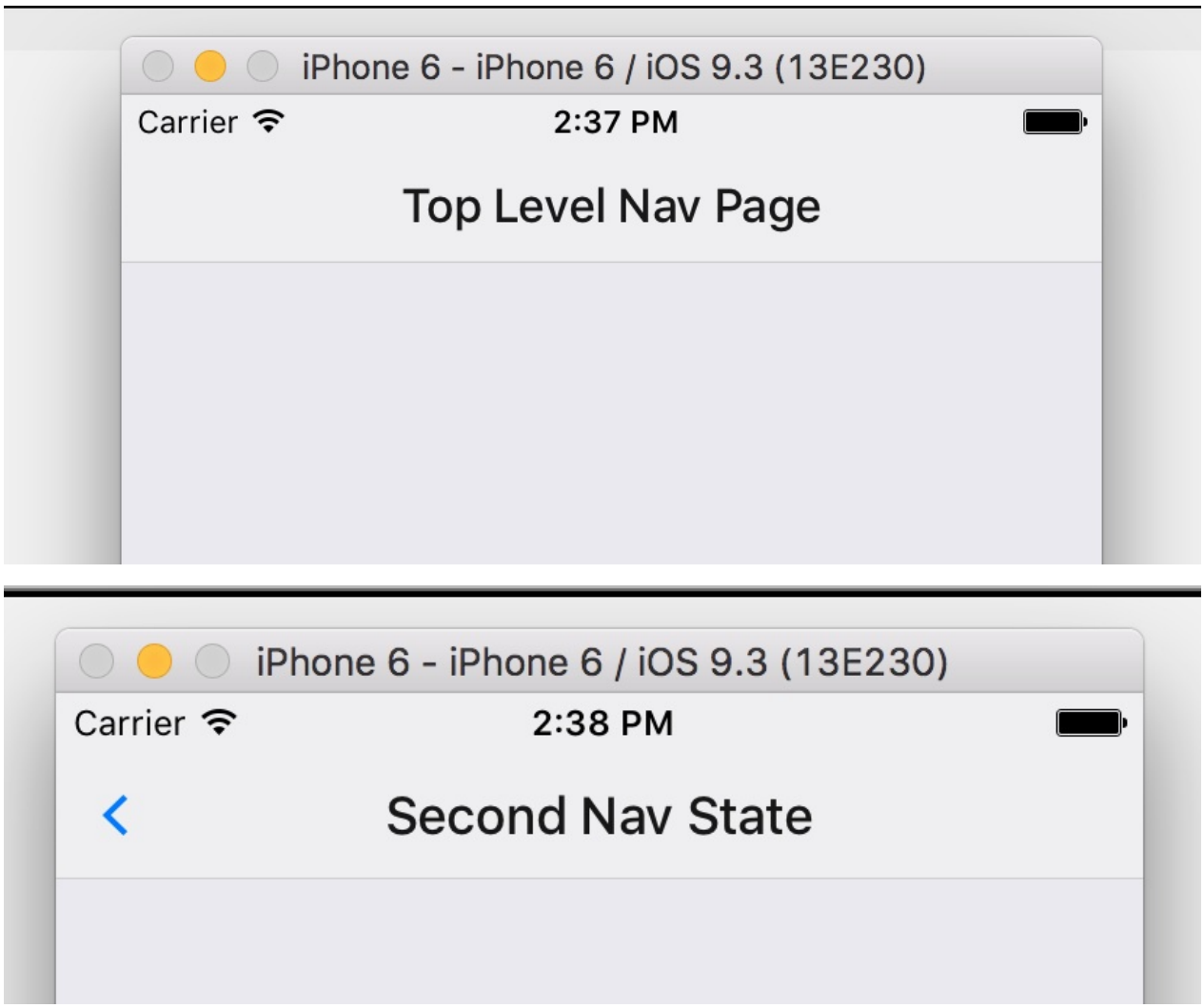
# NavigationExperimental.AnimatedView

This component wraps the rest of your navigation, and provides simple sliding animations for your navigation "stacks". It requires a few different props:

- `navigationState` -> This is where you can pass in your application's current navigation state. When using redux, you can get this via the `mapStateToProps` and `connect` functions.
- `onNavigate` -> This is a function that is used as a reducer, and it is responsible for "dispatching" actions to update your navigation's state.
- `renderOverlay` -> This prop requires a function that returns an `NavigationExperimental.Header`. It will pass `props` into the function to be used when creating the JSX for the navigation header.
- `renderScene` -> Just like `renderOverlay`, this prop takes a function that will render a `NavigationExperimental.Card`. The `props` are also passed into this function.

# NavigationExperimental.Header

This component will handle displaying information about the current state, as well as providing a simple "back" button to return to previous states in a header at the top of your app.




This component takes a prop called `renderTitleComponent`. This prop takes a function to setup an individual states title. It should return a `NavigationExperimental.Header.Title` component with the text you wish to display.

# NavigationExperimental.Card

This component is where your app will render it's content. Under the hood, this is rendering an `Animated.View` component with our content.

This component takes a `renderScene` prop. This prop takes a function that's purpose is to decide which container component should be rendered based on the navigation's state.

# NavigationExperimental.StateUtils

The navigations `StateUtils` object is a collection of helper functions for managing your navigations state. This is useful if you are using the `NavigationRootContainer` and your state is being managed there. For Redux apps, it's not necessary however.

# NavigationExperimental.CardStack

This is another object of helper functions for managing the "stack" of cards in your navigation history. It comes with methods like `pop` and `push` for adding or removing cards from your navigation state's stack. Again, this is useful if you make use of the `NavigationRootContainer` component to manage your navigation state.

# Setting up Our Navigation

Our app is going to use `NavigationExperimental` and setup the navigation state to be handled by Redux. To start, we'll configure some helpers to make the navigation component easier to reason about. Then we'll set up the navigator itself and hook it up with our Redux store. Let's get started.

Create a file called `navigator-helpers.js` in `src/containers/Navigator/` and import the following modules:

```
import {
  Animated,
  NavigationExperimental,
  Easing,
} from 'react-native';
import clrs from '../../utils/clrs';
import React, { PropTypes } from 'react';
import * as actions from '../../actions';

const { Header: NavigationHeader } = NavigationExperimental;
```

# State and Dispatch

Let's export some functions for our navigator component to make use of. To start, we'll configure the functions we'll use to setup our components props and actions with `connect`

`src/containers/Navigator/navigator-helpers.js`

```
export function mapStateToProps(state) {
  return {
    navigationState: state.navigator,
    pokemon: state.pokemon.get('all'),
    activePokemon: state.pokemon.get('active'),
  };
}

export function mapDispatchToProps(state) {
  return {
    goToPokemonDetail: artist => {
      dispatch(actions.goToPokemonDetail(artist));
    },
    gotoPokedex: () => dispatch(actions.gotoPokedex()),
    onNavigate: payload => dispatch(actions.onNavigate(payload)),
    catchEmAll: () => dispatch(actions.catchEmAll()),
    iChooseYou: url => dispatch(actions.iChooseYou(url)),
    clearChoice: () => dispatch(actions.clearChoice()),
  };
}
```

# Animations

We'll also create a helper function for handling animation in our navigator.

```
export function applyAnimation(pos, navState) {
  Animated.timing(pos, {
    toValue: navState.index,
    duration: 500,
    easing: Easing.bezier(0.36, 0.66, 0.04, 1),
  }).start();
}
```

# Title and Header

Finally, we'll create some helper functions to render out our navigations header, and the title for the current state.

```
export function renderHeader(props) {
  return (
    <NavigationHeader
      style={{backgroundColor: clrs.aqua }}
      {...props}
      renderTitleComponent={renderTitle} />
  );
}
```

```
export function renderTitle({ scene }) {
  return (
    <NavigationHeader.Title
      textStyle={{ color: clrs.blue, fontWeight: '700', letterSpacing: 1 }}>
      {scene.navigationState.title.toUpperCase()}
    </NavigationHeader.Title>
  );
}

renderTitle.propTypes = {
  scene: PropTypes.object,
};
```

That does it for the helpers file! Next we'll setup the actual navigator component.

# Our Navigator component

Our application is going to use the navigator and its state to decide which one our of container views to display. We'll use our navigator in the root view of our application and let it handle the logic for switching our views as needed.

Let's start by creating a file `index.js` in `src/containers/Navigator/` and setup our imports:

```
import {Map} from 'immutable';
import {NavigationExperimental, View, StatusBar} from 'react-native';
import React, {Component, PropTypes} from 'react';
import {connect} from 'react-redux';
import Pokedex from '../Pokedex';
import PokemonDetails from '../PokemonDetails';
import {ROUTE_IDS} from '../../constants';
import * as helpers from './navigator-helpers';

const {
  AnimatedView: NavigationAnimatedView,
  Card: NavigationCard,
  Header: NavigationHeader,
} = NavigationExperimental;
```

# The Navigator Container Class

```
class Navigator extends Component {

  componentWillMount() {
    this.props.catchEmAll();
  }

  _getActiveScene = (navigationState) => {
    switch(navigationState.key) {
      case ROUTE_IDS.POKEMON_DETAIL:
        return <PokemonDetails url={navigationState.url} />;
      default:
        return <Pokedex />;
    }
  }

  _renderCard = (props) => (
    <NavigationCard
      {...props}
      key={props.scene.navigationState.key}
      renderScene={this._renderScene} />
  )

  _renderScene = ({ scene: { navigationState } }) => {
    const activeScene = this._getActiveScene(navigationState);

    return (
      <View style={{ flex: 1, marginTop: NavigationHeader.HEIGHT }}>
        <StatusBar barStyle="default" />
        { activeScene }
      </View>
    );
  }

  render() {
    const { navigationState, onNavigate } = this.props;

    return (
      <NavigationAnimatedView
        navigationState={navigationState}
        style={{ flex: 1 }}
        onNavigate={onNavigate}
        renderOverlay={helpers.renderHeader}
        applyAnimation={helpers.applyAnimation}
        renderScene={this._renderCard}
      />
    );
  }
}
```

There is a lot going on here, so let's take a minute to go through this one step at a time.

We start by calling the `catchEmAll` action in our `componentWillMount` method. This is just to get our app setup to display our lists of pokemon in the views we'll be creating later. `_getActiveScene` is a method we'll pass to the `NavigationAnimatedView` in order to determine which container component to render based off the current navigation state.

After that, we're returning a `NavigationCard` component in our `_renderCard` method. This is used by the `NavigationAnimatedView` for rendering a single card in our navigation state's card stack. The `_renderScene` method is in charge of setting up the view around our active scene. In this case we're just creating a simple view that accommodates for the height of our navigation header, and sets the status bar style to the default look and feel (black text with light background).

Finally, we are rendering out our `NavigationAnimatedView` passing it our navigations state, a simple style declaration to cause the navigation to fill whatever space it can, and then we are passing it our appropriate helper functions for rendering our view.

Now let's tell react about our components props and export the connected component!

```
Navigator.propTypes = {
  navigationState: PropTypes.object,
  gotoPokedex: PropTypes.func,
  onNavigate: PropTypes.func,
  catchEmAll: PropTypes.func,
  iChooseYou: PropTypes.func,
  clearChoice: PropTypes.func,
  activePokemon: PropTypes.instanceOf(Map),
};

export default connect(
  helpers.mapStateToProps,
  helpers.mapDispatchToProps,
)(Navigator);
```

# Hook Our Navigator Up

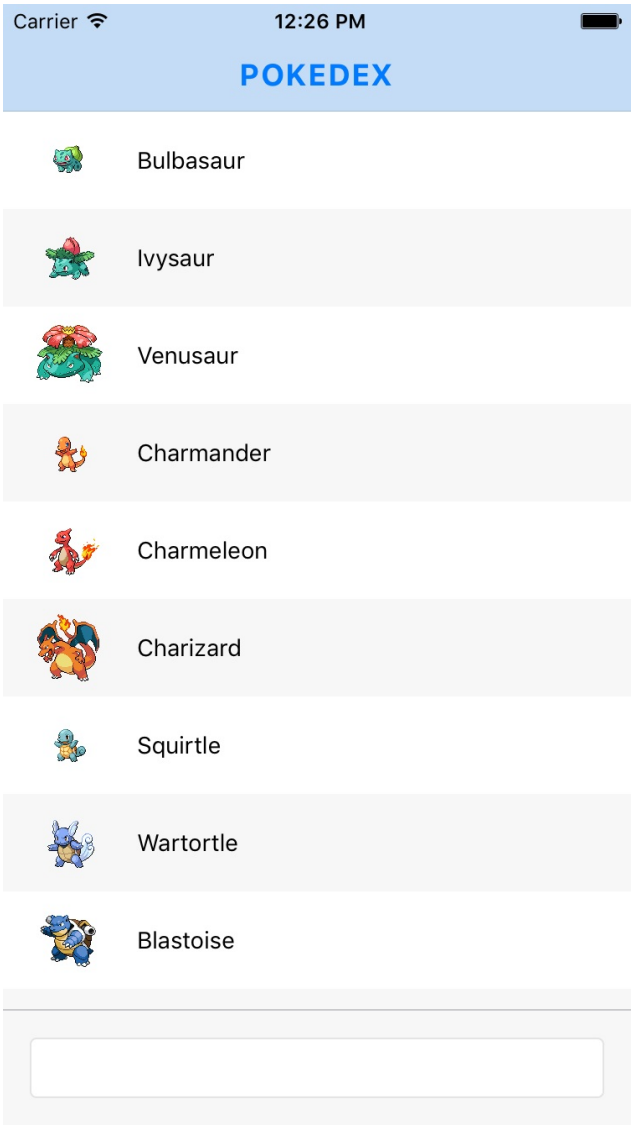
Our final step is to import our newly created Navigator component and plug it into our root container. Edit the `src/containers/Root.js` file to reflect the following:

```
import React from 'react';
import {Provider} from 'react-redux';
import createStore from '../store/configureStore';
import Navigator from './Navigator';

const store = createStore();

const Root = () => {
  return (
    <Provider store={store}>
      <Navigator />
    </Provider>
  );
};

export default Root;
```

# Building the Main Listing

Now that we have our navigator in place, we can start building out our components. First on the menu will be the Pokedex component defined as the default in our Navigator, and we'll define Pokedex as a container component at `/src/containers/Pokedex.js`.

We'll chiefly be using a **ListView**, a React Native core component that extends a more primitive **ScrollView**, and primarily allows us to efficiently display long lists of data.



# ListView

Some implementation-specific details of **ListView**: A minimal implementation requires you to create a ListView.DataSource, populate it with a simple array of data blobs, and instantiate your ListView component with said DataSource. It also requires you to define a `renderRow` callback, which will take individual blobs from your DataSource array, and return them as renderable components.

*Note: We are using Immutables for app state management (i.e.* `pokemon.get('all')` *and* `pokemon.set('all', payload)` *) and for creating the* `ListView.DataSource`

```
class Pokedex extends Component {
  constructor(props) {
    super(props);

    //Define a ListView.DataSource. DataSource requires you to define a rowHasChanged,
 comparator, and we'll use Immutable.is here for that.
    const dataSource = new ListView.DataSource({
      rowHasChanged: (r1, r2) => !Immutable.is(r1, r2),
    });

    //Don't forget to add your dataSource to the state.
    this.state = { pokemon: dataSource };
  }
  componentWillReceiveProps({ pokemon }) {
    this.setState({
      pokemon: this.state.pokemon.cloneWithRows(pokemon.toArray()),
    });
  }
  render() {
    const { pokemon } = this.state;
    const { goToPokemonDetail, ready } = this.props;

    return (
        <View style={ styles.container }>

          <ListView dataSource={ pokemon }
            style={ styles.listView }
            renderRow={ (...args) => renderRow(goToPokemonDetail, ...args) }
            enableEmptySections />

          <SearchBar onChange={ filter } value={ query } />

          <KeyboardSpacer />

        </View>
    );
  }
}
```

Complete the Pokedex container component. We'll also do some housekeeping. Don't forget to:

- Define your styles
- Connect to the Redux store
- Map your state/dispatchers to props
- We'll define `renderRow` in the next section

# ListView: Render Row

## Finally, let's define how our rows get rendered in our ListView.

We've already passed `renderRow` into our ListView component above, and now we need to define it in Pokedex. We'll pass all the information we need to a MediaObject subcomponent that we'll create in a moment.

Note: You'll also need to define your `goToPokemonDetail` action if you haven't already.

```
function renderRow(goTo, pokemon, sId, id) {
  const POKEMON_STATE = {
    title: pokemon.get('name'),
    url: pokemon.get('url'),
  };

  // Some quick parsing to get a bit of extra information for ourselves
  const re = /^.*pokemon\/(.+)\/$/;
  const matches = re.exec(pokemon.get('url'));
  const pokemonID = matches ? matches[1] : null;
  const imageUrl = (
    pokemonID ? `http://pokeapi.co/media/sprites/pokemon/${ pokemonID }.png` :
                null
  );
  return (
    <MediaObject index={ id }
      text={ pokemon.get('name') }
      imageUrl={ imageUrl }
      action={ () => goTo(POKEMON_STATE) } />
  );
}
```

Now that we have the information we need, we'll render it out in our MediaObject component, which we can place at `/src/components/MediaObject.js`. We'll use a couple of new pieces:

- **TouchableOpacity** is a wrapper used for making views respond properly to touches. You can think of it as onClick, but with a visual feedback mechanism added in. Here, we'll use the onPress mechanism to react to touches and redirect the user to the desired pokemon details page.

- **Image** is a core React Native component for loading local or remote images with one neat trick: It's non-blocking. That means we can feed our image into our layout without too much concern for UI stutters or hiccups.

```
//Define a placeholder image to be used in the case of a non-existent external resource

const placeholder = require('../assets/who.jpg');

const MediaObject = ({ index, action, text, imageUrl }) => {
  const image = (
    imageUrl ? { uri: imageUrl } : placeholder
  );
  return (
    <TouchableOpacity
      underlayColor={ clrs.gray }
      onPress={ action }>

      <View style={[ styles.mediaObject, bgColor(Number(index) + 1) ]}>
        <Image source={ image } style={ styles.image } />
        <Text style={ styles.text }>
          { text.charAt(0).toUpperCase() + text.slice(1) }
        </Text>
      </View>

    </TouchableOpacity>
  );
};
```

Again, don't forget to do housekeeping like defining your styles and properly exporting your component as a module here. We've also used a bgColor function above that you can define:

```
function bgColor(idx) {
  return {
    backgroundColor: idx % 2 !== 0 ? clrs.white : clrs.lighterGray,
  };
}
```

That's it! Everything's in place. Just one more thing left to do.

# Text Input

This one's easy.

Add a new custom `SearchBar` component to your Pokedex container. To complete it, you'll need to use a native **TextInput**, which works similarly to the web input you should be familiar with, tracking a value and responding to `onChangeText` events:

```
<TextInput style={ styles.input }
        value={ query }
        onChangeText={ text => onChange(text) } />
```

Also note the use of `onChangeText` which provides the text value as opposed to `onChange` which provides an event.

# Keyboard Spacer

The virtual keyboards used on iOS devices pose unique layout challenges as they require you to push content around and reshape your layouts as the keyboard is activated or deactivated.

For this, we'll use Andrew Hurst's excellent react-native-keyboard-spacer. We'll need to start by importing the `Platform` utility from React Native.

```
// ... src/containers/pokedex.js

import {
  StyleSheet,
  View,
  ListView,
  Platform,
} from 'react-native';

const Spacer = Platform.OS === 'ios' ? <KeyBoardSpacer /> : null;

/* Add our new Spacer just below our SearchBar Component */
<SearchBar onChange={ filter } value={ query } />
{Spacer}
```

# Using Selectors to Filter Data

We're going to want to filter our Pokemon listing data using our search component. We could do this using our actions & reducers, however it is a better idea to implement something called a `Selector`.

To do this, we can make use of the library `reselect`. To install it, just run:

```
npm install --save reselect
```

# What is Reselect?

Reselect is a library built to help us construct "memoized" selectors for efficiently sorting through datasets. "Memoization" is an optimization process that caches the results of an expensive operation, returning that cached value each time the function is called unless given different inputs.

Reselect allows us to easily create selectors that integrate perfectly with our redux store. Here is an example below for creating a selector:

```
import {createSelector} from 'reselect';

const getTodoList = state => state.todos;
const getFilterType = state => state.filterType;

const filterTodoList = createSelector(
  [getTodoList, filterType],
  (todos, filter) => {
    switch (filter) {
      case 'DONE':
        return todos.filter(t => t.status === 'complete');
      case 'INCOMPLETE':
        return todos.filter(t => t.status === 'incomplete');
      default:
        return todos;
    }
  },
);

export default filterTodoList;
```

You could then use this selector inside of a container component's `mapStateToProps` function like so:

```
import filterTodoList from '../selectors';

const mapStateToProps = state => {
  return {
    todos: filterTodoList(state)
  };
}
```

That's all there is to it. Now let's implement a selector of our own to use in our Pokedex!

React Native provides a rich animation API. You can create animations using two systems:

- `LayoutAnimation` allows you to animate global layout transitions.
- `Animated` gives you a more fine-grained control over animating specific values.

# LayoutAnimation

`LayoutAnimation` allows you to define mounting and update animations. They animate every property that changes in a component – usually by calling `setState`.

This is good option for when you want to apply the same animation for all properties or you don't know the specific values you are animating between. These are native animations and are mostly not affected by what is happening in JavaScript world during their execution.

## Example


Run the example: rnplay.org/apps/xdHpQA

```
class App extends Component {
  constructor(props) {
    super(props);
    this.state = { s: 100 };
  }

  _onPress = () => {
    LayoutAnimation.spring();
    this.setState({ s: this.state.s + 15 });
  }

  render() {
    return (
      <View style={ styles.container }>
        <View style={[
          styles.box,
          { width: this.state.s, height: this.state.s }
        ]} />

        <TouchableOpacity onPress={ this._onPress }>
          <View style={ styles.button }>
            <Text style={ styles.buttonText }>Press me!</Text>
          </View>
        </TouchableOpacity>
      </View>
    );
  }
}

var styles = StyleSheet.create({
  container: {
    flex: 1,
    justifyContent: 'center',
    alignItems: 'center',
  },
  box: {
    backgroundColor: 'red',
  },
  button: {
    marginTop: 10,
    paddingVertical: 10,
    paddingHorizontal: 20,
    backgroundColor: 'black',
  },
  buttonText: {
    color: 'white',
    fontSize: 16,
    fontWeight: 'bold',
  },
});
```

# Animated

The `Animated` library allows you to animate specific values of targeted components.

Unlike `LayoutAnimation` these animations are executed on the JavaScript side and rely on `requestAnimationFrame` and `setNativeProps`. This can sometimes lead to stutter.

```
import { Animated } from 'react-native';
```

Creating an animation with the Animted API is generally a three step process and might seem familiar if you've ever used react-motion:

- Create a new animated value
- Trigger the animation/tween
- Connect the animated value to a style property of a component

# Animated.Value

The first step in creating an animation is to create an animated value. We can do that by calling:

```
import { Animated } from 'react-native';

class App extends Component {
  constructor(props) {
    super(props);
    this.state = {
      animatedVal: new Animated.Value(0),
    };
  }

  ...
}
```

You'll want to save this animated value onto the state of a component or as a property. This is because we will need access to it in the `render()` method.

# Animated.ValueXY

It is similar to `Animated.Value` except it supports an `{ x, y }` object as the value. This is useful for dealing with positions of elements and gestures.

```
new Animated.ValueXY({ x: 0, y: 0 })
```

# setValue

There are times when you might want to change the value for this `Animated.Value` but, not trigger an animation. This can be done by using the `setValue` method.

```
this.state.animatedVal.setValue(100)
```

# Timing, Spring & Delay

The three basic options to define the animation of an `Animated.Value` are:

## Animated.timing

Used to define an animation that takes a specific amount of time to execute.

```
import { Animated, Easing } from 'react-native';
this.state = { animatedVal: new Animated.Value(0) };

Animated.timing(this.state.animatedVal, {
  toValue: 100,
  duration: 500,
  easing: Easing.inOut(Easing.ease),
  delay: 200,
}).start(() => console.log('animation complete'));
```

## Animated.spring

Used to define an animation as a spring rather than a specific timing. The spring uses the same physics as Origami.

```
import { Animated, Easing } from 'react-native';
this.state = { animatedVal: new Animated.Value(0) };

Animated.spring(this.state.animatedVal, {
  toValue: 100,
  friction: 7,
  tension: 40,
}).start(() => console.log('animation complete'));
```

## Animated.decay

Used to define a deceleration style transition for something that is already moving.

```
import { Animated, Easing } from 'react-native';
this.state = { animatedVal: new Animated.Value(0) };

Animated.decay(this.state.animatedVal, {
  velocity: { // velocity from a gesture
    x: gestureState.vx,
    y: gestureState.vy,
  },
  deceleration: 0.997,
}).start(() => console.log('animation complete'));
```

# Animated View

The Animated library ships with 3 views that support `Animated.Values`. These animated components allow us to bind `Animated.Values` to the style properties.

- `Animated.View`
- `Animated.Text`
- `Animated.Image`.

You can also make a custom animated component by using `Animated.createAnimatedComponent`.

# Example

Run the example: rnplay.org/apps/moq61w

```
class App extends Component {
  constructor(props) {
    super(props);
    this.state = {
      animatedVal: new Animated.Value(100),
    };
  }

  componentDidMount() {
    Animated.timing(this.state.animatedVal, {
      toValue: 200,
      duration: 3000,
      easing: Easing.inOut(Easing.ease),
    }).start();
  }

  render() {
    return (
      <View style={ styles.container }>
        <Animated.View style={[styles.box, {
          width: this.state.animatedVal,
          height: this.state.animatedVal,
        }]} />
      </View>
    );
  }
}

var styles = StyleSheet.create({
  container: {
    flex: 1,
    justifyContent: 'center',
    alignItems: 'center',
  },
  box: {
    backgroundColor: 'red',
  },
});
```

# More Animated!

The Animated library allows us to do a lot more things such as composing animations or driving animations through gestures and input events. Let's look at a few examples.

## Interpolation

Using the same `Animated.Value` you can drive multiple animations through interpolation. Interpolation can also be used to convert numbers into `rgb` colours or rotation values in `degrees`.

```
this.state = { animatedVal: new Animated.Value(0) };

const marginLeftAnimation = this.state.animatedValue.interpolate({
  inputRange: [-300, -100, 0, 100, 101],
  outputRange: [300,    0, 1,   0,   0],
});
```

| Input | Output |
|---|---|
| -400 | 450 |
| -300 | 300 |
| -200 | 150 |
| -100 | 0 |
| -50 | 0.5 |
| 0 | 1 |
| 50 | 0.5 |
| 100 | 0 |
| 101 | 0 |
| 200 | 0 |

## Composing Animations

```
Animated.sequence([
  Animated.timing(this.state.animatedVal, {
    toValue: 100,
    duration: 500,
  }),
  Animated.timing(this.state.animatedVal, {
    toValue: 0,
    duration: 200,
  })
]).start();

Animated.parallel([
  Animated.spring(this.state.animPosition, {
    toValue: { x: 200, y: 100 }
  }),
  Animated.timing(this.state.animOpacity, {
    toValue: 0.75,
  }),
]).start();

Animated.stagger(100, [
  Animated.timing(this.state.animListItem1, {
    toValue: 100,
    duration: 200,
  }),
  Animated.timing(this.state.animListItem2, {
    toValue: 0,
    duration: 200,
  })
]).start()
```

# Animated.Event

Allows us to extract values from an input event and set values for an `Animated.Value`.

```
class AnimatedEventExample extends Component {
  constructor(props) {
    super(props);
    this.state = {
      animatedY: new Animated.Value(0),
    };
  }

  render() {
    const event = Animated.event([{
      nativeEvent: {
        contentOffset: {
          y: this.state.animatedY,
        },
      },
    }]);

    return (
      <ScrollView onScroll={ event }>
        <Animated.View
          style={{ height: this.state.animatedY }} />
      </ScrollView>
    )
  }
}
```

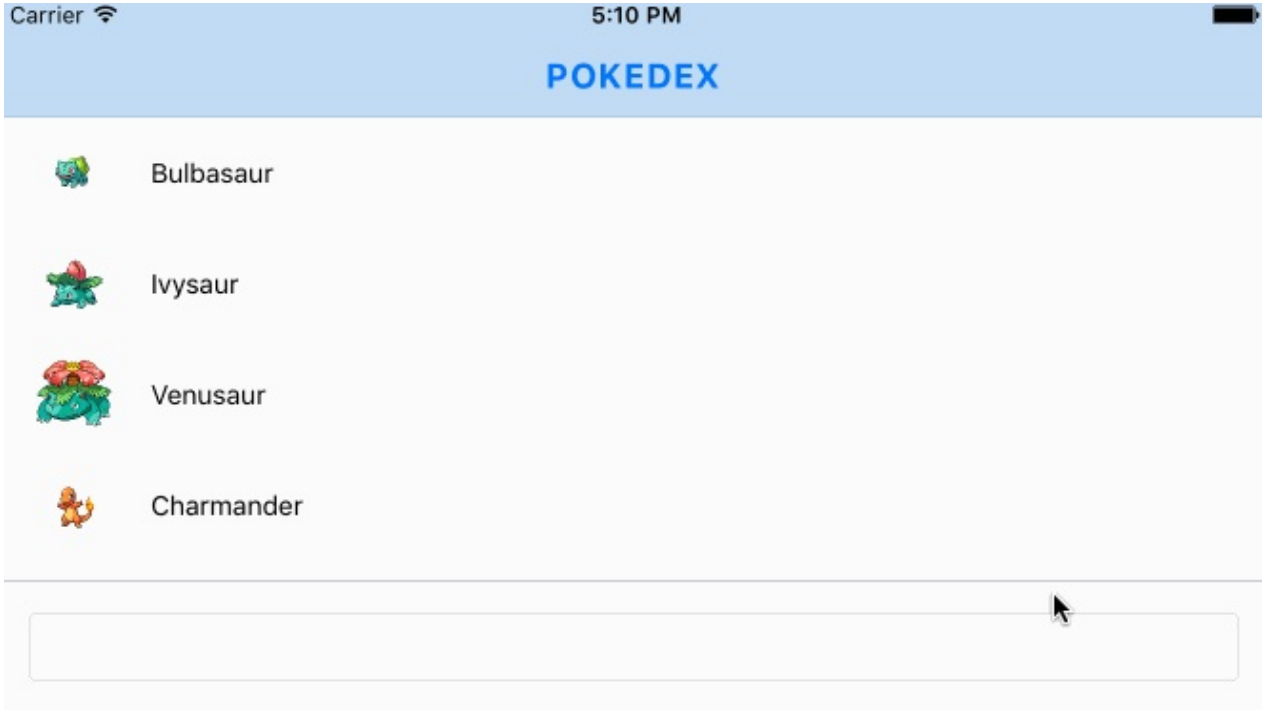
# Resources

- React Native Animation Book byJason Brown
- Offical Animations Guide
- AnimatedGratuitousApp Example
- Animated Documentation
- LayoutAnimation Documentation

# Exercise

Build a `Loader` component that accepts one property: `show`. If `show` is set to `true` then it displays a pulsing pikachu otherwise it displays whatever content is passed in as `children`.

Modify the `Pokedex` & `PokemonDetails` components to use the `Loader`.




```
<Loader show={ !pokemon.get('name') }>
  <ScrollView style={ styles.container }>
    <Sprites sprites={ pokemon.get('sprites') } />
    <BasicInfo { ...basicInfo } />
    <Header>STATS</Header>
    <Stats stats={ pokemon.get('stats') } />
    <Header>MOVES</Header>
    <Moves moves={ pokemon.get('moves') } />
  </ScrollView>
</Loader>
```

# Setting up our Pokemon Detail View

We can now start setting up the detail view for individual pokemon selected in from the list. In order to do this, we'll have to make use of a new React Native component. The `ScrollView` component!

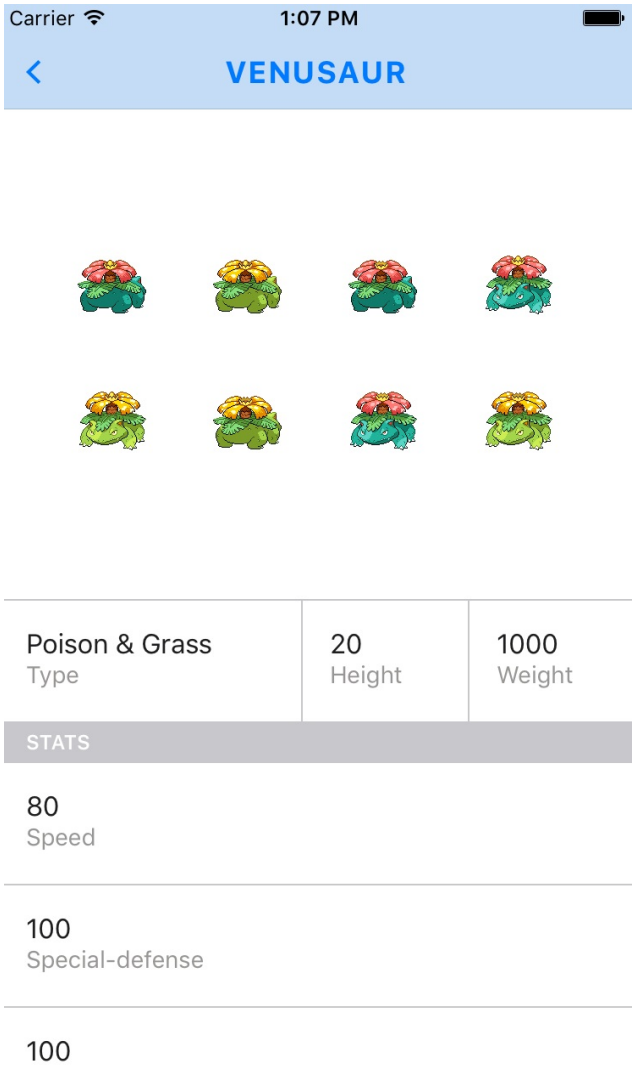
A scroll view is similar to the `ListView` component we have used already, but it has a few differences. To start, the `ListView` component is used when you have long lists of changing data. It has optimizations that make it more efficient for this type of data. The `ListView` also brings the ability to separate your content into sections, as well as displaying section headers. The `ScrollView` on the other hand, is just for displaying content of various heights inside a fixed height container. These components are better to use if you have data that isn't likely to change much.

# ScrollView API

An important prop to be aware of for the `ScrollView` is the `contentContainerStyle` prop. This prop will pass your styles over to the container element that will wrap your `ScrollView` 's child elements. This is a good place to pass styles such as `justifyContent` or `alignItems` .

You can find out more about the ScrollView's props here.

Using the `ScrollView` component, let's setup the display for our Pokemon's abilities and stats!


Carrier
1:07 PM
VENUSAUR
Poison & Grass
Type
20
Height
1000
Weight
STATS
80
Speed
100
Special-defense
100
Special-attack

# Working with MapView and Third-party Components

While you won't be using it in this project, the MapView is a good example of a common element that you'll use in a lot of other projects, and is worth taking some time with.

At its most basic implementation, a MapView will render an interactive map in a defined area.

```
<MapView
  region={{
    latitude: 37.78825,
    longitude: -122.4324,
    latitudeDelta: 0.0922,
    longitudeDelta: 0.0421,
  }}
/>
```

Map features like markers and callouts are achievable through the use of an annotations parameter:

```
<MapView
  style={styles.map}
  region={{
    latitude: 37.78825,
    longitude: -122.4324,
    latitudeDelta: 0.0922,
    longitudeDelta: 0.0421,
  }}
  annotations={[{
    latitude: 37.788,
    longitude: -122.43,
    title: 'Sunset Cafe',
    subtitle: '4230 Sunset Blvd'
  }]}
/>
```

However, it's worth noting that the native MapView shipping in React Native is an iOS-only implementation, and will fail for Android builds. The official recommendation to work around this is to use a react-native-maps by Leland Richardson. You can add it to your project with `npm install react-native-maps --save` and then using `rnpm` to link the native components with `rnpm link`. Usage roughly mirrors the standard MapView component:

```
import MapView from 'react-native-maps';
```

```
<MapView
region={{
  latitude: 37.78825,
  longitude: -122.4324,
  latitudeDelta: 0.0922,
  longitudeDelta: 0.0421,
}}
/>
```

You can set region as a part of your state:

```
getInitialState() {
  return {
    region: {
      latitude: 37.78825,
      longitude: -122.4324,
      latitudeDelta: 0.0922,
      longitudeDelta: 0.0421,
    },
  };
}

onRegionChange(region) {
  this.setState({ region });
}

render() {
  return (
    <MapView
      region={this.state.region}
      onRegionChange={this.onRegionChange}
    />
  );
}
```

However, implementations of advanced features like markers are different from the default MapView, with `react-native-maps` choosing to opt for a nested component format:

```
<MapView
  region={this.state.region}
  onRegionChange={this.onRegionChange}
>
  {this.state.markers.map(marker => (
    <MapView.Marker
      coordinate={marker.latlng}
      title={marker.title}
      description={marker.description}
    />
  ))}
</MapView>
```

# Testing

Testing React Native components using Enzyme is similar to testing React Components for the web. We can use the shallow rendering API to isolate a component for testing. Additionally, `react-native-mock` allows us to run these tests on a CI server.

## Setup

To get started, we need to do a little bit of setup. Install the following dependencies.

- babel-core
- babel-eslint
- babel-plugin-transform-object-rest-spread
- babel-preset-es2015
- babel-preset-react
- chai
- enzyme
- mocha
- react-addons-test-utils
- react-dom
- react-native-mock
- sinon

Then, create a `.babelrc` file:

```
{
  "presets": [
    "es2015",
    "react"
  ],
  "plugins": [
    "transform-object-rest-spread"
  ]
}
```

And finally add this task to `package.json`:

```
"scripts": {
  ...
  "test": "mocha --require react-native-mock/mock --compilers js:babel-core/register t
est/**/*.spec.js --reporter nyan"
}
...
```

# Example

```
import React from 'react';
import {shallow} from 'enzyme';
import {expect} from 'chai';
import sinon from 'sinon';
import {View} from 'react-native';
import Immutable from 'immutable';

import Types from '../../src/components/Types';
import TypedItem from '../../src/components/TypedItem';
import BasicInfo from '../../src/components/BasicInfo';


describe('<BasicInfo />', () => {

  it('should render correct components', () => {
    const basicInfo = shallow(<BasicInfo />);
    expect(basicInfo.find(View)).to.have.length(1);
    expect(basicInfo.find(Types)).to.have.length(1);
    expect(basicInfo.find(TypedItem)).to.have.length(2);
  });

  it('should pass types, height, and weight into child components', () => {
    const testTypes = Immutable.List(['foo', 'bar']);
    const basicInfo = shallow(<BasicInfo types={testTypes} height={20} weight={20} />)
;
    const types = basicInfo.find(Types).props().types;
    const firstTypedItem = basicInfo.find(TypedItem).first().props().value;
    const secondTypedItem = basicInfo.find(TypedItem).last().props().value;
    expect(types).to.deep.equal(testTypes);
    expect(firstTypedItem).to.equal(20);
    expect(secondTypedItem).to.equal(20);
  });

  describe('styles', () => {

    it('should have default styles', () => {
      const basicInfo = shallow(<BasicInfo />);
      const containerStyles = basicInfo.find(View).props().style;
      const typedItemStyles1 = basicInfo.find(TypedItem).first().props().style;
      const typedItemStyles2 = basicInfo.find(TypedItem).last().props().style;
```

```
      const expectedTypedStyles = {
        flex: 1,
        borderLeftWidth: 1,
      };
      expect(containerStyles).to.deep.equal({
        flexDirection: 'row',
        flexWrap: 'wrap',
      });
      expect(typedItemStyles1).to.deep.equal(expectedTypedStyles);
      expect(typedItemStyles2).to.deep.equal(expectedTypedStyles);
    });

  });

});
```

# Gesture Responder System

> Gesture recognition on mobile devices is much more complicated than web. A touch can go through several phases as the app determines what the user's intention is. For example, the app needs to determine if the touch is scrolling, sliding on a widget, or tapping. This can even change during the duration of a touch. There can also be multiple simultaneous touches.

from react-native-docs/gesture-responder-system.html

# Responder Lifecycle

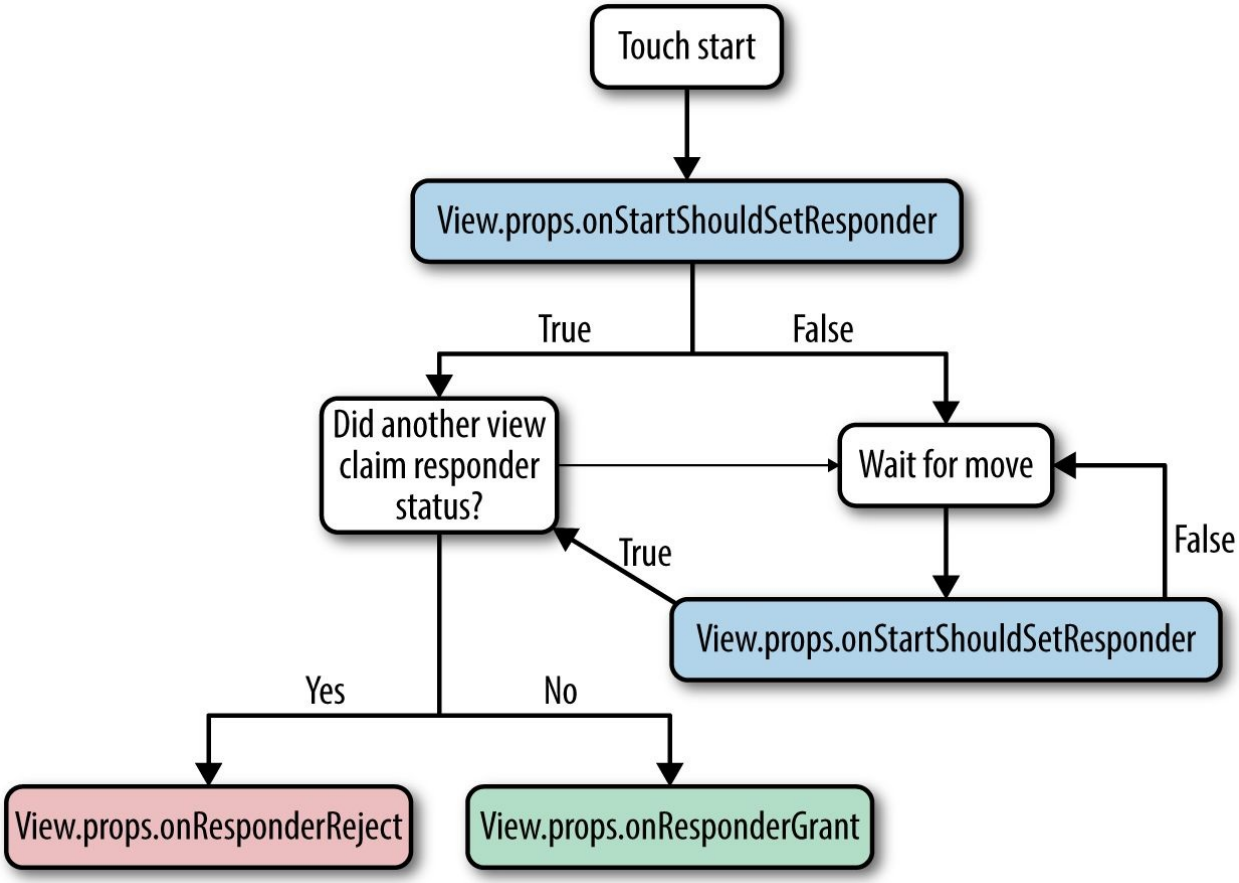



**Image from: Learning React Native: Building Native Mobile Apps with JavaScript by Bonnie Eisenman**

The gesture responder System allows views to negotiate responsibility for a touch event. A touch event has three phases: start, move and release. Let's break down its lifecycle.

## 1. Request

A view can request to become the touch responder in the:

- start phase by returning `true` from `onStartShouldSetResponder`
- move phase by returning `true` from `onMoveShouldSetResponder`

## 2. Bubbling

Similar to the web, these negotiation functions are called in a bubbling pattern. Therefore, the deepest component will become the responder.

## 3. Override

However, a parent can choose to override and claim responsibility. This is done by returning true from either `onStartShouldSetResponderCapture` or `onMoveShouldSetResponderCapture`.

## 4. Granted or Rejected

If a view's request is granted or rejected `onResponderGrant` or `onResponderReject` is invoked appropriately.

## 5. Respond

Finally the view can then respond using one of the following handlers:

- `onResponderMove`
- `onResponderRelease`
- `onResponderTerminationRequest`
- `onResponderTerminate`

After a view has successfully claimed touch responder status, its relevant event handlers may be called.

# PanResponder

PanResponder is a higher level abstraction that can be used to recognize simple touch gestures. It gives you access to both the raw `nativeEvent` and a `gestureState` object.

The `nativeEvent` provides you all the touches and their locations. Whereas, `gestureState` is a wrapper object that also tracks accumulated distance, velocity and the touch origin (coordinates where the touch was when the responder was granted access).

## Example

Run the example: rnplay.org/apps/Sxb2tQ

```
import React, { Component } from 'react';
import {
  AppRegistry,
  PanResponder,
  StyleSheet,
  View,
  processColor,
} from 'react-native';

var CIRCLE_SIZE = 80;


class PanResponderExample extends Component {
```

```
  constructor(props) {
      super(props);
    this._panResponder = {};
    this._previousLeft = 0;
    this._previousTop = 0;
    this._circleStyles = {};
    this.circle = null;
    }

  componentWillMount() {
    this._panResponder = PanResponder.create({
      onStartShouldSetPanResponder: this._handleStartShouldSetPanResponder,
      onMoveShouldSetPanResponder: this._handleMoveShouldSetPanResponder,
      onPanResponderGrant: this._handlePanResponderGrant,
      onPanResponderMove: this._handlePanResponderMove,
      onPanResponderRelease: this._handlePanResponderEnd,
      onPanResponderTerminate: this._handlePanResponderEnd,
    });
    this._previousLeft = 20;
    this._previousTop = 84;
    this._circleStyles = {
      style: {
        left: this._previousLeft,
        top: this._previousTop,
        backgroundColor: 'green',
      }
    };
  }

    componentDidMount() {
    this._updateNativeStyles();
  }

    render() {
    return (
      <View style={styles.container}>
        <View style={styles.circle}
          ref={(circle) => {
            this.circle = circle;
          }}
          { ...this._panResponder.panHandlers }
        />
      </View>
    );
  }

  _highlight = () => {
    this._circleStyles.style.backgroundColor = 'blue';
    this._updateNativeStyles();
  }

  _unHighlight = () => {
    this._circleStyles.style.backgroundColor = 'green';
```

```
    this._updateNativeStyles();
  }

  _updateNativeStyles() {
    this.circle && this.circle.setNativeProps(this._circleStyles);
  }

  _handleStartShouldSetPanResponder() {
    return true;
  }

  _handleMoveShouldSetPanResponder() {
    return true;
  }

  _handlePanResponderGrant = (e, gestureState) => {
    this._highlight();
  }

    _handlePanResponderMove = (e, gestureState) => {
    this._circleStyles.style.left = this._previousLeft + gestureState.dx;
    this._circleStyles.style.top = this._previousTop + gestureState.dy;
    this._updateNativeStyles();
  }

  _handlePanResponderEnd = (e, gestureState) => {
    this._unHighlight();
    this._previousLeft += gestureState.dx;
    this._previousTop += gestureState.dy;
  }

}

var styles = StyleSheet.create({
  circle: {
    width: CIRCLE_SIZE,
    height: CIRCLE_SIZE,
    borderRadius: CIRCLE_SIZE / 2,
    position: 'absolute',
    left: 0,
    top: 0,
  },
  container: {
    flex: 1,
    paddingTop: 64,
  },
});
```