互联网应用开发技术

Web Application Development

第16课
WEB移动端-REACT NATIVE

Episode Sixteen

React Native

陈昊鹏

chen-hp@sjtu.edu.cn

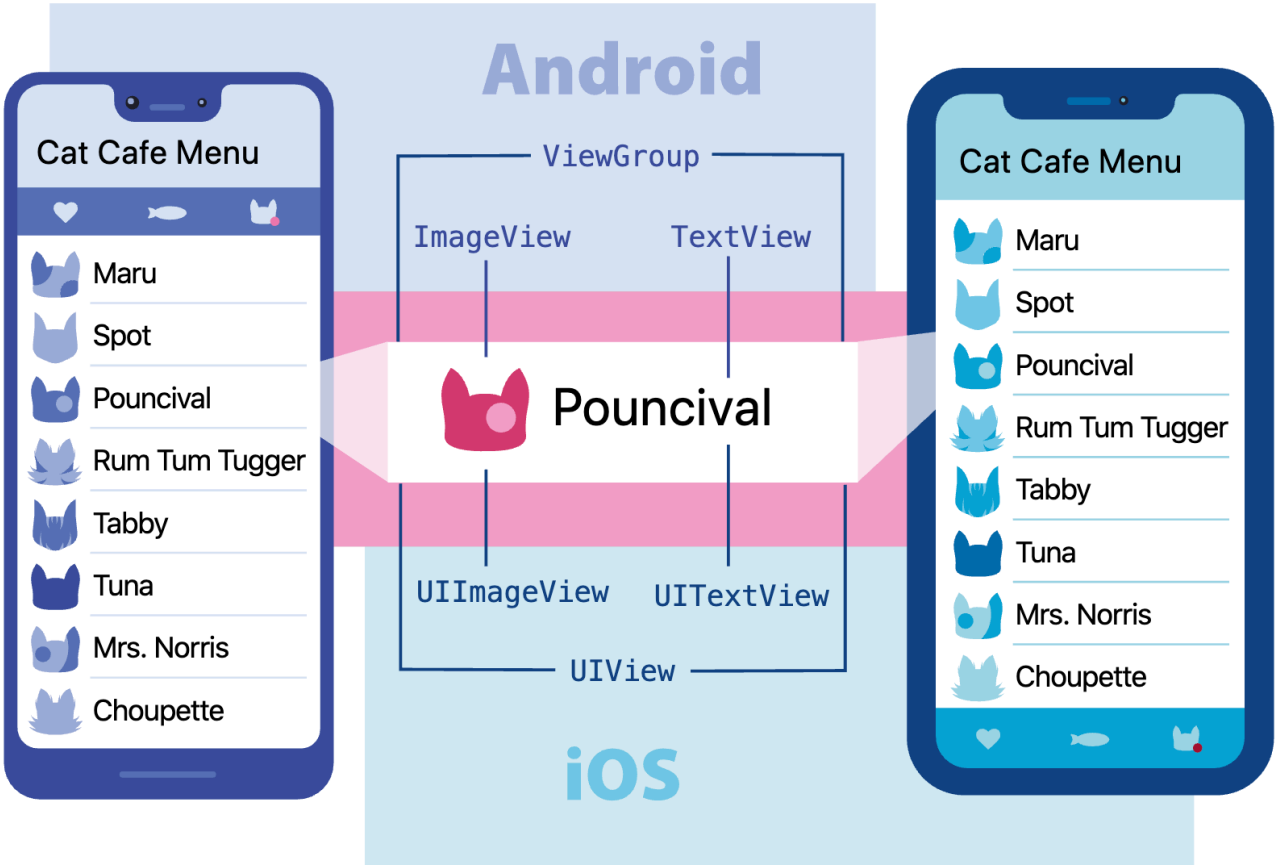
React Native

- React Native is an open source framework for building Android and iOS applications using React and the app platform's native capabilities.

- With React Native, you use JavaScript to access your platform's APIs as well as to describe the appearance and behavior of your UI using React components: bundles of reusable, nestable code.

Views and mobile development

- In Android and iOS development
  - a **view** is the basic building block of UI: a small rectangular element on the screen which can be used to display text, images, or respond to user input.



Just a sampling of the many views used in Android and iOS apps.

Native Components

- With React Native, you can invoke these views with JavaScript using React components.
  - In Android development, you write views in Kotlin or Java;
  - In iOS development, you use Swift or Objective-C.

- At runtime, React Native creates the corresponding Android and iOS views for those components.
  - Because React Native components are backed by the same views as Android and iOS, React Native apps look, feel, and perform like any other apps.
  - We call these platform-backed components **Native Components**.

- React Native also includes a set of essential, ready-to-use Native Components you can use to start building your app today.
  - These are React Native's **Core Components**.

Core Components

- You will mostly work with the following Core Components

REACT NATIVE UI COMPONENT	ANDROID VIEW	IOS VIEW	WEB ANALOG	DESCRIPTION
`<View>`	`<ViewGroup>`	`<UIView>`	A non-scrollling `<div>`	A container that supports layout with flexbox, style, some touch handling, and accessibility controls
`<Text>`	`<TextView>`	`<UITextView>`	`<p>`	Displays, styles, and nests strings of text and even handles touch events
`<Image>`	`<ImageView>`	`<UIImageView>`	`<img>`	Displays different types of images
`<ScrollView>`	`<ScrollView>`	`<UIScrollView>`	`<div>`	A generic scrolling container that can contain multiple components and views
`<TextInput>`	`<EditText>`	`<UITextField>`	`<input type="text">`	Allows the user to enter text

Core Components

Hello World

Expo

```
import React from 'react';
import { View, Text, Image, ScrollView, TextInput } from
'react-native';

export default function App() {
  return (
    <ScrollView>
      <Text>Some text</Text>
      <View>
        <Text>Some more text</Text>
        <Image
source="https://reactnative.dev/docs/assets/p_cat2.png"
style={{width: 200, height: 200}}/>
      </View>
      <TextInput
        style={{
          height: 40,
          borderColor: 'gray',
          borderWidth: 1
        }}
        defaultValue="You can type in me"
      />
    </ScrollView>
  );
}
```

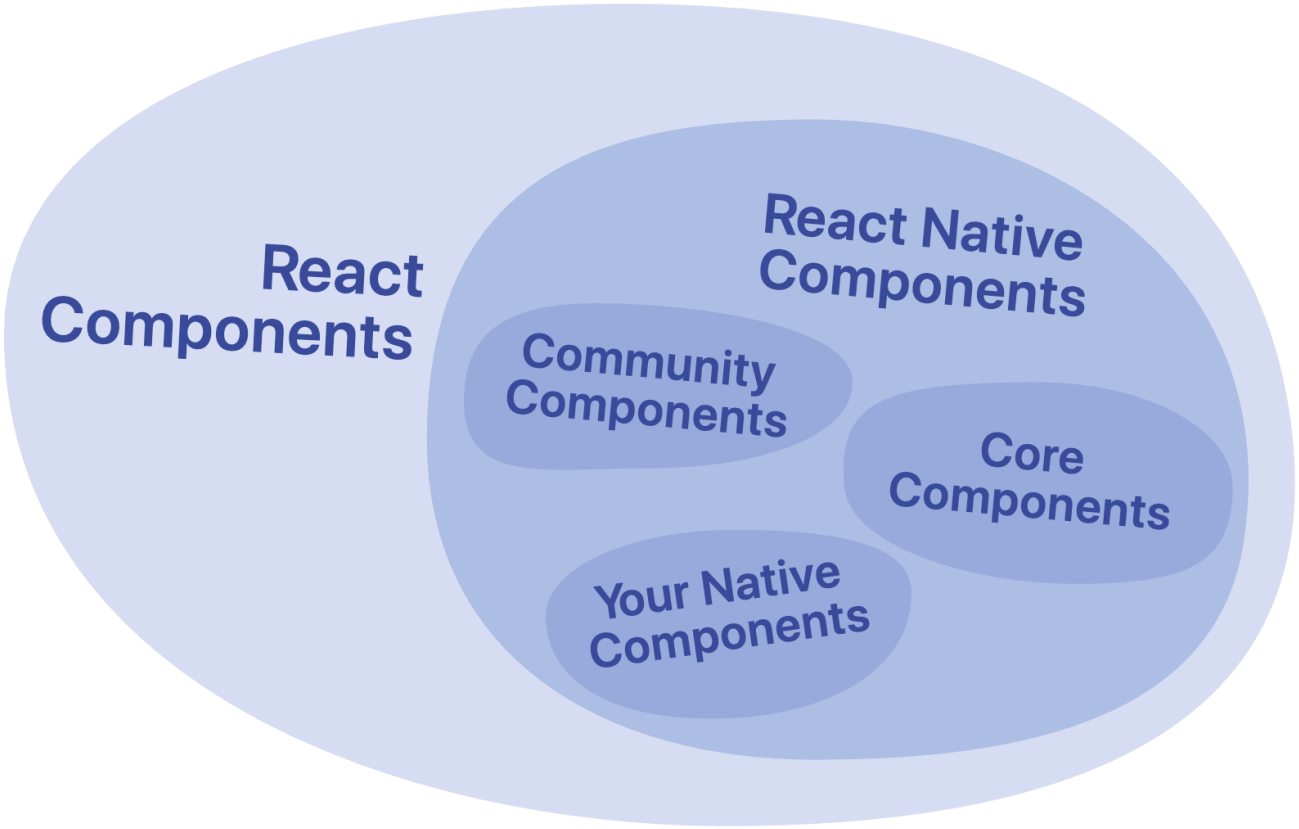
Some text
Some more text

You can type in me

Preview iOS Android Web

Core Components

- React Native uses the same API structure as React components



React Fundamentals

- React Native runs on React, a popular open source library for building user interfaces with JavaScript.

- We're going to cover the core concepts behind React:
  - components
  - JSX
  - props
  - state

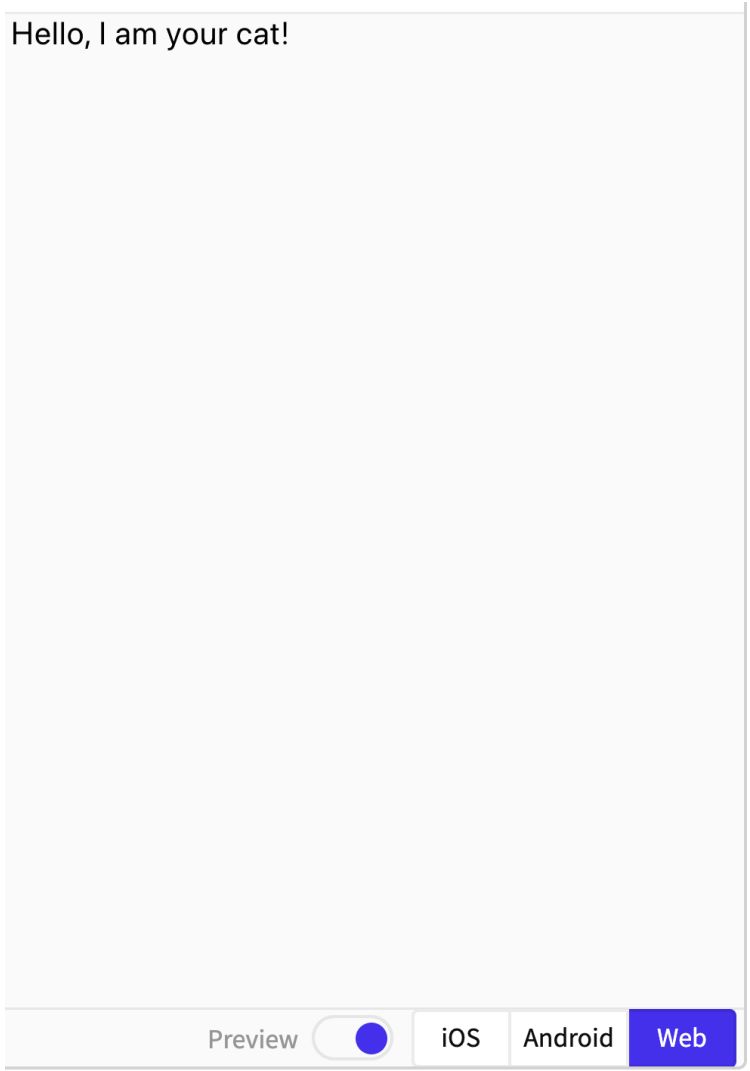
Component

- Funtion Component

```
import React from 'react';
import { Text } from 'react-native';

export default function Cat() {
  return (
    <Text>Hello, I am your cat!</Text>
  );
}
```

- Class Component

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

export default class Cat extends Component {
  render() {
    return (
      <Text>Hello, I am your cat!</Text>
    );
  }
}
```



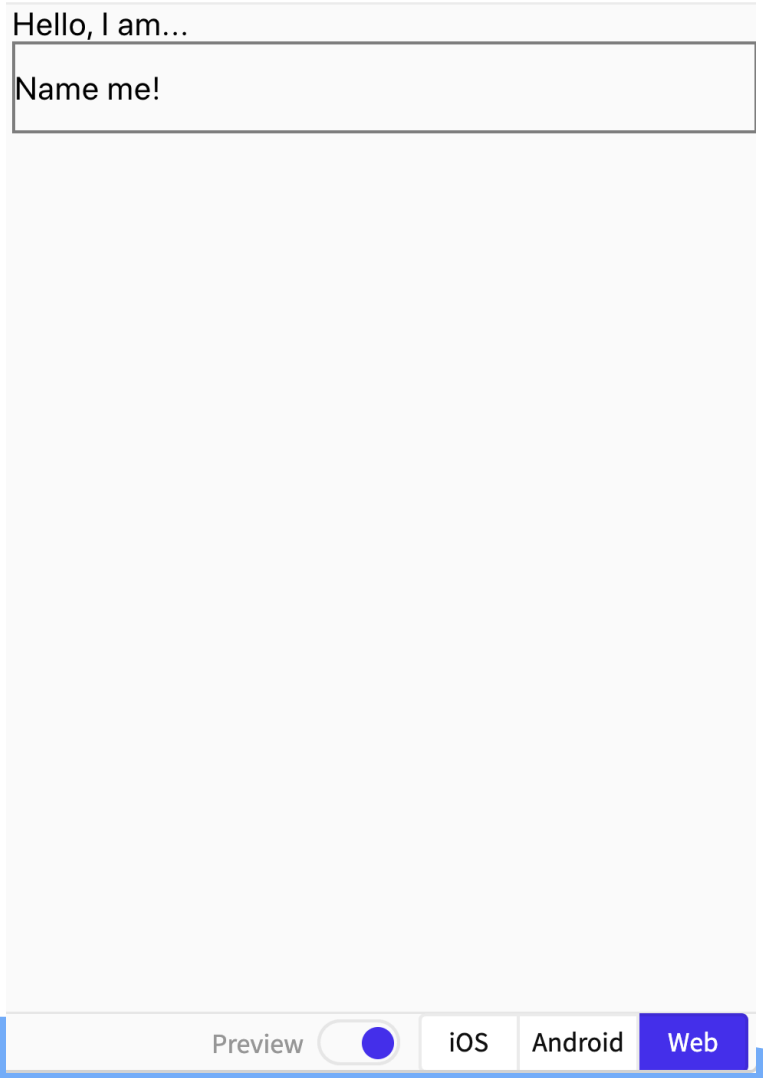
- React and React Native use **JSX,** a syntax that lets you write elements inside JavaScript

```
import React from 'react';
import { Text } from 'react-native';

export default function Cat() {
  function getFullName(firstName, secondName, thirdName) {
    return firstName + " " + secondName + " " + thirdName;
  }

  return (
    <Text>
      Hello, I am {getFullName("Rum", "Tum", "Tugger")}!
    </Text>
  );
}
```

Custom Components

- React lets you nest these components inside each other to create new components

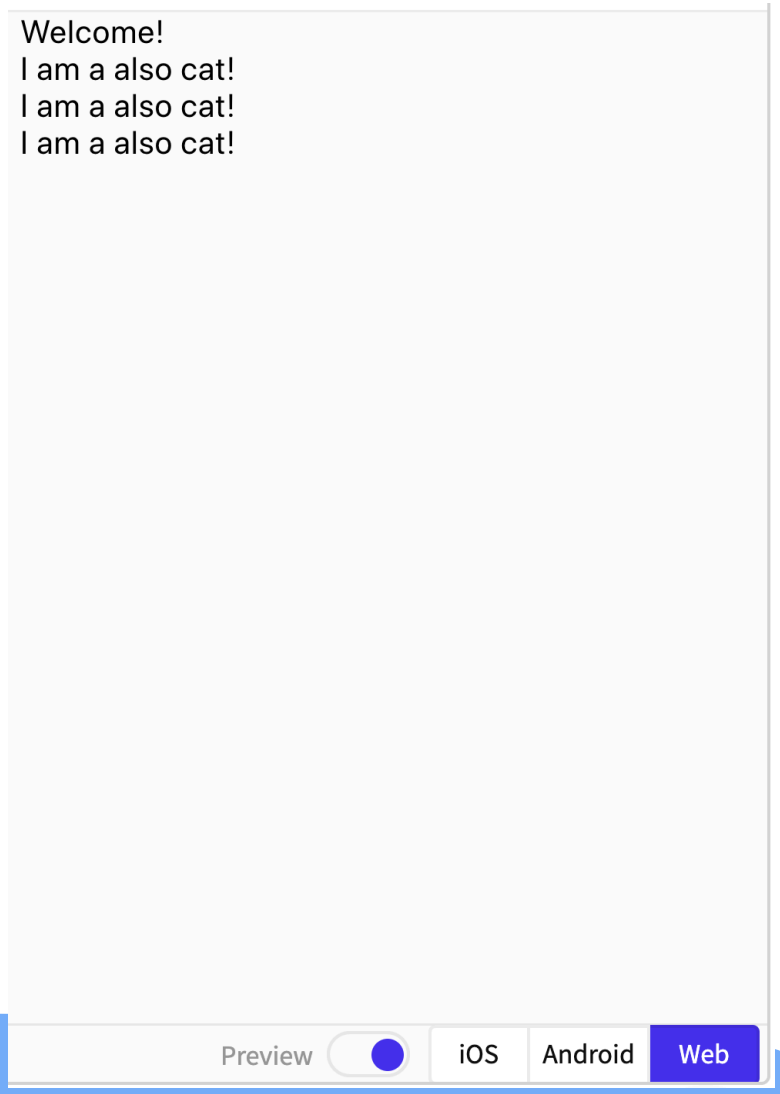
```
import React from 'react';
import { Text, TextInput, View } from 'react-native';

export default function Cat() {
  return (
    <View>
      <Text>Hello, I am...</Text>
      <TextInput
        style={{
          height: 40,
          borderColor: 'gray',
          borderWidth: 1
        }}
        defaultValue="Name me!"
      />
    </View>
  );
}
```



Custom Components

- React lets you nest these components inside each other to create new components

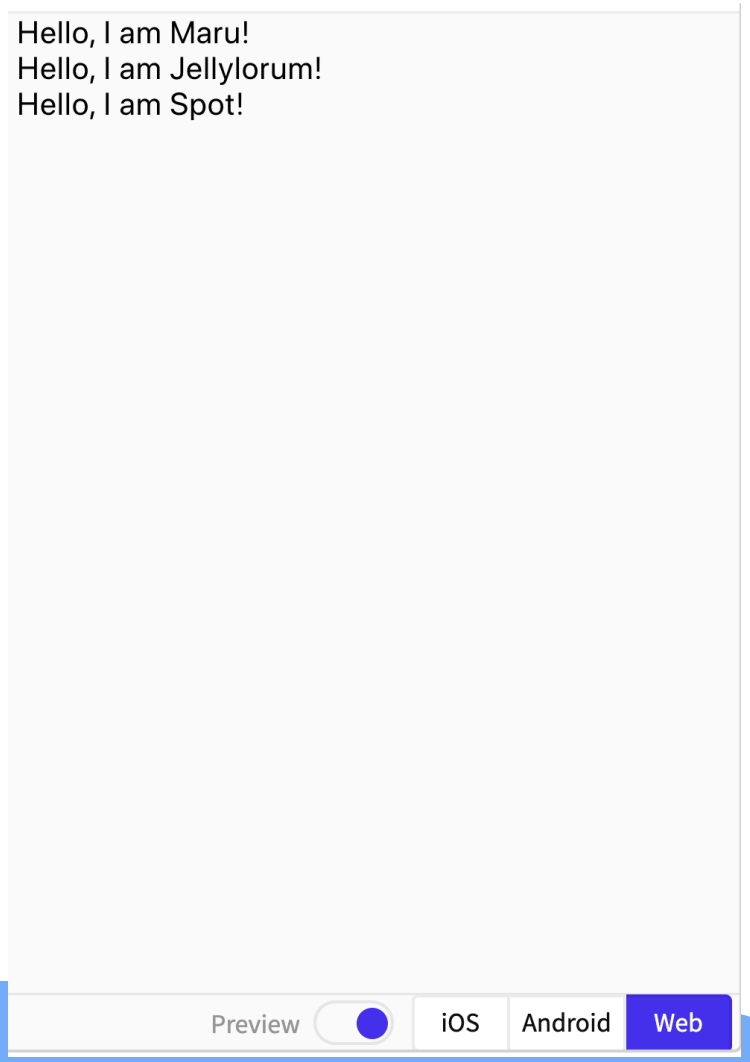
```
import React from 'react';
import { Text, TextInput, View } from 'react-native';

function Cat() {
  return (
    <View>
      <Text>I am a also cat!</Text>
    </View>
  );
}

export default function Cafe() {
  return (
    <View>
      <Text>Welcome!</Text>
      <Cat />
      <Cat />
      <Cat />
    </View>
  );
}
```



Props

- **Props** is short for “properties.” Props let you customize React components.

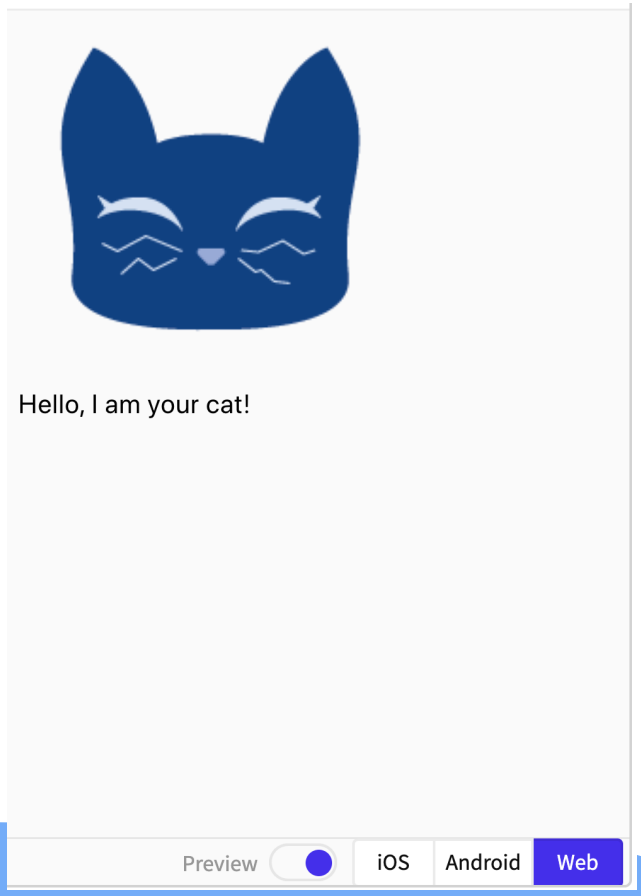
```
import React from 'react';
import { Text, View } from 'react-native';

function Cat(props) {
  return (
    <View>
      <Text>Hello, I am {props.name}!</Text>
    </View>
  );
}

export default function Cafe() {
  return (
    <View>
      <Cat name="Maru" />
      <Cat name="Jellylorum" />
      <Cat name="Spot" />
    </View>
  );
}
```



Props

- Most of React Native's Core Components can be customized with props, too.
  - For example, when using Image, you pass it a prop named source to define what image it shows:

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

export default function CatApp() {
 return (
  <View>
   <Image
    source="https://reactnative.dev/docs/assets/p_cat1.png"
    style={{width: 200, height: 200}}
   />
   <Text>Hello, I am your cat!</Text>
  </View>
 );
}
```

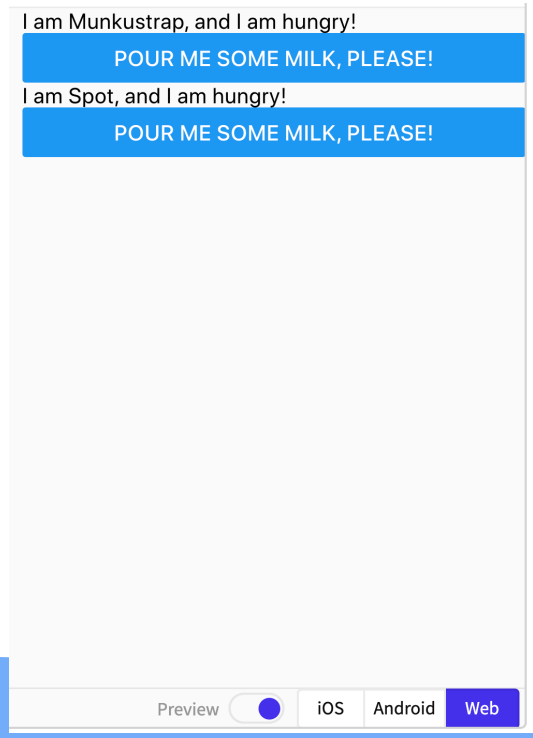


State

- Sate is useful for handling data that changes over time or that comes from user interaction.

```
import React, { useState } from "react";
import { Button, Text, View } from "react-native";

function Cat(props) {
 const [isHungry, setIsHungry] = useState(true);
 return (
  <View>
   <Text>
    I am {props.name}, and I am {isHungry ? "hungry" : "full"}!
   </Text>
   <Button
    onPress={() => {
     setIsHungry(false);
    }}
    disabled={!isHungry}
    title={isHungry ? "Pour me some milk, please!" : "Thank you!"}
   />
  </View>
 );
}
```

```
export default function Cafe() {
 return (
  <>
   <Cat name="Munkustrap" />
   <Cat name="Spot" />
  </>
 );
}
```

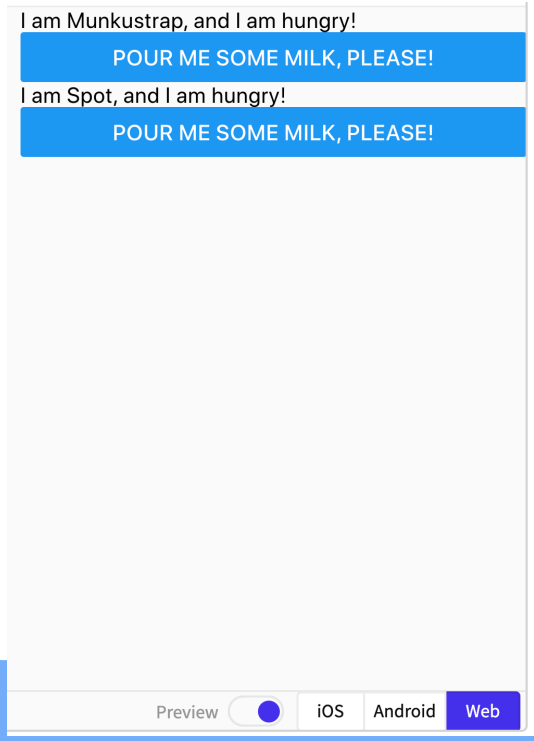


State

- Sate is useful for handling data that changes over time or that comes from user interaction.

```
import React, { Component } from "react";
import { Button, Text, View } from "react-native";
export class Cat extends Component {
  state = { isHungry: true };
  render(props) {
    return (
      <View>
        <Text>
          I am {this.props.name}, and I am
          {this.state.isHungry ? " hungry" : " full"}!
        </Text>
        <Button
          onPress={() => { this.setState({ isHungry: false }); }}
          disabled={!this.state.isHungry}
          title={
            this.state.isHungry ? "Pour me some milk, please!" : "Thank you!"
          }
        />
      </View>
    );
  }
}
```

```
export default class Cafe extends Component {
  render() {
    return (
      <>
        <Cat name="Munkustrap" />
        <Cat name="Spot" />
      </>
    );
  }
}
```



Handling Text Input

- TextInput is a Core Component that allows the user to enter text.
  - It has an onChangeText prop that takes a function to be called every time the text changed,
  - and an onSubmitEditing prop that takes a function to be called when the text is submitted.

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

export default function PizzaTranslator() {
 const [text, setText] = useState('');
 return (
  <View style={{padding: 10}}>
   <TextInput
    style={{height: 40}}
    placeholder="Type here to translate!"
    onChangeText={text => setText(text)}
    defaultValue={text}
   />
   <Text style={{padding: 10, fontSize: 42}}>
    {text.split(' ').map((word) => word && '🍕').join(' ')}
   </Text>
  </View>
 );
}
```

Using a ScrollView

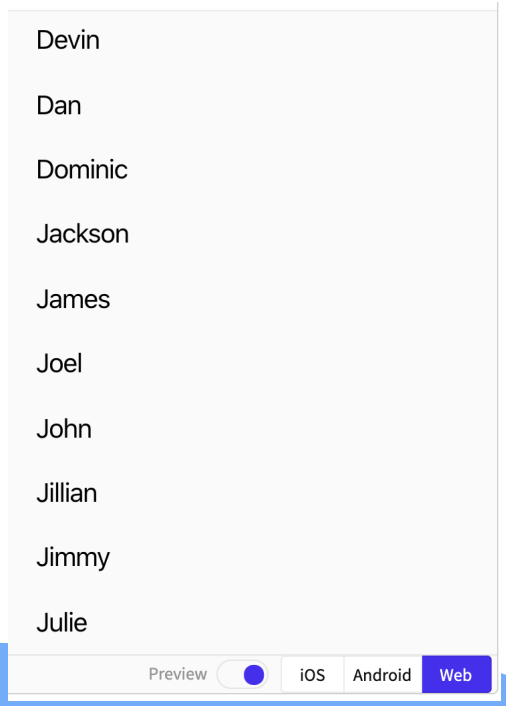
- The ScrollView is a generic scrolling container that can contain multiple components and views.

```
import React from 'react';
import { Image, ScrollView, Text } from 'react-native';

const logo = {
  uri: 'https://reactnative.dev/img/tiny_logo.png',
  width: 64,
  height: 64
};

export default App = () => (
  <ScrollView>
    <Text style={{ fontSize: 96 }}>Scroll me plz</Text>
    <Image source={logo} />
    <Image source={logo} />
    <Image source={logo} />
    <Image source={logo} />
    <Image source={logo} />
    <Text style={{ fontSize: 96 }}>If you like</Text>
    <Image source={logo} />
    <Image source={logo} />
    <Image source={logo} />
    <Image source={logo} />
    <Text style={{ fontSize: 96 }}>Scrolling down</Text>
    <Image source={logo} />
    <Image source={logo} />
    <Image source={logo} />
    <Image source={logo} />
    <Image source={logo} />
    <Text style={{ fontSize: 96 }}>What's the best</Text>
    <Image source={logo} />
    <Image source={logo} />
    <Image source={logo} />
    <Image source={logo} />
    <Image source={logo} />
    <Text style={{ fontSize: 96 }}>Framework around?</Text>
    <Image source={logo} />
    <Image source={logo} />
    <Image source={logo} />
    <Image source={logo} />
    <Image source={logo} />
    <Text style={{ fontSize: 80 }}>React Native</Text>
  </ScrollView>
);
```

Using List Views

- The FlatList component displays a scrolling list of changing, but similarly structured, data.

```
import React from 'react';
import { FlatList, StyleSheet, Text, View } from 'react-native';

export default FlatListBasics = () => {
  return (
    <View style={styles.container}>
      <FlatList
        data={[
          {key: 'Devin'},
          {key: 'Dan'},
          {key: 'Dominic'},
          {key: 'Jackson'},
          {key: 'James'},
          {key: 'Joel'},
          {key: 'John'},
          {key: 'Jillian'},
          {key: 'Jimmy'},
          {key: 'Julie'},
        ]}
        renderItem={({item}) =>
          <Text style={styles.item}>{item.key}</Text>}
      />
    </View>
  );
}

const styles = StyleSheet.create({
  container: {
    flex: 1,
    paddingTop: 22
  },
  item: {
    padding: 10,
    fontSize: 18,
    height: 44,
  },
})
```

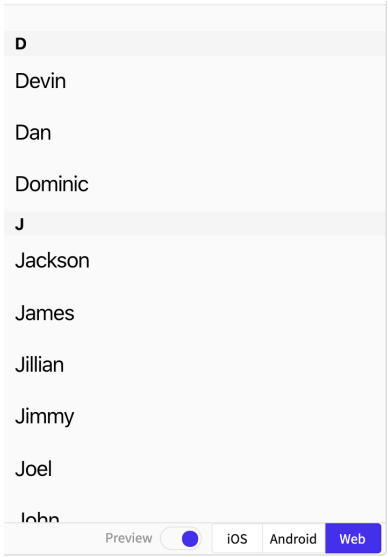


Using List Views

- If you want to render a set of data broken into logical sections, maybe with section headers, similar to UITableViews on iOS, then a SectionList is the way to go.

```
import React from 'react';
import { SectionList, StyleSheet, Text, View } from 'react-native';

export default SectionListBasics = () => {
   return (
    <View style={styles.container}>
     <SectionList
      sections={[
       {title: 'D', data: ['Devin', 'Dan', 'Dominic']},
       {title: 'J', data: ['Jackson', 'James', 'Jillian', 'Jimmy', 'Joel', 'John', 'Julie']},
      ]}
      renderItem={({item}) => <Text style={styles.item}>{item}</Text>}
      renderSectionHeader={({section}) => <Text style={styles.sectionHeader}>{section.title}</Text>}
      keyExtractor={(item, index) => index}
     />
    </View>
   );
}
```

```
const styles = StyleSheet.create({
 container: {
  flex: 1,
  paddingTop: 22
 },
 sectionHeader: {
  paddingTop: 2,
  paddingLeft: 10,
  paddingRight: 10,
  paddingBottom: 2,
  fontSize: 14,
  fontWeight: 'bold',
  backgroundColor: 'rgba(247,247,247,1.0)',
 },
 item: {
  padding: 10,
  fontSize: 18,
  height: 44,
 },
})
```



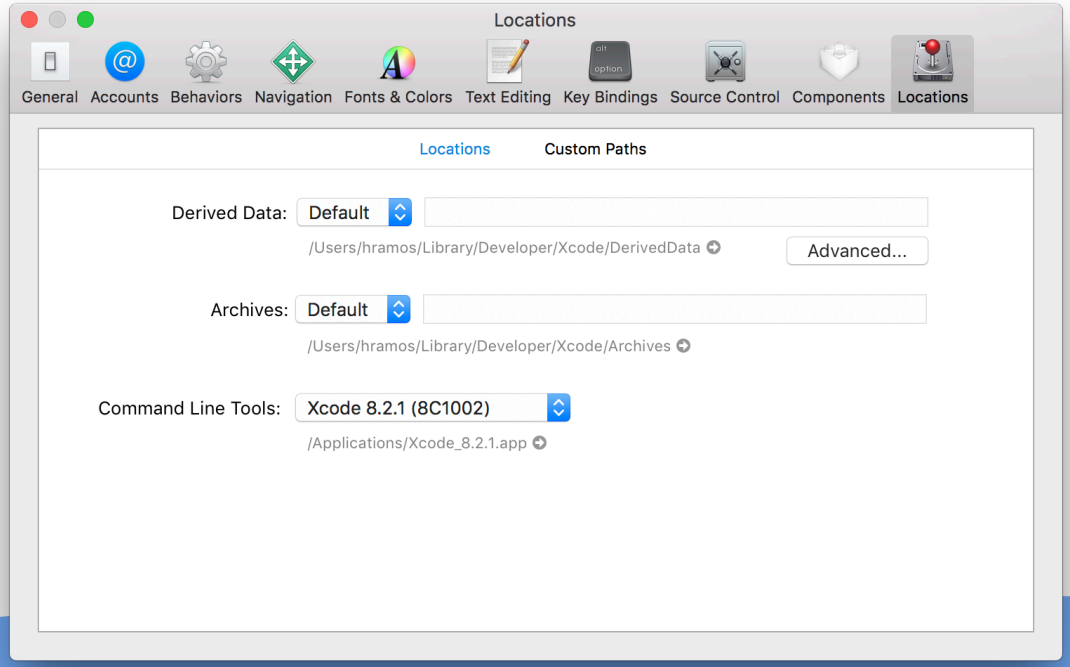
Setting up the development environment

- Node & Watchman

  brew install node

  brew install watchman

- Xcode & CocoaPods
  - Command Line Tools



- installing an iOS Simulator in Xcode
  - To install a simulator, open **Xcode > Preferences...** and select the **Components** tab

- CocoaPods

  sudo gem install cocoapods

Setting up the development environment

- Node & Watchman
  brew install node
  brew install watchman

- Java Development Kit
  brew cask install adoptopenjdk/openjdk/adoptopenjdk8

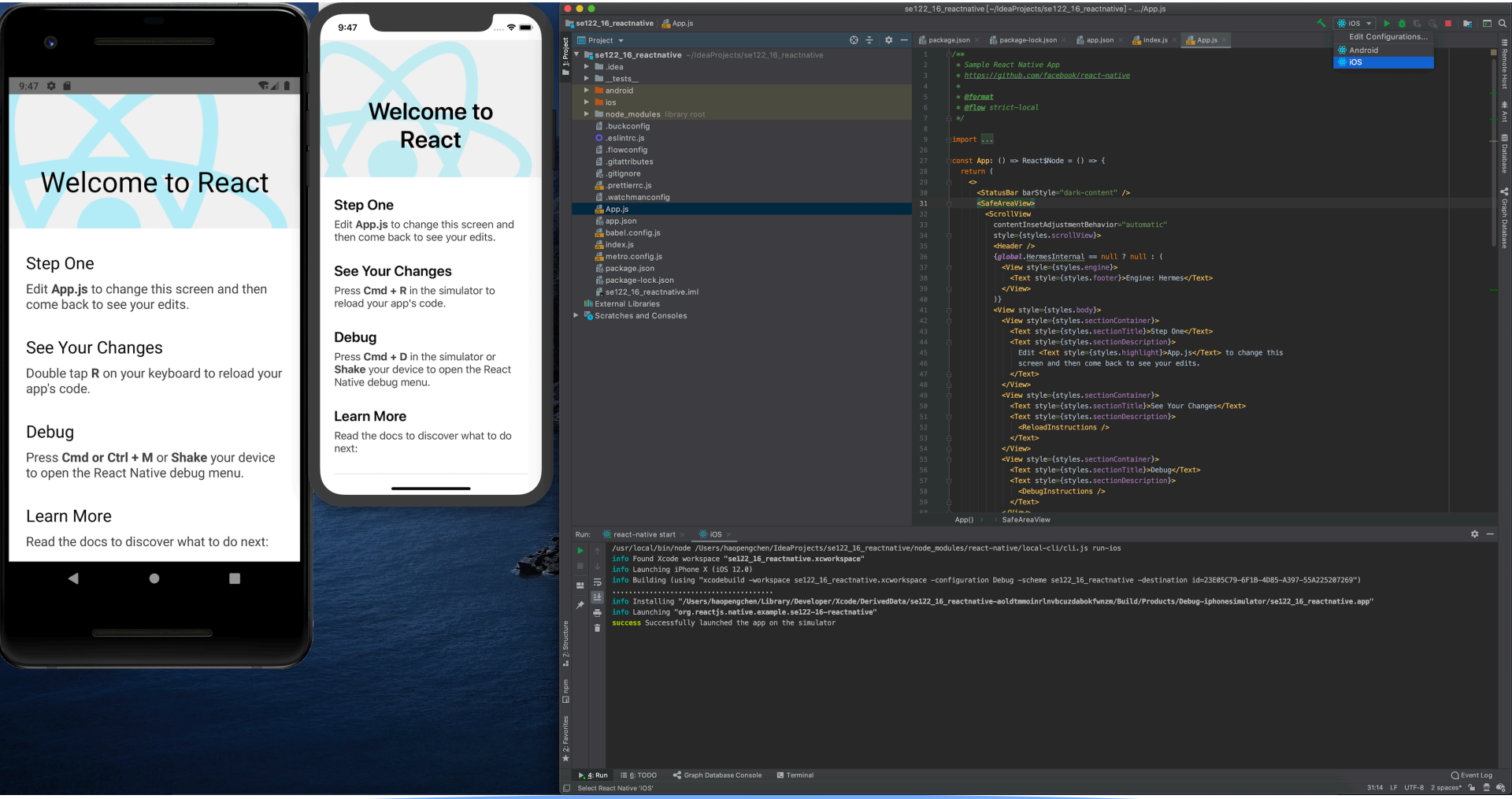
- Android development environment
  1. Install Android Studio
  2. Install the Android SDK
  3. Configure the ANDROID_HOME environment variable

     Add the following lines to your $HOME/.bash_profile or $HOME/.bashrc config file:

     export ANDROID_HOME=$HOME/Library/Android/sdk
     export PATH=$PATH:$ANDROID_HOME/emulator
     export PATH=$PATH:$ANDROID_HOME/tools
     export PATH=$PATH:$ANDROID_HOME/tools/bin
     export PATH=$PATH:$ANDROID_HOME/platform-tools

Setting up the development environment

- Run React Native Sample



Design -Style

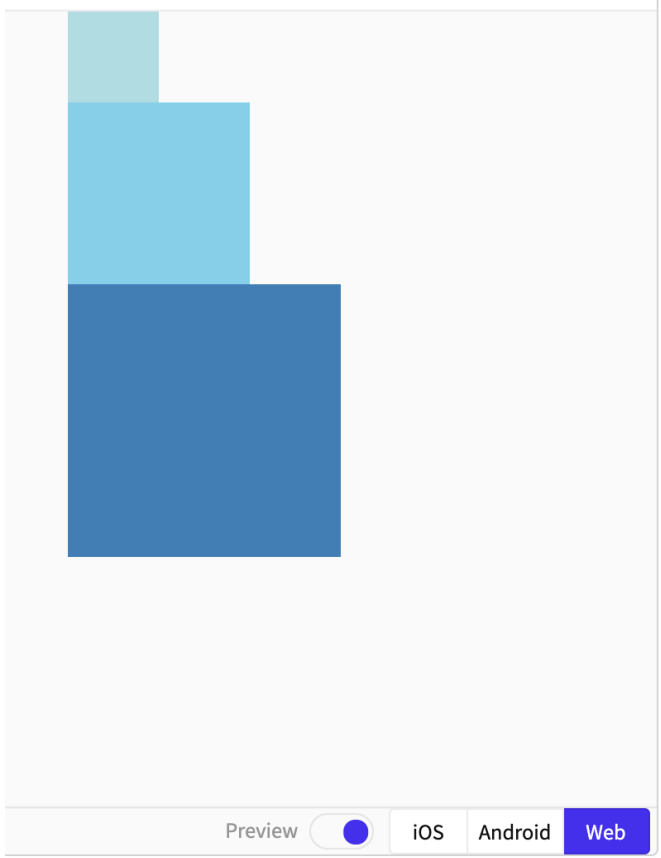
- All of the core components accept a prop named style

```
import React from 'react';
import { StyleSheet, Text, View } from 'react-native';
const styles = StyleSheet.create({
  container: {
    marginTop: 50,
  },
  bigBlue: {
    color: 'blue',
    fontWeight: 'bold',
    fontSize: 30,
  },
  red: {
    color: 'red',
  },
});
export default LotsOfStyles = () => {
    return (
      <View style={styles.container}>
        <Text style={styles.red}>just red</Text>
        <Text style={styles.bigBlue}>just bigBlue</Text>
        <Text style={[styles.bigBlue, styles.red]}>bigBlue, then red</Text>
        <Text style={[styles.red, styles.bigBlue]}>red, then bigBlue</Text>
      </View>
    );
}
```

Design – Height and Width

- Fixed Dimensions

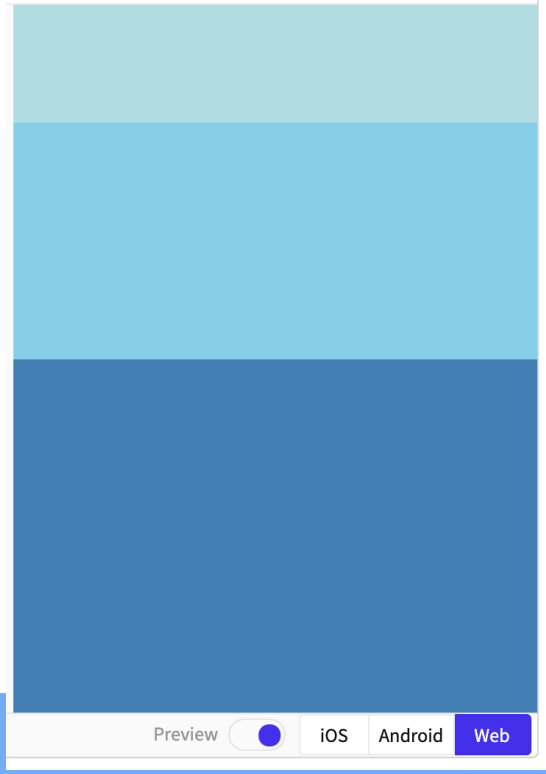
```
import React, { Component } from 'react';
import { View } from 'react-native';

export default function
FixedDimensionsBasics() {
   return (
     <View>
       <View style={{width: 50, height: 50,
backgroundColor: 'powderblue'}} />
       <View style={{width: 100, height: 100,
backgroundColor: 'skyblue'}} />
       <View style={{width: 150, height: 150,
backgroundColor: 'steelblue'}} />
     </View>
   );
}
```



Design – Height and Width

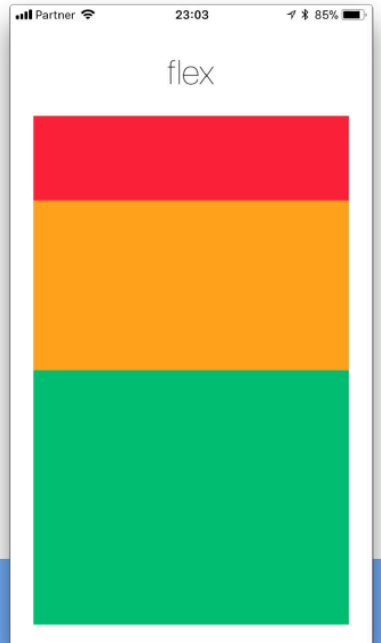
- Flex Dimensions

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

export default function FlexDimensionsBasics() {
  return (
    // Try removing the `flex: 1` on the parent View.
    // The parent will not have dimensions, so the children can't expand.
    // What if you add `height: 300` instead of `flex: 1`?
    <View style={{flex: 1}}>
      <View style={{flex: 1, backgroundColor: 'powderblue'}} />
      <View style={{flex: 2, backgroundColor: 'skyblue'}} />
      <View style={{flex: 3, backgroundColor: 'steelblue'}} />
    </View>
  );
}
```



Layout with Flexbox

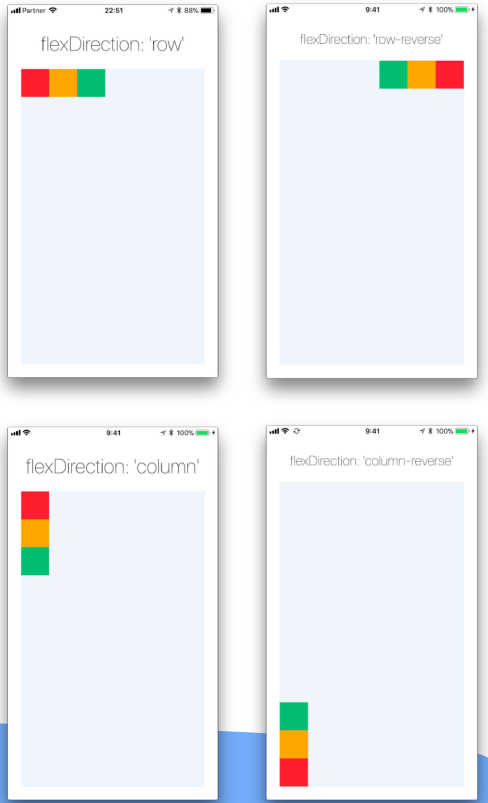
- A component can specify the layout of its children using the Flexbox algorithm.
  - Flexbox is designed to provide a consistent layout on different screen sizes.
- Flex
  - flex will define how your items are going to **“fill”** over the available space along your main axis.
  - Space will be divided according to each element's flex property.



Layout with Flexbox

- Flex Direction
  - row Align children from left to right.
  - column (**default value**) Align children from top to bottom.
  - row-reverse Align children from right to left.
  - column-reverse Align children from bottom to top.

```
import React from 'react';
import { View } from 'react-native';

export default FlexDirectionBasics = () => {
   return (
    // Try setting `flexDirection` to `column`.
    <View style={{flex: 1, flexDirection: 'row'}}>
      <View style={{width: 50, height: 50, backgroundColor: 'powderblue'}} />
      <View style={{width: 50, height: 50, backgroundColor: 'skyblue'}} />
      <View style={{width: 50, height: 50, backgroundColor: 'steelblue'}} />
    </View>
   );
};
```



Layout with Flexbox

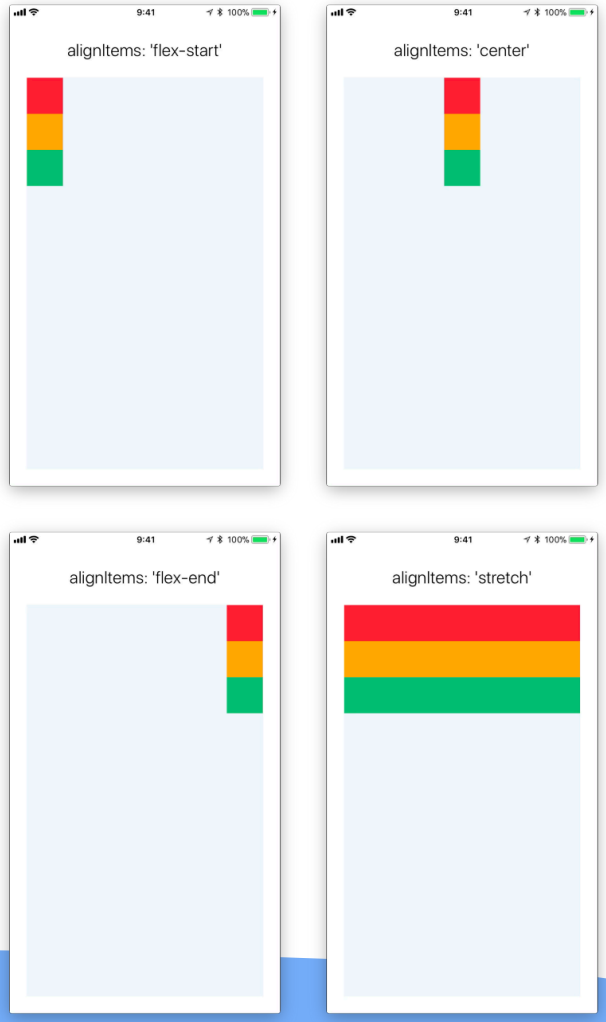
- Justify Content
  - justifyContent describes how to align children within the main axis of their container.

```
import React from 'react';
import { View } from 'react-native';

export default JustifyContentBasics = () => {
  return (
    // Try setting `justifyContent` to `center`.
    // Try setting `flexDirection` to `row`.
    <View style={{
      flex: 1,
      flexDirection: 'column',
      justifyContent: 'space-between',
    }}>
      <View style={{width: 50, height: 50, backgroundColor: 'powderblue'}} />
      <View style={{width: 50, height: 50, backgroundColor: 'skyblue'}} />
      <View style={{width: 50, height: 50, backgroundColor: 'steelblue'}} />
    </View>
  );
};
```

Layout with Flexbox

- Align Items
  - alignItems describes how to align children along the cross axis of their container.

```
import React from 'react';
import { View } from 'react-native';

export default AlignItemsBasics = () => {
  return (
    // Try setting `alignItems` to 'flex-start'
    // Try setting `justifyContent` to `flex-end`.
    // Try setting `flexDirection` to `row`.
    <View style={{
      flex: 1,
      flexDirection: 'column',
      justifyContent: 'center',
      alignItems: 'stretch',
    }}>
      <View style={{width: 50, height: 50, backgroundColor: 'powderblue'}} />
      <View style={{height: 50, backgroundColor: 'skyblue'}} />
      <View style={{height: 100, backgroundColor: 'steelblue'}} />
    </View>
  );
};
```

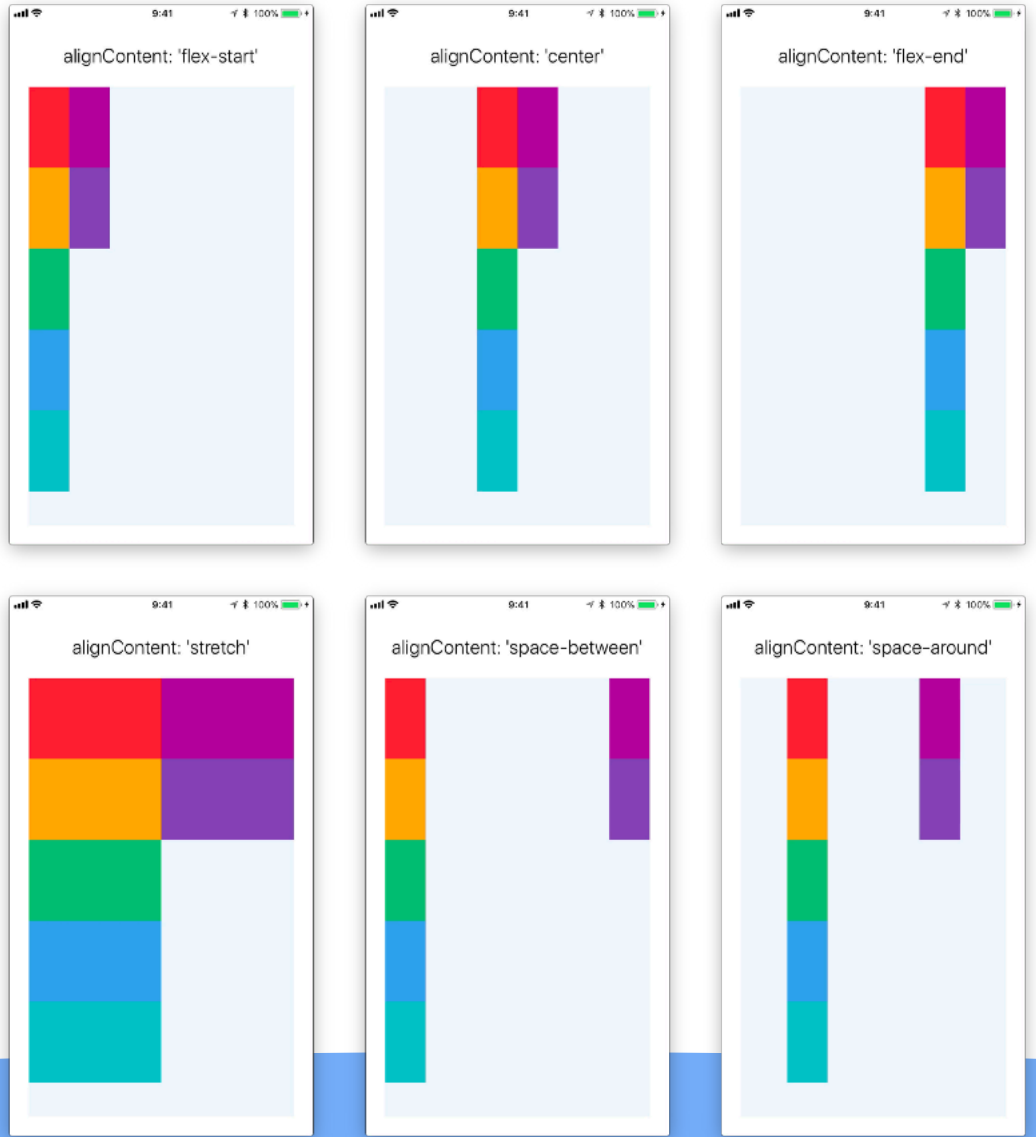


Layout with Flexbox

- Align Self
  - alignSelf has the same options and effect as alignItems but instead of affecting the children within a container, you can apply this property to a single child to change its alignment within its parent.

Layout with Flexbox

- Align Content

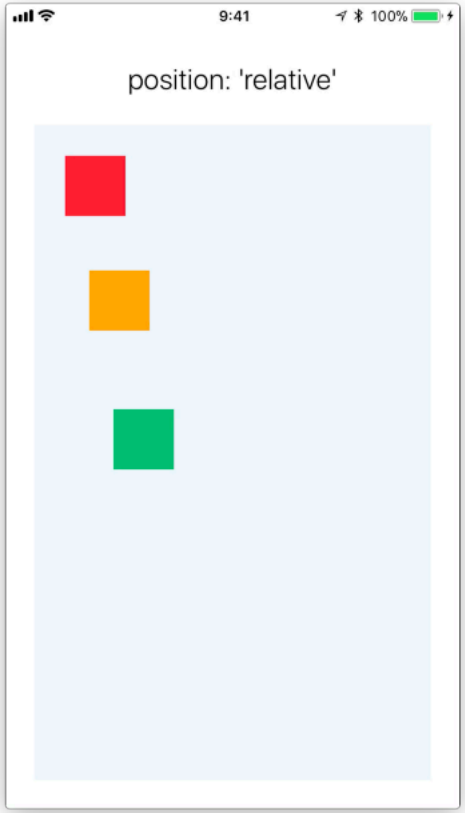


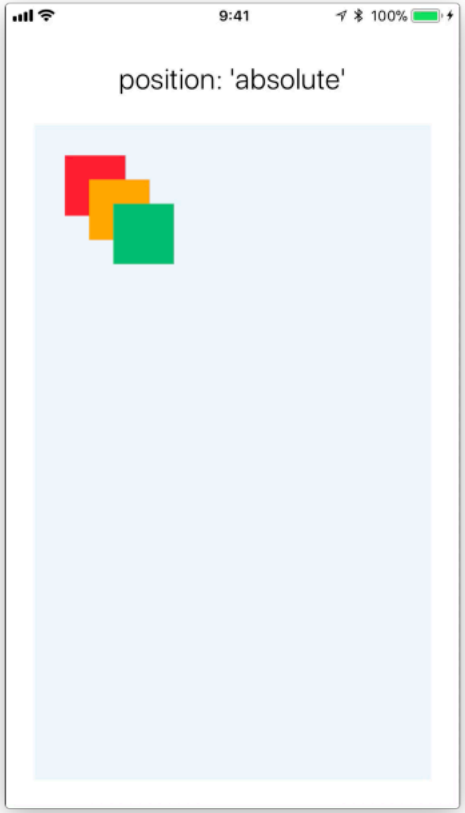
Layout with Flexbox

- The flexWrap property
  - is set on containers and it controls what happens when children overflow the size of the container along the main axis.

Layout with Flexbox

- Absolute & Relative Layout
  - The position type of an element defines how it is positioned within its parent.





Handling Touches

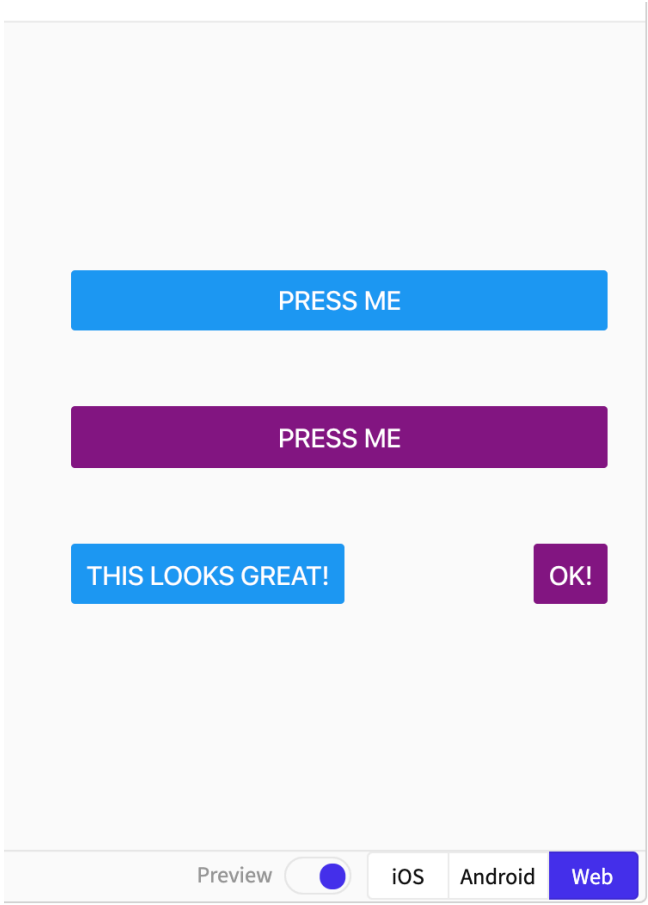
- Displaying a basic button
  - Button provides a basic button component that is rendered nicely on all platforms.

```
import React, { Component } from 'react';
import { Button, StyleSheet, View } from 'react-native';

export default class ButtonBasics extends Component {
 _onPressButton() {
  alert('You tapped the button!')
 }

 render() {
  return (
   <View style={styles.container}>
    <View style={styles.buttonContainer}>
     <Button
      onPress={this._onPressButton}
      title="Press Me"
     />
    </View>
    <View style={styles.buttonContainer}>
     <Button
      onPress={this._onPressButton}
      title="Press Me"
      color="#841584"
     />
    </View>
    <View style={styles.alternativeLayoutButtonContainer}>
     <Button
      onPress={this._onPressButton}
      title="This looks great!"
     />
     <Button
      onPress={this._onPressButton}
      title="OK!"
      color="#841584"
     />
    </View>
   </View>
  );
 }
}
```

Handling Touches

- Displaying a basic button
  - Button provides a basic button component that is rendered nicely on all platforms.

```
const styles = StyleSheet.create({
 container: {
  flex: 1,
  justifyContent: 'center',
 },
 buttonContainer: {
   margin: 20
 },
 alternativeLayoutButtonContainer: {
   margin: 20,
   flexDirection: 'row',
   justifyContent: 'space-between'
 }
});
```



Handling Touches

- Touchables
  - The "Touchable" components provide the capability to capture tapping gestures, and can display feedback when a gesture is recognized.

```
import React, { Component } from 'react';
import { Platform, StyleSheet, Text, TouchableHighlight, TouchableOpacity, TouchableNativeFeedback,
TouchableWithoutFeedback, View } from 'react-native';

export default class Touchables extends Component {
 _onPressButton() {
  alert('You tapped the button!')
 }

 _onLongPressButton() {
  alert('You long-pressed the button!')
 }
```

Handling Touches

- Touchables
  - The "Touchable" components provide the capability to capture tapping gestures, and can display feedback when a gesture is recognized.

```
render() {
 return (
  <View style={styles.container}>
   <TouchableHighlight onPress={this._onPressButton} underlayColor="white">
    <View style={styles.button}>
     <Text style={styles.buttonText}>TouchableHighlight</Text>
    </View>
   </TouchableHighlight>
   <TouchableOpacity onPress={this._onPressButton}>
    <View style={styles.button}>
     <Text style={styles.buttonText}>TouchableOpacity</Text>
    </View>
   </TouchableOpacity>
   <TouchableNativeFeedback
     onPress={this._onPressButton}
     background={Platform.OS === 'android' ? TouchableNativeFeedback.SelectableBackground() : ''}>
    <View style={styles.button}>
     <Text style={styles.buttonText}>TouchableNativeFeedback {Platform.OS !== 'android' ? '(Android only)' : ''}</Text>
    </View>
   </TouchableNativeFeedback>
```

Handling Touches

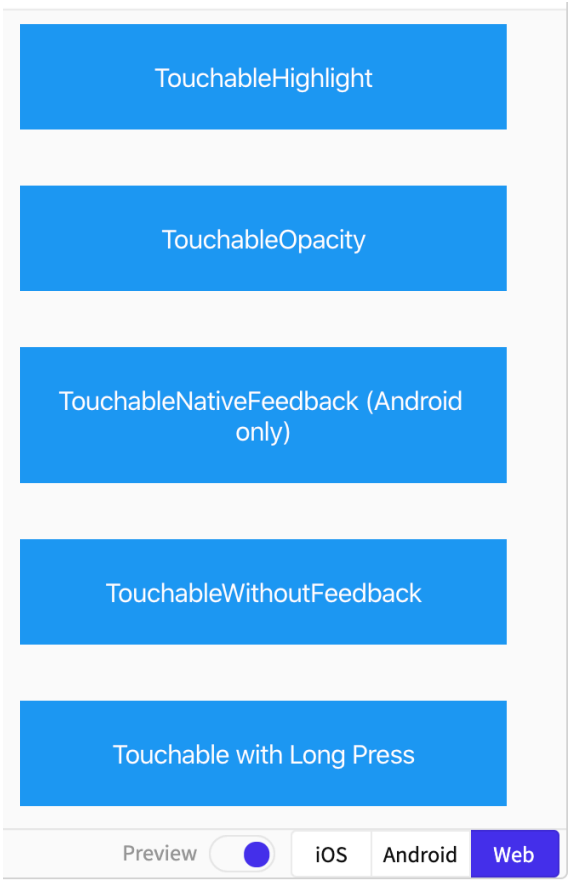
- Touchables
  - The "Touchable" components provide the capability to capture tapping gestures, and can display feedback when a gesture is recognized.

```
      <TouchableWithoutFeedback
       onPress={this._onPressButton}
       >
      <View style={styles.button}>
       <Text style={styles.buttonText}>TouchableWithoutFeedback</Text>
      </View>
     </TouchableWithoutFeedback>
     <TouchableHighlight onPress={this._onPressButton} onLongPress={this._onLongPressButton} underlayColor="white">
      <View style={styles.button}>
       <Text style={styles.buttonText}>Touchable with Long Press</Text>
      </View>
     </TouchableHighlight>
    </View>
   );
  }
}
```

Handling Touches

- Touchables
  - The "Touchable" components provide the capability to capture tapping gestures, and can display feedback when a gesture is recognized.

```
const styles = StyleSheet.create({
  container: {
    paddingTop: 60,
    alignItems: 'center'
  },
  button: {
    marginBottom: 30,
    width: 260,
    alignItems: 'center',
    backgroundColor: '#2196F3'
  },
  buttonText: {
    textAlign: 'center',
    padding: 20,
    color: 'white'
  }
});
```



Navigating Between Screens

- React Navigation
  - React Navigation provides a straightforward navigation solution, with the ability to present common stack navigation and tabbed navigation patterns on both Android and iOS.

- Installation and setup
  - npm install @react-navigation/native @react-navigation/stack
  - Now, you need to wrap the whole app in NavigationContainer. Usually you'd do this in your entry file, such as index.js or App.js:

```
import 'react-native-gesture-handler';
import * as        from 'react';
import {NavigationContainer} from '@react-navigation/native';
export default function App() {
  return (
    <NavigationContainer>{/* Rest of your app code */}</NavigationContainer>
  );
}
```

Navigating Between Screens

- In this example, there are 2 screens (Home and Profile) defined using the Stack.Screen component.

```
import * as React from 'react';
import {NavigationContainer} from '@react-navigation/native';
import {createStackNavigator} from '@react-navigation/stack';

const Stack = createStackNavigator();

function MyStack() {
  return (
    <NavigationContainer>
      <Stack.Navigator>
        <Stack.Screen name="Home" component={Home} options={{title: 'Welcome'}} />
        <Stack.Screen name="Profile" component={Profile} />
      </Stack.Navigator>
    </NavigationContainer>
  );
}
```

Navigating Between Screens

- Each screen takes a component prop that is a React component.
  - Those components receive a prop called navigation which has various methods to link to other screens.
  - For example, you can use navigation.navigate to go to the Profile screen:

```
function HomeScreen({navigation}) {
  return (
    <Button
      title="Go to Jane's profile"
      onPress={() => navigation.navigate('Profile', {name: 'Jane'})}
    />
  );
}
```

- React Navigation Getting Started
  - https://reactnavigation.org/docs/getting-started/

Animations

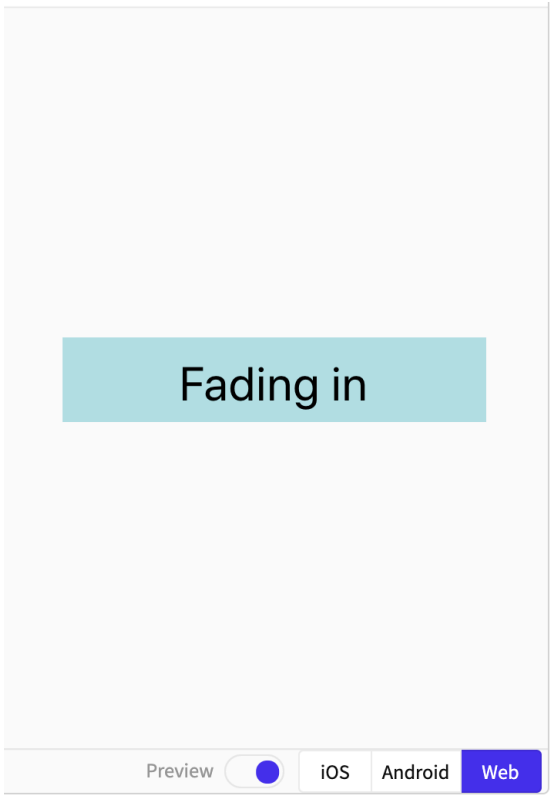
- Animated API
  - The Animated API is designed to concisely express a wide variety of interesting animation and interaction patterns in a very performant way.

```
import React, { useRef, useEffect } from 'react';
import { Animated, Text, View } from 'react-native';

const FadeInView = (props) => {
 const fadeAnim = useRef(new Animated.Value(0)).current  // Initial value for opacity: 0

 React.useEffect(() => {
  Animated.timing(
   fadeAnim,
   {
    toValue: 1,
    duration: 10000,
   }
  ).start();
 }, [])
```

Animations

- Animated API

```
 return (
  <Animated.View                 // Special animatable View
    style={{
      ...props.style,
      opacity: fadeAnim,         // Bind opacity to animated value
    }}
  >
    {props.children}
  </Animated.View>
 );
}

// You can then use your `FadeInView` in place of a `View` in your components:
export default () => {
 return (
  <View style={{flex: 1, alignItems: 'center', justifyContent: 'center'}}>
    <FadeInView style={{width: 250, height: 50, backgroundColor: 'powderblue'}}>
      <Text style={{fontSize: 28, textAlign: 'center', margin: 10}}>Fading in</Text>
    </FadeInView>
  </View>
 )
}
```



Gesture Responder System

- The gesture responder system manages the lifecycle of gestures in your app.
  - 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.

- Best Practices
  - To make your app feel great, every action should have the following attributes:
  - Feedback/highlighting- show the user what is handling their touch, and what will happen when they release the gesture
  - Cancel-ability- when making an action, the user should be able to abort it mid-touch by dragging their finger away

Responder Lifecycle

- A view can become the touch responder by implementing the correct negotiation methods.

- There are two methods to ask the view if it wants to become responder:
  - View.props.onStartShouldSetResponder: (evt) => true, - Does this view want to become responder on the start of a touch?
  - View.props.onMoveShouldSetResponder: (evt) => true, - Called for every touch move on the View when it is not the responder: does this view want to "claim" touch responsiveness?

Responder Lifecycle

- If the View returns true and attempts to become the responder, one of the following will happen:
  - View.props.onResponderGrant: (evt) => {} - The View is now responding for touch events. This is the time to highlight and show the user what is happening
  - View.props.onResponderReject: (evt) => {} - Something else is the responder right now and will not release it

- If the view is responding, the following handlers can be called:
  - View.props.onResponderMove: (evt) => {} - The user is moving their finger
  - View.props.onResponderRelease: (evt) => {} - Fired at the end of the touch, ie "touchUp"
  - View.props.onResponderTerminationRequest: (evt) => true - Something else wants to become responder. Should this view release the responder? Returning true allows release
  - View.props.onResponderTerminate: (evt) => {} - The responder has been taken from the View. Might be taken by other views after a call to onResponderTerminationRequest, or might be taken by the OS without asking (happens with control center/ notification center on iOS)

Responder Lifecycle

- evt is a synthetic touch event with the following form:
- nativeEvent
  - changedTouches - Array of all touch events that have changed since the last event
  - identifier - The ID of the touch
  - locationX - The X position of the touch, relative to the element
  - locationY - The Y position of the touch, relative to the element
  - pageX - The X position of the touch, relative to the root element
  - pageY - The Y position of the touch, relative to the root element
  - target - The node id of the element receiving the touch event
  - timestamp - A time identifier for the touch, useful for velocity calculation
  - touches - Array of all current touches on the screen

Networking – Using Fetch

- Making requests
  - In order to fetch content from an arbitrary URL, you can pass the URL to fetch:

```
fetch('https://mywebsite.com/mydata.json');
```

  - Fetch also takes an optional second argument that allows you to customize the HTTP request. You may want to specify additional headers, or make a POST request:

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

Networking – Using Fetch

- Handling the response
  - Networking is an inherently asynchronous operation. Fetch methods will return a Promise that makes it straightforward to write code that works in an asynchronous manner:

```
function getMoviesFromApi() {
  return fetch('https://reactnative.dev/movies.json')
    .then((response) => response.json())
    .then((json) => {
      return json.movies;
    })
    .catch((error) => {
      console.error(error);
    });
}
```

Networking – Using Fetch

- Handling the response
  - You can also use the async / await syntax in a React Native app: :

```
async function getMoviesFromApiAsync() {
  try {
    let response = await fetch('https://reactnative.dev/movies.json');
    let json = await response.json();
    return json.movies;
  } catch (error) {
    console.error(error);
  }
}
```

  - Don't forget to catch any errors that may be thrown by fetch, otherwise they will be dropped silently.

Networking – Using Fetch


```
import React, { useEffect, useState } from 'react';
import { ActivityIndicator, FlatList, Text, View } from 'react-native';
export default App = () => {
 const [isLoading, setLoading] = useState(true);
 const [data, setData] = useState([]);

 useEffect(() => {
  fetch('https://reactnative.dev/movies.json')
    .then((response) => response.json())
    .then((json) => setData(json.movies))
    .catch((error) => console.error(error))
    .finally(() => setLoading(false));
 });
 return (
  <View style={{ flex: 1, padding: 24 }}>
   {isLoading ? <ActivityIndicator/> : (
     <FlatList
       data={data}
       keyExtractor={({ id }, index) => id}
       renderItem={({ item }) => (
         <Text>{item.title}, {item.releaseYear}</Text>
       )}
     />
   )}
  </View>
 );
};
```

Networking – Using Fetch


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

export default class App extends Component {
 constructor(props) {
  super(props);

  this.state = {
   data: [],
   isLoading: true
  };
 }

 componentDidMount() {
  fetch('https://reactnative.dev/movies.json')
   .then((response) => response.json())
   .then((json) => {
    this.setState({ data: json.movies });
   })
   .catch((error) => console.error(error))
   .finally(() => {
    this.setState({ isLoading: false });
   });
 }

 render() {
  const { data, isLoading } = this.state;

  return (
   <View style={{ flex: 1, padding: 24 }}>
    {isLoading ? <ActivityIndicator/> : (
     <FlatList
      data={data}
      keyExtractor={({ id }, index) => id}
      renderItem={({ item }) => (
       <Text>{item.title}, {item.releaseYear}</Text>
      )}
     />
    )}
   </View>
  );
 }
};
```

Networking – Using Other Libraries


```
var request = new XMLHttpRequest();
request.onreadystatechange = (e) => {
  if (request.readyState !== 4) {
    return;
  }
  if (request.status === 200) {
     console.log('success', request.responseText);
  } else {
     console.warn('error’);
  }
};

request.open('GET', 'https://mywebsite.com/endpoint/’);
request.send()
```

References

- React Native
  - https://reactnative.dev/
- React Native
  - https://www.jetbrains.com/help/idea/react-native.html
- Setting up the development environment
  - https://reactnative.dev/docs/environment-setup
- React/RCTBridgeDelegate.h' file not found
  - https://blog.csdn.net/herman_hy/article/details/98624153
- React native开发中遇到的坑
  - https://www.jianshu.com/p/658d34f2187a
- React Native Apps
  - https://github.com/ReactNativeNews/React-Native-Apps

- *Web*开发技术
- *Web Application Development*

Thank You!