React Native

“An open source, cross-platform framework for building native mobile apps with JavaScript and React using declarative components.”

https://facebook.github.io/react-native/
Who is using React Native?
React Native

- Is a framework for building native apps using Javascript
  - RN apps render using native views
    - RN apps have direct access to all the native APIs and views offered by the underlying mobile OS
    - RN apps have the same feel and performance as that of a native application
  - RN can be considered as a set of React components
    - The developer will be writing the code just like for any other React web app but the output will be a native application
React Native

- Comes with a bunch of atomic components
- Most of the atomic components are platform agnostic, but some are not
- We can implement our own atomic components by leveraging RN helpers for Java and Swift
RN Architecture

• **Native Code/Modules**: Swift and Java code

• **Javascript VM**: RN uses JavaScriptCore (Safari)
  - In case of Android, RN bundles the JavaScriptCore along with the application
  - In case of Chrome debugging mode, RN uses the V8 engine and communicates with native code via WebSocket

• **React Native Bridge**: a C++/Java bridge responsible for communication between the native and Javascript threads
  - A custom protocol is used for message passing
RN architecture

Native Modules
- Android - Java
- iOS - Obj C /Swift

RN Bridge
(Java/C++)

JS Virtual Machine
(JavaScriptCore)
Threaded model

- **Main thread** is spawned as soon as the application launches
  - Loads the app and starts the JS thread to execute the JS code
  - Listens to UI events and passes them to the JS thread via the RN Bridge
- **Javascript thread** runs bundled JS/RN code
- **Additional threads** can be spawned on custom native modules to speed up the performance
  - For example, animations are handled in React Native by a separate native thread to offload the work from the JS thread
• We need to understand some of the basic React concepts, like JSX, components, state, and props
• RN ships with ES (ECMA-262)
  • For example: import, from, class, and extends
• JSX is a syntax for embedding XML within JavaScript
  • JSX lets you write your markup language inside code
  • You can put any JavaScript expression inside braces in JSX
How to create a RN app

```
react-native init <projectName>
```
Quick Start

These are the docs for Expo. Expo is a set of tools, libraries and services which let you build native iOS and Android apps by writing JavaScript.

Introduction

Expo apps are React Native apps which contain the Expo SDK. The SDK is a native-and-JS library which provides access to the device’s system functionality (things like the camera, contacts, local storage, and other hardware). That means you don’t need to use Xcode or Android Studio, or write any native code, and it also makes your pure-JS project very portable because it can run in any native environment containing the Expo SDK.

Expo also provides UI components to handle a variety of use-cases that almost all apps will cover but are not baked into React Native core, e.g. icons, blur views, and more.

Finally, the Expo SDK provides access to services which typically are a pain to manage but are required by almost every app. Most popular among these: Expo can manage your assets for you, it can take care of Push Notifications for you, and it can build native binaries which are ready to deploy to the app store.
• Expo tools allow one to build, deploy, and quickly iterate on native Android, iOS
  • In addition to React Native components, one has access to the Expo SDK, a library that provides a wide variety of native APIs on iOS and Android

• expo init <projectName>
Deco

- android
- index.android.js
- index.ios.js
- ios
- node_modules
- package.json
React Native

Nuclide has built-in support for the React Native framework. React Native provides a set of components and extensions that allows you to easily write native iOS and Android applications using the Flow and JavaScript programming languages and the React UI library.

Features

- Features
  - Running applications
  - Metro
  - Command Line
- Support
  - Loading a React Native Project
  - Metro
  - Run the React Native Application
  - Element Inspector
  - Simulator Logs
  - Debugging

If your React Native apps are primarily written in Flow, you get all of its features within Nuclide, including Autocomplete, Code Diagnostics, etc.
Awesome React Native is an awesome style list that curates the best React Native libraries, tools, tutorials, articles and more. PRs are welcome!

Don't miss out! Subscribe to our weekly newsletter

Buzzing

![React Native Authentication in Depth](image)
![Create React Native App](image)
![Better List Views](image)

Have something that belongs under "Buzzing"? Drop me a line: dotan-at-paracode.com.

Recommended SDK for bug and crash reporting: Instabug
How to run your apps

- First option
  - cd <projectName>
  - react-native run-ios/android

- Second option
  - cd <projectName>
  - expo start
Few peculiarities

• App.js contains the first elements of the app

• index.js

```javascript
import { AppRegistry } from 'react-native';
import App from './App';
import { name as appName } from './app.json';

AppRegistry.registerComponent(appName, () => App);
```

• Deco puts everything directly in either index.ios.js or index.android.js
Components as JS classes

- We can define React components by using an ES class that extends React.Component

```javascript
import React from 'react';

class MyComponent extends React.Component {
  constructor() {
    super();
    // This constructor defines // the state of the component
  }

  render() {
    return (
      // This defines the reactive UI
      // associated with the component
    )
  }

  my_method() {
    // This is a custom method
  }
}
```
import React, { Component } from 'react';
import { Text, View } from 'react-native';

export default class HelloWorldApp extends Component {
  render() {
    return (
      <View>
        <Text>Hello world!</Text>
      </View>
    );
  }
}
Components

• HelloWorldApp is a new component
• RN apps use many components
  • Anything you see on the screen is some sort of component
• A component can be pretty simple
  • The only thing that is required is a render function to render it
• Most components can be customized when they are created
  • These creation parameters are called props
• Views are the most basic building block of React Native apps
  • Like a html <div>
• Views are primarily used for styling and layout of children elements

```
render() {
    return (
      <View style={styles.container}>
        <View style={styles.box} />
      </View>
    )
}
```
React Native Components

- They have the same characteristics of React ones, but they map to Native UI elements
  - `<View>` is a generic container of other UI elements
  - `<Text>` is a piece of text
  - `<Image>` is an image
  - `<Button>` is a button
  - `<FlatList>` is a list that renders only visible elements
  - `<Switch>` is a switch
<table>
<thead>
<tr>
<th>Component</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>View</td>
<td>The most fundamental component for building a UI.</td>
</tr>
<tr>
<td>Text</td>
<td>A component for displaying text.</td>
</tr>
<tr>
<td>Image</td>
<td>A component for displaying images.</td>
</tr>
<tr>
<td>TextInput</td>
<td>A component for inputting text into the app via a keyboard.</td>
</tr>
<tr>
<td>ScrollView</td>
<td>Provides a scrolling container that can host multiple components and views.</td>
</tr>
<tr>
<td>StyleSheet</td>
<td>Provides an abstraction layer similar to CSS stylesheets.</td>
</tr>
<tr>
<td>Component</td>
<td>Description</td>
</tr>
<tr>
<td>-----------</td>
<td>-------------</td>
</tr>
<tr>
<td>Button</td>
<td>A basic button component for handling touches that should render nicely on any platform.</td>
</tr>
<tr>
<td>Picker</td>
<td>Renders the native picker component on iOS and Android.</td>
</tr>
<tr>
<td>Slider</td>
<td>A component used to select a single value from a range of values.</td>
</tr>
<tr>
<td>Switch</td>
<td>Renders a boolean input.</td>
</tr>
<tr>
<td>FlatList</td>
<td>A component for rendering performant scrollable lists.</td>
</tr>
<tr>
<td>SectionList</td>
<td>Like FlatList, but for sectioned lists.</td>
</tr>
</tbody>
</table>

### User interface

- **ActivityIndicator**: Displays a circular loading indicator.
- **Alert**: Launches an alert dialog with the specified title and message.
- **CameraRoll**: Provides access to the local camera roll/gallery.
- **Clipboard**: Provides an interface for copying and pasting text.
- **Dimensions**: Provides an interface for getting device dimensions.
- **KeyboardAvoidingView**: Provides a view that moves out of the way of the virtual keyboard automatically.
- **Linking**: Provides a general interface to interact with both incoming and outgoing app links.
- **PixelRatio**: Provides access to the device pixel density.
- **RefreshControl**: This component is used inside a `ScrollView` to add pull to refresh functionality.
- **StatusBar**: Component to control the app status bar.
### Specific components

<table>
<thead>
<tr>
<th>Component</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>ActionSheetiOS</td>
<td>API to display an iOS action sheet or share sheet.</td>
</tr>
<tr>
<td>AlertiOS</td>
<td>Create an iOS alert dialog with a message or create a prompt for user input.</td>
</tr>
<tr>
<td>DatePickeriOS</td>
<td>Renders a date/time picker (selector) on iOS.</td>
</tr>
<tr>
<td>ImagePickerController</td>
<td>Renders a image picker on iOS.</td>
</tr>
<tr>
<td>NavigatoriOS</td>
<td>A wrapper around UIViewController, enabling you to implement a navigation stack.</td>
</tr>
<tr>
<td>ProgressViewiOS</td>
<td>Renders a UIProgressView on iOS.</td>
</tr>
<tr>
<td>PushNotificationiOS</td>
<td>Handle push notifications for your app, including permission handling and icon badge number.</td>
</tr>
<tr>
<td>SegmentedControlleriOS</td>
<td>Renders a UIBarButtonItem on iOS. Use with TabBarIOS.Item.</td>
</tr>
<tr>
<td>TabBariOS</td>
<td>Renders a UITabBarController on iOS.</td>
</tr>
<tr>
<td>BackHandler</td>
<td>Detect hardware button presses for back navigation.</td>
</tr>
<tr>
<td>DatePickerAndroid</td>
<td>Opens the standard Android date picker dialog.</td>
</tr>
<tr>
<td>DrawerLayoutAndroid</td>
<td>Renders a DrawerLayout on Android.</td>
</tr>
<tr>
<td>PermissionsAndroid</td>
<td>Provides access to the permissions model introduced in Android M.</td>
</tr>
<tr>
<td>ProgressBarAndroid</td>
<td>Renders a ProgressBar on Android.</td>
</tr>
<tr>
<td>TimePickerAndroid</td>
<td>Opens the standard Android time picker dialog.</td>
</tr>
<tr>
<td>ToastAndroid</td>
<td>Create an Android Toast alert.</td>
</tr>
<tr>
<td>ToolbarAndroid</td>
<td>Renders a Toolbar on Android.</td>
</tr>
<tr>
<td>ViewPagerAndroid</td>
<td>Container that allows to flip left and right between child views.</td>
</tr>
</tbody>
</table>
Properties (props)

- May be accessed from within the component's methods as `this.props`.
- Must not be altered from within the component's methods.
- A parent element may alter a child element's props at any time.
  - The child element will generally re-render itself to reflect its new configuration parameters.
Virtual DOM

State Change → Compute Diff → Re-render

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

export default class Bananas extends Component {
  render() {
    let pic = {
      uri: 'https://upload.wikimedia.org/wikipedia/commons/d/de/Bananavarieties.jpg'
    };
    return (<Image source={pic} style={{ width: 380, height: 220 }} />);
  }
}
import React, { Component } from 'react';
import { Text, View } from 'react-native';

class Greeting extends Component {
  render() {
    return (
      <View style={{ alignItems: 'center' }}>
        <Text>Hello {this.props.name}!</Text>
      </View>
    );
  }
}

export default class LotsOfGreetings extends Component {
  render() {
    return (
      <View style={{ alignItems: 'center', top: 200 }}>
        <Greeting name='Luciano' />
        <Greeting name='Giovanni' />
        <Greeting name='Mario' />
      </View>
    );
  }
}
The Basics

Getting Started
Learn the Basics
Props
State
Style
Height and Width
Layout with Flexbox
Handling Text Input
Handling Touches
Using a ScrollView
Using List Views
Networking
More Resources

Guides

Components and APIs
Platform Specific Code
Navigating Between Screens
Images
Animations
Accessibility
Implementing User Experience

Text

A React component for displaying text.

Text supports nesting, styling, and touch handling.

In the following example, the nested title and body text will inherit the fontFamily from styles.baseText, but the title provides its own additional styles. The title and body will stack on top of each other on account of the literal newlines:

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

export default class TextInANest extends Component {
  constructor(props) {
    super(props);
    this.state = {
      titleText: "Bird's Nest",
      bodyText: 'This is not really a bird nest.'
    };
  }

  render() {
    return (
      <Text style={styles.baseText}>
        <Text style={styles.titleText} onPress={this.onPressTitle}>
          {this.state.titleText}{'
        {'
      </Text>
      <Text numberOfLines={5}>
    
Bird's Nest This is not really a bird nest.
```
**URI vs URL**

- URI stands for uniform resource identifier
  - a URI can be a name, locator, or both for an online resource
- URL stands for uniform resource locator
  - a URL is just the locator
- URLs are a subset of URIs
• All core components accept a prop named style
  • Style names and values usually match how CSS works, but names are written using camel casing (e.g. backgroundColor rather than background-color)

• Prop style can be a plain old JavaScript object

• It is often cleaner to use StyleSheet.create to define several styles in one place

• Styles can "cascade" the way they do in CSS
import React, { Component } from 'react';
import { StyleSheet, Text, View } from 'react-native';

const styles = StyleSheet.create(
  {
    bigBlue: {
      color: 'blue',
      fontWeight: 'bold',
      fontSize: 30,
    },
    red: {
      color: 'red',
    },
  },
);

export default class LotsOfStyles extends Component {
  render() {
    return (
      <View style={{ alignItems: 'center', top: 200 }}>
        <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>
    );
  }
}
just red

just bigBlue

bigBlue, then red
red, then bigBlue

Example
Component size

- Fixed dimensions are set through fixed width and height to style
  - All dimensions in React Native are unitless, and represent density-independent pixels
- flex allows a component to expand and shrink dynamically
  - flex: 1 tells a component to fill all available space
  - The higher flex is, the higher the ratio of space a component will take compared to its siblings
  - A component can expand if its parent has dimensions greater than 0
    - If a parent does not have either a fixed width and height or flex, the parent will have dimensions of 0 and the flex children will not be visible
import React, { Component } from 'react';
import { View } from 'react-native';

export default class FixedDimensionsBasics extends Component {
  render() {
    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>
    );
  }
}
Fixed dimensions
Layout with Flexbox

• Flexbox is designed to provide a consistent layout on different screen sizes
• flexDirection determines the primary axis of its layout
  • row or column, the default is column
• justifyContent determines the distribution of children along the primary axis
  • Available options are flex-start, center, flex-end, space-around, space-between and space-evenly
• alignItems determines the alignment of children along the secondary axis
  • Available options are flex-start, center, flex-end, and stretch
The Basics

Getting Started
Learn the Basics
Props
State
Style
Height and Width
Layout with Flexbox
Handling Text Input
Handling Touches
Using a ScrollView
Using List Views
Networking
More Resources

Guides

Components and APIs
Platform Specific Code
Navigating Between Screens
Images
Animations
Accessibility
Improving User Experience

Layout Props

Props

- `alignContent`
- `alignItems`
- `alignSelf`
- `aspectRatio`
- `borderBottomWidth`
- `borderEndWidth`
- `borderLeftWidth`
- `borderRightWidth`
- `borderStartWidth`
- `borderTopWidth`
- `borderWidth`
- `bottom`
- `direction`
- `display`
- `end`
- `flex`
- `flexBasis`
- `flexDirection`
- `flexGrow`
# Layout Styles

<table>
<thead>
<tr>
<th>Property</th>
<th>Type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>flex</td>
<td>number</td>
<td>What proportion of the available space should this element expand to fill? 0 means the element doesn’t expand.</td>
</tr>
<tr>
<td>flexDirection</td>
<td>string</td>
<td>Do you want a vertical (column) or horizontal (row) layout? This choice determines the orientation of the primary axis of your layout. One of: row, column.</td>
</tr>
<tr>
<td>justifyContent</td>
<td>string</td>
<td>How should the content be distributed along the primary axis of your layout? Should it be at the start, the center, the end, or spaced evenly? One of: flex-start, center, flex-end, space-around, space-between.</td>
</tr>
<tr>
<td>alignItems</td>
<td>string</td>
<td>How should the content be aligned along the secondary axis of your layout? (If the primary axis is row, then the secondary axis is column, and vice versa) Should content be aligned at the start, the center, the end, or stretched to fill? One of: flex-start, center, flex-end, stretch.</td>
</tr>
<tr>
<td>width</td>
<td>number</td>
<td>A fixed width</td>
</tr>
<tr>
<td>height</td>
<td>number</td>
<td>A fixed height</td>
</tr>
<tr>
<td>margin</td>
<td>number</td>
<td>How much space should separate this element from other adjacent elements? For more granular control, use the properties: marginTop, marginBottom, marginLeft, marginRight.</td>
</tr>
<tr>
<td>padding</td>
<td>number</td>
<td>How much padding should be within this element before its children are shown? For more granular control, use the properties: paddingTop, paddingBottom, paddingLeft, paddingRight.</td>
</tr>
</tbody>
</table>
## Visual Styles

<table>
<thead>
<tr>
<th>Property</th>
<th>Type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>backgroundColor</td>
<td>string</td>
<td>The background color</td>
</tr>
<tr>
<td>borderWidth</td>
<td>number</td>
<td>The border width. For more granular control, use the properties: borderTopWidth, borderBottomWidth, borderLeftWidth, borderRightWidth.</td>
</tr>
<tr>
<td>borderColor</td>
<td>string</td>
<td>The color of the border. For more granular control, use the properties: borderTopColor, borderBottomColor, borderLeftColor, borderRightColor.</td>
</tr>
<tr>
<td>borderRadius</td>
<td>number</td>
<td>Rounded edges</td>
</tr>
<tr>
<td>opacity</td>
<td>number</td>
<td>How transparent should this element be? 0 means fully transparent, 1 means fully opaque.</td>
</tr>
</tbody>
</table>
import React, { Component } from 'react';
import { View } from 'react-native';

export default class FlexDimensionsBasics extends Component {
  render() {
    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>
    );
  }
}
Flex dimensions
Another example

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

export default class AlignItemsBasics extends Component {
    render() {
        return (
            <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>
        );
    }
};
```
Another example
Rows and columns

Lake Oeschinensee lies at the foot of the Blüemlisalp in the Bernese Alps. Situated 1,578 meters above sea level, it is one of the larger Alpine Lakes. A gondola ride from Kandersteg, followed by a half-hour walk through pastures and pine forest, leads you to the lake, which warms to 20 degrees Celsius in the summer. Activities enjoyed here include rowing, and riding the summer toboggan run.
Ske|to Code

BuilderX is a browser based design tool which codes React Native & React for you

Open BuilderX

By using this site you agree with our Privacy Policy  

×
State

- this.state can be accessed from within component methods
- Unlike props, parent elements cannot access a child's state
- Never directly assign a specific key within the state object
  - e.g. this.state.foo = 'bar', but method this.setState()
- Any keys in the object passed to the method will be merged into (and override any existing keys in) this.state
State

- `this.setState` merges the current state with the given one
  - It intersects the states and it changes properly with the current state
- `this.setState` is not synchronous
  - React may batch multiple state changes to guarantee better performance
import React, { Component } from 'react';
import { Text, View } from 'react-native';

class Blink extends Component {
    constructor(props) {
        super(props);
        // state object
        this.state = { isShowingText: true };  
    }
    render() {
        if (!this.state.isShowingText) {
            return null;
        }
        return (<
            Text>
            {this.props.text}
        </Text>);
    }  
}

componentDidMount() {
    // Toggle the state every second
    setInterval(() => {
        this.setState(previousState => {
            { isShowingText: !previousState.isShowingText }
        }), 1000);
    }  
}
export default class BlinkApp extends Component {
  render() {
    return (
      <View style={{ alignItems: 'center', top: 200 }}>)
        <Blink text='I love to blink' />
        <Blink text='Yes blinking is so great' />
        <Blink text='Why did they ever take this out of HTML' />
        <Blink text='Look at me look at me look at me' />
    </View>
  }
}
Text Input

- TextInput is a basic component that allows the user to enter text
  - Prop onChangeText takes a function called every time the text is changed
  - Prop onSubmitEditing takes a function called when the text is submitted
import React, { Component } from 'react';
import { Text, TextInput, View } from 'react-native';

export default class PizzaTranslator extends Component {
  constructor(props) {
    super(props);
    this.state = { text: '' };  
  }

  render() {
    return (
      <View style={{ padding: 10, top: 200 }}>
        <TextInput
          style={{ height: 40 }}
          placeholder="Type here to translate!"
          onChangeText={({text}) => this.setState({ text })}
          value={this.state.text}
        />
        <Text style={{ padding: 10, fontSize: 42 }}>
          {this.state.text.split(' ').map((word) => word && '🍕').join(' ')}
        </Text>
      </View>
    );
  }
}
Handling Touches

• RN provides components to handle all sorts of common gestures

• Button provides a basic button component that is rendered nicely on all platforms

```jsx
<Button
  onPress={() => {
    Alert.alert('You tapped the button!');
  }}
  title="Press Me"
/>
```

• This renders a blue label on iOS, and a blue rounded rectangle with white text on Android
  • We can specify a prop color to change its color
```javascript
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>
    )
  }
}
```
const styles = StyleSheet.create({
  container: {
    flex: 1,
    justifyContent: 'center',
  },
  buttonContainer: {
    margin: 20
  },
  alternativeLayoutButtonContainer: {
    margin: 20,
    flexDirection: 'row',
    justifyContent: 'space-between'
  }
});

<View style={styles.alternativeLayoutButtonContainer}>
  <Button
    onPress={this._onPressButton}
    title="This looks great!"
  />
  <Button
    onPress={this._onPressButton}
    title="OK!"
    color="#841584"
  />
</View>
**Touchables**

- We can build our own button using any of the "Touchable" components provided by RN
  - They provide the capability to capture tapping gestures, and can display feedback when a gesture is recognized
  - They do not provide any default styling
- Long presses can be handled by passing a function to prop `onLongPress`
We can use

- **TouchableHighlight** anywhere you would use a button or web link
- **TouchableNativeFeedback** on Android to display ink surface reaction ripples that respond to the user's touch
- **TouchableOpacity** to provide feedback by reducing the opacity of the button
- **TouchableWithoutFeedback** to handle a tap gesture but without displaying any feedback
Example

```javascript
<TouchableOpacity
  onPress={() => onSelect(id)}
  style={[
    styles.item,
    { backgroundColor: selected ? '#6e3b6e' : '#f9c2ff' },
  ]}
>
  <Text style={styles.title}>{title}</Text>
</TouchableOpacity>
```
ScrollView

- ScrollView is a generic scrolling container that can host multiple components and views
  - Scrollable items need not be homogeneous
  - We can scroll both vertically and horizontally (by setting the horizontal property)
- ScrollViews can be configured to allow paging through views using swiping gestures by using prop pagingEnabled
- A ScrollView with a single item can be used to allow the user to zoom content
  - Props maximumZoomScale and minimumZoomScale allow the user to use pinch and expand gestures to zoom in and out
import React, { Component } from 'react'
import { SafeAreaView, ScrollView, Text, StyleSheet } from 'react-native'

export default class App extends Component {
  render() {
    return (
      <SafeAreaView style={{styles.container}}>
        <ScrollView style={{styles.scrollView}}>
          <Text style={{styles.text}}>
            Lorem ipsum dolor sit amet, consectetur adipiscing elit, sed do eiusmod tempor incididunt ut labore et dolore magna aliqua. Ut enim ad minim veniam, quis nostrud exercitation ullamco laboris nisi ut aliquip ex ea commodo consequat. Duis aute irure dolor in reprehenderit in voluptate velit esse cillum dolore eu fugiat nulla pariatur. Excepteur sint occaecat cupidatat non proident, sunt in culpa qui officia deserunt mollit anim id est laborum.
          </Text>
        </ScrollView>
      </SafeAreaView>
    );
  }
}
Example

```javascript
const styles = StyleSheet.create({
  container: {
    flex: 1,
    marginTop: 10,
  },
  scrollView: {
    backgroundColor: 'yellow',
    marginHorizontal: 20,
  },
  text: {
    fontSize: 40,
  },
});
```
List Views

• FlatList displays a scrolling list of changing, but similarly structured, data
  • It works well for long lists of data, where the number of items might change over time
  • It only renders elements that are currently showing on the screen, not all the elements at once
• It requires two props
  • data is the source of information for the list
  • renderItem takes one item from the source and returns a formatted component to render
• SectionList renders a set of data broken into logical sections
  • Maybe with section headers
import React, { Component } from 'react';
import { FlatList, StyleSheet, Text, View } from 'react-native';

export default class FlatListBasics extends Component {
  render() {
    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,
    }
})
Another example

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

export default class SectionListBasics extends Component {
  render() {
    return (
      <SafeAreaView style={styles.container}>
        <View>
          <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>
      </SafeAreaView>
    );
  }
}
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,
  },
})
Lifecycle

- Components are instantiated, mounted, rendered, and eventually updated, unmounted, and destroyed
- The lifecycle helps manage the complexity of different platform APIs (iOS, Android) by providing a simple, consistent abstraction layer
- The lifecycle also allows one to optionally execute custom code at each step for more fine-grained control of the rendering
Mounting Cycle

- **constructor(object props)** instantiates the component
  - The parameters to the constructor are the element's initial props
  - One can optionally specify an initial state for the element by assigning an object to this.state.

- **componentWillMount()** is invoked only once, before rendering occurs for the first time

- **render()** -> React Element returns a React Element to render (or null, to render nothing)

- **componentDidMount()** is invoked only once, after rendering occurs for the first time
  - The native UI may be accessed through this.refs
  - Used to make async API calls or execute delayed code with setTimeout
Updating Cycle

- `componentWillReceiveProps(object nextProps)` can call `this.setState()` to update the internal state before the render method is called.
- `shouldComponentUpdate(object nextProps, object nextState) -> boolean` checks if either props or state have been modified.
- `componentWillUpdate(object nextProps, object nextState)` is invoked after the decision has been made to re-render.
- `componentDidUpdate(object prevProps, object prevState)` is invoked after re-rendering occurs.
import React, { Component } from 'react'
import { View, Text, StyleSheet } from 'react-native'

class Counter extends Component {

  state = {count: 0}

  componentDidMount() {
    setInterval(() => {
      this.setState({count: this.state.count + 1})
    }, 1000)
  }

  render() {
    const {count} = this.state
    const {color, size} = this.props

    return (
      <Text style={{color, fontSize: size}}>
        {count}
      </Text>
    )
  }
}
Example (II)

```javascript
export default class App extends Component {
  render() {
    return (
      <View style={{styles.container}}>
        <Counter color='lightblue' size={16} />  
        <Counter color='skyblue' size={32} />  
        <Counter color='steelblue' size={80} />  
        <Counter color='darkblue' size={140} />  
      </View>
    )
  }
}

const styles = StyleSheet.create({
  container: {
    flex: 1,
    justifyContent: 'center',
    alignItems: 'center',
  },
});
```
Networking

• RN provides the Fetch API for networking
  • Similar to XMLHttpRequest or other networking APIs
• Networking is an inherently asynchronous operation
  • Fetch methods return a Promise that makes it straightforward to write code that works asynchronously

```javascript
function getMoviesFromApiAsync() {
  return fetch('https://facebook.github.io/react-native/movies.json')
    .then((response) => response.json())
    .then((responseJson) => {
      return responseJson.movies;
    })
    .catch((error) => {
      console.error(error);
    });
}
```
Fetch API

- `const response = await fetch(uri)`
  - Send a GET request to uri, returning a promise which represents a Response object

- `const json = await response.json()`
  - Parse the body of the response asynchronously as JSON

- `const text = await response.text()`
  - Get the body of the response as text
ES2017 async/await syntax

```javascript
async function getMoviesFromApi() {
  try {
    let response = await fetch('https://facebook.github.io/react-native/movies.json',
     );
    let responseJson = await response.json();
    return responseJson.movies;
  } catch (error) {
    console.error(error);
  }
}
```
import React from 'react';
import { FlatList, ActivityIndicator, Text, View, SafeAreaView } from 'react-native';

export default class FetchExample extends React.Component {

    constructor(props) {
        super(props);
        this.state = { isLoading: true }
    }

    componentDidMount() {
        return fetch('https://facebook.github.io/react-native/movies.json')
            .then((response) => response.json())
            .then((responseJson) => {
                this.setState({
                    isLoading: false,
                    dataSource: responseJson.movies,
                }, function () {
                });
            })
            .catch((error) => {
                console.error(error);
            });
    }
}
render() {
  if (this.state.isLoading) {
    return (  
      <View style={{ flex: 1, padding: 20 }}>
        <ActivityIndicator />
      </View>
    )
  }

  return (  
    <SafeAreaView style={{ flex: 1, paddingTop: 20 }}>
      <FlatList
        data={this.state.dataSource}
        renderItem={({ item }) => <Text>{item.title}, {item.releaseYear}</Text>}
        keyExtractor={({ id }, index) => id}
      />  
    </SafeAreaView>
  );
}
Other options

- The XMLHttpRequest API is built in
  - We can use third party libraries, such as frisbee or axios, that depend on it, or we can use the API directly
- RN also supports WebSockets, a protocol that provides full-duplex communication channels over a single TCP connection
React Native provides a module that detects the platform on which the app is running.

```javascript
import { Platform, StyleSheet } from 'react-native';

const styles = StyleSheet.create({
  height: Platform.OS === 'ios' ? 200 : 100,
});

const styles = StyleSheet.create({
  container: {
    flex: 1,
    ...Platform.select({
      ios: {
        backgroundColor: 'red',
      },
      android: {
        backgroundColor: 'blue',
      },
    })),
  },
});
```
Platform-specific code

- When your platform-specific code is more complex, you should consider splitting the code out into separate files
  - BigButton.ios.js
  - BigButton.android.js

- RN detects when a file has a .ios. or .android. extension and loads the relevant platform file when required from other components.

```javascript
const BigButton = require('./BigButton');
```
Navigator

- Managing the presentation of, and transition between, multiple screens is typically handled by what is known as a navigator
- Navigators are in charge of the transition between multiple screens
- React Navigation provides an easy to use navigation solution, with the ability to present common stack navigation and tabbed navigation patterns

- In your project
  - npm install --save react-navigation
  - npm install --save react-native-gesture-handler
  - npm install --save react-navigation-stack
import React, { Component } from 'react';
import { createAppContainer } from 'react-navigation';
import { createStackNavigator } from 'react-navigation-stack';

import Settings from './screens/Settings';
import Home from './screens/Home';

const AppNavigator = createStackNavigator(
    {
        SettingScreen: { screen: Settings },
        HomeScreen: { screen: Home }
    });

const AppContainer = createAppContainer(AppNavigator);

export default class App extends Component {
    render() {
        return (<AppContainer />);
    }
}
import React, { Component } from 'react';
import { View, Text, StyleSheet } from 'react-native';

export default class Home extends Component {
  render() {
    return (
      <View style={styles.container}>
        <Text>This is the home screen</Text>
      </View>
    )
  }
}

const styles = StyleSheet.create({
  container: {
    flex: 1,
    justifyContent: 'center',
    alignItems: 'center',
  },
})
import React, { Component } from 'react';
import { View, Text, Button, StyleSheet } from 'react-native';

export default class Settings extends Component {

    render() {
        return (
            <View style={styles.container}>
                <Text>This is the Settings screen</Text>
                <Button onPress={() => this.props.navigation.navigate('HomeScreen')} title="Home"/>
            </View>
        )
    }
}

const styles = StyleSheet.create({
    container: {
        flex: 1,
        justifyContent: 'center',
        alignItems: 'center',
    },
})
Result
Expo tab template

Development mode is enabled: your app will be slower but you can use useful development tools. [Learn more]

Get started by opening
screens/HomeScreen.js

Change this text and your app will automatically reload.

Help, it didn't automatically reload!

This is a tab bar. You can edit it in:
navigation/MainTabNavigator.js
# Data management

<table>
<thead>
<tr>
<th>Option</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Component State</strong></td>
<td>Storing data in the <code>state</code> of your components is the simplest way to manage data throughout your app. Every time the user interacts with the app, update the state of a component, or use function <code>props</code> to update the state in parent components. It can be advantageous to maintain all state in the root component of the app, so that all state is managed in one place, making it easier to reason about the app and enabling you to easily switch to Redux if necessary.</td>
</tr>
<tr>
<td><strong>Redux</strong></td>
<td><em>Redux</em> is the most common choice of library for managing data in medium and large apps. Redux provides a <code>store</code> object which maintains the application state, and can notify your React components when state changes. Redux was designed with React in mind, and has official React bindings: <em>React Redux</em>. There are additional tools around Redux to provide: control over asynchronous events, data persistence (for offline usage, etc), and more powerful debugging.</td>
</tr>
<tr>
<td><strong>GraphQL</strong></td>
<td><em>GraphQL</em> is a query language for easily fetching and updating remote data on any GraphQL-compliant server. GraphQL is designed to simplify communication between server and client: compared to traditional REST APIs, GraphQL typically requires fewer roundtrips (better performance) and fewer endpoints (enabling faster iteration cycles). To use GraphQL, you'll need to configure your server to handle GraphQL queries, or use a hosted solution like <em>Graphcool</em>. It's common to use Redux along with GraphQL, if you have a significant amount of client-side data in addition to your server-side data.</td>
</tr>
<tr>
<td><strong>Realm</strong></td>
<td>The <em>Realm</em> library uses a custom database, written from scratch in C to be compatible on both iOS and Android. It's suitable for apps with a tremendous amount of data (10,000+ records). If an app can't fit all its data in RAM, using a database makes it easy to page records in and out. Additionally, searching becomes extremely fast thanks to database indexes, and offline usage becomes trivial.</td>
</tr>
<tr>
<td><strong>Firebase</strong></td>
<td><em>Firebase</em> is a cloud service for storing and syncing data. Firebase is a great option when you don't have a backend or existing API to work with, since you can build a mobile app without setting up any infrastructure. Firebase's offers impressive realtime synchronization of data between (potentially thousands of) clients. The service also provides a web dashboard for viewing/modifying your data, user accounts &amp; authentication, analytics, crash reporting, and more.</td>
</tr>
</tbody>
</table>
Most medium and large React Native apps use Redux for managing data and state throughout the application.

- An application will create a single Redux store to hold all data and state.
- We can view the state of the store by calling `store.getState()`.
- We call `store.dispatch(action)` to dispatch actions into the store.
## Persistence

### Common Options for Client-side Persistence

<table>
<thead>
<tr>
<th>Option</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>AsyncStorage</td>
<td>AsyncStorage is a simple, low-level API for a key-value store. You can easily store and retrieve key-value pairs.</td>
</tr>
<tr>
<td>Redux Persist</td>
<td>Redux Persist is a library for automatically persisting the state of your Redux store to AsyncStorage on store changes, and rehydrating the store with this saved state when the app launches.</td>
</tr>
<tr>
<td>Realm</td>
<td>The Realm library uses a custom database, written from scratch in C to be compatible on both iOS and Android. We covered Realm separately under Data Management, but if you're using Realm for managing your data, you get persistence for free!</td>
</tr>
</tbody>
</table>