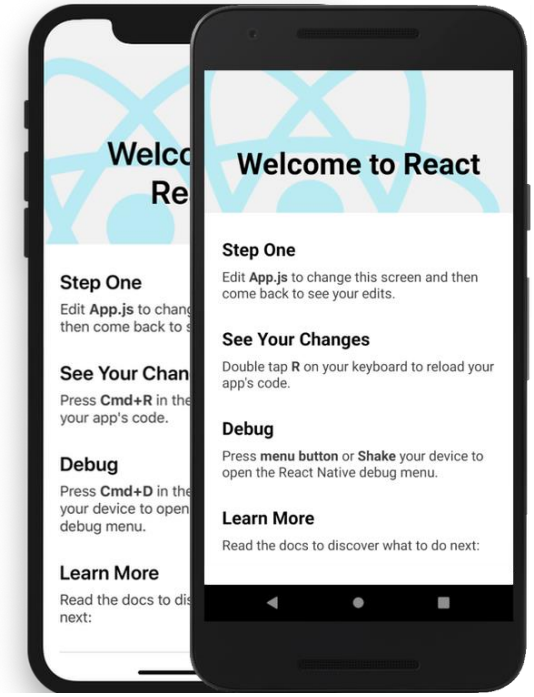
Introduction

Create native apps for Android and iOS using React



React Native combines the best parts of native development with React, a best-in-class JavaScript library for building user interfaces.

Use a little—or a lot. You can use React Native today in your existing Android and iOS projects or you can create a whole new app from scratch.

Written in JavaScript—rendered with native code

React primitives render to native platform UI, meaning your app uses the same native platform APIs other apps do.

Many platforms, one React. Create platform-specific versions of components so a single codebase can share code across platforms. With React Native, one team can maintain two platforms and share a common technology—React.

Native Development For Everyone

React Native lets you create truly native apps and doesn't compromise your users' experiences. It provides a core set of platform agnostic native components like View, Text, and Image that map directly to the platform's native UI building blocks.

Seamless Cross-Platform

React components wrap existing native code and interact with native APIs via React's declarative UI paradigm and JavaScript. This enables native app development for whole new teams of developers, and can let existing native teams work much faster.

Introduction to Expo

Expo is a framework and a platform for universal React applications. It is a set of tools and services built around React Native and native platforms that help you develop, build, deploy, and quickly iterate on iOS, Android, and web apps from the same JavaScript/TypeScript codebase.

Tools and Services

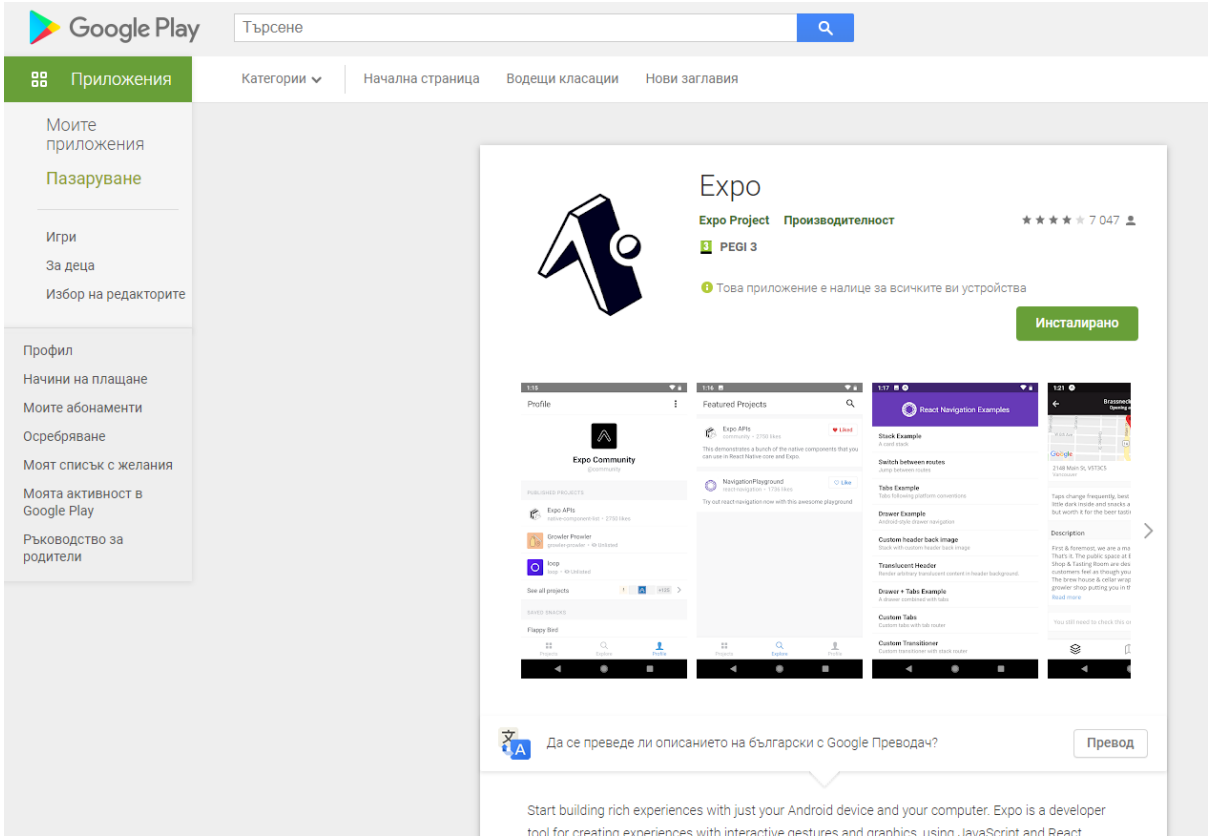
Expo tools and services empower you to create incredible apps using Expo CLI and the Expo SDK, the Expo Go app, our cloud build and submission services, and Expo Snack.

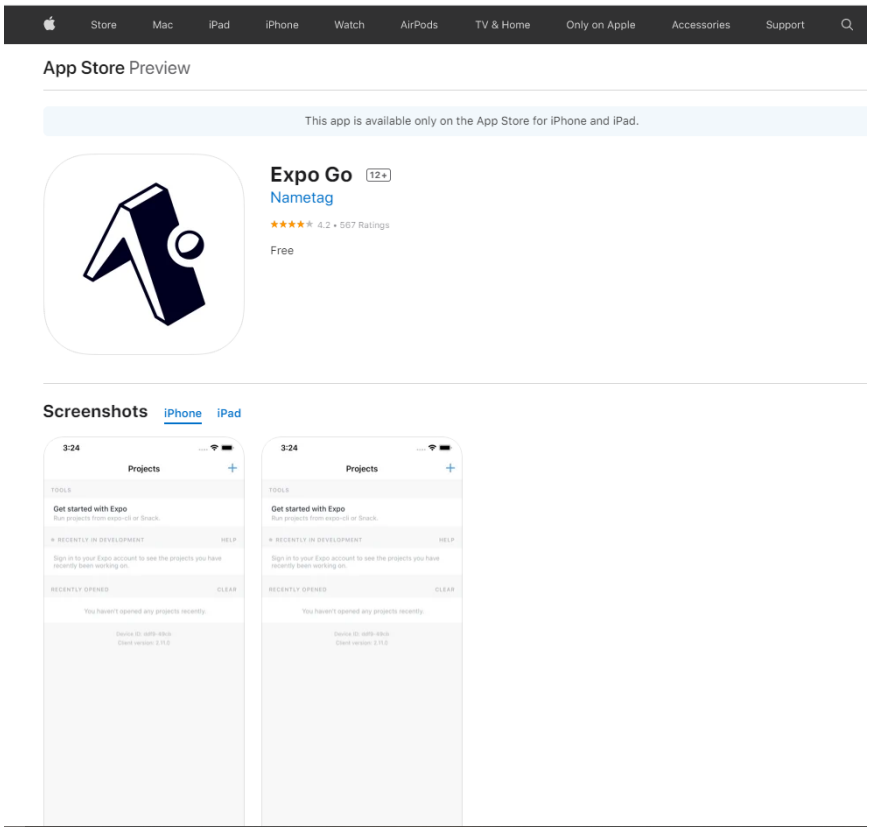
Develop with Expo's CLI and SDK

Install Expo CLI to create and run your project. Then use our SDK's APIs and components to build a fully-featured application.

Run your project with Expo Go

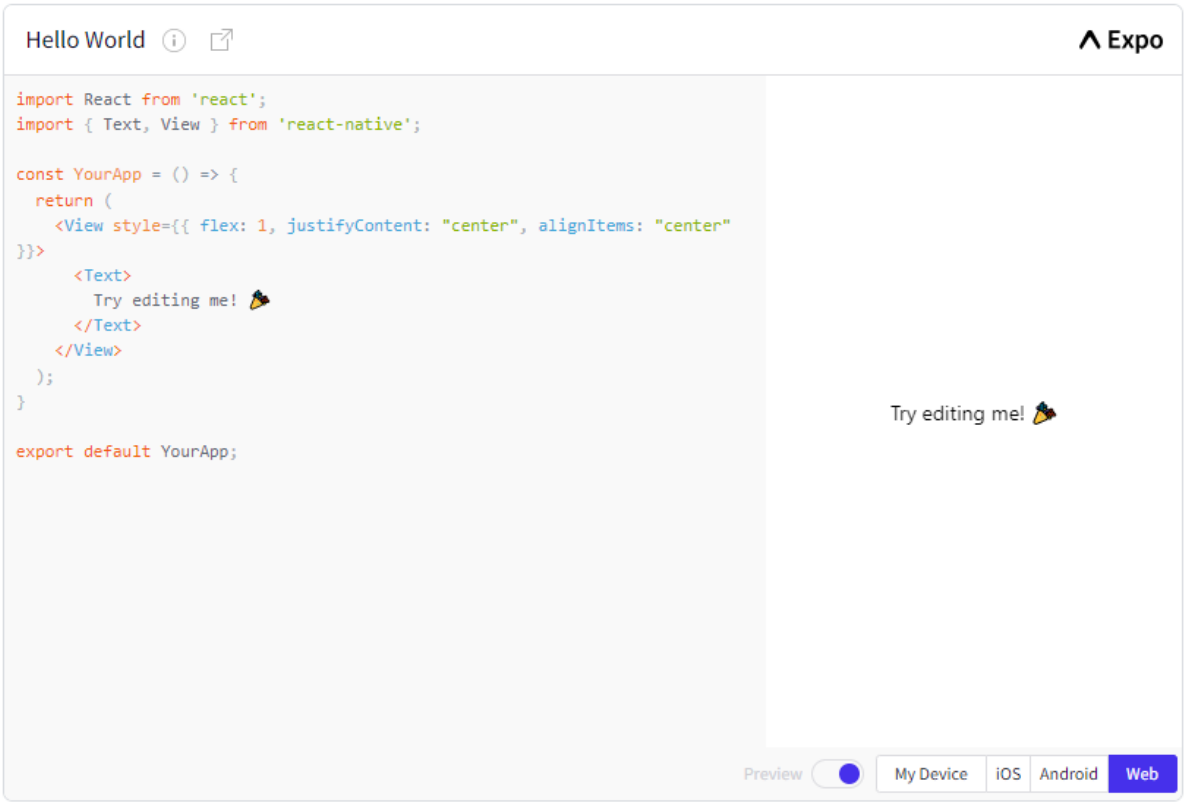
Run your project on your own device in seconds with Expo Go.


Google Play
Търсене
Приложения
Категории
Начална страница
Водещи класации
Нови заглавия
Моите приложения
Пазаруване
Игри
За деца
Избор на редакторите
Профил
Начини на плащане
Моите абонаменти
Осребряване
Моят списък с желания
Моята активност в Google Play
Ръководство за родители
Expo
Expo Project
Производителност
7 047
PEGI 3
Това приложение е налице за всичките ви устройства
Инсталирано
Да се преведе ли описанието на български с Google Преводач?
Превод
Start building rich experiences with just your Android device and your computer. Expo is a developer


Store
Mac
iPad
iPhone
Watch
AirPods
TV & Home
Only on Apple
Accessories
Support
App Store Preview
This app is available only on the App Store for iPhone and iPad.
Expo Go
12+
Nametag
4.2 • 567 Ratings
Free
Screenshots
iPhone
iPad
Projects
Get started with Expo

Interactive examples

This introduction lets you get started immediately in your browser with interactive examples like this one:



https://reactnative.dev/docs/getting-started

The above is a Snack Player. It’s a handy tool created by Expo to embed and run React Native projects and share how they render in platforms like Android and iOS. The code is live and editable, so you can play directly with it in your browser. Go ahead and try changing the "Try editing me!" text above to "Hello, world!"

Function Components and Class Components

With React, you can make components using either classes or functions. Originally, class components were the only components that could have state. But since the introduction of React's Hooks API, you can add state and more to function components.

Hooks were introduced in React Native 0.59., and because Hooks are the future-facing way to write your React components, we wrote this introduction using function component examples. Where useful, we also cover class components under a toggle like s

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

const HelloWorldApp = () => {
  return (
    <View style={{
       flex: 1,
       justifyContent: 'center',
       alignItems: 'center'
     }}>
     <Text>Hello, world!</Text>
    </View>
  );
}

export default HelloWorldApp;
```

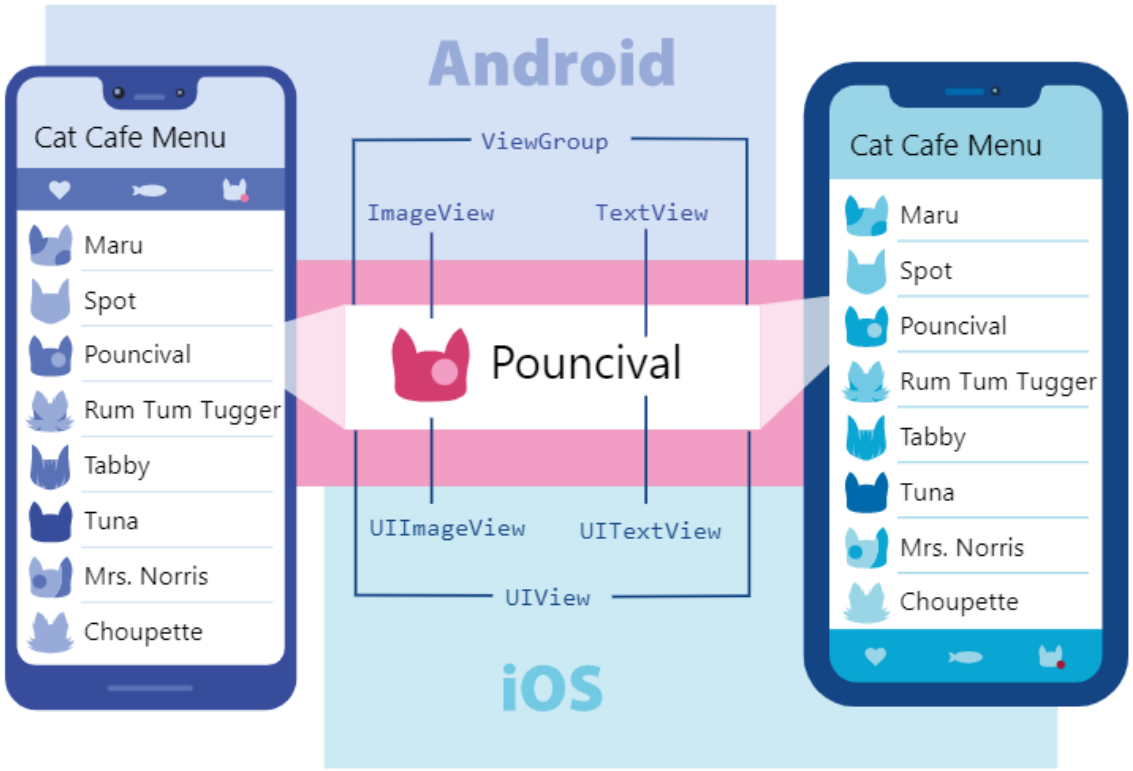
```
import React, { Component } from 'react';
import { Text, View } from 'react-native';

class HelloWorldApp extends Component {
  render() {
    return (
      <View style={{
         flex: 1,
         justifyContent: "center",
         alignItems: "center"
       }}>
       <Text>Hello, world!</Text>
      </View>
    );
  }
}

export default HelloWorldApp;
```

Core Components and Native Components

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. You can learn more about React in the next section. But first, let's cover how components work in React Native.

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. Even the smallest visual elements of an app, like a line of text or a button, are kinds of views. Some kinds of views can contain other views. It's views all the way down!



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

Native Components

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

React Native also lets you build your own Native Components for Android and iOS to suit your app's unique needs. We also have a thriving ecosystem of these **community-contributed components.** Check out Native Directory to find what the community has been creating.

Core Components

React Native has many Core Components for everything from form controls to activity indicators. You can find them all documented in the API section. 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	A container that supports layout with flexbox, style, some touch handling, and accessibility controls

REACT NATIVE UI COMPONENT	ANDROID VIEW	IOS VIEW	WEB ANALOG	DESCRIPTION
<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>		A generic scrolling container that can contain multiple components and views
<TextInput>	<EditText>	<UITextField>	<input type="text">	Allows the user to enter text

```jsx
import React from 'react';

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

const App = () => {
  return (
    <ScrollView>
      <Text>Some text</Text>
      <View>
        <Text>Some more text</Text>
        <Image
          source={{
            uri: '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>
  );
}

export default App;
```

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, { useState } from 'react';

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

const Translator = () => {
  const [text, setText] = useState('');
  return (
    <View style={{padding: 10}}>
      <TextInput
        style={{height: 40}}
        placeholder="Type here to translate!"
        onChangeText={newText => setText(newText)}
        defaultValue={text}
      />
      <Text style={{padding: 10, fontSize: 42}}>
        {text}
      </Text>
    </View>
  );
}

export default Translator;
```

Using a ScrollView

The ScrollView is a generic scrolling container that can contain multiple components and views. The scrollable items can be heterogeneous, and you can scroll both vertically and horizontally (by setting the horizontal property).

This example creates a vertical ScrollView with both images and text mixed together.

https://reactnative.dev/docs/using-a-scrollview

Using List Views

React Native provides a suite of components for presenting lists of data. Generally, you'll want to use either FlatList or SectionList.

The FlatList component displays a scrolling list of changing, but similarly structured, data. FlatList works well for long lists of data, where the number of items might change over time. Unlike the more generic ScrollView, the FlatList only renders elements that are currently showing on the screen, not all the elements at once.

The FlatList component requires two props: data and renderItem. data is the source of information for the list. renderItem takes one item from the source and returns a formatted component to render.

This example creates a basic FlatList of hardcoded data. Each item in the data props is rendered as a Text component. The FlatListBasics component then renders the FlatList and all Text components.

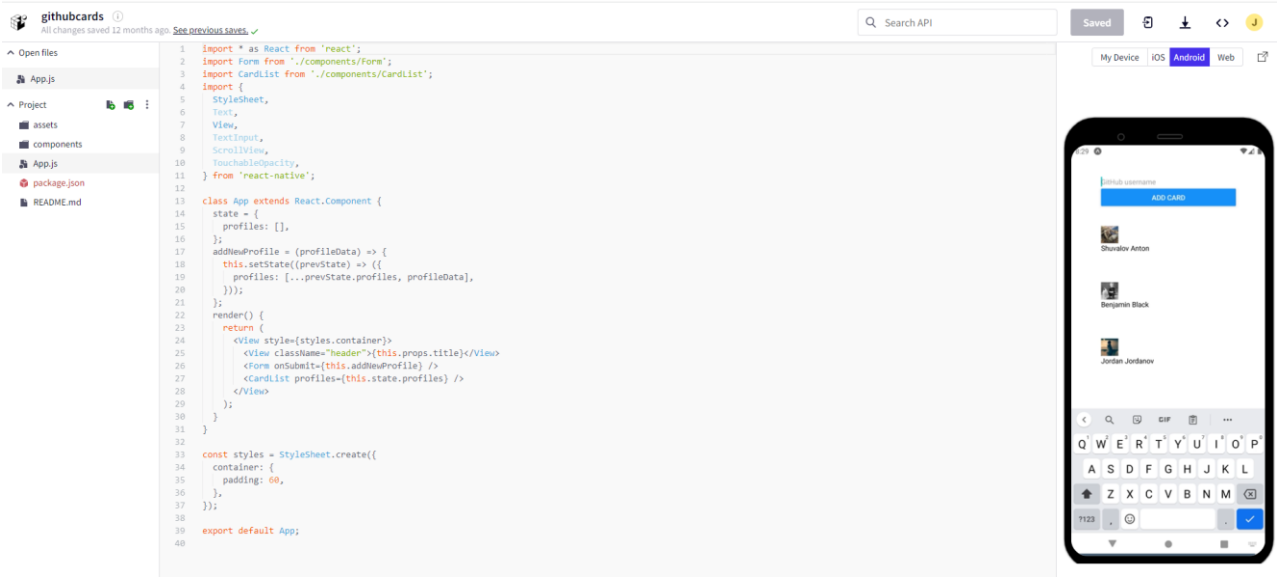
https://reactnative.dev/docs/using-a-listview

Github Cards in React Native

Re-write Github Cards project https://github.com/profjordanov/githubcards

Use Snack - React Native in the browser (expo.dev)

Reference: https://github.com/profjordanov/hybrid-app-development/blob/master/07.%20React%20Native/githubcards.txt



RNClickCounter app

Create and Run a React Native Project

The steps to run the project is the same on all operating systems:

1) Create a React Native project:

npx create-expo-app RNClickCounter

2) Running your React Native application

Install the Expo Go app on your iOS or Android phone and connect to the same wireless network as your computer. On Android, use the Expo Go app to scan the QR code from your terminal to open your project. On iOS, use the built-in QR code scanner of the default iOS Camera app.

Commands:

- cd RNClickCounter
- npx expo install react-native-web@~0.18.10 react-dom@18.2.0 @expo/webpack-config@^18.0.1
- npx expo start

3) Modifying

Open App.js in your text editor of choice and edit some lines. The application should reload automatically once you save your changes.

React Native uses some common mobile components like Button, View, Text, etc., along with React Native-specific components like SafeAreaView and StyleSheet. Let's go though some basic components to build the click counter app:

- SafeAreaView adds the required padding for camera-notches/sensor-housing and reflects the area that is not covered by any of the top views like toolbar, navigation, etc.
- Text displays text on the screen. It is similar to UILabel, TextView, or <p> tag.
- View is a basic UI container element with flexbox layout support. The Native equivalents of view are UIView, View, or div tag.
- Button represents the Native platform-specific button with platform-specific style.
- StyleSheet is used to define the style attributes for elements that will be mapped to Native-style values.
- useState is a React hook that is used to maintain a state (stored values) in a functional component. This is used in the App.js functional component to keep the track of the counter variable's state. The counter variable should be modified by the callback method setCount, and returned by useState.
- export default App is used to allow other components to import the App component. There can be only one default export in a file.
- flex: 1 is used to define the CSS3 flexbox style responsive layout vertically.
- ~~React$Node represents a type of React node (from flow type check) whose value can be a ReactChild, ReactFragment, ReactPortal, boolean, null, number, or string.~~

4) Steps to Implement Click Counter

Follow the below steps to implement click counter in the `App.js` component:

1. Implement `react-hook` to store the updated value of `count`.
   The `setCount` method will be used to update the value of `count`:

```
2.   const [count, setCount] = useState(0);
```

2. Implement callback functions to increment/decrement the value of `count`:

```
3. const counterPlus = () => {
4. 2    setCount(count + 1 <= Number.MAX_SAFE_INTEGER ? count + 1 : count)
5. 3  }
6. 4
7. 5  const counterMinus = () => {
8. 6    setCount(count - 1 >= Number.MIN_SAFE_INTEGER ? count - 1 : count)
9. 7  }
```

3. Create the style object to center the views inside container elements, and design details for other views such as Button and Text:

```
4. const styles = StyleSheet.create({
5. 2  container: {
6. 3    flex: 1,
7. 4    justifyContent: 'center',
8. 5    alignItems: 'center',
9. 6    backgroundColor: '#e6e6fa',
10. 7  },
11. 8  textConter: {
12. 9    fontSize: 28,
13. 10    color: '#000',
14. 11  },
15. 12  buttonStyle: {
16. 13    width: "80%",
17. 14    margin: 10,
18. 15  }
19. 16});
```

4. Now apply the style to the elements and add the listener on the button to complete the implementation:

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

export default function App() {

  const [count, setCount] = useState(0);

  const counterPlus = () => {
    setCount(count + 1 <= Number.MAX_SAFE_INTEGER ? count + 1 : count)
  }

  const counterMinus = () => {
    setCount(count - 1 >= Number.MIN_SAFE_INTEGER ? count - 1 : count)
  }

  return (
    <>
      <StatusBar barStyle="dark-content" />
      <SafeAreaView style={styles.container}>
        <Text style={styles.textConter} >{count}</Text>
        <View style={styles.buttonStyle}>
          <Button
            onPress={counterPlus}
            title='+' />
        </View>
        <View style={styles.buttonStyle}>
          <Button
            onPress={counterMinus}
            title='-' />
        </View>
      </SafeAreaView>
    </>
  );
};

const styles = StyleSheet.create({
  container: {
    flex: 1,
    justifyContent: 'center',
    alignItems: 'center',
    backgroundColor: '#e6e6fa',
  },
  textConter: {
    fontSize: 28,
    color: '#000',
  },
  buttonStyle: {
```

```
52.     width: "80%",
53.     margin: 10,
54.   }
55.});
```

Link: https://github.com/profjordanov/hybrid-app-development/blob/master/07.%20React%20Native/RNClickCounter.txt

Display a List Using the FlatList Component in React Native

Lists are one of the common scrollable components to display similar types of data objects. A list is like an enhanced version of a `ScrollView` component to display data. React Native provides a `FlatList` component to create a list. `FlatList` only renders the list items that can be displayed on the screen. Additionally, `FlatList` offers many inbuilt features like vertical/horizontal scrolling, header/footer views, separator, pull to refresh, lazy loading, etc. This guide will explain the important details to create and optimize a list of cats images using the TheCatsAPI in React Native.

Basics of FlatList

FlatList is a specialized implementation of the `VirtualizedList` component to display a limited number of items that can fit inside the current window. The rest of the items will be rendered with the list scrolling action. FlatList can simply be implemented using the `data` and `renderItem` props to create a list. There are many other optional props available to add more features, so the props can be categorized into *primary* and *optional*.

Primary Props

- `data` takes an array of items, of type `any`, to populate items in the list.
- `renderItem` requires a function that takes an item object as an input from the `data` source to construct and return a list-item component.

Optional Props

Optional props are used to decorate FlatList using an item-divider, header/footer, pull-to-refresh, handle refreshing or optimization logic:

- `ItemSeparatorComponent` is used to add a separator to visually separate items.
- `keyExtractor` is used to provide a unique value (ID, email, etc.) to avoid the recreation of the list by tracking the reordering of the items.
- `extraData` takes a `boolean` value to re-render the current list. FlatList is a `PureComponent` that does not re-render against any change in the `state` or `props` objects, so `extraData` prop is used to re-render the current list if the `data` array is being mutated.
- `initialNumToRender` is used to render a minimum number of item-components in a FlatList for efficiency.
- `ListEmptyComponent` is used to display an empty view while the data is being downloaded.
- `ListHeaderComponent` is used to add a header component like search, menu-items, etc.
- `ListFooterComponent` is used to add a footer component like total items, data summary etc.
- `getItemLayout` returns the predefined size of the list-item component to skip the size calculation process at runtime to speed up the rendering process.
- `horizontal` prop takes a `boolean` value to create a horizontal list by returning like `horizontal={true}`.
- `numColumns` is used to create a column-based list.
- `onRefresh` and `refreshing` are used to implement pull-to-refresh controls, and maintain their visibility using a `boolean` flag.
- `onEndReached` and `onEndReachedThreshold` are used to implement lazy loading callback with a given threshold value. There are other props that can be used to implement style, scrolling, etc.

Downloading List Data

In order to fetch a mock response, use the `search` request API of TheCatsAPI, which doesn't require any registration or API key. The network request can simply be implemented using a `fetch` request:

```
fetchCats() {

fetch('https://api.thecatapi.com/v1/images/search?limit=10&page=1') // 1
      .then(res => res.json()) // 2
      .then(resJson => {
          this.setState({ data: resJson }); // 3
      }).catch(e => console.log(e));
}
```

JSX

The above request will:

1. Download the data with a `limit` of ten images from `page` one.
2. Convert the response into a JSON object.
3. Store the converted response JSON object in the `state` object as a value of the `data` key by using the `setState` function.

Implement a List Item Builder Function

The `fetch` request will provide the data to one of the primary props of the `FlatList`. Now implement a function to return a list-item component for the `renderItem` prop:

```
renderItemComponent = (itemData) => // 1
  <TouchableOpacity>   // 2
      <Image style={styles.image} source={{ uri: itemData.item.url }} /> // 3
  </TouchableOpacity>
```

JSX

The key field in the response is the `url` that will be used to download and display the image. The following steps explain the working of the `renderItemComponent` function:

1. Takes an object as a parameter that will be used by the UI components.
2. A `TouchableOpacity` component is used to implement a click listener because the `Image` component does not have an `onPress` prop.
3. An `Image` component takes a URI as a source data to display the image. It will download the image automatically.

Implement a FlatList

Import the `FlatList` component from the `react-native` module to create a list using `data` and `renderItemComponent`:

```
render() {
  return (
    <SafeAreaView>
      <FlatList
        data={this.state.data}
        renderItem={item =>
this.renderItemComponent(item)}
      />
    </SafeAreaView>)
}
```

j

The `this.state.data` will be updated by the `fetch` request, and it will trigger a re-rendering process in the `FlatList` component. The above implementation works fine, but the next steps will add more features to improve the `FlatList` implementation.

Implement KeyExtractor

React always uses a unique key to track the updates in a component. By default, the `FlatList` looks either for a custom `keyExtractor` implementation or a field named `key` in a data item, otherwise it uses the array index as the value of `key`. The `id` is a unique value in the response that can be used to implement the `keyExtractor`:

```
render() {
  return (
    <SafeAreaView>
      <FlatList
        data={this.state.data}
```

```
6         renderItem={item =>
this.renderItemComponent(item)}
7         keyExtractor={item => item.id.toString()}
8       />
9     </SafeAreaView>)
10}
```
jsx

Implement a Separator

A separator is a component that is placed between the list-items. It can be implemented by using a `View` component:

```
1ItemSeprator = () => <View style={{
2  height: 2,
3  width: "100%",
4  backgroundColor: "rgba(0,0,0,0.5)",
5}} />
6
7<FlatList
8    //...
9    ItemSeparatorComponent={this.ItemSeprator}
10/>
```
JSX

Just like `ItemSeparatorComponent`, different components can be implemented for `ListFooterComponent` and `ListHeaderComponent` props.

Implement Pull-to-refresh

The pull-to-refresh implementation requires a `boolean` flag for the `refreshing` prop to hide/display the loading indication.
The `handleRefresh` is a function for the `onRefresh` prop to update the data and loading indicator:

```
1handleRefresh = () => {
2    this.setState({ refreshing: false },
()=>{this.fetchCats()});
3}
4<FlatList
5    //...
```

```
    refreshing={this.state.refreshing}
    onRefresh={this.handleRefresh}
/>)
```

JSX

The value of `refreshing` is set to `true` before the `fetch` call to display the loading indicator. The value of the `refreshing` field will be set to `false` in case of a successful response or error.

Handle Mutable Data Changes

Since `FlatList` is a `PureComponent`, it assumes that the data is an immutable object and that any change in data is done by setting a new data source object in the state object. Alternatively, use the `extraData={this.state.isChanged}` prop to re-render a `FlatList` component by updating the value of `isChanged` property.
Here's the complete code of the `App.js` component:

```
/**
 * @author Pavneet Singh
 */

import React from "react";
import {
    StyleSheet,
    SafeAreaView,
    FlatList,
     View,
     Image,
     TouchableOpacity
} from "react-native";

export default class App extends React.Component {
    constructor(props) {
        super(props);
        this.state = {
            data: [],
            refreshing: true,
```

```
21          }
22      }
23
24      componentDidMount() {
25          this.fetchCats();
26      }
27
28      fetchCats() {
29          this.setState({ refreshing: true });
30
fetch('https://api.thecatapi.com/v1/images/search?limit
=10&page=1')
31              .then(res => res.json())
32              .then(resJson => {
33                  this.setState({ data: resJson });
34                  this.setState({ refreshing: false });
35              }).catch(e => console.log(e));
36      }
37
38      renderItemComponent = (data) =>
39          <TouchableOpacity style={styles.container}>
40              <Image style={styles.image} source={{
uri: data.item.url }} />
41          </TouchableOpacity>
42
43      ItemSeparator = () => <View style={{
44          height: 2,
45          backgroundColor: "rgba(0,0,0,0.5)",
46          marginLeft: 10,
47          marginRight: 10,
48      }}
49      />
50
```

```
51    handleRefresh = () => {
52        this.setState({ refreshing: false }, () => {
this.fetchCats() }); // call fetchCats after setting
the state
53    }
54
55    render() {
56      return (
57        <SafeAreaView>
58          <FlatList
59            data={this.state.data}
60            renderItem={item =>
this.renderItemComponent(item)}
61            keyExtractor={item => item.id.toString()}
62
ItemSeparatorComponent={this.ItemSeparator}
63            refreshing={this.state.refreshing}
64            onRefresh={this.handleRefresh}
65          />
66        </SafeAreaView>)
67    }
68}
69
70const styles = StyleSheet.create({
71  container: {
72    height: 300,
73    margin: 10,
74    backgroundColor: '#FFF',
75    borderRadius: 6,
76  },
77  image: {
78    height: '100%',
79    borderRadius: 4,
80  },
```

```
81});
```
JSX

Tips

- Use the `initialNumToRender` prop to render the defined numbers of items initially, and `maxToRenderPerBatch` to render items for every scroll to avoid blank space on the screen during scrolling. The default value for both of the props is `10`.
- Use the `getItemLayout` prop if the height or width of list-item is fixed. For example, if the height for item is 100 then use:

```
1  getItemLayout={(data, index) => (
2    {length: 100, offset: 100 * index, index}
3  )}
```
JSX

The `length` used to define item height (vertically) and `offset` is used to define the starting point for the current item index. For example, the starting point of the third item will be `100 * 2 = 200` (array index starts from 0).

Accessing Camera Roll Using React Native

When learning React Native, we find that majority of issues are encountered while trying to implement features such as taking a picture, accessing photos from the mobile device, chats, authentication, etc. These issues are a bit hard to resolve as the React Native documentation does not explain these concepts very well. Thus, we have to use libraries to achieve any of the above functionalities.
In this guide, we will highlight the features implemented using React Native APIs. This guide will help you implement the camera roll API for accessing pictures on a device. Feel free to also check out the official React Native documentation.

Setting up React Native

For someone just starting to build React Native apps, this should be a fairly simple guide to get started with. We'll be building our app for IOS, so the setup is according to that platform. We can go to the React Native site and follow the steps to build projects with native code.
Also for a quick setup, we can use the create-react-native-app.

By now, the app should be up and running on either the emulator or iPhone device. Below is how the folder structure should look. We have a component folder with three components as below:

1. CamScreen.js - The add picture button and accessing the camera roll take place here.
2. ViewPictures.js - Displays pictures from your iPhone.
3. SelectedPicture.js - Displays the selected picture.

We have the boilerplate code in our index.ios.js. It should look something like this:

index.ios.js

```javascript
import React, { Component } from 'react';
import {
  AppRegistry,
  StyleSheet,
  Text,
  View
} from 'react-native';
import CamScreen from './component/CamScreen';

export default class camRollExample extends Component {
  render() {
    return (
      <CamScreen/>
    );
  }
}

AppRegistry.registerComponent('camRollExample', () => camRollExample);
```

javascript

In CamScreen.js, we have a simple interface where we have the add picture button, and we also import the CameraRoll module from React Native.

CamScreen.js

```
import React, { Component } from 'react';
import {
  CameraRoll,
  Image,
  StyleSheet,
  TouchableHighlight,
  View,
} from 'react-native';

class CamScreen extends Component {

  render() {
    return (
      <View style={styles.container}>
        <TouchableHighlight>
          <Image
source={require('../assets/addPicture.png')} />
        </TouchableHighlight>
      </View>
    );
  }
}

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

```
31
32export default CamScreen;
```
javascript

Accessing Photos

For accessing photos, we need a click event. Let's add the `onPress` prop to the button and a handler method called `getPicturesFromGallery()`.

```
1//...
2render() {
3  return (
4      <View style={styles.container}>
5        <TouchableHighlight
6          onPress={() =>
this.getPicturesFromGallery()}>
7          <Image
8
source={require('../assets/addPicture.png')} />
9        </TouchableHighlight>
10      </View>
11    );
12}
```
javascript

getPicturesFromGallery()

```
1//...
2 getPicturesFromGallery() {
3    CameraRoll.getPhotos({ first: 5000 })
4      .then(res => {
5        console.log(res, "images data")
6      })
7  }
8  //...
```
javascript

The object inside the `getPhotos({ first: 5000 })` is used to specify the number of images that we want to get from the gallery. When we run the app, we'll encounter an error:
"Cannot read property 'getPhotos' of undefined"
The above error happens because the camera roll library has not been added or linked to our build phases in Xcode. To fix the error, we'll:

1. Go to our project directory
2. Open IOS directory
3. Navigate to the file that has .xcodeproj as the extension. In our case, it should be camRollExample.xcodeproj
4. Open this file in Xcode.

Next, we should drag RCTCamRoll.xcodeproj in our project directory to Xcode.
Let's drag the RCTCamRoll.xcodeproj file to libraries file in Xcode. We can then click on Build Phases located in the top right-hand corner in Xcode. Let's click the dropdown next to Link Binary With Libraries, then the + sign to add libRCTCamRoll.a.
We can then run the build and restart our emulator or device. The image object should be visible in our log. To check the images, we need to have the uri in the image object. Here is an updated version of CameraRoll.js.

CameraRoll.js

```
//...
import ViewPictures from './ViewPictures';

class CamScreen extends Component {

  state = {
    showPhotoGallery: false,
    pictureArray: []
  }

  getPicturesFromGallery() {
    CameraRoll.getPhotos({ first: 5000})
      .then(res => {
        let pictureArray = res.edges;
        this.setState({ showPhotoGallery: true, pictureArray: pictureArray })
```

```
16      })
17  }
18
19  render() {
20    if (this.state.showPhotoGallery) {
21      return (
22        <ViewPictures
23          pictureArray={this.state.pictureArray} />
24      )
25    }
26    return (
27      <View style={styles.container}>
28
29        <TouchableHighlight
30          onPress={() =>
this.getPicturesFromGallery()}>
31          <Image
32
source={require('../assets/addPicture.png')} />
33        </TouchableHighlight>
34      </View>
35    );
36  }
37}
38//...
```

javascript

In the above code snippets, we just imported ViewPictures.js. This is for displaying the images inside of a list view.

ViewPictures.js

```
1import React, { Component } from 'react';
2import {
3  Image,
4  View,
```

```
5  ListView,
6  StyleSheet,
7  Text,
8  TouchableHighlight
9
10} from 'react-native';
11
12import SelectedPicture from './SelectedPicture';
13
14class ViewPictures extends Component {
15  state = {
16    ds: new ListView.DataSource({
17      rowHasChanged: (r1, r2) => r1 !== r2
18    }),
19    showSelectedPicture: false,
20    uri: ''
21  }
22
23  renderRow(rowData) {
24    const { uri } = rowData.node.image;
25    return (
26      <TouchableHighlight
27        onPress={() => this.setState({
showSelectedPicture: true, uri: uri })}>
28        <Image
29          source={{ uri: rowData.node.image.uri }}
30          style={styles.image} />
31      </TouchableHighlight>
32    )
33  }
34
35  render() {
36    const { showSelectedPicture, uri } = this.state;
```

```
37
38    if (showSelectedPicture) {
39      return (
40        <SelectedPicture
41          uri={uri} />
42      )
43    }
44    return (
45      <View style={{ flex: 1 }}>
46        <View style={{ alignItems: 'center',
marginTop: 15 }}>
47          <Text style={{ fontSize: 30, fontWeight:
'700' }}>Pick A Picture !</Text>
48        </View>
49        <ListView
50          contentContainerStyle={styles.list}
51
dataSource={this.state.ds.cloneWithRows(this.props.pict
ureArray)}
52          renderRow={(rowData) =>
this.renderRow(rowData)}
53          enableEmptySections={true} />
54      </View>
55    );
56  }
57}
58
59const styles = StyleSheet.create({
60  list: {
61    flexDirection: 'row',
62    flexWrap: 'wrap'
63  },
64
65  image: {
```

```
    width: 120,
    height: 130,
    marginLeft: 15,
    marginTop: 15,
    borderRadius: 6,
    borderWidth: 2,
    borderColor: '#efefef'
  }
})

export default ViewPictures;
```

javascript

When we click or select a picture from the list, the selected picture will get displayed in the SelectedPicture.js component.

SelectedPicture.js

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

const SelectedPicture = (props) => {
  const { uri } = props;
  return (
    <View style={styles.container}>
      <Image
        source={{uri: uri}}
        style={styles.image}/>
    </View>
```

```
  );
};

const styles = StyleSheet.create({
  container: {
    flex: 1,
    justifyContent: 'center',
    alignItems: 'center'
  },
  image: {
    height: 300,
    width: 200
  }
});

export default SelectedPicture;
```

Conclusion

React Native is a great way to build hybrid apps. You can either use `Expo` or `npx react-native` CLI to get started with React Native development. The optimized codebase is available at RnClickCounter repository. Hopefully, this guide explained the necessary details to get started with React Native on Android. Happy Coding!