React native

Tags

Introduction

React Native is a mobile application development framework that allows developers to build natively rendered applications for Android and iOS using JavaScript and the React framework. It allows developers to create mobile applications that are indistinguishable from applications built using traditional, platform-specific technologies. React Native enables developers to build high-quality mobile applications that are performant, responsive, and easy to maintain.

Apps built

Here are a few more examples of apps built using React Native:

- Walmart: The popular retail giant uses React Native to power its mobile shopping app.
- Bloomberg: The financial news and data provider uses React Native for its mobile app.
- Discord: The popular gaming and communication platform uses React Native for its mobile app.
- Tesla: The electric vehicle manufacturer uses React Native for its mobile app, which allows users to control and monitor their cars from their phones.

These are just a few examples of the many apps that have been built using React Native. The framework is widely used and trusted by many major companies and organizations to power their mobile apps.

Simple Example app

Here is a simple example of a React Native component that displays a greeting message:

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

const Greeting = ({name}) => {
  return (
    <View>
      <Text>Hello, {name}!</Text>
    </View>
  );
}

export default Greeting;
```

This component takes a `name` prop and uses it to display a greeting message. It is then exported so it can be used in other parts of the app.

To use this component in another file, you would import it and use it like this:

```
import Greeting from './Greeting';

const App = () => {
  return (
```

```
    <Greeting name="Jane" />
  );
}
```

This would render the `Greeting` component and pass it the `name` prop, resulting in the message "Hello, Jane!" being displayed on the screen.

Add CSS to this

To add CSS styles to a React Native app, you can use the `StyleSheet` API. Here is an example of how you could use it to style the `Greeting` component from the previous example:

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

const Greeting = ({name}) => {
  return (
    <View style={styles.container}>
      <Text style={styles.greeting}>Hello, {name}!</Text>
    </View>
  );
}

const styles = StyleSheet.create({
  container: {
    flex: 1,
    alignItems: 'center',
    justifyContent: 'center',
  },
  greeting: {
    fontSize: 32,
    fontWeight: 'bold',
    color: '#333',
  },
});

export default Greeting;
```

In this example, we define a `styles` object that contains the styles for our component. We then apply those styles to the `View` and `Text` components using the `style` prop.

This allows us to add custom styles to our React Native app, giving us more control over the look and feel of our app.

Axios react native app

To connect Axios to a React Native app, you will first need to install the `axios` library. Here is an example of how you could use it in a React Native app:

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

const App = () => {
  // Use Axios to make an HTTP request:
  axios.get('https://www.example.com/api/data')
    .then((response) => {
      // Do something with the response.
    })
    .catch((error) => {
      // Handle the error.
    });

  return (
    <View>
      <Text>Your Axios code goes here!</Text>
    </View>
  );
}

export default App;
```

In this example, we use the `axios.get()` method to make an HTTP GET request to the specified URL. This method returns a promise that is resolved with the response from the server, or rejected if there is an error.

We use the `.then()` method to handle the successful response, and the `.catch()` method to handle any errors. This allows us to easily make HTTP requests and handle the response in our React Native app.

Once you have installed Axios and connected it to your app, you can use it to make HTTP requests to APIs and other web services to fetch data and perform other operations.

Famous libraries

React Native to build high-quality mobile apps. Some examples include:

- Redux: A popular state management library that is often used with React Native to manage application state.
- React Navigation: A library for managing navigation in React Native apps.
- Axios: A popular library for making HTTP requests in React Native apps.
- React Native Elements: A library of pre-built, customizable UI components for React Native apps.
- React Native Paper: A library of customizable, Material Design-style components for React Native.
- React Native Animatable: A library of pre-built animations that can be used in React Native apps.

Navigate between screens

To navigate between different screens in a React Native app, you can use the `navigate()` function from the `navigation` prop that is passed to the screen component. The `navigate()` function takes one argument: the name of the destination screen.

Here is an example of how to navigate to a different screen using the `navigate()` function:

```
import { useNavigation } from '@react-navigation/native';

function HomeScreen() {
  const navigation = useNavigation();

  const handlePress = () => {
    navigation.navigate('DetailScreen');
  };

  return (
    <Button title="Go to detail screen" onPress={handlePress} />
  );
}
```

If you want to pass data to the destination screen, you can use the `navigate()` function with an additional object containing the data you want to pass. For example:

```
navigation.navigate('DetailScreen', {
  name: 'John',
  age: 30,
});
```

On the destination screen, you can use the `route.params` prop to access the passed data. For example:

```
function DetailScreen({ route }) {
  return (
    <View>
      <Text>Name: {route.params.name}</Text>
      <Text>Age: {route.params.age}</Text>
    </View>
  );
}
```

To navigate back to the previous screen, you can use the `goBack()` function from the `navigation` prop. For example:

```
import { useNavigation } from '@react-navigation/native';

function DetailScreen() {
  const navigation = useNavigation();

  const handlePress = () => {
    navigation.goBack();
  };

  return (
    <Button title="Go back" onPress={handlePress} />
  );
}
```

Note that the `navigate()` and `goBack()` functions are part of the `navigation` prop that is provided by a navigation library like `react-navigation`. You will need to set up your app with a navigation library in order to use these functions.

Pass data with navigation

To pass data to different screens in a React Native app, you can use the `navigate()` function from the `navigation` prop that is passed to the screen component. The `navigate()` function takes two arguments: the name of the destination screen and an object containing the data you want to pass.

Here is an example of how to pass data to a different screen using the `navigate()` function:

```
import { useNavigation } from '@react-navigation/native';

function HomeScreen() {
  const navigation = useNavigation();

  const handlePress = () => {
    navigation.navigate('DetailScreen', {
      name: 'John',
      age: 30,
    });
  };

  return (
    <Button title="Go to detail screen" onPress={handlePress} />
  );
}
```

On the destination screen, you can use the `route.params` prop to access the passed data. For example:

```
function DetailScreen({ route }) {
  return (
    <View>
      <Text>Name: {route.params.name}</Text>
      <Text>Age: {route.params.age}</Text>
    </View>
  );
}
```

You can also use the `setParams()` function from the `navigation` prop to update the data passed to a screen. For example:

```
import { useNavigation } from '@react-navigation/native';

function DetailScreen() {
```

```
  const navigation = useNavigation();

  const handlePress = () => {
    navigation.setParams({ name: 'Jane' });
  };

  return (
    <Button title="Update name" onPress={handlePress} />
  );
}
```

Note that the data passed to a screen using the `navigate()` function is only available on the first render of the screen. If you want to persist the data across multiple renders, you can use a state management library like Redux or MobX, or you can use the `useContext()` hook to share data between screens.

Storing data in the App

There are several ways to store data in a React Native app, depending on your needs and the type of data you want to store. Here are a few options:

1. **AsyncStorage:** AsyncStorage is a simple, unencrypted, asynchronous, persistent, key-value storage system that is global to the app. It can be used to store small pieces of data, such as strings or booleans.

To use AsyncStorage, you can import the `AsyncStorage` module and use the `setItem()` and `getItem()` functions to store and retrieve data. For example:

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

// Store data
AsyncStorage.setItem('userId', '123');

// Retrieve data
AsyncStorage.getItem('userId').then(userId => {
  console.log(userId); // '123'
});
```

1. **SQLite:** SQLite is a lightweight, self-contained, SQL database engine that is widely used in mobile apps. It can be used to store structured data, such as user profiles or app settings.

To use SQLite in a React Native app, you can use a library like `react-native-sqlite-storage`. First, install the library using npm:

```
npm install react-native-sqlite-storage
```

Then, you can import the `SQLite` module and use the `openDatabase()` function to create a new database. You can then use SQL commands to create tables and insert, update, and delete data. For example:

```
import SQLite from 'react-native-sqlite-storage';

const db = SQLite.openDatabase('test.db');

db.transaction(tx => {
  tx.executeSql(
    'CREATE TABLE IF NOT EXISTS users (id INTEGER PRIMARY KEY, name TEXT, age INTEGER)',
    [],
    () => {
      console.log('Table created successfully');
    },
    (_, error) => {
      console.error('Error creating table', error);
    }
  );
});

db.transaction(tx => {
  tx.executeSql(
    'INSERT INTO users (name, age) VALUES (?, ?)',
    ['John', 30],
    () => {
      console.log('Insert successful');
    },
    (_, error) => {
      console.error('Error inserting data', error);
    }
  );
});

db.transaction(tx => {
  tx.executeSql(
    'SELECT * FROM users',
    [],
    (_, result) => {
      console.log('Result', result);
    },
    (_, error) => {
      console.error('Error fetching data', error);
```

```
    }
  );
}
```

1. **Firebase:** Firebase is a cloud-based platform that provides a range of services for mobile and web apps, including real-time database, storage, and authentication. You can use Firebase to store and sync data across all clients in real-time.

To use Firebase in a React Native app, you will need to install the Firebase JavaScript library and initialize the Firebase app with your Firebase configuration.

Integrate Firebase

To integrate Firebase with a React Native app, you will need to have a basic understanding of React Native and some experience with programming in JavaScript. Firebase is a popular backend as a service (BaaS) platform that provides a range of services for mobile and web app development, including cloud storage, real-time databases, authentication, and more.

Here is a brief outline of the steps you would need to follow to integrate Firebase with your React Native app:

1. Install the Firebase CLI (Command Line Interface) and create a new Firebase project. This will give you access to your Firebase project's unique credentials, which you will need to connect your app to Firebase.
2. Install the `firebase` package from npm, which provides an interface for interacting with Firebase services from a React Native app.
3. Import the `firebase` package in your React Native project and initialize it with your Firebase project's credentials. This will allow your app to connect to Firebase and access its services.
4. Use the `firebase` package to interact with the Firebase services you want to use in your app. For example, you can use the `auth()` function to enable authentication for your app, the `database()` function to access the real-time database, and the `storage()` function to upload and download files from Firebase storage.
5. Test your app on a physical device or emulator to ensure it is working as expected and that it is successfully interacting with Firebase.

To integrate Firebase in a React Native app, you will need to follow these steps:

1. Install the Firebase JavaScript library using npm:

```
npm install firebase
```

1. Import the Firebase module in your React Native code:

```
import firebase from 'firebase';
```

1. Initialize the Firebase app with your Firebase configuration:

```
const firebaseConfig = {
  apiKey: "your-api-key",
  authDomain: "your-auth-domain",
  databaseURL: "your-database-url",
  projectId: "your-project-id",
  storageBucket: "your-storage-bucket",
  messagingSenderId: "your-messaging-sender-id",
  appId: "your-app-id",
  measurementId: "your-measurement-id"
};

firebase.initializeApp(firebaseConfig);
```

- Add the required permissions to your `AndroidManifest.xml` file:

```
<manifest ...>
  <uses-permission android:name="android.permission.INTERNET" />
  <application ...>
    <!-- Add these lines to your AndroidManifest.xml -->
    <activity
      android:name=".MainActivity"
      android:launchMode="singleTask"
      android:screenOrientation="portrait"
      android:windowSoftInputMode="adjustResize">
      <intent-filter>
        <action android:name="android.intent.action.MAIN" />
        <category android:name="android.intent.category.LAUNCHER" />
      </intent-filter>
      <intent-filter>
        <action android:name="com.google.firebase.MESSAGING_EVENT" />
```

```
      </intent-filter>
    </activity>
  </application>
</manifest>
```

- Add the required dependencies to your `build.gradle` file:

```
implementation 'com.google.firebase:firebase-analytics:17.5.0'
implementation 'com.google.firebase:firebase-auth:19.3.1'
implementation 'com.google.firebase:firebase-firestore:21.6.0'
```

- Run `react-native link` to automatically update your project with the required dependencies.

That's it! You should now be able to use Firebase in your React Native app.

Here are a few examples of common tasks you can do with Firebase in a React Native app:

- **Authentication:** You can use Firebase's built-in authentication services to sign up and sign in users.

```
firebase.auth().signInWithEmailAndPassword(email, password)
  .then(user => {
    console.log("Sign in successful", user);
  })
  .catch(error => {
    console.error("Sign in error", error);
  });
```

Real-time database: You can use Firebase's real-time database to store and sync data across all clients in real-time.

To use Firebase Realtime Database in your React Native app, you can follow these steps:

1. In the Firebase console, create a new Realtime Database for your app.
2. In your React Native code, you can use the `firebase.database()` function to get a reference to the database and then use the `ref()` function to specify a location in

the database. For example:

```
const database = firebase.database();
const usersRef = database.ref('users');
```

1. You can then use the `set()` method to add data to the database, or the `on()` method to listen for changes to the data. For example:

```
usersRef.set({
  user1: {
    name: 'John',
    age: 30
  },
  user2: {
    name: 'Jane',
    age: 25
  }
});

usersRef.on('value', snapshot => {
  console.log(snapshot.val());
});
```

File sharing App

To create a file sharing app with React Native, you can follow these steps:

1. Install the required libraries:

```
npm install react-native-fs react-native-document-picker
```

1. Link the libraries to your project:

```
react-native link react-native-fs react-native-document-picker
```

1. Import the libraries in your React Native code:

```
import RNFetchBlob from 'react-native-fs';
import DocumentPicker from 'react-native-document-picker';
```

1. Use the `DocumentPicker` module to let the user select a file from their device:

```
Copy code
try {
  const res = await DocumentPicker.pick({
    type: [DocumentPicker.types.allFiles],
  });
  console.log(
    res.uri,
    res.type, // mime type
    res.name,
    res.size
  );
} catch (err) {
  if (DocumentPicker.isCancel(err)) {
    // User cancelled the picker, exit any dialogs or menus and move on
  } else {
    throw err;
  }
}
```

- Use the `RNFetchBlob` library to upload the file to a server or cloud storage service. For example, to upload the file to Firebase Storage, you can do the following:

```
const file = {
  uri: res.uri,
  name: res.name,
  type: res.type,
};

const storageRef = firebase.storage().ref();
const fileRef = storageRef.child(file.name);

RNFetchBlob.fs.readFile(file.uri, 'base64')
  .then(data => {
    return RNFetchBlob.polyfill.Blob.build(data, { type: `${file.type};BASE64` });
  })
  .then(blob => {
    fileRef.put(blob, { contentType: file.type })
      .then(() => {
        console.log('File uploaded successfully');
      })
      .catch(error => {
        console.error('Error uploading file', error);
      });
  });
```

- To allow users to download the shared files, you can use the `RNFetchBlob` library to download the file from the server or cloud storage service and save it to the device. For example, to download a file from Firebase Storage, you can do the following:

```
fileRef.getDownloadURL()
  .then(url => {
    RNFetchBlob.config({
      fileCache: true,
      addAndroidDownloads: {
        useDownloadManager: true,
        notification: true,
        path: RNFetchBlob.fs.dirs.DownloadDir + '/' + file.name,
        description: 'File downloaded by MyApp',
      },
    })
      .fetch('GET', url)
      .then(res => {
        console.log('File saved to', res.path());
      })
      .catch(error => {
        console.error('Error downloading file', error);
      });
  })
...
...
...
```

Build an email app

1. Install the React Native CLI (Command Line Interface) and create a new React Native project using the `react-native init` command.
2. Install the `react-native-mail` package from npm, which provides an interface for sending emails from a React Native app.
3. Import the `react-native-mail` package in your project and use its `send` function to send emails. This function takes a number of arguments, including the recipient's email address, the subject of the email, and the body of the email.
4. Use React Native components, such as `TextInput` and `Button`, to build the user interface for your app. These components allow the user to enter the recipient's email address, the subject, and the body of the email, and to trigger the email sending function when the send button is pressed.

5. Test your app on a physical device or emulator to ensure it is working as expected.

Build Web3 react native apps

Basic app

To add web3 to a React Native app, you will need to use a library like `web3.js`. Here is an example of how you could use it in a React Native app:

```
import React from 'react';
import { View, Text } from 'react-native';
import Web3 from 'web3';

const App = () => {
  const web3 = new Web3('https://mainnet.infura.io/v3/YOUR-INFURA-API-KEY');
  // Replace 'YOUR-INFURA-API-KEY' with your actual Infura API key.

  // Use web3 to do something, like getting the balance of an Ethereum address:
  const balance = web3.eth.getBalance('0x0000000000000000000000000000000000000000')
    .then((result) => {
      // Do something with the result.
    });

  return (
    <View>
      <Text>Your web3 code goes here!</Text>
    </View>
  );
}

export default App;
```

In this example, we import the `Web3` class from the `web3.js` library and use it to create a new `web3` object. We then use this object to call methods like `web3.eth.getBalance()` to interact with the Ethereum blockchain.

You will need to replace `YOUR-INFURA-API-KEY` with your actual Infura API key in order for this code to work. Infura is a service that provides secure, reliable access to the Ethereum blockchain via HTTP.

Once you have added web3 to your React Native app, you can use it to build decentralized applications (dApps) that interact with the Ethereum blockchain.

Subscribe to events

Here is another example of how you could use web3 in a React Native app, this time to listen for events on the Ethereum blockchain:

```
import React from 'react';
import { View, Text } from 'react-native';
import Web3 from 'web3';

const App = () => {
  const web3 = new Web3('https://mainnet.infura.io/v3/YOUR-INFURA-API-KEY');
  // Replace 'YOUR-INFURA-API-KEY' with your actual Infura API key.

  // Listen for events on the Ethereum blockchain:
  const eventEmitter = web3.eth.subscribe('logs', {
    address: '0x0000000000000000000000000000000000000000',
  }, (error, result) => {
    if (error) {
      // Handle the error.
    } else {
      // Do something with the result.
    }
  });

  return (
    <View>
      <Text>Your web3 code goes here!</Text>
    </View>
  );
}

export default App;
```

In this example, we use the `web3.eth.subscribe()` method to listen for events on the Ethereum blockchain. This method takes a few arguments: the type of events to listen for, a filter object, and a callback function to be called when an event is received.

In this case, we are listening for `logs` events and passing a filter object that specifies the address to listen for events on. When an event is received, the callback function is called with the result of the event.

Using web3 in this way allows you to build React Native apps that can interact with the Ethereum blockchain and respond to events on the network. This opens up many

possibilities for building decentralized applications (dApps) that can harness the power of the Ethereum network.

Connect to contracts

```
const eventEmitter = web3.eth.subscribe('logs', {
    address: '0x0000000000000000000000000000000000000000',
    topics: ['0x0000000000000000000000000000000000000000000000000000000000000001']
  }, (error, result) => {
    if (error) {
      // Handle the error.
    } else {
      // Do something with the result.
    }
  });
```

we use the `web3.eth.subscribe()` method to listen for events emitted by a smart contract. This method takes a few arguments: the type of events to listen for, a filter object, and a callback function to be called when an event is received.

In this case, we are listening for `logs` events and passing a filter object that specifies the address of the smart contract to listen for events on, as well as the topic(s) of the events to listen for. When an event is received, the callback function is called with the result of the event.

Additional functions

The `web3.js` library contains many different methods and functions that you can use to interact with the Ethereum blockchain. Here are just a few examples:

- `web3.eth.getBlockNumber()`: Gets the number of the latest block on the Ethereum blockchain.
- `web3.eth.getBalance()`: Gets the balance of an Ethereum address.
- `web3.eth.getTransaction()`: Gets the details of a specific Ethereum transaction.
- `web3.eth.getTransactionReceipt()`: Gets the receipt of a specific Ethereum transaction.
- `web3.eth.sendTransaction()`: Sends an Ethereum transaction to the network.
- `web3.eth.subscribe()`: Subscribes to events on the Ethereum blockchain.

These are just a few examples of the many methods and functions that are available in the `web3.js` library. You can use these methods to build powerful, decentralized applications (dApps) that interact with the Ethereum blockchain.