React Native - A Beginner's Guide

This is a beginner's guide to React Native. It covers the basics of React Native and how to build mobile apps using JavaScript and React. It's a collection of notes and code snippets I created while learning React Native.

Contents

What is React Native?

React Native is a framework for building native mobile apps using JavaScript and React. It allows you to create a single codebase that can be used to build apps for both iOS and Android. React Native was created by Facebook in 2015 and is being used by Instagram, Uber, Tesla, Walmart, and more. It is one of the most popular frameworks for building mobile apps.

Advantages

- Cross-platform development (iOS and Android)
- Hot Reloading (instantly see changes)
- Strong community support (lots of packages)
- Easy to learn (if you know React)

Native vs Hybrid Apps

React Native is technically a Hybrid App but it uses native mobile components to give a native feel to the app. It's not a web app running on mobile, it's a mobile app built using JavaScript and React.

Native

- Create 2 different apps, one for iOS using Objective C or Swift and one for Android using Java or Kotlin
- Great user experience
- Expensive to develop and maintain since dual work

Hybrid

- Create 1 app using a single codebase
- Examples: Ionic, Cordova, PhoneGap
- Cheaper to develop and maintain
- Not as great user experience since we just create a web app that runs on mobile

With React Native, we get the best of both worlds. It acts as a bridge between JavaScript and native platforms. It takes our JavaScript code and converts it to native code.

React Native vs. React

Instead of using HTML elements, you use React Native components.

React

```
import react from 'react';

const App = () => {
  return (
    <div className="container">
      <h1>Hello World</h1>
    </div>
  );
};
```

React Native

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

const App = () => {
  return (
    <View style={{ flex: 1, justifyContent: 'center', alignItems: 'center' }}>
      <Text>Hello World</Text>
    </View>
  );
};
```

So instead of rendering HTML elements, you render Native mobile components.

Expo

- A tool like *create-react-app/ Vite* for React Native
- Allows you to build React Native apps without having to install Xcode or Android Studio
- External platform that helps us run and build React Native apps
- Not compulsory but makes it easier to develop React Native apps
- If not for expo, we would need to install Android Studio and Xcode to run our apps on simulators
- No need to configure native build tools
- Gives you access to native APIs (camera, navigation, push notifications, maps, etc.)

Environment Setup

- Get the latest version of Node.js from Download - Node.js
- Install Expo CLI using

  ```
  npm install -g expo-cli
  ```

- Install Expo app on your phone from the App Store or Play Store
- Move to the directory where you want to create your project

  ```
  cd Desktop
  ```

- Create a new project

  ```
  npx create-expo-app my-app
  ```

- Move to the project directory

  ```
  cd my-app
  ```

- Start the project

  ```
  npm start # you can also use: expo start
  ```

- Scan the QR code using the Expo app on your phone to preview the app

> Note: Your phone and computer should be on the same network for this to work.

We can also create a custom setup for our app using React Native CLI. However, this can get a bit complicated and we would need to install a lot of different dependencies. So we will stick to Expo for now.

Developer Setup

Install a Linter

- Get a linter to make sure our code is clean and consistent

```
npm i eslint --save-dev # dev dependency
```

- Setup eslint

```
npx eslint --init
```

- Choose the required options and install the required dependencies

Install React Native Community ESLint Config

- Install the plugin

```
npm i @react-native-community/eslint-config --save-dev
```

- Add any additional rules to the `.eslintrc.js` file

Install Prettier

- Install prettier

```
npm i --save-dev --save-exact prettier
```

- Create a `.prettierrc.js` file and add the following code

```
module.exports = {
  bracketSpacing: true,
  jsxBracketSameLine: false,
  semi: false,
  singleQuote: true,
  tabWidth: 2,
  useTabs: false,
  trailingComma: 'none',
};
```

We can find out more about setting up these configuration files via their documentation.

ESLint and Prettier Extension

- Install the ESLint and Prettier extension in your code editor
- Add the following code to the `settings.json` file if needed

```
"editor.codeActionsOnSave": {
  "source.fixAll.eslint": true
},
"editor.formatOnSave": true,
"editor.defaultFormatter": "esbenp.prettier-vscode"
```

Script to Run ESLint

- Open the `package.json` file and add the following script

```
"scripts": {
  "lint": "eslint ."
}
```

- Run the script

```
npm run lint
```

Debugging

- Depends on how the app is running
- If running on a simulator, we can use the developer tools
- If running on a physical device, we can use the developer tools or use the `console.log()` method

Native Components

There are a lot of native components that do the same thing but differ in name based on the platform. For example, iOS uses `<UIImageView` for images whereas Android uses `<ImageView>`.

React Native comes as a bridge between both platforms. We can use `<Image>` and React Native will automatically use the correct component based on the platform.

Core Components

These are components that work like different elements in HTML.

Text Component

- `<Text>` is the equivalent of `<p>` in HTML
- `<Text>` is the only component that can render text
- `<Text>` can be styled with the `style` prop

View Component

- `<View>` is the equivalent of `<div>` in HTML
- `<View>` is used to create layout structures for other components (rows, columns, etc.)
- Has a number of props to control its appearance and behavior
- Uses flexbox layout by default

Image Component

- `<Image>` is the equivalent of `<img>` in HTML
- `<Image>` is used to display images
- Has several props to control its appearance and behavior

ScrollView Component

- `<ScrollView>` is the equivalent of `<div>` with `overflow: scroll` in HTML
- `<ScrollView>` is used to create a scrollable view
- It can contain multiple components and views
- Supports both vertical and horizontal scrolling

TextInput Component

- `<TextInput>` is the equivalent of `<input>` in HTML
- `<TextInput>` is used to create an input field
- It has several props to control its appearance and behavior

SafeAreaView Component

- `<SafeAreaView>` is used to render content within the safe area boundaries of a device
- It's used to prevent content from being hidden by the status bar, notches, rounded corners, etc.

Creating Components

For every new component, first import React and then import the required core components from React Native.

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

There are a lot of ways to create a component. We can use a function or a class. We will be using Functional Components for now.

```
const ComponentName = () => {
  return (
    <View>
      <Text>Component</Text>
    </View>
  );
};
```

Make sure to export the component.

```
export default ComponentName;
```

If we render this right now, it'll be hitting the borders of the screen. We can't just fix this by adding a margin to the component since it'll be different for different devices. So we need to use another component called `<SafeAreaView>`.

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

const ComponentName = () => {
  return (
    <SafeAreaView>
      <View>
        <Text>Component</Text>
      </View>
    </SafeAreaView>
  );
};
```

Now the component will be rendered inside the safe area of the screen.

Styling

We don't use CSS in React Native. Instead, we use JavaScript objects to style our components. It's very similar to CSS.

There are a bunch of different ways to apply style but let's start with -

Inline Styling

Here we pass a style object to the `style` prop of the component.

```
const ComponentName = () => {
  return (
    <SafeAreaView>
      <View style={{ backgroundColor: 'dodgerblue', height: 100, width: 100 }}>
        <Text>Component</Text>
      </View>
    </SafeAreaView>
  );
};
```

StyleSheet Object

Here we create a StyleSheet object and pass it to the `style` prop of the component.

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

const ComponentName = () => {
  return (
    <SafeAreaView>
      <View style={styles.container}>
        <Text>Component</Text>
      </View>
    </SafeAreaView>
  );
};

const styles = StyleSheet.create({
  container: {
    backgroundColor: 'dodgerblue',
    height: 100,
    width: 100,
  },
});

export default ComponentName;
```

Usually, it's best to do this before we export the component. Also, refer to the documentation for the list of available props for each component.

Layout Props

In React Native, we use Flexbox for layout. Flex 1 means that the component will take up all the available space. We can also use `flexDirection` to change the direction of the flexbox.

```
const ComponentName = () => {
  return (
    <SafeAreaView style={styles.wrapper}>
      <View style={styles.container}>
        <Text>Component</Text>
      </View>
    </SafeAreaView>
  );
};

const styles = StyleSheet.create({
  wrapper: {
    flex: 1,
  },
  container: {
    backgroundColor: 'dodgerblue',
    flex: 1,
  },
});

export default ComponentName;
```

> Note: No style values in React Native don't have units. So we don't need to use `px` or `rem`. They have a default unit of `dp` which is a device-independent pixel.

Lists in React Native

- One of the most common and important components of mobile apps
- 3 types of lists
  - FlatList
  - SectionList
  - VirtualizedList
- Most of the requirements are fulfilled by FlatList and SectionList

FlatList vs SectionList

- FlatList is used when we have a list of similar items
- Standard list of items

```
 - Apple
- Banana
- Orange
```

- SectionList is used when we have a collection of multiple lists of items
- A list is broken up into multiple sections

```
 Fruits

- Apple
- Banana
- Orange

Vegetables

- Potato
- Tomato
- Onion

Grains

- Rice
- Wheat
- Corn
```

FlatList

- Renders a list of items
- 2 props are mandatory
  - `data` : An array of data to be rendered
  - `renderItem` : A function that returns the component to be rendered for each item
- Has a bunch of other props that can be used to customize the list

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

const ComponentName = () => {
  const data = ['Apple', 'Banana', 'Orange'];

  return (
    <SafeAreaView style={styles.wrapper}>
      <FlatList data={data} renderItem={({ item }) => <Text>{item}</Text>} />
    </SafeAreaView>
  );
};
```

- Can have an array of elements as well as objects as data
- If the component to render is complex, we can put it in a separate component and import it
- We can also use the `keyExtractor` prop to specify the key for each item

```
const data = [
  { id: 1, name: 'Apple' },
  { id: 2, name: 'Banana' },
  { id: 3, name: 'Orange' },
];

<FlatList
  data={data}
  renderItem={({ item }) => <ListItem title={item.name} />}
  keyExtractor={(item) => item.id.toString()}
/>;
```

SectionList

- Renders a list broken up into sections
- 2 props are mandatory
  - `sections` : An array of sections to be rendered
  - `renderItem` : A function that returns the component to be rendered for each item
- Has a bunch of other props that can be used to customize the list

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

const ComponentName = () => {
  const data = [
    {
      title: 'Fruits',
      data: ['Apple', 'Banana', 'Orange'],
    },
    {
      title: 'Vegetables',
      data: ['Potato', 'Tomato', 'Onion'],
    },
    {
      title: 'Grains',
      data: ['Rice', 'Wheat', 'Corn'],
    },
  ];

  return (
    <SafeAreaView style={styles.wrapper}>
      <SectionList
        sections={data}
        renderSectionHeader={({ section }) => <Text>{section.title}</Text>}
        renderItem={({ item }) => <ListItem item={item} />}
      />
    </SafeAreaView>
  );
};
```

- Can only an array of objects as data

Key Extractor

- Used to specify a unique key for each item
- When working with lists, React needs to know which item has changed, been added, or been removed
- Must be a string

How keys are handled under the hood

When an item is updated or deleted, React by instinct will rebuild the app. But if we have a key, React will only update the item that has changed. This helps in improving performance.

Some Other Props

There are a bunch of other props that can be used to customize the list.

`ItemSeparatorComponent`

A component to render between each item just to make UI look better. It works similarly to <hr> in HTML.

```
<FlatList
  data={data}
  renderItem={({ item }) => <ListItem title={item.name} />}
  keyExtractor={(item) => item.id.toString()}
  ItemSeparatorComponent={() => <View style={styles.separator} />}
/>
```

`ListEmptyComponent`

A component to render when the list is empty.

```
<FlatList
  data={data}
  renderItem={({ item }) => <ListItem title={item.name} />}
  keyExtractor={(item) => item.id.toString()}
  ItemSeparatorComponent={() => <View style={styles.separator} />}
  ListEmptyComponent={() => <Text>No items to display</Text>}
/>
```

There's tons to learn about lists. Refer to the documentation for more information.

> Note: There's no need to worry about performance if the list isn't very long. FlatList and SectionList are optimized for performance. FlatList only renders the items that are visible on the screen. It only rerenders when the data changes.

Icons

There are a few ways to use icons in React Native. We are going to be using a library called `react-native-vector-icons` which comes pre-installed with Expo. However, it can also be installed separately.

Get the Icon in your project

- To get the list of icons, refer to Expo Icons
- Find the icon you want to use and click on it
- Follow the specified steps
- Import the icon in your component

```
import { MaterialCommunityIcons } from '@expo/vector-icons';
```

- Use the icon in your component

```
<MaterialCommunityIcons name="email" size={100} color="dodgerblue" />
```

Images

We have a few different ways to work with images. For example, local or network images.

Local Images

- We can import images from our project
- We use `require()` to import images
- We can use the `source` prop to specify the image like we used `src` in HTML

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

const ComponentName = () => {
  return (
    <SafeAreaView style={styles.wrapper}>
      <Image source={require('./assets/icon.png')} />
    </SafeAreaView>
  );
};
```

Network Images

- Used when we want to display images from the internet
- We can use the `source` prop to specify the image URL

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

const ComponentName = () => {
  return (
    <SafeAreaView style={styles.wrapper}>
      <Image
        source={{
          uri: 'https://picsum.photos/200/300',
        }}
      />
    </SafeAreaView>
  );
};
```

Gotchas

- Network images must have a height set
- When sizing images, use height and width props or aspectRatio with a defined width/height
- Be mindful of storage sizes when using static images

> Note: If you see images with file names like `@2x` or `@3x`, it means that the image is optimized for different screen sizes. The `@2x` image is for retina displays and the `@3x` image is for high-resolution displays.

`ImageBackground` Component

- Used to display an image as the background of a component
- We can add any children components inside the `ImageBackground` component

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

const ComponentName = () => {
  return (
    <SafeAreaView style={styles.wrapper}>
      <ImageBackground
        source={require('./assets/icon.png')}
        style={styles.image}
      >
        <Text>Inside</Text>
      </ImageBackground>
    </SafeAreaView>
  );
};
```

Buttons in React Native

There are a few different ways to create buttons in React Native. We can use the `Button` component, the `TouchableOpacity` component, or the `TouchableWithoutFeedback` component.

The `Button` Component

- A built-in component used to create a button
- It's a simple button that can be used to perform an action when clicked

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

const ComponentName = () => {
  return (
    <Button title="Press Me" onPress={() => console.log('Button Pressed')} />
  );
};
```

The `TouchableOpacity` Component

- A built-in component used to create a button
- It's similar to the `Button` component but it's more customizable

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

const ComponentName = () => {
  return (
    <TouchableOpacity onPress={() => console.log('Button Pressed')}>
      <Text>Press Me</Text>
    </TouchableOpacity>
  );
};
```

The `TouchableWithoutFeedback` Component

- A built-in component used to create a button
- It's similar to the `TouchableOpacity` component but it doesn't have any visual feedback when pressed which means that it doesn't change color when pressed

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

const ComponentName = () => {
  return (
    <TouchableWithoutFeedback onPress={() => console.log('Button Pressed')}>
      <Text>Press Me</Text>
    </TouchableWithoutFeedback>
  );
};
```

Props

- Props are used to pass data from one component to another
- Short for properties
- Used to customize components
- Promotes component reuse based on different data

Every core component in React Native has a bunch of props that can be used to customize the component. For example, the `Text` component has a `numberOfLines` prop that can be used to specify the number of lines to show.

```
<Text numberOfLines={1}>This is a very long text</Text>
```

- Props are immutable and cannot be changed once the component is rendered
- Use unidirectional data flow to pass data from parent to child components
- Unidirectional data flow means that data flows in one direction from parent to child components

Code and Folder Structure

- Always advisable to create a decent folder structure
- Can have a folder called screens for all the viewable screens
- Can have a folder called components for all the components

> Note: Every time you move a file or folder within different directories, make sure to check the imports and update them accordingly.

- Always try to use one component per file although it's not compulsory
- Use destructuring to get the styles when there are a lot of styles

```
const styles = StyleSheet.create({
  container: {
    backgroundColor: 'dodgerblue',
    flex: 1,
  },

  text: {
    color: 'white',
    fontSize: 20,
  },
});

const { container, text } = styles;

<View style={container}>
  <Text style={text}>Component</Text>
</View>;
```

Passing Multiple Styles

- We can pass multiple styles to a component
- We can pass an array of styles to the `style` prop of the component

```
<View style={[styles.container, { backgroundColor: 'tomato' }]}>
  <Text style={[styles.text, styles.titleText]}>Title</Text>
  <Text style={[styles.text, styles.subtitleText]}>Subtitle</Text>
</View>;

const styles = StyleSheet.create({
  container: {
    backgroundColor: 'dodgerblue',
    flex: 1,
  },

  text: {
    color: 'white',
    fontSize: 20,
  },

  titleText: {
    fontWeight: 'bolder',
  },

  subtitleText: {
    fontWeight: 'bold',
  },
});
```

This is done to avoid writing the same style again and again. Otherwise, we would have to write the same styles for each component which is not a good practice.

```
<View style={styles.container}>
  <Text style={styles.titleText}>Title</Text>
  <Text style={styles.subtitleText}>Subtitle</Text>
</View>;

const styles = StyleSheet.create({
  container: {
    backgroundColor: 'dodgerblue',
    flex: 1,
  },

  titleText: {
    color: 'white',
    fontSize: 20,
    fontWeight: 'bolder',
  },

  subtitleText: {
    color: 'white',
    fontSize: 20,
    fontWeight: 'bold',
  },
});
```

Navigation in React Native

In React Native, there's no built-in navigation system. We can use external libraries to handle navigation in our app. The 2 of the most popular libraries are -

1. React Navigation (Well integrated with Expo)
2. React Native Navigation (Not integrated with Expo)

React Navigation

- Install the required packages

  ```
  npm install @react-navigation/native
  ```

- Install the required expo dependencies

  ```
  npx expo install react-native-screens react-native-safe-area-context
  ```

- Wrap the app with the `NavigationContainer` component to enable navigation

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

  const App = () => {
    return (
      <NavigationContainer>
        <HomeScreen />
      </NavigationContainer>
    );
  };
  ```

Types of Navigation

There are a few different types of navigation. For example, Stack, Tab, Drawer, etc.

- **Stack Navigation:** Used to navigate between different screens by stacking them on top of each other
- **Tab Navigation:** Used to navigate between different screens using tabs at the top or bottom of the screen
- **Drawer Navigation:** Used to navigate between different screens using a drawer that slides in from the left

You can use any of these types of navigation based on your requirements. We are going to be using Tab Navigation for now.

Install Tab Navigation

- Install the required packages

  ```
  npm install @react-navigation/bottom-tabs
  ```

- Wrap the app with Tab Navigation

```
import { NavigationContainer } from '@react-navigation/native';
import { createBottomTabNavigator } from '@react-navigation/bottom-tabs';

const Tab = createBottomTabNavigator();

const App = () => {
  return (
    <NavigationContainer>
      <Tab.Navigator>
        <HomeScreen />
        <WeatherScreen />
      </Tab.Navigator>
    </NavigationContainer>
  );
};
```

- Add the required screens

```
import { createBottomTabNavigator } from '@react-navigation/bottom-tabs';

const Tab = createBottomTabNavigator();

const App = () => {
  return (
    <NavigationContainer>
      <Tab.Navigator>
        <Tab.Screen name="Home" component={HomeScreen} />
        <Tab.Screen name="Weather" component={WeatherScreen} />
      </Tab.Navigator>
    </NavigationContainer>
  );
};
```

Here **name** is the text that will show up on the tab button and **component** is the component to be rendered when the tab button is clicked.

This sets up a basic navigation in our app. We can also customize the navigation based on our requirements.

Customizing Tab Navigation

There are a bunch of different props that can be used to customize the tab navigation. Use the React native documentation for more information.

Here are a few examples -

- Active and Inactive Colors

```
<Tab.Navigator
  screenOptions={{
    tabBarActiveTintColor: 'tomato',
    tabBarInactiveTintColor: 'gray',
  }}
>
  <Tab.Screen name="Home" component={HomeScreen} />
  <Tab.Screen name="Weather" component={WeatherScreen} />
</Tab.Navigator>
```

- Tab Bar Icons

```
import { Feather } from '@expo/vector-icons';

<Tab.Navigator>
  <Tab.Screen
    name="Home"
    component={HomeScreen}
    options={{
      tabBarIcon: ({ focused }) => (
        <Feather name="home" color={focused ? 'tomato' : 'black'} />
      ),
    }}
  />
  <Tab.Screen
    name="Weather"
    component={WeatherScreen}
    options={{
      tabBarIcon: ({ focused }) => (
        <Feather name="droplet" color={focused ? 'tomato' : 'black'} />
      ),
    }}
  />
</Tab.Navigator>;
```

State

- An object that contains information about the component
- When the state of the component changes, the component re-renders
- State allows us to track the information about the component and do things based on that information

For example, let's say we have a state called `showButton`. If the value of `showButton` is `true`, we can show a button. If the value of `showButton` is `false`, we can hide the button. It's as simple as that.

```
let showButton = true;

<div>{showButton && <Button />}</div>;
```

Props vs State

Props - Let us pass data from parent to child components

State - Let us internally manage data within a component

If our app grows, we can use a state management library like Redux or MobX.

The `useState` Hook

- A hook that lets us add state to functional components
- It returns an array with 2 elements - the current state and a function that lets us update the state

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

- When using `useState`, we pass the initial value of the state that is used for the initial render of the component
- Initial state can be a number, string, boolean, object, or array
- We can use the `setState` function to update the state and when the state changes, the component re-renders
- We can also use the `useState` hook multiple times to add multiple states to a component

Here's an example of how we can use the `useState` hook to create a simple counter.

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

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

  return (
    <View>
      <Text>{count}</Text>
      <Button title="Increase" onPress={() => setCount(count + 1)} />
      <Button title="Reset" onPress={() => setCount(0)} />
      <Button title="Decrease" onPress={() => setCount(count - 1)} />
    </View>
  );
};
```

The `useEffect` Hook

- A hook that lets us perform side effects in functional components
- Side effects are things like data fetching, subscriptions, or manually changing the DOM
- By default, it runs after every render of the component but we can also specify when it should run

```
useEffect(() => {
  // Side effect
}, [dependency]);
```

- The second argument is an array of dependencies
- If the dependencies change, the effect runs
- If the dependencies don't change, the effect doesn't run

Here's an example of how we can use the `useEffect` hook to perform a side effect.

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

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

  useEffect(() => {
    console.log('Effect');
  }, [count]);

  return (
    <View>
      <Text>{count}</Text>
      <Button title="Increase" onPress={() => setCount(count + 1)} />
      <Button title="Reset" onPress={() => setCount(0)} />
      <Button title="Decrease" onPress={() => setCount(count - 1)} />
    </View>
  );
};
```

Here, the `useEffect` hook runs after every render of the component. If the value of `count` changes, the effect runs. If the value of `count` doesn't change, the effect doesn't run.

Cleanup Function

- We can also return a function from the `useEffect` hook
- This function is used to clean up the effect
- It runs before the effect runs next time or when the component is unmounted

```
useEffect(() => {
  console.log('Effect');

  return () => {
    console.log('Cleanup');
  };
}, [count]);
```

This isn't needed for every effect but it's useful when we need to clean up something like a subscription or a timer.

Activity Indicator

- A component used to show a loading indicator
- It's a spinner that shows that something is happening
- Can be customized using the `size` and `color` props

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

const ComponentName = () => {
  return <ActivityIndicator size="large" color="dodgerblue" />;
};
```

GeoLocation

We can use any external library to get the user's location. However, we are going to be using the `expo-location` library which comes pre-installed with Expo.

- Install the required package

  ```
  npx expo install expo-location
  ```

- Import the required module

  ```
  import * as Location from 'expo-location';
  ```

- Ask for permission to access the user's location

  ```
  const getLocation = async () => {
    const { status } = await Location.requestForegroundPermissionsAsync();

    if (status !== 'granted') {
      console.log('Permission denied');
      return;
    }
  };
  ```

- Use the `getCurrentPositionAsync` method to get the user's location

  ```
  const getLocation = async () => {
    const location = await Location.getCurrentPositionAsync();
    console.log(location);
  };
  ```

- The `getCurrentPositionAsync` method returns an object with the user's location

  ```
  {
    "coords": {
      "latitude": 37.785834,
      "longitude": -122.406417,
      "altitude": 0,
      "accuracy": 65,
      "altitudeAccuracy": -1,
      "heading": -1,
      "speed": -1
    },
    "timestamp": 1631533200000
  }
  ```

Environment Variables

Environment variables are used to store sensitive information like API keys, database URLs, etc. We can use a package called `react-native-dotenv` to use environment variables in our app.

- Install the required package

```
npm install -D react-native-dotenv
```

- Add the required configuration to the `babel.config.js`

```
module.exports = function (api) {
  api.cache(true);
  return {
    presets: ['babel-preset-expo'],
    plugins: ['module:react-native-dotenv'], // Add this line
  };
};
```

- Create a `.env` file in the root of the project

```
API_KEY=<some-secret-api-key>
```

- Use the environment variable in the app

```
import { API_KEY } from '@env';

console.log(API_KEY);
```

- Add the `.env` file to the `.gitignore` file

```
.env
```

Data Fetching

We can use the `fetch` API to fetch data from an API.

```
const getData = async () => {
  try {
    const response = await fetch('https://api.example.com/data');
    const data = await response.json();
    console.log(data);
  } catch (error) {
    console.log(error);
  }
};
```

We can also use the `axios` library to fetch data from an API.

- Install the required package

```
npm install axios
```

- Use the `axios` library to fetch data from an API

```
import axios from 'axios';

const getData = async () => {
  const response = await axios.get('https://api.example.com/data');
  console.log(response.data);
};
```

Passing Props to Tab Screens

To pass props to the component, we need to render a callback function that returns the component.

```
<Tab.Screen
  name="Home"
  component={() => <HomeScreen data={data} />}
```

Or we can also do it like this -

```
<Tab.Screen name="Home">{() => <HomeScreen data={data} />}</Tab.Screen>
```

Context API

It's a way to pass data down the component tree without having to pass props down manually at every level. It can be pretty useful when we have a lot of nested components that need to access the same data.

To use the Context API, we need to create a context and then use the `Provider` and `Consumer` components to provide and consume the data.

- Create a context

```
import { createContext } from 'react';

const DataContext = createContext();

export default DataContext;
```

- Wrap the app with the `Provider` component

```
import DataContext from './DataContext';

const App = () => {
  return (
    <DataContext.Provider value={{ data: 'Hello' }}>
      <HomeScreen />
    </DataContext.Provider>
  );
};
```

- Consume the data using the `useContext` hook

```
import DataContext from './DataContext';

const HomeScreen = () => {
  const { data } = useContext(DataContext);

  return <Text>{data}</Text>;
};
```

While Context API is useful, it doesn't mean that we should use it everywhere. The use case depends on the requirements of the app.

Fonts in React Native

We can use custom fonts in our app. We can use the `expo-font` library to use custom fonts in our app.

- Install the required package

```
npx expo install expo-font
```

- Import the required module

```
import * as Font from 'expo-font';
```

- Load the font

```
const loadFont = async () => {
  await Font.loadAsync({
    'open-sans': require('./assets/fonts/OpenSans-Regular.ttf'),
  });
};
```

- Use the font in the app

```
<Text style={{ fontFamily: 'open-sans' }}>Hello</Text>
```

We can also use the `useFonts` hook to load fonts in our app.

```
import { Stack } from 'expo-router';
import { useCallback } from 'react';
import { useFonts } from 'expo-font';
import * as SplashScreen from 'expo-splash-screen';

SplashScreen.preventAutoHideAsync();

const Layout = () => {
  const [fontsLoaded] = useFonts({
    DMBold: require('../assets/fonts/DMSans-Bold.ttf'),
    DMMedium: require('../assets/fonts/DMSans-Medium.ttf'),
    DMRegular: require('../assets/fonts/DMSans-Regular.ttf'),
  });

  const onLayoutRootView = useCallback(async () => {
    if (fontsLoaded) {
      await SplashScreen.hideAsync();
    }
  }, [fontsLoaded]);

  if (!fontsLoaded) return null;

  return <Stack onLayout={onLayoutRootView} />;
};

export default Layout;
```

Conclusion

Thank you for viewing this cheatsheet!

If you found it helpful please check out more of my work on yodkwtf.com or follow me on twitter. I also run a small youtube channel called Yodkwtf Academy.