React Native with Expo

By

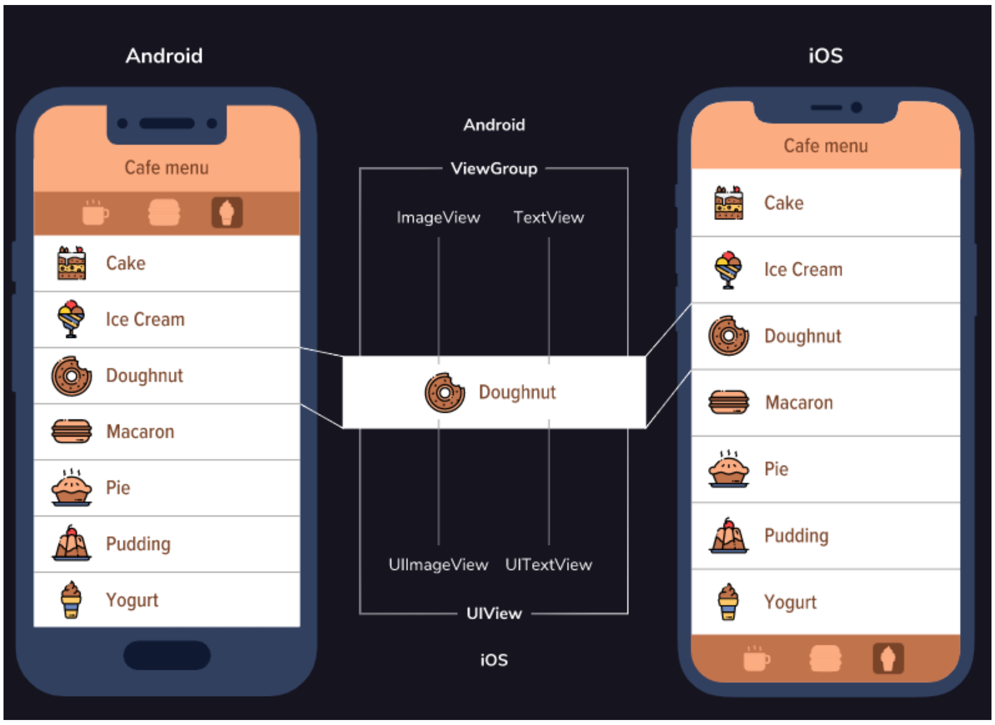
Dr. DSR Murthy

What is React Native?

React Native is a library built with the same API as React.

It uses the same *declarative UI paradigm* but renders directly to native components. Write app once in React and run it on multiple native platforms, like Android and iOS.

Android vs IOS image:



Expo and React Native still use JavaScript to run app. Instead of rendering JavaScript with a web engine, they use the actual *native components* from the platform with Hermes JS engine.

`

Note:

- React Native will use Hermes, an open-source JavaScript engine optimized for React Native.
- If Hermes is disabled, React Native will use JavaScriptCore, the JavaScript engine that powers Safari. Note that on iOS, JavaScriptCore does not use JIT due to the absence of writable executable memory in iOS apps.
- When using Chrome debugging, all JavaScript code runs within Chrome itself, communicating with native code via WebSockets. Chrome uses V8 as its JavaScript engine.
- ==

Expo and React Native

Expo is a platform to make universal React apps that helps to develop, build, deploy, and quickly iterate on mobile apps.

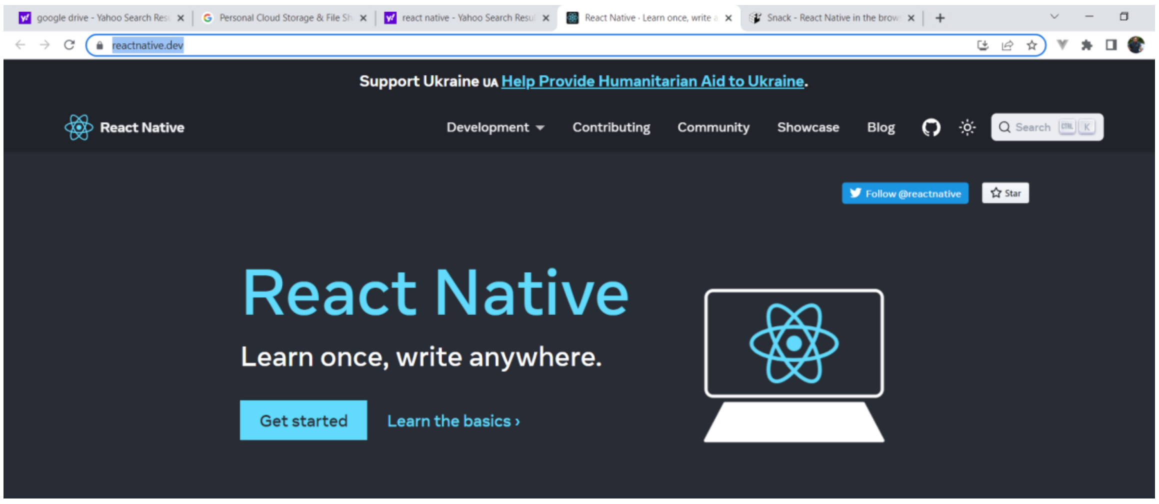
It provides a range of tooling to make development with React Native even easier.

Although Expo isn't required to create React Native apps, it helps developers by removing the need for basic native knowledge.

- Expo Go is an app. Download on mobile phone to "view" app in development.
- Expo CLI is a tool to create, manage, and develop apps.
    - npm install –g expo-cli
- Expo SDK is a modular set of packages that provide access to native APIs, like Camera or Notifications.
    - Expo Snack is a web-based playground to write React Native snippets and run them in the browser.   https://snack.expo.dev/

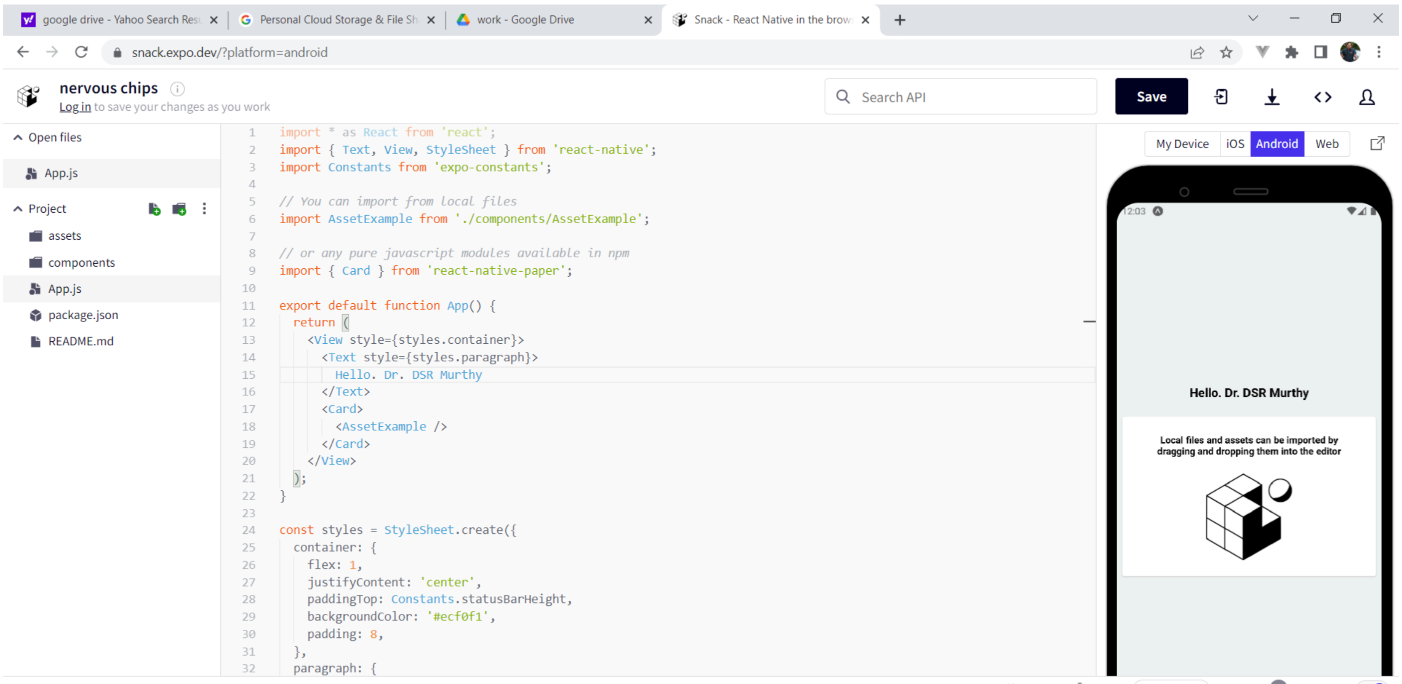
`

Visit : https://reactnative.dev/ for learning with example code.


Support Ukraine 🇺🇦 Help Provide Humanitarian Aid to Ukraine.

React Native

Learn once, write anywhere.



```
import * as React from 'react';
import { Text, View, StyleSheet } from 'react-native';
import Constants from 'expo-constants';

// You can import from local files
import AssetExample from './components/AssetExample';

// or any pure javascript modules available in npm
import { Card } from 'react-native-paper';

export default function App() {
  return (
    <View style={styles.container}>
      <Text style={styles.paragraph}>
        Hello. Dr. DSR Murthy
      </Text>
      <Card>
        <AssetExample />
      </Card>
    </View>
  );
}

const styles = StyleSheet.create({
  container: {
    flex: 1,
    justifyContent: 'center',
    paddingTop: Constants.statusBarHeight,
    backgroundColor: '#ecf0f1',
    padding: 8,
  },
  paragraph: {
```

Hello. Dr. DSR Murthy

Local files and assets can be imported by dragging and dropping them into the editor

Render a Component

All apps in Expo and React Native are made out of *components*.

These *components* are small reusable pieces of app, all working together.

Each of these components has a single responsibility.

Component renders all other components in app, from screens to simple text. Instead of rendering this component to DOM, Expo and React Native renders it using the *entry point*. (AppEntry) like index.js

`

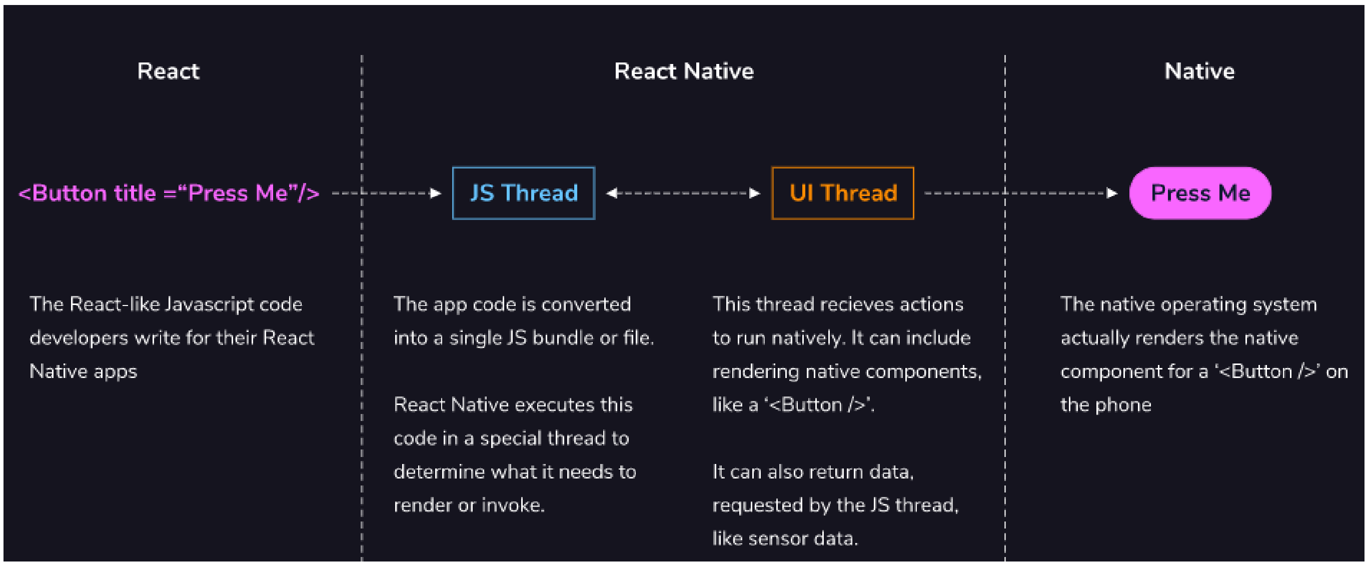
```
import React from 'react';
import { Text } from 'react-native';

const App = () => (
  <Text style={{ margin: 64 }}>
    Welcome to React Native!
  </Text>
);

export default App;
```

Native rendering

React native has Core components like <Text>.

The JavaScript with the components is bundled in app and executed in a separate thread from the native UI. This JS thread instructs React Native what it needs to render. Splitting this into a JS thread and a UI thread allows the platform to understand what needs to be rendered without blocking the actual interface components.

Besides the visible UI components, the native UI thread is also handling native API requests. Some functionality, like GPS location, needs to be requested from the native APIs. If JS code uses this kind of functionality, it interacts with the native API using *native code*. The data from this native code is sent back to the JS code and handled in app.



The React-like Javascript code developers write for their React Native apps

The app code is converted into a single JS bundle or file.

React Native executes this code in a special thread to determine what it needs to render or invoke.

This thread recieves actions to run natively. It can include rendering native components, like a '<Button />'.

It can also return data, requested by the JS thread, like sensor data.

The native operating system actually renders the native component for a '<Button />' on the phone

`

Should I use Expo and React Native?

Expo and React Native provide a comprehensive framework with a big community.

For most apps, Expo and React Native are a good choice:

- Apps built with Expo or React Native can run on multiple platforms. That means faster development and less code to maintain while sharing most of the code.
- It provides direct access to native functionality, allowing developers to make the app as performant as pure native apps.
- Getting started with Expo and React Native only requires basic web development and basic native platform understanding.

Big companies like Bloomberg, Shopify, and Coinbase are using React Native for their mobile apps. Coinbase started with native and moved to React Native with great success.

In some areas, React Native doesn't do well:

- Pure native apps have a higher performance ceiling compared to Expo and React Native apps.
- Expo and React Native are abstractions on top of the native platform. They need to follow the latest changes and functionality from the native platforms.
- Complex apps often require you to optimize and customize native code— that requires a good understanding of every platform you need to support.

`

Summary

- React Native is a library that uses React for mobile app development to create performant apps with JavaScript.
- Expo is a platform for universal React apps that contains React Native and helps you iterate fast, without any native platform knowledge.
- Components in React Native have a *native component* counterpart that is rendered on the native platform.
- Expo and React Native runs React JavaScript in the actual app, in a different thread from the UI to keep app running smoothly.
- Just like React, Expo and React Native uses the *entry point* of app to render it on different native platforms.
- Because different native platforms aren't identical, some components behave differently on some platforms.

`

Installation (Setup)

Requirements

Pre-requisites:

- Node.js @16.x - A JavaScript runtime
- Visual Studio Code - A text editor
- Expo-cli
- Google chrome
- Android studio for emulator
- Java sdk

Terminal Commands

- `node -v` - Checks version of Node.js
- `npm -v` - Checks version of Node package manager
- `npm install -g expo-cli` - Installs the Expo command-line tools
    - Expo command-line tools make it really easy to create, maintain, and test an Expo project!
- `expo init rn-app` - Creates a new Expo app
- `ce rn-app && npm start`
- `start emulator and press 'a' in metro bundler of expo to see the output`

- **Testing the App with Expo Go**
- **Install the Expo Go App**
- The Expo Go app will sync up with the Expo command-line tools that we used to create our app in the last exercise.

Ready, Set, Go!

Once you have the Expo Go app installed, let's run the React Native project. From inside the Visual Studio Code terminal, type the following:

- `npm start` - Starts the app

Now open up the Expo Go app you just installed, and scan the QR code which should have popped up in the terminal.

`

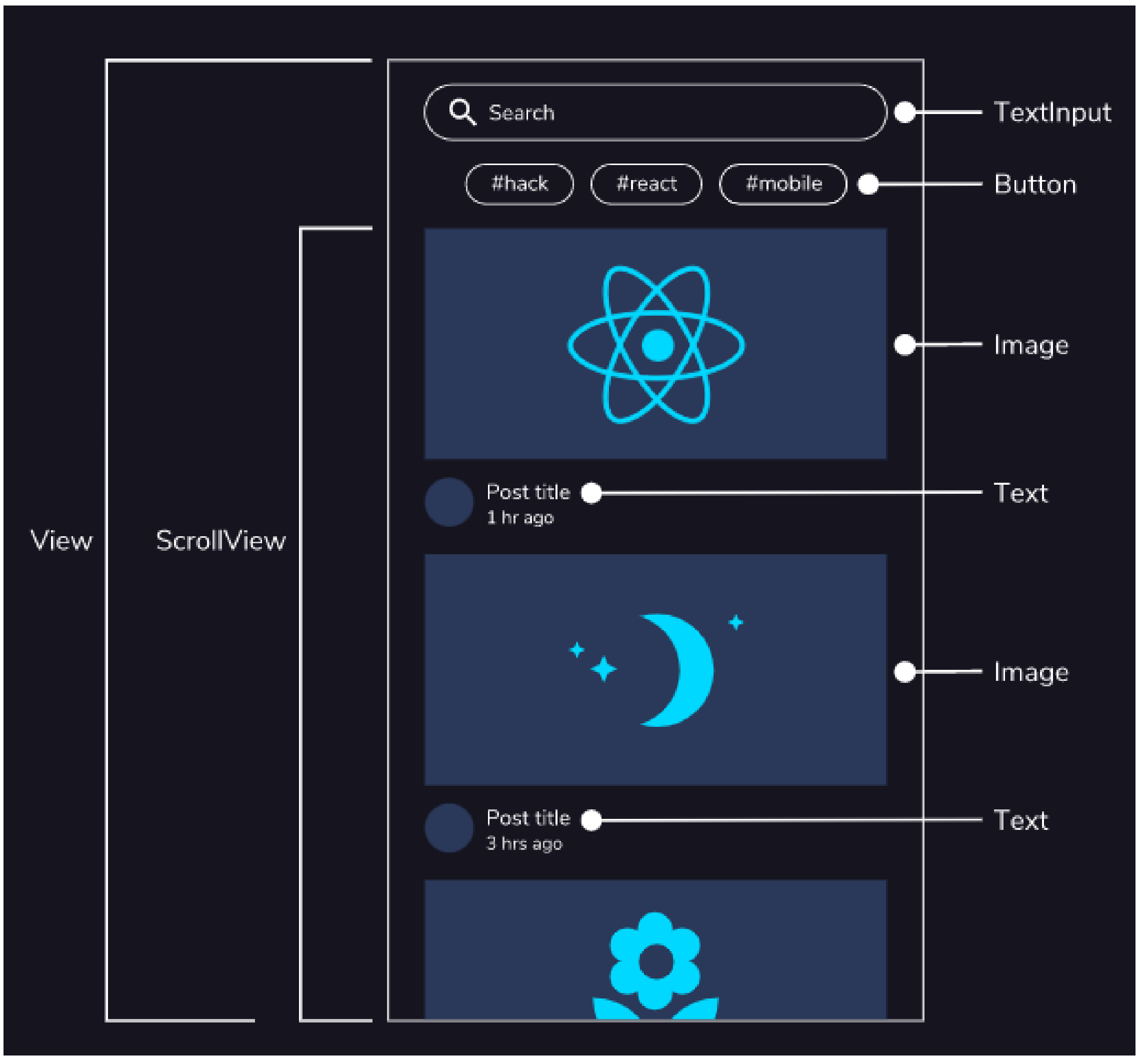
Core Components

What is a core component?

In Expo and React Native, components are translated to native components for the platform it's running on.

React Native provides a set of ready-to-use components called "core components" and most of them have built-in Android and iOS implementations.

You can import these components from the `react-native` package



`

View component

One of the most fundamental core components is the `View` component. With `View`, we can create responsive layouts using flexbox or add basic styling to nested components.

The component is best comparable with a `div` HTML element. Just like `div`, the `View` component is not visible unless styling is applied. We can apply this styling through the `style` property.

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

const App = () => (
  <View style={{ flex: 1, justifyContent: 'center', alignItems: 'center' }}>
    <View style={{ width: 100, height: 100, backgroundColor: 'red' }} />
    <View style={{ width: 100, height: 100, backgroundColor: 'blue' }} />
  </View>
);

export default App;
```

Render Image:

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

const image = require('./react-native.jpg');

const App = () => (
  <View style={{ flex: 1, justifyContent: 'center' }}>
    <Image
      source={image}
      style={{ width: 100, height: 100 }}
    />
  </View>
);

export default App;
```

`

ScrollView component

Simply apply a fixed width and height to a container to make out-of-bounds content scrollable.

`ScrollView` allows us to fully manage and customize how the content should be scrolled.

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

const App = () => (
  <View style={{ flex: 1, justifyContent: 'center' }}>
    <Text style={{ fontSize: 24, textAlign: 'center' }}>
      Scroll me!
    </Text>
    <View style={{ height: 400, backgroundColor: '#e5e5e5' }}>
      {/* This is our scrollable area */}
      <ScrollView vertical>
        <View style={{ width: 300, height: 300, backgroundColor: 'red' }} />
        <View style={{ width: 300, height: 300, backgroundColor: 'green' }} />
        <View style={{ width: 300, height: 300, backgroundColor: 'blue' }} />
      </ScrollView>
    </View>
  </View>
);

export default App;
```

Button

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

const App = () => {
  const [pressedCount, setPressedCount] = useState(0);

  return (
    <View style={{ flex: 1, justifyContent: 'center' }}>
      <Text style={{ margin: 16 }}>
        {pressedCount > 0
          ? `The button was pressed ${pressedCount} times!`
```

`

```
        : 'The button isn\'t pressed yet'
      }
    </Text>
    <Button
      title='Press me'
      onPress={() => setPressedCount(pressedCount + 1)}
      disabled={pressedCount >= 3}
    />
  </View>
 );
};

export default App;
```

TextInput for accepting Data

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

const App = () => {
  const [name, setName] = useState('');

  return (
    <View style={{
      flex: 1,
      alignContent: 'center',
      justifyContent: 'center',
      padding: 16,
    }}>
      <Text style={{ marginVertical: 16 }}>
        {name ? `Hi ${name}!` : 'What is  name?'}
      </Text>
      <TextInput
        style={{ padding: 8, backgroundColor: '#f5f5f5' }}
        onChangeText={text => setName(text)}
        secureTextEntry
      />
    </View>
  );
};

export default App;
```

`

Combining Components (TREE)

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

const App = () => (
  <View style={{ flex: 1, justifyContent: 'center' }}>
    <Box color="red" />
    <Box color="green" />
    <Box color="blue" />
  </View>
);

export default App;

export const Box = (props) => (
  // Move a box `View` component here
  <View style={{ width: 100, height: 100, backgroundColor: props.color }} />
);
```

Styling in Expo and React NativeStyling in Expo and React Native

- Inline styling
- Refactoring inline styles with StyleSheets
- Responsive layouts with unit-less values
- Common flexbox properties

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

const App = () => (
  <View style={styles.layout}>
    <View style={styles.card} />
    <View style={styles.card} />
  </View>
);

export default App;
```

`

```
export const styles = StyleSheet.create({
  layout: {
    flex: 1,
    backgroundColor: '#e5e5e5',
    justifyContent: 'center',
  },
  card: {
    width: 100,
    height: 100,
    backgroundColor: 'white',
    margin: 16,
    borderRadius: 2,
    shadowColor: 'black',
    shadowOpacity: 0.3,
    shadowRadius: 1,
    shadowOffset: { height: 1, width: 0.3 }
  },
});
```

Combining styles:

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

const App = () => (
  <View style={styles.layout}>
    <Pressable>
      {(state) => <Box pressed={state.pressed} />}
    </Pressable>
  </View>
);

export default App;

export const Box = (props) => (
  <View style={[styles.box, props.pressed && { backgroundColor: 'blue' }]} />
);

export const styles = StyleSheet.create({
  layout: {
    flex: 1,
    alignItems: 'center',
```

`

```
    justifyContent: 'center',
  },
  box: {
    width: 100,
    height: 100,
    backgroundColor: 'red',
  },
});
```

Width and Height

Expo and React Native don't always use units, like pixels (`px`), when setting dimensions or font sizes. That's because using pixels in mobile apps will cause a lot of deviations when running the app on older and newer devices.

In mobile development, it's common to use a different type of unit called "Density-independent Pixels" or `dp`. The `dp` unit takes the screen precision into account when setting the actual amount of pixels. This unit type is also the default unit for React Native when setting styling properties, such as `height`, `width`, and `fontSize`.

Flex (box)

There are a lot of different screen sizes and shapes for mobile apps. The screens can also "change shape" by rotating the device. It's important to make app responsive because of these changing factors.

The best way to make layout responsive is by using Flexbox.

All elements are flexboxes; you don't have to specify `display: flex`.

- Some default values are changed to better suit mobile apps.

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

const App = () => (
  <View style={styles.layout}>
    <View style={[styles.box, { backgroundColor: 'red' }]} />
    <View style={[styles.box, { backgroundColor: 'green' }]} />
    <View style={[styles.box, { backgroundColor: 'blue' }]} />
  </View>
);
```

`

```
export default App;

// Get the maximum width/height (in dp) from the Dimensions API
const MAX_WIDTH = Dimensions.get('window').width;
const MAX_HEIGHT = Dimensions.get('window').height;

export const styles = StyleSheet.create({
 layout: {
   flex: 1,
   backgroundColor: '#e5e5e5',
 },
 box: {
   flex: 1,
   backgroundColor: 'black',
 },
});
```

Flex Direction

we can also control the direction of the children elements within the flexbox. The default direction of the flexbox in Expo and React Native is different compared to web: it's set vertically.

There are two directions in which we can render the child elements - horizontally `row` or vertically `column`. `row` renders child elements from left to right, or *horizontally*.

- `row-reverse` renders child elements from right to left, or *reversed horizontally*.
- `column` renders child elements from top to bottom, or *vertical*.
- `column-reverse` renders child elements from bottom to top, or *reversed vertically*.

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

const App = () => (
 <View style={styles.layout}>
   <View style={[styles.box, { backgroundColor: 'red' }]} />
   <View style={[styles.box, { backgroundColor: 'yellow' }]} />
```

`

```
    <View style={[styles.box, { backgroundColor: 'blue' }]} />
  </View>
);

export default App;

export const styles = StyleSheet.create({
  layout: {
    flex: 1,
    flexDirection: 'row',
    backgroundColor: '#e5e5e5',
  },
  box: {
    flex: 1,
    backgroundColor: 'black',
  },
});
```

Flex Direction

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

const App = () => (
  <View style={styles.layout}>
    <View style={[styles.box, { backgroundColor: 'red' }]} />
    <View style={[styles.box, { backgroundColor: 'green' }]} />
    <View style={[styles.box, { backgroundColor: 'blue' }]} />
  </View>
);

export default App;

export const styles = StyleSheet.create({
  layout: {
    flex: 1,
    justifyContent: 'space-evenly',
    backgroundColor: '#e5e5e5',
  },
  box: {
    backgroundColor: 'black',
    height: 100 },});
```

`

Summary:

- Inline styling with the `style` property
- Refactoring inline styles with `StyleSheet.create`
- Dynamically styling with arrays of style objects
- Building responsive layouts with unit-less values, or “density-independent” pixels
- Laying out elements with `flex`, `flexDirection`, and `justifyContent`

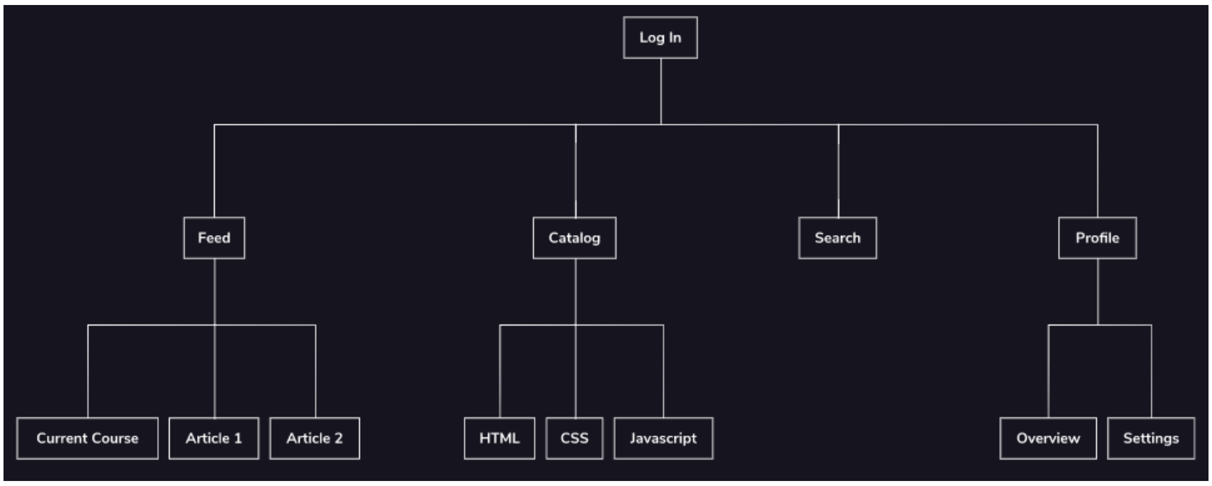
`

Navigation

- **Tab Navigation** - This pattern uses a tab bar to allow users to switch between screens.
- **Stack Navigation** - Instead of using a menu or tab bar, the user has to go from screen to screen to navigate through all screens. When a user navigates from one screen to another, the screen is pushed on a stack. The order of the stack doesn't matter; every transition is another screen in the stack. However when going back a page, the last screen is removed from the stack and the previous screen is displayed.
- **Drawer Navigation** - Instead of using a tab bar, it uses a pane that can be opened by either swiping or opening a menu button. In this pane, there is a menu where the users can switch between screens. Like the tab bar pattern, these screens often contain the primary functionality of app.

Use react-navigation

Navigation Hirearchy



```
import React from 'react';
import { StyleSheet, Text, View } from 'react-native';
import { NavigationContainer } from '@react-navigation/native';
import { createStackNavigator } from '@react-navigation/stack';

const FeedScreen = () => (
  <View style={styles.layout}>
    <Text style={styles.title}>First screen</Text>
  </View>
);
```
`

```
const Stack = createStackNavigator();

const App = () => (
  <NavigationContainer>
    <Stack.Navigator>
      <Stack.Screen name="Feed" component={FeedScreen} />
    </Stack.Navigator>
  </NavigationContainer>
);

export default App;

const styles = StyleSheet.create({
  layout: {
    flex: 1,
    justifyContent: 'center',
    alignItems: 'center',

  },
  title: {
    fontSize: 32,
    marginBottom: 16,
  },
});
```

Tab Navigation

```
import React from 'react';
import { StyleSheet, Text, View } from 'react-native';
import { NavigationContainer } from '@react-navigation/native';
import { createBottomTabNavigator } from '@react-navigation/bottom-tabs';

const FeedScreen = () => (
  <View style={styles.layout}>
    <Text style={styles.title}>Feed</Text>
  </View>
);

const CatalogScreen = () => (
  <View style={styles.layout}>
    <Text style={styles.title}>Catalog</Text>
```
`

```
  </View>
);

const Tab = createBottomTabNavigator();

export const AppNavigator = () => (
  <Tab.Navigator>
    <Tab.Screen name="Feed" component={FeedScreen} />
    <Tab.Screen name="Catalog" component={CatalogScreen} />
  </Tab.Navigator>
);

const App = () => (
  <NavigationContainer>
    <AppNavigator />
  </NavigationContainer>
);

export default App;

const styles = StyleSheet.create({
  layout: {
    flex: 1,
    justifyContent: 'center',
    alignItems: 'center',
  },
  title: {
    fontSize: 32,
    marginBottom: 16,
  },
});
```

Programattic navigation

```
import React from 'react';
import { Button, StyleSheet, Text, View } from 'react-native';
import { NavigationContainer, useNavigation } from '@react-navigation/native';
import { createStackNavigator } from '@react-navigation/stack';
```

`

```
const FeedScreen = () => {
  const navigation = useNavigation();

  return (
    <View style={styles.layout}>
      <Text style={styles.title}>Feed</Text>
      <Button
        title="Go to catalog"
        onPress={() => navigation.navigate('Catalog')}
      />
    </View>
  );
};

const CatalogScreen = () => (
  <View style={styles.layout}>
    <Text style={styles.title}>Catalog</Text>
  </View>
);

const Stack = createStackNavigator();

export const AppNavigator = () => (
  <Stack.Navigator>
    <Stack.Screen name="Feed" component={FeedScreen} />
    <Stack.Screen name="Catalog" component={CatalogScreen} />
  </Stack.Navigator>
);

const App = () => (
  <NavigationContainer>
    <AppNavigator />
  </NavigationContainer>
);

export default App;

const styles = StyleSheet.create({
  layout: {
    flex: 1,
    justifyContent: 'center',
    alignItems: 'center',
  },
  title: {
    fontSize: 32,
    marginBottom: 16,
  },
});
```

Nested Navigation

```
import React from 'react';
import { StyleSheet, Text, View } from 'react-native';
import { NavigationContainer } from '@react-navigation/native';
import { createBottomTabNavigator } from '@react-navigation/bottom-tabs';
import { createStackNavigator } from '@react-navigation/stack';

const Stack = createStackNavigator();

const OverviewScreen = () => (
  <View style={styles.layout}>
    <Text style={styles.title}>Overview</Text>
  </View>
);

const ProfileNavigator = () => (
  <Stack.Navigator>
    <Stack.Screen name="Overview" component={OverviewScreen} />
  </Stack.Navigator>
);

// Add the new stack navigator above this line

const FeedScreen = () => (
  <View style={styles.layout}>
    <Text style={styles.title}>Feed</Text>
  </View>
);

const Tab = createBottomTabNavigator();

export const AppNavigator = () => (
  <Tab.Navigator>
    <Tab.Screen name="Feed" component={FeedScreen} />
    <Tab.Screen name="Profile" component={ProfileNavigator} />
  </Tab.Navigator>
);

const App = () => (
  <NavigationContainer>
    <AppNavigator />
  </NavigationContainer>
);

export default App;

const styles = StyleSheet.create({
  layout: {
    flex: 1,
    justifyContent: 'center',
    alignItems: 'center',
  },
  title: {
    fontSize: 32,
    marginBottom: 16,
  },
});
```

Authentication Flow with useContext

```
import React, { createContext, useContext, useState } from 'react';
import { Button, StyleSheet, Text, View } from 'react-native';
import { NavigationContainer } from '@react-navigation/native';
import { createStackNavigator } from '@react-navigation/stack';

// Our global authentication state, with default values
export const AuthContext = createContext({
  hasUser: false,
  setUser: () => {},
});

const LoginScreen = () => {
  const { setUser } = useContext(AuthContext);

```

`

```
  return (
    <View style={styles.layout}>
      <Text style={styles.title}>Login</Text>
      <Button title="login" onPress={() => setUser(true)} />
    </View>
  );
};

const FeedScreen = () => {
  const { setUser } = useContext(AuthContext);

  return (
    <View style={styles.layout}>
      <Text style={styles.title}>Feed</Text>
      <Button title="logout" onPress={() => setUser(false)} />
    </View>
  );
};

const Stack = createStackNavigator();

export const AppNavigator = () => {
  const { hasUser } = useContext(AuthContext);

  return (
    <Stack.Navigator>
      {hasUser
        ? <Stack.Screen name="Feed" component={FeedScreen} />
        : <Stack.Screen name="Login" component={LoginScreen} />
      }
    </Stack.Navigator>
  );
};

const App = () => {
  // This is linked to our global authentication state.
  // We connect this in React to re-render components when changing this value.
  const [hasUser, setUser] = useState(false);

  return (
    <AuthContext.Provider value={{ hasUser, setUser }}>
      <NavigationContainer>
```
`

```
        <AppNavigator />
      </NavigationContainer>
    </AuthContext.Provider>
  );
};

export default App;

const styles = StyleSheet.create({
  layout: {
    flex: 1,
    justifyContent: 'center',
    alignItems: 'center',
  },
  title: {
    fontSize: 32,
    marginBottom: 16,
  },
});
```

Summary:

- Most common mobile navigation patterns; stack, drawer, and tab navigation
- Drawn a navigation hierarchy
- Set up the basic components for `react-navigation`
- Implementing common navigation patterns
- Connecting navigation to custom events
- Combining different navigators for more complex structures

`