Mobile Web Application

Chapter One: Introduction

React Native is a framework developed by Facebook in **2015** for creating native-style applications for Android & iOS under one common language, i.e. JavaScript. Initially, Facebook only developed React Native to support iOS. However, with its recent support of the Android operating system, the library can now render mobile UIs for both platforms.

What is React Native?

React Native is an open-source framework developed by Facebook that allows developers to build mobile applications using JavaScript and React. The key advantage of React Native is that it enables the development of apps for both iOS and Android platforms using a single codebase, which significantly reduces development time and effort.

Key Features of React Native

- **Cross-Platform Development:** Write once, run anywhere. React Native allows developers to create apps for both iOS and Android platforms using a shared codebase.
- **Native Performance:** React Native components are compiled into native code, which means that the apps deliver performance close to native apps.
- **Reusability:** React Native enables code reusability, which allows developers to use the same code for different platforms with minimal adjustments.
- **Hot Reloading:** This feature allows developers to instantly see the results of the latest change to the source code without rebuilding the app from scratch.
- **Component-Based Architecture:** Similar to React for the web, React Native uses a component-based architecture, making it easier to manage complex applications by breaking them down into smaller, reusable components.

How React Native Works?

React Native bridges the gap between JavaScript and native code. When you write code in React Native, it gets compiled into native components for iOS and Android. This means the app runs with the performance and look-and-feel of a native app.

Components of React Native

- **View:** This is the fundamental building block of UI in React Native. It maps to the native view on both iOS and Android.
- **Text:** Used for displaying text. It renders as native text elements on both platforms.
- **TextInput:** A component for text input fields.
- **ScrollView:** A component that provides a scrolling container.
- **StyleSheet:** A utility for defining styles. It works similarly to CSS.

React Native allows you to build cross-platform apps using JavaScript. It combines the best parts of native development with the React framework.

Why Learn React Native?

- React Native allows developers to create mobile apps using website technology. So, a developer who is handy in web development can easily develop a mobile app using React Native.
- React Native allows developers to build cross-platform apps that look and feel entirely Native since it uses JavaScript components that are both built on iOS and Android components.
- Since React Native uses JSX or TSX, a developer isn't required to learn complex languages. It needs Typescript to learn but is not complex language.
- React Native uses fundamental Android and iOS building blocks to compile Native apps for both platforms in JavaScript. This makes handling the code base easier.
- React Native allows you to build apps faster. Instead of recompiling, you can reload an app instantly.
- In React Native we can easily develop and test features by using various libraries and tools like Expo, ESLint, Jest, and Redux.
- On top of being responsive and providing an impressive user experience, React Native apps are faster and agile.
- React Native uses Native components which makes the rendering and execution of the app much faster

Prerequisites for Learning React Native

To begin with React-Native you should have a basic knowledge of the following technologies

- HTML, CSS and JavaScript
- ReactJS
- NodeJS Should be Installed in Your System

Installation

React Native uses Node.js, a JavaScript runtime, to build your JavaScript code. If you don't already have Node.js installed, it's time to get it!

Here we will use the **Expo CLI version** which will be much smoother to run your React Native applications. Follow the below steps one by one to setup your React native environment.

Step 1: Open your terminal and run the below command.

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

Step 2: Now expo-cli is globally installed so you can create the project folder by running the below command.

```
expo init "projectName"
```

Step 3: Now go into the created folder and start the server by using the following command.

```
cd "projectName"
npm start web
```

Modern Approach

You can use **npx** to create and manage Expo projects without globally installing Expo CLI. This method is very convenient because **npx** allows you to run **Expo CLI** directly without the need for installation.

Step 1: Create the application using the following command.

```
npx create-expo-app my-new-project
```

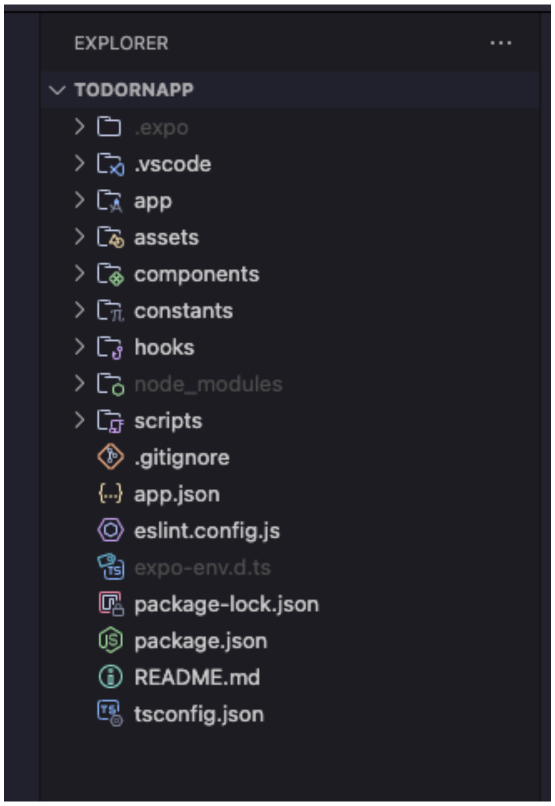
Step 2: Navigate to Your Project Folder

```
cd my-new-project
```

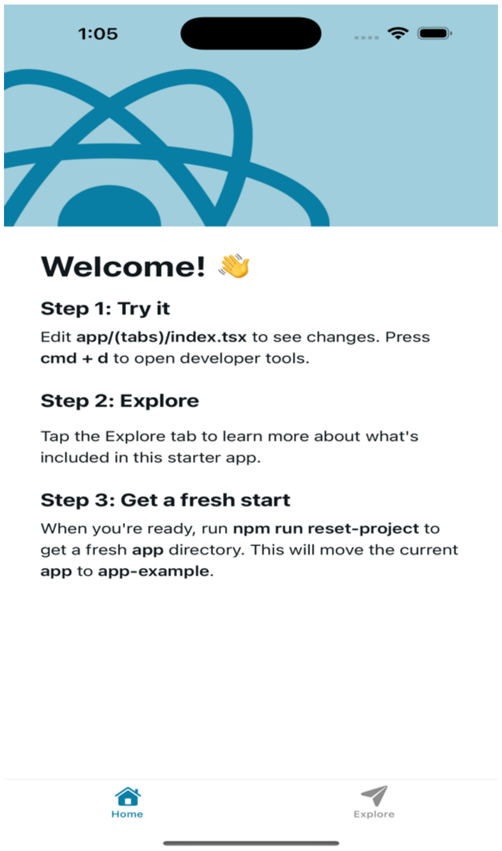
Step 3: Start the Development Server

The first app will look like

Project Structure: **Welcome page**



```
EXPLORER
TODORNAPP
> .expo
> .vscode
> app
> assets
> components
> constants
> hooks
> node_modules
> scripts
  .gitignore
  app.json
  eslint.config.js
  expo-env.d.ts
  package-lock.json
  package.json
  README.md
  tsconfig.json
```



Welcome! 👋

Step 1: Try it

Edit **app/(tabs)/index.tsx** to see changes. Press **cmd + d** to open developer tools.

Step 2: Explore

Tap the Explore tab to learn more about what's included in this starter app.

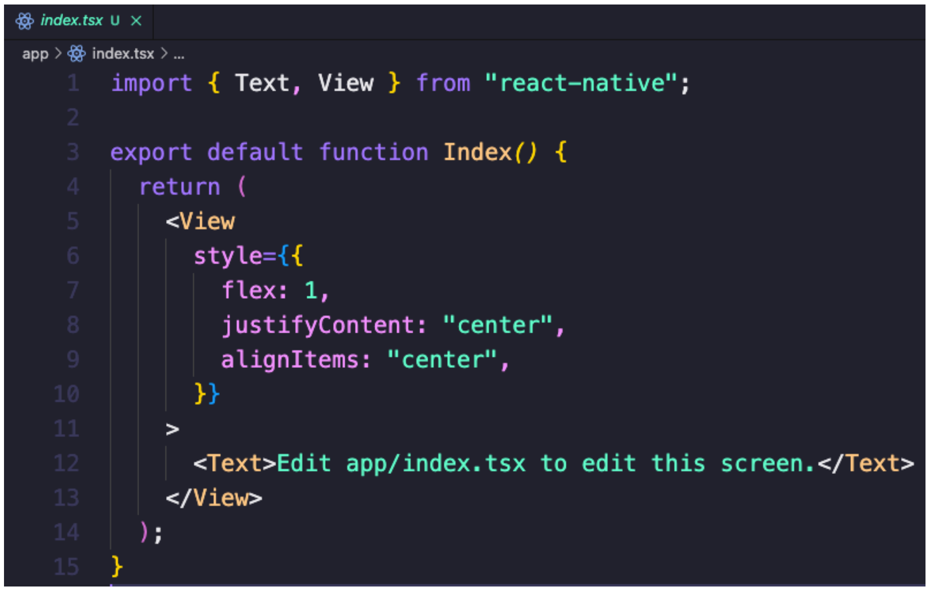
Step 3: Get a fresh start

When you're ready, run **npm run reset-project** to get a fresh **app** directory. This will move the current **app** to **app-example**.

Run this command

npm run reset-project (for typescript app)

After this command you will see index.tsx like this



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

export default function Index() {
  return (
    <View
      style={{
        flex: 1,
        justifyContent: "center",
        alignItems: "center",
      }}
    >
      <Text>Edit app/index.tsx to edit this screen.</Text>
    </View>
  );
}
```

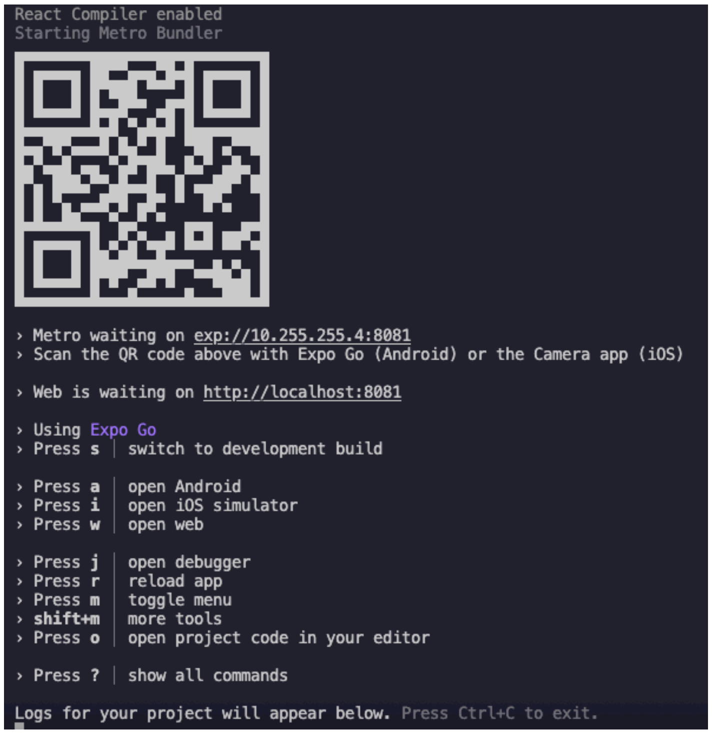
Step 3: Run the app

To start the react-native program, execute this command in the terminal of the project folder.

```
npx expo start or npx expo or npx expo –tunnel(for diff network)
```

Then, the application will display a QR code.

- For Android users, download the "Expo Go" app from the Play Store. Open the app, and you will see a button labeled "Scan QR Code." Click that button and scan the QR code; it will automatically build the Android app on your device.
- For iOS users, simply scan the QR code using the Camera app.
- If you're using a web browser, it will provide a local host link that you can use as mentioned in below image.



```
React Compiler enabled
Starting Metro Bundler

› Metro waiting on exp://10.255.255.4:8081
› Scan the QR code above with Expo Go (Android) or the Camera app (iOS)

› Web is waiting on http://localhost:8081

› Using Expo Go
› Press s │ switch to development build

› Press a │ open Android
› Press i │ open iOS simulator
› Press w │ open web

› Press j │ open debugger
› Press r │ reload app
› Press m │ toggle menu
› shift+m │ more tools
› Press o │ open project code in your editor

› Press ? │ show all commands

Logs for your project will appear below. Press Ctrl+C to exit.
```

Threads in React Native App

React Native uses multiple threads to manage the tasks involved in running an app. The main threads are:

UI Thread (Main Thread)

The **UI Thread** is responsible for rendering the user interface of the app. UI Thread, also known as **Main Thread**, is used for native android or iOS UI rendering. For example, in android this thread is used ensures that the app's interface is responsive and visually rendered.

JS Thread (JavaScript Thread)

JS thread or JavaScript thread is the thread where the logic will run. For e.g., this is the thread where the application's JavaScript code is executed, API calls are made, touch events are processed and many other. For performance, **React Native** ensures that the JS Thread sends updates to the UI Thread before the next frame rendering deadline (which is around 16.67ms for 60 frames per second on iOS).

- If the JS Thread performs complex computations that take **more than 16.67ms**, the UI can become sluggish or unresponsive. This is why the JS Thread must be optimized to avoid long delays in execution.
- Notably, some components like **NavigatorIOS** and **ScrollView** run completely on the UI Thread, which prevents them from being blocked by slow JS thread operations.

Native Modules Thread

The **Native Modules Thread** comes into play when the app needs to access platform-specific APIs (e.g., **camera, GPS)**. This thread allows React Native to call native functions from Android or iOS directly, bridging the gap between JavaScript and native code.

Render Thread (Android Only)

In Android (specifically for version L and later), the **Render Thread** is responsible for generating OpenGL commands used to draw the UI. The Render Thread enables the app to efficiently render graphics and images on the screen.

Process Involved in Working of React Native

React Native works by following a sequence of processes from app startup to rendering:

1) App Startup

On app startup, the main thread is used for executing and loading the JavaScript bundles. The JS bundles have the app's logic and UI components.

2) JavaScript Execution

After the successful loading of the JavaScript bundle, the **JS Thread** takes over the execution. This enables the JS Thread to execute the heavy computations without influencing the UI Thread, which keeps the user interface responsive.

3) Virtual DOM and Reconciliation
React Native employs Reconciliation to effectively render UI updates. The Reconciler then contrasts the current **virtual DOM** with the new virtual DOM ("**diffing**"), and where there are updates to be made, sends them off to the **Shadow Thread.**
4) Layout Calculation
The Shadow Thread determines the layout of the UI components and passes the layout parameters to the UI Thread. "**Shadow**" in this case is the virtual UI that is created during this calculation. The layout consists of position, size, and other characteristics of UI components.
5) Screen Rendering
As only the UI Thread is allowed to display UI on the screen, the layout data computed by the **Shadow Thread** is dispatched to the UI Thread. The UI Thread renders the final result on the screen and displays it to the user.
Parts of React Native
React Native can broadly be divided into three primary parts:
React Native - Native Side
The native side comprises the native modules and views that belong to the **Zombie** or **iOS** platform. It contains features such as the native UI and platform-specific APIs.
React Native - JS Side
The JS side is where the JavaScript code executes. This is the code you, as a developer, write using React and JavaScript. It handles app logic, state, events, and UI rendering.
React Native - Bridge
The Bridge provides a communication mechanism between the JS side and native side. Asynchronous communication is supported, meaning that JavaScript and native code will not block either thread. It enables sending updates, data, and events from JavaScript to native modules.

What do you understand by Virtual DOM?

The Virtual DOM is an in-memory representation of the DOM. DOM refers to the Document Object Model that represents the content of XML or HTML documents as a tree structure so that the programs can be read, accessed and changed in the document structure, style, and content.

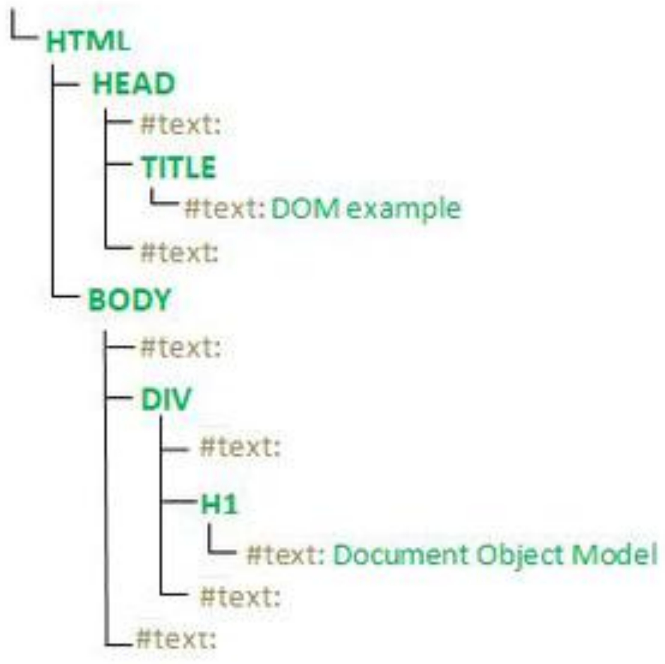
Prerequisites:

- DOM
- HTML
- React JS ReactDOM

Let's see how a document is parsed by a DOM. Consider the following sample HTML code.

```
<html>
<head>
   <title>DOM example</title>
<body>
   <div>
      <h1>Document Object Model</h1>
   </div>
</body>
</head>
</html>
```

The above code creates a tree structure as shown below:



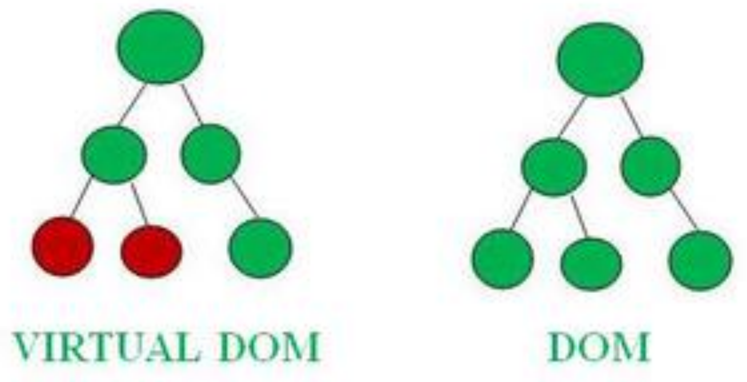
DOM Structure

Now we got a basic idea of what DOM is, If you are still unsure about it please learn here.

Virtual DOM:

The name itself says that it is a virtually created DOM. Virtual DOM is exactly like DOM and it has all the properties that DOM has. But the main difference is Whenever a code runs JavaScript Framework updates the whole DOM at once which gives a slow performance. whereas virtual DOM updates only the modified part of the DOM. Let's understand clearly:

When you run a code, the web page is divided into different modules. So, virtual DOM compares it with DOM and checks if there is any difference. If it finds a difference then DOM updates only the modified part and the other part remains the same.



As shown in the above image, virtual DOM is different from DOM, now DOM updates the child components which are different and the other remains exactly the same. This increases the performance.

Virtual DOM Key Concepts:

- Virtual DOM is the virtual representation of Real DOM
- React update the state changes in Virtual DOM first and then it syncs with Real DOM
- Virtual DOM is just like a blueprint of a machine, can do changes in the blueprint but those changes will not directly apply to the machine.
- Virtual DOM is a programming concept where a virtual representation of a UI is kept in memory synced with “Real DOM” by a library such as React DOM and this process is called reconciliation
- Virtual DOM makes the performance faster, not because the processing itself is done in less time. The reason is the amount of changed information – rather

than wasting time on updating the entire page, you can dissect it into small elements and interactions

React JS vs React Native

Differences between React and React Native:

Category	React JS	React Native
Definition	A JavaScript library, widely used for developing the user interface.	A cross-platform mobile framework used for developing native mobile applications.
Platform	Since it is mainly used for web browsers, it can be easily executed on all platforms.	Since it is used for native applications, it takes a sufficient amount of developer effort to be customized and executed on all platforms.
User Interface	ReactJS renders HTML tags in its user interface. React components can include simple HTML tags.	React Native renders JSX in its user interface. React Native supports specific JSX tags that are used.
Styling	ReactJS uses Cascading Style Sheets (CSS).	React Native uses a Stylesheet object (JavaScript object).

Category	React JS	React Native
Rendering	ReactJS uses VirtualDOM, a tool that allows for easy interaction with DOM elements.	React Native widely uses native APIs.
Navigation	ReactJS uses the React router to allow users to visit different web pages.	React Native uses its built-in Navigator library to allow users to visit different screens.
External library support	ReactJS supports third-party packages but lacks native library support.	React Native lacks both native libraries and third-party packages.
Animation	Since ReactJS focuses on UI, it requires animations, which can be easily added using CSS.	To incorporate animations in React Native, it uses an animated API.
Security	It has comparatively higher security.	It has comparatively lower security.
Uses	It is widely used to develop a dynamic user interface for web applications.	It is used to develop true native mobile applications.
Applications	Facebook, Netflix, Medium, Udemy	Uber Eats, Tesla

React Native Debugging

Debugging is very important for building applications and removing errors. A good knowledge of debugging techniques allows for the faster and efficient development of software.
Here we are going to discuss a few debugging techniques in React Native. We will be using *expo-cli* to develop, run, and debug our applications, which is one of the simplest and fastest ways of building a React Native application.
The following are the debugging techniques generally used in React Native:
Logging
It is a very quick and easy technique to debug your application in the development phase. It is one of the easiest techniques to get an insight into the functioning of the application. To do logging, we simply use the console.log() statements to log the required information or indicators. However, we should always remember to remove these console.log() statements before we push our product into the development phase, as these statements will simply create an overhead there.

Chapter Two: React Native Components

Component
We know that React Applications are a collection of interactive components. With React Native, 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 Hooks API, you can add state and more to function components. The Components are the building blocks in React Native, which are similar to the container where all UI elements are gathered and help in rendering details in the foreground as a native UI on either Android or iOS devices. React comes with multiple built-in components such as Text, View, Image, ScrollView, TextInput, etc.

Importing Component
The world of JavaScript is always moving, and one of the latest ECMAScript versions now provides a more advanced module importing pattern. In the previous version, the developer had to use the following command.

```
module. exports = { // Define your exports here. };
const module = require('./file');
```

But now each module can have a default export or may export several named parameters, and if it is possible to export, it will surely be possible to import the same. Thus, with the recent ECMAScript version, every module may import the default export or several named parameters or even a valid combination.

Approach

React Native uses the same features as mentioned above, and you may treat each React Component as a module itself. Thus, it is possible to import React Native Components, and it is one of the basic operations to be performed. In React, we use the keyword **import** and **from** to import a particular module or a named parameter. Let's see the different ways we can use the import operation in a React Native Application.

Importing the default export

Each module in reacts native needs at least one default export. In order to import the default export from a file, we can use the location of the file and use the keyword import before it, or we could give a specific name i.e. COMP_NAME to the import which makes the syntax as the following.

```
import COMP_NAME from LOCATION
```

Importing named values

Every module can have no named parameters, and in case we need to import one we should use the syntax as follows.

```
import { COMP_NAME } from LOCATION
```

Similarly, for multiple imports, we can use a comma (**,**) separator to separate two-parameter names within the curly braces. As shown below.

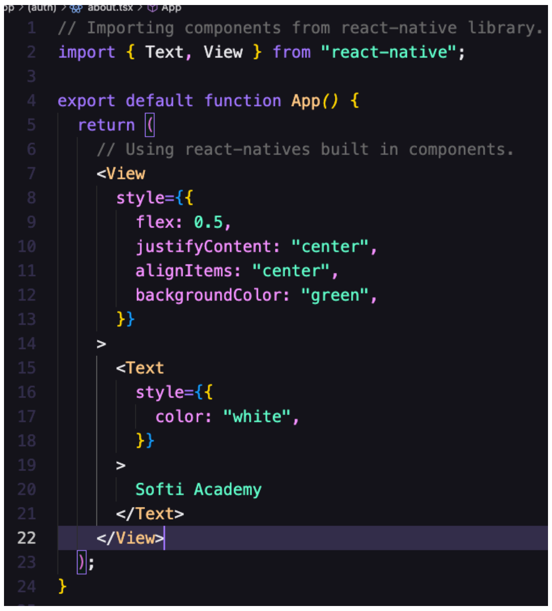
```
import { COMP_NAME1, COMP_NAME2, ... , COMP_NAMEn } from LOCATION
```

Importing a combination of Default Exports and Named Values

The title makes it clear **that** what we need to see is the syntax of the same. In order to import a combination, we should use the following syntax.

```
import GIVEN_NAME, { PARA_NAME, ... } from ADDRESS
```

Example:



```
// Importing components from react-native library.
import { Text, View } from "react-native";

export default function App() {
  return (
    // Using react-natives built in components.
    <View
      style={{
        flex: 0.5,
        justifyContent: "center",
        alignItems: "center",
        backgroundColor: "green",
      }}
    >
      <Text
        style={{
          color: "white",
        }}
      >
        Softi Academy
      </Text>
    </View>
  );
}
```

Stack Component

A **Stack** is a navigation layout in Expo Router that organizes your screens in a **stack-based navigation**, where screens slide in on top of each other.
It works like a **page history**, allowing users to go forward and backward between screens.

Key Points

- Part of **Expo Router** (file-based navigation).
- Uses **folders and file names** to create screens automatically.
- Stack controls **header**, **animations**, and **navigation style**.
- Each screen becomes a **stack route**.
- Allows customization using <Stack.Screen />.

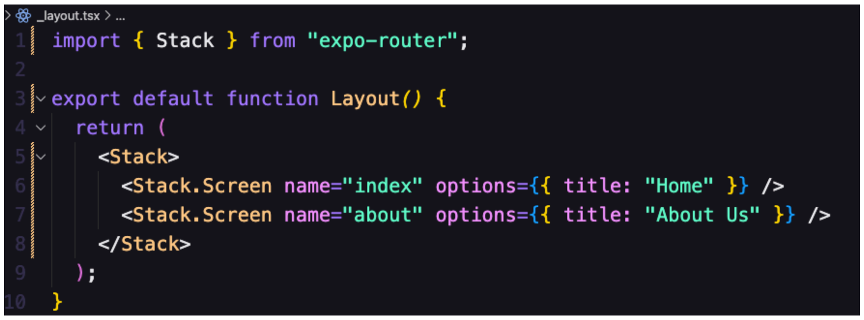
Usage

You place a Stack component inside your **_layout.tsx** file.
This layout controls how pages inside the folder are displayed.

Example folder structure:

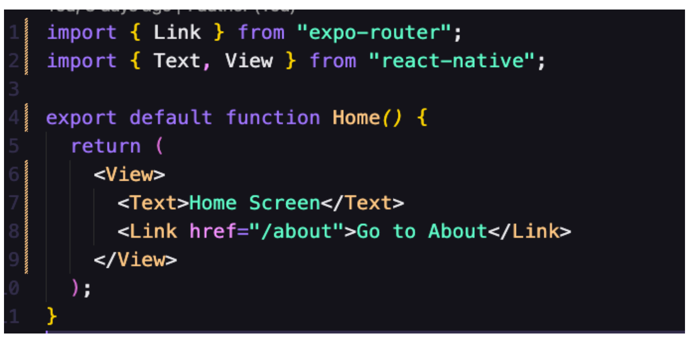
```
app/
 ├── _layout.tsx
 ├── index.tsx
 └── about.tsx
```

Index.tsx


_layout.tsx > ...

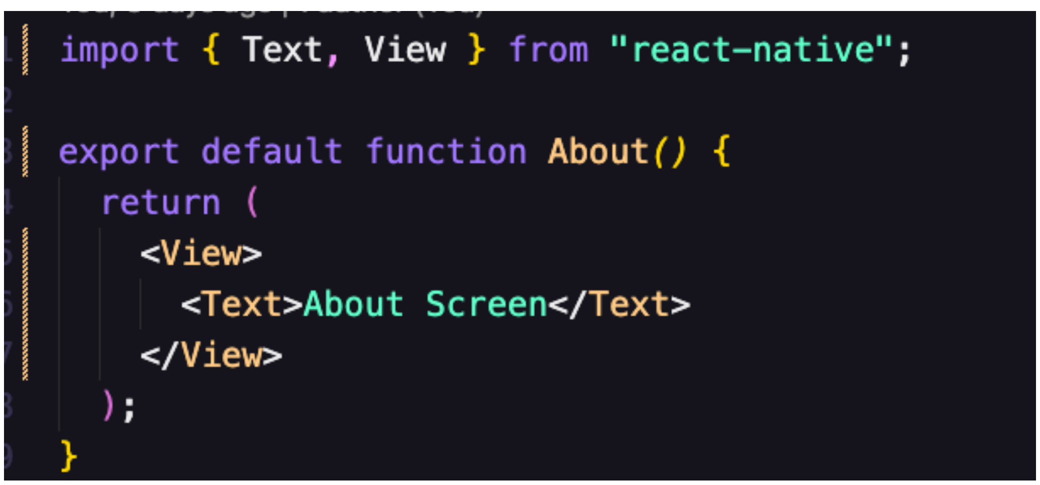
```tsx
import { Stack } from "expo-router";

export default function Layout() {
  return (
    <Stack>
      <Stack.Screen name="index" options={{ title: "Home" }} />
      <Stack.Screen name="about" options={{ title: "About Us" }} />
    </Stack>
  );
}
```

index.tsx



```tsx
import { Link } from "expo-router";
import { Text, View } from "react-native";

export default function Home() {
  return (
    <View>
      <Text>Home Screen</Text>
      <Link href="/about">Go to About</Link>
    </View>
  );
}
```

about.tsx



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

export default function About() {
  return (
    <View>
      <Text>About Screen</Text>
    </View>
  );
}
```

Props (Options) for Stack & Stack.Screen

🔶 Stack Props

Prop	Description
screenOptions	Default options for all screens like header, animations
initialRouteName	First screen to show (rare in Expo Router because folders decide)

Example:

<Stack screenOptions={{ headerStyle: { backgroundColor: "#222" } }} />

🔶 Stack.Screen Props

Option	Description
name	Screen filename (without .tsx)
options	Customize the screen
options.title	Change header title
options.headerShown	Show/hide header
options.headerStyle	Header background
options.animation	Change screen animation
options.presentation	modal / card / transparent modal
options.headerBackVisible	Show or hide the back button
options.headerTintColor	Color of back button + title

🎬 Animation Options

Value	Effect
"default"	normal stack animation
"fade"	fade between screens
"slide_from_right"	iOS-style push
"slide_from_bottom"	modal style

Tab Navigation

Definition

Tabs show **icons & labels** at the **bottom of the screen**.
In Expo Router, tabs are created using a folder group (tabs) with _layout.js.

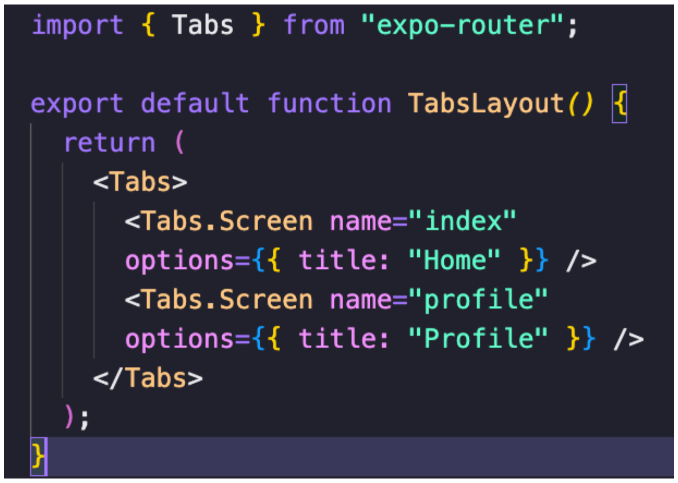
Key Points

- Use (tabs)/_layout.js to create tab navigation.
- Each file inside the (tabs) folder becomes a tab.
- Easy to add icons & titles.
- Tabs can contain Stack screens.

Folder Structure:

app/

```
├─ (tabs)/
|    ├─ _layout.js
|    ├─ index.js
|    ├─ profile.js
```



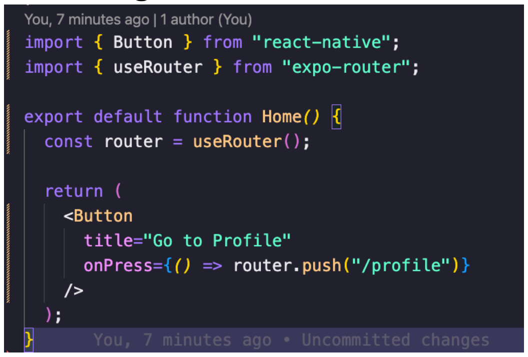
```
import { Tabs } from "expo-router";

export default function TabsLayout() {
  return (
    <Tabs>
      <Tabs.Screen name="index"
      options={{ title: "Home" }} />
      <Tabs.Screen name="profile"
      options={{ title: "Profile" }} />
    </Tabs>
  );
}
```

Navigation (Pushing from one screen to another)

Key Points

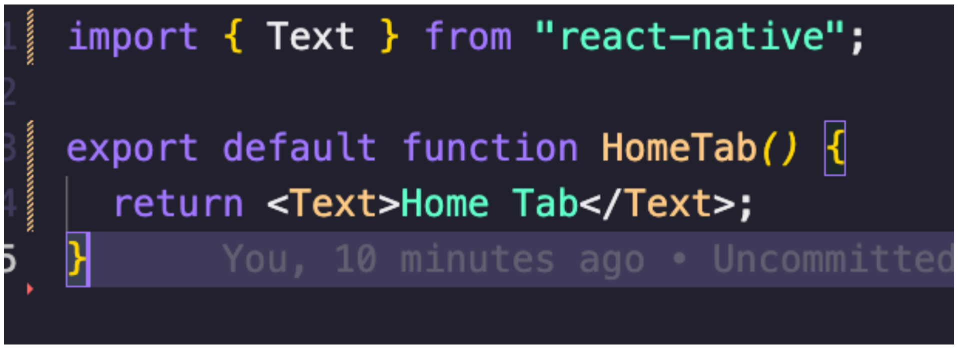
- Use useRouter() for navigation.
- router.push("/path")
- router.replace("/login")
- router.back()

1. Index Page

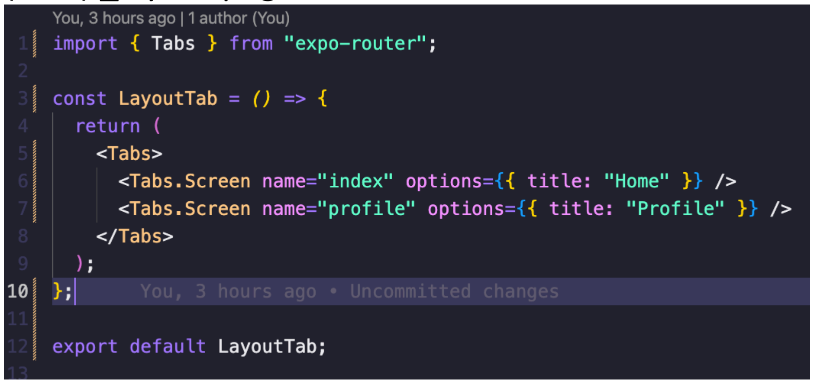


```
You, 7 minutes ago | 1 author (You)
import { Button } from "react-native";
import { useRouter } from "expo-router";

export default function Home() {
  const router = useRouter();

  return (
    <Button
      title="Go to Profile"
      onPress={() => router.push("/profile")}
    />
  );
}    You, 7 minutes ago • Uncommitted changes
```

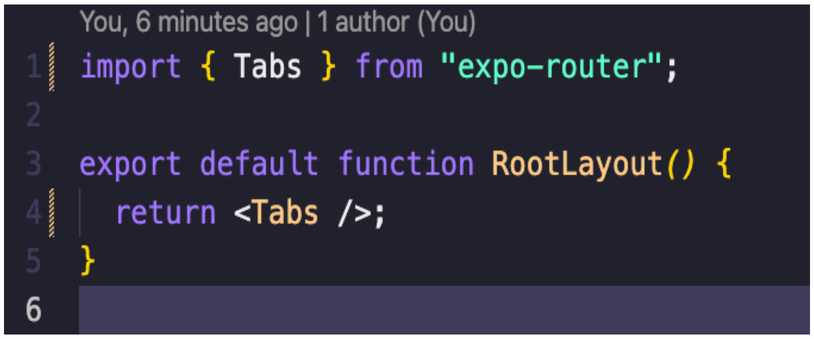
2. Profile Tab


```
import { Text } from "react-native";

export default function HomeTab() {
  return <Text>Home Tab</Text>;
}        You, 10 minutes ago • Uncommitted
```

3. (tabs) _layout page


```
You, 3 hours ago | 1 author (You)
import { Tabs } from "expo-router";

const LayoutTab = () => {
  return (
    <Tabs>
      <Tabs.Screen name="index" options={{ title: "Home" }} />
      <Tabs.Screen name="profile" options={{ title: "Profile" }} />
    </Tabs>
  );
};        You, 3 hours ago • Uncommitted changes

export default LayoutTab;
```

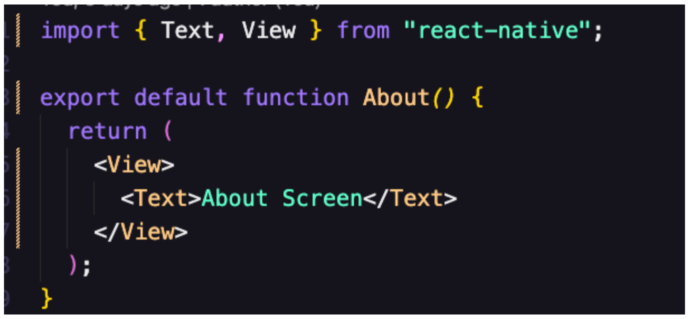
4. App/ index page


```
You, 6 minutes ago | 1 author (You)
import { Tabs } from "expo-router";

export default function RootLayout() {
  return <Tabs />;
}
```

View Component

Definition

View is the **basic container component** in React Native used to build layouts.
It works like a **div** in web development and is used to organize, wrap, and style UI elements.


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

export default function About() {
  return (
    <View>
      <Text>About Screen</Text>
    </View>
  );
}
```

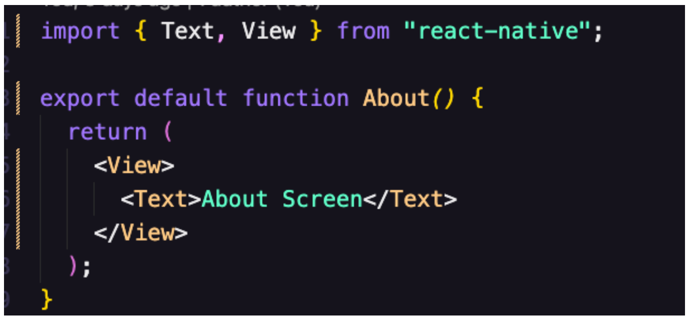
Key Points

- The **main layout component** in React Native.
- Used for **structuring** your UI (rows, columns, boxes, sections).
- Supports **flexbox** for layout design.
- Can contain **other Views**, Text, Buttons, Images, etc.
- Useful for **styling**, **spacing**, **alignment**, **backgrounds**, and **borders**.
- Does **not** display text—only acts as a container.
- Lightweight and very commonly used.

Text Component

Definition

Text is the component used to **display text strings** in React Native.
It renders readable content like titles, paragraphs, labels, buttons, etc.



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

export default function About() {
  return (
    <View>
      <Text>About Screen</Text>
    </View>
  );
}
```

Key Points

- Only component used to **render text** in React Native.
- Supports **styling** like color, font size, weight, alignment.
- Can contain other **Text** components inside it.
- Can handle **touch events** (onPress).
- Text does **not** automatically wrap unless styled properly.
- Great for headings, labels, descriptions, and inline text.

Chapter Three: Typescript, Data types and Props

TypeScript Basics

What is TypeScript?

TypeScript is a strongly typed superset of JavaScript that adds static typing, type checking, interfaces, and other features to help developers write safer and more maintainable code. TypeScript code is compiled into regular JavaScript so it can run anywhere JavaScript runs.

Basic Data Types

Here are some commonly used TypeScript primitive data types:
- string:

```
let name: string = "John";
```

- number:

```
let age: number = 30;
```

- boolean:

```
let isActive: boolean = true;
```

- any:

```
let value: any = 10;
```

- unknown:

```
let data: unknown = 'Hello';
```

- null:

```
let empty: null = null;
```

- undefined:

```
let notAssigned: undefined = undefined;
```

- array:

```
let numbers: number[] = [1, 2, 3];
```

- tuple:

```
let person: [string, number] = ['Alice', 25];
```

User-Defined Types

1. Interface

Interfaces define the structure of an object. They describe property names and their corresponding types.
Example:

```
interface User {
  id: number;
  name: string;
  isAdmin: boolean;
}

const user: User = {
  id: 1,
  name: 'Alice',
  isAdmin: false,
};
```

---

2. Type Alias

Type aliases let you create custom types for primitives, objects, unions, and more.
Example:

```
type ID = number | string;
type Product = {
  id: ID;
  title: string;
  price: number;
};

const item: Product = {
  id: 101,
  title: 'Laptop',
  price: 999,
};
```

---

3. React Native Components Using Props and Children

3.1 Custom Button Component (Props)

Example:

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

type MyButtonProps = {
  title: string;
  onPress: () => void;
};

export function MyButton({ title, onPress }: MyButtonProps) {
  return (
    <TouchableOpacity
      style={{ backgroundColor: "blue", padding: 10, borderRadius: 8 }}
      onPress={onPress}
    >
      <Text style={{ color: "white" }}>{title}</Text>
    </TouchableOpacity>
  );
}
```

3.2 Card Component Using Children

Example:

```
import { View } from "react-native";
import { ReactNode } from "react";

type CardProps = {
  children: ReactNode;
};

export function Card({ children }: CardProps) {
  return (
    <View
      style={{
        padding: 15,
        borderRadius: 10,
        backgroundColor: "#f2f2f2",
        marginVertical: 10,
      }}
    >
      {children}
    </View>
  );
}
```

3.3 Combined Usage Example

Example Usage:

```
export default function App() {
  return (
    <Card>
      <MyButton title="Click Me" onPress={() => console.log("Pressed!")} />
    </Card>
  );
}
```

Props (short for *properties*) are values passed **from a parent component to a child component**.
They allow components to become **reusable**, **dynamic**, and **configurable**.

Think of props like **parameters passed to a function**.

Key Points

- Props are **read-only** → A child **cannot modify** the props it receives.
- Props make components **reusable** with different values.
- Props can be **strings, numbers, booleans, arrays, objects, functions**, or even **components**.
- Props are passed using **JSX attributes**.
- Inside a component, props are accessed using:

Chapter Four: Event Handlers and List Components

Definition

A **button** in React Native is:

- A **touchable element** that triggers a function when pressed.

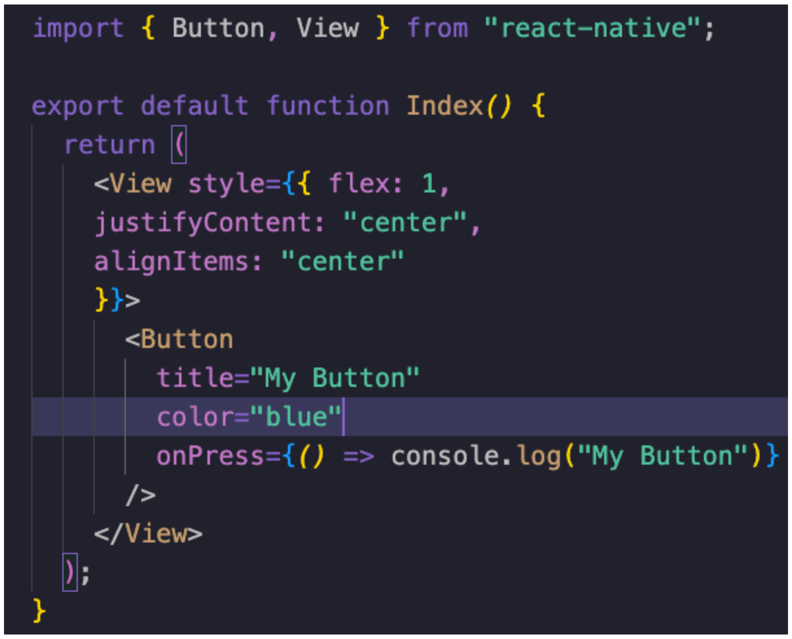
- Used for **navigation**, **submitting forms**, **interacting with the UI**, or **performing any action**.
- Can be **built-in** (Button) or **custom** using Pressable, TouchableOpacity, TouchableHighlight, etc.

Key Features of Buttons

- **onPress:** Function that runs when the button is tapped.
- **Customizable appearance:** Buttons can be styled using style or children elements.
- **Feedback on touch:** Buttons usually provide visual feedback like changing color, opacity, or highlighting.
- **Accessibility:** Buttons can be labeled for screen readers.

1. Button

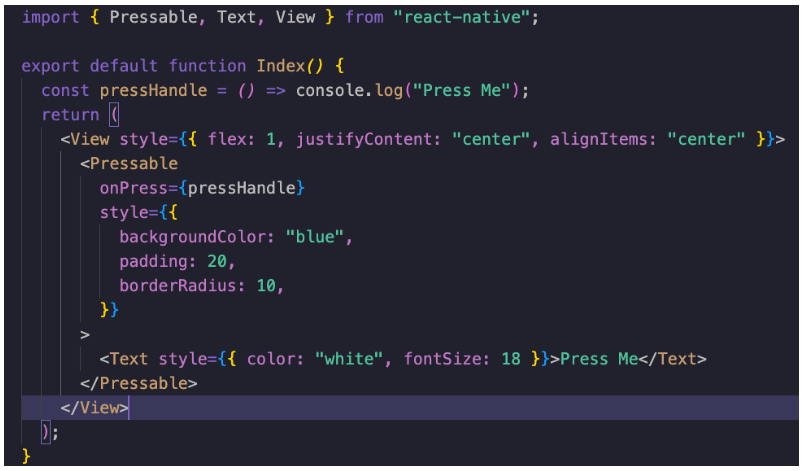
- **Built-in component** in React Native.
- Simple to use, limited customization.
- Props: title, onPress, color, disabled.



```
import { Button, View } from "react-native";

export default function Index() {
  return (
    <View style={{ flex: 1,
    justifyContent: "center",
    alignItems: "center"
    }}>
      <Button
        title="My Button"
        color="blue"
        onPress={() => console.log("My Button")}
      />
    </View>
  );
}
```

Pressable

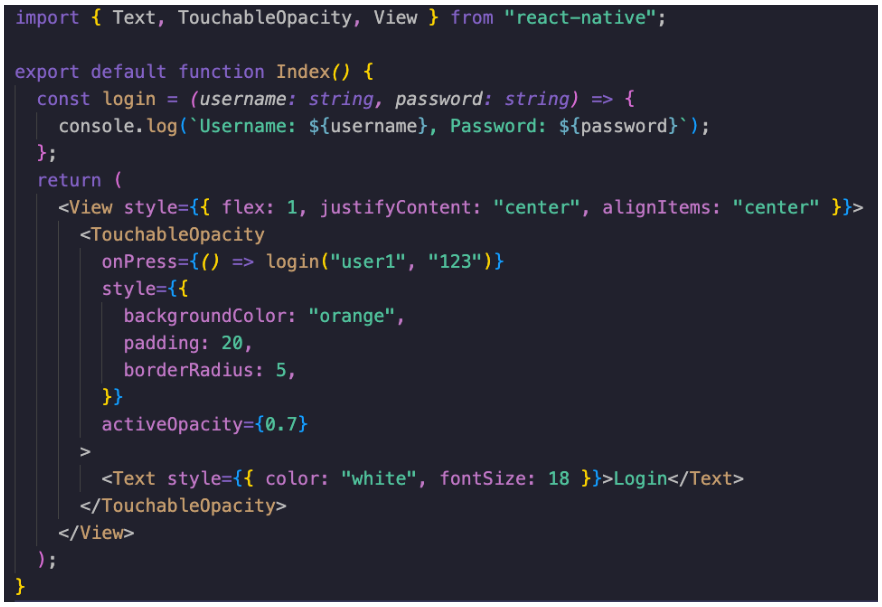
- Can detect **press in, press out, long press**.
- Can apply **custom styles** on press.



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

export default function Index() {
  const pressHandle = () => console.log("Press Me");
  return (
    <View style={{ flex: 1, justifyContent: "center", alignItems: "center" }}>
      <Pressable
        onPress={pressHandle}
        style={{
          backgroundColor: "blue",
          padding: 20,
          borderRadius: 10,
        }}
      >
        <Text style={{ color: "white", fontSize: 18 }}>Press Me</Text>
      </Pressable>
    </View>
  );
}
```

TouchableOpacity

- Changes **opacity** when pressed.
- Popular for **custom buttons**.
- Props: onPress, activeOpacity, style.



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

export default function Index() {
  const login = (username: string, password: string) => {
    console.log(`Username: ${username}, Password: ${password}`);
  };
  return (
    <View style={{ flex: 1, justifyContent: "center", alignItems: "center" }}>
      <TouchableOpacity
        onPress={() => login("user1", "123")}
        style={{
          backgroundColor: "orange",
          padding: 20,
          borderRadius: 5,
        }}
        activeOpacity={0.7}
      >
        <Text style={{ color: "white", fontSize: 18 }}>Login</Text>
      </TouchableOpacity>
    </View>
  );
}
```

Summary

- Button → simplest, less customization.
- Pressable → most modern, highly customizable.
- TouchableOpacity → easy custom buttons with opacity effect.
- TouchableHighlight → good for feeds/list items with highlight effect.

FlatList Component

FlatList is a React Native component that is a scrolling list that shows changing information while keeping the same look. It's great for long lists where the number of items can change. Instead of loading all items simultaneously, this component only shows what you can see on the screen. This makes it faster and gives users a better experience.

Syntax of FlatList:

```
<FlatList
        data={}
        renderItem={}
        keyExtractor={}
 />
```

FlatList Props:

Props	Type	Description
renderItem	function	Extracts an item from the data and displays it in the list.
data	ArrayLike	A list of items to render
ItemSeparatorComponent	component, function, element	Used to render in between each item, but not at the top or bottom. For example, a separate line between two items.
ListEmptyComponent	component, element	Used to render when the list is empty. For example, displaying "No Data Found" when the list is empty.
ListHeaderComponent	component, element	Used to render at the bottom of all the items. For example, "End of List"

Props	Type	Description
ListFooterComponent	component, element	Used to render at the bottom of all the items. For example, "Heading of the List"
ListHeaderComponentStyle	ViewStyle	Used to Style ListHeaderComponent
ListFooterComponentStyle	ViewStyle	Used to Style ListFooterComponent
columnWrapperStyle	ViewStyle	Used to customize the style for multi-item rows when numColumns > 1.
extraData	any	Used to re-render the list when the state changes.
getItemLayout	function	Used to optimize performance by providing item layout.
horizontal	boolean	Set to true for horizontal scrolling
initialNumToRender	number (default: 10)	Number of items to render initially
initialScrollIndex	number	Set the initial scroll index
inverted	boolean	Set to true for inverted scrolling

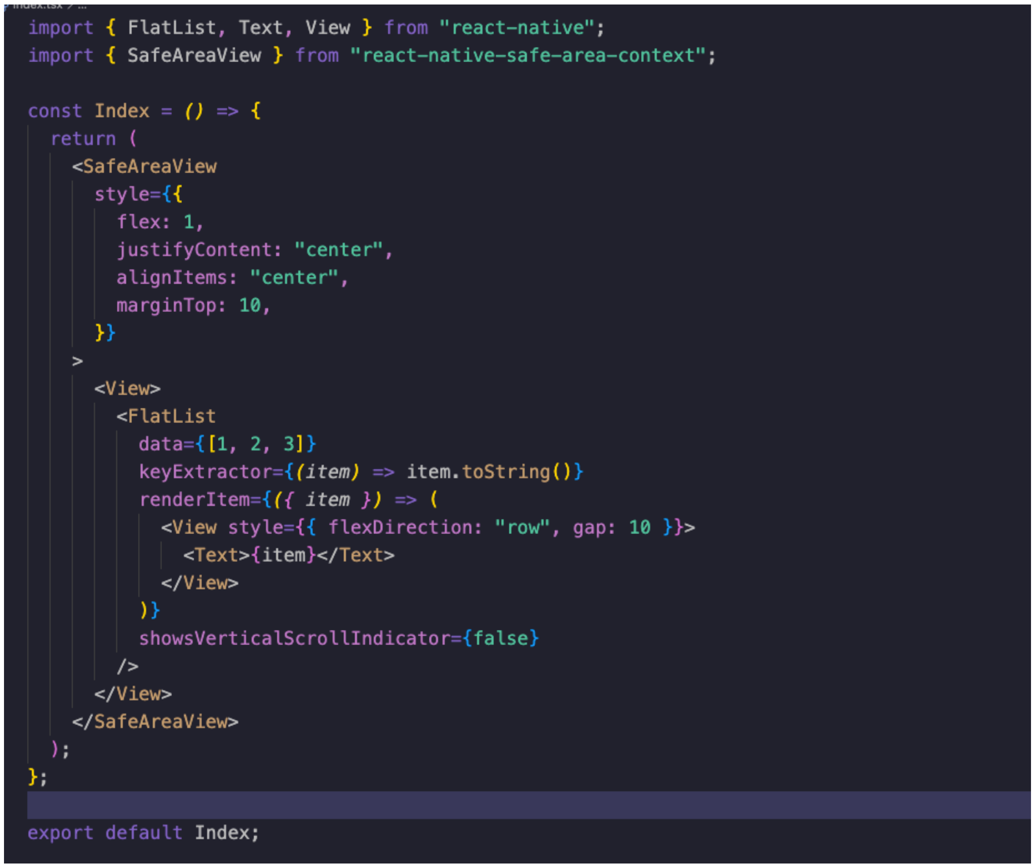
Props	Type	Description
keyExtractor	function	Function to generate unique keys for items.
numColumns	number	Set the number of columns
onRefresh	function	Used to call for pull-to-refresh; requires refreshing prop.
refreshing	boolean	Set to true during refresh loading.
removeClippedSubviews	boolean	Used to improve performance by removing invisible views (Android only).
onViewableItemsChanged	function	Used to call back when viewable items change.
viewabilityConfig	object	Used to configure for determining item viewability (e.g., visibility%, waitForInteraction).
viewabilityConfigCallbackPairs	array	It is an Array of ViewabilityConfig + callback pairs for tracking multiple configs.

FlatList Methods:

Method	Description
flashScrollIndicators()	Used to display the scroll indicators momentarily

Method	Description
getNativeScrollRef()	It provides a reference to the underlying scroll component
getScrollResponder()	It provides a handle to the underlying scroll responder
getScrollableNode()	It provides a handle to the underlying scroll node
scrollToEnd()	Scrolls to the end of the content
scrollToIndex()	Scrolls to the particular item index of the list

First install SafeAreaView React Context
npm install react-native-safe-area-context



```
import { FlatList, Text, View } from "react-native";
import { SafeAreaView } from "react-native-safe-area-context";

const Index = () => {
  return (
    <SafeAreaView
      style={{
        flex: 1,
        justifyContent: "center",
        alignItems: "center",
        marginTop: 10,
      }}
    >
      <View>
        <FlatList
          data={[1, 2, 3]}
          keyExtractor={(item) => item.toString()}
          renderItem={({ item }) => (
            <View style={{ flexDirection: "row", gap: 10 }}>
              <Text>{item}</Text>
            </View>
          )}
          showsVerticalScrollIndicator={false}
        />
      </View>
    </SafeAreaView>
  );
};

export default Index;
```

State and useState in React Native

1. What is State?

State is data that changes over time inside a component. When the state changes, React Native automatically re-renders the UI to reflect the new data.

Key Points About State

- State is mutable (can change)
- State is managed inside the component
- Updating state triggers UI re-render
- Used for dynamic UI such as counters, inputs, modals, etc.

2. What is useState?

useState is a React Hook that allows functional components to have state. It returns the current state and a function to update it.
Syntax:
```
const [value, setValue] = useState(initialValue);
```

3. Example 1: Counter

Example:
```
import { useState } from "react";
import { View, Text, Button } from "react-native";

export default function App() {
  const [count, setCount] = useState(0);

  return (
    <View>
      <Text>Count: {count}</Text>
      <Button title="Increase" onPress={() => setCount(count + 1)} />
    </View>
  );
}
```

4. Example 2: Input State

Example:
```
import { useState } from "react";
import { TextInput, Text } from "react-native";
```

```
export default function App() {
  const [name, setName] = useState("");

  return (
    <>
      <TextInput
        placeholder="Enter your name"
        value={name}
        onChangeText={setName}
      />
      <Text>Your name is: {name}</Text>
    </>
  );
}
```

5. Example 3: Toggle Visibility

Example:

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

export default function App() {
  const [visible, setVisible] = useState(true);

  return (
    <View>
      {visible && <Text>This text is visible</Text>}
      <Button title="Toggle" onPress={() => setVisible(!visible)} />
    </View>
  );
}
```

6. State vs Props

Key Differences: Props vs. State

Feature	**Props (Properties)**	**State**
Owner	**Parent Component**	**Component itself**

Mutability	**Immutable** (Read-only)	**Mutable** (Can be changed)
Purpose	Passing data/configuration **down** from parent to child components.	Managing data that **changes over time** within the component (e.g., user input, loading status).
How to Change	Can only be changed by the **parent** component.	Changed using the dedicated function: setState (in class components) or the **useState hook** (in functional components).
Analogy	**Arguments** passed to a function.	**Internal memory** of a component.

SafeAreaView

Definition

SafeAreaView is a React Native component that renders content within the safe area boundaries of a device.
This prevents UI elements from being hidden behind notches, rounded corners, or system UI (like the status bar).

Key Points

- Ensures your layout avoids device notches and rounded corners.
- Works on both **iOS and Android** (better support on iOS).
- In Expo, you should use:
  - SafeAreaView from **react-native**, OR
  - SafeAreaView from **react-native-safe-area-context** (recommended)

Why You Need SafeAreaView

On modern smartphones, particularly iPhones with the "notch" (e.g., iPhone X and newer), the visible area for your application is often constrained. If you simply use a standard View for your root component, your content may look fine on an older Android device, but it could be hidden or cut off on a newer iPhone:

- **Status Bar:** Content can get hidden underneath the transparent status bar at the top.
- **The Notch:** The camera and speaker housing (the notch) cuts into the available display area.
- **Home Indicator:** The horizontal line at the bottom of the screen (on devices without a physical home button) can obscure bottom-aligned content.

The SafeAreaView automatically applies padding to respect these areas, ensuring your content is always viewable.

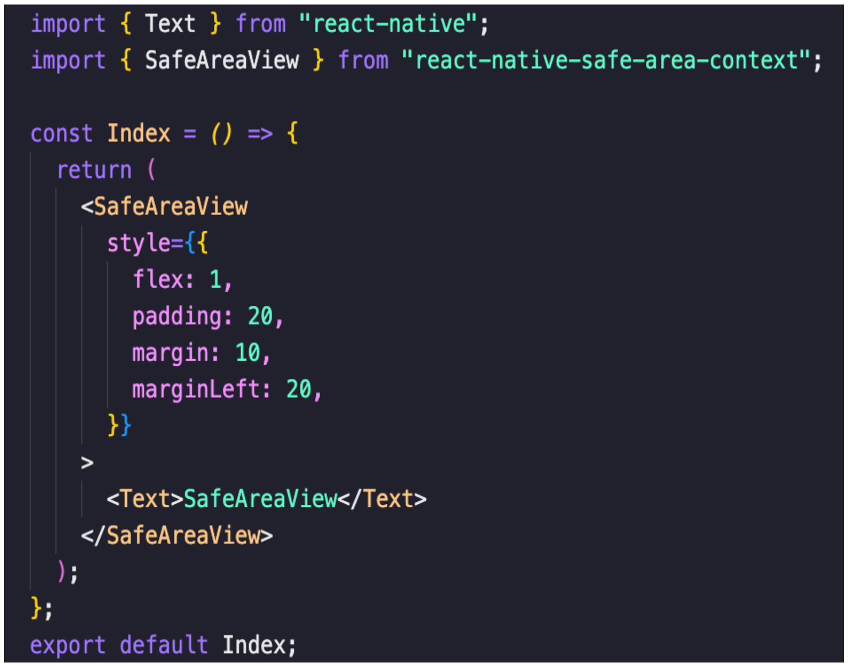
How to Use It in Expo

Since Expo uses standard React Native components, you import SafeAreaView directly from react-native. You should wrap your entire screen's content with it.

Install

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

Basic Implementation



```
import { Text } from "react-native";
import { SafeAreaView } from "react-native-safe-area-context";

const Index = () => {
  return (
    <SafeAreaView
      style={{
        flex: 1,
        padding: 20,
        margin: 10,
        marginLeft: 20,
      }}
    >
      <Text>SafeAreaView</Text>
    </SafeAreaView>
  );
};
export default Index;
```

⚠️ Important Considerations

1. **Android:** On Android, SafeAreaView generally renders as a normal View and doesn't apply special padding because Android handles the status bar differently. If you need to handle the Android status bar, you'll often use the **StatusBar** component or the **Constants.statusBarHeight** value from the expo-constants package, though this is less common for basic layout.
2. **Layout:** You must apply the style **flex: 1** to the SafeAreaView so it expands to fill the entire screen, allowing it to correctly calculate and apply the necessary insets.
3. **Cross-Platform Solution:** For more complex safe area needs or accessing the safe area values programmatically (like how much padding was applied), the **useSafeAreaInsets** hook from the community library **react-native-safe-area-context** is the recommended modern solution that works reliably across both iOS and Android.

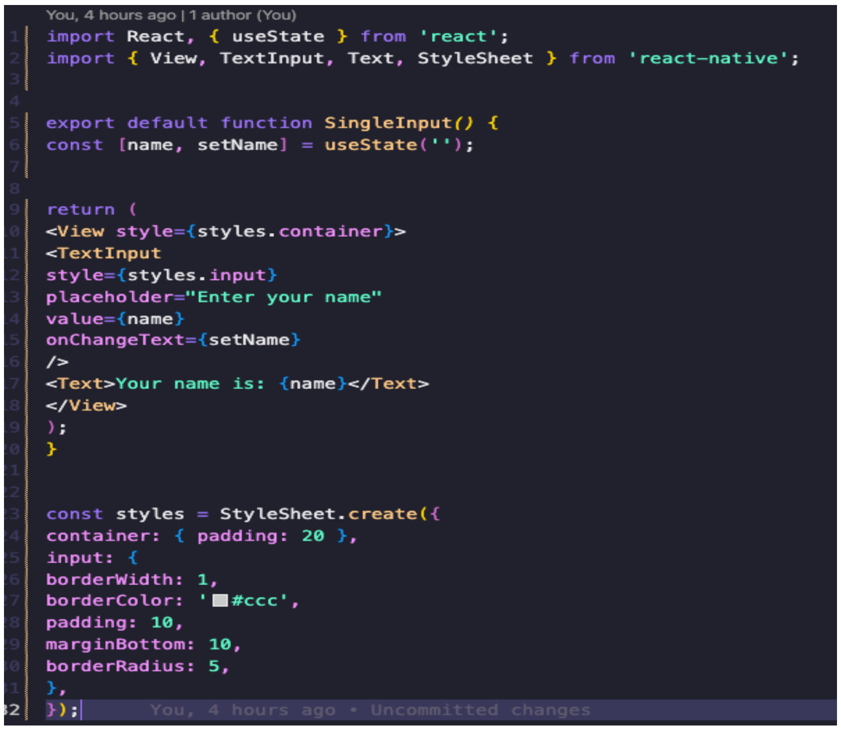
Chapter Five: Form Input and Checkbox in React Native

Definition: Form input in React Native refers to the components used to capture user data within an application. The primary component for form input is TextInput, which allows users to enter text, numbers, or passwords.

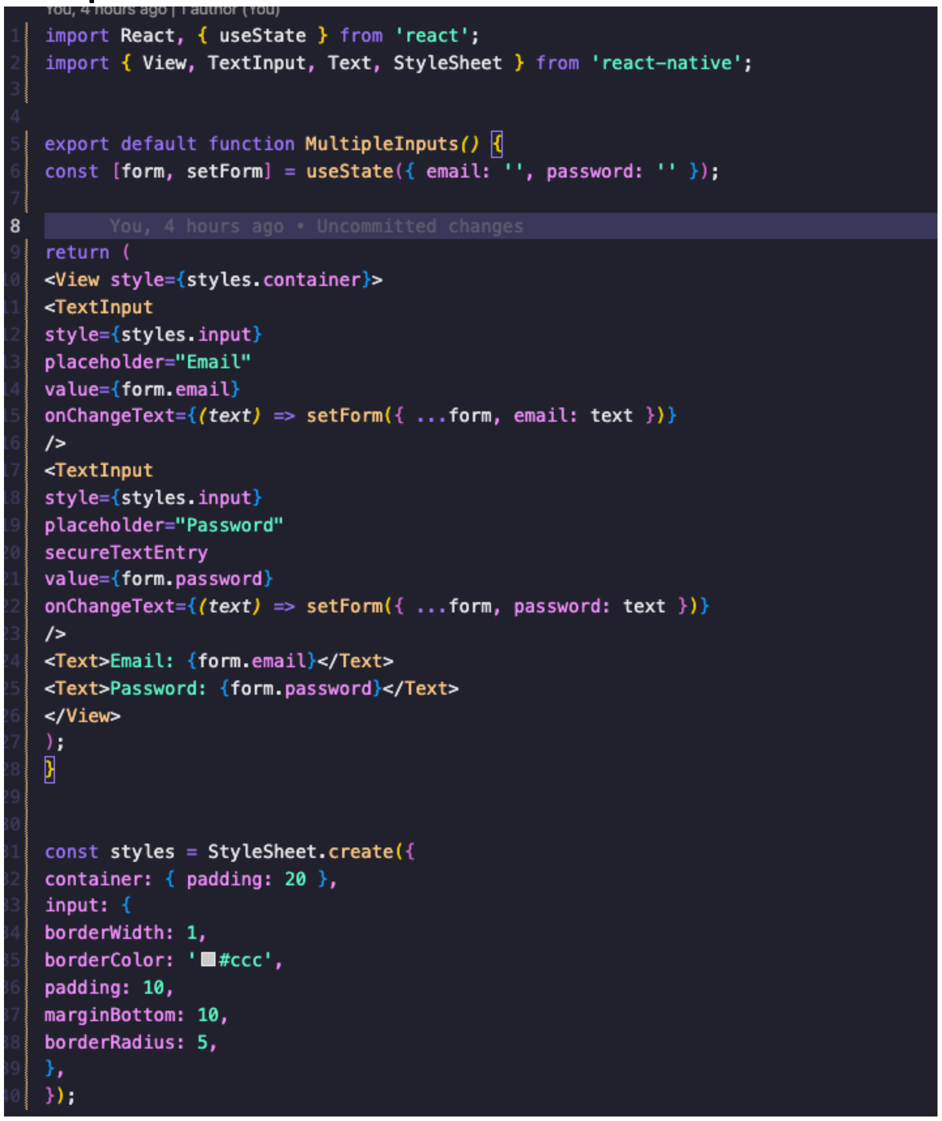
Key Points:

1. TextInput is the core component for capturing user input.
2. State management is essential to handle the value entered by the user.
3. Multiple inputs can be managed using separate state variables or a single state object.
4. KeyboardAvoidingView is used to ensure inputs are not hidden by the keyboard.
5. Styling can be customized using standard React Native styles.

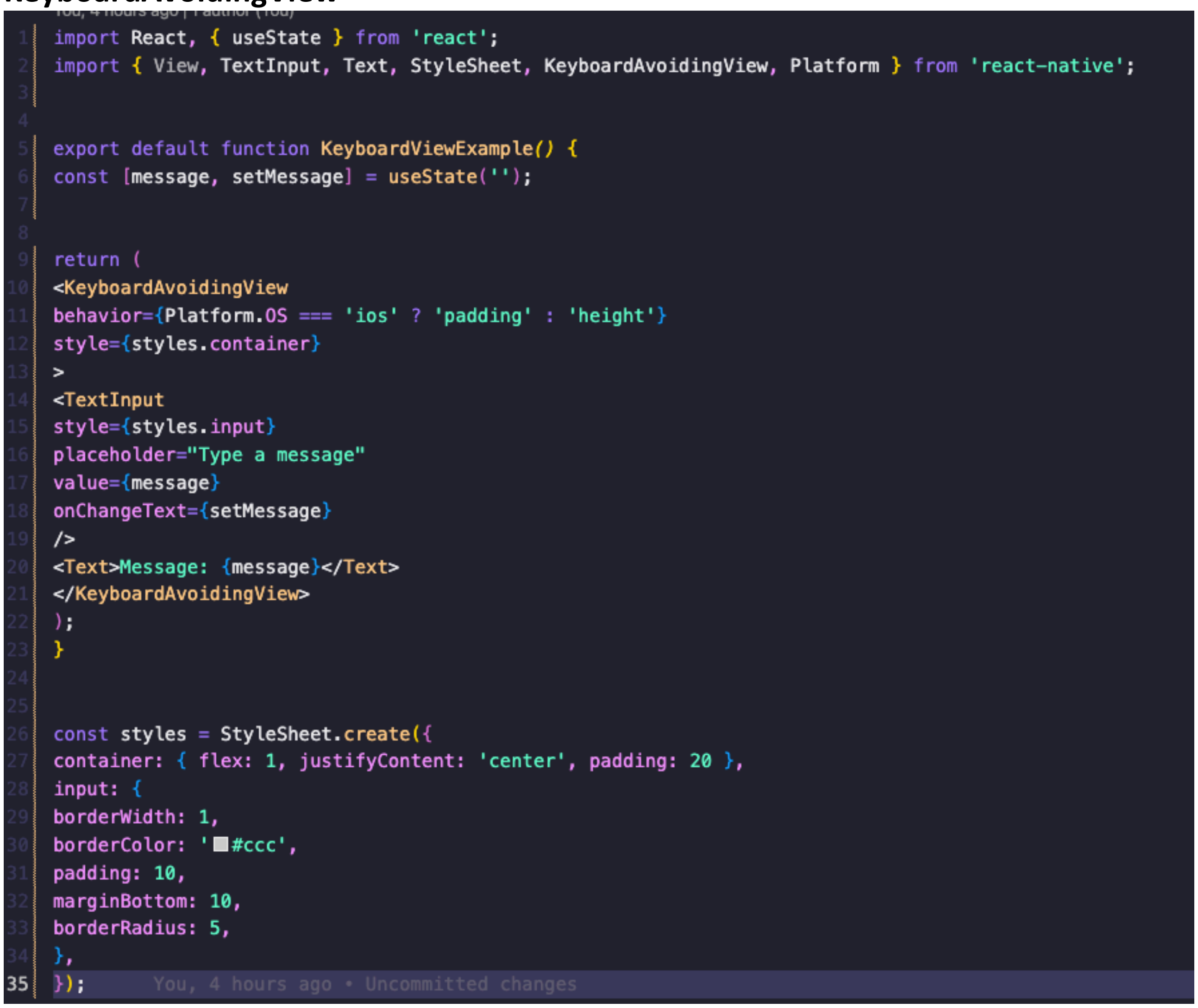
One Input


```
You, 4 hours ago | 1 author (You)
import React, { useState } from 'react';
import { View, TextInput, Text, StyleSheet } from 'react-native';


export default function SingleInput() {
const [name, setName] = useState('');


return (
<View style={styles.container}>
<TextInput
style={styles.input}
placeholder="Enter your name"
value={name}
onChangeText={setName}
/>
<Text>Your name is: {name}</Text>
</View>
);
}


const styles = StyleSheet.create({
container: { padding: 20 },
input: {
borderWidth: 1,
borderColor: '#ccc',
padding: 10,
marginBottom: 10,
borderRadius: 5,
},
});        You, 4 hours ago • Uncommitted changes
```

Multiple


```
You, 4 hours ago | 1 author (You)
import React, { useState } from 'react';
import { View, TextInput, Text, StyleSheet } from 'react-native';


export default function MultipleInputs() {
const [form, setForm] = useState({ email: '', password: '' });

        You, 4 hours ago • Uncommitted changes
return (
<View style={styles.container}>
<TextInput
style={styles.input}
placeholder="Email"
value={form.email}
onChangeText={(text) => setForm({ ...form, email: text })}
/>
<TextInput
style={styles.input}
placeholder="Password"
secureTextEntry
value={form.password}
onChangeText={(text) => setForm({ ...form, password: text })}
/>
<Text>Email: {form.email}</Text>
<Text>Password: {form.password}</Text>
</View>
);
}


const styles = StyleSheet.create({
container: { padding: 20 },
input: {
borderWidth: 1,
borderColor: '#ccc',
padding: 10,
marginBottom: 10,
borderRadius: 5,
},
});
```

KeyboardAvoidingView



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


export default function KeyboardViewExample() {
const [message, setMessage] = useState('');


return (
<KeyboardAvoidingView
behavior={Platform.OS === 'ios' ? 'padding' : 'height'}
style={styles.container}
>
<TextInput
style={styles.input}
placeholder="Type a message"
value={message}
onChangeText={setMessage}
/>
<Text>Message: {message}</Text>
</KeyboardAvoidingView>
);
}


const styles = StyleSheet.create({
container: { flex: 1, justifyContent: 'center', padding: 20 },
input: {
borderWidth: 1,
borderColor: '#ccc',
padding: 10,
marginBottom: 10,
borderRadius: 5,
},
});      You, 4 hours ago • Uncommitted changes
```

Checkbox

Definition

A **Checkbox** in Expo (React Native) is a UI component that allows the user to select or unselect an option, usually represented by a square box with a checkmark.
React Native does **not include a built-in checkbox**, so Expo apps typically use one of:

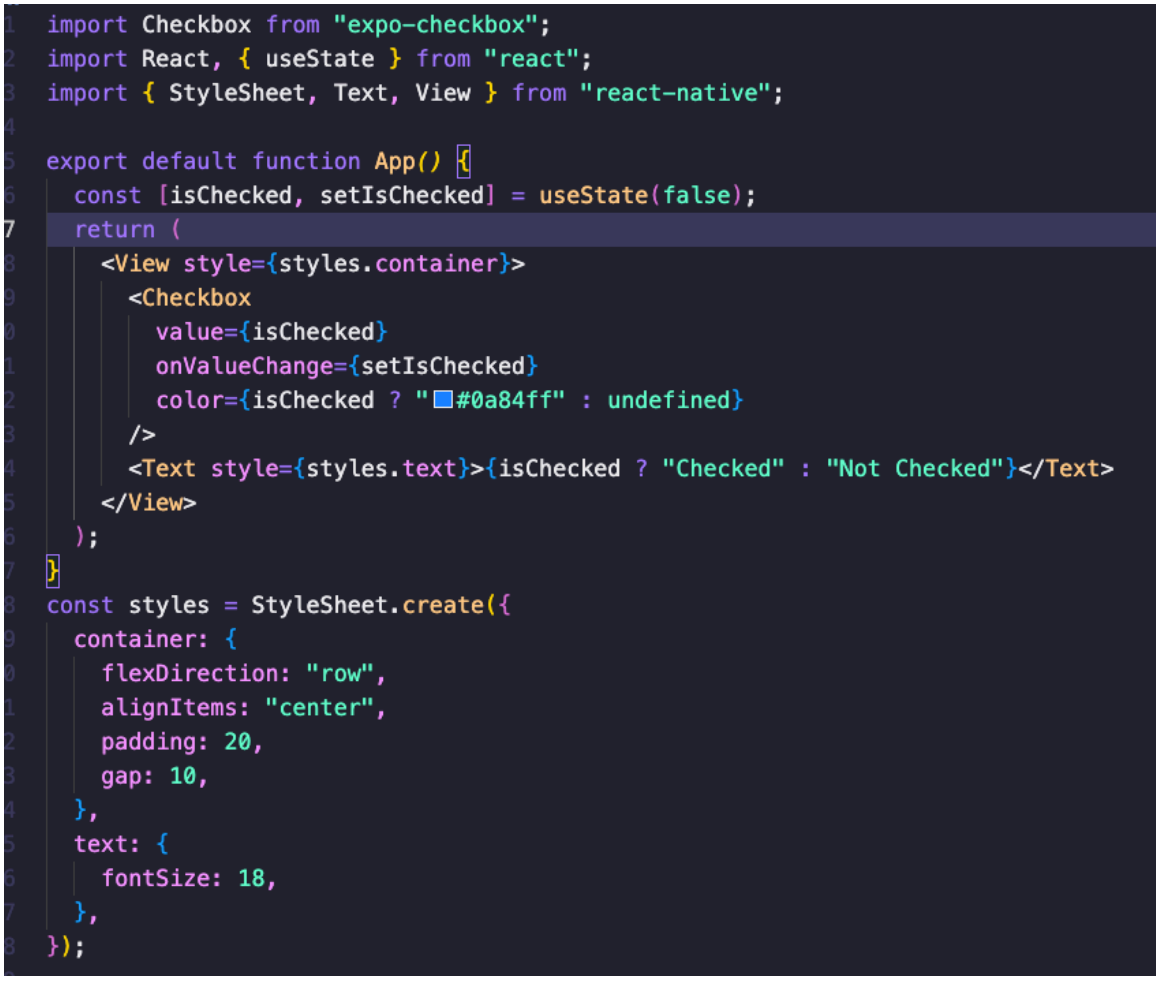
Key Points

- Expo does not include a built-in checkbox; use expo-checkbox or another library.

- Checkboxes work well for todos, preferences, forms, and settings pages.
- They use a **boolean state** (true/false).
- Lightweight and easy to integrate.
- Works on both **iOS and Android**.
- Supports styling (color, size, border, etc.).

Install

expo install expo-checkbox



```
import Checkbox from "expo-checkbox";
import React, { useState } from "react";
import { StyleSheet, Text, View } from "react-native";

export default function App() {
  const [isChecked, setIsChecked] = useState(false);
  return (
    <View style={styles.container}>
      <Checkbox
        value={isChecked}
        onValueChange={setIsChecked}
        color={isChecked ? "#0a84ff" : undefined}
      />
      <Text style={styles.text}>{isChecked ? "Checked" : "Not Checked"}</Text>
    </View>
  );
}
const styles = StyleSheet.create({
  container: {
    flexDirection: "row",
    alignItems: "center",
    padding: 20,
    gap: 10,
  },
  text: {
    fontSize: 18,
  },
});
```

Chapter Six: AsyncStorage in React Native

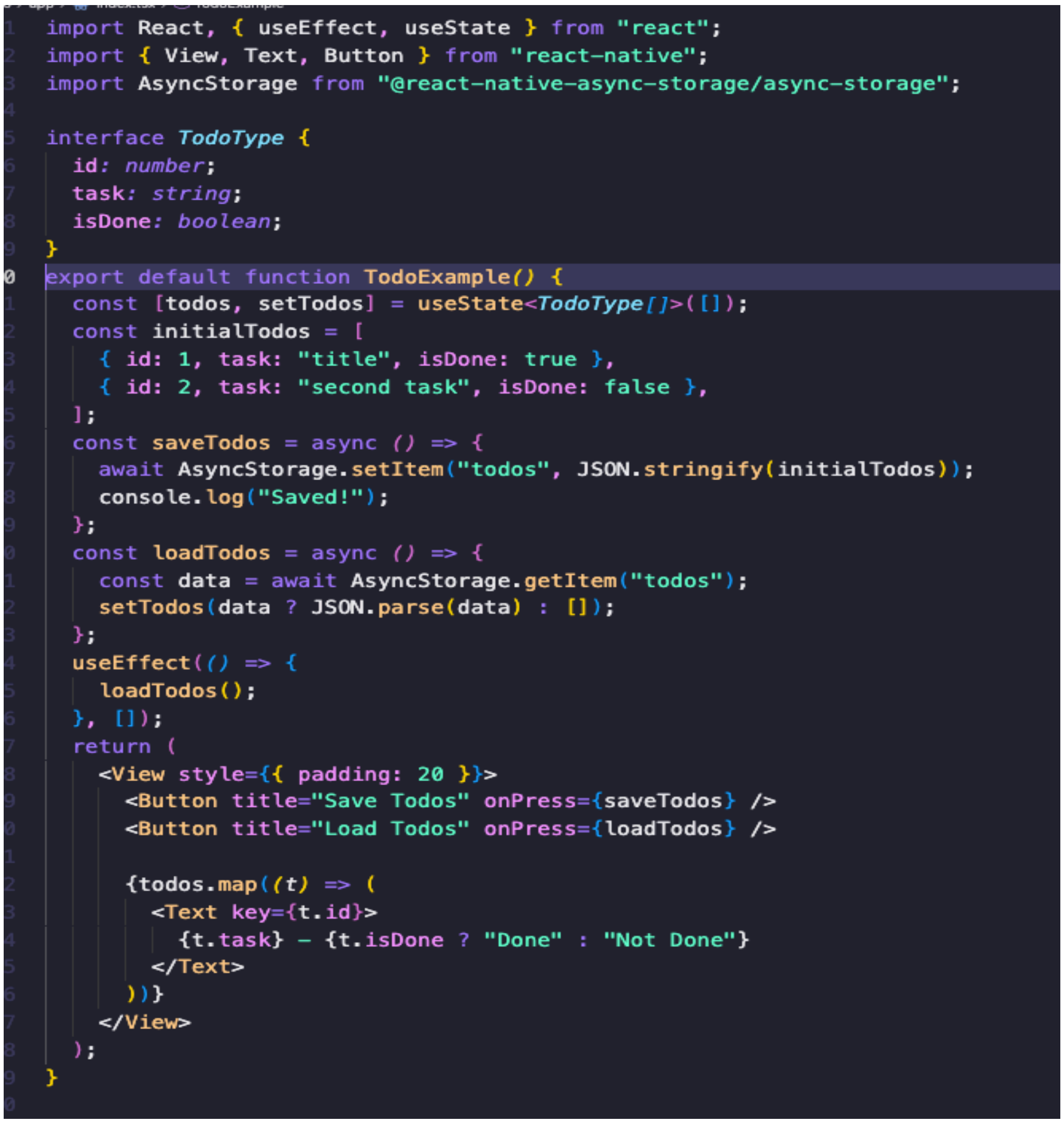
Definition

AsyncStorage is a simple, unencrypted, key-value storage system used in React Native for saving small amounts of data **persistently** on the device.

It works like a local database for storing user preferences, login tokens, or small datasets (like todos).

Key Points

- Stores data **persistently** (survives app restarts).
- Works as **key–value** storage.
- Supports only **string values**, so objects must be saved using JSON.stringify.
- Retrieval requires JSON.parse.
- Great for small data; not suitable for large databases.
- Promises-based API (async/await friendly).



```
import React, { useEffect, useState } from "react";
import { View, Text, Button } from "react-native";
import AsyncStorage from "@react-native-async-storage/async-storage";

interface TodoType {
  id: number;
  task: string;
  isDone: boolean;
}
export default function TodoExample() {
  const [todos, setTodos] = useState<TodoType[]>([]);
  const initialTodos = [
    { id: 1, task: "title", isDone: true },
    { id: 2, task: "second task", isDone: false },
  ];
  const saveTodos = async () => {
    await AsyncStorage.setItem("todos", JSON.stringify(initialTodos));
    console.log("Saved!");
  };
  const loadTodos = async () => {
    const data = await AsyncStorage.getItem("todos");
    setTodos(data ? JSON.parse(data) : []);
  };
  useEffect(() => {
    loadTodos();
  }, []);
  return (
    <View style={{ padding: 20 }}>
      <Button title="Save Todos" onPress={saveTodos} />
      <Button title="Load Todos" onPress={loadTodos} />

      {todos.map((t) => (
        <Text key={t.id}>
          {t.task} - {t.isDone ? "Done" : "Not Done"}
        </Text>
      ))}
    </View>
  );
}
```