React Native: Developing Android and iOS Apps

Cheat Sheet: React Native Using Expo for Styling and State Management with Props

Expo Commands	Description	Command Example

Inline Styles	Inline style in React Native refers to applying styles directly to a component using the style prop, allowing for quick and specific styling without creating a separate stylesheet.	`<View>`
		`      <Text style={{ color: 'blue', fontSize: 20 }}>Hello, World!</Text>`
		`    </View>`

StyleSheet.create()	A stylesheet in React Native is an object created using StyleSheet.create() that defines styles for components, promoting better performance and organization by separating styling from the component logic.	`import React from 'react';`
		`import { View, Text, StyleSheet } from 'react-native';`
		
		`const App = () => {`
		`  return (`
		`    <View style={styles.container}>`
		`      <Text style={styles.text}>Hello, World!</Text>`
		`    </View>`
		`  );`
		`};`
		
		`const styles = StyleSheet.create({`
		`  container: {`
		`    flex: 1,`
		`    justifyContent: 'center',`
		`    alignItems: 'center',`
		`    backgroundColor: '#f0f0f0',`
		`  },`
		`  text: {`
		`    color: 'blue',`
		`    fontSize: 20,`
		`  },`
		`});`

		`export default App;`

Combined Styles	Combined styles in React Native refer to the practice of merging multiple style objects into a single style prop using an array, allowing for the application of multiple styles to a component for greater flexibility and reusability.	`import { View, Text, StyleSheet } from 'react-native';`
		
		`const App = () => {`
		`  return (`
		`    <View style={styles.container}>`
		`      <Text style={[styles.text, styles.boldText]}>Hello,`
		`World!</Text>`
		`    </View>`
		`  );`
		`};`
		
		`const styles = StyleSheet.create({`
		`  container: {`
		`    flex: 1,`
		`    justifyContent: 'center',`
		`    alignItems: 'center',`
		`  },`
		`  text: {`
		`    fontSize: 20,`
		`    color: 'blue',`
		`  },`
		`  boldText: {`
		`    fontWeight: 'bold',`
		`  },`
		`});`
		
		`export default App;`

Platform-Specific Styles	Platform specific styling can also be created via the Platform module or by creating stylesheets for each platform (iPhone and Android).	`import { View, Text, StyleSheet, Platform } from 'react-native';`
		
		`const App = () => {`
		`  return (`

		`    <View style={styles.container}>`
		`      <Text style={styles.text}>Hello, World!</Text>`
		`    </View>`
		`  );`
		`};`
		
		`const styles = StyleSheet.create({`
		`  container: {`
		`    flex: 1,`
		`    justifyContent: 'center',`
		`    alignItems: 'center',`
		`    backgroundColor: '#f0f0f0',`
		`  },`
		`  text: {`
		`    ...Platform.select({`
		`      ios: {`
		`        color: 'blue',`
		`        fontSize: 20,`
		`      },`
		`      android: {`
		`        color: 'green',`
		`        fontSize: 18,`
		`      },`
		`    }),`
		`  },`
		`});`
		
		`export default App;`

Global Styles	A style might reflect a global style if styles are defined in a seperate file which is then imported when needed.	`// styles.js`
		`import { StyleSheet } from 'react-native';`
		
		`export const globalStyles = StyleSheet.create({`
		`  container: {`
		`    flex: 1,`

```
    justifyContent: 'center',
    alignItems: 'center',
    backgroundColor: '#f0f0f0',
  },
  text: {
    color: 'blue',
    fontSize: 20,
  },
});

// App.js
import React from 'react';
import { View, Text } from 'react-native';
import { globalStyles } from './styles';

const App = () => {
  return (
    <View style={globalStyles.container}>
      <Text style={globalStyles.text}>Hello, World!</Text>
    </View>
  );
};

export default App;
```

Themed Styles

Themed styles in React Native refer to applying consistent styling based on a defined theme (like light or dark mode) using context or libraries, allowing for a cohesive look and feel across the app that can easily adapt to user preferences.

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

const App = () => {
  const theme = useColorScheme(); // 'light' or 'dark'

  return (
    <View style={[styles.container, theme === 'dark' &&
styles.darkContainer]}>
```

		`      <Text style={[styles.text, theme === 'dark' &&`
		`styles.darkText]}>`
		`        Hello, World!`
		`      </Text>`
		`    </View>`
		`  );`
		`};`
		
		`const styles = StyleSheet.create({`
		`  container: {`
		`    flex: 1,`
		`    justifyContent: 'center',`
		`    alignItems: 'center',`
		`    backgroundColor: '#f0f0f0',`
		`  },`
		`  darkContainer: {`
		`    backgroundColor: '#333',`
		`  },`
		`  text: {`
		`    color: 'blue',`
		`    fontSize: 20,`
		`  },`
		`  darkText: {`
		`    color: 'white',`
		`  },`
		`});`
		
		`export default App;`

Layout with Flexbox	A way of creating a simple layout that places three boxes side by side, aligned at the center of the container.	`import React from 'react';`
		`import { View, StyleSheet } from 'react-native';`
		
		`const FlexboxExample = () => {`
		`  return (`
		`    <View style={styles.container}>`

		`      <View style={styles.box} />` `      <View style={styles.box} />` `      <View style={styles.box} />` `    </View>` `  );` `};` ` ` `const styles = StyleSheet.create({` `  container: {` `    flex: 1,` `    flexDirection: 'row',` `    justifyContent: 'center',` `    alignItems: 'center',` `    backgroundColor: '#f5f5f5',` `  },` `  box: {` `    width: 50,` `    height: 50,` `    backgroundColor: 'skyblue',` `    margin: 5,` `  },` `});` ` ` `export default FlexboxExample;`
flex Property for Dynamic Layouts	The flex property is versatile when it comes to usage and results in the profuse creation of flexible layouts. It lets the apparatus widen or contract enabling stellar objects to have sizes proportional to the space available, sized by flex value.	`import { View, StyleSheet } from 'react-native';` ` ` `const FlexPropertyExample = () => {` `  return (` `    <View style={styles.container}>` `      <View style={[styles.box, { flex: 1 }]} />` `      <View style={[styles.box, { flex: 2 }]} />` `      <View style={[styles.box, { flex: 3 }]} />` `    </View>` `  );`

		`};`
		`const styles = StyleSheet.create({`
		`  container: {`
		`    flex: 1,`
		`    flexDirection: 'row',`
		`  },`
		`  box: {`
		`    height: 100,`
		`    backgroundColor: 'lightseagreen',`
		`    margin: 5,`
		`  },`
		`});`
		`export default FlexPropertyExample;`
Styled Component Package	The command npm i styled-components installs the styled-components library, enabling the use of styled components in a React Native project for modular and dynamic styling.	`npm i styled-components`
Styled Components	Styled components in React Native are a library that enables developers to create component-level styles using tagged template literals, allowing for writing CSS directly in JavaScript and promoting a more modular and maintainable styling approach.	`import React from 'react';`
		`import styled from 'styled-components/native';`
		`const StyledComponent = () => {`
		`  return (`
		`    <Container>`
		`      <Title>Hello, Styled Components!</Title>`
		`      <StyledButton onPress={() => alert('Button Pressed!')}>`
		`        <ButtonText>Press Me</ButtonText>`
		`      </StyledButton>`
		`    </Container>`
		`  );`
		`};`

		export default StyledComponent;
		
		// Styled components
		const Container = styled.View\`
		  flex: 1;
		  justify-content: center;
		  align-items: center;
		  background-color: #20A271;
		\`;
		
		const Title = styled.Text\`
		  font-size: 24px;
		  color: #333;
		  margin-bottom: 20px;
		\`;
		
		const StyledButton = styled.TouchableOpacity\`
		  background-color: #6200ea;
		  padding: 10px 20px;
		  border-radius: 5px;
		\`;
		
		const ButtonText = styled.Text\`
		  color: #fff;
		  font-size: 16px;
		\`;

useState Hook	Initializes state variable using useState() hook with default value 0.	const [state, setState] = useState(0);

useState For Toggle Button	This code explains how the useState hook can be used to toggle the button to perform operations.	import React, { useState } from 'react';
		import { View, Text, Button, StyleSheet } from 'react-native';
		
		function ThemeToggle() {

		`const [isDarkMode, setIsDarkMode] = useState(false);`
		
		`const toggleTheme = () => {`
		`  setIsDarkMode(previousMode => !previousMode);`
		`};`
		
		`return (`
		`  <View style={[styles.container, { backgroundColor: isDarkMode ? '#333' : '#FFF' }]}>`
		`    <Text style={[styles.text, { color: isDarkMode ? '#FFF' : '#000' }]}>`
		`      {isDarkMode ? 'Dark Mode' : 'Light Mode'}`
		`    </Text>`
		`    <Button title="Toggle Theme" onPress={toggleTheme} />`
		`  </View>`
		`);`
		`}`
		
		`const styles = StyleSheet.create({`
		`  container: {`
		`    flex: 1,`
		`    justifyContent: 'center',`
		`    alignItems: 'center',`
		`  },`
		`  text: {`
		`    fontSize: 24,`
		`    marginBottom: 20,`
		`  },`
		`});`
		
		`export default ThemeToggle;`

useReducer	useReducer is a hook that is used in cases when the action performed will affect several state values and when	`const [state, dispatch] = useReducer(reducer, initialState);`

	depending on the action, different pieces of state have to be modified.	

useReducer Effect	This code defines a UserForm component in React Native that uses the useReducer hook to manage the state of a form with fields for name, email, and phone, allowing users to update their input and submit or reset the form.	
		```
		import React, { useReducer } from 'react';
		import { View, TextInput, Button, Text, StyleSheet } from 'react-native';
		const initialState = {
		  name: '',
		  email: '',
		  phone: ''
		};
		function reducer(state, action) {
		  switch (action.type) {
		   case 'updateName':
		      return { ...state, name: action.payload };
		    case 'updateEmail':
		      return { ...state, email: action.payload };
		    case 'updatePhone':
		      return { ...state, phone: action.payload };
		    case 'reset':
		      return initialState;
		    default:
		      throw new Error('Unknown action type');
		  }
		}

		function UserForm() {
		  const [state, dispatch] = useReducer(reducer, initialState);

		  const handleSubmit = () => {
		    console.log('Form Data:', state);
		  };

		  return (
		    <View style={styles.container}>
		```

```
      <TextInput
        style={styles.input}
        placeholder="Name"
        value={state.name}
        onChangeText={text => dispatch({ type: 'updateName', payload: text })}
      />
      <TextInput
        style={styles.input}
        placeholder="Email"
        value={state.email}
        onChangeText={text => dispatch({ type: 'updateEmail', payload: text })}
      />
      <TextInput
        style={styles.input}
        placeholder="Phone"
        value={state.phone}
        onChangeText={text => dispatch({ type: 'updatePhone', payload: text })}
      />
      <Button title="Submit" onPress={handleSubmit} />
      <Button title="Reset" onPress={() => dispatch({ type: 'reset' })} />
    </View>
  );
}

const styles = StyleSheet.create({
  container: {
    padding: 20,
  },
  input: {
    height: 40,
    borderColor: 'gray',
```

		`    borderWidth: 1,`
		`    marginBottom: 10,`
		`    paddingHorizontal: 10,`
		`  },`
		`});`
		
		`export default UserForm;`

Props	Props are properties in React native use to pass data from one component to another and this is usually done from the parent component to the child component.	`import React from 'react';`
		`import { View, StyleSheet } from 'react-native';`
		`import EmployeeDetail from './EmployeeDetail';`
		
		`function App() {`
		`  return (`
		`    <View style={styles.container}>`
		`      <EmployeeDetail name="John Doe" dept="Finance"/>`
		`    </View>`
		`  );`
		`}`
		`const styles = StyleSheet.create({`
		`  container: {`
		`    flex: 1,`
		`    justifyContent: 'center',`
		`    alignItems: 'center',`
		`    backgroundColor: '#fff',`
		`  },`
		`});`
		
		`export default App;`

Default Props	Default props are used to set a value to props just in case they are not passed by the parent component.	`const EmployeeDetail = (props) => {`
		`  return (`
		`    <View>`
		`      <Text>Hello, {props.name}!</Text>`
		`    </View>`

		`  );`
		`}`
		
		`// Set default props`
		`EmployeeDetail.defaultProps = {`
		`  name: “Default Name”`
		`}`

Context API	The Context API in React provides a way to share state and functionality across the component tree without having to pass props down manually at every level, facilitating global state management.	`import React, { createContext, useState } from 'react';`
		`const LanguageContext = createContext();`

Context API Example Part-1	Create a file called LanguageProvider.js where you'll specify the context and the provider.	`import React, { createContext, useState } from 'react';`
		`// Create the LanguageContext`
		`export const LanguageContext = createContext();`
		
		`// Create the LanguageProvider component`
		`export function LanguageProvider({ children }) {`
		`  const [language, setLanguage] = useState('en');`
		
		`  const changeLanguage = (newLanguage) => {`
		`    setLanguage(newLanguage);`
		`  };`
		
		`  return (`
		`    <LanguageContext.Provider value={{ language, changeLanguage }}>`
		`      {children}`
		`    </LanguageContext.Provider>`
		`  );`
		`}`

Context API Example Part-2	Incorporate LanguageContext in Child Component.	`import React, { useContext } from 'react';`
		`import { View, Text, Button, StyleSheet } from 'react-native';`

```
import { LanguageContext } from './LanguageProvider'; // Import the context

function LanguageSwitcher() {
  const { language, changeLanguage } = useContext(LanguageContext); // Access context values

  const displayText = language === 'fr' ? 'Langue Actuelle: Français' : 'Current Language: English'; // Static conversion

  return (
    <View style={styles.container}>
      <Text style={styles.text}>{displayText}</Text> {/* Use static conversion text */}
      <Button title="Switch to French" onPress={() => changeLanguage('fr')} />
      <Button title="Switch to English" onPress={() => changeLanguage('en')} />
    </View>
  );
}

const styles = StyleSheet.create({
  container: {
    flex: 1,
    justifyContent: 'center',
    alignItems: 'center',
    backgroundColor: '#fff',
  },
  text: {
    fontSize: 20,
    marginBottom: 20,
  },
});
```

		`export default LanguageSwitcher;`

Context API Example Part-3	Wrap App Component with LanguageProvider.	
		```
		import React from 'react';
		import { LanguageProvider } from
		'./components/ContentVideo/LanguageProvider'; // Import the provider
		import LanguageSwitcher from
		'./components/ContentVideo/LanguageSwitcher'; // Import the
		LanguageSwitcher component

		function App() {
		  return (
		    <LanguageProvider>
		      <LanguageSwitcher />
		    </LanguageProvider>
		  );
		}

		export default App;
		```

Redux Toolkit	The command npm i @reduxjs/toolkit installs the Redux Toolkit library, which simplifies the process of setting up and managing state in Redux applications by providing a set of tools and best practices.	`npm i @reduxjs/toolkit`
createSlice	The createSlice function in Redux Toolkit simplifies the process of creating a slice of the Redux state, automatically generating action creators and reducers based on the provided reducers and initial state.	`import { createSlice } from '@reduxjs/toolkit';`

Redux Toolkit Basic Layout	A reducer is a function that determines how the state of an application changes in response to actions, while an action is	
		```
		import { createSlice } from '@reduxjs/toolkit';

		export const counterSlice = createSlice({
		```

	a plain object that describes an event or user interaction that triggers a state update.	`  name: 'counter',`
		`  initialState: { count: 0 },`
		`  reducers: {`
		`    increment: (state) => {`
		`      state.count += 1;`
		`    },`
		`    decrement: (state) => {`
		`      state.count -= 1;`
		`    },`
		`  },`
		`});`
		
		`export const { increment, decrement } = counterSlice.actions;`
		
		`export default counterSlice.reducer;`

configureStore	The configureStore function in Redux Toolkit simplifies the setup of a Redux store by providing a default configuration and allowing for easy integration of middleware, enhancers, and slice reducers.	`import { configureStore } from '@reduxjs/toolkit';`
useSelector	The useSelector hook in React Redux allows components to extract data from the Redux store's state, enabling them to subscribe to specific parts of the state.	`const count = useSelector(state => state.counter.count);`
useDispatch	The useDispatch hook in React Redux provides a way to access the store's dispatch function, allowing components to send actions to the Redux store.	`const dispatch = useDispatch();`
Code Example of Redux Part-1	The code is in counterSlice.js component that creates a Redux slice	`import { createSlice } from '@reduxjs/toolkit';`

	named counter using Redux Toolkit, initializing the state with a count value of 0 and defining two reducers (increment and decrement) to update the count state when the corresponding actions are dispatched.	`export const counterSlice = createSlice({`
		`  name: 'counter',`
		`  initialState: { count: 0 },`
		`  reducers: {`
		`    increment: (state) => {`
		`      state.count += 1;`
		`    },`
		`    decrement: (state) => {`
		`      state.count -= 1;`
		`    },`
		`  },`
		`});`
		
		`export const { increment, decrement } = counterSlice.actions;`
		
		`export default counterSlice.reducer;`

Code Example of Redux Part-2	The code is in store.js file that configures a Redux store using Redux Toolkit, integrating the counterReducer to manage the counter slice of the state, making it available for the application's state management.	`import { configureStore } from '@reduxjs/toolkit';`
		`import counterReducer from './components/ReduxToolKit/counterSlice';`
		`export const store = configureStore({`
		`  reducer: {`
		`    counter: counterReducer,`
		`  },`
		`});`

Code Example of Redux Part-3	The above is in App.js component that defines a React Native app that uses Redux for state management, displaying a Counter component that shows the current count and provides buttons to increment or decrement the count, with the Redux store provided at the app's root level.	`import React from 'react';`
		`import { Provider, useDispatch, useSelector } from 'react-redux';`
		`import { View, Text, Button, StyleSheet } from 'react-native';`
		`import { store } from './store';`
		`import { increment, decrement } from`
		`'./components/ReduxToolKit/counterSlice';`
		
		`function Counter() {`
		`  const count = useSelector((state) => state.counter.count);`
		`  const dispatch = useDispatch();`

```
  return (
    <View style={styles.container}>
      <Text style={styles.text}>Count: {count}</Text>
      <Button title="Increment" onPress={() => dispatch(increment())}
/>
      <Button title="Decrement" onPress={() => dispatch(decrement())}
/>
    </View>
  );
}

export default function App() {
  return (
    <Provider store={store}>
      <Counter />
    </Provider>
  );
}
```