Lecture 03

React Native (part 2)

Contents

- Fast Refresh
- Error Resilience
- Debugging
- Building Adaptive User Interfaces
- Handling user input
- Core Components:
  - ScrollView, FlatList, Pressable, Image, Modal

Theory: What is Fast Refresh?

- **Fast Refresh** is a feature in React that updates the application instantly after editing the source code without reloading the entire page.
- **Key benefits:**
  - Preserves state in **function components** and **Hooks**.
  - Speeds up development and reduces interruptions.

Error Resilience

- **Syntax Errors:**
  - When a syntax error occurs, React prevents the module from running and shows a red box.
  - After fixing the error and saving the file, the red box disappears without needing to reload the app.
- **Runtime Errors:**
  - **During module initialization:** Fixing the error resumes the Fast Refresh session.
  - **Inside components:** After fixing the error, React remounts the component with the updated code.

Error Resilience

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

// Syntax error: Missing closing tag
const App = () => {
  return (
    <View>
      <Text>Hello, React Native!  // Syntax error
    </View>
  );
};

// Runtime error: Using an incorrect API
const App = () => {
  const styles = Style.create({}); // Error: Should be StyleSheet.create
  return (
    <View style={styles.container}>
      <Text>Hello, React Native!</Text>
    </View>
  );
};
```

Limitations

- **State is not preserved in:**
  - Class components.
  - Modules with multiple exports besides React components.
  - Higher-order components returning class components.

Limitations

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

// State is not preserved in class components
class App extends Component {
  state = { count: 0 };

  increment = () => this.setState({ count: this.state.count + 1 });

  render() {
    return (
      <View>
        <Text>{this.state.count}</Text>
        <Button title="Increment" onPress={this.increment} />
      </View>
    );
  }
}
```

Fast Refresh and Hooks

- **useState** and **useRef**: Preserve their values if the arguments are unchanged.
- **useEffect**, **useMemo**, and **useCallback**: Always update during Fast Refresh, even if dependencies don't change.

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

const App = ({ multiplier }) => {
  const [count, setCount] = useState(0);

  // useMemo will re-run when edited
  const double = useMemo(() => count * multiplier, [count]);

  useEffect(() => {
    console.log('Component re-rendered');
  }, []);

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

Debugging React Native App

- Log things to the Console by adding `console.log()` statement

```
38          userNumber={userNumber}
39          roundsNumber={guessRounds}
40          onStartNewGame={startNewGameHandler}
41        />
42      )
43    }
44
45    console.log('this is message')      You, 4 minutes ago • Uncommitted ch
46
47    return (
48      <LinearGradient
49        colors={[Colors.primary700, Colors.accent500]}
50        style={styles.rootScreen}
51      >
52        <ImageBackground
53          source={require('./assets/images/background.png')}
54          resizeMode='cover'
55          style={styles.rootScreen}
56          imageStyle={styles.backgroundImage}
```

```
PROBLEMS   OUTPUT   DEBUG CONSOLE   PORTS   TERMINAL   GITLENS   SQL CONSOLE

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

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

› Press a | open Android
› Press w | open web

› Press j | open debugger
› Press r | reload app
› Press m | toggle menu
› Press o | open project code in your editor

› Press ? | show all commands

› Reloading apps
Android Bundled 43ms (C:\Users\ADMIN\VSCode\workspace\react-nat
 LOG  this is message
```

Logging to the Console

`console.log()` is a powerful tool for developers offering several advantages:

- **Print variables and expressions** at any point in your code
  - Identify errors
  - Track the execution flow
- **Inspect the state of your program** by inserting `console.log()` statements throughout your code.
  - This makes debugging a controlled and efficient process.
- Make your code more self-documenting by strategically placing `console.log()` statements.

Debugging JavaScript Remotely

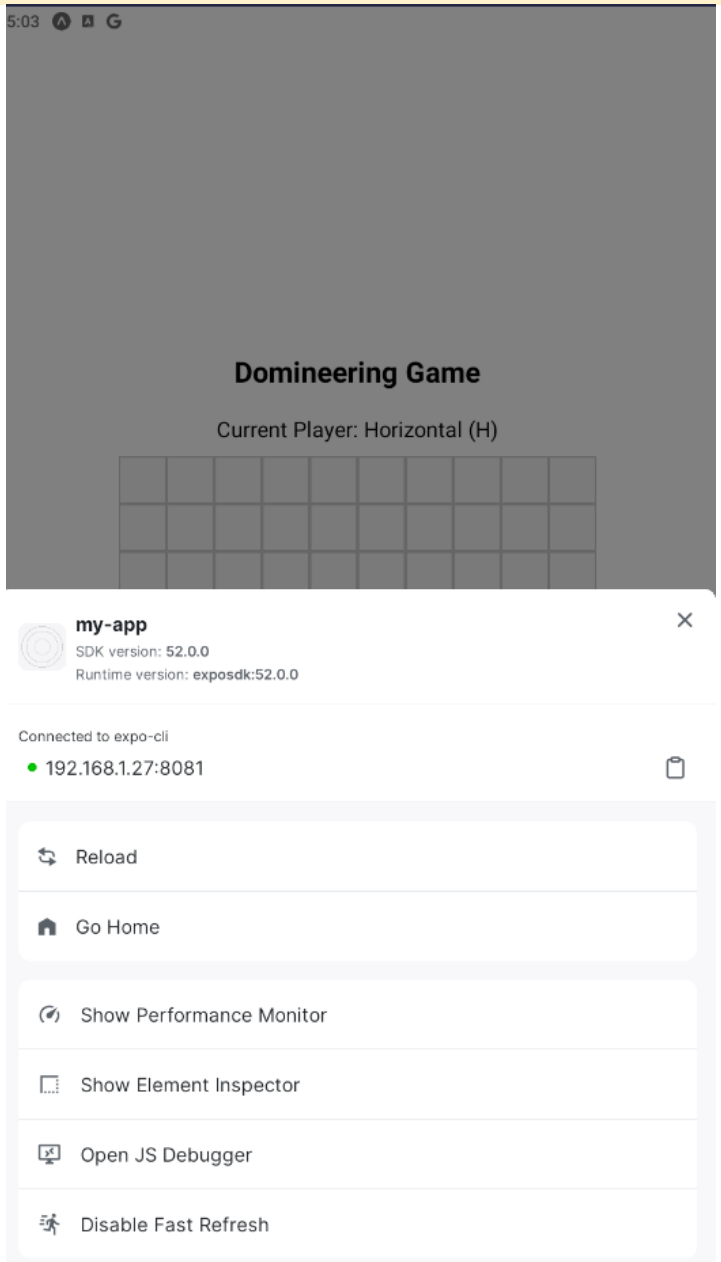
- Type **"?"** on Terminal to see full list of command while `npm start` process is running

```
› Using development build
› Press s   | switch to Expo Go

› Press a   | open Android
› shift+a   | select a device or emulator
› Press w   | open web

› Press r   | reload app
› Press j   | open debugger
› Press m   | toggle menu
› shift+m   | more tools
› Press o   | open project code in your editor
› Press c   | show project QR
```

Debugging JavaScript Remotely

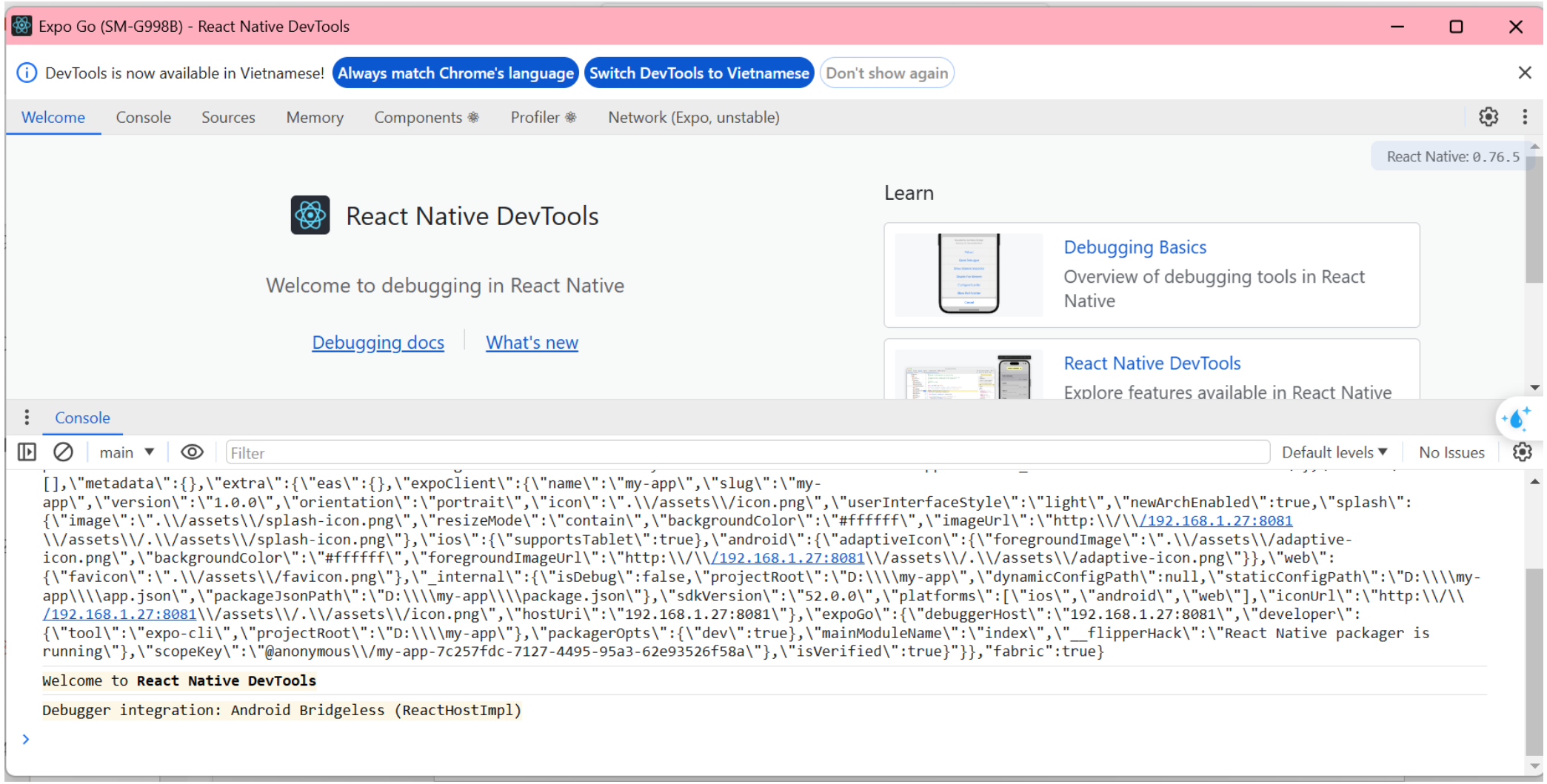
1. Type **"?"** on Terminal to see full list of command while npm start process is running
2. Press **m** to toggle a menu on device/emulator

```
Logs for your project will appear below. Press Ctrl+C to exit.
Android Bundled 34014ms index.js (656 modules)
 (NOBRIDGE) LOG  Bridgeless mode is enabled
 INFO
 💡 JavaScript logs will be removed from Metro in React Native 0.77! Please use React Native DevTools as you
r default tool. Tip: Type j in the terminal to open (requires Google Chrome or Microsoft Edge).

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

› Press a │ open Android
› shift+a │ select an Android device or emulator
› Press w │ open web

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

› Toggling dev menu
Debug: Launching DevTools...
```



Debugging JavaScript Remotely

3. Press “Open JS Debugger” to open **DevTools** to use features like Console and Network

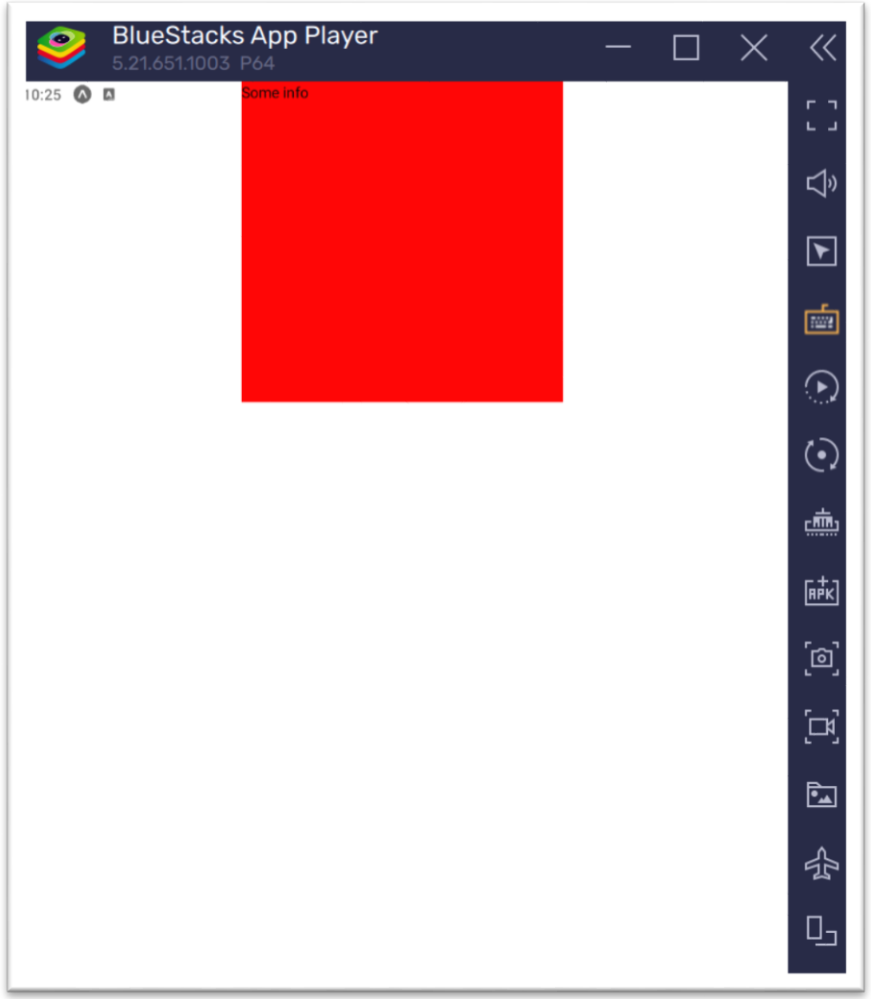


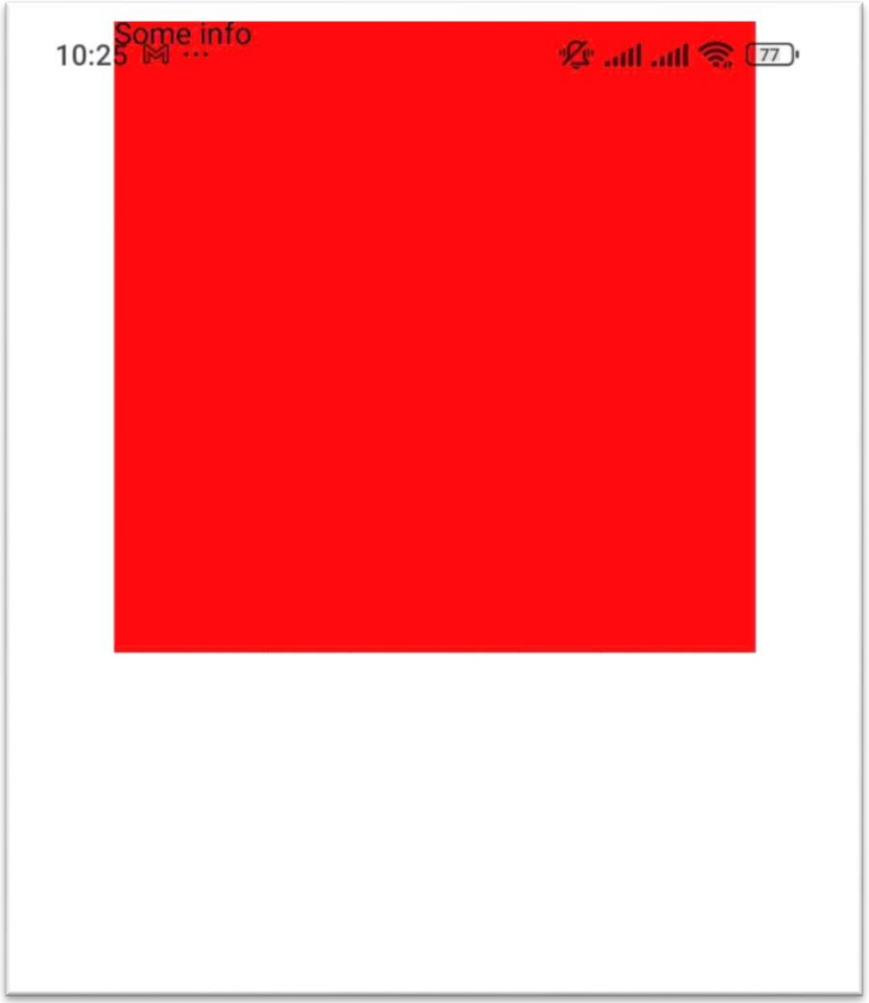
Building Adaptive User Interfaces

- Different devices have different screen sizes and resolutions.
- By default, React Native supports Density Independent Pixel (dp)
  - But we cannot build responsive UI with dp because different devices have different physical sizes (think about iPad vs iPhone).
- Problem when using absolute size:
  - Responsiveness issue
  - Inconsistent layout
  - Potential conflict with layout system

Building Adaptive User Interfaces

- Example: 300dp x 300dp square on different devices.





Building Adaptive User Interfaces

- **Responsive Units:**
  - Use relative units like `%`, `vh`, or `vw` for widths and heights.
  - These units adapt based on the parent element's size or size of the device screen.
  - Use scalable units like `em` or `rem` for font sizes
- `Dimensions` API and `PixelRatio` API
  - Query device’s screen size and calculate actual size for components
- Use third-party libraries such as `react-native-size-matters` to simplify responsive development.

Building Adaptive User Interfaces

- Using `maxWidth` or `minWidth` besides the regular width to create more responsive sizes.
  - `maxWidth` defines the maximum width an element can take up.
  - Similar to `width`, it accepts units like `dp`, `%`, or viewport units.
  - Unlike `width`, the element will only shrink if the available space in its container is less than the specified max width.
  - It acts like a ceiling for the element's width, allowing it to shrink based on available space but never exceeding the set maximum.

Building Adaptive User Interfaces

- Using `maxWidth` or `minWidth` besides the regular width to create more responsive sizes.

```
const styles = StyleSheet.create({
  title: {
    fontSize: 24,
    color: 'white',
    textAlign: 'center',
    borderWidth: 2,
    borderColor: 'white',
    padding: 12,
    maxWidth: '80%',
    width: 300
  }
})
```

Screen Orientation

- Screen orientation and styling-related problems, especially spacing, can be common challenges when developing mobile applications using React Native.

```
export default GameOverScreen;

const deviceWidth = Dimensions.get('window').width;

const styles = StyleSheet.create({
  rootContainer: {
    flex: 1,
    padding: 24,
    justifyContent: 'center',
    alignItems: 'center',
```

Dimensions API

- The Dimensions API provides a powerful tool for building responsive and adaptable layouts in React Native.
- It offers precise control over component widths, enabling you to create dynamic and visually consistent user interfaces across various screen sizes and orientations.

Dimensions API

Usage:

```javascript
import {Dimensions} from 'react-native';
```

You can get the application window's width and height using the following code:

```javascript
const windowWidth = Dimensions.get('window').width;
const windowHeight = Dimensions.get('window').height;
```

Dimensions API

- **Example:** Setting responsive padding with ternary operator

```
13  export default NumberContainer
14
15  const diviceWidth = Dimensions.get('window').width
16
17  const styles = StyleSheet.create({
18    container: {
19      borderWidth: 4,
20      borderColor: Colors.accent500,
21      padding: diviceWidth < 400 ? 16 : 24,
22      margin: 24,
23      borderRadius: 8,
24      alignItems: 'center',
25      justifyContent: 'center'
26    },
27    textNum: {
28      color: Colors.accent500,
29      fontSize: 36
30    }
31  })
32
```

Dimensions API

- useWindowDimensions is the preferred API for React components.
  - Unlike Dimensions, it updates as the window's dimensions update (this works nicely with the React model).
- Differences between using **%** and **Dimensions** API:

Feature	Using percentages (%)	Using Dimensions API
Simplicity	Easy	More complex
Responsiveness	Good	Excellent
Control over width	Limited	Precise
Potential issues	Nested elements, inconsistency	None

useWindowDimensions()

- Import the hook from `react-native`:

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

- Use it in your component:

```
const { width, height } = useWindowDimensions();
```

useWindowDimensions()

```
import React from 'react'  6.9k (gzipped: 2.7k)
import { View, StyleSheet, Text, useWindowDimensions } from 'react-native'

const App = () => {
  const { height, width, scale, fontScale } = useWindowDimensions()
  return (
    <View style={styles.container}>
      <Text style={styles.header}>Window Dimension Data</Text>
      <Text>Height: {height}</Text>
      <Text>Width: {width}</Text>
      <Text>Font scale: {fontScale}</Text>
      <Text>Pixel ratio: {scale}</Text>
    </View>
  )
}
const styles = StyleSheet.create({
  container: {
    flex: 1,
    justifyContent: 'center',
    alignItems: 'center'
  },
  header: {
    fontSize: 20,
    marginBottom: 12
  }
})

export default App
```

Managing layout when keyboard is visible

- `KeyboardAvoidingView` is a useful component that automatically adjusts its layout to ensure that important content remains visible when the virtual keyboard is displayed.
  - **When the keyboard appears** (e.g., during text input): `KeyboardAvoidingView` adjusts its height, position, or bottom padding to prevent content from being obscured by the keyboard.
  - Helpful for creating a smooth user experience when dealing with forms, input fields, and other interactive elements.

Managing layout when keyboard is visible

- Import the component from `react-native`:

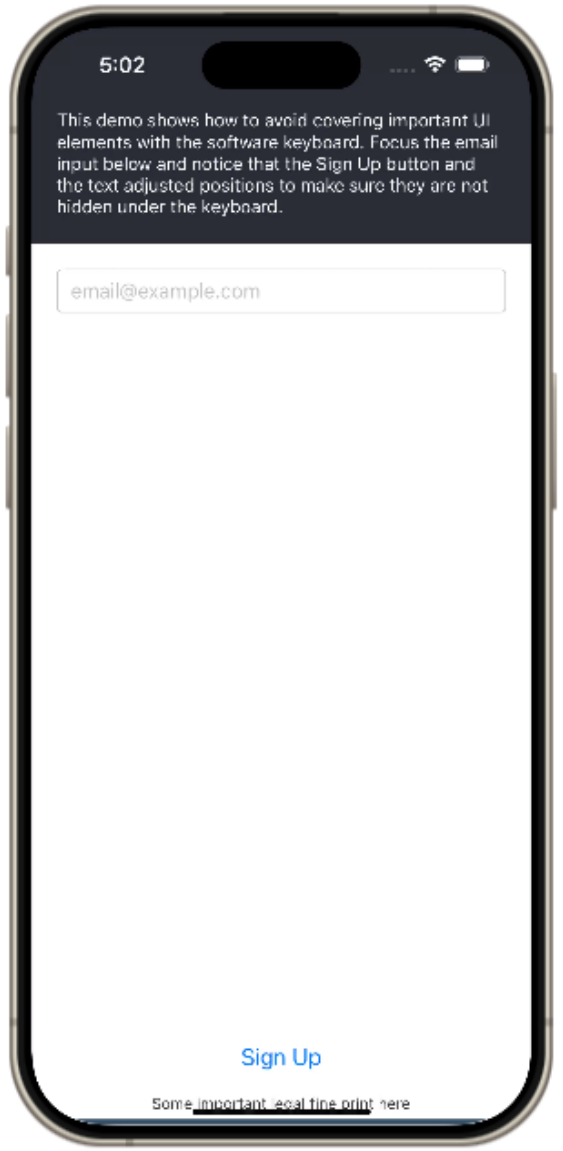
```
import { useState } from 'react';
import {
  TextInput,
  View,
  StyleSheet,
  Alert,
  useWindowDimensions,
  KeyboardAvoidingView
} from 'react-native';
```

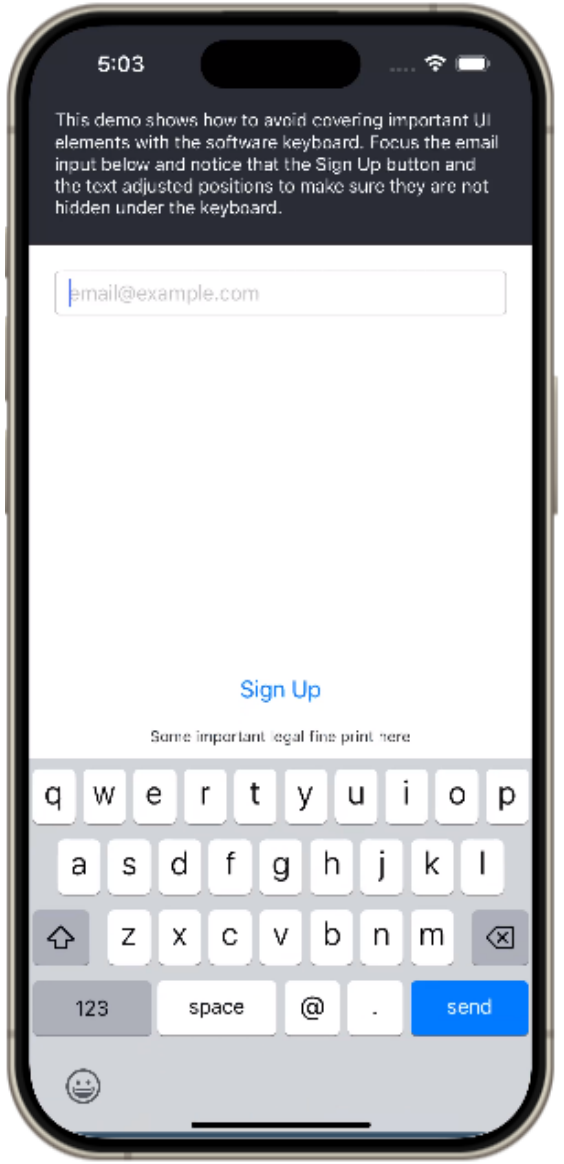
- Use the component:

```
return (
  <ScrollView style={styles.screen}>
    <KeyboardAvoidingView style={styles.screen} behavior="position">
      <View style={[styles.rootContainer, { marginTop: marginTopDistance }]}>
```

Managing layout when keyboard is visible

Example





Handling user input

- **State Management:**
  - React Native uses React's `useState` hook to manage the input values dynamically.
  - The state holds the value entered by the user.
- **Input Components:**
  - `TextInput`: The core component for receiving user input.
  - `Button`: Used to handle submission or trigger an action.
- **Events:**
  - `onChangeText`: captures changes in `TextInput` (so that we can update the state).
  - `onPress`: handles button clicks (to trigger a function to process input data).

Handling user input

Sign-Up Form Example

```
const SignUpForm = () => {
  // Step 1: Define states to store user input
  const [name, setName] = useState('');
  const [email, setEmail] = useState('');

  // Step 2: Handle form submission
  const handleSubmit = () => {
    if (!name || !email) {
      Alert.alert('Error', 'Please fill out all fields.');
      return;
    }
    Alert.alert('Success', `Hello, ${name}! A confirmation email has been sent
    to ${email}.`);
  };
```

```
return (
  <View style={styles.container}>
    <Text style={styles.label}>Name:</Text>
    <TextInput
      style={styles.input}
      placeholder="Enter your name"
      value={name}
      onChangeText={(text) => setName(text)}
    />
    <Text style={styles.label}>Email:</Text>
    <TextInput
      style={styles.input}
      placeholder="Enter your email"
      value={email}
      onChangeText={(text) => setEmail(text)}
      keyboardType="email-address"
      autoCapitalize="none"
    />
    <Button title="Submit" onPress={handleSubmit} />
  </View>
);
```

Handling user input

Notes

- **Always Validate Inputs:** Never assume the user will enter correct data.
- **Test on Devices:** Keyboard behavior might vary between platforms (iOS vs. Android).
- **Improve UX:** Use libraries like `KeyboardAvoidingView` to adjust layouts when the keyboard is visible.

Core component: `ScrollView`

```
<View style={styles.bottomSection}>
    <ScrollView>
    {goals.map(
        (g, i) => <Text key={i} style={styles.goalItem}>{g}</Text>)}
    </ScrollView>
</View>
```

- Why wrap a `View` around the `ScrollView`?
  - To display large amount of content in restricted space.
- How does the `ScrollView` behave when the content doesn't exceed the height limit?
  - It behaves like a regular `View`.

ScrollView vs FlatList

- FlatList is also scrollable
- FlatList doesn't always render all of its contents like ScrollView
- FlatList only renders visible items (better performance)
- FlatList is for lists only

```
<View style={styles.bottomSection}>
    <FlatList
        data={goals}
        renderItem={(obj) => <Text
            key={obj.index} style={styles.goalItem}
        >{obj.item}</Text>}
    />
</View>
```

Item keys in `FlatList`

- Make data a list of objects, each object has the `key` property
- Use the `keyExtractor` prop
  - When each object doesn't have a suitable key property
  - When each list item is not an object

```
<View style={styles.bottomSection}>
    <FlatList
        data={goals}
        renderItem={
            (obj) => <Text style={styles.goalItem}>{obj.item}</Text>
        }
        keyExtractor={(item, index) => index}
    />
</View>
```

Core component: `Pressable`

- Used to make other components *pressable* just like `Button` or `TouchableOpacity`
  - It supports the `onPress` event as long as other states such hover, focus, and long press.
  - More styling options than `Button`

```
<Pressable onPress={() => { console.log('Pressed') }}>
    <Text style={styles.goalItem}>{obj.item}</Text>
</Pressable>
```

Core component: `Pressable`

- Key Features of `Pressable`:
  - `onPress`: Handles the basic press event.
  - `onPressIn`: Triggered when the press gesture starts.
  - `onPressOut`: Triggered when the press gesture ends.
  - `onLongPress`: Triggered when the user presses and holds the component.
  - `style`: Allows you to define styles that change based on the component's state (e.g., pressed, hovered).

`android_ripple` effect for `Pressable`

- A feedback effect when a `Pressable` is touched
- Only works on Android!

```
<Pressable
    android_ripple={{ color: '#cccccc' }}
    onPress={() => { console.log('Pressed') }}
>
    <Text style={styles.goalItem}>{obj.item}</Text>
</Pressable>
```

Some feedback for `Pressable` on iOS

- The `style` prop can receive a callback function
  - This function receives an object provided by React Native
  - This function should return a style object

```
<Pressable
    style={(action) => {
        if (action.pressed) {
            return styles.pressedItem
        } else {
            return styles.normalItem
        }
    }}
>
    <Text style={styles.goalItem}>{obj.item}</Text>
</Pressable>
```

Core component: Image

- A component for displaying different types of images
  - Network images, static resources, temporary local images, images from local disk (such as from camera roll)

Static image:

```
<Image source={require('./assets/favicon.png')} />
```

Network image:

```
<Image
    style={{ width: 64, height: 64 }}
    source={{
        uri: 'https://reactnative.dev/img/tiny_logo.png'
    }}
/>
```

Core component: `Modal`

- A basic way to present content above an enclosing view
- A `Modal` always take the entire screen
- Let's practice with `Modal` using React Native's example:

https://reactnative.dev/docs/modal