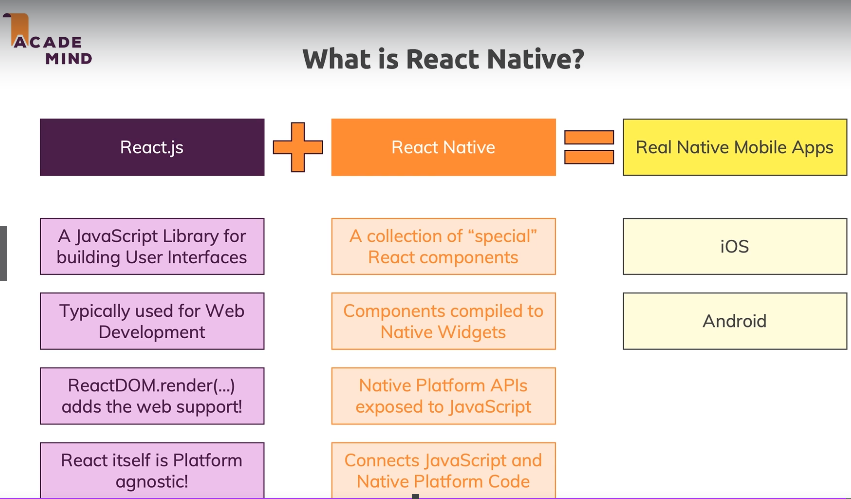
**React Native - The Practical Guide**

**Intro**

**What is React Native**



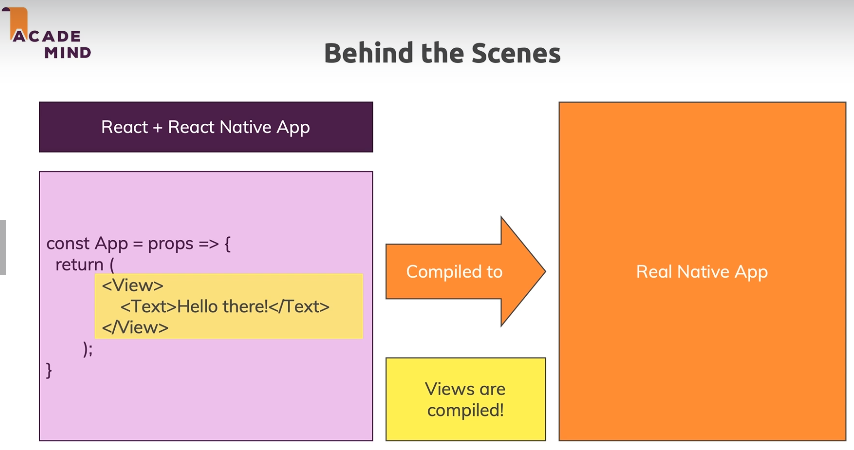
Rember React.js is only a library to build user interfaces. It does not render at any specific environment (its platform agnostic)! React for JS development uses ReactDOM library to mount itself, for example.

Which means React is an abstract conceptualization, which is capable of rendering anywhere given the proper library to do so!

React Native is a collection of React components that compile to Native Widgets, that is, to render in a native environment. It also gives you access to native apps (like the device camera and GPS location), and give you tools to bridge Javascript to native language.

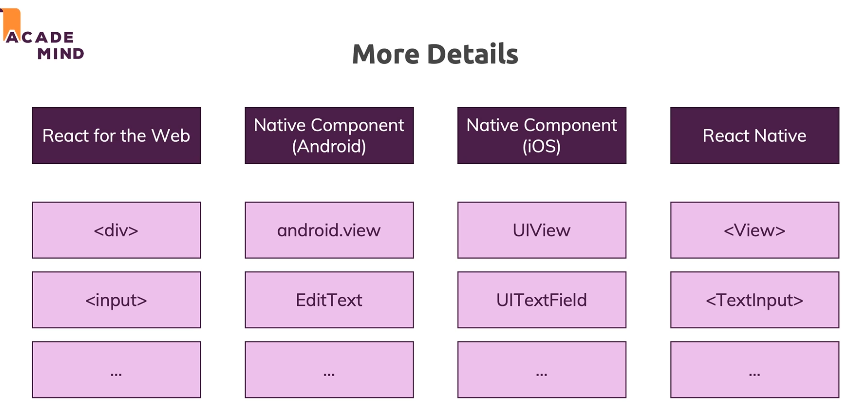
If you combine ReactJS and React Native, there you have everything you need to create and run Real Native Mobile Apps

**How React Native Works**

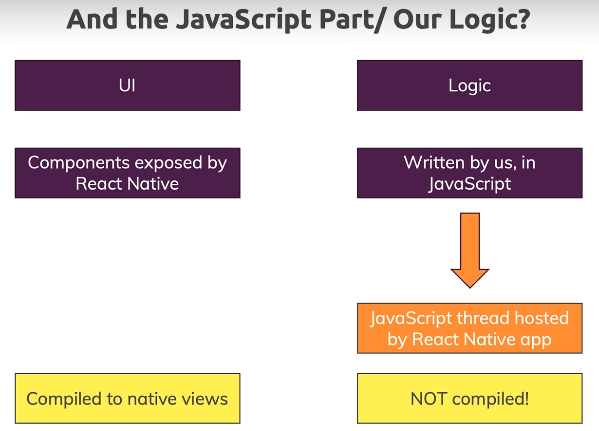


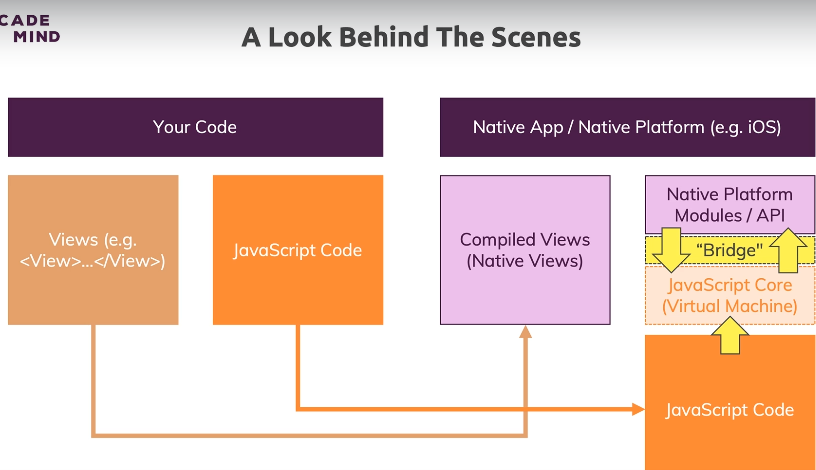
Unlike normal React.JS components in Web, React Native gives you custom components that are already linked to -and handles- the whole native logic. They compile to native widgets, elements, code.

A table of translations between web - android - iOS - React Native:



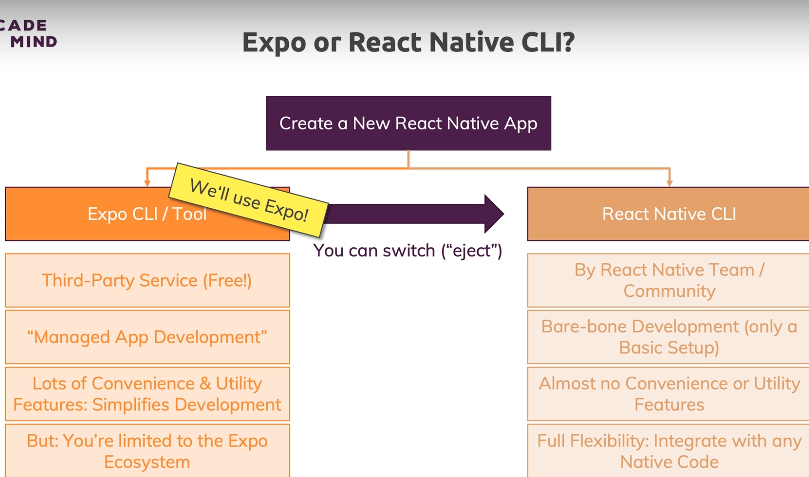
Your additional Javascript code is NOT compiled! Instead, it is run in a special thread of React Native. This means that all views are compiled into native widgets, but the logic is constantly run and handled by React Native Thread!





So, to sum it up: (1) Views are compiled to native code, (2) Javascript code constantly runs in a native thread which “bridges” (communicates) with Native Modules / APIs.

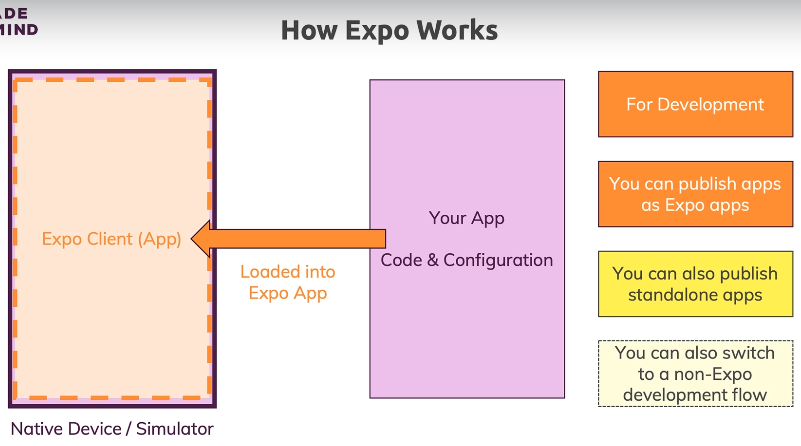
**Expo or React Native CLI?**



Expo is a wrapper around your app, an extra layer. It gives you access to plenty of native widgets and apis, but you are limited to its ecosystem. It removes some of the fine grain control, but offers you a lot of flexibility in time, in return. Plenty of components and utilities are there bridged and ready to use.

RN CLI is made by the React Team, it gives you a bare-bone setup, which needs the respective setups (IDEs), no features or convenience utilities, but is offers you fine grain control on all native widgets, as well as easier integration with any native code already in existence (Expo would require you to eject).

If you are getting started, Expo is the right choice. You can always eject to React Native CLI by ejecting. That’s why we are to use Expo to build apps here.



Behind the scenes, Expo uses a simulator to run your app (code and configs). You can also publish your app to Expo, so anyone developing with expo can also run it without downloading it.

Of course, Expo helps you to build and distribute standalone apps, directly to the iOS and Android stores.

**Node JS download**

At the moment, the Node webpage (nodejs.org) which we're going to use in the next lecture looks different.

This will only be the case temporarily (as you can tell if you visit the page) but in the meantime, since you're going to need to download NodeJS from the site, here's the download link: <https://nodejs.org/en/download/>

**Creating our first React Native App**

1. Go to nodejs.org and install it.

2. On a new folder, install expo cli globally:

***$*** (***sudo)*** ***npm install expo-cli --global***

3. Create a project:

***$ expo init rn-first-app***

*>**expo-template-blank*

4. ***$ cd rn-first-app && npm start***

Leave the process running in the terminal. This is such because an auto-updater runs each time you save your project, as it constantly runs in a watcher server.

5. **Download the Expo Client app** from the app store.

6. Open the app and **scan the QR code** given to you by Expo when you initialized the app. This compiles your app shows it in your device.

**Working our first App**

View and Text are compiled to native code, while StyleSheet is a utility function that takes an object with styles and assigns those styles to components rendered as UI.

Starting example:

// import { StatusBar } from 'expo-status-bar';

import React, { useState } from "react"

import { StyleSheet, Text, View, Button } from "react-native"

export default function App() {

const [text, setText] = useState("")

return (

<View style={styles.container}>

<Text>{text}</Text>

<Button title="Change text" onPress={() => setText("asd")} />

{/\* <StatusBar style="auto" /> \*/}

</View>

)

}

const styles = StyleSheet.create({

container: {

flex: 1,

backgroundColor: "#fff",

alignItems: "center",

justifyContent: "center"

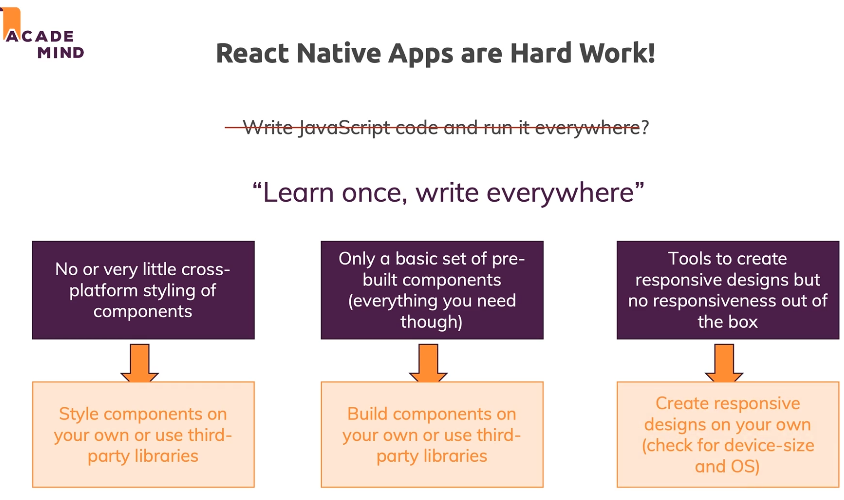
}

})

**React Native Apps are Hard Work**

Expo does make things a lot easier, but its important to know that React Native is not about Writing code once and run it everywhere, but all about Learning once and writing everywhere.

Components adjust themselves to any platform with Expo, but you will have to learn code that’s flexible depending the platform on the go.

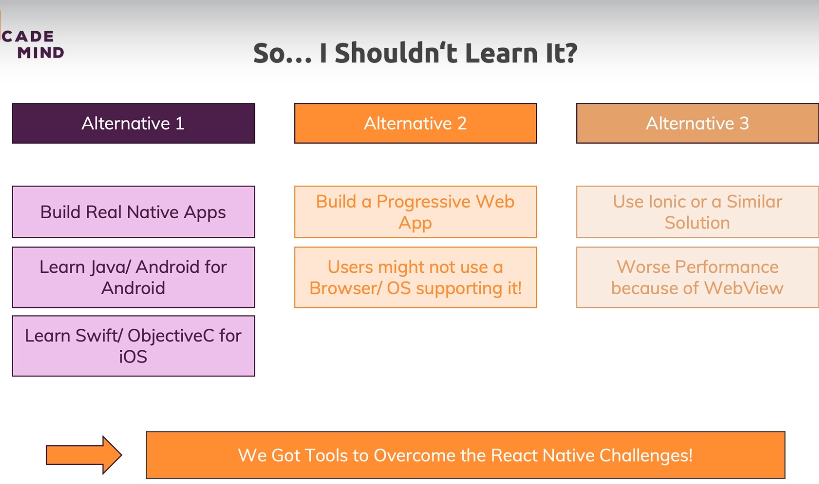


**React Native alternatives**

React Native keeps on changing. It is inevitable to go back to an app you build half a year ago to update it, breaking changes can and will happen.

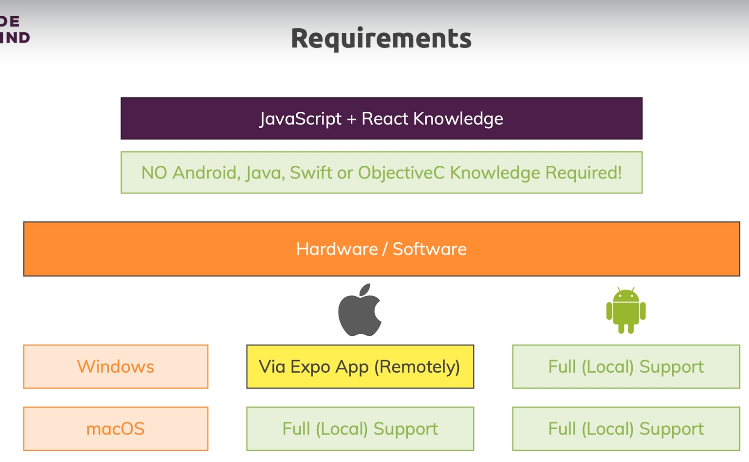
It is also highly dependant on third-party packages, that can also change. Though Expo updating generally solves this issue. However, you still have the extra dependencies you add, to be adjusted.

Sometimes, there also are bugs in RN, which you will have to keep an eye on and workaround.



Ionic provides a native wrapper for web apps to be shipped to app stores. Apps can be ran as mobile ones, but they will still appear as web apps.

**Course requirements**



**Running the app on Android Emulator**

For development, we need a simulator. It is probably not too efficient to fully develop using your own device.

For those, we will either need XCode (iOS simulator) to be run on MacOS, and/or Android Studio, for Android development, on any device.

<https://expo.io> documentation holds instructions for both.

1. Download and install Android Studio.

You need these stops on MaxOS and Linux:



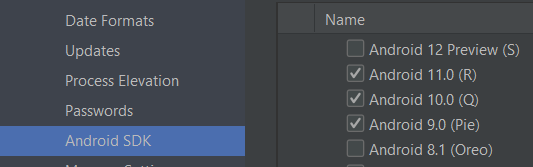
2. Open AS and on the opened window, tap Configure > SDK Manager

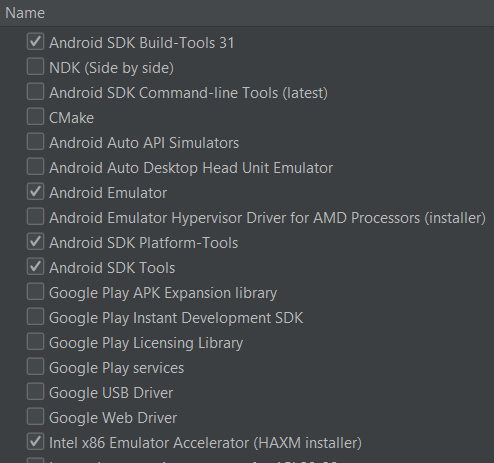
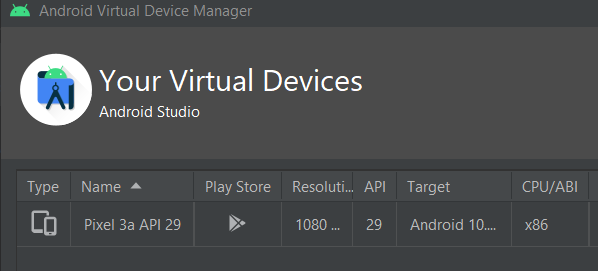
Download the latest stable SDKs, and also Android Emulator, SDK Platform, SDK Tools and SDK Build Tools on SDK Tools. Additionally, you need the intel x86 Emulator Accelerator. Tap apply.

3. Now, Configure > ADB manager.

Create different devices with different sizes (select ones with play store included), and configure them with the latest stable Android OS (these are device images, independent of the other ones you installed before).

Once installed, emulators can be launched with the play button on the emulator screen.

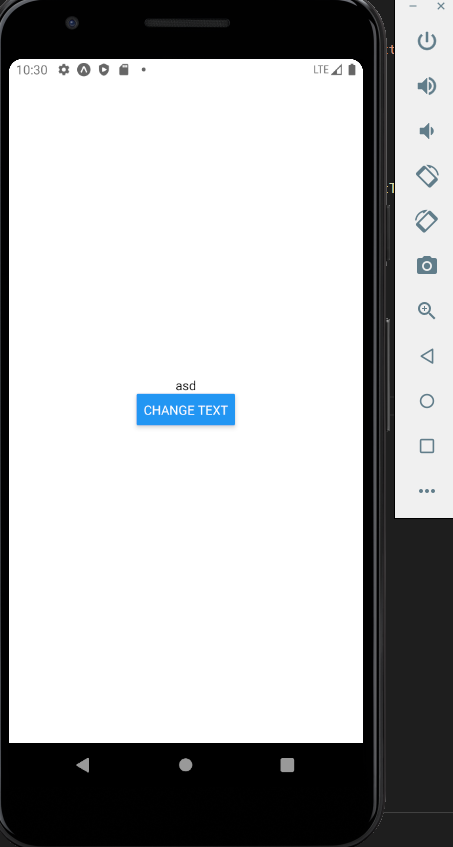




4. Once the emulator launched, run it. Go to Expo in localhost, and run the app on Android Emulator, it should run in the emulated device.

Or, on your terminal, press “a”, and the emulator should open.

Expo client will automatically install and run in the emulator.



**Running App on iOS simulator**

Note that all this only works in MacOS.

1. Go to app store and search for “XCode”, install the developer tools.

2. Open XCode and go to Preferences. Enable Command-Line Tools there.

3. Launch the simulator

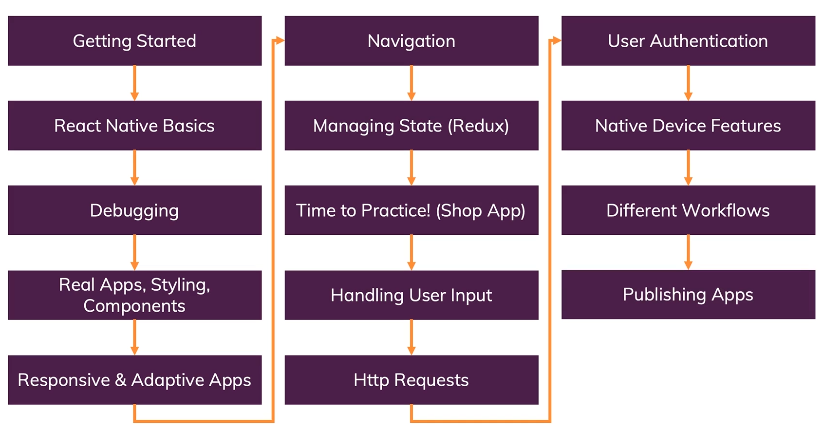
*Developer Tools > Simulator*

*Hardware > device > (select device)* -> changes the device

4. Go to Expo devtools and launch the app on iOS simulator.

Alternatively, you can tap “i” on the command line.

**Course outline**



**Useful links**

Official Expo Docs:

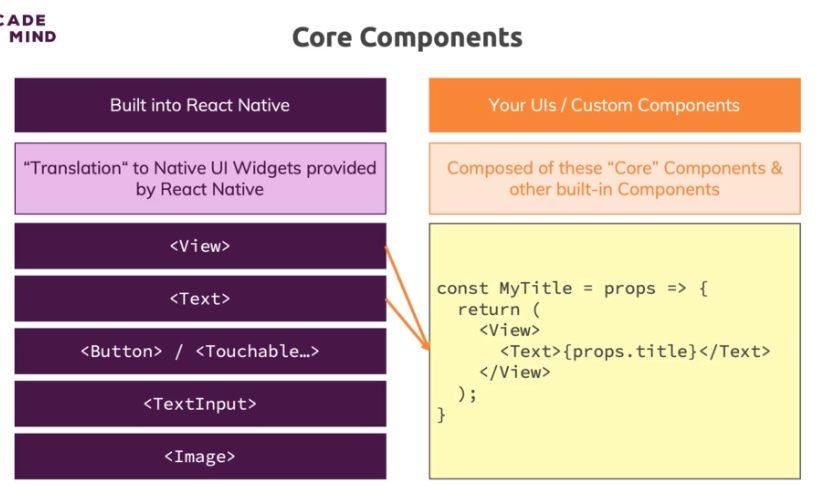
<https://docs.expo.io/versions/v34.0.0/introduction/installation/>

Official React Native Docs:

<https://facebook.github.io/react-native/docs/getting-started>

**Diving into basics**

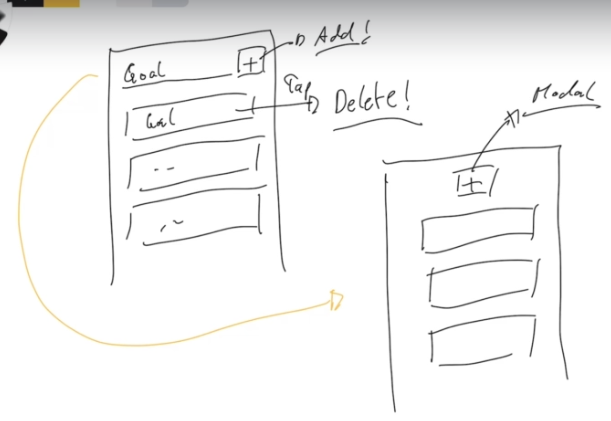
**How to work with React Native Components**





**Setting up a new project**

1. ***$ expo init rn-complete-guide && npm start***



**Working with core components**

**View** is like a div for web development. You can wrap other components and add styles.

**Text** is a generic component to display text. You cannot output text without being wrapped in this component, this is a strong difference from web development.

**Getting started with styles**

Let’s start by creating a TextInput and a Button with some stylings.

Note you can always check the available styles for each component in the documentation.

**App.js**

import React from "react"

import { StyleSheet, TextInput, View, Button } from "react-native"

export default function App() {

  return (

    <View>

      <View style={{ padding: 50 }}>

        <TextInput

          placeholder="Example placeholder"

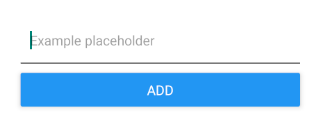
          style={{

            borderBottomColor: "black",

            borderBottomWidth: 1,

            padding: 10,

            marginVertical: 10

          }}

        />

        <Button title="add" />

      </View>

      <View></View>

    </View>

  )

}

const styles = StyleSheet.create({

  container: {

    flex: 1,

    backgroundColor: "#fff",

    alignItems: "center",

    justifyContent: "center"

  }

})

**Flexbox and layouts**

Flex stretches over the cross axis by default.

If flexDirection is row, components are stretched over the Y axis, and viceversa.

**Flex: 1** will stretch the container as much as possible. All of its siblings will only take the necessary width to render themselves visible (with their intrinsic dimensions, or their children.

The number value is itself \* 100 of the available space once the least necessary space to render everything is calculated.

If one child has flex 2, another one flex 1, and the last one, flex 3 -> first child takes 2/6 of the available space, child 2, 1/6 and last child 3/6

**App.js**

export default function App() {

return (

<View

style={{

padding: 70, width: "80%", height: 200,

flexDirection: "row",

justifyContent: "space-between",

alignItems: "stretch"

}}

>

<View

style={{

backgroundColor: "blue",

flex: 1, justifyContent: "center", alignItems: "center"

}}

>

<Text>1</Text>

</View>

<View

style={{

backgroundColor: "green",

flex: 2, justifyContent: "center", alignItems: "center"

}}

>

<Text>2</Text>

</View>

<View

style={{

backgroundColor: "red",

flex: 3, justifyContent: "center", alignItems: "center"

}}

>

<Text>3</Text>

</View>

</View>

)

}

**Inline styles and stylesheet objects**

We cannot use inline styles all the time, it gets tricky to follow. Instead, we use StyleSheet objects, which configures a CSS-in-JS style syntax to pass as classes to RN components.

**App.js**

import React from "react"

import { StyleSheet, TextInput, View, Button } from "react-native"

export default function App() {

return (

<View style={styles.container}>

<View style={styles.inputContainer}>

<TextInput

placeholder="Example placeholder" style={styles.input} />

<Button title="add" />

</View>

</View>

)

}

const styles = StyleSheet.create({

container: { padding: 70 },

inputContainer: {

flexDirection: "row", justifyContent: "space-evenly",

alignItems: "center"

},

input: {

borderBottomColor: "black", borderBottomWidth: 1,

padding: 10, width: "75%"

}

})

**Working with state and events**

Enough stylings for now, let’s add some functionality, starting with the input.

**App.js**

import React, { useState } from "react"

import {

StyleSheet, TextInput, Text, View, Button, ScrollView

} from "react-native"

export default function App() {

const [text, setText] = useState("")

const [texts, setTexts] = useState([])

function changeText(text) {

setText(text) // value is already passed as a string, not an event

}

function handleSubmit() {

setTexts((prevTexts) => [...prevTexts, text]) }

return (

<View style={styles.container}>

<View style={styles.inputContainer}>

<TextInput

placeholder="Example placeholder" style={styles.input}

value={text} onChangeText={changeText}

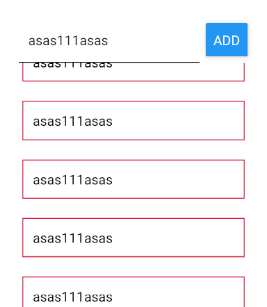
/>

<Button title="add" onPress={handleSubmit} />

</View>

<ScrollView>

{texts.map((t, i) => (

 <View style={styles.listItems} key={i}>

<Text>{t}</Text>

</View>

))}

</ScrollView>

</View>

)

}

const styles = StyleSheet.create({

container: { padding: 70 },

inputContainer: {

flexDirection: "row", justifyContent: "space-evenly",

alignItems: "center"

},

input: {

borderBottomColor: "black", borderBottomWidth: 1,

padding: 10, width: "75%"

},

listItems: {

padding: 10, margin: 10, borderColor: "crimson", borderWidth: 1

}

})

**FlatList**

ScrollView is great to scroll a little amount of components, as it renders them all to the screen. So, on a list that’s capable of having many components, we need a more efficient solution: FlatList.

FlatList handles lots of rendering logic, but expects an object as item inside the array state, with a key to use when mapping.

**App.js**

import React, { useState } from "react"

import {

StyleSheet, TextInput, Text, View, Button, FlatList

} from "react-native"

export default function App() {

const [text, setText] = useState("")

const [texts, setTexts] = useState([])

function changeText(text) { setText(text) }

function handleSubmit() {

setTexts((prevTexts) => [

...prevTexts,

**{ key: Math.random().toString(), value: text }**

])

}

return (

<View style={styles.container}>

<View style={styles.inputContainer}>

<TextInput

placeholder="Example placeholder"

value={text} onChangeText={changeText}

style={styles.input}

/>

<Button title="add" onPress={handleSubmit} />

</View>

**<FlatList**

**data={texts}**

**renderItem={(data) => (** // -> automatically sets key

<View style={styles.listItems}> // if defined in data

<Text>{**data.item.value**}</Text> // -> data.item!

</View>

**)}**

**/>**

</View>

)

}

Alternatively, you can pass a keyExtractor function to calculate the key for each item.

<FlatList

**keyExtractor={(data, i) => data.key}** // -> this is default

data={texts}

renderItem={(data) => (

<View style={styles.listItems}>

<Text>{data.item.value}</Text>

</View>

)}

/>

**Splitting the app into components**

Time to build our own components to reuse them and spread the logic into different files.

**App.js**

import React, { useState } from "react"

import { StyleSheet, View, FlatList } from "react-native"

import TextItem from "./components/TextItem"

import TextField from "./components/TextField"

export default function **App**() {

const [texts, setTexts] = useState([])

function handlePress(text) {

setTexts((prevTexts) => [

...prevTexts,

{ key: Math.random().toString(), value: text }

])

}

return (

<View style={styles.container}>

**<TextField**

**buttonProps={{ title: "add", onPress: handlePress }} />**

<FlatList

keyExtractor={(data, i) => data.key}

data={texts}

renderItem={(data) => **<TextItem>{data.item.value}</TextItem>**}

/>

</View>

)

}

const styles = StyleSheet.create({ container: { padding: 70 } })

**TextField.js**

import React, { useState } from "react"

import { View, Button, TextInput, StyleSheet } from "react-native"

export default function **TextField**({

inputProps = {}, buttonProps = {} })

{

const [text, setText] = useState("")

function handleChangeText(text) { setText(text) }

return (

<View style={styles.container}>

<TextInput

placeholder={inputProps?.placeholder || "Example placeholder"}

value={text} onChangeText={handleChangeText}

style={styles.input} {...inputProps}

/>

<Button

{...buttonProps}

**onPress={buttonProps?.onPress?.bind(this, text)}**

/>

</View>

)

}

const styles = StyleSheet.create({

container: {

flexDirection: "row", justifyContent: "space-evenly",

alignItems: "center"

},

input: {

borderBottomColor: "black", borderBottomWidth: 1,

padding: 10, width: "75%"

}

})

**TextItem.js**

import React from "react"

import { View, Text, StyleSheet } from "react-native"

export default function **TextItem**({ children }) {

return (

<View style={styles.container}>

<Text>{children}</Text>

</View>

)

}

const styles = StyleSheet.create({

container: {

padding: 10, margin: 10, borderColor: "crimson",

borderWidth: 1

}

})

**Working with touchable components**

Let’s try to delete elements now, and also use this opportunity to introduce touchable functionality.

View is not pressable by default. There are special components for this, which conveniently wrap on almost any other children: **Touchable**.

And even better, if you wish to use them as full components with added UI visibility feedback you have **TouchableOpacity**, **TouchableHighlight, TouchableWithoutFeedback** (has touchable functionality but no visual feedback)and **TouchableNativeFeedback** (this last one is the ripple effect that only works on Android).

activeOpacity prop controls the opacity intensity of Opacity.

underlayColor props controls the background color of Highlight.

**App.js**

export default function **App**() {

const [texts, setTexts] = useState([])

function handlePress(text) {

setTexts((prevTexts) => [

...prevTexts,

{ key: Math.random().toString(), value: text }

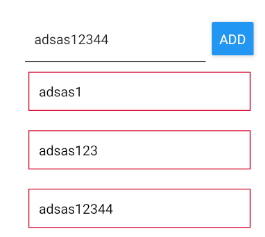
])

}

**function handleDeleteText(targetKey) {**

**setTexts(texts.filter((t) => t.key !== targetKey))**

**}**



return (

<View style={styles.container}>

<TextField buttonProps={{

title: "add", onPress: handlePress }} />

<FlatList

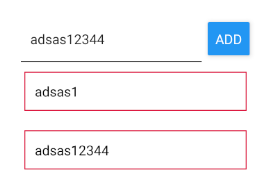
data={texts}

renderItem={(data) => (

<TextItem

**onDeleteText={handleDeleteText.bind(this, data.item.key)}>**

{data.item.value}

 </TextItem>

)}

/>

</View>

)

}

**TextItem.js**

export default function **TextItem**({ children, **onDeleteText** }) {

return (

**<TouchableOpacity activeOpacity={0.8} onPress={onDeleteText}>**

<View style={styles.container}>

<Text>{children}</Text>

</View>

**</TouchableOpacity>**

)

}

**Adding a modal overlay**

Modal is a built in, easy to use, RN component capable of adding a modal screen. Let’s use it to move input functionality there.

**App.js**

import React, { useState } from "react"

import { StyleSheet, View, FlatList, Button } from "react-native"

import TextItem from "./components/TextItem"

import TextField from "./components/TextField"

export default function **App**() {

const [texts, setTexts] = useState([])

**const [isModalOpen, setIsModalOpen] = useState(false)**

function handlePress(text) {

setTexts((prevTexts) => [

...prevTexts,

{ key: Math.random().toString(), value: text }

])

**setIsModalOpen(false)**

}

function handleDeleteText(targetKey) {

setTexts(texts.filter((t) => t.key !== targetKey))

}

**function closeModal() { setIsModalOpen(false) }**

return (

<View style={styles.container}>

**<Button title="Open" onPress={() => setIsModalOpen(true)} />**

<TextField

**modalProps={{ visible: isModalOpen }}**

**cancelButtonProps={{ onPress: closeModal }}**

**addButtonProps**={{ title: "add", onPress: handlePress }}

/>

<FlatList

data={texts}

renderItem={(data) => (

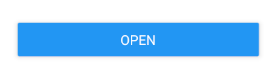
<TextItem

onDeleteText={handleDeleteText.bind(this, data.item.key)}>

{data.item.value}

</TextItem>

)}

 />

</View>

)

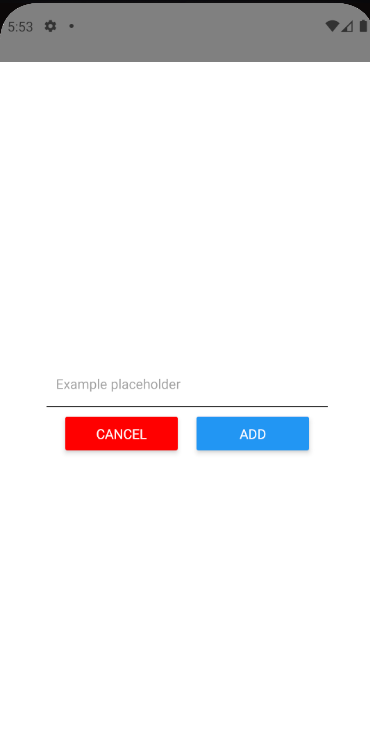
}

const styles = StyleSheet.create({ container: { padding: 70 } })

**TextField.js**

import React, { useState } from "react"

import { View, Button, TextInput, StyleSheet, **Modal** } from "react-native"



export default function **TextField**({

**modalProps = {},**

inputProps = {},

**addButtonProps** = {},

**cancelButtonProps** = {}

}) {

const [text, setText] = useState("")

function handleChangeText(text) { setText(text) }

function addNewText() {

addButtonProps?.onPress?.(text)

**setText("")**

}

return (

**<Modal animationType="slide" {...modalProps}>**

<View style={styles.container}>

<TextInput

placeholder={inputProps?.placeholder || "Example

placeholder"}

value={text}

onChangeText={handleChangeText}

style={styles.input}

{...inputProps}

/>

**<View style={styles.buttonsContainer}>**

**<View style={styles.button}>**

**<Button**

**title="cancel"**

**color="red"**

**onPress={cancelButtonProps?.onPress}**

**/>**

**</View>**

**<View style={styles.button}>**

**<Button {...addButtonProps} onPress={addNewText} />**

**</View>**

**</View>**

</View>

**</Modal>**

)

}

const styles = StyleSheet.create({

container: {

flex: 1, // <- as much space as parent gives it

justifyContent: "center",

alignItems: "center"

},

input: {

borderBottomColor: "black", borderBottomWidth: 1,

padding: 10, width: "75%", marginBottom: 10

},

**buttonsContainer: {**

**width: "75%", flexDirection: "row", justifyContent: "space-evenly"**

**},**

**button: { width: "40%" //** <- buttons need View to apply styles! **}**

})

**TextItem.js**:

import React from "react"

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

export default function **TextItem**({ children, onDeleteText }) {

return (

<TouchableOpacity activeOpacity={0.8} onPress={onDeleteText}>

<View style={styles.container}>

<Text>{children}</Text>

</View>

</TouchableOpacity>

)

}

const styles = StyleSheet.create({

container: {

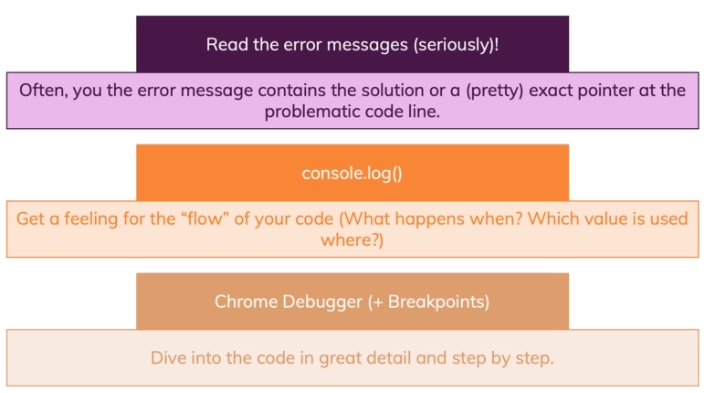
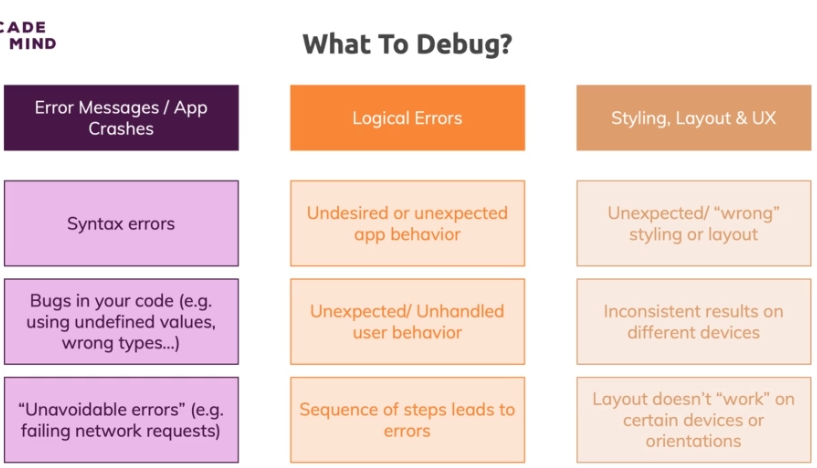
padding: 10, margin: 10, borderColor: "crimson", borderWidth: 1

}

})

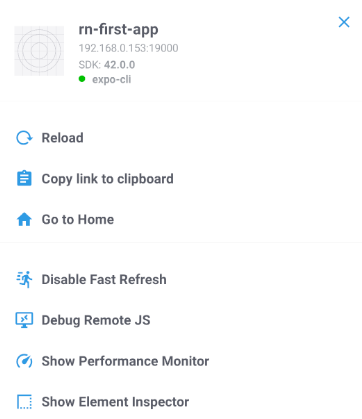
**Debugging React Native apps**

**What to and how to debug**



**Remotely debugging**

To open the developer tools, you **CMD+D** on iOS, or CTRL+M on Android.



Go to “**Debug JS remotely**” (make sure

Connection is LAN or Local in Expo

Devtools, since Tunnel is slow).

If you enable it, a web browser tab

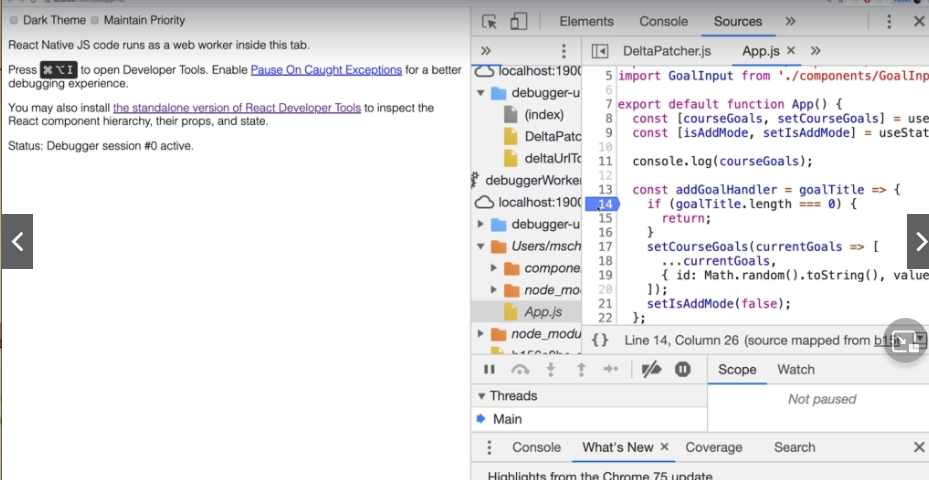
Will open where you can debug the app

On a traditional JS fashion (adding

Breakpoints).

You can also add “**debugger**” statements

In the code itself.

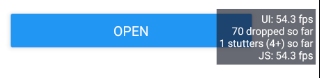


**Device Devtools Overlay**

Side note: app can be fast reloaded typing “**rr**” on Android or **CMD+R** in iOs.

Now, You can also debug on the emulated device.

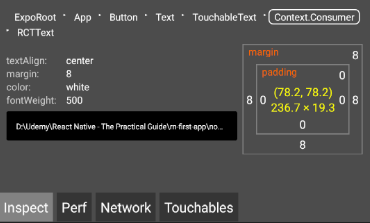
On Developer tools, tap “Performance Monitor”, a screen will appear with data on how your app is performing (FPS, render time)

Be aware that this is an INDICATOR!

It is the performance in development

Mode, in production it usually drops.

**Debugging UI and using React Native Debugger**



UI can be debugged by tapping

“**Show Element Inspector**”, and

then tapping on the element

you want to inspect.

You’ll see the layout, position

in tree and styles.

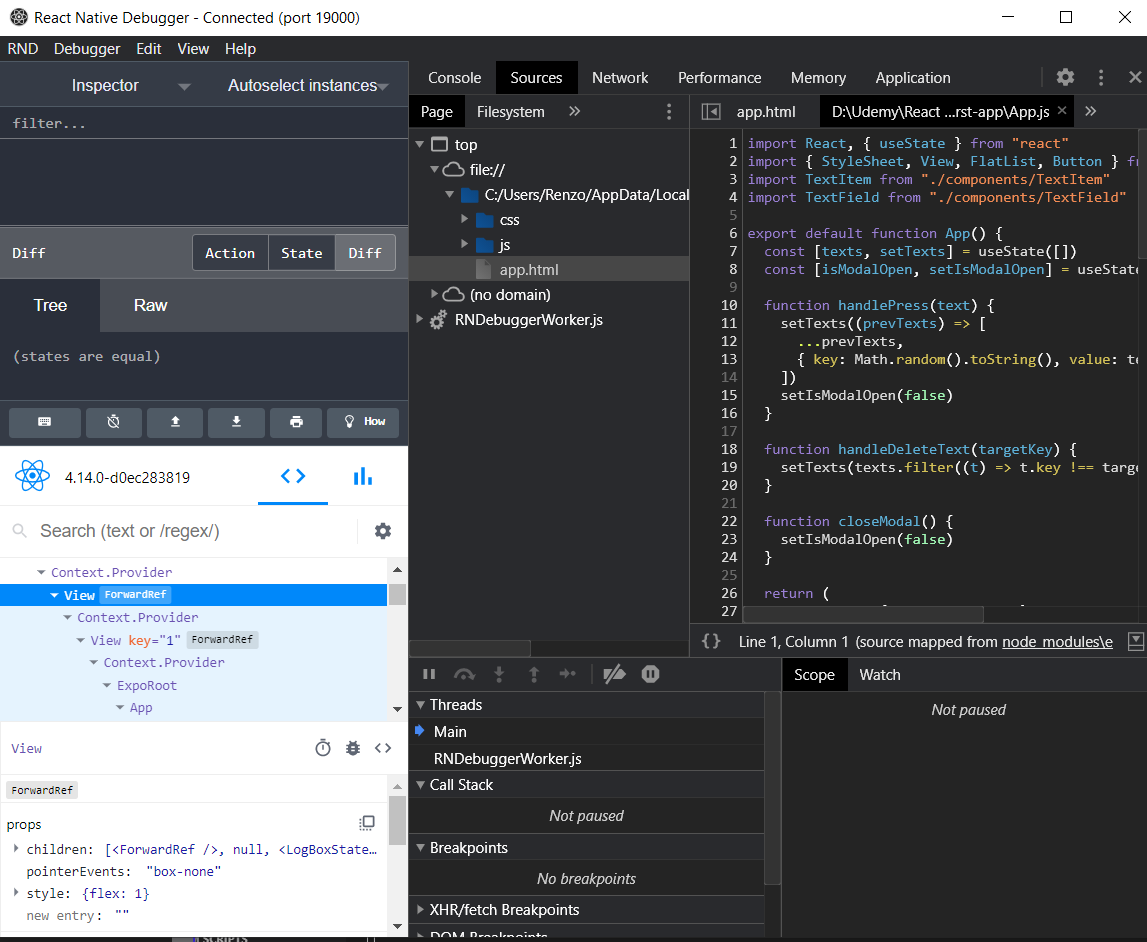
However, a better experience can be archieved by downloading the React Native Debugger.

[*https://github.com/jhen0409/react-native-debugger/blob/master/docs/getting-started.md*](https://github.com/jhen0409/react-native-debugger/blob/master/docs/getting-started.md)

> Install the correct binary according to your setup.

> Enable Javascript debugging in Development Mode.

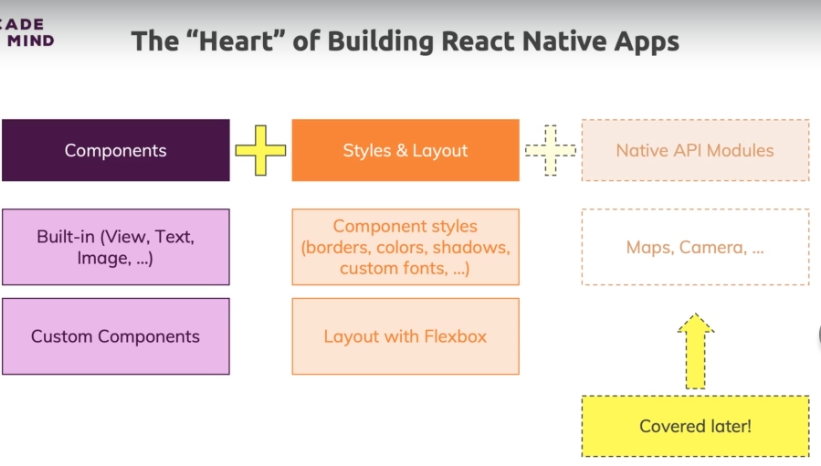
> Open React Native Debugger, CMD+T / CTRL+T and type the port (19000)



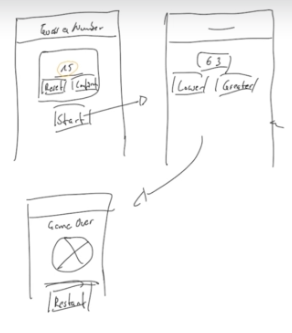
You can right click to enable network requests, which lets you check for requests sent and responses got.

**Components, stylings, layouts**

**Intro**



**Setup**



**Adding header and first screen components**

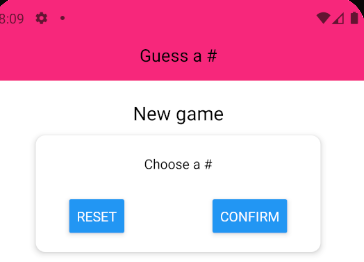
**App.js**

import React from "react"

import { StyleSheet, View } from "react-native"

**import Header from "./components/Header"**

**import StartGame from “./screens/StartGame”**

export default function **App**() {

return (

<View style={styles.container}>

**<Header title="Guess a #" />**

**<StartGame />**

</View>

)

}

const styles = StyleSheet.create({ container: { flex: 1 } })

**components/Header.js**:

import React from "react"

import { View, Text, StyleSheet } from "react-native"

export default function **Header**(props) {

return (

**<View style={styles.container}>**

**<Text style={styles.title}>{props.title}</Text>**

**</View>**

)

}

const styles = **StyleSheet.create({**

**container: {**

**width: "100%", height: 90, paddingTop: 36,**

**backgroundColor: "#f7287b", alignItems: "center",**

**justifyContent: "center"**

**},**

**title: { color: "black", fontSize: 18 }**

**})**

**screens/StartGame.js**:

import React from "react"

import { View, Text, StyleSheet, TextInput, Button } from "react-native"

export default function **StartGame**(props) {

return (

**<View style={styles.container}>**

**<Text style={styles.title}>New game</Text>**

**<View style={styles.inputContainer}>**

**<Text>Choose a #</Text>**

**<TextInput />**

**<View style={styles.buttonsContainer}>**

**<Button title="Reset" onPress={() => {}} />**

**<Button title="Confirm" onPress={() => {}} />**

**</View>**

**</View>**

**</View>**

)

}

const styles = StyleSheet.create({

**container**: { flex: 1, padding: 10, alignItems: "center" },

**title**: { fontSize: 20, marginVertical: 10 },

**inputContainer**: {

width: 300, maxWidth: "80%", alignItems: "center", padding: 20,

backgroundColor: "white", // always white bgColor.

borderRadius: 10,

// these shadow styles only work on iphone

**shadowColor: "black",**

**shadowOffset: { width: 0, height: 2 },** // right, top

**shadowRadius: 6,**

**shadowOpacity: 0.26,**

// and these ones only on android

**elevation: 5**

},

buttonsContainer: {

flexDirection: "row", width: "100%",

justifyContent: "space-between", paddingHorizontal: 15

}

})

*Screens are views for entire pages, so they go on their separate folder, other than components/. They are similar to pages/ in web development.*

**React Native Styling vs CSS Styling**

Styling in React Native is inspired by CSS - but it's not equivalent!

You must never forget that React Native in the end is all about translating React components (like <View> or <Text>) to native widgets (like UIView or widget.view)

These native widgets don't understand CSS. They have nothing to do with the web, HTML or anything like that!

What the React Native team does, is the following: They also provide "style translations" => CSS-inspired styling commands/ properties which also are translated to styling configurations those native widgets understand.

Hence backgroundColor: 'black' works - it simply targets the platform-specific configurations for the native widget that will result in a black background to be drawn. Even if these native instructions look nothing like CSS. React Native does the heavy lifting behind the scenes.

That's why many but absolutely not all CSS properties are supported in React Native. That's also why styling is done via JavaScript and not CSS. In addition, not all React Native components support all style properties.

<Text> doesn't support flexbox-related properties for example

**Extracting Card component**

**Card.js**

import React from "react"

import { View, StyleSheet } from "react-native"

export default function **Card**(**{ children, style }**) {

return <View **style={{ ...styles.container, ...style }}**>

{children}

</View>

}

const styles = StyleSheet.create({

container: {

padding: 20, backgroundColor: "white", // always white bgColor.

borderRadius: 10,

// these shadow styles only work on iphone

shadowColor: "black",

shadowOffset: { width: 0, height: 2 }, // right, top

shadowRadius: 6,

shadowOpacity: 0.26,

// and these ones only on android

elevation: 5

}

})

**StartGame.js**

import React from "react"

import { View, Text, StyleSheet, TextInput, Button } from "react-native"

**import Card from "../components/Card"**

export default function **StartGame**(props) {

return (

<View style={styles.container}>

<Text style={styles.title}>New game</Text>

**<Card style={styles.card}>**

<Text>Choose a #</Text>

<TextInput />

<View style={styles.buttonsContainer}>

<Button title="Reset" onPress={() => {}} />

<Button title="Confirm" onPress={() => {}} />

</View>

**</Card>**

</View>

)

}

const styles = StyleSheet.create({

container: { flex: 1, padding: 10, alignItems: "center" },

title: { fontSize: 20, marginVertical: 10 },

**card: { width: 300, maxWidth: "80%", alignItems: "center" },**

buttonsContainer: {

flexDirection: "row", width: "100%",

justifyContent: "space-between", paddingHorizontal: 15

}

})

**Adding TextInput and Button functionality**

**Input.js**

import React from "react"

import { TextInput, StyleSheet } from "react-native"

export default function **Input({ style, ...otherProps })** {

return **<TextInput**

**{...otherProps} style={{ ...styles.container, ...style }}**

**/>**

}

const styles = StyleSheet.create({

container: {

height: 30, borderRightColor: "gray",

borderBottomWidth: 1, marginVertical: 10

}

})

**StartGame.js**

import React, { useState } from "react"

import {

View, Text, StyleSheet, Button, **TouchableWithoutFeedback, Keyboard,**

**Alert**

} from "react-native"

import **Card** from "../components/Card"

import **Input** from "../components/Input"

import **colors** from "../constants/colors"

export default function **StartGame(props) {**

**const [num, setNum] = useState("")**

**const [confirm, setConfirm] = useState(false)**

**const [selectedNum, setSelectedNum] = useState()**

**const handleChangeText = (text) => setNum(**

**text.replace(/[^0-9]/g, "")**

**)**

**const handleResetInput = () => setNum("")**

**const handleConfirm = () => {**

**if (isNaN(chosenNum) || chosenNum <= 0 || chosenNum > 99) {**

**return Alert.alert(**

**"Invalid #",**

**"It has to be between 1 and 99 inclusive",**

**[**

**{ text: "Back",**

**style: "destructive",**

**onPress: handleResetInput**

**}**

**]**

**)}}**

**let confirmedOutput**

**if (confirm) confirmedOutput = <Text>Chosen #: {selectedNum}</Text>**

return (

**<TouchableWithoutFeedback onPress={Keyboard.dismiss}>**

<View style={styles.container}>

<Text style={styles.title}>New game</Text>

<Card style={styles.card}>

<Text>Choose a #</Text>

**<Input**

**blurOnSubmit autoCapitalize="none"**

**keyboardType="numeric" maxLength={2}**

**value={num} onChangeText={handleChangeText}**

**style={styles.input}**

**/>**

<View style={styles.buttonsContainer}>

<View style={styles.button}>

**<Button**

**title="Reset" color={colors.DANGER}**

**onPress={handleResetInput}**

**/>**

</View>

<View style={styles.button}>

**<Button**

**title="Confirm" color={colors.PRIMARY}**

**onPress={handleConfirm}**

**/>**

</View>

</View>

</Card>

**{confirmedOutput}**

</View>

**</TouchableWithoutFeedback>**

)

}

const styles = StyleSheet.create({

container: { flex: 1, padding: 10, alignItems: "center" },

title: { fontSize: 20, marginVertical: 10 },

card: { width: 300, maxWidth: "80%", alignItems: "center" },

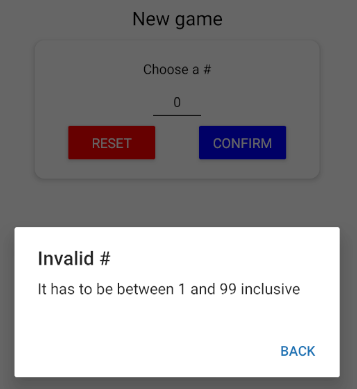
**input: { width: 50, textAlign: "center" },**

buttonsContainer: {

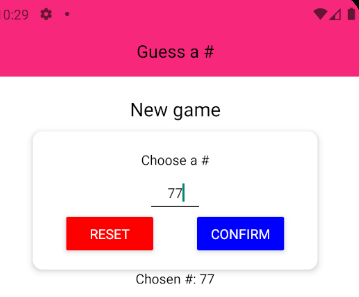
flexDirection: "row", width: "100%",

justifyContent: "space-between", paddingHorizontal: 15

},

** button: { width: "40%" }**

})



**Finishing confirmation box**

**NumberContainer.js**

import React from "react"

import { StyleSheet, View, Text } from "react-native"

import colors from "../constants/colors"

export default function **NumberContainer**({ children }) {

return (

<View style={styles.container}>

<Text style={styles.text}>{children}</Text>

</View>

)

}

const styles = StyleSheet.create({

container: {

borderWidth: 2, borderColor: colors.SECONDARY,

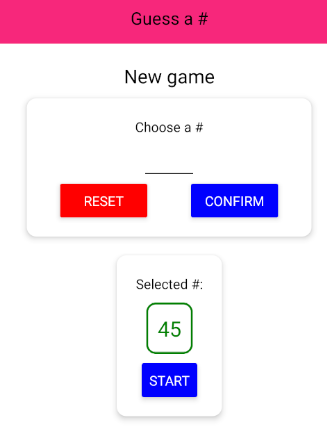
padding: 10, borderRadius: 10, marginVertical: 10,

alignItems: "center", justifyContent: "center"

},

text: { color: colors.SECONDARY, fontSize: 22 }

})

****

**StartGame.js**

…

if (confirm)

confirmedOutput = (

**<Card style={styles.summaryCard}>**

<Text>Selected #:</Text>

**<NumberContainer>**

**{selectedNum}**

**</NumberContainer>**

**<Button title="Start"**

**color={colors.PRIMARY} onPress**

**/>**

**</Card>**

)

…

const styles = StyleSheet.create({

…

**summaryCard: { marginTop: 20, alignItems: "center" }**

})

**Adding GameScreen and game logic**

**App.js**

import React, { useState } from "react"

import { StyleSheet, View } from "react-native"

import Header from "./components/Header"

import **GameScreen** from "./screens/Game"

import **GameOverScreen** from "./screens/GameOver"

import StartGameScreen from "./screens/StartGame"

export default function **App**() {

**const [userNum, setUserNum] = useState(0)**

**const [guessRounds, setGuessRounds] = useState(0)**

**const handleRestart = () => {**

**setGuessRounds(0)**

**setUserNum(0)**

**}**

const handleStartGame = (selectedNum) => setUserNum(selectedNum)

**const handleGameOver = (numGuesses) => setGuessRounds(numGuesses)**

**let content = <StartGameScreen onStartGame={handleStartGame} />**

**if (userNum && guessRounds <= 0) {**

**content = <GameScreen**

**userChoice={userNum} onGameOver={handleGameOver} />**

**} else if (guessRounds > 0) {**

**content = (**

**<GameOverScreen**

**{...{ userNum, guessRounds }} onRestart={handleRestart} />**

**)**

**}**

return (

<View style={styles.container}>

<Header title="Guess a #" />

**{content}**

</View>

)

}

const styles = StyleSheet.create({ container: { flex: 1 } })

**GameOverScreen.js**

import React from "react"

import { View, Text, StyleSheet, Button } from "react-native"

export default function **GameOverScreen**(**{ guessRounds, userNum, onRestart }**) {

return (

<View style={styles.container}>

<Text>Game over!</Text>

<Text>PC Guesses: {**guessRounds**}</Text>

<Text>Number: {**userNum**}</Text>

<Button title="new game" onPress={**onRestart**} />

</View>

)

}

const styles = StyleSheet.create({

container: { flex: 1, justifyContent: "center", alignItems: "center"}

})

**GameScreen.js**

import React, { useState, **useRef**, **useEffect** } from "react"

import { View, Text, StyleSheet, Button, **Alert** } from "react-native"

import **Card** from "../components/Card"

import **NumberContainer** from "../components/NumberContainer"

export default function **GameScreen({ userChoice, onGameOver })** {

**const [curGuess, setCurGuess] = useState(rng(1, 100, userChoice))**

**const [guesses, setGuesses] = useState(0)**

**const curLow = useRef(1)**

**const curHi = useRef(100)**

**useEffect(() => {**

**if (curGuess === userChoice) onGameOver(guesses)**

**}, [curGuess, userChoice, onGameOver])**

**const handleNextGuess = (direction) => {**

**if (**

**(direction === "lower" && curGuess < userChoice) ||**

**(direction === "greater" && curGuess > userChoice)**

**) {**

**return Alert.alert("No cheating", "You know that's not true.", [**

**{ text: "Sorry :c", style: "cancel" }**

**])**

**}**

**if (direction === "lower") curHi.current = curGuess**

**else curLow.current = curGuess**

**const newGuess = rng(curLow.current, curHi.current, curGuess)**

**setCurGuess(newGuess)**

**setGuesses((prevGuesses) => prevGuesses + 1)**

**}**

return (

<View style={styles.container}>

<Text>PC's guess</Text>

**<NumberContainer>{curGuess}</NumberContainer>**

**<Card style={styles.buttonsContainer}>**

**<Button**

**title="lower"**

**onPress={handleNextGuess.bind(this, "lower")} />**

**<Button**

**title="greater"**

**onPress={handleNextGuess.bind(this, "greater")}**

**/>**

</Card>

</View>

)

}

const styles = StyleSheet.create({

container: { flex: 1, padding: 10, alignItems: "center" },

**buttonsContainer: {**

**flexDirection: "row", justifyContent: "space-evenly",**

**marginTop: 20, width: 300, maxWidth: "80%"**

**}**

})

**function rng(min, max, exclude) {**

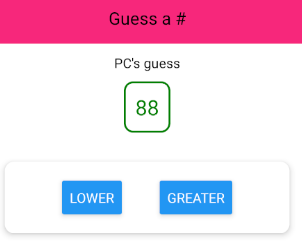
**min = Math.ceil(min) // include 1**

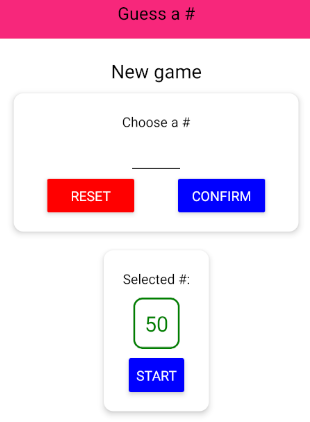
**max = Math.floor(max) // exclude 100**

**const rndNum = Math.floor(Math.random() \* (max - min)) + min**

**if (rndNum === exclude) return rng(min, max, exclude)**

**else return rndNum**

**}**





**Adding custom fonts**

1. Extract and add two fonts in **.ttf** format to *assets/fonts/*

2. To import, expo-font should be available by default, but if not:

***$ expo install expo-font***

**App.js**

import React, { useEffect, useState } from "react"

import { **ActivityIndicator**, StyleSheet, View } from "react-native"

**import \* as Font from "expo-font"**

…

export default function **App**() {

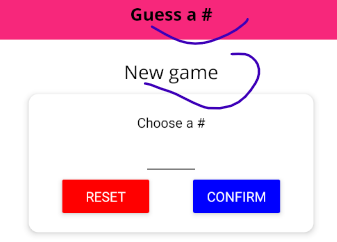
const [userNum, setUserNum] = useState(0)

const [guessRounds, setGuessRounds] = useState(0)

**const [isLoaded, setIsLoaded] = useState(false)**

// useFont hook from "expo-font" is the default today!

**useEffect(() => {**

** fetchFonts()**

**.then(() => setIsLoaded(true))**

**.catch(() => setIsLoaded(true))**

**}, [])**

**if (!isLoaded) {**

**return (**

**<View style={**

**{ ...styles.container, ...styles.loadingContainer }}**

**>**

**<ActivityIndicator size="large" color={colors.SECONDARY} />**

**</View>**

**)**

**}**

const handleRestart = () => { setGuessRounds(0); setUserNum(0) }

…

return (

<View style={styles.container}>

<Header title="Guess a #" /> {content} </View>

)

}

const styles = StyleSheet.create({

container: { flex: 1 },

**loadingContainer: { justifyContent: "center", alignItems: "center" }**

})

**function fetchFonts() {**

**return Font.loadAsync({**

**"open-sans": require("./assets/fonts/OpenSans-Regular.ttf"),**

**"open-sans-bold": require("./assets/fonts/OpenSans-Bold.ttf")**

**})**

**}**

**Header.js**

…

export default function **Header**(props) {

return (

<View style={styles.container}>

<Text style={styles.title}>{props.title}</Text>

</View>

)

}

const styles = StyleSheet.create({

…

**title**: { color: "black", fontSize: 18, **fontFamily: "open-sans-bold"** }

})

**StartGame.js**

…

export default function **StartGameScreen**({ onStartGame }) {

…

const styles = StyleSheet.create({

container: { flex: 1, padding: 10, alignItems: "center" },

title: { fontSize: 20, marginVertical: 10, **fontFamily: "open-sans"** },

…

})

**Installing expo-font**

Depending on the version of Expo you're using, you very likely need to install the expo-font package.

You can do this in two different ways and it's important to understand the difference:

1) npm install --save expo-font

2) expo install expo-font

2) is recommended - but what is the difference?

npm install installs a packages a dependency into the project - we use this command for most packages which we do install.

Some packages (typically all expo-\* packages) can break the app if you install the wrong version though - because they closely work together with Expo itself.

To get the right package version for the specific version of Expo your app relies on, expo install is the right "tool". It also just executes npm install behind the scenes but it picks a specific (i.e. the correct) version of the package to be installed.

Hence for all expo-\* packages, npm install can be used but expo install is the preferred command to avoid errors. Of course you could always try npm install first and only run expo install if you thereafter do face any errors.

**Synthetic Style Cascade custom component**

If you want to use the same styles for one component (example with fontFamily used before), you have two alternatives:

1. ***Create a custom component and re-use it everywhere****.*

**NormalText.js**

import React from "react"

import { Text, StyleSheet } from "react-native"

export default function **NormalText**(

{ children, style, ...otherProps }

) {

return (

<Text **style={{ ...styles.container, ...style }}** {...otherProps}>

{children}

</Text>

)

}

const styles = StyleSheet.create({

**container: { fontFamily: "open-sans" }**

})

**GameOverScreen.js**

import React from "react"

import { View, StyleSheet, Button } from "react-native"

**import NormalText from "../components/NormalText"**

export default function **GameOverScreen**({ guessRounds, userNum, onRestart }) {

return (

<View style={styles.container}>

<**NormalText**>Game over!</**NormalText**>

<**NormalText**>PC Guesses: {guessRounds}</**NormalText**>

<**NormalText**>Number: {userNum}</**NormalText**>

 <Button title="new game" onPress={onRestart} />

</View>

)

}

const styles = StyleSheet.create({

container: { flex: 1, justifyContent: "center", alignItems: "center" }

})

2. ***Create a constants file with a default exported StyleSheet to be imported anywhere as styles****.*

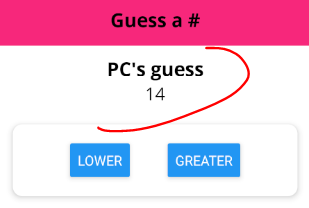
**constants/defaultStyles.js**

import { StyleSheet } from "react-native"

**const defaultStyles = StyleSheet.create({**

**NORMAL\_TEXT: { fontFamily: "open-sans", fontSize: 18 },**

**BOLD\_TEXT: { fontFamily: "open-sans-bold", fontSize: 20 }**

**})**

export default defaultStyles

**Game.js**

…

export default function **GameScreen**({ userChoice, onGameOver }) {

…

return (

<View style={styles.container}>

<Text **style={defaultStyles.BOLD\_TEXT}**>PC's guess</Text>

<Text **style={defaultStyles.NORMAL\_TEXT}**>{curGuess}</Text>

<Card style={styles.buttonsContainer}>

<Button

title="lower" onPress={handleNextGuess.bind(this, "lower")}

/>

<Button

title="greater"

onPress={handleNextGuess.bind(this, "greater")}

/>

</Card>

</View>

)

}

**Adding local images**

Two ways:

1. ***Locally:***

**GameOver.js**

import React from "react"

import { View, StyleSheet, Button, **Image** } from "react-native"

import BoldText from "../components/BoldText"

import NormalText from "../components/NormalText"

export default function **GameOverScreen**(

{ guessRounds, userNum, onRestart }

) {

return (

<View style={styles.container}>

<BoldText>Game over!</BoldText>

**<View style={styles.imageContainer}>**

**<Image**

**source={require("../assets/images/success.png")}**

**style={styles.image}**

**resizeMode="cover"** // 'contain' 'center' 'repeat' 'stretch'

**/>**

**</View>**

<NormalText>PC Guesses: {guessRounds}</NormalText>

<NormalText>Number: {userNum}</NormalText>

<Button title="new game" onPress={onRestart} />

</View>

)

}

const styles = StyleSheet.create({

container: { flex: 1, justifyContent: "center", alignItems: "center" },

**imageContainer: {**

**width: 300,** // hard coded for now. Will media query later!

**height: 300,**

**borderRadius: 150, borderWidth: 3, borderColor: "black",**

**overflow: "hidden", marginVertical: 30**

**},**

**image: { width: "100%", height: "100%" }**

})



2. ***Remotely:***

**GameOver.js**

…

export default function **GameOverScreen**(

{ guessRounds, userNum, onRestart }

) {

return (

<View style={styles.container}>

<BoldText>Game over!</BoldText>

<View style={styles.imageContainer}>

<Image

**source={{**

**uri:** "https://explorersweb.com/wp-content/

uploads/2021/05/Summit-Everest-MingmaG.jpg"

**}}**

**fadeDuration={500}** // fadein effect on img load, 300 default

style={styles.image}

resizeMode="cover" // 'contain' 'center' 'repeat' 'stretch'

/>

</View>

<NormalText>PC Guesses: {guessRounds}</NormalText>

<NormalText>Number: {userNum}</NormalText>

<Button title="new game" onPress={onRestart} />

</View>

)

}

const styles = StyleSheet.create({

container: { flex: 1, justifyContent: "center", alignItems: "center" },

imageContainer: {

width: 300, // hard coded for now. Will media query later!

height: 300,

borderRadius: 150, borderWidth: 3, borderColor: "black",

overflow: "hidden", marginVertical: 30

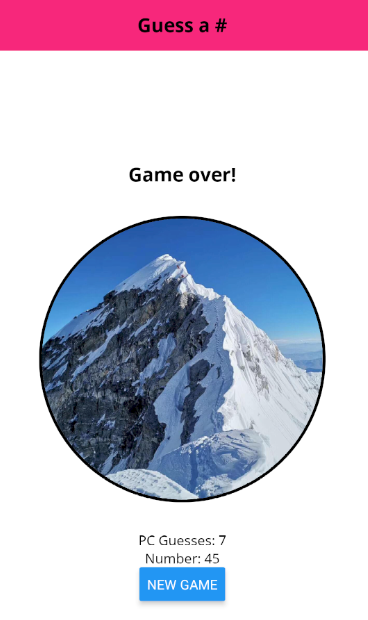
},

// RN is able to calculate width/height of local images, but NOT

// from remote ones! **These last ones** **ALWAYS need height and width!**

image: { width: "100%", height: "100%" }

})



**Closer look on <Text>**

**You can nest <Text /> components!**

This is used to style texts inside other ones.

Also, this way, inner Texts “inherit” styles from parents.

> One of the few instances this apply in RN!

<View> can also be nested inside <Text>, but there can be issues with that.

Text does NOT use flexbox by default, like other RN components

Text wraps into a new line if it does not fit inside the screen.

If you do not want this behavior, you can set **numberOfLines** prop, combined with **ellipsizeMode** ‘truncate’

**GameOver.js**

import React from "react"

import { View, StyleSheet, Button, Image, **Text** } from "react-native"

import BoldText from "../components/BoldText"

import NormalText from "../components/NormalText"

import **colors** from "../constants/colors"

export default function GameOverScreen({ guessRounds, userNum, onRestart }) {

return (

<View style={styles.container}>

<BoldText>Game over!</BoldText>

<View style={styles.imageContainer}>

<Image

source={require("../assets/images/success.png")}

style={styles.image}

resizeMode="cover"

/>

</View>

**<NormalText style={styles.textWrapper}>**

PC Guesses:

**<Text style={styles.highlight}>{**guessRounds**}</Text>.**

# was:

**<Text style={styles.highlight}>{**userNum**}</Text>**

**</NormalText>**

<Button title="new game" onPress={onRestart} />

</View>

)

}

const styles = StyleSheet.create({

…

**textWrapper: { marginVertical: 20, textAlign: "center" },**

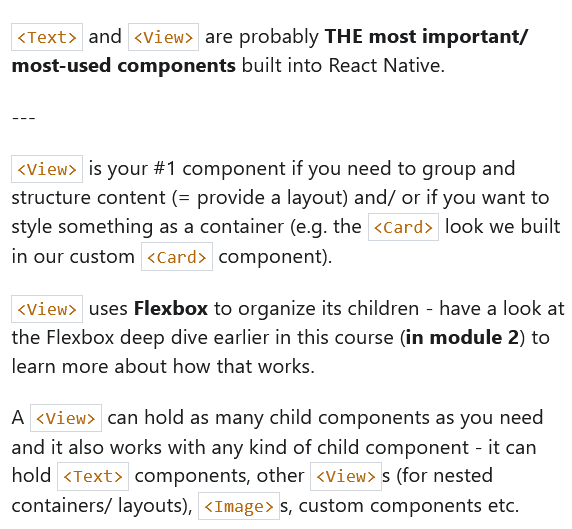
**highlight: {**

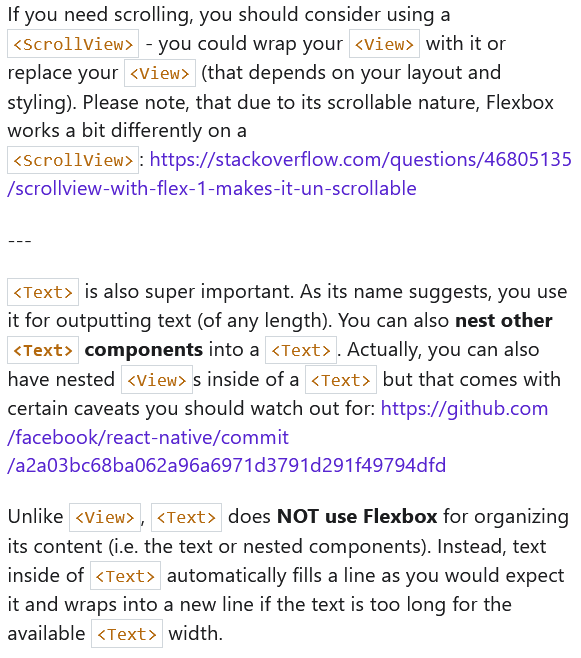
**color: colors.PRIMARY, fontFamily: "open-sans-bold",**

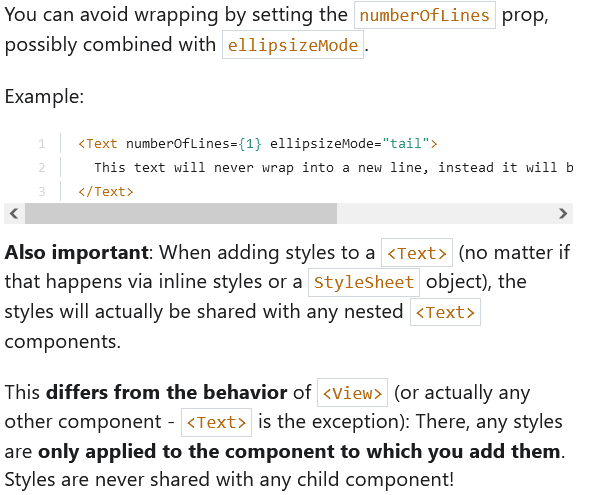
**marginHorizontal: 30**

**} })**

**<View> vs <Text>**







<Text **numberOfLines={1} ellipsizeMode="tail"**>

Text will be cut off like this if it is too **lon...**

</Text>

**Custom Button component and icons**

There a many icon packages in expo: IonicIcons, EvilIcons, MaterialIcons, AntDesign, and such. **Check the docs!**

**MainButton.js**

import React from "react"

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

import colors from "../constants/colors"

export default function **MainButton({**

**children, onPress, styles = {}, ...rest**

**}) {**

**return (**

**<TouchableOpacity**

**style={{ ...\_styles.container, ...styles.container }}**

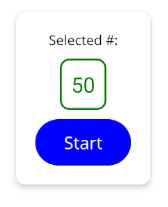
**activeOpacity={0.4} onPress={onPress} {...rest}**

**>**

**<View style={{ ...\_styles.view, ...styles.view }}>**

**<Text style={{ ...\_styles.text, ...styles.text }}>**

**{children}**

 **</Text>**

**</View>**

**</TouchableOpacity>**

**)**

**}**

**const \_styles = StyleSheet.create({**

**container: {},**

**view: {**

**backgroundColor: colors.PRIMARY, paddingVertical: 12,**

**paddingHorizontal: 30, borderRadius: 25**

**},**

**text: { color: "white", fontFamily: "open-sans", fontSize: 18 }**

**})**

**StartGameScreen.js**

…

import **MainButton** from "../components/MainButton"

…

export default function **StartGameScreen**({ onStartGame }) {

…

if (confirm)

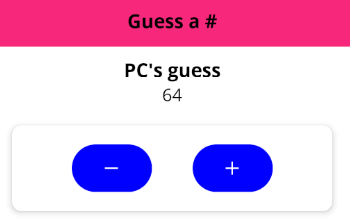
confirmedOutput = (

<Card style={styles.summaryCard}>

<NormalText>Selected #:</NormalText>

<NumberContainer>{selectedNum}</NumberContainer>

**<MainButton** onPress={() => onStartGame(selectedNum)}**>**

**** Start

**</MainButton>**

</Card>

)

…

**GameScreen.js**

…

import { Ionicons } from "@expo/vector-icons"

import Card from "../components/Card"

export default function **GameScreen**({ userChoice, onGameOver }) {

…

return (

<View style={styles.container}>

<Text style={defaultStyles.BOLD\_TEXT}>PC's guess</Text>

<Text style={defaultStyles.NORMAL\_TEXT}>{curGuess}</Text>

<Card style={styles.buttonsContainer}>

**<MainButton** onPress={handleNextGuess.bind(this, "lower")}**>**

**<Ionicons name="md-remove" size={24} color="white" />**

**</MainButton>**

**<MainButton** onPress={handleNextGuess.bind(this, "greater")}**>**

<Ionicons name="md-add" size={24} color="white" />

**</MainButton>**

</Card>

</View>

)

}

const styles = StyleSheet.create({

container: { flex: 1, padding: 10, alignItems: "center" },

buttonsContainer: {

flexDirection: "row", justifyContent: "space-evenly",

marginTop: 20, width: 400, maxWidth: "90%"

}

})

**Exploring UI libraries**

Keep in mind most RN components do require you styling lots.

You can always do so if you need or want to, but there are many UI libraries that have components ready to use in RN, like AntDesign, Ionic, Material Design, Bootstrap, and others.

<https://reactnativeelements.com/>

<https://github.com/GeekyAnts/NativeBase>

**Managing past guesses as list**

***Note:*** *If you wish to add width to a ScrollView,* ***wrap it in View****!*

***Note****: When using ScrollView, you need to set* ***flexGrow*** *(not flex).*

*This ensures that all space is taken, AS WELL AS being able to exceed boundaries, which will show first and last items on screen.*

**GameScreen.js**

import React, { useState, useRef, useEffect } from "react"

import { View, Text, StyleSheet, Alert, ScrollView } from "react-native"

import { Ionicons } from "@expo/vector-icons"

import Card from "../components/Card"

import MainButton from "../components/MainButton"

**import NormalText from "../components/NormalText"**

import defaultStyles from "../constants/default-styles"

export default function **GameScreen**({ userChoice, onGameOver }) {

const [curGuess, setCurGuess] = useState(rng(1, 100, userChoice))

**const [guesses, setGuesses] = useState([curGuess])**

const curLow = useRef(1)

const curHi = useRef(100)

useEffect(() => {

if (curGuess === userChoice) onGameOver(**guesses.length**)

}, [curGuess, userChoice, onGameOver])

const handleNextGuess = (direction) => {

if (

(direction === "lower" && curGuess < userChoice) ||

(direction === "greater" && curGuess > userChoice)

) {

return Alert.alert("No cheating", "You know that's not true.", [

{ text: "Sorry :c", style: "cancel" }

])

}

if (direction === "lower") curHi.current = curGuess

else curLow.current = curGuess **+ 1** // as not to repeat keys

const newGuess = rng(curLow.current, curHi.current, curGuess)

setCurGuess(newGuess)

setGuesses((prevGuesses) => **[newGuess, ...prevGuesses]**)

}

**const renderListItem = (value, index) => (**

**<View key={value} style={styles.listItem}>**

**<NormalText>#{guesses.length - index}</NormalText>**

**<NormalText>{value}</NormalText>**

**</View>**

**)**

return (

<View style={styles.container}>

<Text style={defaultStyles.BOLD\_TEXT}>PC's guess</Text>

<Text style={defaultStyles.NORMAL\_TEXT}>{curGuess}</Text>

<Card style={styles.buttonsContainer}>

<MainButton onPress={handleNextGuess.bind(this, "lower")}>

<Ionicons name="md-remove" size={24} color="white" />

</MainButton>

<MainButton onPress={handleNextGuess.bind(this, "greater")}>

<Ionicons name="md-add" size={24} color="white" />

</MainButton>

</Card>

**<View style={styles.listContainer}>**

**<ScrollView contentContainerStyle={styles.list}>**

**{guesses.map(renderListItem)}**

**</ScrollView>**

**</View>**

</View>

)

}

const styles = StyleSheet.create({

container: { flex: 1, padding: 10, alignItems: "center" },

buttonsContainer: {

flexDirection: "row", justifyContent: "space-evenly",

marginTop: 20, width: 400, maxWidth: "90%"

},

**listContainer: {**

**width: "80%",**

**flex: 1 // stretch to the whole available space**

**},**

**list: {**

**flexGrow: 1**, // keeps styles of ScrollView. Exceeds boundaries.

**alignItems: "center",**

**justifyContent: "flex-end"**

**},**

**listItem: {**

**borderColor: "#ccc", borderWidth: 1, padding: 15,**

**marginVertical: 10, backgroundColor: "white",**

**flexDirection: "row",**

**justifyContent: "space-evenly",**

**width: "60%"**

**}**

})

function rng(min, max, exclude) {

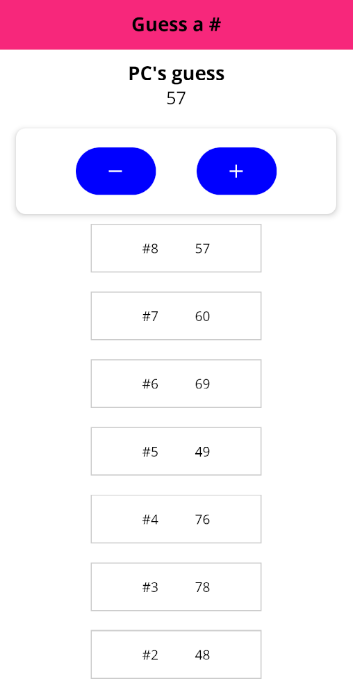
min = Math.ceil(min) // include 1

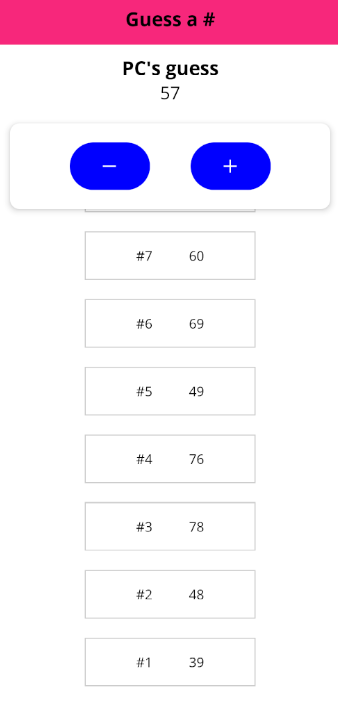
max = Math.floor(max) // exclude 100

const rndNum = Math.floor(Math.random() \* (max - min)) + min

return rndNum === exclude ? rng(min, max, exclude) : rndNum

}





**Using FlatList instead of ScrollView**

ScrollView always keeps its components rendered, making it an efficient component only when there are not too many items for it to show.

If we need to display plenty of items, we use a FlatList.

**GameScreen.js**

import React, { useState, useRef, useEffect } from "react"

import { View, Text, StyleSheet, Alert, **FlatList** } from "react-native"

…

export default function **GameScreen**({ userChoice, onGameOver }) {

const [curGuess, setCurGuess] = useState(rng(1, 100, userChoice))

const [guesses, setGuesses] = useState(**[{ guess: curGuess }]**)

const curLow = useRef(1)

const curHi = useRef(100)

useEffect(() => {

if (curGuess === userChoice) onGameOver(guesses.length)

}, [curGuess, userChoice, onGameOver])

const handleNextGuess = (direction) => {

if (

(direction === "lower" && curGuess < userChoice) ||

(direction === "greater" && curGuess > userChoice)

) {

return Alert.alert("No cheating", "You know that's not true.",

[{ text: "Sorry :c", style: "cancel" }])

}

if (direction === "lower") curHi.current = curGuess

else curLow.current = curGuess + 1 // as not to repeat keys

const newGuess = rng(curLow.current, curHi.current, curGuess)

setCurGuess(newGuess)

setGuesses((prevGuesses) => [**{ guess: newGuess }, ...prevGuesses]**)

}

const renderListItem = (**{ item, index }**) => (

<View style={styles.listItem}>

<NormalText>#{guesses.length - index}</NormalText>

<NormalText>{**item.guess**}</NormalText>

</View>

)

return (

<View style={styles.container}>

<Text style={defaultStyles.BOLD\_TEXT}>PC's guess</Text>

<Text style={defaultStyles.NORMAL\_TEXT}>{curGuess}</Text>

<Card style={styles.buttonsContainer}>

<MainButton onPress={handleNextGuess.bind(this, "lower")}>

<Ionicons name="md-remove" size={24} color="white" />

</MainButton>

<MainButton onPress={handleNextGuess.bind(this, "greater")}>

<Ionicons name="md-add" size={24} color="white" />

</MainButton>

</Card>

<View style={styles.listContainer}>

**<FlatList**

**contentContainerStyle={styles.list}**

**data={guesses}**

**keyExtractor={({ guess }) => guess.toString()}**

**renderItem={renderListItem}**

**/>**

</View>

</View>

)

}

const styles = StyleSheet.create({

container: { flex: 1, padding: 10, alignItems: "center" },

buttonsContainer: {

flexDirection: "row", justifyContent: "space-evenly",

marginTop: 20, width: 400, maxWidth: "90%"

},

listContainer: {

**width: "60%",** // FlatList container is the real width for items

flex: 1

},

list: {

flexGrow: 1,

// nothing to align to center, listItems are 100% width now

justifyContent: "flex-end"

},

listItem: {

borderColor: "#ccc", borderWidth: 1, padding: 15,

marginVertical: 10, backgroundColor: "white",

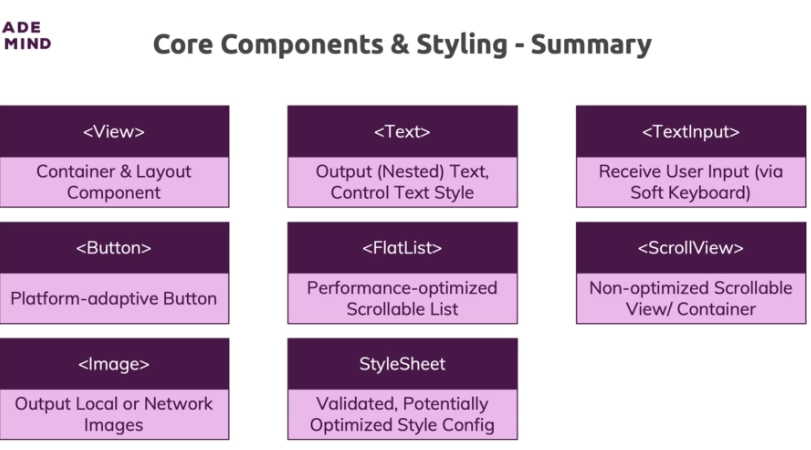
flexDirection: "row", justifyContent: "space-evenly",

**width: "100%"** // 100% of 60%, now in listContainer

}

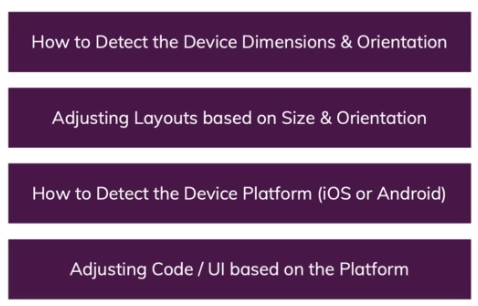
})

**Wrap up**



**Responsive and Adaptive interfaces and Apps**

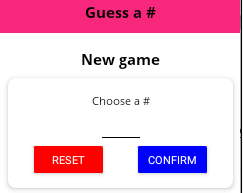
**Intro**



**Working with more flexible style rules**

First thing first, it is convenient to create some device simulator images with different sizes to be tested.

Now, you can set width as %, then set min and max widths to make sure the item is never too large or small, regardless of the device.

Like this:

**StartGameScreen.js**

…

const styles = StyleSheet.create({

…

card: {

**width: "80%", minWidth: 300, maxWidth: "95%"**, alignItems: "center" },

…

})

**Dimensions API**

Sometimes, we cannot get away with just using percentages. We might need to know how many pixels are available, and here is when Dimensions API comes in handy.

Mind that percentages calculate over parent containers, **while Dimensions API can be executed on window and on screen**.

**Dimensions.get()** returns an object with values for current **fontScale**, **scale**, **width** and **height.** Both for screen and window as arguments.



**StartGameScreen.js**

…

const styles = StyleSheet.create({

…

button: { width: **Dimensions.get("window").width / 3** },

…

})

We can also include if checks anywhere in the javascript file.

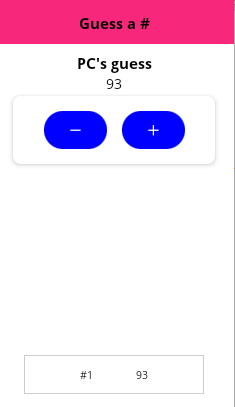
**GameScreen.js**

…

const styles = StyleSheet.create({

…

listContainer: {

 **width: Dimensions.get("window").width > 500 ? "60%" : "80%",**

flex: 1 // stretch to the whole available space

},

…

})

And of course, we can dynamically calculate values.

**GameOver.js**

…

export default function **GameOverScreen**({ guessRounds, userNum, onRestart }) {

return (

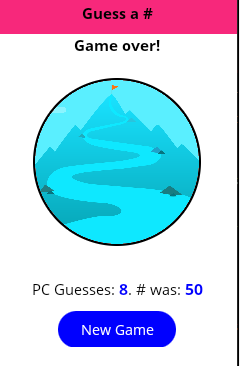
{/\* if device is extra small, make sure it is scrollable \*/}

**<ScrollView>**

<View style={styles.container}>

<BoldText>Game over!</BoldText>

<View style={styles.imageContainer}>

 <Image

source={require("../assets/images/success.png")}

style={styles.image}

resizeMode="cover"

/>

…

**</ScrollView>**

)

}

const styles = StyleSheet.create({

container: { flex: 1, justifyContent: "center", alignItems: "center" },

imageContainer: {

**width: Dimensions.get("window").width \* 0.7,**

**height: Dimensions.get("window").width \* 0.7,**

**borderRadius: (Dimensions.get("window").width \* 0.7) / 2,**

borderWidth: 3, borderColor: "black", overflow: "hidden",

**marginVertical: Dimensions.get("window").height / 20** // 5%

},

image: { width: "100%", height: "100%" },

textWrapper: {

**marginVertical: Dimensions.get("window").height / 40,** // 2.5%

textAlign: "center",

**fontSize: Dimensions.get("window").height < 400 ? 16 : 20**

},

highlight: {

color: colors.PRIMARY, fontFamily: "open-sans-bold",

marginHorizontal: 30

}

})

**Problems with device orientations**

Some apps have to be locked to a certain device orientation, being portrait or landscape. But most times we would want it to be used in both modes.

This is configurable in app.json

**app.json**

{

"expo": {

"name": "guess-a-number",

"slug": "guess-a-number",

"version": "1.0.0",

**"orientation": "default",**

"icon": "./assets/icon.png",

"splash": {

"image": "./assets/splash.png",

"resizeMode": "contain",

"backgroundColor": "#ffffff"

},

"updates": {

"fallbackToCacheTimeout": 0

},

"assetBundlePatterns": ["\*\*/\*"],

"ios": {

"supportsTablet": true

},

"android": {

"adaptiveIcon": {

"foregroundImage": "./assets/adaptive-icon.png",

"backgroundColor": "#FFFFFF"

}

},

"web": {

"favicon": "./assets/favicon.png"

}

}

}

**KeyboardAvoidingView**

This breaks all stylings, so lets start adjusting them, and we will start with KeyboardAvoidingView, a wrapper View component which makes sure the screen adjusts itself when keyboard opens.

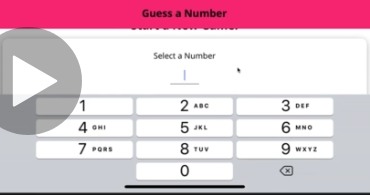
Note it must be used inside ScrollView if we are applying the latter.

**Behaviors** can be:

**‘position’**, which adjusts an offset of certain amount of pixels,

controlled by keyboardVerticalOffset

*<KeyboardAvoidingView behavior="position" keyboardVerticalOffset={30}>*



**‘padding’**, adding padding styling to the bottom of the screen.

*<KeyboardAvoidingView behavior="padding">*

**‘height’**, changing the overall height of the screen.

Typically, on iOS ‘position’ works best, and on android, ‘padding’.

**Listening to orientation changes**

**Dimensions.get() calculates once when app starts**.

This means that if the device rotates, values will not re-calculate, and styles will break.

To recalculate Dimensions, we have to pluck it off StyleSheet variable, and add checks in the component itself, with Dimensions.addEventListener. Like this:

**StartGameScreen.js**

…

export default function **StartGameScreen**({ onStartGame }) {

const [num, setNum] = useState("")

const [confirm, setConfirm] = useState(false)

const [selectedNum, setSelectedNum] = useState()

**const [buttonWidth, setButtonWidth] = useState(**

**Dimensions.get("window").width / 3.5**

**)**

**useEffect(() => {**

// create a function that returns the width of the window / 3.5

**const updateLayout = () =>**

**setButtonWidth(Dimensions.get("window").width / 3.5)**

// add a listener to call for it

**Dimensions.addEventListener("change", updateLayout)**

// clear the event listener on unmount

**return () => Dimensions.removeEventListener("change",**

**updateLayout)**

**}, [])**

…

return (

<ScrollView>

<KeyboardAvoidingView

behavior="position" keyboardVerticalOffset={30}>

<TouchableWithoutFeedback onPress={Keyboard.dismiss}>

<View style={styles.container}>

…

<View style={styles.buttonsContainer}>

<View **style={{ width: buttonWidth }}>**

<Button

title="Reset"

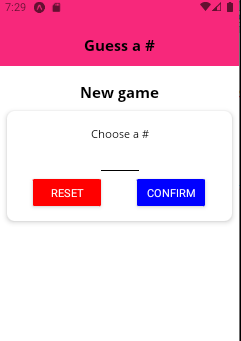
color={colors.DANGER}

onPress={handleResetInput}

/>

</View>

<View **style={{ width: buttonWidth }}>**

 <Button vertical

title="Confirm"

color={colors.PRIMARY}

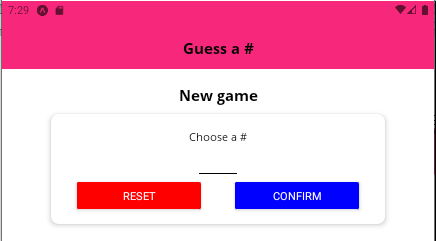
onPress={handleConfirm}

/>

</View>

</View>

…

 ) horizontal

}

**Rendering different layouts**

**GameScreen.js**

…

export default function **GameScreen**({ userChoice, onGameOver }) {

…

**const [currDeviceWidth, setCurrDeviceWidth] = useState(**

**Dimensions.get("window").width**

**)**

**const [currDeviceHeight, setCurrDeviceHeight] = useState(**

**Dimensions.get("window").height**

**)**

…

**useEffect(() => {**

**const updateLayout = () => {**

**setCurrDeviceHeight(Dimensions.get("window").height)**

**setCurrDeviceWidth(Dimensions.get("window").width)**

**}**

**Dimensions.addEventListener("change", updateLayout)**

**return () => Dimensions.removeEventListener("change",**

**updateLayout)**

**}, [])**

…

**if (currDeviceHeight < 500) {**

return (

<View style={styles.container}>

<Text style={defaultStyles.BOLD\_TEXT}>PC's guess</Text>

**<Card style={styles.buttonsContainer}>**

**<View style={styles.control}>**

**<MainButton onPress={handleNextGuess.bind(this, "lower")}>**

**<Ionicons name="md-remove" size={24} color="white" />**

**</MainButton>**

**<Card>**

**<Text style={defaultStyles.NORMAL\_TEXT}>{curGuess}</Text>**

**</Card>**

**<MainButton onPress={handleNextGuess.bind(this, "greater")}>**

**<Ionicons name="md-add" size={24} color="white" />**

**</MainButton>**

**</View>**

**</Card>**

<View style={styles.listContainer}>

<FlatList

contentContainerStyle={styles.list}

data={guesses}

keyExtractor={({ guess }) => guess.toString()}

renderItem={renderListItem}

/>

</View>

</View>

)

}

return (

<View style={styles.container}>

<Text style={defaultStyles.BOLD\_TEXT}>PC's guess</Text>

**<Card style={styles.guessWrapper}>**

**<Text style={defaultStyles.NORMAL\_TEXT}>{curGuess}</Text>**

**</Card>**

**<Card style={styles.buttonsContainer}>**

**<MainButton onPress={handleNextGuess.bind(this, "lower")}>**

**<Ionicons name="md-remove" size={24} color="white" />**

**</MainButton>**

**<MainButton onPress={handleNextGuess.bind(this, "greater")}>**

**<Ionicons name="md-add" size={24} color="white" />**

**</MainButton>**

**</Card>**

<View style={styles.listContainer}>

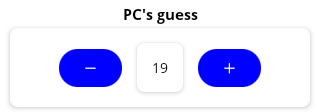
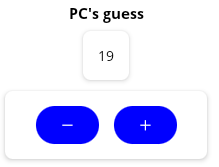
<FlatList

contentContainerStyle={styles.list}

data={guesses}

keyExtractor={({ guess }) => guess.toString()}

renderItem={renderListItem}

 />

</View>

</View>

)

}

const styles = StyleSheet.create({

container: { flex: 1, padding: 10, alignItems: "center" },

**control: {**

**flexDirection: "row", justifyContent: "space-around",**

**alignItems: "center", width: "80%"**

**},**

**guessWrapper: { marginVertical: 10 },**

…

})

**Expo’s ScreenOrientation API**

Built-in Dimensions API does not give you the device orientation, but expo’s ScreenOrientation does.

***$ expo install expo-screen-orientation***

This gives you the screen orientation and screen lock capabilities.

You have diverse functionalities, as **getOrientationAsync**, **lockAsync**, **addOrientationListener**, **lockPlatformAsync**, and many more. All async because they all for the underlying native API.

You also get some constants, like **OrientationLock.PORTRAIT** and **LANDSCAPE** strings.

A simple example to lock orientation to portrait mode would be:

*ScreenOrientation.lockAsync(*

*ScreenOrientation.OrientationLock.PORTRAIT*

*)*

**Platform API**

Some APIs and components behave differently in Android and in iOS, so inevitably, we need a way to detect on which platform are we running. Platform API does just that.

It offers back values as **OS**, **Version**, **constants**, **isTV**, **isTVOS**, **isPad**, **isTesting**; and a function to **select** a platform.

**Header.js**

import React from "react"

import { View, StyleSheet, **Platform** } from "react-native"

import BoldText from "./BoldText"

import colors from "../constants/colors"

export default function **Header**(props) {

return (

<View style={styles.container}>

<BoldText style={styles.title}>{props.title}</BoldText>

</View>

)

}

const styles = StyleSheet.create({

container: {

width: "100%", height: 90, paddingTop: 36,

alignItems: "center", justifyContent: "center",

**...Platform.select({**

**ios: {**

backgroundColor: "white",

borderBottomColor: "gray",

borderBottomWidth: 1

**},**

**android:** { backgroundColor: colors.SECONDARY }

})

},

**title**: { color: **Platform.OS === "ios"** ? colors.PRIMARY : "white" }

})

**MainButton.js**

import React from "react"

import { View, Text, StyleSheet, TouchableOpacity,

**TouchableNativeFeedback, Platform**

} from "react-native"

import colors from "../constants/colors"

export default function **MainButton**({

children, onPress, styles = {}, ...rest

}) {

**let ButtonComponent = TouchableOpacity**

// 'ripple' effect was introduced in android v.21

**if (Platform.OS === "android" && Platform.Version >= 21) {**

**ButtonComponent = TouchableNativeFeedback**

**}**

return (

**<View style={[\_styles.container, styles.container]}>**

**<ButtonComponent**

activeOpacity={0.4} style={styles.button}

onPress={onPress} {...rest}

>

<View style={[\_styles.view, styles.view]}>

<Text style={[\_styles.text, styles.text]}>{children}</Text>

</View>

**</ButtonComponent>**

**</View>**

)

}

const \_styles = StyleSheet.create({

// make 'ripple' effect do not exceed boundaries of visible button

**container: { borderRadius: 25, overflow: "hidden" },**

view: {

backgroundColor: colors.PRIMARY, paddingVertical: 12,

paddingHorizontal: 30, borderRadius: 25

},

text: { color: "white", fontFamily: "open-sans", fontSize: 18 }

})

Or, if the file ends up clogged with platform checks, **you can create two independent files width .ios.js and .android.js extensions**.

React Native will compile the standard file to the correct platform, under the hood.

**MainButton.android.js**

…

export default function **MainButton**({

children, onPress, styles = {}, ...rest

}) {

let ButtonComponent = TouchableOpacity

// 'ripple' effect was introduced in android v.21

**if (Platform.Version >= 21)** { // <- no more Platform.OS check

ButtonComponent = TouchableNativeFeedback

**}**

return (

<View style={[\_styles.container, styles.container]}>

<ButtonComponent

activeOpacity={0.4} style={styles.button}

onPress={onPress} {...rest}

>

<View style={[\_styles.view, styles.view]}>

<Text style={[\_styles.text, styles.text]}>{children}</Text>

</View>

</ButtonComponent>

</View>

)

}

const \_styles = StyleSheet.create({

// make 'ripple' effect do not exceed boundaries of visible button

container: { borderRadius: 25, overflow: "hidden" },

view: {

backgroundColor: colors.PRIMARY,

paddingVertical: 12,

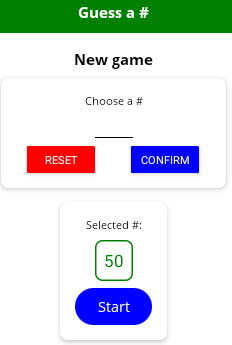
paddingHorizontal: 30,

borderRadius: 25

},

text: { color: "white", fontFamily: "open-sans", fontSize: 18 }

})

**MainButton.ios.js** android

…

export default function **MainButton**({

children, onPress, styles = {}, ...rest

}) {

return (

**<TouchableOpacity**

activeOpacity={0.4} style={styles.button}

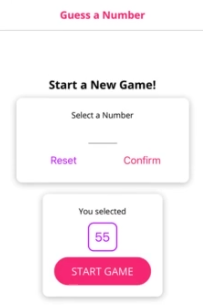
onPress={onPress} {...rest}

>

<View style={[\_styles.view, styles.view]}>

<Text style={[\_styles.text, styles.text]}>{children}</Text>

</View>

 **</TouchableOpacity>** ios

)

}

const \_styles = StyleSheet.create({

view: {

backgroundColor: colors.PRIMARY,

paddingVertical: 12,

paddingHorizontal: 30, borderRadius: 25

},

text: { color: "white", fontFamily: "open-sans", fontSize: 18 }

})

**SafeAreaView**

1.

2.

3.

4.

5.

6.

7.

8.

9.

10.

11.

12.

13.

14.

**A taste of React**

**Intro to React**

1.

2.

3.

4.

5.

6.

7.

8.

9.

10.

11.

12.

13.

14.